

11th Symposium on Languages, Applications and Technologies

SLATE 2022, July 14–15, 2022, Universidade da Beira Interior,
Covilhã, Portugal

Edited by

João Cordeiro

Maria João Pereira

Nuno F. Rodrigues

Sebastião Pais



Editors

João Cordeiro 

HULTIG, INESC TEC, UBI, Covilhã, Portugal
jpcc@ubi.pt

Maria João Pereira 

CeDRI, IPB, Bragança, Portugal
mjoao@ipb.pt

Nuno F. Rodrigues 

ALGORITMI, IPCA, Barcelos, Portugal
nfr@ipca.pt

Sebastião Pais 

HULTIG, Nova LINCS, GREYC, UBI, Covilhã, Portugal
sebastiao@di.ubi.pt

ACM Classification 2012

Theory of computation → Formal languages and automata theory; Computing methodologies → Natural language processing; Information systems → World Wide Web

ISBN 978-3-95977-245-7

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-245-7>.

Publication date

August, 2022

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASlcs.SLATE.2022.0

ISBN 978-3-95977-245-7

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

OASlcs – OpenAccess Series in Informatics

OASlcs is a series of high-quality conference proceedings across all fields in informatics. OASlcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

To all whose effort keeps SLATE alive.

■ Contents

Preface	
<i>João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais</i>	0:ix
List of Authors	
.....	0:xi
Committees	
.....	0:xiii

Papers

Assessing Similarity Between Two Ontologies: The Use of the Integrity Coefficient	
<i>Aly Ngoné Ngom, Papa Ousseynou Mbaye, and Ibrahima Gaye</i>	1:1–1:11
Automatic Classification of Portuguese Proverbs	
<i>Jorge Baptista and Sónia Reis</i>	2:1–2:8
Question Answering For Toxicological Information Extraction	
<i>Bruno Carlos Luís Ferreira, Hugo Gonçalo Oliveira, Hugo Amaro, Ângela Laranjeiro, and Catarina Silva</i>	3:1–3:10
Generation of Document Type Exercises for Automated Assessment	
<i>José Paulo Leal, Ricardo Queirós, and Marco Primo</i>	4:1–4:6
Synthetic Data Generation from JSON Schemas	
<i>Hugo André Coelho Cardoso and José Carlos Ramalho</i>	5:1–5:16
Extending PyJL – Transpiling Python Libraries to Julia	
<i>Miguel Marcelino and António Menezes Leitão</i>	6:1–6:14
EWVM, a Web Virtual Machine to Support Code Generation in Compiler Courses	
<i>Sofia Teixeira, José Carlos Ramalho, and Pedro Rangel Henriques</i>	7:1–7:9
OMT, a Web-Based Tool for Ontology Matching	
<i>João Rodrigues Gomes, Alda Lopes Gançarski, and Pedro Rangel Henriques</i>	8:1–8:12
Classification of Public Administration Complaints	
<i>Francisco Caldeira, Luís Nunes, and Ricardo Ribeiro</i>	9:1–9:12
Comparing Different Approaches for Detecting Hate Speech in Online Portuguese Comments	
<i>Bernardo Cunha Matos, Raquel Bento Santos, Paula Carvalho, Ricardo Ribeiro, and Fernando Batista</i>	10:1–10:12
Semi-Supervised Annotation of Portuguese Hate Speech Across Social Media Domains	
<i>Raquel Bento Santos, Bernardo Cunha Matos, Paula Carvalho, Fernando Batista, and Ricardo Ribeiro</i>	11:1–11:14
Large Semantic Graph Summarization Using Namespaces	
<i>Ana Rita Santos Lopes da Costa, André Santos, and José Paulo Leal</i>	12:1–12:9

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Metaobject Protocols for Julia <i>Marcelo Santos and António Menezes Leitão</i>	13:1–13:15
The Visual Programming Environment ROBI for Educational Robotics <i>Gustavo Galvão, Alvaro Costa Neto, Cristiana Araújo, and Pedro Rangel Henriques</i>	14:1–14:15
Down-Translating XML: The Python Way <i>Alberto Simões and José João Almeida</i>	15:1–15:9
Determining Programming Languages Complexity and Its Impact on Processing <i>Gonçalo Rodrigues Pinto, Pedro Rangel Henriques, Daniela da Cruz, and João Cruz</i>	16:1–16:15
Reasoning with Portuguese Word Embeddings <i>Luís Filipe Cunha, J. João Almeida, and Alberto Simões</i>	17:1–17:14
ScraPE – An Automated Tool for Programming Exercises Scraping <i>Ricardo Queirós</i>	18:1–18:7
Analysing Off-The-Shelf Options for Question Answering with Portuguese FAQs <i>Hugo Gonçalo Oliveira, Sara Inácio, and Catarina Silva</i>	19:1–19:11

■ Preface

Language allows humans to interact versatile and complexly, consolidating bonds that underlie solid social structures. With language, we think collectively and act more effectively. Also, it is through human language that precious knowledge is stored and disseminated throughout communities and along with history. It firmly bases our collective identity, our culture. Some anthropologist suggests that human communication patterns appeared around 50 000 years ago, but this origin is lost to the ages. One thing is sure: language equipped humans with an advantageous framework that allowed them to reach this complex and technological era we live in. Thus, at the origin, we have *Human to Human Languages (HHL)*.

With the development of the computer, from the middle of the 20th century, it became necessary to think of languages that would allow humans to communicate with machines at a level of abstraction above the underlying electronic processes and, therefore, more efficiently and conveniently to the human being. These are the kind of *Human to Computer Languages (HCL)*, including the wide range of programming languages that have been developed and are still being researched.

More recently, with the advent of networks, we found a way to make computers communicate between themselves. Therefore, languages that allow machines to exchange information efficiently and effectively are also being investigated. *These are the kind of Computer-to-Computer Languages (CCL)*, e.g., representation and transformation languages like XML.

These different forms of communication use different languages, but they still share many similarities and joint issues and challenges. In SLATE, we are interested in discussing these three kinds of languages and eventually spark some interesting synergies that might emerge from them. Therefore, SLATE is organized into three different tracks. It is a forum where researchers, developers, and educators exchange ideas and information involving natural language processing (**HHL Track**), the latest academic or industrial work on language design, processing, assessment, and applications (**HCL Track**), and markup and transformation languages (**CCL Track**).

This book of proceedings compiles the accepted papers for the Eleventh edition of the Symposium on Languages, Applications and Technologies, SLATE'2022, held at the University of Beira Interior, Portugal, on the 14th and 15th of July, 2022. In this edition, a total twenty two papers were submitted by a total of forty one authors. The Scientific Committee executed a thorough revision of each paper and nineteen papers were selected to publication and oral presentation in Symposium. We end up with five papers in the HCL track, seven papers in the HHL track, and seven papers in the CCL track.

Our gratitude goes to the many people without whom this event would never have been possible: all the Members of the Scientific Program Committee for their valuable effort reviewing the submissions, contributing with corrections and new ideas, and helping on deciding the final list of accepted papers; the members of the Organizing Committee, for the help on dealing with the bureaucratic issues; the invited Speakers (Ricardo Campos and Simão Melo de Sousa) for accepting our invitation to share their knowledge; and finally, we thank all the Authors for their contributions to SLATE with their current projects and research problems.

João Cordeiro
Main Chair



■ List of Authors

Alberto Simões
2Ai, School of Technology, IPCA
Barcelos, Portugal
asimoes@ipca.pt

Alda Lopes Gançarski
Centro Algoritmi & University of Minho
Braga, Portugal
aldalopesgancarski@gmail.com

Alvaro Costa Neto
Instituto Federal de Educação, Ciência e
Tecnologia de São Paulo
Barretos, Brazil
alvaro@ifsp.edu.br

Aly Ngoné NGOM
Gaston Berger University
Saint – Louis, Sénégal
ngom.aly-ngone@ugb.edu.sn

Ana Rita Santos Lopes da Costa
University of Porto
Porto, Portugal
up201605706@edu.fc.up.pt

André Santos
INESC TEC & University of Porto
Porto, Portugal
afs@inesctec.pt

António Leitão
INESC-ID & IST – University of Lisbon
Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

Ângela Laranjeiro
Cosmedesk
Coimbra, Portugal
angela@cosmedesk.com

Bernardo Cunha Matos
INESC-ID & IST – University of Lisbon
Lisbon, Portugal
bernardo.cunha.matos@tecnico.ulisboa.pt

Bruno Carlos Luís Ferreira
University of Coimbra
Coimbra, Portugal
brunof@student.dei.uc.pt

Catarina Silva
University of Coimbra
Coimbra, Portugal
catarina@dei.uc.pt

Cristiana Araújo
Centro Algoritmi & University of Minho
Braga, Portugal
decrisianaaraujo@hotmail.com

Daniela da Cruz
Checkmarx
Braga, Portugal
daniela.dacruz@checkmarx.com

Fernando Batista
INESC-ID & ISCTE
University Institute of Lisbon
Lisbon, Portugal
fernando.batista@inesc-id.pt

Francisco Caldeira
ISCTE – University Institute of Lisbon
Lisbon, Portugal
fmsca1@iscte-iul.pt

Gonçalo Rodrigues Pinto
University of Minho
Braga, Portugal
a83732@alunos.uminho.pt

Gustavo Galvão
Centro Algoritmi & University of Minho
Braga, Portugal
gustavolinhares1995@gmail.com

Hugo Amaro
Instituto Pedro Nunes, LIS
Coimbra, Portugal
hamaro@ipn.pt

Hugo Cardoso
University of Minho
Braga, Portugal
a85006@alunos.uminho.pt

Hugo Gonçalo Oliveira
University of Coimbra
Coimbra, Portugal
hroliv@dei.uc.pt

11th Symposium on Languages, Applications and Technologies (SLATE 2022).
Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ibrahima Gaye
Alioune Diop University
Bambey, Sénégal
ibrahima2.gaye@uadb.edu.sn

Jorge Baptista
INESC-ID & University of Algarve
Faro, Portugal
jbaptis@ualg.pt

José João Almeida
University of Minho
Braga, Portugal
jj@di.uminho.pt

José Paulo Leal
INESC TEC & University of Porto
Porto, Portugal
zp@dcc.fc.up.pt

José Ramalho
Centro Algoritmi & University of Minho
Braga, Portugal
jcr@di.uminho.pt

João Cruz
Checkmarx
Braga, Portugal
joao.cruz@checkmarx.com

João Rodrigues Gomes
Centro Algoritmi & University of Minho
Braga, Portugal
a82238@alunos.uminho.pt

Luis Filipe Cunha
University of Minho
Braga, Portugal
filipe-cunha1@hotmail.com

Luis Nunes
ISCTE – University Institute of Lisbon
Lisbon, Portugal
luis.nunes@iscte-iul.pt

Marcelo Santos
IST – University of Lisbon
Lisbon, Portugal
marcelocmsantos@tecnico.ulisboa.pt

Marco Primo
University of Porto
Porto, Portugal
up201800388@edu.fc.up.pt

Miguel Marcelino
INESC-ID & IST – University of Lisbon
Lisbon, Portugal
miguel.marcelino@tecnico.ulisboa.pt

Papa Ousseynou Mbaye
SET, Sup de Co of Dakar
Dakar, Sénégal
papaousseynoumbaye@gmail.com

Paula Carvalho
INESC-ID
Lisbon, Portugal
pcc@inesc-id.pt

Pedro Rangel Henriques
Centro Algoritmi & University of Minho
Braga, Portugal
prh@di.uminho.pt

Raquel Bento Santos
INESC-ID & IST – University of Lisbon
Lisbon, Portugal
raquel.bento.santos@tecnico.ulisboa.pt

Ricardo Queirós
INESC TEC & uniMAD, ESMAD
Porto, Portugal
ricardoqueiros@esmad.ipp.pt

Ricardo Ribeiro
INESC-ID & ISCTE
University Institute of Lisbon
Lisbon, Portugal
ricardo.ribeiro@inesc-id.pt

Sara Inácio
University of Coimbra
Coimbra, Portugal
sara.inacio.1997@gmail.com

Sofia Teixeira
Centro Algoritmi & University of Minho
Braga, Portugal
so.texera@gmail.com

Sónia Reis
INESC-ID Lisboa & Universidade do Algarve
Faro, Portugal
smreis@ualg.pt

■ Committees

Conference Chairs

Main Program Chair

João Cordeiro
University of Beira Interior, PT

António Miguel Cruz
Polytechnic Institute of Viana do Castelo,
PT

António Teixeira
University of Aveiro, PT

Track Chairs

Human-Human Languages:

Sebastião Pais
University of Beira Interior, PT

Arkaitz Zubiaga
Queen Mary University of London, UK

Bostjan Slivnik
University of Ljubljana, SI

Human-Computer Languages:

Maria João Pereira
Polytechnic Institute of Bragança, PT

Brett Drury
Liverpool Hope University, UK

Cristina Ribeiro
University of Porto, PT

Computer-Computer Languages:

Nuno F. Rodrigues
Polytechnic Institute of Cávado and Ave, PT

David Aveiro
University of Madeira, PT

David Martins de Matos
University of Lisbon, PT

Organization Committee

Alberto Simões
Polytechnic Institute of Cávado and Ave, PT

Diana Santos
University of Oslo, NO

Dietmar Seipel
University of Würzburg, DE

João Cordeiro
University of Beira Interior, PT

Dušan Kolář
Brno University of Technology, CZ

Pedro Rangel Henriques
University of Minho, PT

Fernando Batista
ISCTE-IUL, PT

Sebastião Pais
University of Beira Interior, PT

Gergő Balogh
University of Szeged, HU

Scientific Committee

Alberto Simões
Polytechnic Institute of Cávado and Ave, PT

Giovani Librelotto
Universidade Federal de Santa Maria, BR

Hugo Gonçalo Oliveira
University of Coimbra, PT

Alexander Paar
Duale Hochschule Schleswig-Holstein, DE

Hugo Peixoto
University of Minho, PT

Antoni Oliver
Universitat Oberta de Catalunya, ES

Irene Rodrigues
University of Evora, PT

Antonio Leitao
University of Lisbon, PT

Ivan Luković
University of Belgrade, RS

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais



OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- Jakub Swacha
University of Szczecin, PL
- Jan Janousek
Czech Technical University Prague, CZ
- Jaroslav Porubän
Technical University of Košice, SK
- Joao Cardoso
University of Porto, PT
- Jose Luis Sierra
Universidad Complutense de Madrid, ES
- Josep Silva
Universitat Politècnica de València, ES
- José Carlos Paiva
University of Porto, PT
- José Carlos Ramalho
University of Minho, PT
- José Paulo Leal
University of Porto, PT
- João Cordeiro
University of Beira Interior, PT
- João M. Lourenço
NOVA University Lisbon, PT
- João Saraiva
University of Minho, PT
- Luis Morgado Da Costa
Nanyang Technological University, SG
- Luís Ferreira
Polytechnic Institute of Cávado and Ave, PT
- Maria João Pereira
Polytechnic Institute of Bragança, PT
- Mario Berón
National University of San Luis, AR
- Mario Pinto
Polytechnic Institute of Porto, PT
- Nuno Mamede
University of Lisbon, PT
- Nuno Rodrigues
Polytechnic Institute of Cávado and Ave, PT
- Pablo Gamallo
University of Santiago de Compostela, ES
- Paulo Matos
Polytechnic Institute of Bragança, PT
- Pedro Rangel Henriques
University of Minho, PT
- Ricardo Queirós
Polytechnic Institute of Porto, PT
- Ricardo Rocha
University of Porto, PT
- Ricardo Rodrigues
University of Coimbra, PT
- Salvador Abreu
University of Evora, PT
- Sebastião Pais
University of Beira Interior, PT
- Simão Melo-de-Sousa
Universidade da Beira Interior, PT
- Teresa Guarda
Universidade Estatal da Península de Santa Elena, EC
- Tomaz Kosar
University of Maribor, SI
- Vasco Amaral
NOVA University Lisbon, PT
- Vladimir Ivančević
University of Novi Sad, RS

Assessing Similarity Between Two Ontologies: The Use of the Integrity Coefficient

Aly Ngoné Ngom ✉

LANI, Gaston Berger University, Saint-Louis, Sénégal
SET, Sup de Co of Dakar, Sénégal

Papa Ousseynou Mbaye ✉

SET, Sup de Co of Dakar, Sénégal

Ibrahima Gaye ✉

Alioune Diop University of Bambey, Sénégal

Abstract

The aim of this paper is to propose a new coefficient of integrity I_{new} for improving N_{Plus} measure which is an improvement of the T_{Ngom} measure. In N_{Plus} measure, the coefficient of integrity used (I) decreases and tends to 0 fastly when we just add some concepts for extending set of resemblance of ontologies. To fix this problem, we introduce R , the coefficient of representativeness of concepts added in the ontology for its extension. I_{new} decreases slowly compared to I and depends to the cardinality of the ontology extended and the number of concepts added to it too.

2012 ACM Subject Classification Theory of computation → Operational semantics; Information systems → Data structures

Keywords and phrases Semantic similarity measures, Ontologies similarities, Tversky's measures, Concepts similarities

Digital Object Identifier 10.4230/OASICS.SLATE.2022.1

Acknowledgements We want to thank Sup De Co of Dakar which supports all fund of the registration.

1 Introduction

Ontologies allow to formalize knowledge related to the description of the world by making them accessible and shareable across the Web. They introduce the semantic layer into the architecture of the on based-systems [1]. When several ontologies are used for an application, it is necessary that these ontologies present some similarity. The assessing of similarity between ontologies may be very interesting. Indeed, it can make easy the choice of ontologies in the case of elaboration of a system, which uses them. In addition, it can help to evaluate the ontology evolution by comparing its different versions. In [6], we have proposed an approach for assessing similarity between two ontologies O_1 and O_2 . This approach has given good results but doesn't take into account some properties (relations, axioms) for extending the formed sets (set of resemblance and set of differences) and improving the proposed measure. In [5], a new measure N_{Plus} is proposed in the goal to improve the T_{Ngom} measure proposed in [6]. N_{Plus} measure improves T_{Ngom} in the fact of it takes into account "is-a" relation to enlarge the formed set of resemblance by extending ontologies. In the formula of the N_{Plus} measure, a coefficient of integrity I is defined. The integrity coefficient of an ontology O_j named I_i is a value which is related to the number of concepts of O_j (n_j) that we have to add to another ontology O_i in goal to extend it ($i, j \in \mathbb{N}$). The larger is n_j , the smaller is I_i . The major problem of I is that it decreases fastly and tends fastly to 0 when we just add some concepts. To fix this problem, we propose in this paper a new coefficient of integrity I_{new} which will decrease slowly in related to the added concepts for extending ontologies and the size (cardinality) of ontologies.



© Aly Ngoné Ngom, Papa Ousseynou Mbaye, and Ibrahima Gaye;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 1; pp. 1:1–1:11

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This paper continues by presenting our research context. Section 2 presents the related work. Then, we present the new coefficient of integrity I_{new} in section 3 before making its experimentation in section 4. We end with conclusion and perspectives of this work.

2 Related Works

There are several works, which are dedicated to the evaluation of similarity between two concepts in an ontology. However, there are not many works which deal with evaluation of similarity between ontologies. The following are some works about similarity between two ontologies.

Maedche and Staab [3] propose a method for comparing two ontologies. This method is based on two levels:

- the **Lexical level** which consists of investigation on how terms are used to convey meanings ;
- the **Conceptual level** which is the investigation of what conceptual relations exist between terms.

The Lexical comparison allows to find concepts by assessing syntactic similarity between concepts. It is based on Levenshtein [2] edit distance (ed) formula which allows to measure the minimum number of change required to transform one string into another, by using a dynamic programming algorithm. The Conceptual Comparison Level allows to compare the semantic of structures of two ontologies. Authors use Upwards Cotopy (UC) to compare the Concept Match (CM). Then, they use the CM to determine the Relation Overlap (RO). Finally they assess the average of RO.

This approach allows to assess similarity between two ontologies by using the Lexical and Conceptual Comparison Level. However, if we reverse the position of some concepts in the hierarchy, we can get the same results because the method only considers the presence of the concept in the hierarchy.

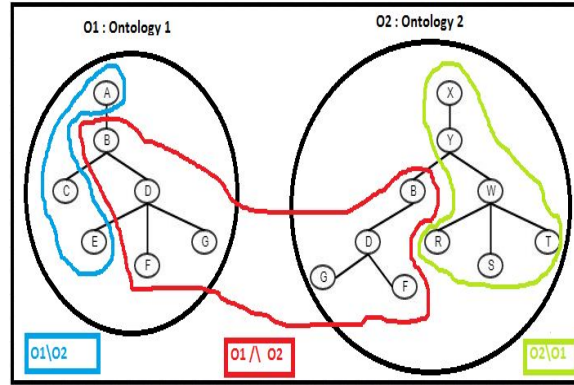
In [6], we propose an approach which assesses similarity between two ontologies. The approach is based on set theory, edges based [4] and feature-based similarity based [7]. It is composed of three steps. The first step consists to determine the different sets : set of concepts shared by the two ontologies and sets of concepts specific to each ontology. In the second step, we have assessed the average similarity values between concepts for each sets thanks to semantic similarity measures. Finally, in the last step, we have assessed similarity between ontologies by redefining Tversky's measure relying to the two first steps. Figure 1 represents ontologies O_1 and O_2 .

In figure 1, we distinguish three parts :

- $(O_1 \setminus O_2) = \{A, C, E\}$: set of concepts present in O_1 and not in O_2 ;
- $(O_2 \setminus O_1) = \{R, S, T, W, X, Y\}$: set of concepts present in O_2 and not in O_1 ;
- $(O_1 \wedge O_2) = \{B, D, F, G\}$: set of concepts present in O_1 and O_2 .

The three steps can be summarized in below :

- The **Step 1** consists to determine the sets $(O_1 \setminus O_2)$, $(O_2 \setminus O_1)$ and $(O_1 \wedge O_2)$.
- Once the sets are determined, we assess the average of the semantic similarity values between concepts of each set in the **step 2**.
- Finally, in the **step 3**, we assess similarity between ontologies by using the results of the step 2 in our measure which is a redefinition of the Tversky measure.



■ **Figure 1** Representation of ontologies O_1 and O_2 with Tversky's feature model.

To assess similarity between two ontologies, we have defined a measure which readjust the Tversky measure. This measure takes into account shared features and differences of ontologies. Applying the Tversky measure, the similarity between O_1 and O_2 is given by the formula 1.

$$Tvr_{(O_1, O_2)} = \frac{f(O_1 \cap O_2)}{f(O_1 \cap O_2) + \alpha \cdot f(O_1 \setminus O_2) + \beta \cdot f(O_2 \setminus O_1)} \quad (1)$$

Instead of the function f , we use Wu and Palmer [8] semantic similarity measure. for every determined set, we have computed the average of the similarity values between concepts. Using Wu and Palmer similarity measure, the similarity between two concepts c_1 and c_2 is given by the formula 2.

$$Sim(c_1, c_2) = \frac{2 \times depth(c_3)}{depth(c_1) + depth(c_2)} \quad (2)$$

The concept c_3 represents the Least Common Subsumer (LCS) of concepts c_1 and c_2 . By replacing the terms of the Tversky measure with the average of the similarity values between concepts of the determined sets, formula 1 becomes formula 3.

$$T_{Ngom}(O_1, O_2) = \frac{\theta \cdot \bar{x}_{O_1} + \omega \cdot \bar{x}_{O_2}}{\theta \cdot \bar{x}_{O_1} + \omega \cdot \bar{x}_{O_2} + \alpha \cdot \bar{y}_{(O_1 \setminus O_2)} + \beta \cdot \bar{z}_{(O_2 \setminus O_1)}} \quad (3)$$

with :

- $\theta = \frac{cardinality(O_1 \cap O_2)}{cardinality(O_1)}$;
- $\omega = \frac{cardinality(O_1 \cap O_2)}{cardinality(O_2)}$;
- $\alpha = \frac{cardinality(O_1 \setminus O_2)}{cardinality(O_1)}$;
- and $\beta = \frac{cardinality(O_2 \setminus O_1)}{cardinality(O_2)}$;
- $cardinality(O)$ is the number of elements (concepts) of the set (ontology) O ;

and where :

- \bar{x}_{O_1} (respectively \bar{x}_{O_2}) is the average value of similarity between concepts (x_i, x_j) in ontology O_1 (respectively (x_i, x_j) in ontology O_2). $i, j \in \mathbb{N}$ and $i \neq j$.
- $\bar{y}_{(O_1 \setminus O_2)}$ (respectively $\bar{z}_{(O_2 \setminus O_1)}$) is the average value of similarity between concepts (y_i, y_j) (respectively (z_i, z_j)) present in ontology O_1 but not in O_2 (respectively present in ontology O_2 but not in O_1). $i, j \in \mathbb{N}$ and $i \neq j$.
- the coefficients θ , ω , α and β allow to take into account the similarity values in relation to the number of concepts of the sets and number of concepts of ontologies.

1:4 Similarity Between Two Ontologies

The measure presented by formula 3 respects this properties :

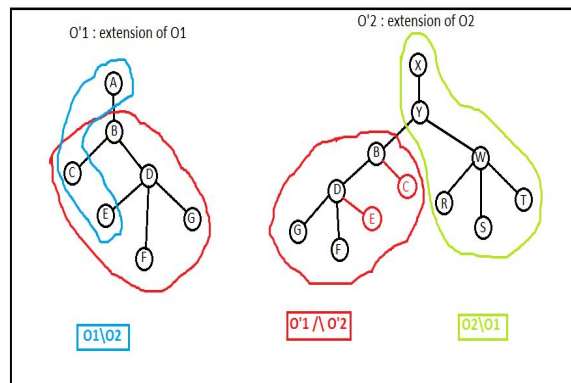
- the measure is symmetric : $T_{Ngom}(O_1, O_2) = T_{Ngom}(O_2, O_1)$;
- the measure is bounded between 0 and 1 ;
- if $T_{Ngom}(O_1, O_2) = 1$ then $O_1 = O_2$.

The method we have proposed in [6] gives satisfactory results. Indeed, it allows to assess similarity between two ontologies while taking into account the semantic links that exist between the concepts in ontologies. However, it doesn't take into account properties of concepts for extending the formed sets (set of resemblance and set of difference).

For taking into account properties of concepts for extending the formed sets (set of resemblance and set of difference), we have improved the T_{Ngom} measure [6] by proposing in [5] an approach which assesses similarity between ontologies by extending set of resemblance thanks to the is-a relation of ontologies. The improved approach can be summarized in five steps :

- **Step 1** consists to determine the sets $(O_1 \setminus O_2)$, $(O_2 \setminus O_1)$ and $(O_1 \wedge O_2)$.
- Once the sets are determined, we assess the average of the semantic similarity values between concepts of each set in **step 2**.
- In **step 3**, we extend ontologies O_1 and O_2 by using the set $(O_1 \wedge O_2)$. In this step, for each concept c of $(O_1 \wedge O_2)$, we search there sons x_i ($i \in \mathbb{N}$) in O_1 (respitively O_2) and we add them as sons of c in O_2 (respitively O_1) if they don't exist in this ontology. At the end of this step, we obtain two ontologies : O'_1 (respitively O'_2) which extends O_1 (respitively O_2) with concepts of O_2 (respitively O_1). Thus, extension of ontologies allows us to determine the set of concepts $(O'_1 \wedge O'_2)$ shared by the two ontologies.
- In **Step 4**, we determine $(O'_1 \wedge O'_2)$ which is the set of shared concepts by ontologies O'_1 and O'_2 .
- Finally, in the **step 5**, we assess similarity between ontologies by using the results of the step 2 and 4 in our measure which is a redefinition of the T_{Ngom} measure [6].

In summary, for assessing similarity between ontologies, we use sets $(O_1 \setminus O_2)$, $(O_2 \setminus O_1)$ and $(O'_1 \wedge O'_2)$; i.e we consider the difference between O_1 and O_2 by using sets $(O_1 \setminus O_2)$ and $(O_2 \setminus O_1)$, and the resemblance between the two ontologies by using set $(O'_1 \wedge O'_2)$. Figure 2 represents the differents that we use for assessing similarity between ontologies O_1 and O_2 .



■ **Figure 2** Representation of extensions of ontologies O'_1 and O'_2 with Tversky's feature model.

In Figure 2, we distinguish three parts :

- $(O_1 \setminus O_2) = \{A, C, E\}$: set of concepts present in O_1 and not in O_2 ;
- $(O_2 \setminus O_1) = \{R, S, T, W, X, Y\}$: set of concepts present in O_2 and not in O_1 ;
- $(O'_1 \wedge O'_2) = \{B, C, D, E, F, G\}$: set of concepts present in O'_1 and O'_2 .

The measure is given by the formula 4 :

$$N_{Plus}(O_1, O_2) = \frac{(\theta \cdot \bar{x}_{O'_1} + I_2) + (\omega \cdot \bar{x}_{O'_2} + I_1)}{(\theta \cdot \bar{x}_{O'_1} + I_2) + (\omega \cdot \bar{x}_{O'_2} + I_1) + \alpha \cdot \bar{y}_{(O_1 \setminus O_2)} + \beta \cdot \bar{z}_{(O_2 \setminus O_1)}} \quad (4)$$

with :

- $\theta = \frac{\text{cardinality}(O'_1 \cap O'_2)}{\text{cardinality}(O_1) + n_1 + n_2}$;
- $\omega = \frac{\text{cardinality}(O'_1 \cap O'_2)}{\text{cardinality}(O_2) + n_1 + n_2}$;
- $\alpha = \frac{\text{cardinality}(O_1 \setminus O_2)}{\text{cardinality}(O_1)}$;
- $\beta = \frac{\text{cardinality}(O_2 \setminus O_1)}{\text{cardinality}(O_2)}$;
- $I_1 = \frac{1}{1 + n_2}$;
- $I_2 = \frac{1}{1 + n_1}$;
- $\bar{x}_{O'_1}$ (respectively $\bar{x}_{O'_2}$) is the average value of similarity between concepts (x_i, x_j) in ontology O'_1 (respectively (x_i, x_j) in ontology O'_2). $i, j \in \mathbb{N}$ and $i \neq j$.
- $\bar{y}_{(O_1 \setminus O_2)}$ (respectively $\bar{z}_{(O_2 \setminus O_1)}$) is the average value of similarity between concepts (y_i, y_j) (respectively (z_i, z_j)) present in ontology O_1 but not in O_2 (respectively present in ontology O_2 but not in O_1). $i, j \in \mathbb{N}$ and $i \neq j$.
- $\text{cardinality}(O)$ is the number of elements (concepts) of the set (ontology) O ;
- I_i : Integrity coefficient of Ontology O_i ($i \in \mathbb{N}$);
- n_i : number of concepts of O_i added for extending O_j ($i, j \in \mathbb{N}$);
- As in [6], θ , ω , α and β are parameters which allow to take into account the similarity values in relation to the number of concepts of the sets and number of concepts of ontologies.

The integrity coefficient of Ontology (I_i) is a value which is related to the number of concepts of ontology O_j (n_j) that we have to add to O_i in goal to extend it ($i, j \in \mathbb{N}$). The larger is n_j , the smaller is I_i . We have the expression 5:

$$\begin{cases} \lim_{n \rightarrow \infty} I = \lim_{n \rightarrow \infty} \frac{1}{1 + n} = 0; \\ \lim_{n \rightarrow 0} I = \lim_{n \rightarrow 0} \frac{1}{1 + n} = 1; \end{cases} \quad (5)$$

with ($n \in \mathbb{N}$).

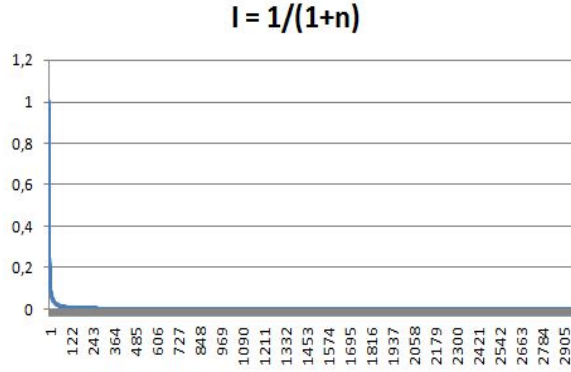
We note that measure presented by formula 4 like formula 3 respects this properties :

- the measure is symmetric : $N_{Plus}(O_1, O_2) = N_{Plus}(O_2, O_1)$;
- the measure is bounded between 0 and 1 ;
- if $N_{Plus}(O_1, O_2) = 1$ then $O_1 = O_2$.

The N_{Plus} measure presented in [5] improves T_{Ngom} measure of [6] because it takes into account is-a relation of ontologies to extend set of resemblance of ontologies by adding concepts. The limits of this measure is in the integrity coefficient of ontologies (I). I decreases fastly if we add just a few of concepts. Figure 3 shows the evolution of I_i when we add concepts for extending set of resemblance.

In Figure 3, we remark that I decreases very fastly. When we add concepts in an ontology for extending the resemblance set, we note that variation of $I = \frac{1}{1 + n}$:

- $I = 1$ if $n = 0$;
- $I = 0,5$ if $n = 1$;
- $I = 0,33$ if $n = 2$;
- ...
- $I = 0,1$ if $n = 9$.



■ **Figure 3** Variation of the integrity coefficient of ontologies I .

The integrity coefficient is equal to 0,5 when we just add one concept. It means that with one concept added, we lose 50% of the integrity of the ontology. We need to find a good coefficient formula for improving N_{Plus} .

3 New definition of integrity coefficient of ontology (I_{new})

In section 2, we presented N_{Plus} measure of [5] which improves T_{Ngom} measure of [6] by extending set of resemblance of ontologies. In definition of N_{Plus} formula, we introduced I which represents the integrity coefficient. The integrity coefficient (I) is a value introduced to express how an ontology loses its integrity when concepts are added in the goal to extend set of resemblance shared with another ontology. This value is between 0 and 1. Initially, when there is no concept added to the ontology, the value of I is 1. This value decreases as we add concept to the ontology. Figure 3 shows that I decreases fastly to 0 when we add just a lot of concepts. To fix this problem, we propose to redefine I . We introduce R for expressing coefficient of representativeness of concepts added in the ontology for its extension. R is expressed by the formula 6 :

$$R = \frac{n}{cardinality(O) + n} \tag{6}$$

with :

- n is the number of concepts added to the ontolgy O for his extension ($n \in \mathbb{N}$) ;
- $cardinality(O)$ represents the number of concepts of the ontology O ;

We redefine I with I_{new} in terms of R according to the formula 6 and we obtain the formula 7:

$$I_{new} = 1 - R$$

$$I_{new} = 1 - \frac{n}{cardinality(O) + n} \tag{7}$$

$$I_{new} = \frac{cardinality(O)}{cardinality(O) + n}$$

The new coefficient of integrity I_{new} represented by formula 7 is between 0 and 1 as shown in below :

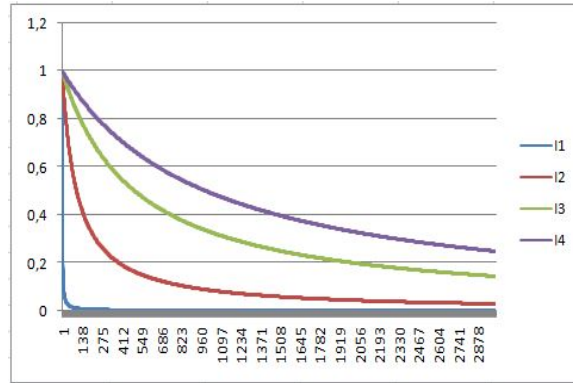
$$\begin{cases} \lim_{n \rightarrow \infty} I_{new} &= \lim_{n \rightarrow \infty} \frac{cardinality(O)}{cardinality(O) + n} &= 0; \\ \lim_{n \rightarrow 0} I_{new} &= \lim_{n \rightarrow 0} \frac{cardinality(O)}{cardinality(O) + n} &= 1; \end{cases} \tag{8}$$

with ($n \in \mathbb{N}$).

For studying the evolution of I_{new} , we use 4 functions I_1 , I_2 , I_3 and I_4 , and we draw there evolutions curves. Functions are defined in below :

- $I_1 = \frac{1}{1+n}$: The same formula of I in [5];
- $I_2 = \frac{100}{100+n}$: $cardinality(O) = 100$
- $I_3 = \frac{500}{500+n}$: $cardinality(O) = 500$
- $I_4 = \frac{1000}{1000+n}$: $cardinality(O) = 1000$

Figure 4 represents the evolutions of differents curves of functions I_1 , I_2 , I_3 and I_4 .



■ **Figure 4** Variations of the integrities coefficients of ontologies I and I_{new} with differents number of concepts of ontologies.

In Figure 4, we remark that I represented by I_1 decreases fastly than I_2 , I_3 and I_4 . We note that, the new coefficient of integrity I_{new} improves I and its evolution relies to the cardinality of ontology extended and the number of concepts added to it.

4 Simulations

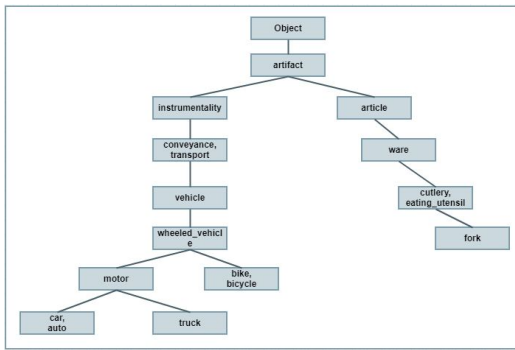
In this section, we experiment our methodology. We compare T_{Ngom} and N_{Plus} measures using coefficient of integrity I proposed in [5] and the new coefficient of integrity I_{new} proposed in this paper. We assess similarity of ontologies by using ontologies extracted from Wordnet¹. Figure 5 to 12 represent ontologies that we use to simulate the measures.

Ontologies are explained as following:

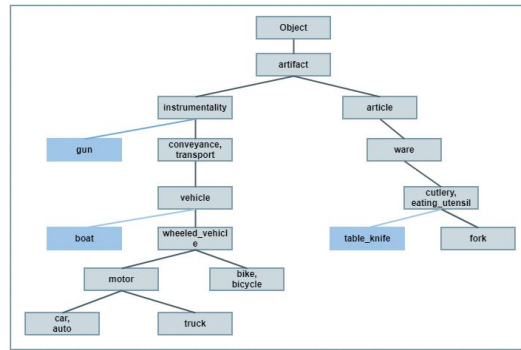
- ontologies O_3 and O_5 are fragments of Wordnet ;
- ontology O_4 is obtained by adding 3 concepts to O_3 (*gun*, *boat* and *table_knife*) ;
- O_6 is a sub-ontology of O_4 with concepts : *instrumentality*, [*conveyance*, *transport*], *vehicle*, *wheeled_vehicle*, *motor*, [*bike*, *bicycle*], [*car*, *auto*], *truck*, *gun*, *boat* ;
- O_7 is a sub-ontology of O_4 with concepts : *article*, *ware*, [*cutlery*, *eating_utensil*], *fork*, *table_knife* ;
- ontology O_8 is obtained by adding concept *plate* to O_7 ;
- ontology O_9 is obtained by adding concepts *bowl* to O_8 ;
- finally, ontology O_{10} is obtained by adding concepts *spoon* and *glass* to O_9 .

¹ <http://wordnet.princeton.edu>

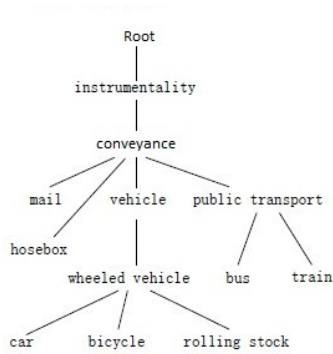
1:8 Similarity Between Two Ontologies



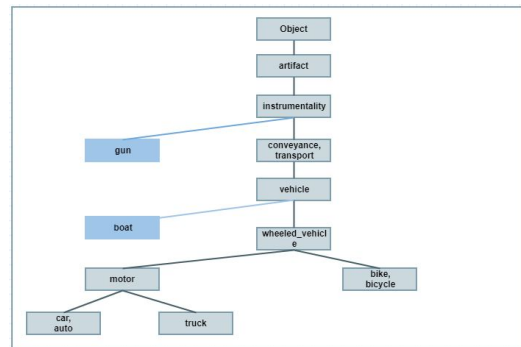
■ Figure 5 Ontology O_3 .



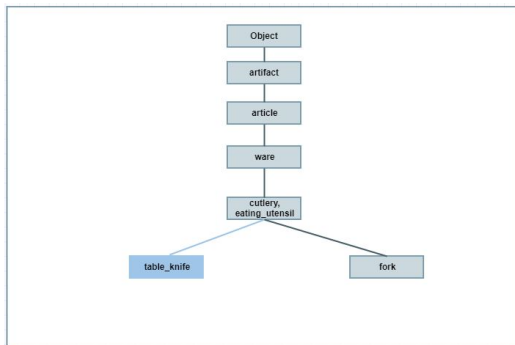
■ Figure 6 Ontology O_4 .



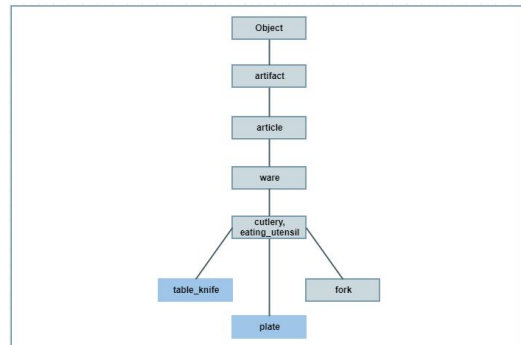
■ Figure 7 Ontology O_5 .



■ Figure 8 Ontology O_6 .



■ Figure 9 Ontology O_7 .



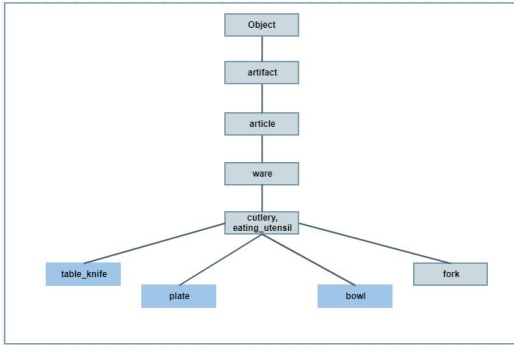
■ Figure 10 Ontology O_8 .

Table 1 gives results of comparisons between ontologies using T_{Ngom} , N_{Plus} and $N_{Plus}I_{new}$. $N_{Plus}I_{new}$ represents N_{Plus} with the new coefficient of integrity I_{new} . Note that similarities between ontologies O_3 and O_4 , and between O_3 and O_5 are assessed in [5].

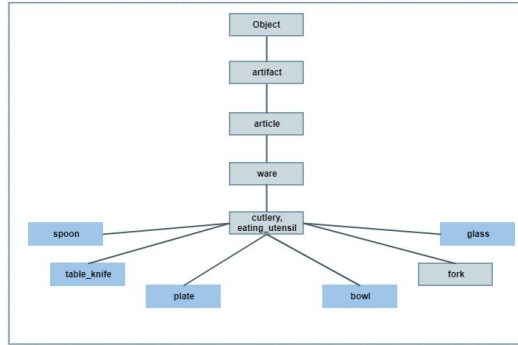
In table 1 we note that N_{Plus} improves T_{Ngom} by considering “is-a” relations to extends ontologies. In the same way, $N_{Plus}I_{new}$ improves N_{Plus} thanks to the improvement of the coefficient of integrity. Similarity of ontologies is better with $N_{Plus}I_{new}$. Values of similarities between ontologies are improved. The correlation coefficient between these three measures presented in table 2 is very good and very close to 1 ($correlation(N_{Plus}, N_{Plus}I_{new}) =$

■ **Table 1** Results of comparisons of ontologies with measures T_{Ngom} , N_{Plus} and $N_{PlusI_{new}}$.

	T_{Ngom}	Hierarchies added to ontologies for extensions		N_{Plus}	$N_{PlusI_{new}}$
(O_3, O_4)	0.95	in O_3: h_1 (instrumentality, gun), h_2 (vehicle, boat), h_3 ([cutlery, eating_utensil], table_knife)	in O_4: none	0.98	0.98
(O_3, O_5)	0.57	in O_3: h_1 (conveyance, mail), h_2 (conveyance, public_transport), h_3 (conveyance, hosebox), h_4 (wheeled_vehicle, rolling_stock)	in O_5: h_1 (wheeled_vehicle, motor)	0.74	0.82
(O_3, O_6)	0.76	in O_3: h_1 (instrumentality, gun), h_2 (vehicle, boat)	in O_6: none	0.88	0.9
(O_3, O_7)	0.6	in O_3: h_1 ([cutlery, eating_utensil], table_knife)	in O_7: none	0.83	0.86
(O_3, O_8)	0.49	in O_3: h_1 ([cutlery, eating_utensil], table_knife), h_2 ([cutlery, eating_utensil], plate)	in O_8: none	0.74	0.8
(O_3, O_9)	0.47	in O_3: h_1 ([cutlery, eating_utensil], table_knife), h_2 ([cutlery, eating_utensil], late), h_3 ([cutlery, eating_utensil], bowl)	in O_9: none	0.74	0.78
(O_3, O_{10})	0.43	in O_3: h_1 ([cutlery, eating_utensil], table_knife), h_2 ([cutlery, eating_utensil], late), h_3 ([cutlery, eating_utensil], bowl), h_4 ([cutlery, eating_utensil], spoon), h_5 ([cutlery, eating_utensil], glass)	in O_{10}: none	0.71	0.76
(O_7, O_9)	0.85	in O_7: h_1 ([cutlery, eating_utensil], plate), h_2 ([cutlery, eating_utensil], bowl)	in O_9: none	0.93	0.94
(O_7, O_{10})	0.76	in O_7: h_1 ([cutlery, eating_utensil], plate), h_2 ([cutlery, eating_utensil], bowl), h_3 ([cutlery, eating_utensil], spoon), h_4 ([cutlery, eating_utensil], glass)	in O_{10}: none	0.89	0.9



■ Figure 11 Ontology O_9 .



■ Figure 12 Ontology O_{10} .

■ Table 2 Coefficient correlation between measures T_{Ngom} , N_{Plus} and $N_{Plus}I_{new}$.

Measures	Correlation
$N_{Plus} - N_{Plus}I_{new}$	0.99
$N_{Plus} - T_{Ngom}$	0.98
$T_{Ngom} - N_{Plus}I_{new}$	0.99

0.99, $correlation(N_{Plus}, T_{Ngom}) = 0.98$ and $correlation(T_{Ngom}, N_{Plus}I_{new}) = 0.99$. With $N_{Plus}I_{new}$ measure, the correlation is better with N_{Plus} and T_{Ngom} measures. We can say that $N_{Plus}I_{new}$ is central with respect to the two measures T_{Ngom} and N_{Plus} .

5 Conclusion

In this paper, we have proposed a new coefficient of integrity I_{new} for the improvement of N_{Plus} measure proposed in [5]. In [5], the coefficient of integrity I introduced in N_{Plus} measure decreases and tends to 0 fastly. When we add just some concepts, the integrity becomes very poor. To fix this problem, we introduced R , the coefficient of representativeness of concepts added in the ontology for its extension. I_{new} decreases slowly compared to I and depends to the cardinality of the ontology extended and the number of concepts added to it too. Figure 4 shows how I_{new} decreases slowly relative to I . For simulations, we used wordnet and compared N_{Plus} , T_{Ngom} and $N_{Plus}I_{new}$. $N_{Plus}I_{new}$ is the measure which uses I_{new} and N_{Plus} uses I . The results of simulations show that $N_{Plus}I_{new}$ improves N_{Plus} thanks to I_{new} . The assessing of the coefficient of correlation between the three measures gives as results, good correlation between measures. According to the coefficient of correlation, we remark that $N_{Plus}I_{new}$ is central with respect to the two measures T_{Ngom} and N_{Plus} . In our future works, we will propose a methodology for extracting the smaller sub ontology of an ontology which represents a large domain. This sub ontology will be used for assessing similarity between ontologies because with a large ontology, the set of concepts not shared by ontologies will be very large and the similarity will be very poor.

References

- 1 Khadim Dramé. *Contribution à la construction d'ontologies et à la recherche d'information: application au domaine médical*. PhD thesis, Université De Bordeaux, 2014.
- 2 I. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Cybernetics and Control Theory*, 10(8), pages 707–710, 1966.

- 3 A. Maedche and S. Staab. Measuring similarity between ontologies. In A. Gómez-Pérez and V.R. Benjamins, editors, *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web. EKAW 2002. Lecture Notes in Computer Science, vol 2473*. Springer, Berlin, Heidelberg, pages 251–263, 2002.
- 4 A. N. Ngom. Etude des mesures de similarité sémantique basée sur les arcs. In *CORIA, Paris, France*, pages 535–544, 2015.
- 5 A. N. Ngom, G. Kaladzavi, F.K. Sangare, and M. Lo. Assessing similarity value between two ontologies. In *The 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3k 2018) – Volume 2:KEOD*, pages 343–350, 2018.
- 6 Aly Ngoné Ngom, Fatou Kamara-Sangaré, and Moussa Lo. Proposition of a method for assessing similarity between two ontologies. In *4th Annual Conf. on Computational Science and Computational Intelligence (CSCI'17) | Dec 14-16, 2017 | Las Vegas, Nevada, USA*, pages 174–179, 2017.
- 7 A. Tversky. Features of similarity. In *Psychological Review*, 84(4), pages 327–352, 1977.
- 8 Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *U.A.f.C.L. Stroudsburg, PA (ed.), In Proceedings of the 32nd annual meeting on ACL, volume 2 de ACL '94*, pages 133–138, 1994.

Automatic Classification of Portuguese Proverbs

Jorge Baptista   

University of Algarve, Faro, Portugal
INESC-ID Lisbon, Portugal

Sónia Reis   

University Algarve, Faro, Portugal
INESC-ID Lisbon, Portugal

Abstract

In this paper, natural language processing (NLP) and machine learning methods and tools are applied to the task of topic (thematic or semantic) classification of Portuguese proverbs. This is a difficult task since proverbs are usually very short sentences. Such classification should allow an easier selection of the most relevant proverbs for a given situation, considering their context in discourse or within a text. For that, we used, on the one hand, a collection of +32,000 proverbial expressions organized “thematically” into a large set of previously attributed topics (+2,200) and, on the other hand, the ORANGE data mining toolkit, along with the NLP and machine learning tools it provides. Since the classification provided in the collection of proverbs is, for the most part, based only on a keyword in the body of the proverbs, 2 experiments were set up, to determine the feasibility of the task with a modicum of effort and the most promising configurations applicable. Different sample sizes, 100 and 50 proverbs randomly selected per topic, corresponding to Scenario 1 and 2, respectively, were contrasted; several preprocessing strategies were explored, and different data representation methods tested against several learning algorithms. Results show that Neural Networks is the best performing model, achieving the best classification accuracy of 70% and 61%, in the two different experimental scenarios, Scenario 1 and 2, respectively. Some of the inaccurate classification cases seem to indicate that the machine learning approach can sometimes do a better job than a human classifier, especially considering the manual attribution of the topics by the collection’s author, the sheer number of topics involved, and the very unbalanced distribution of proverbs per topic. Based on the results achieved, the paper presents some proposals for future work to cope with such difficulties.

2012 ACM Subject Classification Human-centered computing

Keywords and phrases Portuguese Proverbs, Automatic Topic Classification, Machine Learning

Digital Object Identifier 10.4230/OASICS.SLATE.2022.2

Supplementary Material *Dataset*: <https://doi.org/10.13140/RG.2.2.22354.02242>

Funding *Jorge Baptista*: This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project ref. UIDB/50021/2020.

1 Introduction

This paper aims at the automatic topic/thematic classification of (European) Portuguese proverbs. The semantic (or thematic) classification of proverbs is not, *a priori*, a trivial task, not only for the very nature of proverbs as short sentences, relatively insulated from the surrounding text; but especially because of the idiomatic (figurative) character of many of these expressions; this difficulty is also due to the fact that, for a given theme, the meaning of the words forming the proverb is not necessarily related to the expression by which we might designate that same theme. For example, in [5], we find *Água mole em pedra dura, tanto dá até que fura* lit.: “water soft on stone hard, so much it hits until it bores the stone” “Water dropping day by day wears the hardest rock away”¹ under the topic PERSEVERANÇA

¹ Equivalent suggested by: <https://www.sk.com.br/sk-proverbios-portugues-ingles.html>



“perseverance”. Arguably, even if no clue is given in its wording, it may be easier to assign this topic to this very well known proverb, than to the following one: *A pouco e pouco fia a velha a touca* “Little by little the old woman spins the cap” which the author also classes under the same topic, but, as it is much less known and used, where the lexical clues may hint at several other potential meanings (persistence, carefulness/wisdom, progressive, wealth accumulation, etc.). Compare how much easier it is to classify under that same topic a proverb that explicitly has that word as its subject, and whose meaning is clearly denoted by its elements: *A perseverança sempre/tudo alcança/vence* “Perseverance always pays off/wins out everything” (Lexical or punctuation variants of the same *paremiological unit* are represented in a compact way using “/”. On the definition of this concept, see [10].) Added to these difficulties is the fact that, on the one hand, some proverbs can be used in different situations, with subtle variations in meaning.

Identifying the theme of a proverb can be useful in several situations. According to [8], “the task for the automatic classification of proverbs which can assist users in selecting a suitable proverb according to a given context prove[s] to be important and interesting but challenging.” Thus, when serving as a *motto* to a text, either in the title or in the *lead* of a newspaper article, or even in the conclusion of the text, the proverb anticipates (or ends the text with) a judgment of value that the author avoids making in the first person, taking refuge in the so-called “popular wisdom”. This may be viewed as the underlying *rationale* behind the work of [6], who assessed different methods of associating proverbs to news headlines, exploring different semantic similarity metrics.

In this paper we address the practical problem of automatic topic classification of Portuguese proverbs and try to provide answers to the following research questions:

1. How are proverbs classified by topic/theme in the relevant literature? How good is the adequacy, the consistency or even the usefulness of extant classifications in the available collections of Portuguese proverbs?
2. To what extent can the available classification of Portuguese proverbs be captured by NLP and ML tools and techniques, using only the words in the proverbs? Is the size of each class relevant for this task, considering that proverb classifications in the literature are very fragmentary (a large number of topics, with a small number of proverbs each), in spite of the large collections of proverbs available?

This paper is organized as follows: Next, Section 2 presents the most similar works found in the literature on machine learning strategies to proverb classification. Section 3 presents a critical assessment of the collection of proverbs used here as corpus, its structure and composition. Then, Section 4, explains the sampling method used to select the expressions here used and outlines the two experiments conducted using the ORANGE [2] data mining toolkit ², along with the results obtained. In Section 5, a short discussion ensues, Finally, Section 6 concludes the paper and presents prospects for future work.

2 Related work

There seems to be very little related work dealing with the automatic, machine-learning based, classification of proverbs. These are presented in this Section.

Noah and Ismail (2008) [8] is the first work, to the best of our knowledge, to address this challenge in a similar way, aiming at helping an end-user to better select the most appropriate proverb for a given context. The authors comment on the non-trivial nature of the task,

² <https://orangedatamining.com/>

since proverbs are very concise structures, thus limiting the type and number of features that can be extracted from them and the classification techniques that can be applied. The paper draws on a relatively small corpus of 1,000 Malayan proverbs, split into two partitions 50% for training and 50% for testing. The data was divided into equal-sized 5 classes, according to the respective topics (FAMILY, LIFE, DESTINY, SOCIAL and KNOWLEDGE). These topics are generic in nature, and encompass proverbs that pertain to many situations and contexts. Three classification scenarios were considered: (i) the proverbs, alone; (ii) the proverbs along with an explanation of their meaning; and (iii) the proverbs, their explanation and an example of a sentence containing those proverbs. The data was preprocessed, removing the stopwords from the proverbs, to achieve a reduction of the number of extracted features, as well as obtaining better performing ones. No other preprocessing step was indicated. Two Naïve Bayes models (multinomial and Bernoulli's multivariate) were tested. In the training stage, a classification accuracy (CA) of 99% is reported. In the testing step, the best reported result was the third scenario (proverb+explanation+example), with a CA=72.2% for the multinomial model and 68.2% for the multivariate. The authors also report an increase in the performance depending on the size of the vocabulary, from a CA of 36% for a minimal vocabulary of 632 words, to the reported value of 72.2% with a 3,203 words' vocabulary.

For this paper, only the first scenario can be reproduced. Firstly, no collection of Portuguese proverbs was found with their respective explanation. Secondly, while some experiments have been made to determine the distribution of proverbs from the Portuguese *paremiological minimum* [10] in 3 large corpora [11], in most cases, the proverb forms by itself an isolated sentence, with very little formal relation with the surrounding text, working as a textual “island”, as it has been already remarked in the literature [3, p.37; 221-222]. This offers little to no help in the classification of proverbs by topic.

Next, Mendes & Oliveira (2020) [6] assessed different similarity metrics in order to associate proverbs with news headlines. This is basically an automatic recommendation task, similar to the one outlined by [8]. The corpus of news headlines was obtained using a news API from online Portuguese newspapers, involving some keywords (CLIMA “climate”, AMBIENTE “environment” and AQUECIMENTO GLOBAL “global warming”), retrieved during the 3 months prior to February 2020 (the precise number of news headlines was not provided). The corpus of proverbs consisted of a list of approximately 1,600 expressions, drawn from the project NATURA (U. Minho) [1]³ (The selection criteria of these proverbs, from the source database, which contains 2,293 proverbs, was not given). Both corpora were processed (tokenized, part-of-speech tagged and lemmatized, and stopwords removed). To estimate semantic similarity between the news headlines and the proverbs, several basic techniques were used (Jaccard coefficient, word count and TF-IDF vectorization), as well as static word embeddings (Glove and FastText) and BERT (references within the paper). To assess the results, the authors used a questionnaire, which revealed that most of the time people could establish a relation between the automatically selected proverb and the news headline. This was more obvious when there were common lexical elements between them, thus recommending the use of simpler computational methods, like the Jaccard coefficient. Deeper semantic representations, such as word embeddings (BERT), produced poorer results, which is explained by the authors by the figurative nature typical of these proverbial expressions.

³ <https://natura.di.uminho.pt/~jj/pln/proverbio.dic>

Though this paper does not frame the proverbs' classification within an extrinsic evaluation, that is, in an applicational context, as a recommendation task; it does compare bag-of-words with word embeddings data representation methods, as well as different types of learning models for the classification. A previously classified list of proverbs were used, selecting the most frequently occurring topics within a very large collection (+32,000 expressions), and using same-sized classes.

3 Corpus

For this work, we used a relatively large collection [5] of over 32,000 proverbs, organized by what the author called an “interpretative classification essay” (p.12). This is a task that has raised doubts for the author himself, who even mentions that some proverbs included in the collection “do not contain any “far-reaching principle” nor contain a moral maxim confirmed by the course of generations”, while some cases can be deemed as just “non-sense”. The author also added a list of 167 proverbs “without classification”: “[...] some dozens of sayings, of difficult interpretation because its meaning is multiple or diffuse, or eventually so personalized that only a deep investigation could, in some cases, determine it (p.14, our translation)”. The extensive list of proverbs from this collection had already been digitized and integrated into the database that was the main source of [9]’s work.

Going through the collection, it turns out that the “organization” of the proverbs collected there is essentially based on the fact that most proverbs under a given “topic” present that same word or cognate forms of the word designating the “topic” itself. While this may very well be adequate for the topic *brigas* “fights” and proverbs like *Brigam dois se um quer* “Two fight if one will”, it is much less obvious when the proverb’s interpretation is predominantly idiomatic, like the inclusion under the topic *brilho* “shine” of the proverb *O brilho abre o trilho* (lit.: “The shine/glow opens the trail/way”) “Someone’s success/fame makes it easier for something to happen or for someone to do something”. In other cases, cognate words are split into distinct topics: PERDA (noun) “loss”, PERDER (verb) “loose” and PERDER-SE (pronominal verb) “loose oneself/get lost”, without a consistent distribution of the proverbs within each group, namely, proverbs with the verb but noun in the group headed by the noun, and vice-versa. It is unlikely that joining some topics designated by cognate words from different parts-of-speech would reduce this fragmentary classification, since this seldom happens in this collection.

This also leads to a fragmentary nature of the “classes” construed by the author. There are cases where a given “theme/topic” presents only one, two or very few proverbs, or even different variants of the same proverb, sometimes even with the same keyword, e.g. ALQUIMIA “alchemy”: *Alquimia e/está provada :/, ter renda e não pagar/gastar nada* “Alchemy is proven :/, to have income and don’t spend/pay anything”. In other cases, a synonym of the keyword is used instead: (ALMOFARIZ “mortar”: *Nem corte sem chocarreiro, nem gral sem malhadeiro* (gral = *almofariz* “mortar”) “Neither a court without a court jester, nor a mortar without a pestle”.

In the other extreme situation, some “topics” are so broad, e.g. AGRICULTURA “agriculture”, that the proverbs there included address the most disparate situations: from the quality of the soil to produce: *Terra negra dá bom pão* “Black soil makes good bread”; to the relationship between weather and crops: *Ano de rosas, ano de pão* “Year of roses, year of bread”; or the need to care for the crops: *Vinha sem guarda, vindima feita* “Unguarded vine, harvest done”. In other cases (e.g. ALIMENTO “food”), different food products (e.g. *carne* “meat” and *peixe* “fish”; *fruta* “fruit” and *legumes* “vegetables”; *leite* “milk”, *pão* “bread” and *mel* “honey”) give title to the different “subclasses”.

Finally, several proverbs occur more than once in the collection, but they are not classified consistently, e.g. the proverb *Se queres mel, suporta a abelha* “If you want honey, support the bee” appears both under the “topic” MEL “honey” and as a subtype of ALIMENTO “food” (but not under ABELHA “bee”), thus confirming the difficulty in identifying (and naming) the proper topic of a given proverbial expression.

Despite all the reservations that this so-called “classification” may rise, there are few works, especially recent ones, that present this kind of topic/thematic organization (we know of [12], for instance). Other collections present only alphabetically ordered listings of the words appearing in the proverbs [4], eventually accompanied by a remissive index of the vocabulary [7]. For lack of a better resource, this collection [5] was used to this paper, not least because its size, but also because it had already been digitized into the database that was used for [9], and it only required that topics be associated to the proverbs.

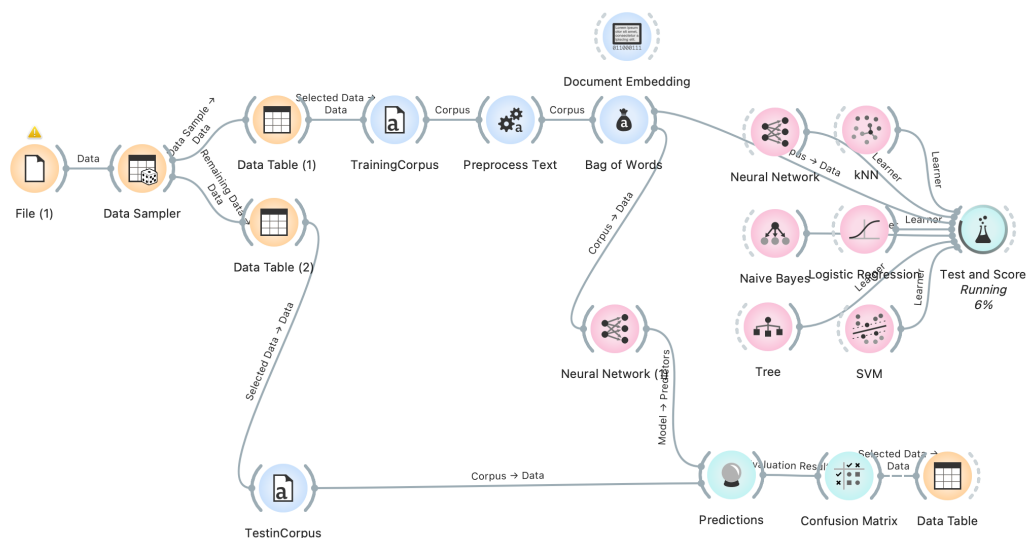
4 Methods

In total, [5] collects 32,067 proverbs, distributed over 2,227 different “topics” (an average of approximately 14,4 proverbs per topic). There is a long tail of topics with only one proverb (482), 112 topics with 50 or more proverbs each (14,078 proverbs), and only 43 topics with 100 (9,537 proverbs) or more.

A subcorpus with the largest classes, that is, the topics with the largest set of proverbs, were selected. Two classifications scenarios were considered in these experiments:

- **Scenario 1:** only the 43 most frequent topics, with +100 proverbs each, were selected;
- **Scenario 2:** the 112 topics with 50+ proverbs were considered.

A random sample of 100 and 50 proverbs per topic and for each scenario, respectively, was retrieved from the subcorpus⁴. Each topic has exactly the same number of proverbs in both scenarios. The proverbs in Scenario 2 are included in Scenario 1. Using the ORANGE [2] machine-learning toolkit, the workflow presented in Figure 1 was set up.



■ **Figure 1** ORANGE workflow configuration.

⁴ Available at: <http://dx.doi.org/10.13140/RG.2.2.22354.02242>

2:6 Automatic Classification of Portuguese Proverbs

Each scenario was tested consecutively. The datasets for the two scenarios consist of samples of 4,300 and 5,600 proverbs each. As shown in Fig. 1, a DATASAMPLER widget was used to split the data into fixed-sized, training/testing partitions, with replicable, stratified sampling options selected, 4,000/300 in Scenario 1 and 5,000/600 in Scenario 2, respectively. Preprocessing consisted of text transformation (conversion to lowercase), tokenization (keeping words and punctuation) and part-of-speech tagging (averaged perceptron tagger), using the default configurations of the system. Two data representation options were compared: BAG-OF-WORDS (BoW) and DOCUMENTEMBEDDINGS. After preliminary experiments, it was clear that the Document Embeddings data representation method consistently produced very poor results (Classification Accuracy (CA): 0.030) against the bag-of-words (CA: 0.415), so it was discarded. Several models were evaluated, using the TEST&SCORE widget in a 10-fold cross-validation setting: Neural Networks (NN), Nearest Neighbour (kNN), Naive Bayes (NB), Logistic Regression (LR), a forward pruning decision tree algorithm (Tree), and Support Vector Machines (SVM). Table 1 shows the performance of each model, estimated by the TEST&SCORE tool in Scenario 1, and Table 2 in Scenario 2. Evaluation of the best performing model (Neural Networks) with the testing corpus yielded the results shown in Table 3.

■ **Table 1** TEST&SCORE: Evaluation of different models in Scenario 1: 43 most frequent topics and 100 randomly selected proverbs per topic. Metrics: AUC: area under the ROC curve, CA: classification accuracy, F1: harmonic mean (of Precision and Recall).

Model	AUC	CA	F1	Precision	Recall
SVM	0.69522	0.01375	0.01066	0.05234	0.01375
kNN	0.78336	0.36150	0.36166	0.43864	0.36150
Tree	0.76424	0.44850	0.45341	0.47162	0.44850
Naive Bayes	0.93931	0.59500	0.58550	0.60109	0.59500
Logistic Regression	0.95132	0.66150	0.66299	0.66799	0.66150
Neural Network	0.95068	0.68825	0.68531	0.68640	0.68825

■ **Table 2** TEST&SCORE: Evaluation of different models in Scenario 2: 112 most frequent topics and 50 randomly selected proverbs per topic.

Model	AUC	CA	F1	Precision	Recall
SVM	0.67129	0.00220	0.00310	0.01805	0.00220
Tree	0.60671	0.12900	0.13154	0.17824	0.12900
kNN	0.74069	0.24480	0.25509	0.37730	0.24480
Naive Bayes	0.91620	0.48580	0.48219	0.52645	0.48580
Logistic Regression	0.92985	0.57740	0.57985	0.59201	0.57740
Neural Network	0.93662	0.60480	0.60425	0.61241	0.60480

■ **Table 3** Results from best performing model, Neural Networks (NN), in both Scenario 1 (train: 4,000 proverbs / test: 300 proverbs) and Scenario 2 (train: 5,000/test: 600).

Scenario	Model	AUC	CA	F1	Precision	Recall
Scenario 1	NN	0.95900	0.70300	0.69900	0.71800	0.70300
Scenario 2	NN	0.94300	0.61200	0.61000	0.64600	0.61200

5 Discussion

From the TEST&SCORE widget (Tables 1 and 2), the best performing model in both scenarios is the Neural Networks (CA: 0.688 and 0.605, respectively), with Logistic Regression following close behind (CA: 0.662 and 0.577, respectively). The relative performance of the tested models is similar and consistent across the two scenarios, with only kNN and Tree changing ranks. Overall, the reduction of the number of proverbs by topic degraded the models' predicted performance. The testing step (Table 3), however, produced slightly better results, while keeping the same difference. The best performance was achieved in Scenario 1 (CA: 0.703).

The experimental designs presented in related work (Section 2) do not allow for a straightforward comparison of results. Even so, we notice that the best performing models in the experiments on Malayan proverbs, while including in the training dataset not only the proverb itself, but also both a definition and an example of the proverb in its context of use, only achieved a 72.2% accuracy.

Looking deeper into the classification errors in Scenario 2, we find that in 77 out of 233 errors the proverb contains the predicted keyword, and in most cases, instead of the target keyword, a cognate word, from a different part-of-speech appears: Proverb: *Prometer e não dar é dever e não pagar* “To promise and not give is to owe and not pay”; Actual: *promessas* “promises” (noun-pl.); Predicted: *dever* “to owe” (verb)/“duty” noun.

Some of these results are very interesting, for even if the topic classification is inaccurate, some clear relation can be established between the predicted tag and the respective proverb. In some cases, one can even say that the machine learning model produced a better result than the human annotator: Proverb: *Onde entra o ar e o sol, não entra o doutor* “Where the air and the sun enter, the doctor does not enter”; Actual: *sol* “sun”; Predicted: *saúde* “health”; Proverb: *Mal vai ao passarinho na mão do menino* “Badly goes the little bird in the little boy’s hand”; Actual: *aves* “birds”; Predicted: *criança* “child”.

6 Conclusion and future work

This study aimed at an automatic topic (semantic) classification of Portuguese proverbs. The ORANGE data mining toolkit was used to produce models that could appropriately assign a topic/theme to Portuguese proverbs. Different models were trained and tested in 2 experimental Scenarios, varying the number of proverbs (100 and 50) per topic. These were drawn from a large collection [5] of Portuguese proverbs (over 32,000 expressions), already classified according to a large set of topics (over 2,200). The data sampling, the preprocessing configuration and the data representation methods were presented and discussed. In these experiments, the best performing model was Neural Networks, achieving a best classification accuracy of 70% in Scenario 1 (100 proverbs per topic) and 61% in Scenario 2 (50 proverbs per topic). Results are encouraging and in line with previous related work [8]. However, there is still much room for improvement. Going through the cases of inaccurate automatic classification, it seems that sometimes the model does a better job in predicting the appropriate class than the human classification.

Some of the problems observed derive from the topic classification itself as it was presented in the proverb collection, since this seems to have been less than rigorous and often inconsistent, perhaps as a consequence of the large set of proverbs here collected and the (apparently) manual procedure in the classification (no explicit criteria nor method of classification was provided). Furthermore, the high number of classes and the non-uniform distribution of proverbs across these classes do not allow for a straightforward application of the machine-learning methods here used to the entire collection.

In the future, we would like to be able to automatically classify the entire database of proverbs (+114,000) used by [9], which contains those proverbs of the collection here used [5], but also encompasses 3 other collections. For this, there is still a long way to go. First, we believe that the set of topics must be substantially reduced, as the machine-learning methods available in the ORANGE toolkit degrades when over 100 classes are used. Secondly, the classification, in our view, should be more semantic in nature, and not so much based on the presence of a given keyword, as it seems to be the case in [5]. It is likely that, in spite of the *caveat* from [6], different word embeddings (other than from those distributed with ORANGE), purposefully built for Portuguese (e.g. BERTimbau, [14]; LX-DSemVectors [13]) could improve the results. Eventually, vocabulary filtering or other machine-learning techniques such as clustering algorithms could be used to this end, and help determine the most adequate set of topics. Finally, semantic similarity metrics could be used to rank/associate same-topic proverbs and then to confront those data-driven classifications with human assessment, or, alternatively, with instances of proverbs in context [11].

References

- 1 José João Almeida. Dicionário aberto de calão e expressões idiomáticas [online]. Available at <http://natura.di.uminho.pt/~jj/pln/calao/dicionario.pdf>, 2014.
- 2 Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. *Orange: Data Mining Toolbox in Python*. *Journal of Machine Learning Research*, 14:2349–2353, 2013.
- 3 Ana Lopes. *Texto Proverbial Português: elementos para uma análise semântica e pragmática*. PhD thesis, Universidade de Coimbra, Coimbra, Portugal, 1992.
- 4 José Pedro Machado. *O Grande Livro dos Provérbios*. Editorial Notícias, Lisboa, 1996.
- 5 José Ricardo Marques da Costa. *O Livro dos Provérbios Portugueses*. Editorial Presença, Lisboa, 1999.
- 6 Rui Mendes and Hugo Gonçalo Oliveira. Comparing different methods for assigning Portuguese proverbs to news headlines. In *11th International Conference on Computational Creativity (ICCC'20)*, pages 153–160, 2020.
- 7 António Moreira. *Provérbios Portugueses*. Editorial Notícias, 1996.
- 8 S.A. Noah and F. Ismail. Automatic classifications of Malay proverbs using naïve Bayesian algorithm. *Information Technology Journal*, 7:1016–1022, 2008.
- 9 Sónia Reis. *Expressões proverbiais do português: Usos, variação formal e identificação automática*. PhD thesis, Universidade do Algarve, Faro, Portugal, 2020.
- 10 Sónia Reis and Jorge Baptista. Determinação de um mínimo paremiológico do português europeu. *Acta Scientiarum. Language and Culture*, 2(42):e52114, 2020.
- 11 Sónia Reis, Jorge Baptista, and Nuno Mamede. Provérbios portugueses usuais: distribuição em corpora. In *Anais do XIII Simpósio Brasileiro de Tecnologia da Informação e da Linguagem Humana*, pages 325–334. Sociedade Brasileira de Computação, 2021.
- 12 Fernando Ribeiro de Mello. *Nova Recolha e Provérbios Portugueses e outros lugares-comuns*. Edições Afródite, 2 edition, 1986.
- 13 João Rodrigues, António Branco, Steven Neale, and João Silva. LX-DSemVectors: Distributional Semantics Models for Portuguese. In *International Conference on Computational Processing of the Portuguese Language (PROPOR 2016)*, pages 259–270. Springer, 2016.
- 14 Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. BERTimbau: Pretrained BERT Models for Brazilian Portuguese. In Ricardo Cerri and Ronaldo C. Prati, editors, *Intelligent Systems. BRACIS 2020*, pages 403–417. Springer, 2020.

Question Answering For Toxicological Information Extraction

Bruno Carlos Luís Ferreira ✉

DEI, CISUC, University of Coimbra, Portugal

Hugo Gonalo Oliveira ✉ 


DEI, CISUC, University of Coimbra, Portugal

Hugo Amaro ✉

LIS, Instituto Pedro Nunes, Portugal

Ângela Laranjeiro ✉

Cosmedesk, Coimbra, Portugal

Catarina Silva ✉ 

DEI, CISUC, University of Coimbra, Portugal

Abstract

Working with large amounts of text data has become hectic and time-consuming. In order to reduce human effort, costs, and make the process more efficient, companies and organizations resort to intelligent algorithms to automate and assist the manual work. This problem is also present in the field of toxicological analysis of chemical substances, where information needs to be searched from multiple documents. That said, we propose an approach that relies on *Question Answering* for acquiring information from unstructured data, in our case, English PDF documents containing information about physicochemical and toxicological properties of chemical substances. Experimental results confirm that our approach achieves promising results which can be applicable in the business scenario, especially if further revised by humans.

2012 ACM Subject Classification Computing methodologies → Information extraction

Keywords and phrases Information Extraction, Question Answering, Transformers, Toxicological Analysis

Digital Object Identifier 10.4230/OASICS.SLATE.2022.3

Funding This work was partially funded by: the project SafetyDesk: Smart Toxicological Analysis of Chemical Substances (CENTRO-01-0247-FEDER-113485), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Regional Operational Programme Centro 2020; and national funds through the FCT – Foundation for Science and Technology, I.P., within the scope of the project CISUC – UID/CEC/00326/2020 and by the European Social Fund, through the Regional Operational Program Centro 2020.

1 Introduction

With the increasing volume of available data, companies need to develop processes for mining information that may be essential for their business. Unfortunately, much of this information is not present in structured databases, but rather in unstructured or semi-structured texts. In many cases, humans are capable of doing this process, however, it can take a long time to complete. *Information Extraction* (IE) emerges as a solution to deal with this problem [4].

In real business scenarios, document processing typically focuses on narrow and specific topics rather than general and wide domains. In such scenarios, annotated data, categorized and labeled for *Artificial Intelligence* (AI) applications, and thus ready for supervised learning approaches, is very limited, and the annotation process is still a challenging task, due to the required time and logistics.



© Bruno Carlos Luís Ferreira, Hugo Gonalo Oliveira, Hugo Amaro, Ângela Laranjeiro, and Catarina Silva;

licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: Jo o Cordeiro, Maria Jo o Pereira, Nuno F. Rodrigues, and Sebast o Pais; Article No. 3; pp. 3:1–3:10

OpenAccess Series in Informatics



Schloss Dagstuhl – Leibniz-Zentrum f r Informatik, Dagstuhl Publishing, Germany

In our case study, the problem in question emerged from the necessity of optimising the time it takes to elaborate the toxicological profile of a chemical substance. Currently, the process consists of a human searching for information about the chemical substance and preparing a report with all the relevant information, i.e., physicochemical and toxicological properties. The research resorts to different types of databases, including some where data is structured (e.g., websites) and where data is unstructured (e.g., PDFs, specific papers).

Reviews and reports on toxicological profiles, available in PDF, contain much information written by human experts in an unstructured format, i.e. natural language (English). This includes relevant information, such as physicochemical and toxicological information about substances, which currently needs to be manually extracted for further comparison with the other sources, e.g., for cross validation. This is typically a time-consuming and labor-intensive process.

We address the problem of extracting toxicological information from those PDFs by using state-of-the-art *Transformer* models fine-tuned for Extractive *Question Answering* (QA). The key of our approach is to ask useful questions in order to extract relevant information about toxicological properties given paragraphs from the PDFs.

We next review related work, and describe the task and data preparation. We introduce and elaborate our approach in Section 3, report on some experiments, including preliminary results, in Section 4 and we finally draw conclusions in Section 5.

2 Related Work

Information Extraction from text is an important task of *Natural Language Processing* (NLP), that converts unstructured documents to structured data. There are two main methods for this: rule-based and supervised machine learning. Rule-based methods typically rely on handcrafted textual and linguistic patterns that commonly transmit the entities and relations to extract. In contrast, supervised machine learning exploits features to train a classifier that can distinguish extracted information, either for labelling the sequence of words [13] or for generating entities and relations from it [3].

As supervised machine learning techniques require manually labeled training data, which is one of the major drawbacks of these techniques, unsupervised IE techniques emerged. These techniques extract entity mentions from the text, clusters the similar entities and identify relations [6]. Researchers have introduced Open Information Extraction (OpenIE), an unsupervised machine learning technique, which is a relation-independent paradigm that extracts a large set of relational tuples in an open-domain paradigm [1]. However, given that is an unsupervised method, OpenIE has no idea about the types of entities and relations extracted, so the usage of other knowledge bases from external sources is necessary in order to learn the relations in a corpus [1], a drawback of OpenIE.

Great advances taken in the state of the art in 2017 with the introduction of the *Transformer* neural networks [14] and the consequent emergence of *neural language models* (LMs), like BERT [5], RoBERTa [9], ELECTRA [2], which can be fine-tuned for a broad range of NLP tasks.

An alternative to the previous approaches for IE, especially when lacking training data, is to formulate IE as a Question Answering problem, using transformer models fine-tuned for this task. In order to do so, other researchers [11, 10, 7] acquired a pre-defined list of required information and represented it as key phrases, e.g., “Name of institution” or “Deadline for bidding” [10]. Using the pre-defined key phrases, they can be considered as a question, or part of one, the input document can be treated as the context and the extracted information from a document can be considered as an answer.

3 Our Approach

Within the broad space of business documents, as mentioned in Section 1, we were faced with the challenge of accelerating the process of filling toxicological reports, so we focused on one specific type of documents: studies of individual chemical substances. Our objective is to extract specific properties (information) from those studies (input documents).

However, given an input document, there are multiple pages, multiple paragraphs and multiple phrases that contain information, regarding physicochemical and toxicological properties of chemical substances. Simply providing the complete document to the LMs is a problem because the LMs used are limited in the size of the context. To tackle that problem we decided to do divide the documents into sections, where each section contains the information regarding a specific property of the chemical substance.

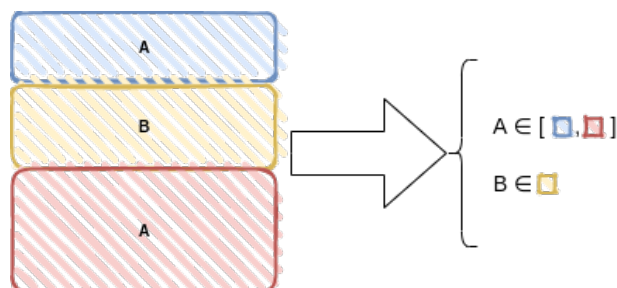
Based in our objective and restriction our approach follows two main steps:

1. Identify the section of the document about a specific property.
2. Get the information for that property by asking a question to the model, using the previously identified section as context.

3.1 Information Pre-processing

The pre-processing process consists of dividing the input document into sections, where each section contains the information regarding a specific property of the chemical substance. That way, we minimize the context given to the LMs, eliminating noise, i.e. parts of the document not relevant for each property.

As a visual example, in Figure 1 we have the properties *A* and *B* where the property *A* has information in section *Blue* and *Red* and property *B* has information in section *Yellow*. It is not necessary to search for information regarding property *B* in all the three sections, only in section *Yellow*.



■ **Figure 1** Graphical example of the pre-processing process.

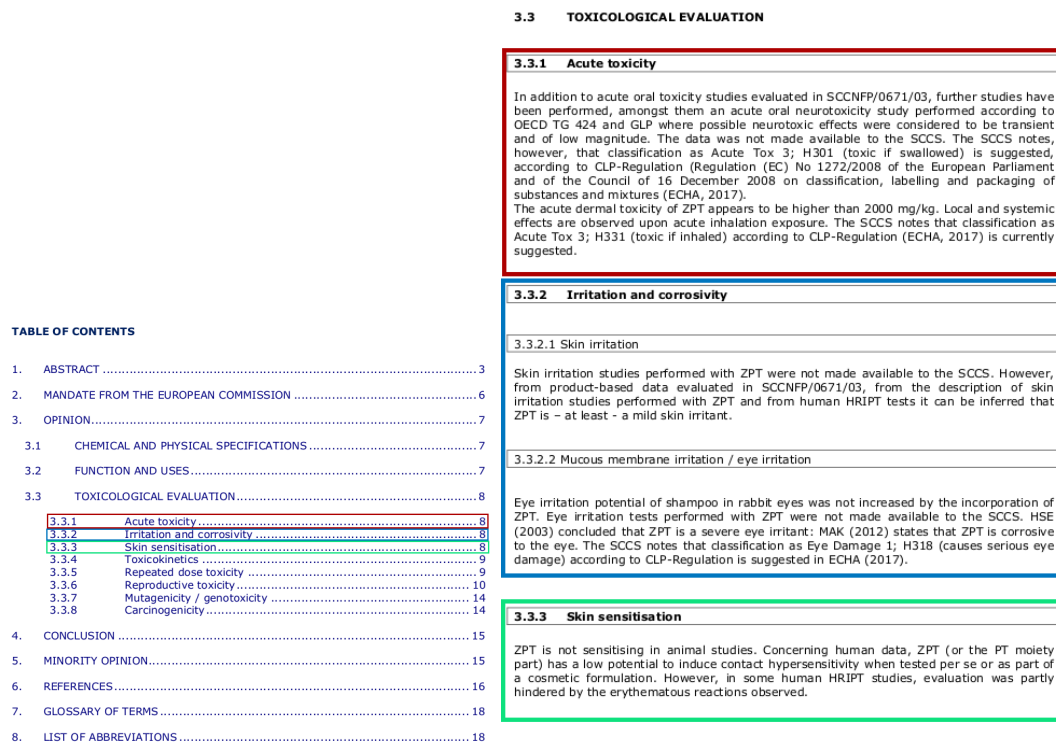
In our approach, the division of the document in sections is based on the *Table of Contents* (TOC) that the document has. This process is similar to that of a human when navigating and searching the document using the Index/TOC. It is possible to use the TOC as the reference point for the division of the document into sections because one is commonly present in the reports of toxicological profiles we have been using.

The input PDF documents were converted to text with the *pdfplumber*¹ parser, and, combined with Regular Expressions, we could obtain the TOC of the document. The usage of the TOC allows us to find the start and the end of each section, i.e., by considering the

¹ <https://github.com/jsvine/pdfplumber>

3:4 Question Answering For Toxicological Information Extraction

number and title of the sections, where the start corresponds to the section title obtained in the obtained TOC and the end of the section is the starting of the next section with the same hierarchical level. Figure 2 represents the pre-processing process, where the information obtained from the TOC (Figure 2a) help us divide the document into sections (Figure 2b).



(a) Table of contents of document with sections visually indicated. (b) Page of document with sections visually divided.

■ **Figure 2** Example of the pre-processing process in a document.

Being able to divide the input document into sections allows us to increase the performance and reduces the noise because, by limiting the context that we provide to the LMs, we can guarantee that the answers obtained are connected to the substance's property information.

3.2 QA for IE

Having the context defined, we can use the QA models for extracting information. In order to do so, we need to identify and set questions related to the context and to the information that we want to obtain. For this task, we can explore available Transformer models fine-tuned in the *The Stanford Question Answering Dataset* [12] (SQuAD), which includes paragraphs (contexts), questions about them, and extracted answers. SQuAD has two versions, 1.1 and 2.0. The main difference between them is that version 1.1 contains 100,000+ question-answer pairs on 500+ contexts, while version 2.0 combines the 100,000 questions in SQuAD 1.1 with over 50,000 unanswerable questions written adversarially by crowdworkers to look similar to answerable ones. This means that models trained with SQuAD 2.0 not only answer questions when possible, but can also determine when no answer is possible from the given paragraph and abstain from answering [12].

As SQuAD uses the Six W’s (Who, What, When, Where, Why and How) in the formulation of the questions, we also need to create questions of this kind, regarding each information that we want to extract. For example, in the sentence present in one of the PDF documents used, “Eye irritation potential of shampoo in rabbit eyes was not increased by the incorporation of ZPT” we want to obtain the species that the test applies to, so we can formulate a question as “What is the species?”. Given the sentence (as the context) and the question, we hope to obtain from the QA models the right answer, in this case, “rabbit”.

After the pre-processing step, we can give to the QA models each section of the document that corresponds to a specific property of a substance as context. In order to optimize our approach, we need to create the right set of questions per section.

4 Experiments

We focus our experiments in one source of studies of individual chemical substances: *Scientific Committee on Consumer Safety* (SCCS) Opinions². For experimentation purposes, we focus on a subset of 60 documents, issued by the Committee³, dated from April 2016 to December 2021. Our objective is to extract relevant information about certain toxicological properties of substances. Table 1 shows a sample of relevant information about the target toxicological properties for this case study.

■ **Table 1** Substances properties information.

Substance Property	Information to Extract
Repeated Dose Toxicity	NOAEL ⁴ value
Acute Toxicity	Species used in study; OECD ⁵ Guideline used; Exposure route
Irritation	Species used in study; OECD Guideline used; Exposure route
Mutagenicity	OECD Guideline used; Classification
Skin Sensitization	OECD Guideline used; Classification; Concentration used in study
Carcinogenicity	Species used in study; OECD Guideline used; Classification
Photo-induced Toxicity	OECD Guideline used; Classification
Reproductive Toxicity	Species used in study; OECD Guideline used; Classification

4.1 Setup

The source documents were pre-processed (see Section 3.1) and a set of questions was formulated for each information to extract (see Table 2). In this case study, all the questions start with “what” because, by trial and error, we noticed that relevant information was frequently obtained with questions like “what is the *[specific information to extract]* ?”.

In our experiments we tested the set of questions in three different QA models, all available from the Huggingface Transformers hub, and usable from the transformers library⁶: *BERT-base-cased-squad2*⁷, *RoBERTa-base-squad2*⁸ and *BioBERT-v1.1-pubmed-squad-v2*⁹.

² https://ec.europa.eu/health/scientific-committees/scientific-committee-consumer-safety-sccs/sccs-opinions_en

³ https://ec.europa.eu/health/scientific-committees/former-scientific-committees/scientific-committee-consumer-safety-2016-2021/sccs-opinions-2016-2021_en

⁴ No Observed Adverse Effect Level

⁵ Organisation for Economic Co-operation and Development

⁶ <https://huggingface.co/>

⁷ <https://huggingface.co/deepset/bert-base-cased-squad2>

⁸ <https://huggingface.co/deepset/roberta-base-squad2>

⁹ https://huggingface.co/ktrapeznikov/biobert_v1.1_pubmed_squad_v2

■ **Table 2** Example of set of questions per property tested.

Substance Property	Questions
Repeated Dose Toxicity	What is the NOAEL value?
Acute Toxicity	What is the guideline?;What is the species?
Irritation	What is the guideline?;What is the species?
Mutagenicity	What is the Guideline?;What is the conclusion?
Skin Sensitization	What is the Guideline?;What is the conclusion?;What is the concentration?
Carcinogenicity	What is the species?;What is the Guideline?;What is the conclusion?
Photo-induced Toxicity	What is the Guideline?;What is the conclusion?
Reproductive Toxicity	What is the Guideline?;What is the species?;What is the conclusion?

Although all the models are Transformers, they also have their differences, at the architecture level or at the pre-train level, that have impact in the results. BERT is the basic model fine-tuned for QA. When released, it achieved state-of-the-art performance in many NLP tasks, including QA [10]. RoBERTa builds on BERT and modifies key hyperparameters, removing the next-sentence pre-training objective and training with much larger mini-batches and learning rates¹⁰. BioBERT is a domain-specific language representation model, pre-trained on large-scale biomedical corpora that, while having the same architecture, outperforms BERT in a variety of biomedical text mining tasks [8].

We provided the QA models with:

1. The sections of the document that contain information about each specific substance property as context.
2. The set of questions that we defined specifically for each substance property.

As a result, we expected to extract relevant information about each substance property.

4.2 Evaluation metrics

In order to achieve a quantitative evaluation of our experiments, a confusion matrix was built with the number of *True Positives* (TP), *False Positives* (FP), *False Negatives* (FN) and *True Negatives* (TN). In the context of this work, those are defined as:

- TP: There is information in the document to be extracted and the information extracted is correct;
- FP: There is no information in the document to be extracted but there is some information extracted or there is information in the document to be extracted but the information extracted is not correct;
- FN: There is information in the document to be extracted but there is no information extracted;
- TN: There is no information in the document to be extracted and there is no information extracted;

The extracted outputs were matched to ground-truth data, i.e. the extracted information was manually compared with the information present in each document tested. Using the outcomes, we calculated the precision, recall, and F1 score in order to evaluate the performance of our experiments.

¹⁰https://huggingface.co/docs/transformers/model_doc/roberta (March 2022)

4.3 Combination Process

We can use the models individually or, to take full advantage of them, we can aggregate the answers obtained from each. The combination process consists of using an answer, i.e. information extracted, if the same or similar answer was given as an output from at least two of three models, as shown in the example of Table 3. At the point of this evaluation the combination process was done manually despite the development of the process having already started using text similarity measures.

■ **Table 3** Visual example of combination process.

Original text excerpt			
Guideline: OECD TG 429 Skin Sensitization: Local Lymph Node Assay 24 th April 2002			
Species/strain: female CBA/J mice			
Group size: 4 mice per group, 20 animals per experiment, 2 independent experiments			
Batch: R0060245B 002 L 002			
Concentration: 0.1, 1 and 10 %			
Study period: 13 June - 12 September 2008			
The test item was not soluble in any of the recommended vehicles. However, a homogeneous suspension was obtained at the maximum tested concentrations of 10% and 15%, with propylene glycol, after sonication for 10 minutes. Therefore propylene glycol was selected as vehicle. On days 1, 2 and 3 of each experiment, a dose-volume of 25 μ L of the control or dosage form preparations was applied to the dorsal surface of both ears.			
On day 6 of each experiment, all animals of all groups received a single intravenous injection of 20 μ Ci of 3H-TdR.			
SCCS comment			
Based on this LLNA study in which a maximum concentration of 15% was used, A164 is considered not to have skin sensitising potential.			
	What is the Guideline?	What is the conclusion?	What is the concentration?
BERT	OECD TG 429		0. 1, 1 and 10 %
BioBERT	Skin Sensitization : Local Lymph Node Assay	Skin Sensitization : Local Lymph Node Assay	0. 1, 1 and 10 %
		not to have skin sensitising potential	25 μ L
RoBERTa	OECD TG 429	The test item was not soluble in any of the recommended vehicles	
		A164 is considered not to have skin sensitising potential	
Combo	OECD TG 429	A164 is considered not to have skin sensitising potential	0. 1, 1 and 10 %

4.4 Results and Discussion

From the 60 documents gathered, 10 were randomly selected for testing our approach. The pre-processing process worked as planned and we were able to find each section regarding each property in the documents without faults in the process. From the 10 documents, the average section size was 899 tokens, the minimum size was 27 and the maximum of 4860 tokens.

We report the performance of each QA model in Table 4, individually, and in Table 5, after the combination process. Both tables also include the micro average, where all the outcomes are taken into account, in order to deliver a fair general evaluation of model performances. For this evaluation the information extracted was not verified by the expert that currently gathers the information manually despite direct contact throughout the development.

By first analysing each QA model individually (Table 4) we were able to understand that some optimizations can be developed even though some strong results were obtained. In some cases the precision and the recall were perfect, which can be due to the disposition of the information in the document, i.e., better results can be achieved if the information is present in bullet points than if it is in the middle of the sentences. In terms of optimisation, we used the same set of questions for each model and there are some performance gains if we use each model in its strong points. For example, *BioBERT*, pre-trained in biomedical data,

■ **Table 4** Individual evaluation of QA models on SCCS documents.

	BERT			BioBERT			RoBERTa		
	F1	P	R	F1	P	R	F1	P	R
Acute Toxicity Information	0.77	0.87	0.70	0.74	0.59	1.00	1.00	1.00	1.00
Irritation Information	0.68	0.76	0.61	0.76	0.61	1.00	0.85	0.85	0.85
Skin Sensitisation Information	0.86	0.82	0.92	0.72	0.56	1.00	0.85	0.84	0.87
Mutagenicity Information	0.67	0.53	0.92	0.57	0.40	1.00	0.71	0.55	1.00
Carcinogenicity Information	0.84	0.78	0.91	0.66	0.50	1.00	0.58	0.43	0.90
Photo-induced Toxicity Information	0.44	0.40	0.50	0.58	0.41	1.00	0.54	0.37	1.00
Reproductive Toxicity Information	0.80	0.66	1.00	0.70	0.54	1.00	0.73	0.57	1.00
Repeated Dose Toxicity Information	1.00	1.00	1.00	0.80	0.66	1.00	1.00	1.00	1.00
Micro Average	0.76	0.68	0.85	0.64	0.48	1.00	0.76	0.65	0.94

■ **Table 5** Approach evaluation on SCCS documents.

	BERT + BioBERT + RoBERTa		
	F1	Precision	Recall
Acute Toxicity Information	1.00	1.00	1.00
Irritation Information	0.89	0.96	0.83
Skin Sensitisation Information	0.96	0.96	0.96
Mutagenicity Information	0.78	0.65	0.96
Carcinogenicity Information	0.84	0.80	0.88
Photo-induced Toxicity Information	0.75	0.60	1.00
Reproductive Toxicity Information	0.85	0.74	1.00
Repeated Dose Toxicity Information	1.00	1.00	1.00
Micro Average	0.87	0.82	0.93

was the best model for acquiring the names of species, but the worst for identifying guidelines. Even in terms of time, there are gains if we just use the *BioBERT* for identification of the species and nothing else.

Also regarding evaluation, the definition of each outcome, i.e. TP or FP, is subjective for some information. Some can be a quantitative value (e.g., “357 mg/kg bw/day”) or a species name (e.g., “Rat / F344 / DuCr1Crlj”), where it is easy to define if the extracted value is a TP or a FP, but sometimes, the extracted information is a short sentence (e.g., “There was no evidence of carcinogenic activity”), which is subject to human interpretation.

By analysing Table 5, we confirm that there are gains when the models are combined, both in terms of precision and recall, when compared with the individual models results. In general, and despite the limited set of questions, we can affirm our approach obtained solid results. Still, in the future, we can experiment with more and different questions, in order to achieve better performances.

Overall, we would characterize our approach and experiments as an important step into extracting information from unstructured documents. In our point of view, this means that, although the human cannot be replaced, our approach can supply them with a set of extracted information that they can: (i) accept as is, (ii) change or complement minimally or (iii) use as the starting point of a more thorough search.

5 Conclusions

In this paper we treat *Information Extraction* as a *Question Answering* problem and propose an approach that, with limited data, can be a solution for the former. To do that, we take advantage of state-of-the-art extractive QA models. We conducted experiments and analysis on one source of studies of individual chemical substances, and the results obtained are promising, although performance gains can be achieved with some optimizations, in particular, achieving the right set of questions to use with each QA model.

In the future, we will work on generalizing this approach for documents where the TOC is not available, which will enable its application to different sources of information on chemical substances (other than SCCS); on automating the combination and evaluation processes, where text similarity measures like ROUGE can be considered; and involve the expert in the final evaluation of the results. We plan to make this approach available through a REST API, which will provide an easier integration with applications like cosmetics regulatory software, and also consider the application of this approach to other domains.

References

- 1 Sally Ali, Hamdy Mousa, and M Hussien. A review of open information extraction techniques. *IJCI. International Journal of Computers and Information*, 6(1):20–28, 2019.
- 2 Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. ELECTRA: Pre-training text encoders as discriminators rather than generators. *arXiv preprint*, 2020. [arXiv:2003.10555](https://arxiv.org/abs/2003.10555).
- 3 Lei Cui, Furu Wei, and Ming Zhou. Neural Open Information Extraction. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 407–413, 2018.
- 4 A. Cvitaš. Information extraction in business intelligence systems. In *The 33rd International Convention MIPRO*, pages 1278–1282, 2010.
- 5 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol 1 (Long and Short Papers)*, pages 4171–4186. ACL, 2019.
- 6 Anthony Fader, Stephen Soderland, and Oren Etzioni. Identifying relations for open information extraction. In *Proceedings of the 2011 conference on empirical methods in natural language processing*, pages 1535–1545, 2011.
- 7 Lin Gui, Jiannan Hu, Yulan He, Ruifeng Xu, Qin Lu, and Jiachen Du. A question answering approach to emotion cause extraction. *arXiv preprint*, 2017. [arXiv:1708.05482](https://arxiv.org/abs/1708.05482).
- 8 Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020.
- 9 Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint*, 2019. [arXiv:1907.11692](https://arxiv.org/abs/1907.11692).
- 10 Minh-Tien Nguyen, Dung Tien Le, and Linh Le. Transformers-based information extraction with limited data for domain-specific business documents. *Engineering Applications of Artificial Intelligence*, 97:104100, 2021.
- 11 Minh-Tien Nguyen, Dung Tien Le, Nguyen Hong Son, Bui Cong Minh, Akira Shojiguchi, et al. Information extraction of domain-specific business documents with limited data. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.
- 12 Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of 56th Annual Meeting of the Association for Computational Linguistics (Vol 2: Short Papers)*, pages 784–789. ACL, 2018.

3:10 Question Answering For Toxicological Information Extraction

- 13 Gabriel Stanovsky, Julian Michael, Luke Zettlemoyer, and Ido Dagan. Supervised Open Information Extraction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 885–895, 2018.
- 14 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Generation of Document Type Exercises for Automated Assessment

José Paulo Leal   

CRACS – INESC-TEC, Porto, Portugal

Department of Computer Science, Faculty of Science, University of Porto, Portugal

Ricardo Queirós  

CRACS – INESC-TEC, Porto, Portugal

uniMAD, ESMAD/P.PORTO, Portugal

Marco Primo  

Faculty of Sciences, University of Porto, Portugal

Abstract

This paper describes ongoing research to develop a system to automatically generate exercises on document type validation. It aims to support multiple text-based document formalisms, currently including JSON and XML. Validation of JSON documents uses JSON Schema and validation of XML uses both XML Schema and DTD. The exercise generator receives as input a document type and produces two sets of documents: valid and invalid instances. Document types written by students must validate the former and invalidate the latter. Exercises produced by this generator can be automatically accessed in a state-of-the-art assessment system. This paper details the proposed approach and describes the design of the system currently being implemented.

2012 ACM Subject Classification Applied computing → Computer-assisted instruction; Information systems → Information storage systems; Information systems → Web data description languages; Information systems → Extensible Markup Language (XML)

Keywords and phrases exercise generation, automated assessment, document type assessment

Digital Object Identifier 10.4230/OASICS.SLATE.2022.4

Funding This paper is based on the work done within the Automatic Assessment Of Computing Exercises project supported by the European Union’s Erasmus Plus programme (agreement no. 72020-1-ES01-KA226-VET-096004).

1 Introduction

The motivation for this research comes from the JuezLTI, a project that aims at the integration of automated assessment in Learning Management Systems (LMS) using the Learning Tool Interoperability (LTI) specification. Different assessment domains are in development as part of this project, ranging from programming languages to database query languages, including languages for serialization formalisms such as XML and JSON. This paper focuses on the automated assessment of document type definitions using languages such as JSON Schema, XML Schema, and Document Type Definition (DTD).

To support a wide range of domains JuezLTI follows a simple but effective assessment strategy. Exercises are assessed using a set of test cases, each including an input and an expected output. In the case of document type definitions, the inputs are instance documents, each with an expected validation result (“output”), either valid or invalid.

One of the drawbacks of this assessment strategy is the toil of producing good exercises. Each exercise should have a solution and a comprehensive set of tests covering all corner cases. Small and incomplete test sets may accept wrong solutions and preclude the use of automated feedback. The proposed approach is to generate test sets from solutions - document type definitions in this case. For example, from a JSON schema definitions two sets



© José Paulo Leal, Ricardo Queirós, and Marco Primo;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 4; pp. 4:1–4:6

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

4:2 Generation of Document Type Exercises for Automated Assessment

are generated: one with instances, valid JSON documents according to the definition; and another with non-instances, invalid JSON documents according to the definition. Although instance generators are available for document type definition languages, the authors are not aware of non-instance generators. Moreover, both instances and non-instances must be generated for the purpose of automated assessment: with documents as small as possible, covering all corner cases, and providing explanations for what is being tested to be used in automated feedback.

The following sections provide background on this research. Section 2 introduces serialization formalisms, document type definitions, and validation exercises. Section 3 reviews the concepts of automated assessment. Section 4 describes the proposed approach. Section 5 summarizes the current results and describes future work.

2 Serialization formalisms

Text-based document formalisms such as XML and JSON play an important role in software development. They are both human and machine-readable and thus are widely used in tasks that require data serialization. These formalisms prescribe a set of well-formedness rules that documents must follow. For example, in XML documents close tags must end the previous open tag, and attribute values must be delimited by either single or double quotes; in JSON documents property name-value pairs must be separated using commas, and property names must be delimited with double-quotes. This kind of basic rule allows the creation of a wide range of documents for different domains. Nevertheless, on a certain domain, or on specific software, only very specific types of documents are allowed.

```

{
  "name": "Jill Doe",
  "height": 173,
  "hobbies": [
    "skating",
    "reading"
  ]
}

{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "height": { "type": "number" },
    "hobbies": {
      "type": "array",
      "items": { "type": "string" }
    }
  }
}
```

(a) Instance describing a person. (b) Schema to validate persons.

■ **Figure 1** Valid JSON instance according to JSON Schema.

Document type definitions describe documents beyond well-formedness rules. The most popular document type definition languages, such as JSON Schema and XML Schema, describe the structure of documents. Documents can be seen as trees and document types in these languages specify the relationships among neighbouring nodes (parent-children and siblings nodes), and the basic types of leaf nodes. For example, Figure 1a presents a JSON document with information on a person, her name, height, and a list of hobbies; and Figure 1b presents a JSON Schema document that accepts an object with properties “name” (a string), “height” (a number) and “hobbies” (an array of strings).

Document types are instrumental to validate document instances. A validator is a program that verifies if a given document is valid according to a given document type. They are typically used when documents are generated or before processing them, to check if they are according to the specification. Validators can also be used to access document types produced as exercises.

To solve an exercise on document types a student must produce a schema in a certain language such as JSON Schema. The exercise statement describes the kind of document and may give examples of a valid document, such as that on the Figure 1a. The schema produced by the student must be equivalent to a reference solution. This assessment can be easily automated using two sets of instances: one with valid documents and another with invalid ones. Using a validator and the schema produced by the student, the documents in the first set must be all valid and those on the second set must be all invalid.

This approach to schemata assessment is comparable to what the traditional black-box model used on programming exercises assessment. In this model, the student's program is executed with a set of test cases. Each test provides an input file that is fed to the program and the obtained output is compared with the test's output file. If they all coincide then the program is considered correct. Due to the complexity of programming languages, test cases can be produced automatically from the solution just for simple cases, using programming properties [2]. But this strategy can be systematically used for document schemata. This paper presents an approach to generate both instances and non-instances of a given schema. Two formalisms are supported - JSON and XML. For the former is used JSON Schema, and for the latter, both XML Schema and DTD are used.

3 Automated assessment

Automated assessment is a widely used approach in many domains, particularly in computer programming [4]. In this context, special tools called automatic judge systems are responsible for grading programming assignments, by comparing the obtained output with the expected output [3]. Most of these tools grade a submission according to a set of rules, following a black-box approach and produce an evaluation report. The validation process has two phases: static analysis, which tests the consistency of the source code; and dynamic analysis, which includes running the program with each test case loaded with the problem and comparing its output to the expected output.

In order to automate the assessment process, programming exercises are assembled in packages with their resources described specialized metadata [5]. Most of the existent metadata formats support only assessment of blank-sheet coding questions. However, the different phases of a student's learning path may demand distinct types of exercises (e.g., bug fix and block sorting) to foster new competencies such as debugging programs and understanding unknown source code or, otherwise, to break the routine and keep engagement. YAPEXIL [5] is format to describe programming exercises supporting different types of activities.

Property-based testing can also be used for assessment [2]. Instead of fixed test cases, test data are generated randomly from properties of a functions by a test script. There are several advantages in this approach: 1) it is easier to conduct more tests covering the scope of all possible inputs (thus find more mistakes), 2) in case of failure, shrinking heuristics can be used to automatically simplify failing cases.

A domain closer to this research is web services assessment. Petrova-Anonova [6] proposed an automatic generator of test data for XML Schema-based testing of Web Services. The tool automatically extracts an XML Schema from WSDL or WS-BPEL documents and

generates both correct and incorrect XML instances needed for web service interactions. This way, the tool can be used both for testing web services at the functional and robustness levels. Other contributions were made to generate XML documents with data both valid and invalid depending on the type of testing performed. Other works [7, 1] focus on the generation of random values for the invocation parameters providing automatic derivation of data instances from WSDL descriptions.

4 Test case generation

The cornerstone of the proposed approach is the generation of instances and non-instances of a given schema. These documents, the given schema and an exercise statement are then assembled in a programming exercise in the YAPEXIL format. Although this strategy is used both for JSON and XML, (non)-instance generation is based on JSON and JSON Schema. XML type definition languages are converted to JSON Schema for (non)-instance generation, and the generated JSON documents are then converted to XML.

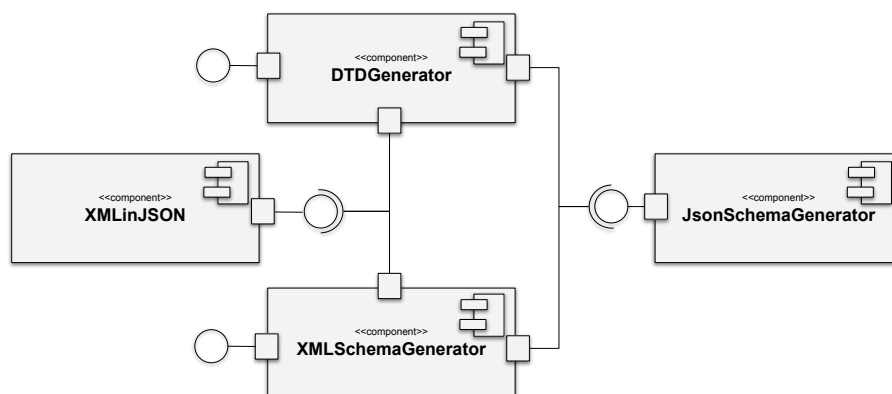


Figure 2 System architecture.

Figure 2 depicts a UML component diagram of the generator's architecture. It has four main components, three of which are generators for document type definition languages. These components have similar interfaces, with methods for generating both instances and non-instances, and validating instances against definitions. The core component is the `JsonSchemaGenerator`. The XML type definition languages generators, namely for DTD and XMLSchema, use the previous generator and convert documents between XML into JSON using the `XMLinJSON` component.

The following subsections detail the generation of JSON documents from JSON schema, performed by the `JsonSchemaGenerator`, and conversion between XML and JSON, performed by the `XMLinJSON` component.

4.1 Document generation from JSON Schema

The `JsonSchemaGenerator` component was designed having in mind the creation of test sets for exercises on document type definitions. It generates both instances and non-instances - documents that are invalid according to a given type definition. These two sets of documents are generated having in mind they will be used for automated assessment and feedback. Hence, it produces instances of different sizes, covering different kinds of possible mistakes, and also short hints that can be used as feedback.

To ensure the creation of sets of instances as large as needed, the instance generator produces either an infinite cyclical sequence of instances or an empty set. For example, a schema that validates booleans produces the infinite sequence alternating `true` and `false`; and a schema that validates no instance (such as the schema `false`) produces the empty set. With this approach, the generation can be tailored to test sets of any size. Instances are generated with increasing size, starting with the smallest possible instances. For example, in a schema of an array, the first generated instance has the smallest size allowed, zero if a minimum size is not defined. This approach improves the quality of the feedback when instances are presented to students to explain the error since smaller examples are easier to understand.

An important feature of the instance generator is the ability to generate non-instances. Two kinds of non-instances are generated: with a different type and of the same type but with invalid constraints. For example, in the case of a schema for an even number, a non-instance is either be a non-number (an instance generated by any type different from the type `number`) or an odd number (a number with an invalid constraint). The sequence of generated non-instances is varied, as well as incremental in size. In the same example, the first generated non-instance is an odd number (1) - followed by an empty list, an empty object, an empty string, a boolean, etc. The subsequent instances include larger numbers, longer strings, lists with more elements, and objects with more properties.

4.2 Conversion between JSON and XML

The `JsonSchemaGenerator` described in the previous subsection is the core of the document type generator. To handle XML document types these need to be converted into JSON Schema, and JSON (non-)instances converted to back XML. For that purpose, a representation of XML in JSON - or XIJ - was developed, implemented by the `XMLinJSON` component.

The goal of XIJ is to support all XML features in JSON, rather than to produce concise documents. It supports mixed content, i.e. text mixed with annotations, namespaces, and all types of XML nodes, such as comments and processing instructions. The module developed for XIJ includes a parser and a serializer. The overall strategy to convert generate XML instances and non-instances for definitions is as follows. Document type definitions in DTD or XML Schema are converted in JSON Schema definitions. This JSON Schema validates a document in XIJ equivalent to an XML document in the original definition. Then, the JSON Schema generator described in the previous section is used to create sets of (non)-instances, and these are finally converted to XML. However, the creation of a JSON Schema equivalent to a DTD or an XML Schema posed a few challenges.

The features of type definition languages are not equivalent. The most obvious difference relates to basic types: DTDs have only text (`#PCDATA` or `CDATA`), XML Schema has a comprehensive library, and JSON schema has JSON basic types plus integer. A not-so-obvious difference is how repetitions are handled: XML Schema provides fine-grain control over repetitions, DTDs control with regular expression operations, and JSON Schema cannot control repetitions. In fact, JSON Schema uses regular expressions but only for strings, either as values or as property names.

JSON schema had to be extended to make it compatible with DTD and XML Schema. Fortunately, the required extensions are within the scope of JSON Schema itself [8]. The creation of a library of basic types compatible with that XML Schema is straightforward by referencing an external schema document containing those definitions. The support for repetition control was obtained by introducing the `minOccurs` and `maxOccurs` properties in schema definitions, with a syntax and semantics equivalent to the attributes with the same name in XML Schema; these new properties are only effective when schemata are used to validate array items.

5 Current status and future work

The goal of project JuezLTI is to integrate several assessment domains in Learning Management Systems (LMS). One of these assessment domains supports the automated assessment of exercises on type definitions to validate JSON and XML documents. The research presented in this paper proposes an approach to automatically generate exercises from solutions; that is, from type definitions in JSON Schema, for JSON documents, or DTD and XML Schema for XML documents.

The distinctive feature of this approach is the generation of both valid and invalid instances to check students attempts. The core of the system is a document generator for JSON Schema definitions. To handle XML documents, DTDs and XSDs are converted to JSON Schema that validates documents in XIJ, a representation of XML in JSON. The generated documents in this representation are then serialized to XML.

The advantage of the proposed approach is twofold. Firstly, it saves time since it generates test cases directly from solutions. Secondly, the generated tests thoroughly cover all corner cases, unlike those produced manually, resulting in a more effective assessment.

The proposed approach is a work in progress. The core JSON generator is already developed and tested, as well as the representation of XML in JSON. The DTD converter is almost completed and the XML Schema converter is still in the design stage. In the next stage, the set of instances will be packaged as exercises using the YAPeXIL format and stored in a learning object repository.

The results obtained so far with the JSON version are very promising. The generator produces a large number of varied instances of incremental sizes, adapted to the intended use. To validate the proposed system, students will solve generated exercises as part of a Learning Management System activity.

References

- 1 Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Ws-taxi: A wsdl-based testing tool for web services. In *2009 International Conference on Software Testing Verification and Validation*, pages 326–335. IEEE, 2009.
- 2 Clara Benac Earle, Lars-Åke Fredlund, and John Hughes. Automatic grading of programming exercises using property-based testing. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 47–52, 2016.
- 3 Katerina Georgouli and Pedro Guerreiro. Incorporating an automatic judge into blended learning programming activities. In *International Conference on Web-Based Learning*, pages 81–90. Springer, 2010.
- 4 José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, 2022.
- 5 José Carlos Paiva, Ricardo Queirós, José Paulo Leal, and Jakub Swacha. Yet another programming exercises interoperability language (short paper). In *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 6 Dessislava Petrova-Antonova, Kunka Kuncheva, and Sylvia Ilieva. Automatic generation of test data for xml schema-based testing of web services. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–8. IEEE, 2015.
- 7 Harry M Sneed and Shihong Huang. The design and use of wsdl-test: a tool for testing web services. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(5):297–314, 2007.
- 8 Austin Wright, Henry Andrews, and G Dennis. Json schema: A media type for describing json documents. In *IETF, Internet-Draft draft*. IETF - Internet Engineering Task Force, 2020.

Synthetic Data Generation from JSON Schemas

Hugo André Coelho Cardoso ✉

University of Minho, Braga, Portugal

José Carlos Ramalho ✉ 

Department of Informatics, University of Minho, Braga, Portugal

Abstract

This document describes the steps taken in the development of DataGen From Schemas. This new version of DataGen is an application that makes it possible to automatically generate representative synthetic datasets from JSON and XML schemas, in order to facilitate tasks such as the thorough testing of software applications and scientific endeavors in relevant areas, namely Data Science. This paper focuses solely on the JSON Schema component of the application.

DataGen's prior version is an online open-source application that allows the quick prototyping of datasets through its own Domain Specific Language (DSL) of specification of data models. DataGen is able to parse these models and generate synthetic datasets according to the structural and semantic restrictions stipulated, automating the whole process of data generation with spontaneous values created in runtime and/or from a library of support datasets.

The objective of this new product, DataGen From Schemas, is to expand DataGen's use cases and raise the datasets specification's abstraction level, making it possible to generate synthetic datasets directly from schemas. This new platform builds upon its prior version and acts as its complement, operating jointly and sharing the same data layer, in order to assure the compatibility of both platforms and the portability of the created DSL models between them. Its purpose is to parse schema files and generate corresponding DSL models, effectively translating the JSON specification to a DataGen model, then using the original application as a middleware to generate the final datasets.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Theory of computation → Grammars and context-free languages; Information systems → Open source software

Keywords and phrases Schemas, JSON, Data Generation, Synthetic Data, DataGen, DSL, Dataset, Grammar, Randomization, Open Source, Data Science, REST API, PEG.js

Digital Object Identifier 10.4230/OASICS.SLATE.2022.5

1 Introduction

The current landscape of the technological and software development market is being increasingly occupied with scientific areas that operate with large amounts of data. A prime example is that of Data Science, which aims to apply methods of scientific analysis and algorithms to bulky datasets, in order to try to extract knowledge and conclusions from the available information, which may then be used for several other means. Another case is that of Machine Learning, which looks to use this method to equip informatic systems with the capacity of self-learning, accessing data and learning from its analysis, as to become more efficient in their function.

However, in a world where the need for bulky and representative datasets increases by each passing day, data privacy policies and other concerns related to the privacy and safety of users [5] present themselves as difficult obstacles to the growth of these sciences and to the progression of many projects in the development [4] and testing phases.

In this context, the generation of synthetic data arises as a possible solution for these problems, under the premise of being able to create realistic, large datasets artificially, from the given structural specifications. This approach was originally proposed by Rubin in



© Hugo André Coelho Cardoso and José Carlos Ramalho;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 5; pp. 5:1–5:16

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1993 [6], as an alternative that made it possible to use and share data without disrespecting the present rigorous regulations regarding the handling of sensitive data (such as the European Union's GDPR [10]), since the data in question, although very similar to corporate datasets concerning real users, would not be obtained by direct measurement. Since then, this method has been further explored and refined, being applied in the most diverse areas such as Smart Homes [2], spacial microsimulation models [8], Internet of Things (IoT) [1], Deep Learning [3] and automotive applications [9].

This paper intends to expose and document the processes of ideation and development of the application DataGen From Schemas, specifically its JSON Schema component, whose objective is to generate synthetic datasets directly from JSON schemas. This application must fulfill two requisites: on the one hand, it must be able to generate datasets of ample size and with realistic information; on the other hand, it must also be able to parse the users' schemas and obey their specifications, so that the created data is formally and structurally compliant. In order to satisfy these conditions, the platform will be built upon another existing application, DataGen, that will be contextualized in the following section.

2 DataGen

DataGen is a versatile tool that allows the quick prototyping of datasets and testing of software applications. Currently, this solution is one of the few available that offers both the complexity and the scalability necessary to generate datasets adequate for demanding tasks, such as the performance review of data APIs or complex applications, making it possible to gauge their ability to handle an appropriate volume of heterogeneous data.

The core of DataGen is its own Domain Specific Language (DSL), which was created for the purpose of specifying datasets, both at a structural level (field nesting, data structures) and a semantic level (values' data types, relations between fields). This DSL is endowed with a wide range of functionalities that allow the user to specify different types of data, local relations between fields, the usage of data from support datasets depicting different categories, the structural hierarchy of the dataset and many other relevant properties, as well as powerful mechanisms of repetition, fuzzy generation, etc. This allows for the generation of very miscellaneous and representative datasets in either JSON or XML, while dealing with intricate and demanding requirements.

It is strongly encouraged to check the published paper on DataGen's development and functionality [7] first, in order to be better contextualized in the capacities of this product and have a greater understanding of what will serve as the foundation for the new software described. For the sake of brevity, DataGen's DSL mechanisms will not be explained in this document and it will be assumed that the reader is familiar with them, going forward.

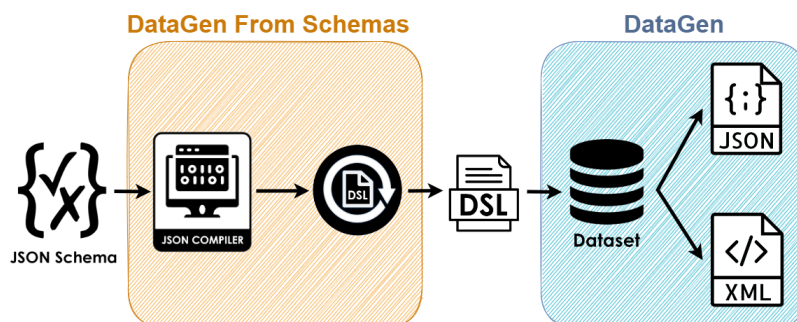
In the current scheme of software development, any enterprise that operates with JSON or XML data must have well-defined and thorough formal specifications to control and monitor the data flow in their applications, in order to assure that incoming information is well-structured and compliant with the software's requirements and outgoing data is presented as intended and does not produce unexpected behaviour. As such, it is essential to formulate schemas for semantic and structural validation, by modeling data either internally or via third parties with tools oriented to this goal, in order to restrict and enforce the content intended for each solution.

Considering the present necessity for these schemas in enterprises' business models and its recurring usage, it stands to reason that a program capable of producing large and accurate datasets from JSON/XML schemas is an incredibly valuable asset, as it provides the capacity

to quickly and effortlessly create representative datasets to test and debug platforms in development, as well as evaluate their performance under heavy stress, without having to manually concoct the information or wait for third parties to provide such resources.

As such, DataGen From Schemas emerges as a complement and an extension to its prior version, looking to generate datasets directly from JSON schemas. By doing this, the user is given the option to specify the structure of the intended dataset in JSON Schema. This arises as an alternative to the definition of the operational rules of the dataset in DataGen's native DSL, for which the user must first learn how to use it, through the lengthy documentation available. With this, the formulation of the DSL model becomes an intermediate step executed in the background and the user only has to interact with the JSON schema and the resulting dataset. However, the generated DSL model will also be made available, to enable further customizability in DataGen.

As such, this new product aims to offer a solution for a present and generalized need in the software development process and increase DataGen's use cases significantly, making the dataset generation process simpler and more accessible to any user. This new component acts as an abstraction layer over the existing application, ignoring the necessity to learn how to use the DSL from its documentation and greatly expediting the process of structural specification of the dataset. The intended workflow for this JSON Schema component is depicted in the following image:



■ **Figure 1** Intended workflow for the JSON Schema component.

The program will accept the user's input in the form of a JSON schema, which will then be parsed by a PEG.js grammar-based compiler. The parser will generate an intermediate data structure with the relevant information and a converter program will then translate it to a DSL model. Afterwards, DataGen will take care of the remaining workload, parsing the model and generating a dataset, finally converting it to either JSON or XML format, according to the user's preference.

DataGen's prior version was created by the same authors of this new tool, so there is access to the original application. The goal is to place the new compilers and translators in DataGen's own backend, in order to centralize the functionalities, avoiding additional requests between servers and possible downtime, as would be the case if DataGen From Schemas was deployed in an entirely separate web application and had to communicate with DataGen's API routes via HTTP requests to generate datasets from its models.

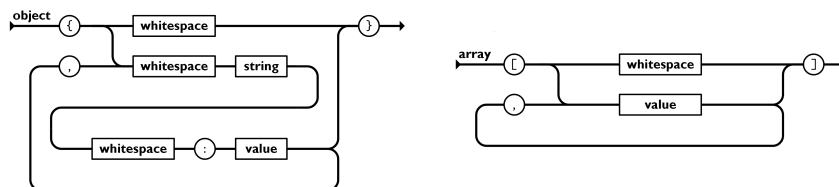
3 JSON Format

This section aims to give a bit of contextualization about the format JSON, in order to have a better grasp of what DataGen From Schemas must be able to analyse and parse, what data is relevant to be extracted, and also to understand the limits and potential of this simple, yet versatile format and why it was chosen for this application, instead of another of many popular data formats nowadays, i.e. its relevance and adequacy to this particular project. As mentioned in section 2, DataGen already has the ability of generating JSON and XML instances from DSL models, so this new product's focus is uniquely translating the data in a JSON schema to a DSL model, basically the opposite procedure.

What will be exposed about JSON also applies to JSON Schema, as the latter is a JSON vocabulary, which is to say, it is written in JSON and operates under a strict set of rules, where specific keys have precise meanings and can be used to annotate or validate JSON documents. This common syntax makes it easy to interpret the schema and validate the instance.

JavaScript Object Notation (JSON) is a lightweight data exchange format derived from the programming language Javascript, whose simplicity and readability contributed to its current vast popularity. It has ample and growing support in countless programming languages (C++, Java, JavaScript, Python, etc), which make it a strong candidate for fast and compact exchange of information between applications.

This format is utilized to represent structured information, i.e. data with rigid and well-defined configurations, where no divergencies are allowed from the structures established in the schema, namely a different data type for a certain field. This reflects in the JSON syntax, where there are only four different primitive types for values - string, number, boolean or null - and the instances are built upon solely two different data structures: objects (non-ordered collections of key-value pairs) and arrays (ordered lists of values).



■ **Figure 2** Structures of a JSON object and array.

DataGen's own DSL is built upon the JSON format, so there is very high compatibility with JSON Schema and a guarantee that all kinds of data specified in the schema can be represented in the model. In addition, DataGen also has a tool named interpolation functions, which makes it possible to randomize the value of a certain field, given its type and some other restrictions. With this, it is possible to convert a JSON schema into a corresponding non-deterministic DSL model, where the final values are not hard-coded, but decided in runtime, which may result in endless different valid datasets from a single schema.

4 Development

In order to generate DSL models from JSON schemas, DataGen From Schemas will need two different components: firstly, a grammar-based parser to analyze the schema and extract all useful information. For this, PEG.js was the technology of choice, as it was already used in DataGen and proved adequate and capable of parsing JSON-like structures. Lastly, a program

capable of translating the intermediate structure into a DSL model, for which JavaScript was used, for direct compatibility with PEG.js (that also incorporates this programming language) and JSON.

The user may input one or more schemas into the program, which is necessary to allow cross-schema referencing. In this case, the user must indicate which is the primary schema from which the dataset is to be generated. The parser then analyzes all schemas sequentially, building an intermediate structure for each of them, a procedure that will be explained in detail in this section.

4.1 Grammar

The grammar was built with a JSON grammar available in the PEG.js Github as its foundation, given that JSON Schema uses the same syntax. JSON Schema's specification is made available by drafts, which represent versions. Each time the vocabulary is majorly updated, a new draft is released, where new features can be found and existing ones altered. These drafts are not backward compatible, for example there are cases in which a certain key has entirely different semantics depending on the draft considered. As such, it seemed only logical to adapt the most recent draft to the application, which is, at the time of writing this document, JSON Schema 2020-12.

As such, the aforementioned JSON grammar was modified into a dialect (a specific version of JSON Schema) grammar for this particular draft: the set of keywords and semantics that can be used to evaluate a schema was restricted to those made available in the draft and custom vocabularies defined by the user are not accepted. With this, it was possible to implement other important features in this grammar, namely:

- **restriction of each keyword's value to its rigid lexical space** – for example, the keyword *uniqueItems*'s value must be a boolean and nothing else, while the value of the keyword *additionalProperties* may be any subschema;
- **rigorous semantic validation of the schema** – in JSON Schema, it is possible to create invalid and contradictory schemas. This may range from something as simple as a number with a *maximum* of 20 and *minimum* of 50, to more contrived cases such as establishing that a schema must be both of type boolean and string, with the keyword *allOf*. As such, a semantic validation procedure was created for this grammar, that checks for erroneous combinations of values or other incongruities in the schema's logic. This validation only does not work with references, since these are only resolved posteriorly;
- **error reports** – following the previous points, whenever the parser finds an error, be it a syntactic or semantic error, it halts the execution of the pipeline and reports it to the users, for them to correct and try again.

The other central point of the grammar is building the intermediate data structure, to where it extracts the relevant information. Before addressing this topic, it is best to categorize and explain the different kinds of JSON Schema keywords. It is encouraged to follow this section of the paper along with the official JSON Schema documentation, as it has all the keywords listed and sorted in a relevant taxonomy. For this solution, the following categorization was used:

- **type-specific keywords** – these are keywords that apply only to the data type in question. For example, numeric types have a way of specifying a numeric range, that would not be applicable to other types;
- **generic keywords** – *const*, *enum*, and *type*. The latter defines the type(s) that the schema validates, the others may be or contain values of any of those types;

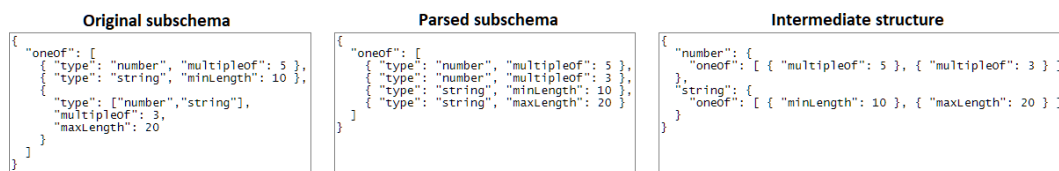
5:6 Synthetic Data Generation from JSON Schemas

- **schema composition keywords** – the purpose of these is to combine together schemas, and they correspond to well-known algebra concepts like AND, OR, XOR, and NOT;
- **keywords to apply subschemas conditionally** – based on logical conditions or the presence of certain properties in the final object;
- **structural keywords** – *\$id*, *\$anchor*, *\$ref*, and *\$defs*. These keywords do not reflect values of the schema explicitly, but are used to structure complex schemas, allowing the user to break them down into simpler, reusable subschemas, and to reference these from anywhere, to avoid duplication and write schemas that are easier to read and maintain;
- **ignored keywords** – comments and annotation/media keywords (string-encoding non-JSON data). The parser recognizes these keywords but willingly ignores them, since they have little to no use in a dataset generation context.

After thorough reflection, it was concluded that the best approach would be a type-oriented structure, where the keywords and respective values would be stored under the type of data they produce. There are multiple points in favor of this line of reasoning:

- each data type has a specific set of keywords that applies to them and only them (the aforementioned type-oriented keywords), so it is easy to separate most keywords by type;
- a single JSON schema may validate against multiple data types - the same schema can have keywords respective to arrays, numbers, and strings, for example. As such, it is useful to know, at all times, exactly what data types are produceable from the schema;
- following the previous point, a certain type may be established and then “disallowed” further into the schema, with a keyword of schema composition. As such, it would simply be removed from the intermediate structure, preventing its generation or further unnecessary parsing;
- this kind of organization makes it easy to update the structure for each new keyword parsed and facilitates the translation exercise executed later on. With this, the translation program will need only to choose a random type and parse the keywords associated with that type, ignoring all others. It is efficient and compact.

However, not all keywords are related strictly to a single type. Generic and schema composition keywords, as defined previously, plus *if/then/else* (that apply subschemas conditionally), may take values or subschemas of multiple types. In these cases, the grammar follows the ensuing method: firstly, it makes sure each of the keyword’s values has a single type. For generic keywords, this is already the case, as its values are already the final product. However, the values of the other keywords mentioned are subschemas, which may be multi-type: if so, the grammar breaks down the subschema into smaller subschemas, one per each of its types. Then, it separates the keyword’s values by type and introduces one instance of said keyword in each of its generateable data types, in the intermediate structure, along with that type’s respective values. An example of this is shown below:



■ **Figure 3** Example of parsing done on a schema composition keyword.

With this algorithm, the program is able to classify these keywords and, by extension, all JSON Schema keywords by type, which makes it possible to use the described data structure to store all relevant data, in a way designed to facilitate and make more efficient the following translator program’s routine.

In conclusion, the main objective of the grammar, and consequent parser, was to reproduce the JSON Schema syntax meticulously and collect data from any given schema to a well thought-out and efficient data structure, to set up the next phase of the process - the construction of the DSL model. Furthermore, it was also to make the solution as sturdy and fault-tolerant as possible, preventing it from trying to parse impossible schemas and crashing or producing unexpected behaviour, which in turn helps the user understand their schema better and detect unwilling errors.

4.2 Referencing

JSON Schema references can vary a lot: there are absolute and relative references, depending on if they include the schema's base URI. It is not mandatory for a schema to have an *id*, which is its URI-reference, but without one, it is unable to be referenced by other schemas, although it can still have local references. Furthermore, a subschema may be referenced either by JSON pointer, which describes a slash-separated path to traverse the keys of the objects in the document, or by anchor, using the keyword *\$anchor* to create a named anchor in the subschema to be referenced. The reader is invited to check out the official documentation on schema structuring, in order to gain a more in-depth understanding of schema identification and referencing, which is crucial for this component of the solution.

DataGen From Schemas supports all types of referencing defined in JSON Schema 2020-12 and standardizes that any schema's base URI must begin with `https://datagen.di.uminho.pt/json-schemas/` and be followed by the schema's name.

Besides the configuration exposed in the previous subsection, the intermediate data structure has two additional sections: one for storing the object pointers to all references found in the schema, and another for separately storing all subschemas with their own declared identifier. This division is useful to resolve all references later on, after the parser has finished analyzing every schema submitted by the user. The program is unable to resolve references as it is reading the schemas, since they might be pointing to schemas or subschemas that have yet to be reached. For the sake of consistency and efficiency, instead of checking if that is the case whenever a new reference is found, the parser simply caches the references and subschemas, in a way that it is ultimately able to quickly find each referenced subschema and substitute its content in the main body.

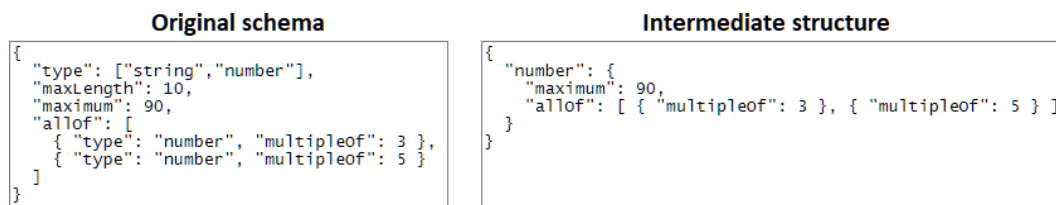
Thereby, it becomes possible to generate datasets from schemas with local and/or external references, as well as more intricate mechanisms, such as recursion and bundling.

4.3 Creating the DSL model

Once the intermediate structure of the main schema is finalized and all references resolved, it is then sent to the translator program to begin creating the correspondent DSL model. This component interprets the keywords present on the structure and generates an according DSL string, taking into account how they influence each other. For the sake of brevity, the keywords will not be explained minutely in this paper, so it is strongly recommended to follow along this subsection with the official JSON Schema documentation, as it thoroughly details the function of every keyword, illustrated with meaningful and intuitive examples.

The intermediate structure is type-oriented, meaning that each value of the schema is described by a JavaScript object that maps each of its createable data types to their respective keywords, and values are organized in a hierarquical structure. To produce the model for the whole schema, the program recursively iterates this intermediate structure, generating its values' DSL strings from the leaves to the root and joining them together.

The types present in the structure of each value already reflect the whole logic of its schema, since the parser relates the keywords and determines the generateable data types common to all of them, as described in subsection 4.1. Take the following example, illustrated below: even though the keyword *type* defines that the instance may be either a string or a number, the keyword *allof* implies that only a number is valid, since all its subschemas are only of that type. As such, the section of the intermediate data structure that describes this value will not have the string type:



■ **Figure 4** Example of parsing done on a apparent multi-type schema.

The program then randomly selects one of the generateable types and moves on to translating its keywords. The first step is to resolve any keywords of schema composition or conditional application of subschemas that there may be - these keywords are not directly translated to the DSL string, but rather parsed and its contents added to the structure.

In JSON Schema, the aforementioned keywords are not used to extend or merge schemas, in the sense of object-oriented inheritance. Instead, instances must independently validate against each of the keywords. This is understandable when validating instances against schemas, which is the purpose of JSON Schema. However, DataGen From Schemas reverses this workflow and looks to create instances from schemas, so the same logic does not apply. It is not possible to generate a different value for each of these keywords and ultimately merge the values together. This method would result in values valid against individual parts of the schema, but possibly not the whole of it.

As such, in the context of data generation, it is necessary to parse these “compound” keywords beforehand and extend the base schema with their content, obtaining a cohesive and coherent final schema, with only type-specific keywords, that incorporates the restraints specified in these keywords. Since these “compound” keywords’ values are or contain schemas, which may, in turn, have nested such keywords, the program recursively checks all subschemas for these keywords and resolves them, before using these subschemas in the extension process, so that ultimately the base schema is extended only with type-specific keywords.

4.3.1 Schema composition keywords

There are four keywords belonging to this category: *allof*, *anyOf*, and *oneOf* allow the user to define an array of subschemas, against all, one or more, or exclusively one of which the data must be valid, respectively; *not* declares that an instance must not be valid against the given subschema.

For the first three, a subset of their values’ schemas is chosen: with *allof*, all of its schemas are considered; for *anyOf* and *oneOf*, either an arbitrary number of its schemas or only one of them, respectively, are randomly selected. The base schema is then extended with these, sequentially, and the original keywords are erased from the structure.

As for *not*, the program must first “invert” this keyword’s schema, in order to obtain a complementary/opposite schema, which ensures that no value that is valid against it, also validates against the original schema. Then, the base schema is extended with this inverted schema and the keyword *not* is removed from the structure.

For this purpose, a schema inverter capable of generating complementary schemas was developed, which takes into account the meaning of each JSON Schema type-specific keyword. There is never a need to invert any other kind of keyword, since those are parsed recursively before the actual schema to which they belong (as was described in subsection 4.3), which guarantees that the schema to be inverted will only have type-specific keywords.

On the same note, the solution also incorporates a schema extender program to manually and sequentially extend a base schema with each type-specific keyword of others. For each data type, the new keyword is compared to the already existing ones and incorporated in a reasonable way - the result may be different depending on whether the base schema already has the same keyword or not, for example. This solution must handle each individual JSON Schema keyword differently, as they all have different meanings.

Due to its complexity, the functionality of these two components will not be addressed in detail in this document. The reader can find their explanation in another more substantial and exhaustive academic paper on the entirety of DataGen From Schemas that will be published in the near future.

4.3.2 Keywords that apply subschemas conditionally

These keywords are the reason for JSON Schema's dynamic semantics, i.e. their meaning can only be uncovered after the context has actually been instantiated, since these keywords establish conditions based on actual values of the instance. Take the following example:

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "country": {
      "default": "United States of America",
      "enum": ["United States of America", "Canada"]
    }
  },
  "if": {
    "properties": { "country": { "const": "United States of America" } }
  },
  "then": {
    "properties": { "postal_code": { "pattern": "[0-9]{5}(-[0-9]{4})?" } }
  },
  "else": {
    "properties": { "postal_code": { "pattern": "[A-Z][0-9][A-Z][0-9][A-Z][0-9]" } }
  }
}
```

■ **Figure 5** Example of a schema with dynamic semantics.

In this case, the schema will only know what schema to validate the property 'postal_code' with after checking the actual instance for the value of the property 'country', and never before. While with other keywords, the schema can determine a priori the structure of the instance, with these it is not possible to do so.

Once again, this approach is not reasonable for a solution such as DataGen From Schemas, since it is not viable to generate an intermediate value from the rest of the schema and only then exert these keywords, which may imply large changes to the remaining schema - at least with *if*, *then*, and *else*. Since the keywords *dependentRequired* and *dependentSchemas* are specific to the object data type, it is possible to incorporate them into the translation procedure of object schemas, as will be detailed ahead in subsection 4.4.3.

The solution found for the keywords *if*, *then*, and *else* was to determine their outcome probabilistically - unless the *if* schema is explicitly true or false, in which case it is deterministic whether the instance should be validated with either *then* or *else*, respectively. Otherwise, the program determines the veracity of the condition based on a probability, customizable

by the user (by default, a 50/50% chance) - if true, the base schema is extended with the *if* and *then* schemas, otherwise it is extended with a complementary schema of *if* (produced with the aforementioned schema inverter) and the schema of *else*. This way, it is possible to produce a coherent, simpler schema that incorporates the logic of these conditional keywords and to generate the final instance in a single iteration.

4.4 Translating the intermediate structure

Finally, once the intermediate structure of the selected data type is finalized and possesses only type-oriented keywords and/or *const* and *enum*, the program is able to generate a DSL string that translates the logic of the original schema. In this subsection, it will be carefully detailed the method behind translating each type of value.

The first step is to check if the intended value must belong to a fixed set of values, i.e. the usage of any of the keywords *const* or *enum*, which, at this point, already reflect the absence of any values unallowed with *not*. If any of these are present, the remaining keywords are ignored and the DSL string is generated from these alone. In case of both keywords, *const* takes precedence over *enum*, as it defines that the value is constant and immutable. The output DSL string produced from these keywords is a random choice from all of their respective values (typically, *const* will map to a single value, but if the user composes multiple instances of this keyword in the same schema, it will be treated as an alternative between several).

If no fixed set of values is defined, the solution goes on to actually translate the type-oriented keywords. Data types *null* and *boolean* are basic, since they possess no specific keywords. These and the previously addressed keywords are translated as follows:

<pre>{ "enum": ["red", "amber", "green", null, 42] } { "const": "United States of America" } { "type": "null" } { "type": "boolean" }</pre>	<pre>gen => { return gen.random(...["red","amber","green",null,42]) } gen => { return gen.random(...["United States of America"]) } null '{{boolean()}}'</pre>
---	--

■ **Figure 6** Translation of the keywords *enum* and *const*.

4.4.1 String type

The string type only has four different keywords: *pattern*, *format*, and *minLength/maxLength*. Only in this case, it was decided that there would be an order of precedence to these keywords since, realistically speaking, they would very rarely be used together (except for both length keywords): for example, it does not make much sense to define a string value according to a regular expression or format and then further constrain its length, as the former keywords already establish a very rigid template.

As such, if the schema has the keyword *pattern*, that will be the one to be translated, followed by *format* and, finally, the length keywords. While *pattern* and length keywords translate directly to one of DataGen's interpolation functions, *format* is more contrived, since there is a need to program a different DSL string for each format, to generate an according value, some of which map directly to existing interpolation functions and others that do not. Below are presented some examples of the models generated from this data type's keywords:

<pre>{ "type": "string", "pattern": "^\\([0-9]{3}\\)?[0-9]{3}-[0-9]{4}\$" }</pre>	<pre>'{{pattern("^\\([0-9]{3}\\)?[0-9]{3}-[0-9]{4}\$")}}'</pre>
<pre>{ "type": "string", "minLength": 2, "maxLength": 3 }</pre>	<pre>'{{stringOfSize(2, 3)}}'</pre>
<pre>{ "type": "string", "pattern": "time" }</pre>	<pre>'{{time("hh:mm:ss", 24, false, "00:00:00", "23:59:59")}}'</pre>

■ **Figure 7** Translation of string type schemas.

4.4.2 Numeric types

As for numeric types, there are five applicable keywords: *multipleOf*, *minimum*, *exclusiveMinimum*, *maximum*, and *exclusiveMaximum*. The constraints on numeric types can get a lot more complicated if the user specifies, via schema composition keywords, that the value must be a multiple of several numbers simultaneously and/or not a multiple of one or more values. Let's first consider simpler cases where the *not* keyword is not used.

If the instance must be a multiple of one or more values, the program starts by calculating the least common multiple (LCM) of all these numbers. This way, it is possible to consider a single value for the multiplicity of the instance, since the LCM and its multiples are the easiest way to obtain any number simultaneously multiple of the original values. If the type in question is an integer, this is also taken into account before determining the LCM, by considering that the instance must also be a multiple of 1.

Next, the range keywords are evaluated, if present. The program determines the biggest and smallest integers that it is possible to multiply by the LCM (or 1, if no *multipleOf* constraint exists), in order to obtain values that belong to the intended range. This is doable by rounding down the result of the division of the upper bound by the LCM and rounding up the result of the division of the lower bound by the LCM, respectively. If the schema only has either an upper or lower range boundary, the other one is offset by 100 units to provide comfortable margin for generateable values. At this point, there are three different possible outcomes, detailed next and further illustrated in the image below:

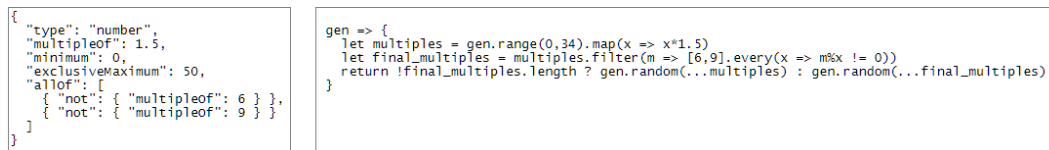
- if only the type is specified and nothing else, the program simply generates a DSL string for a random integer or float, accordingly;
- if no range keys are used (the only type-specific is *multipleOf*), the DSL's interpolation function with the same name is used to generate a multiple of the LCM;
- if both range and multiplicity constraints are present, the value is calculated by randomly choosing an integer between the predetermined bounds and multiplying it by the LCM.

<pre>{ "type": "number" }</pre>	<pre>'{{float(-1000,1000)}}'</pre>
<pre>{ "type": "number", "multipleOf": 7 }</pre>	<pre>'{{multipleOf(7)}}'</pre>
<pre>{ "type": "integer", "multipleOf": 2.5, "minimum": 0, "exclusiveMaximum": 20 }</pre>	<pre>gen => { return gen.integer(0,3) * 5 }</pre>

■ **Figure 8** Translation of numeric type schemas.

5:12 Synthetic Data Generation from JSON Schemas

However, the negation of the keyword *multipleOf* makes these use cases a lot more contrived, since there is the necessity to further restrict the set of produceable values. As such, the DSL string produced in such occasion is different for all the alternatives above. Only one variant will be explained, since the same logic applies across all others:



```
{
  "type": "number",
  "multipleOf": 1.5,
  "minimum": 0,
  "exclusiveMaximum": 50,
  "allOf": [
    { "not": { "multipleOf": 6 } },
    { "not": { "multipleOf": 9 } }
  ]
}
```

```
gen => {
  let multiples = gen.range(0,34).map(x => x*1.5)
  let final_multiples = multiples.filter(m => [6,9].every(x => m%x != 0))
  return !final_multiples.length ? gen.random(...multiples) : gen.random(...final_multiples)
}
```

■ **Figure 9** Translation of a complex numeric type schema.

As seen in the preceding image, the schema has impositions on the range of the value, as well as what it must and must not be a multiple of. This translates to the DSL via a JavaScript anonym function, where DataGen first calculates all valid multiples in the designated range and stores them in an array, then removing all elements that are multiples of unallowed numbers. The final value is selected randomly from the alternatives with the interpolation function *random*, however the program firstly does a safety check to ensure that the final array is not empty: if that is not the case, then its impossible to produce a value that obeys all restrictions specified in the schema, so it simply selects a regular multiple in the range from the first array.

4.4.3 Object type

The set of type-specific keywords for objects is *properties* and *patternProperties*, to specify property schemas according to their key, *additionalProperties* and *unevaluatedProperties*, for unspecified properties, *required*, to make properties mandatory, *propertyNames*, to validate the properties' keys, and finally size keywords (*minProperties* and *maxProperties*).

DataGen From Schemas also treats *dependentRequired* (used to require properties based on the presence of others) and *dependentSchemas* (to apply subschemas if certain properties exist) as object-specific keywords - for every new property selected for the final instance, the solution also checks these keywords for dependencies. In case of the former, if any properties are dependent on the newest property, these are also translated and added to the object. As for the latter, if there is a subschema dependent on the latest property, it is parsed and used to extend the current object schema. This verification is executed for every property produced along the pipeline, required or not.

For this type, the instance's model is firstly delineated in an intermediate object, mapping each key to the DSL string of their respective value, in order to manage more easily all the different properties that may be produced. Only after determining all properties does the program produce a DSL string for the whole object, from this second intermediate structure.

The first operation executed by the program is determining a random size for the final object, taking into account all relevant factors such as *minProperties* and *maxProperties*, *required* properties and the permission or not of unspecified properties (*additionalProperties*, *unevaluatedProperties*). The calculated size will always necessarily allow room for at least the required properties, and if only the object type is specified and no other keywords are used, the program will generate an object with between 0 and 3 properties, where both the key and value are random.

Then, DataGen From Schemas iterates the *required* properties and generates an according DSL string for each of their value's schema, storing the pair in the intermediate object structure. Once the required properties have been produced, the solution executes the following pipeline sequentially, until it reaches the designated size for the final instance:

- iterates the non-required properties sequentially, producing the DSL strings of their respective values and storing the pairs in the intermediate object;
- iterates the value of the keyword *patternProperties* - for each pair, there is a probability (customizable by the user) to produce, at most, one according instance property, where the name of the property is obtained through a regular expression value generator;
- if additional properties are not allowed or not explicitly mentioned in the schema, the intermediate structure is considered finalized with the properties it currently has. Do note the mention of explicitly allowing additional properties - if neither of the keywords *additionalProperties* and *unevaluatedProperties* are specified, then the user most likely wants an instance with only the properties covered by the keywords *properties* and *patternProperties* - as such, it would not make much sense to generate other random properties, just because it is not explicitly disallowed;
- if the schema specifies additional properties, *additionalProperties* has precedence over *unevaluatedProperties*, so if both keywords are specified, *additionalProperties*'s schema prevails, else it is the only used keyword's. The program translates this schema into a DSL string and generates random names for the properties keys, either according to the *propertyNames* schema, if present, or by generating small chunks of *lorem ipsum*.

Having finalized the intermediate object with each property's DSL string, these properties are all joined in a string encased by curly braces, in the syntax of a regular JavaScript object that DataGen is able to parse and then generate the according dataset.

```

{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" },
    "type": { "enum": ["residential", "business"] }
  },
  "required": ["street_address", "city", "state"],
  "propertyNames": { "pattern": "inhabitant[1-3]" },
  "additionalProperties": { "enum": ["John", "James", "Mary"] },
  "minProperties": 5,
  "maxProperties": 7
}

```

```

{
  street_address: '{{stringofsize(0, 100)}}',
  city: '{{stringofsize(0, 100)}}',
  state: '{{stringofsize(0, 100)}}',
  type: gen => { return gen.random(...["residential", "business"]) },
  inhabitant1: gen => { return gen.random(...["John", "James", "Mary"]) },
  inhabitant3: gen => { return gen.random(...["John", "James", "Mary"]) }
}

```

■ **Figure 10** Translation of an object type schema.

4.4.4 Array type

There are several keywords for this data type: *items* and *prefixItems*, the main way to evaluate items, *additionalItems* and *unevaluatedItems*, for others that do not validate against former keywords, *contains*, *minContains*, and *maxContains*, for inclusion of items with certain schemas, and finally length keywords (*minItems*, *maxItems*) and *uniqueItems*, to determine the uniqueness of items in the instance.

For this data type, the program creates an intermediate array structure firstly, in order to maleably plot the intended array, and only at the end converts it to a DSL string. The solution keeps track, at all times, of the allowed number of items, which can be established with *minItems* and *maxItems*. If the maximum amount of items is ever reached (and also never before the minimum is attained), the program stops the execution of the pipeline that will be described ahead and produces the DSL string from the intermediate array as is.

The first step of the algorithm is to check for prefixed items in the schema: if any were specified, the program sequentially generates their respective values' DSL strings and pushes these to the intermediate array, in order.

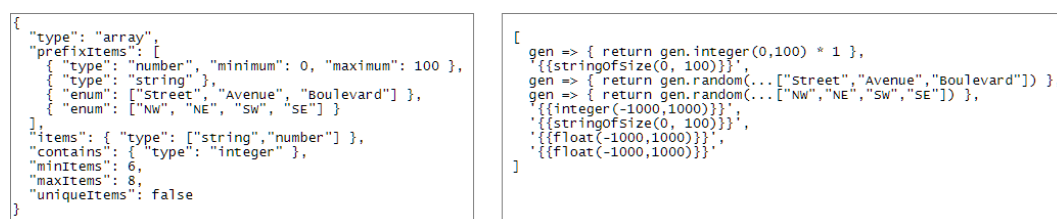
5:14 Synthetic Data Generation from JSON Schemas

Then, DataGen From Schemas parses the inclusion keywords. In the workflow of this solution, these keywords effectively have a dynamic semantic, since it is impossible to validate the array for the inclusion of elements with the specified schema before generating it, even in the DSL model. As such, a compromise was needed to adapt these keywords: after parsing the prefixed items, if more items are allowed in the array, then the solution will push the necessary amount of new elements with the structure of the *contains* schema (one if *minContains* and *maxContains* are not used, otherwise a random number in the indicated range). Note that if the intent of the user is for the values validated against the inclusion keys to be some of the prefixed items, then the program cannot guarantee this.

Once all the “hard-coded” elements are in the intermediate array, the program then checks for other items. The keyword *additionalItems* has precedence over *unevaluatedItems*, so if additional items are allowed and both keywords are specified, *additionalItems*’s schema is considered, otherwise it is the only specified keyword’s. If the user also disallowed any instance of the keyword *contains*, their schemas are inverted (using the schema inverter mentioned in 4.3.1) and used to extend the schema for additional items, in order to guarantee that all additional items wholly conform to the user’s configuration. This final schema is used to sequentially generate a random amount of additional items’ DSL strings, within the intended range for the array, which are appended to the intermediate structure.

Finally, DataGen From Schemas checks if all these items must be unique or not (*uniqueItems*). This keyword, as is the case with the inclusion keys, demands a compromise and an intelligent workaround, since it is impossible to check for the items’ uniqueness before generating them, and is the reason that warrants the usage of an intermediate array to store the DSL strings of each element, instead of simply building a single string and appending each item to it. As such, depending on the value of this keyword, two different types of DSL strings for the final array may be created:

- **not unique items** – if either the keyword is not specified or its value is false, then the procedure is to simply create a string of an array, where its elements are the DSL strings stored in the intermediate array structure maintained along this pipeline. DataGen can interpret this syntax and generate all elements according to their respective model;



```
{
  "type": "array",
  "prefixItems": [
    { "type": "number", "minimum": 0, "maximum": 100 },
    { "type": "string" },
    { "enum": ["Street", "Avenue", "Boulevard"] },
    { "enum": ["Nw", "NE", "Sw", "SE"] }
  ],
  "items": { "type": ["string", "number"] },
  "contains": { "type": "integer" },
  "minItems": 6,
  "maxItems": 8,
  "uniqueItems": false
}
```

```
[
  gen => { return gen.integer(0,100) * 1 },
  '{{stringOfSize(0, 100)}}',
  gen => { return gen.random(...["Street", "Avenue", "Boulevard"]) },
  gen => { return gen.random(...["Nw", "NE", "Sw", "SE"]) },
  '{{integer(-1000,1000)}}',
  '{{stringOfSize(0, 100)}}',
  '{{float(-1000,1000)}}',
  '{{float(-1000,1000)}}',
]
```

■ **Figure 11** Translation of an array type schema with non-unique items.

- **unique items** – in this case, a DataGen anonym function is used to implement an algorithm that attempts to generate an array where all elements are different. As can be observed below, the procedure is the following: whenever DataGen creates the next item, it checks if the array already contains the new value. If not, it pushes the element to the array and moves on to the next one. Else, it generates the same item again (which can produce a different result, since the DSL strings accommodate margin for randomness), for a maximum of 10 tries per item. In case none of them produces a new value, the value of the latest try is pushed to the array anyway, to prevent crashing the program, and the resulting array ends up not obeying the *uniqueItems* restriction.


```

{
  "type": "array",
  "prefixItems": [
    { "type": "number", "minimum": 0, "maximum": 100 },
    { "type": "string", "enum": ["Street", "Avenue", "Boulevard"] },
    { "enum": ["NW", "NE", "SW", "SE"] }
  ],
  "items": { "type": ["string", "number"] },
  "contains": { "type": "integer" },
  "minItems": 6,
  "maxItems": 8,
  "uniqueItems": true
}

```

```

gen => {
  let arr = []
  for (let i = 0; i < 7; i++) {
    for (let j = 0; j < 10; j++) {
      let newItem
      if (i==0) newItem = gen.integer(0,100) * 1
      if (i==1) newItem = gen.stringOfSize(0, 100)
      if (i==2) newItem = gen.random(... ["Street", "Avenue", "Boulevard"])
      if (i==3) newItem = gen.random(... ["NW", "NE", "SW", "SE"])
      if (i==4) newItem = gen.integer(-1000,1000)
      if (i==5) newItem = gen.float(-1000,1000)
      if (i==6) newItem = gen.stringOfSize(0, 100)
      if (!arr.includes(newItem) || j==9) {arr.push(newItem); break}
    }
  }
  return arr
}

```

■ **Figure 12** Translation of an array type schema with unique items.

As a by-product of this algorithm, another compromise arises: DataGen From Schemas can only attempt to generate arrays with unique items if all items are elementary, i.e. neither objects nor arrays. This is the case because DSL strings for complex values are bigger, more convoluted and impossible to incorporate in a DataGen function syntax, in the same way a basic interpolation or anonym function can. To summarize, DataGen From Schemas is not the best tool to adapt the keyword *uniqueItems* in specific, due to its ill-suitability to the workflow and DataGen’s DSL, however it is still possible to satisfy some use cases.

5 Conclusion

The main contributions of this paper are the design of a synthetic data generator from JSON schemas and its implementation, which culminate in DataGen From Schemas - a quick and sturdy solution built upon DataGen that effectively greatly expands the base application’s functionality and makes it more accessible to any user, by allowing their input to be in a vastly popular and utilized vocabulary, JSON Schema, and automating the creation of DataGen’s DSL model.

With this software, it becomes possible to very quickly specify a dataset and generate several different instances of it, given that the program randomizes its values with each attempt, as well as further customizing its requirements directly in DataGen, via the provided intermediate DSL model, with support from custom datasets and other tools provided by the base application, which is not possible to do in JSON Schema’s syntax.

DataGen From Schemas has been experimented with real-life cases and convoluted schemas that use mechanisms such as bundling or recursion, and proves to be an adequate solution, capable of interpreting the schemas it is given and generating representative datasets accordingly. The product will soon be put in a production environment and made available for the general public, as a free and open-source application, after an arduous yet successful development phase.

References

- 1 Jason W. Anderson, K. E. Kennedy, Linh B. Ngo, Andre Luckow, and Amy W. Apon. Synthetic data generation for the internet of things. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 171–176, 2014. doi:10.1109/BigData.2014.7004228.
- 2 Jessamyn Dahmen and Diane Cook. Synsys: A synthetic data generation system for healthcare applications. *Sensors*, 19(5), 2019. doi:10.3390/s19051181.
- 3 Hadi Keivan Ekbatani, Oriol Pujol, and Santi Segui. Synthetic data generation for deep learning in counting pedestrians. In *ICPRAM*, pages 318–323, 2017.

5:16 Synthetic Data Generation from JSON Schemas

- 4 GAO. Artificial intelligence in health care: Benefits and challenges of machine learning in drug development (staa)-policy briefs & reports-epta network. In *GAO Technology Assessment: Artificial Intelligence in Health Care: Benefits and Challenges of Machine Learning in Drug Development*, 2020. Accessed: 2021-04-25. URL: <https://eptanetwork.org/database/policy-briefs-reports/1898-artificial-intelligence-in-health-care-benefits-and-challenges-of-machine-learning-in-drug-development-staa>.
- 5 Menno Mostert, Annelien Bredenoord, Monique Biesart, and Johannes Delden. Big data in medical research and eu data protection law: challenges to the consent or anonymise approach. *European Journal of Human Genetics*, 24:1096–1096, July 2016. doi:10.1038/ejhg.2016.71.
- 6 Donald B. Rubin. Statistical disclosure limitation. In *Journal of Official Statistics*, pages 461–468, 1993.
- 7 Filipa Alves dos Santos, Hugo André Coelho Cardoso, João da Cunha e Costa, Válder Ferreira Picas Carvalho, and José Carlos Ramalho. DataGen: JSON/XML Dataset Generator. In Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira, editors, *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, volume 94 of *Open Access Series in Informatics (OASIS)*, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIS.SLATE.2021.6.
- 8 Dianna M Smith, Graham P Clarke, and Kirk Harland. Improving the synthetic data generation process in spatial microsimulation models. *Environment and Planning A: Economy and Space*, 41(5):1251–1268, 2009. doi:10.1068/a4147.
- 9 Apostolia Tsirikoglou, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *arXiv preprint*, 2017. arXiv:1710.06270.
- 10 P. Voigt and A. von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer International Publishing, 2017. URL: <https://books.google.pt/books?id=cWAwDwAAQBAJ>.

Extending PyJL – Transpiling Python Libraries to Julia

Miguel Marcelino  

INESC-ID/Instituto Superior Técnico University of Lisbon, Portugal

António Menezes Leitão  

INESC-ID/Instituto Superior Técnico University of Lisbon, Portugal

Abstract

Many high-level programming languages have emerged in recent years. Julia is one of these languages, claiming to offer the speed of C, the macro capabilities of Lisp, and the user-friendliness of Python. Julia’s syntax is one of its major strengths, making it ideal for scientific and numerical computing. Furthermore, Julia’s high-performance on modern hardware makes it an appealing alternative to Python. However, Python has a considerable advantage over Julia: its extensive library set.

There have been efforts to make Python libraries available to Julia either through Foreign Function Interfaces (FFI’s), or through manual translation, but both have their tradeoffs: FFI’s do not take advantage of Julia’s performance, as they call Python’s Virtual Machine, and manual translation is demanding and time-consuming.

To address these issues and bridge the gap between the two languages, we propose PyJL, a transpilation tool that converts Python source-code to human-readable Julia source-code. Although the development of PyJL is still at an early stage, our preliminary results reveal that the generated code follows the pragmatics of Julia and is capable of high performance.

2012 ACM Subject Classification Software and its engineering → Source code generation; General and reference → Cross-computing tools and techniques

Keywords and phrases Source-to-Source Compiler, Automatic Transpilation, Library Translation, Python, Julia

Digital Object Identifier 10.4230/OASICS.SLATE.2022.6

Supplementary Material *Software (Source Code)*: https://github.com/MiguelMarcelino/pyjl_translations

Funding This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) (references PTDC/ART-DAQ/31061/2017 and UIDB/50021/2020).

1 Introduction

Transpilation and compilation share identical mechanisms for processing source code, but they have two distinct goals: compilers convert an input language to a lower-level language, such as machine code, while transpilers translate it to another language with a similar level of abstraction.

Transpilers were initially developed to convert lower-level source code. The first transpiler, CONV86 [6], was developed in 1978 by Intel to translate assembly source code from the 8080/8085 to the 8086 processor, providing compatibility between an 8-bit and a 16-bit processor. Nowadays, as most modern programming languages are high-level languages, it makes sense to develop transpilers that operate at this level. As an example, consider Babel.js [1], a transpiler that converts newer versions of ECMAScript, such as ES6, to ES5.

Modern programming languages became more dependent on the quality and quantity of available libraries, which highly influence its adoption. One solution to this problem is to translate libraries from more established languages to newer and/or less popular ones [15]. However, manual translation requires an extensive amount of time and resources. On the other hand, using a transpiler to automate this process can significantly speed up the translation of libraries between languages.



© Miguel Marcelino and António Menezes Leitão;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 6; pp. 6:1–6:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Automatic translation is challenging, as different languages offer distinct syntactic and semantic constructs. Additionally, if transpilers work with static source code information, dynamic languages become considerably difficult to translate. All of these challenges influence the translation’s quality.

There have been previous efforts regarding the transpilation of Python source code to Julia. For instance, Py2JL[13] aims at translating Python source code to human-readable Julia source code. Similarly to the work presented in this paper, it uses rule-based transpilation to generate Julia source code. However, it can only translate simple Python code excerpts. Thus, having a tool that can operate on larger code samples could benefit the community.

This paper extends our previous work on translating Python libraries to Julia [17]. PyJL is a rule-based transpilation tool that translates Python source code into human-readable Julia source code. Our previous evaluation of PyJL concluded that it is possible to translate Python source code to Julia with minimal programmer intervention. Still, the transpiler required more improvements regarding the generation of pragmatic code. Furthermore, with improved translation coverage of Python’s standard library, we can now perform a more in-depth analysis of the transpilers abilities.

2 Related Work

With the rise of many new high-level programming languages, translating between them has become an increasingly important topic. Some programming languages are inclusively using transpilers to maximize compatibility with existing languages. This is the case of TypeScript [11], which transpiles to JavaScript and offers an improved syntax and additional functionalities.

The topic of library translation was also discussed when developing LinJ [15], which transpiles Common Lisp source code to Java. This work describes how the syntactic and semantic incompatibilities of these two languages make in very hard to automatically translate between them. This becomes increasingly more difficult if we want to preserve the pragmatics of the target language.

Regarding the transpilers that use Python as a source language, PyRS is a transpiler that translates Python to Rust and is also part of Py2Many [21]. It currently requires manual intervention in some cases to generate running Rust source code. The Fortran-Python transpiler [4] uses Python’s type hints to translate Legacy Fortran to Python and vice versa, while producing human-readable code. It does not intend to entirely automate the translation process, instead requiring manual intervention in certain cases. There are also transpilers, such as Prometeo [23], which transpile Python source code to high-performance C source code. However, the generated code is not human-readable, an important aspect of this work.

Transpilers that translate from Julia are less common, as the language is also very recent. We previously mentioned Py2JL[13], which also transpiles Python to human-readable Julia source code. Other transpilation approaches are not concerned with human-readability.

More generic tools include TXL [5], which was designed to restructure and modify source code. Other notable tools include DMS [2], which targets automatic management and improvement of source code in large software solutions.

3 Automatic Translation

Automatically generating code that preserves the pragmatics of the target language is challenging, requiring the transpiler to at least use proper language constructs and suitable code formatting with appropriate use of indentation rules. In practice, the translation process

requires a careful analysis of syntactic and semantic incompatibilities that can be difficult to overcome. For instance, in the case of Python and Julia, the semantics of language constructs often require specific mappings. Python's operators must be carefully mapped to Julia, as they use overloading and apply different operations depending on their operator types.

Additionally, there is the problem of memory management, in which a transpiler must account for memory allocations and remove unused objects in memory, which may require the use of a garbage collection mechanism. This is important for transpilers such as `ts2c` [18], which convert JavaScript and TypeScript to human-readable C source code.

Lastly, one should also consider the difference between dynamically and statically typed languages. In a statically typed language, such as Java, all types have to be defined at compile time, while in a dynamically typed language, types are determined at runtime. Translating a dynamically typed language to a statically typed one requires a type inference mechanism. However, the reliability of static type-inference mechanisms largely depends on the type information available at compile time. Therefore, a transpiler might require restrictions regarding which types must be available at compile time. Moreover, some languages, such as Julia, also benefit from type annotations to provide better performance. That is the case of Julia's Arrays [3], where memory allocation can largely be improved if type annotations are used, allowing for contiguous element allocations and less boxing/unboxing operations.

4 Translating Python to Julia

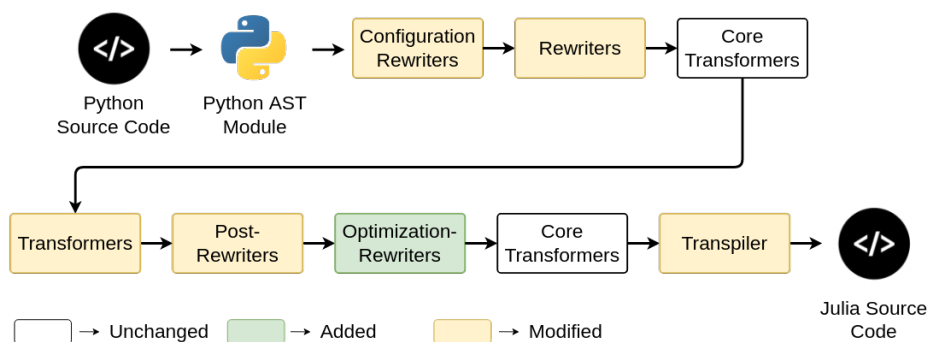
In the previous section, we highlighted several aspects of automatic translation that make the conversion of syntax, semantics, and generation of pragmatic code more difficult. However, Python and Julia have similar levels of abstraction, which might indicate that the translation of Python libraries to Julia can occur with minimal programmer intervention. In this section, we will briefly introduce both languages.

Python was introduced in 1991, and has gained popularity in recent years due to its extensive library set and its use in Data Science. It is worth noting that there are several implementations of Python. Its reference implementation is CPython, which was created by Guido van Rossum in 1989 and is written in the C programming language. Two common alternatives to CPython are Jython [9] and IronPython [10], both developed by Jim Hugunin. The first compiles the input Python source code to JVM bytecode for the Java platform, and the latter compiles it to IL bytecode for the .NET platform. However, these implementations currently lack support for Python's latest version.¹ Furthermore, there is also PyPy [20], an implementation of Python using a Just-In-Time compiler written in RPython.

For the purpose of this research, we will use CPython, as it is the reference implementation. However, CPython suffers from slow performance due to Python's implicit dynamism. A common problem of CPython is known as the two language problem, which occurs when the prototype language differs from the main implementation language. Programmers who require high-performance typically convert the kernel parts of their programs to C, using Python only as a prototyping language.

On the other hand, Julia promises to solve the two language problem and is proving to be a high performance alternative to Python. It is one of the few languages that belongs to the Petaflop club, along with C, C# and Fortran. However, its library set is still reduced, particularly when compared to more established languages, such as Python. We plan to address this issue by speeding up the library development in Julia.

¹ As of April 2022, IronPython supports version 2.7.11 (version 3.4 is still an Alpha release) and Jython supports version 2.7.2



■ **Figure 1** PyJL Architecture.

5 PyJL

PyJL² is part of the Py2Many [21] transpiler, which is a rule-based transpilation tool that offers a generic framework to transpile Python to many C-like programming languages, such as Rust, Go, and C++. PyJL builds upon that framework to translate Python source code to Julia. Figure 1 shows the architecture of the PyJL transpiler. The changed and added phases show up with different colours for distinction.

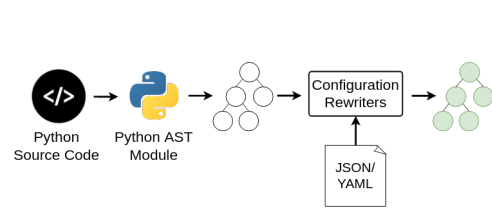
The transpiler first parses the input Python source code using Python’s `ast` module,³ which generates an Abstract Syntax Tree (AST). It then uses several intermediate transformation phases to generate the equivalent Julia source code. A brief description of each phase follows:

1. *Configuration Rewriters* support configuration files in JSON and YAML format that specify AST modifications.
2. *Rewriters* can be both language-specific or -independent. A rewriter changes the structure of nodes in the AST to match equivalent nodes in the target language.
3. *Core Transformers* are language-independent transformers that add relevant information to nodes in the AST. An example is to add scope context to the AST’s nodes, allowing for node searches.
4. *Transformers* are language-specific and add information to nodes in the form of attributes. An example would be to add type annotations for type inference.
5. *Post-Rewriters* are rewriters that have dependencies on some previous phase. Their functionality is identical to the *Rewriters* phase.
6. *Optimization Rewriters* are rewriters that optimize a small set of the generated source code. This is similar to peephole optimization, commonly used in compilers.
7. *Transpiler* translates language syntax and semantics, and converts the AST to a string representation in the target language.

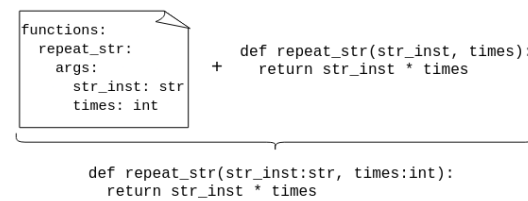
Notice from figure 1 that the *Core Transformers* phase executes at two stages. The first makes core information available to the *Transformers*, *Post-Rewriters* and *Optimization-Rewriters* stages. The second ensures that core Py2Many transformations are not overwritten, making them available in the *Transpiler* phase.

² PyJL Development Repository: <https://github.com/MiguelMarcelino/py2many> Retrieved April 24th, 2022

³ Abstract Syntax Tree – Python 3.10: <https://docs.python.org/3/library/ast.html> (Retrieved on January 27th, 2022)



■ **Figure 2** Annotation pipeline.



■ **Figure 3** Annotation Example.

There are scenarios where the transpiler provides more than one translation method. In these scenarios, the programmer can decide which alternative to use by using one of three methods: (1) manually annotating the Python source code, (2) using JSON or YAML annotation files, supported by the *Configuration Rewriters* phase, or (3) using flags to make global changes to source code generation. These approaches provide the flexibility to cover a broader range of translation scenarios, allowing the programmers to selectively apply annotations if necessary.

To demonstrate the use of the annotation mechanism, we show the processing pipeline in more detail in figure 2. It parses the provided YAML/JSON files and adds the information to the AST. The current supported features are adding type hints or decorators to function definitions. An example of code annotations can be seen in figure 3. In this example, we can use the annotations provided from the YAML file and merge them with the `repeat_str` function. This is equivalent to manually annotating the code.

We will now describe the changes to the transpiler. Section 5 describes the most notable improvements. In section 6, we compare the performance of several translation scenarios and provide an optimization use-case, where we analyse the steps required by the programmer to improve the performance of the generated code. Lastly, section 7 describes some limitations of the transpiler and translation methodology applied.

5.1 Scoping Rules

The first mismatch between Python and Julia is related to the scoping rules, which define the behaviour of assigning names to values and solve possible conflict scenarios. Both Python and Julia use lexical scoping, which determines, at compile-time, the section in the source code where a name is bound to a value. This section analyses the different scoping rules and how they affect the translation process.

In Python, scopes are defined according to the *LEGB* rule [16, Ch. 16], which stands for Local, Enclosing, Global, and Built-in scopes. Local scopes define the scope of a Python function or lambda expression. Enclosing scopes define the outer scope of a nested scope. The Global scope is the top scope of a Python module. Lastly, the Built-in scope contains automatically loaded special keywords, such as built-in functions, exceptions, etc. In Python, this rule is used when searching for an unqualified name. The search for a name reference starts on the Local scope, following the *LEGB* order, and stops at the first encounter of that name.

On the other hand, in Julia, scopes can either be global or local. Furthermore, Julia's local scopes are divided into hard and soft scopes [14, Ch. 10]. To explain this concept, let us consider that a variable named `a` is defined in the global scope: if the enclosing scope is a hard scope and if there is an assignment to a new local variable `a`, then a new local variable will be created and will shadow the global variable; if all the enclosing scopes are soft scopes, the behaviour changes when used in non-interactive or interactive (REPL) contexts.

■ Listing 1 Python Mandelbrot.

```

1 def mandelbrot(limit, c) -> int:
2     z = 0 + 0j
3     for i in range(limit + 1):
4         if abs(z) > 2:
5             return i
6         z = z * z + c
7     return i + 1

```

■ Listing 2 Julia Mandelbrot.

```

1 function mandelbrot(limit, c)::Int
2     z = 0 + 0im
3     i = 0
4     for _i = 0:limit
5         i = _i
6         if abs(z) > 2
7             return i
8         end
9         z = z * z + c
10    end
11    return i + 1
12 end

```

In non-interactive contexts, if an assignment to a new local variable `a` occurs, it will shadow the global variable similarly to the hard scope, the only difference is that it emits a warning when shadowing occurs. In interactive contexts, the global variable is always assigned. If there is no global variable with the same name, then a new local variable will be created both in hard and soft scopes.

Global scopes include `modules` and `baremodules`. Local soft scopes include `structs`, `for`, `while` and `try`, while local hard scopes include `macros`, `functions`, `do blocks`, `let blocks`, `comprehensions`, and `generators`. Furthermore, Julia’s constructs are only allowed in certain scopes. These scoping rules impose some limitations in the translation process.

For instance, one of Julia’s scoping restrictions is that `structs` can only be defined in the global scope. As an example, the transpiler generates `structs` to translate Python classes. However, classes in Python can be defined in local scopes. Automatically changing the scope of classes could potentially result in name clashes. In addition, this might not match the programmer’s intent of the code. Therefore, we include a `remove_nested` Python decorator that the programmer can use to annotate the classes that should be moved to the global scope.

This problem also affects the `resumable` macro, which is used by the transpiler to simulate Python’s generators in Julia. This macro defines a Finite State Machine to simulate Python’s generator functions. To save the machine’s state, it creates a `struct`, restricting its use to the global scope. To account for these cases, we have added an optional argument field `remove_nested` to the `resumable` decorator, which has a similar functionality as the previously defined decorator.

Another mismatch occurs with control flow operators. Despite their syntactic similarities in Python and Julia, they have considerable differences in their scoping rules. As an example, translating Python’s loops to Julia could potentially result in errors if loop target variables are used outside its body. To detect these cases, the transpiler analyses the enclosing scope to find any assignments that have the same variable name as any of the loop target variables. This enforces the concept of for-loop scopes during transpilation and can then be used in two different ways:

1. Have the transpiler emit a warning message when loop target variables are used outside the loop’s scope.
2. Create a new variable in the enclosing scope and update it in every iteration of the loop.

To demonstrate a use-case where this occurs, consider the `mandelbrot` function shown in listing 1 that tests if a complex number `c` belongs to the Mandelbrot set by computing the number of iterations required (up to a given `limit`) to get a value greater than 2.

■ **Listing 3** Python Augmented Assignments. ■ **Listing 4** Augmented Assignment Expansion.

```

1 x = [1,2]
2 y = x
3 x += [3,4]
4 x[1:2] *= 2
5 y[1:2] += [1]

```

```

x += [3, 4]
↓
x = x + [3, 4]
↓
x = append!(x, [3, 4])

```

Notice how the loop variable `i` is used outside the scope of the loop. This is valid in Python, as the loop does not define its own scope, but cannot be translated directly to Julia. By using code analysis and applying the second translation method, the transpiler generates the code shown in listing 2, which now works in Julia with the intended result.

5.2 Augmented Assignments

Augmented assignments combine binary operations with assignment statements. These are supported by both Python and Julia, but given Python's use of operator overloading, not all augmented assignments can be translated directly to Julia.

As an example, consider the code excerpt in listing 3. The first two assignments are equivalent in both languages. However, executing the third statement in Julia does not yield the same results, as it performs an element-wise addition instead of the concatenation that is done in Python. To solve this problem, the transpiler expands augmented assignments into assignments and binary operations and then transpiles the result.

The conversion of this statement can be seen in listing 4. Notice how the function `append!` was used, which performs an in-place concatenation of elements to the list `x`, matching the behaviour of Python's augmented assignments [22, Ch. 7].

Another mismatch occurs when translating *slices* from Python to Julia. Lines 4 and 5 of listing 3 represent that scenario. To translate these augmented assignments into Julia, one can use the `splice!` function, which replaces or inserts new elements in a given list. The translated Julia source code for the previous example is the following:

```

splice!(x, 2:2, repeat(x[2:2], 2))
splice!(y, 3:2, [1])

```

Notice that the second call to `splice!` uses the form `n:n-1` for the second argument, which inserts a new element in the list [14, Ch. 42].

5.3 Subscripts

In Python, a subscript is used to represent indexing and slicing on sequences and key lookups on mapping types. Translating subscripts to Julia becomes a challenge when trying to generate pragmatic code. Furthermore, there are several cases that require special handling, which will be discussed in this section.

Indexing is used to look up a particular position in a sequence, i.e., tuples, lists, strings, etc. The main difference between indexing in Python and Julia, is that Python uses 0-based indexing while Julia uses 1-based indexing. Indexing can be performed using integer literals or generic expressions. Integer literals can be incremented to match Julia's 1-based indexing, but non-literal expressions require adding the literal 1, which can reduce the readability of the code. However, because most indexing is performed in loops, we can optimize the entire scenario instead.

■ Listing 5 Julia Combination Sort.

```

1 def comb_sort(
2     seq: List[int]) -> List[int]:
3     gap = len(seq)
4     swap = True
5     while gap > 1 or swap:
6         gap = max(1, floor(gap / 1.25))
7         swap = False
8         for i in range(len(seq) - gap):
9             if seq[i] > seq[i + gap]:
10                seq[i], seq[i + gap] = \
11                    seq[i + gap], seq[i]
12                swap = True
13     return seq

```

■ Listing 6 Julia Combination Sort.

```

1 function comb_sort(
2     seq::Vector{Int})::Vector{Int}
3     gap = length(seq)
4     swap = true
5     while gap > 1 || swap
6         gap = max(1, floor(Int, gap / 1.25))
7         swap = false
8         for i = 0:length(seq) - gap - 1
9             if seq[i + 1] > seq[i + gap + 1]
10                seq[i + 1], seq[i + gap + 1] =
11                    (seq[i + gap + 1], seq[i + 1])
12                swap = true
13            end
14        end
15    end
16    return seq
17 end

```

For instance, consider the implementation of the combination sort algorithm in listing 5. The simplest translation, as shown in listing 6, is to preserve the ranges and adjust the indexing operation. As an alternative, we provide two additional optimization methods that the programmer can use:

1. Determine if operations in loops are only performed on sequences, and increment loop ranges.
2. Use the `OffsetArrays` package [12] to define custom index ranges for sequences.

The first approach requires analysing the source code to verify if the loop ranges can be optimized. The transpiler validates the applicability of this optimization by analysing if all nodes in the loop’s body using its target variables are `Subscript` nodes. To demonstrate this translation method, we used the transpiler to translate the Python combination sort implementation by changing the loop ranges instead of the indexes. The transpilation result, shown in listing 7, respects the pragmatics of Julia, being much closer to what a Julia programmer would write.

The second translation method, which uses `OffsetArrays`, allows defining arrays with the same index ranges as Python. The code generated by the transpiler is available in listing 8. The call to `OffsetArray` creates a wrapper around the array `seq` and decreases its indexing value by 1, which is equivalent to performing 0-based indexing. In this case, the transpiler used a *let*-block, to restrict the wrapping of the input vector `seq` to the block’s scope.

Both alternatives have tradeoffs. The first is arguably more pragmatic in this particular example, but changes the algorithm’s implementation to use 1-based indexing, which might not be desired for all scenarios. The second preserves the program’s original indexing, but can be a less pragmatical solution in some cases. The programmer can select his preferred method, as demonstrated in the beginning of section 5.

Another scenario is the use of subscripts with slices. In Julia, slices are called `UnitRanges` and are very similar to those of Python. One notable exception is that in Python, slices have an inclusive beginning and exclusive end, whereas in Julia, both are inclusive, requiring the transpiler to adjust the ranges. Furthermore, Python allows slices that do not explicitly define a beginning or end index, whereas Julia’s `UnitRanges` require all values to be specified. An example of this can be found below, where the Python code excerpt on the left can be translated to the corresponding Julia code on the right:

■ **Listing 7** Julia Optimized Indexing.

```

1 function comb_sort(
2     seq::Vector{Int}::Vector{Int}
3     gap = length(seq)
4     swap = true
5     while gap > 1 || swap
6         gap = max(1, floor(Int, gap / 1.25))
7         swap = false
8         for i = 1:length(seq) - gap
9             if seq[i] > seq[i + gap]
10                seq[i], seq[i + gap] =
11                    (seq[i + gap], seq[i])
12                swap = true
13            end
14        end
15    end
16    return seq
17 end

```

```

l = [1,2,3,4]
a = l[1:]
b = l[:-1]

```

■ **Listing 8** Julia Offset Arrays.

```

1 function comb_sort(
2     seq::Vector{Int64}::Vector{Int64}
3     let seq = OffsetArray(seq, -1)
4     gap = length(seq)
5     swap = true
6     while gap > 1 || swap
7         gap = max(1, floor(Int, gap / 1.25)
8             )
9         swap = false
10        for i = 0:length(seq) - gap - 1
11            if seq[i] > seq[i + gap]
12                seq[i], seq[i + gap] =
13                    (seq[i + gap], seq[i])
14            end
15        end
16    end
17    return seq
18 end
19 end

```

```

l = [1,2,3,4]
a = l[2:end]
b = l[begin:end-1]

```

Notice that the last line covers the translation of negative indexing. Currently, PyJL only supports negative indexing if the index is a literal.

Lastly, subscripts can be used for key lookups, where a common scenario is a dictionary lookup. In practice, distinguishing subscripts that use indexing from ones that use keys depends on the container's type. The transpiler currently relies on the type-inference mechanism to infer the container types and transpile the corresponding subscripts to Julia. The inference mechanism is described in section 5.5.

5.4 Functions

Both in Python and Julia, functions are first-class values. Most notably, they can be returned, passed as arguments, and assigned to variables. Some functions defined in Python can be mapped to equivalent functions in Julia. However, even if their behaviour is equivalent, they may differ in terms of the argument order or even argument count. For instance, consider the Python code excerpt on the left and its corresponding translation to Julia found on the right:

```

write = sys.stdout.buffer.write    write = x -> Base.write(stdout, x)
write(b"test")                    write(b"test")

```

If no argument is provided, the Python built-in function `write` will output the content to `stdout`. However, in Julia, the equivalent function requires an explicit parameter indicating where to write the contents to. The transpiler translates such cases using lambda expressions to automatically input the default parameters.

Notice how in the translated code above, the assignment to variable `write` overwrites any future calls to Julia's built-in `write` function within the scope where it is defined. To prevent this issue, every time an assignment name clashes with one of Julia's built-in functions,

we translate function references using Julia’s `module.name` notation. This can be seen in the example above, where `sys.stdout.buffer.write` is translated to `Base.write` in Julia, mitigating the name clashes.

5.5 Type Inference

Having a type-inference mechanism is crucial to transpile Python source code to Julia, as the translation outcome might depend on the type information available. Py2Many already offers a type-inference mechanism, which we extended with new inference rules. This section briefly describes the current inference mechanism.

The inference mechanism in Py2Many is implemented using a define-use set. This mechanism recursively walks the AST and aggregates type information from node assignments for each scope. The defined scopes are more extensive than Python’s scopes, to cover the scoping rules of all supported languages. The current scopes include `modules`, `functions`, `classes`, `for` and `while` loops, `if`-statements, and `with`-statements.

We extended the current inference mechanism with a more extensive rule-set. The changes include adding more information to binary operations, as their translation to Julia is largely dependent on the types of their operands. This involves adding type annotations to the left and right operands, to support more complex operations.

Similarly to MyPy, PyJL also requires programmers to annotate function definitions. PyJL uses these definitions to mitigate type instability and to translate operations that depend on the operand types.

Lastly, PyJL strictly enforces static typing using Python’s type hints. Therefore, any user-annotated variables are only allowed to have one type within their scope. As an example, consider the following assignment operations in Python:

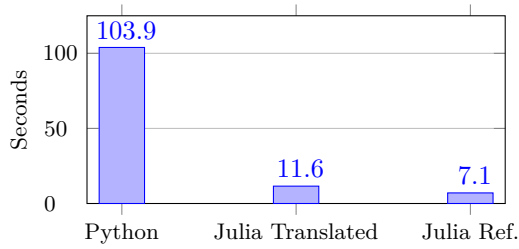
```
1: List[str] = ["a", "c", "g", "t"]
...
l = "acgt"
```

In this case, the transpiler will reject the second assignment to variable `l`, as the type of its value does not match the previous type annotation. Alternatively, if the value of the second assignment is another variable, the transpiler will search for the variable’s type. If the type was provided by a type hint and does not match the previous annotation, the transpiler will reject the assignment.

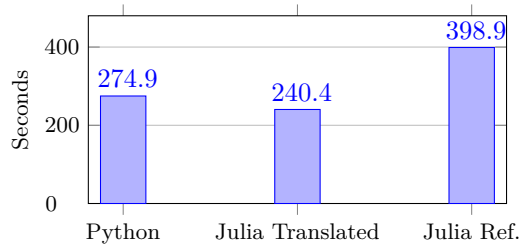
6 Performance

To evaluate the performance of the generated code, we chose three benchmarks: (1) the binary trees benchmark, which tests the garbage collection mechanisms of each language by allocating short-lived trees and traversing them, (2) the sieve of Eratosthenes, which evaluates the performance of Julia’s iterators and compares the performance of the generated code to a high performance NumPy implementation, and (3) the fasta benchmark, which generates random DNA sequences using a Linear Congruential Generator (LCG), testing the performance of IO operations and generator functions.

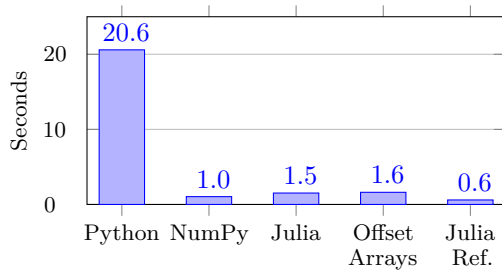
For all benchmarks, we have chosen a reference version, representing the best-case single-threaded implementation. Julia’s reference version for the fasta benchmark was simplified by removing the use of threads. We measured the results using the `bencher` benchmarking tool [8]. The results were measured on a machine with an Intel(R) Core(TM) i7 4790K @4.4GHz



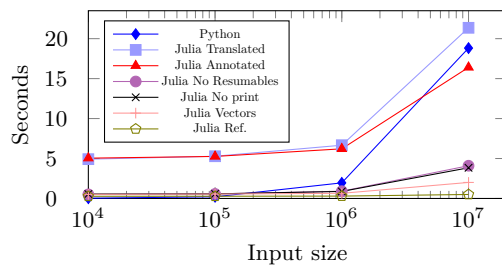
■ **Figure 4** Binary Trees Benchmark.



■ **Figure 5** Binary Trees Memory Allocation.



■ **Figure 6** Sieve Benchmark.



■ **Figure 7** Fasta Benchmark.

with 16GB of RAM under Linux. We used an input of 21 for the binary trees benchmark, and 100,000,000 for the sieve benchmark. The fasta benchmark was tested with varying input sizes. The results of the translation are publicly available.⁴

From figure 4, we observe that translating Python’s implementation of the binary trees benchmark to Julia yields a $9\times$ faster execution time. Both the Python reference version and the generated Julia code use a similar amount of memory, as can be observed in figure 5, measured at 275MB and 240MB respectively. The reference Julia version manages a $1.64\times$ faster execution time compared to the translated version, although it also uses $1.66\times$ more memory.

Regarding the sieve benchmark, figure 6 shows the obtained results, where the generated Julia source code is $13.5\times$ faster than the Python implementation. We also translated this implementation using OffsetArrays, mentioned in section 5.3, where the results only differ by 6% from the translated code. Nonetheless, Python also allows programmers to use NumPy [19], a scientific computing library implemented in C, to speedup code execution. The NumPy implementation is $1.46\times$ faster than the generated code. Julia’s reference implementation still manages a faster execution time, but it uses sophisticated techniques, such as loop unrolling, and exploits cache allocation.

Lastly, the results of the fasta benchmark can be seen in figure 7. As the Python implementation uses generator functions, we have chosen to translate this benchmark using the Resumables package, which simulates the use of Python’s generator functions in Julia. Similarly to Python, it uses a finite state machine to save the generator’s state. However, despite its similarities, the translated version is slightly slower across all input ranges. This is especially noticeable with smaller inputs from 10^4 to 10^6 . We now describe possible steps a programmer can follow to improve the performance of the translated code.

⁴ PyJL Benchmarks Repository: https://github.com/MiguelMarcelino/pyjl_benchmarks (Retrieved June 6th, 2022)

6:12 Extending PyJL – Transpiling Python Libraries to Julia

To discover what is causing the slowdown, the first step is to look for unannotated code. After a closer inspection of the fasta benchmark, we determined that there was one instance where this occurred. The translated Julia function can be found below, for which we added the type annotations manually:

```
1 function makeCumulative(table)
2     P::Vector{Float64} = []
3     C::Vector{String} = []
4     prob = 0.0
5     for (char, p) in table
6         prob += p
7         P = append!(P, [prob])
8         C = append!(C, [char])
9     end
10    return (P, C)
11 end
```

This function calculates cumulative probabilities from the predicted probabilities of choosing each nucleotide in a DNA sequence, represented by the `table` argument. A list `P` is used to store the cumulative probabilities, and a list `C` is used to store the nucleotides. Both lists are translated into generic arrays, which have large overheads in Julia, resulting in less efficient code due to excessive boxing and unboxing. The annotations on lines 2 and 3 solve this problem. As seen in figure 7, the changes are more noticeable with larger inputs of 10^7 , where the execution time is $1.3\times$ faster than Python.

Next, we measured the overhead of using resumables. In this case, as it is only required to save a seed value, we can use a memory reference holding that value, similar to what the Julia reference version does. Removing the use of resumables resulted in much faster execution times across all input ranges. This lead us to conclude that the slowdown observed with the smaller inputs was likely caused by the initial operations required to setup the finite state machine.

Another aspect that can be improved is IO performance, as the fasta benchmark outputs large nucleotide sequences to the standard output. Python's `print` function was translated by the transpiler to Julia's equivalent `println` function, which has notable overheads when called repeatedly, as it does not buffer the data, issuing more calls to operating system functions that require expensive context-switching operations. This can be improved by using Julia's `write` function, which, similarly to Python's `println` function, buffers the data and reduces the number of operating system calls. This improvement is more noticeable with the larger input size of 10^7 , resulting in a $1.06\times$ faster execution time. Despite being a small improvement, this becomes more noticeable with input values larger than 10^7 .

Finally, we can use Julia's more efficient data types and convert Python's strings into vectors holding `UInt8` values. This not only makes the implementation more efficient, but also reduces the amount of memory required to allocate the nucleotide sequences. These changes are more noticeable with input sizes of 10^6 and 10^7 , resulting in a further $1.36\times$ and $1.91\times$ improvement, respectively.

Julia's reference implementation is still faster than our optimized fasta implementation, as it stores and retrieves the random nucleotide sequences in a more efficient way. Instead of the `makeCumulative` function used by Python, Julia builds a lookup table that holds all the nucleotides in their respective positions, given the calculated cumulative probabilities. It then relies on indexing to retrieve the nucleotides. In contrast, Python uses the `bisect_right` function to locate the appropriate nucleotides, which has more overheads and results in slower execution times when translated to Julia.

7 Future Work

The changes made to PyJL bring it closer to our goal of automatically translating Python libraries to Julia. However, there are still limitations, which we discuss in this section.

The transpiler currently analyses modules independently, without considering module-level dependencies. Py2Many already uses a topological sort algorithm to sort all modules according to their import dependencies, but still requires a more sophisticated mechanism to analyse the program's context as a whole.

Another aspect that has to be addressed is the handling of exceptions. As an example, consider Python's `ValueError` exception, which is raised when an argument has the correct type but an incorrect value. Two Python calls that throw this exception are `math.sqrt(-1)` and `float("-")`. In Julia, the first call throws a `DomainError` exception, but the second call throws an `ArgumentError` exception. Therefore, an exception in Python does not have a deterministic corresponding exception in Julia. Furthermore, some Python exceptions, such as the `ZeroDivisionError` that is raised in Python when the quotient of a division operation is zero, are not considered as exceptions in Julia. Julia instead returns `Inf`, which represents infinity. Given that the transpiler performs static analysis, it will not be possible to detect such situations in advance, and it is not pragmatic to generate code that does that at runtime. Such cases remain a topic for future work.

Regarding Py2Many's type-inference mechanism, it is currently rather conservative when annotating generic containers, which largely impact the performance of the generated Julia source code. This is due to its limitations regarding intra-procedural analysis, which considers the whole program's context. An external type inference mechanism supporting intra-procedural analysis, such as `pytype` [7], could potentially increase the available type information at transpilation time.

Lastly, the readability of the generated code is an aspect that requires a more extensive evaluation. We are currently in the process of preparing user tests, but these still require a thorough analysis and remain a topic for future studying.

8 Conclusions

This work aims to automate the translation of Python libraries to Julia to increase Julia's library set. The generated code should conform to Julia's pragmatics, allowing Julia programmers to further maintain it.

In this paper, we presented recent additions to our previous work on translating Python libraries to human-readable and modifiable Julia source code. In particular, we further automated the translation of source code by covering a more extensive subset of Python. The readability and pragmatics of the generated code were also improved using code analysis, making it harder to distinguish from human-written code.

Automatically converting Python source code is challenging due to the semantic differences between Python and Julia. Python's dynamic typing further makes this a difficult process, as types are not available at compile time. The improved inference mechanism now covers a broader range of scenarios, but this is still a limitation when automatically converting Python source code. Requiring type-hints in function definitions mitigates most of these limitations.

As demonstrated by our performance results, the PyJL transpiler requires little programmer intervention to generate high-performance Julia source code. Furthermore, as the generated source code is human-readable, programmers can further optimize and maintain it.

References

- 1 Babel. Babel, September 2012. Retrieved April 7th, 2022 from: <https://github.com/babel/babel>.
- 2 Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Edinburgh International Conference Centre, Scotland, UK, 2004. Semantic Designs.
- 3 Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al. Julia Language Documentation - Performance Tips, 2014. Chapter 3, p.279-299.
- 4 Mateusz Bysiek, Aleksandr Drozd, and Satoshi Matsuoka. Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance through Transpilation and Type Hints. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 9–18, 2016. doi:10.1109/PyHPC.2016.006.
- 5 James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13), pages 827–837, 2018. doi:10.1016/s0950-5849(02)00104-0.
- 6 Intel Corporation. MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users, 1979.
- 7 Google. Pytype: A static type analyzer for Python code, March 2015. [Online. Retrieved February 25th, 2022 from: <https://github.com/google/pytype>].
- 8 Isaac Gouy. The Computer Language Benchmarks Game, 2007. Retrieved April 16th, 2022 from: <https://salsa.debian.org/benchmarksgame-team/benchmarksgame>.
- 9 Jim Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th international Python conference*, volume 9, pages 2–18, Reston, VA, 1997. Citeseer.
- 10 Jim Hugunin. IronPython Home Page, 2013. [Online. Retrieved April 6th, 2022 from: <https://ironpython.net/>].
- 11 Philip Japikse, Kevin Grossnicklaus, and Ben Dewey. *Introduction to TypeScript*. Apress, 2017. Chapter 7. doi:10.1007/978-1-4842-2478-6.
- 12 JuliaArrays. Offsetarrays.jl, January 2014. Retrieved April 11th, 2022 from: <https://github.com/JuliaArrays/OffsetArrays.jl>.
- 13 JuliaCN. Py2Jl, 2018. [Online. Retrieved 19th April, 2022 from: <https://github.com/JuliaCN/Py2Jl.jl>].
- 14 JuliaLang. The julia language - 1.7.3, May 2022.
- 15 António Menezes Leitão. The next 700 programming libraries. In *Proceedings of the 2007 International Lisp Conference, ILC '07*, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1622123.1622147.
- 16 Mark Lutz. *Learning Python - 3rd Edition*. O'Reilly Media, Inc., 2007.
- 17 Miguel Marcelino and António Menezes Leitão. Transpiling Python to Julia using PyJL. In *Proceedings of the 2022 European Lisp Conference*. Zenodo, March 2022. doi:10.5281/zenodo.6332890.
- 18 Andrei Markeev. ts2c: JavaScript/TypeScript to C Transpiler, 2016. [Online. Retrieved March 30th, 2022 from: <https://github.com/andrei-markeev/ts2c>].
- 19 Travis Oliphant. NumPy, 2009. [Online. Retrieved May 30th, 2022 from: <https://numpy.org>].
- 20 Armin Rigo. Pypy, 2007. [Online. Retrieved April 17th, 2022 from: <https://www.pypy.org/>].
- 21 Arun Sharma, Lukas Martinelli, Julian Konchunas, and John Vandenberg. Py2many: Python to many CLike languages transpiler, 2015. Retrieved April 22nd, 2022 from: <https://github.com/adsharma/py2many>.
- 22 Guido van Rossum and Python Development Team. *The Python Language Reference - Release 3.9.7*, August 2021. [Online. Retrieved April 9th, 2022 from: <https://docs.python.org/release/3.9.7/>].
- 23 Andrea Zanelli, Tommaso Sartor, Peter Bowyer, and Dominik Moritz. Prometeo - An experimental Python-to-C transpiler, 2017. [Online. Retrieved 19th October, 2022 from: <https://github.com/zanellia/prometeo>].

EWVM, a Web Virtual Machine to Support Code Generation in Compiler Courses

Sofia Teixeira¹ ✉

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

José Carlos Ramalho ✉ 🏠 

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Pedro Rangel Henriques ✉ 🏠 

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Abstract

This paper describes a project which goal is to analyze and model a complete Virtual stack Machine (VM) environment and build a Web application with a graphical interface to deploy an environment to compile and execute VM programs. The new tool offers two main features: assembles and reports errors in programs written in the assembly language of the Virtual Machine; and animates the execution of the compiled code, displaying the internal state of the VM and providing an interface to control the execution step-by-step. In the paper, after discussing related concepts and works, a proposal to build such a tool, so far called EWVM, will be presented along the architecture drawn. A prototype will be shown, and its impact as an educational tool is argued.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Virtual machines

Keywords and phrases Virtual Machine, Stack Machine, Assembler, Debugger, Compiler, Code Generation

Digital Object Identifier 10.4230/OASICS.SLATE.2022.7

Supplementary Material *Software (Web Application)*: <https://ewvm.ep1.di.uminho.pt/>

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

A Virtual Machine (VM) is a software layer over the actual machine. A VM creates a virtualized environment that mimics a computer system. It behaves like a completely separate computer, running independently from the actual computer (host) and other virtual machines (guests). Each VM runs its own operating system and functions, also having virtual hardware. Allowing multiple operative systems to run in a single physical computer is a very relevant feature of a virtual machine [10].

Many virtualization solutions have been implemented for various intents. In cloud environments, VMs are fundamental since they provide virtual application resources to multiple users at once. Virtual Machines are also demanded for security purposes, since they allow the user to take risks that could otherwise harm the computer, making them great for things like malware analysis [12]. The most obvious use for VMs is to run incompatible software or to test new operating systems. Some VMs, like Java Virtual Machine [16], were created to guarantee software compatibility, allowing different equipment or operating systems to have the same compiler. That feature is responsible for the well-known portability of Java applications, and it is also present in other popular languages as Perl or Python that use their own virtual machines, Parrot [7] and Python Virtual Machine [13].

¹ corresponding author



A growing use for Virtual Machines is in the field of education, since these allow for a more specialized environment. There has been a focus on virtual machines being used with pedagogical purpose [2], such as teaching in different domains, namely cybersecurity [3], system software development [6] and others [9].

At the moment, a Virtual Machine, named VM, developed by Jean-Christophe Filliatre at LRI/Université de Paris Saclay for his Compiler courses [5], is being used at our University in the context of Language Processing courses to teach students about compilers and code generation. As a Learning Resource, VM is valuable due to its logic and clear working principle and to the simplicity of its reduced instruction set. However VM runtime environment, that interpreters the Assembly programs and allows their execution step-by-step for debugging purposes, is a stand alone application that is not easy to install and work with. This fact complicates its usage during the Compiler course and motivates us to start a new project aimed at developing a new, more user-friendly resource. As such, VM is the foundation of the new Web-based Virtual Machine that is introduced and explained along this paper.

This paper is organized in five sections. This section presents the project context, motivation and paper focus. Section 2 introduces the two main types of virtual machines and discusses their features. Moreover, in Section 2 there is also a description of the basic ideas underlying the VM in use at moment. Section 3 presents the proposed architecture to develop the new system, EWVM, that extends VM and makes it available in a Web browser. Section 4 defines the Assembly language recognized by EWVM and presents the prototype already available for tests and improvements. Section 5 closes the paper with some conclusions and future work.

2 Virtual Machines and related work

Virtual Machines bring tremendous progress and practicability to the everyday life of programmers. Whereas in previous times, one would have multiple physical machines for different requirements and needs, in these days, one can implement all those distinct machines with different features in a single computer.

The current technologies are constructed by one of the two prevalent architectures up-to-date. Effectively, present-day Virtual Machines are either register based or stack based.

A **Stack Virtual Machine** uses a stack data structure as memory. It works by pushing and popping values to or from the top of the stack. To execute these movements, the machine relies on the stack pointer (SP), which points to the top of the stack at all times. Any instruction that requires operand values to be performed pops them from the top of the working stack. Then the operation is executed outside of the stack and its result is pushed back to the stack [15].

A **Register Virtual Machine** stores its data in registers of the CPU. The instructions must indicate the registers used to store the operands required in each case. It works faster than a stack based virtual machine within the instruction dispatch loop in the sense that it avoids the overhead of all the popping and pushing that would be otherwise necessary [15]. However that more efficient approach requires from the compiler an extra task that is complex: the registers management.

Which type of Virtual Machine is better seems to be a debatable subject. Interestingly, there is a research paper where the authors rewrite the Java Virtual Machine as a register based Virtual Machine [14] and accomplish a superior performance.

However, Stack Virtual Machines are more attractive as a Learning Resource considering that the operands location is implicit in the stack pointer, unlike register based machines where it needs to be specified [4]. Consequently, stack based machines also tend to run a

simpler and easier to comprehend instruction set, a valuable feature in terms of execution and application [11], though it must be noted that this feature often results in more extensive programs.

Although Register Virtual Machines seem to be able to attain a higher performance, this does not seem to be easily achievable. Their more complex and extended instructions require a broader instruction set, raising the interpretation difficulty. For a further discussion about how to deal with register based instructions, see the Dragon book [1].

2.1 Current Virtual Machine

As stated, a Virtual Machine is currently being used in the Language Processing Course at our university to teach students about compilers and code generation. The so-called VM is a Stack-Machine implemented in C that runs programs in VM-Assembly language.

In addition to the Virtual Machine not being easily accessible, there are also some difficulties in executing it to its full capacity and features, notably the fact that not all computers are compatible with all needed software.

2.1.1 Architecture

VM is composed of several structures that allow it to function correctly and in an organized manner, storing information in a systematically arranged configuration. As one can see in Figure 1, there are three main memory blocks, two of them being stacks, and two additional heaps in the machine. The two essential constituents that are needed to run the most basic programs are the **Operand Stack** and the **Code Zone**.

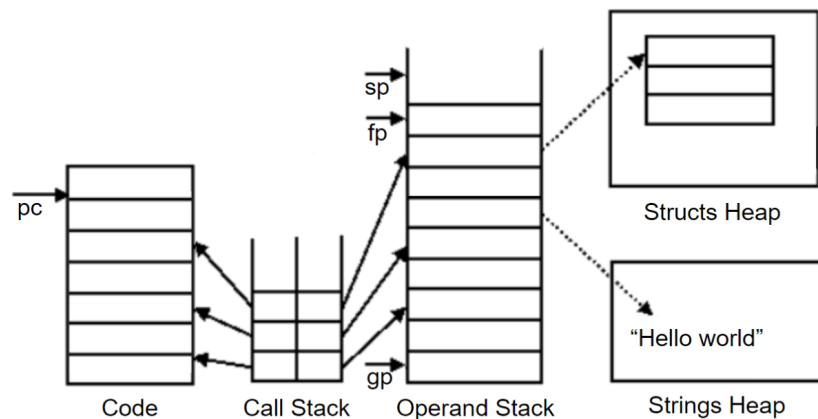
The **Code Zone** is, as implied by the name, the memory block that holds the instructions uploaded by the programmer.

The **Operand Stack** is a pile that contains numbers (integers or reals) and addresses. It is where the instructions operate, therefore it is used to store all the operands needed to be processed by the program operators.

In order to be able to manipulate more complex data than numbers, there are two heaps, **Strings Heap** and **Structs Heap**, to which the addresses in the Operand Stack may point to. VM is also equipped with four **registers**, aimed at the management of the memory components and responsible for the proper functioning of the machine.

- **Program Counter (PC)**: points to the current instruction in the code zone, i.e., the next to be fetched and executed.
- **Stack Pointer (SP)**: points to the top of the operand stack, i.e., the first free cell.
- **Frame Pointer (FP)**: points to the local variables base in the operand stack.
- **Global Variables Pointer (GP)**: holds the global variables base address.

And finally, a more complex program with functions and local variables, justifies the need for the other main component of this machine, the **Call Stack**. This pile contains pairs of pointers (i, j) that save present execution context (PC and FP) before a JUMP is executed to run the code of the called function. As such, every time a RETURN occurs, meaning that the function code has finished executing, the machine can recover the previous context and resume the normal execution. In the cell, the **Pointer i** holds the **PC** address and the **Pointer f** holds the **FP** address.



■ **Figure 1** VM's Architecture.

2.1.2 Functioning Principle

The Virtual Machine stores in its Code Zone memory the sequence of instructions that compose the program (machine code) provided by the programmer. Consequently, it works by accessing that block of memory and going through it. Looking at each memory cell pointed by the PC register, the VM checks for an operator, iterates through its operands, if there are any, and executes the instruction. This process repeats until the end of the instruction sequence that might be identified by a specific operation, STOP, completing the program execution.

2.1.3 Instruction Set

Each machine instruction, which may be preceded by a tag (label) followed by a colon, is a machine operation that may accept up to two parameters. The arguments can be integers, real numbers, chains of characters delimited by quotation marks (*strings*) or symbolic tags (labels) assigned to a code zone.

2.2 Related Work

In University of Beira Interior, Nuno Gaspar and Simão Melo de Sousa [8] had stumbled upon the same problem, the need of a web-based tool to easily teach about compilers. As such, they created a web application in which the user could choose one of the implemented Virtual Machines, upload the code he wishes to run and either visualize the result or a step by step visualization of the machine's state evolution. Moreover, they provide the option to increment the number of virtual machines available, making the system more versatile although more complex.

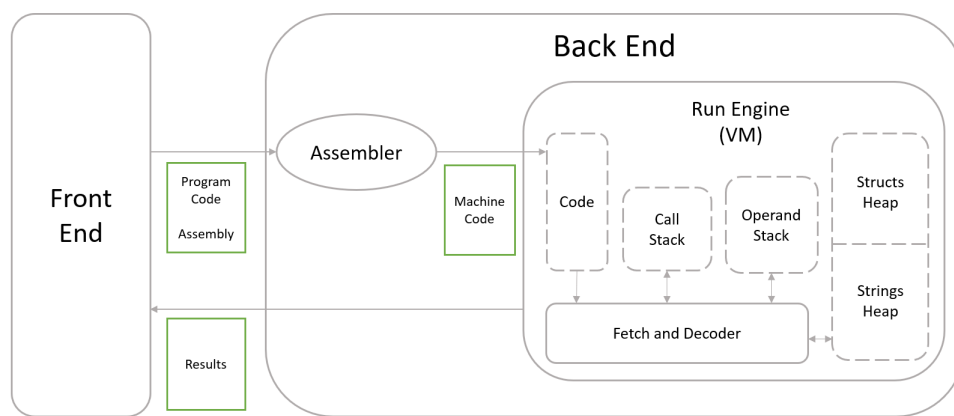
In conclusion, the concept is the same: to facilitate the learning process by providing a step by step visualization of the machine's state. However, EWVM focuses only on a low-level language and tries to maximize user experience in a simpler non intimidating way.

3 EWVM, proposal and architecture

The goal for creating this Educational Web Virtual Machine, EWVM, is to replicate the current Virtual Machine's behaviour with a superior graphical user interface along with easy accessibility. To this end, the new Virtual Machine will be developed as a Web Application

and will embody the same architecture and behaviour of the current one. As such, the new VM will contain the same memory components, **Code Zone**, **Call Stack**, **Operand Stack**, **String and Structured Blocks Heaps**, and its four registers, **Program Counter (PC)**, **Stack Pointer (SP)**, **Frame Pointer (FP)** and **Global Variables Pointer (GP)**. Evidently, the instruction set and instructions format will remain the same, apart from some additional features implemented to facilitate some basic operations that are missing in the present instruction set, such as the boolean operators AND and OR.

The Virtual Machine will be embedded in the *back-end* server of the Web Application, as depicted in Figure 2. The interface, in *front-end*, will take the programmer's code, send it to the server and wait for the results. In the server, the Program Code will pass through an **Assembler** which will turn it into Machine Code and send it into the **Run Engine**, where the Virtual Machine will be. The VM will then execute the code as explained and send back the result.



■ **Figure 2** EWVM Architecture.

The interface will be composed of three main areas, aiming to offer more powerful features while remaining user friendly (see the final result in Figure 3). Each sector will have various features implemented for user interaction, as described below,

- **Code Sector:** contains a text area where the user can write the program and the following buttons:
 - **Upload File:** reads the selected file and uploads its content (an Assembly program), displaying it in the text area of this sector;
 - **Save to File:** downloads into an external file the code (Assembly program) in the text area;
 - **Run:** sends the assembly code in the text area to the Back-End to be assembled and executed;
- **Animation Sector:** holds a container in which the user can visualize the virtual machine's internal state in each step of the code execution. It also contains the following features:
 - **Numbering:** each display is numbered and sorted by execution order
 - **Navigate:** by clicking on arrows, the user can move forward or backwards through the displays; it is also possible to move directly to the first or last display
- **Interaction Sector:** composed of two windows, one where the machine writes its outputs and the other from where it reads the user's inputs

4 EWVM, the tool

The Web Application is written in JavaScript. It is being developed in **Node.js**², a back-end event-driven JavaScript runtime environment that executes JavaScript code outside the web browser. It is designed to build scalable network applications and supports the **Express**³ framework, which offers a set of features for web applications.

Accordingly, since the program is written in JavaScript, the **Assembler** has been developed in **peggy**⁴, a JavaScript API. It is a parser generator that integrates both lexical and syntactical analysis and is based on a context free grammar formalism. The input grammar is written in Extended BNF (Backus Naur Form). Each grammar rule can be associated to a semantic action that is a fragment of javascript code written between curly brackets. Peggy processes a grammar and generates a fast and powerful compiler which receives an input text and returns either the results or a thorough and clear error report.

The grammar written to generate the Assembler is presented in Listing 1. It was created to analyse the assembly code written by the user and check if it is lexically and syntactically correct according to the Instruction Set rules. The grammar also semantically analyses the input and either detects an error or translates the received instructions to Machine Code.

Listing 1 Grammar.

```
Code = Line* _

Line = (_ Instruction) ([ \t\r]* Comment)*
      / Comment

Instruction = Label ':'
            / Inst_Atom
            / Inst_Int _ Integer
            / "pushf" _ Float
            / "pushs" _ String
            / "err" _ String
            / "check" _ (Integer _ "," _ Integer)
            / "jump" _ Label
            / "jz" _ Label
            / "pusha" _ Label

Inst_Atom = "stop" / "start" / "add" / "sub" / "mul" / "div" / "mod"
           / "not" / "infeq" / "inf" / "supeq" / "sup" / "fadd"
           / "fsub" / "fmul" / "fdiv" / "fcos" / "fsin" / "finfeq"
           / "finf" / "fsupeq" / "fsup" / "concat" / "equal" / "nop"
           / "atoi" / "atof" / "itof" / "ftoi" / "stri" / "strf"
           / "pushsp" / "pushfp" / "pushgp" / "loadn" / "storen"
           / "swap" / "writei" / "writef" / "writes" / "read" / "call"
           / "return" / "allocn" / "free" / "dupn" / "popn" / "padd"
           / "writel" / "and" / "or"

Inst_Int = "pushi" / "pushn" / "pushg" / "pushl" / "load" / "dup"
          / "pop" / "storel" / "storeg" / "store" / "alloc"
```

² <https://nodejs.org/en/>

³ <https://expressjs.com/>

⁴ <https://peggyjs.org/>

To transform the given Program Code into Machine Code, the generated compiler parses the Program Code and replaces the instruction mnemonics with their respective internal codes while managing the labels.

The Machine Code is in the form of an array and in each index every instruction code is saved along with its operands. To enable future connection between the code area and the animation, the Machine Code also saves the line number in which the instruction was written on. At the end, the grammar replaces the label addresses in the array with their respective array index.

```
Machine Code:
[ Instruction Code 1, Instruction Code 2, ...]

Instruction Code:
[ line number, instruction internal code, operands ]
```

In our tool, the Assembler generated by *peggy* is called each time the user clicks in the **Run** button, receiving as an input the text from the Code Sector and sending its result to the Virtual Machine.

Multiple features were implemented in the various sectors in order to improve this machine as an educational tool. We call *machine state* to the set of values stored in the data memory blocks (the two stacks and the two heaps) together with the values contained in the four registers after the execution of one specific instruction of the program stored in the code memory. Each sector enables the user to choose a machine state he wants to visit. After the choice of a state in one sector, the values displayed in the other sectors also change accordingly.

■ Code Sector:

- **Clickable Instructions:** by clicking on a code instruction (operator mnemonic), a text box appears with the operation description.
- **State Choice:** after a complete run of the input program, the user can, by clicking on an instruction line number, select the state corresponding to that instruction; if the instruction has been executed more than once, additional clicking should iterate through the respective states.
- **Connection:** the code sector highlights the instruction line number corresponding to the displayed state

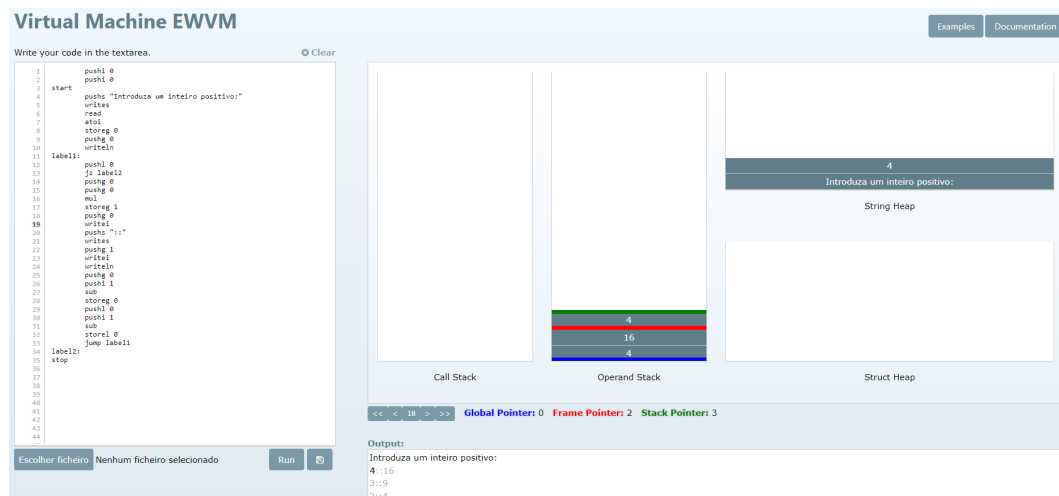
■ Animation Sector:

- **State Choice:** this sector contains a group of four navigation buttons that allow the user to visualize the various machine states
- **Connection:** illustrates the selected machine state

■ Interaction Sector:

- **State Choice:** by clicking on the text visible on the output window, it is possible to select the state in which that text was printed
- **Connection:** points out which text has already been printed (or is being printed) according to the machine state

The interface of the Web application that implements the desired educational version of the virtual machine VM, EWVM, can be observed in Figure 3. As planned, the user can analyse the state of the machine in each step of the code execution, enabling the student to get a clear understanding of the actual effect of each instruction over the components of the machine (this is the so-called *operational semantics* of his input program). For that purpose, the student can observe the values of registers and memory blocks using the information



■ **Figure 3** EWVM

displayed in the Animation Sector, while the corresponding instruction is highlighted in the Code Sector. In the Interaction Sector, the text printed is differentiated by colour. In this manner, the user can have a better understanding of the machine's behaviour.

As it is essential to the learning experience, a **Manual** is included containing the code's Documentation, where instructions are listed and explained. In order to give users a little more guidance and help, the website offers **Program Examples** categorized in terms of topic and difficulty. These are easily accessed and ran, providing the user the opportunity to explore these programs and learn from them.

5 Conclusion

When this work was initiated the aim was to deploy an environment to compile and execute VM programs with all the features described along the paper. We consider that we have attained this goal and that we went a little further. The created environment will be a valuable tool to teach students and lower their learning curve, as it will allow them to learn faster most of the key concepts in code generation. This tool allows teachers to create scenarios and test those scenarios before taking them to the classroom.

EWVM is available in a public URL: <https://ewvm.ep1.di.uminho.pt/>.

Although we still do not have sufficient data to sustain these claims, EWVM is now being used in the Language Processing Course with more than 160 students and is receiving very positive feedback. At the end of the next year we will have more data to demonstrate our beliefs.

Concerning future work, one other side goal was to have a tool easy to install. Since EWVM is a web application, once installed all users can access it with a simple browser. However, someone has to install it and to ease that job we will dockerize the application reducing its installation to a command line execution. We also intend to tune the VM language and now we have a platform that will help us doing that. For instance, the VM instruction set has some graphical instructions that, due to many difficulties in the graphical output, have been little explored in the actual tool. As future work we intend to give space to these instructions adding a panel to the interface with an HTML canvas or having the VM compiler produce SVG or other format that browsers can display and animate.

References

- 1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.
- 2 Sameer M AlNajdi, Malek Q Alrashidi, and Khalid S Almohamadi. The effectiveness of using augmented reality (ar) on assembling and exploring educational mobile robot in pedagogical virtual machine (pvm). *Interactive Learning Environments*, 28(8):964–990, 2020.
- 3 Tom Chothia and Chris Novakovic. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, Washington, D.C., August 2015. USENIX Association. URL: <https://www.usenix.org/conference/3gse15/summit-program/presentation/chothia>.
- 4 Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49, 2003.
- 5 Simão Melo de Sousa. *Máquina Virtual para o projecto da disciplina de compiladores*. Dep. de Informática, Universidade da Beira Interior, Covilhão, Portugal, September 2006.
- 6 Joseph A. Driscoll, Ralph M. Butler, and Joelle M. Key. A virtual machine environment for teaching the development of system software. In *Proceedings of the 42nd Annual Southeast Regional Conference*, ACM-SE 42, pages 440–441, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/986537.986647.
- 7 Fabian Fagerholm. Perl 6 and the parrot virtual machine, 2005.
- 8 Nuno Gaspar and Simão Melo de Sousa. WebVm – A web-based host platform for pedagogical virtual machines. *Special issue of the journal Informática na Educação: teoria & prática*, 1(4), January/June 2009.
- 9 Xuelian Hu and Dong Han. The design, implementation and application of minijava/ad as an object-oriented compiler teaching model. In *2009 4th International Conference on Computer Science Education*, pages 1488–1491, 2009. doi:10.1109/ICCSE.2009.5228571.
- 10 IBM Cloud Education. Virtual machines. <https://www.ibm.com/cloud/learn/virtual-machines>, June 2019. Accessed: 05-10-2021.
- 11 Kexugit. Why have a stack?, November 2011. URL: <https://docs.microsoft.com/pt-pt/archive/blogs/ericlippert/why-have-a-stack>.
- 12 Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *2009 Annual Computer Security Applications Conference*, pages 441–450. IEEE, 2009.
- 13 Michael Prantl. Python internals: An introduction, October 2020. URL: <https://blog.sourcerer.io/python-internals-an-introduction-d14f9f70e583>.
- 14 Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4), January 2008. doi:10.1145/1328195.1328197.
- 15 Mark Vinod Sinnathamby. Stack based vs register based virtual machine architecture, and the dalvik vm, September 2012. URL: <https://www.codeproject.com/Articles/461052/Stack-Based-vs-Register-Based-Virtual-Machine-Arch>.
- 16 Hasitha Subhashana. Understanding how java virtual machine (jvm) works, May 2021. URL: <https://hasithas.medium.com/understanding-how-java-virtual-machine-jvm-works-a1b07c0c399a>.

OMT, a Web-Based Tool for Ontology Matching

João Rodrigues Gomes¹ ✉

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Alda Lopes Gançarski ✉ 

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Pedro Rangel Henriques ✉ 

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Abstract

In recent years ontologies have become an integral part of storing information in a structured and formal manner and a way of sharing said information. With this rise in usage, it was only a matter of time before different people wrote distinct ontologies to represent the same knowledge domain. The area of Ontology Matching was created with the purpose of finding correspondences between different ontologies that represented information in the same domain area. This paper starts with a study of already existing ontology matching methods in order to understand the existing techniques, focusing on the advantages and disadvantages of each one. Then, we propose an approach and an architecture to develop a new web-based tool using the knowledge acquired during the bibliographic research. The paper also includes the presentation of a prototype of the proposed tool, called OMT.

2012 ACM Subject Classification Information systems → Web Ontology Language (OWL); Computing methodologies → Ontology engineering

Keywords and phrases Ontology, Ontology Matching, Ontology Alignment

Digital Object Identifier 10.4230/OASICS.SLATE.2022.8

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

In computer science, the concept of an ontology was firstly introduced in [7] as being a model to store data within a domain that allows the representation and definition of concepts, often referred to as classes, their properties and the existing relations between them. An ontology knowledge base is the result of populating an ontology with data that matches the concepts and properties that have been formally defined.

Ontology Matching methods were created with the purpose of relating information coming from multiple heterogeneous ontology sources into a common ontology model that encapsulates the entire knowledge base from all the sources [11]. On a simpler level, these methods have the task of finding correspondences between ontologies. The same concept or property can be defined using different terminologies on different ontologies.

There has already been a lot of research and work done in the field of Ontology Matching, and currently there are many approaches available that automatically generate matches between Ontologies. However, these techniques are still far from being perfect due to the fact that human input is still needed when the use case requires an accurate matching, making these methods impractical when dealing with big and complex ontologies [6].

This paper will focus, firstly, on the study of the already existing techniques, analysing the advantages and disadvantages of each one, and, secondly, on the presentation of a new web-based approach to create a tool that relies as less as possible on human input.

¹ corresponding author



To accomplish that twofold objective, the paper is structured in five sections. After the Introduction and before the Conclusion (Section 5), Section 2 defines the problem, Ontology Matching, presents the state-of-the-art in this area, and discusses related work found in the literature review. Then Section 3 proposes a solution to develop OMT, our web-based tool to match ontologies. The architecture of the proposed system is depicted and its modules explained. In Section 4, OMT prototype developed is presented.

2 Ontology Matching, related work

Ontology matching is the process of relating information from heterogeneous sources into a common model that can be queried and reasoned upon. On an abstract level, ontology matching is the task of finding correspondences between ontologies identifying a similar concept, property or relationship defined in both ontologies however using different terminologies.

Despite the efforts that have already been made in creating approaches to match ontologies, there is still a lot of room to grow regarding their ability to work in an autonomous way. Usually, these approaches evaluate ontologies given as input and come out with a group of possible correspondences. These correspondences have then to be examined by a person to determine which ones are correct, remove the false positives and create additional correspondences that were missed [6], making this whole process impractical when dealing with big and complex ontologies.

2.1 Matching Techniques

The approaches developed to solve the matching problem all make use of different techniques that each, in its own way, aims to exploit the information present in the ontology.

There are multiple criteria of classification that one can use to group up these approaches [11].

One simple way to divide these approaches is based on whether they analyze the information orthographically (string similarity) or semantically.

On the one hand, analysing information orthographically means that only the strings are important and taken into account. Many string techniques are available for that approach. On the other hand, analysing information semantically means that the meaning and context of the words are also taken into account. Obviously these approaches are not mutually exclusive and are used together in many different tools to achieve better results.

The following subsections take a closer look at some techniques used in these approaches.

2.1.1 String Based Techniques

In this category all the techniques analyse the information orthographically. They all measure the similarity of the Strings used to represent the concepts, properties, labels, comments and relationships of the ontology.

Distance functions map a pair of strings (x, y) to a real number r , where the smaller the value of r , the greater the similarity between the string pair [3]. There are many different implementations of these functions. One of the most important ones is the group of edit distance functions which measure the distance between two strings by calculating the most efficient sequence of pre-established operations that convert one string into the other. These pre-established operations are usually character insertion, deletion and substitution.

Another type of distance functions are the ones based on tokenization which is convert every string into a set composed of all its words (tokens). Afterwards, the similarity of the two strings is calculated by comparing the set of tokens that corresponds to each one. A simple metric is the *Jaccard Index*. Being (S,T) the respective sets of tokens of the string pair (s,t) , the Jaccard Index is obtained by the following formula:

$$Jac(s, t) = \frac{|S \cap T|}{|S \cup T|}$$

The Jaccard Index calculates the similarity of a pair of strings taking into account the relation between the number of common tokens in regards with the whole set of tokens.

One last type of distance functions are the called Hybrid functions which make use of different techniques. For example, the **Monge-Elkan Similarity Function** proposes a recursive approach to this problem, that is best used when comparing two long strings. This technique splits both strings into a set composed of all their sub-strings. It then compares every sub-string of one set with every sub-string of the other set, only saving the maximum value calculated through every iteration. It then computes the average of every maximum value as the value for the distance of two strings. Given a string pair (s,t) , let K and L the amount of sub-strings respectively, and **sim** a generic function to calculate the similarity between two strings, the Monge-Elkan Similarity Function is given by the following expression:

$$ME(s, t) = \frac{1}{K} \sum_{i=1}^K \max_{j=1}^L sim(S_i, T_j)$$

The *Jac* and *ME* are some of the many functions and approaches possible based on strings. This type of techniques is usually the first one to be used in a system in order to find the matching concepts, properties and relationships that are syntactically defined in a similar way.

2.1.2 Language Based Techniques

While, in the previous category, the techniques analyse the information of the ontologies syntactically, the techniques in the language based category analyse the information semantically. This means that the concepts, properties, labels, comments and relationships are not analysed as mere strings but are analysed as words that have a meaning in some language. Therefore, a lot of the techniques used rely on Nature Language Processing (NLP) and on the use of external sources such as dictionaries, lexicons and databases [11].

This approach makes use of different techniques such as **tokenization**, which, as has been explained previously, is the act of decomposing a String into smaller parts that compose it. In the specific case of language based techniques, tokenization is used to identify the words (tokens) that exist in the input string in order to better understand and exploit the possible meaning a word can have in the domain it is being expressed in.

Lemmatisation is another technique that is used. It is the process of finding the **lemma** that corresponds to the word that is being analysed. By definition, a lemma is the canonical form of a word and is considered the base form of a whole set of words that derive from it. For example, considering the set of words **walk**, **walking**, **walked**, **walkabout** all derive from the word **walk** and therefore **walk** is lemma of this set of words. One of the most common ways to find the lemma of a word is by looking it up in a dictionary.

Another technique also used is often referred to as **stop-word elimination**. This process is not as well defined as the other previous two and it depends on who is using it and in what context. It consists on creating a list composed of words that are considered not useful in the context of the problem and are therefore filtered out on the pre-processing stage of the string analysis phase.

These techniques are used to prepare the information being analysed. After they are applied, the resulting terms (**tokens**, **lemmas**) are then compared to check for their similarity. If they are not similar, then dictionaries and thesaurus are used to check if the terms have the same meaning, i.e if they are synonyms.

Language based techniques are some of the most important because they can find matches that, for example, the string based techniques do not. When dealing with ontologies written in different languages, these are the techniques that are mostly used, making use of dictionaries for example. Given that most matches in an ontology matching problem are not going to be orthographically equivalent, these techniques are in great use and hence is why tokenisation and lemmatisation are still being worked on and perfected nowadays.

2.1.3 Constraint and Instance Based Techniques

Constraint based techniques try to exploit the information of the constraints that can be placed on the properties and relationships of an ontology, such as the **Type** of data properties or the **Domain** and **Co-Domain** of relationships. This approach is based on the idea that, if properties and relationships on different ontologies match on the level of the constraints, then they are a potential match that needs to be studied. This technique is harder to be used alone and is mostly used in combination with other techniques to generate potential matches or analyse already generated matches.

Instance based techniques are considered an extension of all the techniques already presented, because they make use of the individuals when dealing with populated ontologies. Despite having the drawback of a much bigger quantity of information to analyze and consequently making the whole process take more time and use more resources, these type of techniques are great at both confirming already potential matches and generating new possible matches. When generating new matches, the general strategy is that if two individuals are alike, then the concepts they belong to are probably alike as well. When confirming potential matches, the general strategy is that if two concepts are alike and to be matched, then their individuals should also be alike [11].

2.2 Existing Ontology Matching Tools

In the following subsections, some of the many already existing ontology matching tools are presented focusing on the techniques each one uses as well as some of the restrictions the tools may have.

2.2.1 AROMA

AROMA (Association Rule Ontology Matching Approach) [5], is a tool for matching web directories, catalogs and ontologies designed in the OWL language.

The main technique used to find and reason matches is the **Association Rule Paradigm**. This paradigm focuses on the creation of implications/rules of the type $x \rightarrow y$ that should be interpreted as: “if a term x is found on a given schema of an ontology, then that ontology may be associated with the concept y ”.

Having this rule paradigm in mind, the first step in AROMA's algorithm is the extraction and selection of relevant terms associated to each concept. Then the process of creating and associating different rules to form matches in the different schemas is done. This is a simple overview of the methodology used when dealing with web directories or catalogs.

When the input schemas are those of an OWL ontology, the first step of the process had to be adapted. Rather than only using information contained in the schema of the ontology, the individuals data is also considered.

Although they do not go much in detail on the specific techniques used in each step of the algorithm, it is possible to deduce that language based techniques were probably used, especially tokenization to perform the extraction of the relevant terms.

This tool is not adequate to use in a general problem as it was developed with a specific target in mind, only allowing ontologies written in OWL to be used as input.

For a more in depth explanation of the methodology used in the AROMA tool, refer to [5].

2.2.2 AUTOMS

AUTOMS is a tool designed for the automatic matching of domain OWL ontologies [10]. As a tool, it integrates multiples methods that must be run in a particular sequence in order for the matching to be successful.

Firstly, a string based method is applied. It uses information concerning names, labels and comments of the ontologies concepts and properties and computes their similarity. The similarity between two terms is computed by making use of methods that compare the typographic similarities of Strings, sub Strings, sequences of ASCII characters and calculates how similar they are. It then proceeds to create pairs of possible matchings.

Secondly, WordNet is used to access if any of the pairs created are synonyms.

Following these first two methods, the string based and language based methods are combined into a single structure to determine their similarities in regards to concepts and properties. The individuals of the ontology are then considered to help find matchings for concepts that have not been determined to be similar in the input ontologies.

This tool is a perfect example how some of the techniques explained in the previous sections can be used in combination to each other in order to create an algorithm that yields good results. It uses syntactic and semantic based techniques, followed by instance based techniques to generate and confirm possible matches.

For a more in depth explanation of the entire methodology or for any individual methods used, refer to [10].

2.2.3 Hertuda

Hertuda is an ontology matching tool [9] designed to only accept as input ontologies compatible with the Lite or DL versions of the OWL language. It was developed to be a very simple matcher that takes advantages of tokenization and string measure to obtain alignments. It handles concepts, relations and properties independently, which results in three different sets of results, one for which.

For each concept, all its labels, comments and URIs are extracted, forming a set. To then compare concepts, its respective sets are compared, resulting in a similitary measure value. Before this comparison takes place, all the terms in all the sets are subjected to a pre-processing step, where tokenization occurs.

This is a much simpler tool compared to the others already studied. Its purpose was to push to its limits the string based techniques and the String similarity functions in order to obtain matches. It can be used for comparison and analysis but it will fail in cases where the terminology used in the matching ontologies is very different.

2.2.4 AgreementMaker

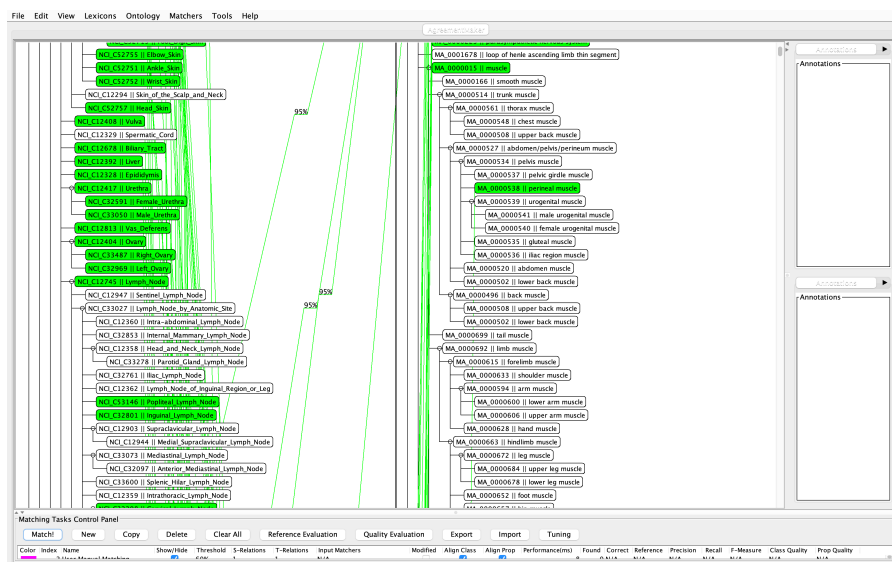
AgreementMaker [4] is one of the most developed and matured tools in this area as since its initial publication, it has been enhanced and improved upon multiples times, having been written follow up articles every single time.

One of its advantages is the amount of customization it offers the users in terms of its matching methods used, conceptual or structural methods, the amount of user interaction they require and if only the schema should be considered or the individuals as well. It also has built in metrics such as precision and runtime to present to the user.

The first group of matching algorithms include string-based and language based methods that compare labels, comments, annotations and instances. The second group of algorithms try to exploit the information present in the structure of the ontologies, such as the concepts, properties and the relations that exist amongst them. The final group combines all the results obtained in the first two groups with the goal of obtaining a unique final matching.

After the process of matching is done, Agreement Maker also offers automatic methods of evaluating the final matches. It considers the most effect evaluation technique to be the comparison of the matching found by the tool against a *gold standard* of the domain the ontologies are included in, preferably built by domain experts.

In the Figure 1 can be seen the interface of this tool.



■ **Figure 1** AgreementMaker Tool.

For further detail on this tool, refer to its original publishing article [4] and its source code is publicly available in its github repository.

2.2.5 MEDLEY

MEDLEY [8] is an OWL ontology matching system that takes a different approach from the tools that have been covered. It transforms every ontology triple into a graph structure, with the nodes being the ontology’s concepts, properties and individuals and links being the relations that links them.

The techniques used are mostly based on string based techniques and language based techniques, such as the similarity between strings, tokenization and stemmatisation while also making use of dictionaries to find equivalences.

The base logic of this method is that, if a given entity is alike an entity that is already in the graph structure, then the neighbours of that entity must also be neighbours of the given entity, generating possible matches.

For more details on this system, refer to [8].

2.2.6 Summary

In this subsection, already existing ontology matching tools are presented. A summary of the analysis made can be seen in Table 1. A last column was added to compare our tool, explained in the next section, with existing ones.

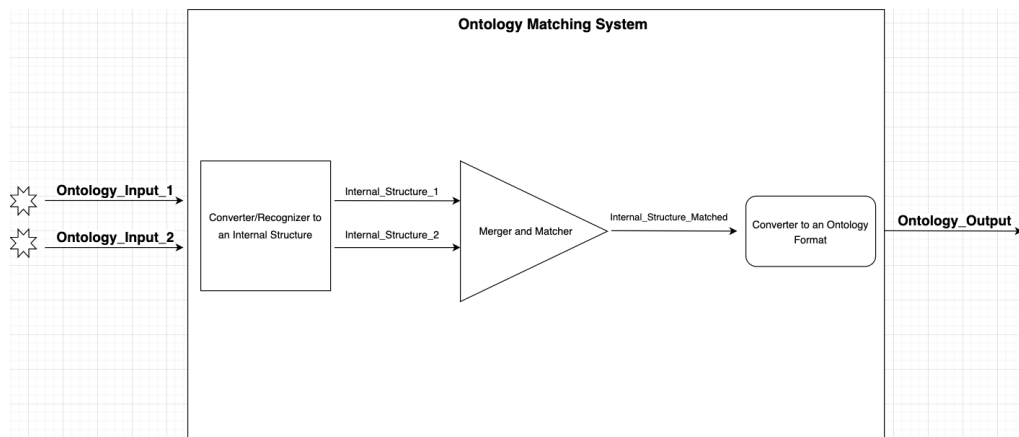
■ **Table 1** Ontology Matching Tools – Comparison.

	Ontology Matching Tools					
	AROMA	AUTOMS	Hertuda	AM	MEDLEY	OMT
multiple input formats	x			x		x
syntactic techniques		x	x	x	x	x
semantic techniques	x	x		x	x	x
association rules techniques	x					x
graphs structure					x	
evaluation methods				x		x

3 OMT, proposal and architecture

This section provides an initial architectural design of the ontology matching tool we propose as well as a more detailed explanation of the different components that make it up.

Figure 2 represents the architecture of the web-based ontology matching tool desired.



■ **Figure 2** General overview of the system architecture.

The two most important components of the matching tool are the initial converter (Step 2) and the Merger/Matcher (Step 3), as they are the ones that convert the initial ontologies into an homogeneous format to be analysed in order to find matches.

In the following sections, it will be given a more in depth look at each module of the process.

3.1 Ontology Matching tool inputs

The inputs of an ontology matching tool are a critical part of it, as they determine what and who can use that tool. The majority of the tools already created for this purpose restrict the format of the input ontologies, limiting the group of potential users of the tool. One of the goals for this tool is to be adaptable and easily accessible. Adaptable in the sense that the tool applies different techniques depending on the contents of the ontologies and easily accessible since it will be a web-based tool, making it available to anyone that wishes to use it. With that mind, we intend for the tool not to restrict the format of the input ontologies to a specific language and instead accept different formats such as the OWL language family [1] and Turtle [2]. The input ontologies can also be written in two different natural languages: english or portuguese.

3.2 Converter/Recognizer

The Converter/Recognizer is the first module of the process this tool is going to use. As mentioned before, the two input ontologies may be written in different ontology or natural languages. Therefore, this step is crucial in homogenizing the information present in the ontologies into an internal structure (an intermediate neutral ontology representation) that will hold the information relevant of each input ontology. If the ontology content is written in portuguese, an extra step is necessary to translate the respective information to english before starting the next step.

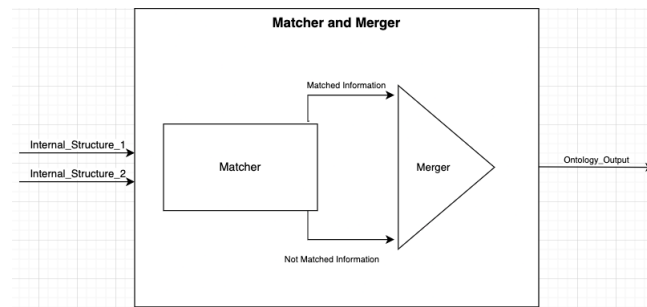
3.3 Matcher and Merger

The Matcher and Merger is the most important block. It is where the matching techniques are chosen and applied in order to find alignments in the two input ontologies.

This component can be broken into two smaller parts, the Matcher and the Merger. The Matcher finds the correspondences between the two input ontologies. The Merger takes all the concepts and relations from the given ontologies that match (i.e., all the elements that are discovered as common or belonging to both inputs) and joins them with all the elements that have not match, i.e., that belong to one of the input ontologies but do not belong to the other. This way it is possible to create in the internal structure a new ontology that is the merge of the given ontologies. This can be better understood in Figure 3.

3.4 Converter to an Ontology

The Converter to an ontology is the last block of the proposed architecture that implements the last task. It receives as input the internal structure generated by the Matcher/Merger component and converts the information on that structure into an ontology language. This process is the inverse of the process executed by the converter/recognizer. As it was said regarding that component, one of the goals is to allow diversity of ontology formats, so the objective concerning this module is to allow the output ontology to be written in different possible languages and it is up to the user to decide in what language he would like their matched ontology to be written. As is the case with the input ontologies, we are aiming to at least allow the output ontology to be written in OWL language family and Turtle.



■ **Figure 3** Detailed Matcher/Merger component.

4 OMT, the tool

This section presents a prototype of the proposed tool, OMT. The example shown corresponds to the analysis of two ontologies extracted from the Ontology Alignment Evaluation Initiative (OAEI) site², the first one describing the parts of the Human Body, and the second one describing the parts of the Mouse Body. Figure 4 shows how the users choose and upload the input ontologies to be matched, *human* and *mouse* ontologies as examples. After uploading an ontology, OMT identifies the natural language used to write its elements (concepts, properties and relations), and computes the size of concepts, properties and relations subsets, counting the number of elements in each subset. Figure 5 shows the information computed for both ontologies presenting it in a table to the user before starting the matching process. As it was mentioned in Section 3.2, if the input ontologies are written in portuguese, an additional step of translating the information to english is required. Figure 6 shows an example of said translation. Last but not least, Figure 7 shows the results of comparing the concepts of the two input ontologies. The process of comparing concepts makes use of techniques introduced in Section 2.1, such as **stop-word elimination** and **tokenization** to prepare the information and then distance functions such as the **Jaccard Index** to compare the similarity. The **matching value** column in Figure 7 represents how similar the 2 concepts are. The user has the option of clicking the **Details** link, that presents him with a new page (on the right, in Figure 6) showing the concepts and corresponding descriptions defined in the ontologies.

■ **Figure 4** Input Form to submit the ontologies to be matched.

These are the functionalities that have been implemented so far. After improving the Matcher module, the Merger module will be worked out, as well as the Web application that will host the OMT tool will be implemented.

² OAEI is accessible at <http://oaei.ontologymatching.org/>, and contains a large amount of different ontologies to be used as benchmarks to test Ontology Matching Tools.

Ontology Data

Ontology	Language	Concepts	ObjProperties	DataProperties
mouse.owl	EN	2744	25	19
human.owl	EN	3304	32	8

■ **Figure 5** Input ontologies data.

Original Ontology Data in PT**Concepts**

- Filho
- Mãe
- Pai

ObjProperties:

- filhoDe
- mãeDe
- paiDe

DataProperties:

- Idade
- Nome

Translated Ontology Data to EN**Concepts**

- Child
- Mother
- Father

ObjProperties:

- childOf
- motherOf
- fatherOf

DataProperties:

- Age
- Name

■ **Figure 6** Translation of the elements of a PT ontology to an EN counterpart.

5 Conclusion

Ontology matching is an important functionality in many applications for relating information, e.g., from heterogeneous sources into a common model that can be queried and reasoned upon. The work in progress we present in this paper contributes to the ontology matching research issue proposing a new adaptable and easily accessible system called OMT. We summarize the study based on the extensive literature review done on already existing ontology matching methods. This study gave rise to new ideas we integrate in our system. An overview of the architecture of the system is then presented in the paper. The Matching and Merge component of the system integrates techniques coming from String, Language and Constraint and Instance based techniques.

The prototype, so far implemented as a proof of concept of our proposal, is also introduced. As explained, that prototype allows for the choice of the two ontologies to be merged. Before the matching process, if one of the input ontologies is written in portuguese, it is automatically converted to an english version. After converting the input ontologies into the same internal format, OMT tool outputs a results table with the concepts matching. This table exhibits the concepts identified as similar after analysing the two given ontologies, being also displayed the matching percentage evaluated for each pair.

Ontology1 Concepts	Ontology2 Concepts	MatchingValue	
NCI_C12220	MA_0000343	1.0	Details
NCI_C38617	MA_0000006	0.81	Details
NCI_C12419	MA_0000023	1.0	Details
NCI_C12221	MA_0002232	0.88	Details
NCI_C12223	MA_0001320	0.88	Details
NCI_C12222	MA_0002232	0.83	Details
NCI_C12224	MA_0001018	0.75	Details
NCI_C12226	MA_0001018	0.8	Details
NCI_C12225	MA_0001018	0.78	Details
NCI_C13166	MA_0000353	0.83	Details
NCI_C12227	MA_0002365	0.76	Details
NCI_C12228	MA_0001761	0.81	Details
NCI_C38626	MA_0001550	0.77	Details
NCI_C12422	MA_0000347	1.0	Details
NCI_C12229	MA_0002476	1.0	Details
NCI_C12421	MA_0000055	0.88	Details
NCI_C12230	MA_0002477	1.0	Details
NCI_C12231	MA_0002412	1.0	Details
NCI_C12232	MA_0002480	0.88	Details

NCI_C38617 and MA_0000006
Matching Value: 0.81
<hr/>
Concept: NCI_C38617
Ontology: human.owl
Description: 'Head_and_Neck_Part'
<hr/>
Concept: MA_0000006
Ontology: mouse.owl
Description: 'head/neck'
<hr/>
NCI_C12230 and MA_0002477
Matching Value: 1.0
<hr/>
Concept: NCI_C12230
Ontology: human.owl
Description: 'Hard_Palate'
<hr/>
Concept: MA_0002477
Ontology: mouse.owl
Description: 'hard palate'

■ **Figure 7** Matching the Concepts of two input ontologies.

As future work, we will formalise the algorithms involved in the system components, giving a special focus on the Matching and Merge algorithm. The system will then be fully implemented and tested with several heterogenous data. Finally, an evaluation of the system will be conducted, comparing it with other existing tools.

References

- 1 Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah McGuinness, Peter Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. Recommendation, World Wide Web Consortium (W3C), February 10 2004. See <http://www.w3.org/TR/owl-ref/>.
- 2 Gavin Carothers and Eric Prud'hommeaux. RDF 1.1 turtle, February 2014. URL: <http://www.w3.org/TR/2014/REC-turtle-20140225/>.
- 3 William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IWeb*, 2003.
- 4 Isabel F. Cruz, Flavio Palandri Antonelli, and Cosmin Stroe. Agreementmaker: Efficient matching for large real-world schemas and ontologies. *Proc. VLDB Endow.*, 2:1586–1589, 2009.
- 5 Jérôme David, Fabrice Guillet, and Henri Briand. Matching directories and owl ontologies with aroma. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management, CIKM '06*, page 830–831, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1183614.1183752.
- 6 Sean M. Falconer and Natalya F. Noy. *Interactive Techniques to Support Ontology Matching*, pages 29–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- 7 Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
- 8 Walid Hassen. Medley results for oaei 2012. In *Proceedings of the 7th International Conference on Ontology Matching - Volume 946, OM'12*, pages 168–172, Aachen, DEU, 2012. CEUR-WS.org.

8:12 OMT, for Ontology Matching

- 9 Sven Hertling. Hertuda results for oaei 2012. In *OM*, 2012.
- 10 Konstantinos Kotis, Alexandros G Valarakos, and George A Vouros. Automs: Automated ontology mapping through synthesis of methods. In *Ontology Matching*, page 96, 2006.
- 11 Lorena Otero-Cerdeira and Francisco Javier Rodríguez-Martínez and Alma María Gómez-Rodríguez. Ontology matching: A literature review. *Expert Syst. Appl.*, 42:949–971, 2015.

Classification of Public Administration Complaints

Francisco Caldeira ✉

Iscte, University Institute of Lisbon, Portugal

Luís Nunes ✉ 

Iscte, University Institute of Lisbon, Portugal

ISTAR, Lisbon, Portugal

Ricardo Ribeiro ✉ 

Iscte, University Institute of Lisbon, Portugal

INESC-ID Lisbon, Portugal

Abstract

Complaint management is a problem faced by many organizations that is both vital to customer image and highly dependent on human resources. This work attempts to tackle a part of the problem, by classifying summaries of complaints using machine learning models in order to better redirect these to the appropriate responders. The main challenges of this task is that training datasets are often small and highly imbalanced. This can have a big impact on the performance of classification models. The dataset analyzed in this work suffers from both of these problems, being relatively small and having labels in different proportions. In this work, two different techniques are analyzed: combining classes together to increase the number of elements of the new class; and, providing new artificial examples for some classes via translation into other languages. The classification models explored were the following: k -NN, SVM, Naïve Bayes, boosting, and Deep Learning approaches, including transformers. The paper concludes that although, as expected, the classes with little representation are hard to classify, the techniques explored helped to boost the performance, especially in the classes with a low number of elements. SVM and BERT-based models outperformed their peers.

2012 ACM Subject Classification Information systems → Clustering and classification

Keywords and phrases Text Classification, Natural Language Processing, Deep Learning, BERT

Digital Object Identifier 10.4230/OASICS.SLATE.2022.9

Funding This work was partly funded through national funds by FCT - Fundação para a Ciência e Tecnologia, I.P. under project UIDB/04466/2020 (ISTAR).

1 Introduction

The process of manual classification of user generated data can be regarded as a tedious and error prone task. In the attempt to render this process more efficiently, this paper explores the use of Machine Learning and Natural Language Processing tools and models for this task.

This work will focus on the classification of summarized complaints received by a Portuguese public institute. After being received, the complaints are read and manually processed by a worker from the public institute. Then, after being correctly classified, the complaint is forwarded to the respective department for additional processing, accompanied by a textual summary. We concentrate on the classification of summaries for redirection to the appropriate responders. The final goal is to render assistance in the whole process. Due to privacy questions, the dataset is not available.

The related work is presented in Section 2. Data is explored in Section 3, along with the techniques used for processing, providing some insights on the structure and contents of the dataset. The implementation is described in Section 4. In Section 5 the achieved results are presented. Finally, the document closes with the conclusions and some possible future work in Section 6.



© Francisco Caldeira, Luís Nunes, and Ricardo Ribeiro;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 9; pp. 9:1–9:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Related Work

Complaint data can be viewed and analyzed as a “form of user generated data” and thus the current state of the art for processing user commentaries of social media content can be deemed useful for the task analyzed in this work.

A similar problem to ours was also tackled by Lopes-Cardoso et al. [5]. This work focus on the classification of complaint data from another Portuguese public institute in three different dimensions: economic activity prediction, infraction severity prediction, and competence prediction. The authors reported SVM with a linear kernel was the best performing model with around 79% accuracy on the competence prediction task, which is the one most similar to ours. In a comparable stream of work [7, 8, 2], the focus was also the classification of data from a multitude of Portuguese public services while also dealing with noisy and imbalanced data. The best results were achieved by SVM and BERT-based approaches. Oliveira et al. [4] used a set of frequently asked questions from a Portuguese public institute for a classification task. To create additional data the authors used Google API to translate back and forth to Portuguese and English to develop new data. They also used native speakers to create and improve the alternative constructions of the questions. In their results, the best performing model was SVM with a better training time/performance ratio when compared to other models. In the end they also noted that using a fine tuned BERT model improved the results at cost of higher train time. For benchmark they used the set of the generated question to match the original ones and a classification based of the area of business of each question. In [3] the authors performed an analysis of user commentaries from a Portuguese telecommunication company for a sentiment analysis task. They noted that in stemming in Portuguese is not very useful and resorted to custom built list of rules to work along side the stemming to reduce edge cases. In [6], the authors compared SVM with Universal Language Model Fine-tuning (ULMFiT), for classification of official Brazilian Government data. The concluded that, even though ULMFiT is a state-of-the-art technique for classification, it only corresponded to a small increase in classification accuracy when compared to the SVM model.

Wang et al. [13] explore “*Label-Embedding Attentive Model*” (LEAM) by proposing a word embedding approach in which both words and labels are joined in the same latent space in order to measure the compatibility of word-label used as document representations. While in some cases the LEAM model was unable to out perform other models the authors stated the algorithm is much less demanding in comparison to other state-of-the-art algorithms. Tang et al. [12] also worked in classification of complaints and proposed a combination of BERT and word2vec models to try and improve the overall accuracy. To tackle imbalanced data they experimented with translations of the text to expand the training data.

Further expanding on deep learning strategies, a specific BERT model was trained on Brazilian Portuguese data, the model was named BERTimbau [11]. When comparing BERTimbau performance against BERT for Named Entity Recognition tasks and Sentence Textual Similarity the first would outperform the latter with the authors stating “*large pre-trained learning models can be valuable assets especially for languages that have few annotated resources but abundant unlabeled data, such as Portuguese*” [11].

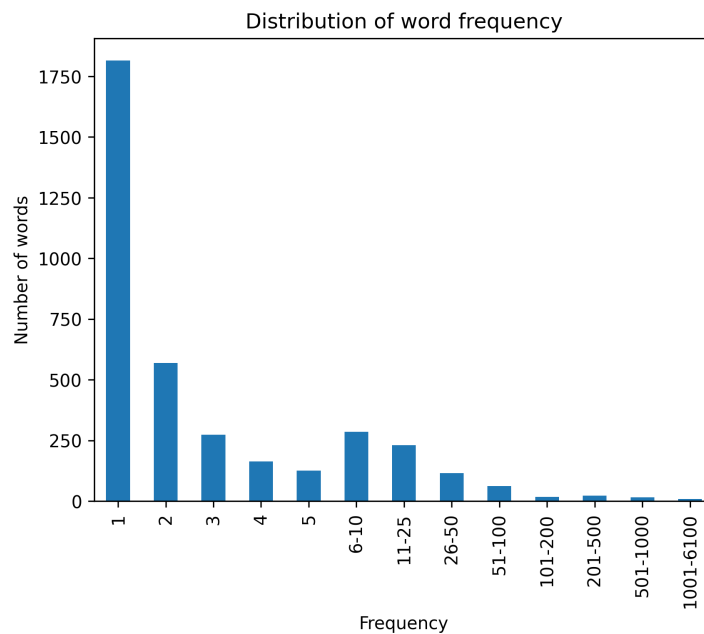
Also worth considering a portuguese wordnet, semantically structured lexical database, expansion done in [10]. The word database is useful for extracting synonyms to create more artificial entries in the data set.

For this work, the techniques used to enhance are tested in combination with the Portuguese text processing strategies in order to tackle the classification task. With the main goal of understanding the benefit of these techniques.

3 Data

The complaints dataset features 4459 complaints gathered from 2020 to 2021 and spread over 17 different labels. Each category directly references to an entity governed by the portuguese Ministry of Justice and are listed here <https://igsj.justica.gov.pt/Servicos/Apresentar-queixa>. Not all entities are featured in the dataset since not all receive complaints. The complaints were subject to an initial processing from the institute that summarized the content of the complaint into a short text, leaving mostly the relevant words in the text. The dataset featured two main columns used for the classification task, the complaint and the assigned department.

The summarized complaints have an average of 16 words per complaint with the shortest having only 2 words and longest featuring 38 words, all of them written in Portuguese. The smallest summaries(of only two words) was repeated 2 times and the text can be seen as *"process delays"* and the summaries of with 3 and 4 words were similar with the added words giving more detail regarding the actual complaint. Of the 4459 summarized complaints it was extracted a corpus of 3705 different tokens, the full distribution can be seen in Figure 1, with 1805 tokens only appearing once and two appearing almost all summaries.



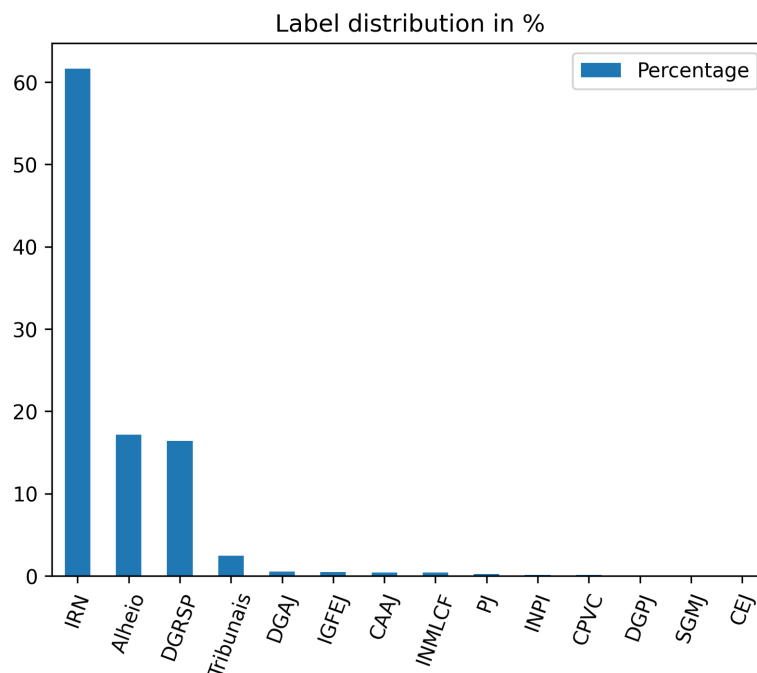
■ **Figure 1** Word frequency distribution.

The dataset presents high variability in what regards label cardinality: the category with more complaints covers almost 60% of the dataset while the second and third classes only account for 35%. Six labels also contained ten or less complaints with a single class only containing one complaint.

One single complaint was assigned two different classes and was re-assigned to the class that presented the lowest number of complaints.

Since the text data was preprocessed by a worker of the public institute, the text data features little spelling errors unlike some other user generated data like social media content.

9:4 Classification of Public Administration Complaints



■ **Figure 2** Label distribution of the data explored, more than 90% of the data was labeled to the top 3 labels.

4 Experiments

In this section the text processing tasks and the techniques used to handle the data imbalance are detailed.

Considering the properties of the dataset, both traditional classification and deep learning approaches are compared. We aim to better understand the performance of the experimented models with the techniques used for dealing with imbalanced data.

4.1 Data Preparation

Handling data imbalance was achieved with two different strategies, translating the texts of the complaints and creating additional labels to group classes with lower cardinality.

To increase the number of class representatives, the documents of the classes that had between 10 and 30 elements were translated into several languages (English, Spanish, Italian, Polish, and German) and back to Portuguese. This strategy increased the number of representatives by sixfold (Table 1). The artificially produced complaints were only used for model training to ensure the validation was performed with real world data.

For the classification experiments both datasets needed to undergo a battery of processing steps in order to be used as input for the models:

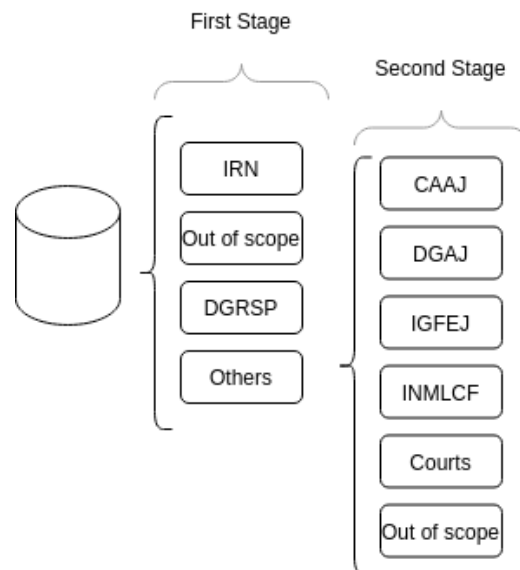
- lower casing all characters;
- removing numerical data;
- removing special characters;
- removing diacritics.

As noted in [3] stemming is not very useful for the portuguese language was not considered for this task.

The texts also featured some acronyms that could be more insightfully expanded. For example, “cc” was replaced to match “*cartao do cidadao*” (identity card) and “ep” was replaced by “*estabelecimento prisional*” (prison establishment).

The documents were then regrouped into different labels. Firstly, the complaints were separated into the top 3 classes and the remaining classes. For the final split, the top 4 to the top 8 classes were given the respective label and labels that had less than 10 example were combined into a single class. Figure 3 illustrates this process. From this point on wards, the first stage classification will refer the classification of the top 3 classes along with the “Others” (new label for the remaining classes) and the second stage classification will refer to classification of the others to the remaining considered classes. The full data classification will refer to the top 8 classes only along with the remaining grouped into the class “Others”. For all the experiments, the classes with a cardinality lower than 15 were not considered as a full class: they were assigned the label of “Others”.

The groups were assembled based on the labels distribution in Figure 2 with the first stage containing only the more frequent labels and for the second stage the split were the categories with more than 15 examples, the size of the classes can be seen in Table 2, the complete dataset without any segmentation was used to compared the performance of the multistage classification.



■ **Figure 3** To handle the label imbalance issue a multistage classification method was analyzed, the top 3 classes were initially classified and the remaining top 5 classes were also considered, class with lower representation were not considered for the task.

The translation technique was particularly useful. When translating back to Portuguese some synonyms appeared adding more words to the corpus and complementing the shortcomings of the less populated labels. After increasing the number of examples, the dataset featured 5000 complaints, almost an increase of 500 new complaints and an extra 300 new tokens.

■ **Table 1** Distribution of complaints per label, the first 3 labels contain almost 80% of the data set. Using the translation technique made the classes more equal in regards to their size.

	Class	Class with translations
IRN	2748	2748
Alheio	765	765
DGRSP	732	732
Courts	109	109
DGAJ	22	132
IGFEJ	21	123
CAAJ	19	114
INMLCF	17	102
PJ	10	60
INPI	5	30
CPVC	5	30
DGPJ	3	18
SGMJ	2	12
CEJ	1	6

4.2 Experimental Setup

The complaints were processed using the standard tokenization pipelines (special characters were removed). All characters were lowercased and the TF-IDF model was used to get the feature vectors for each complaint. For the BERT-based approach, instead of TD-IDF the tokens were preprocessed using the model encodings. For additional testing and comparison the words that only appeared once were removed as well as the top 3 words.

The experimented models were evaluated using the standard metrics with the goal to compare performance gain by using the techniques referred to handle the highlighted issues: low cardinality of the dataset and the class imbalance that was previously mentioned.

From here the experiments were separated into two distinct experiences, having the translations in the training and the original complaints in order to evaluate the performance gain by using the fabricated examples.

The dataset was split into two groups 30% for testing, 70% was used for training and from the training set 15% was used for validation. For the deep learning models, the training set was split into an additional validation set for training purposes. For comparison, the training set for the original dataset had 520 examples for the first stage while the expanded training set featured 1713. This difference is even noticeable in the second stage training from 56 to 169 examples for the training, refer to Table 2.

■ **Table 2** Number of inputs for the different tasks for each model.

	Classification	Original	Augmented
Training	Full	84	627
	First Stage	520	1713
	Second Stage	56	169
Validation	Full	15	111
	First Stage	92	303
	Second Stage	10	74
Testing		1338	

4.3 Methods

As previously mentioned, given the properties of the dataset, we experimented several classification models, ranging from more traditional approaches like Naïves Bayes, k -NN, and SVM models to deep learning-based approaches as Multilingual-BERT and XGBoosting.

To validate and optimize the hyper parameters of the models it was used cross-validation, splitting the data into multiple smaller subsets with equal class cardinality to validate the classifier while also avoiding to overfit the experimented models.

Several classification pipelines were also tested, classifying the full data set, only the first stage classification, and performing a multistage classification as illustrated in Figure 3.

5 Results and Discussion

The results for the classification and using the original dataset can be viewed in the Tables 3, 4, and 5.

■ **Table 3** Results for the first stage using the original dataset, SVM outperform all the other models for this task but Naïve Bayes presented a marginally higher f-score.

Model	Processing	Accuracy	Precision	Recall	F-score
Naive Bayes	No special processing	0.894619	0.921471	0.894619	0.903991
	Removing stopwords, low frequency words	0.902093	0.930147	0.902093	0.911670
SVM	No special processing	0.899851	0.929783	0.899851	0.910339
	Removing stopwords, low frequency words	0.894619	0.934678	0.894619	0.908281
k -NN	No special processing	0.853513	0.887257	0.853513	0.867000
	Removing stopwords, low frequency words	0.855007	0.895267	0.855007	0.870560
XGBoost	No special processing	0.826607	0.883714	0.826607	0.848738
	Removing stopwords, low frequency words	0.828102	0.903759	0.828102	0.856595
Multilingual-Bert	No special processing	0.869207	0.91050	0.86920	0.883518
	Removing stopwords, low frequency words	0.857249	0.9442	0.857249	0.88504

Considering the first stage classification for the original dataset, the deep learning models and the more traditional models yielded similar results to SVM, proving to be the model with best performance in this experiment.

For the second stage classification, the results were the worst from all the experiments. While presenting around 70% precision, the accuracy and f-score values were extremely low. Due to low number of class representatives, the BERT model was unable to be fine tuned. The training and validation data would be very short for the complexity of a neural network.

For the full classification when using the original dataset, k -NN and SVM presented good results with an 80% accuracy and 92% precision, respectively. When comparing the results for each class in Table 6 it becomes apparent the good results are heavily weighted by the distribution of the testing set. The classes with more representatives have more weight and yield better performance.

The deep learning models had an accuracy value of 58% when compared to k -NN with 80% accuracy.

■ **Table 4** Results for the second stage classification using the original dataset, low performance across all models. BERT model was unable to trained for this stage.

Model	Processing	Accuracy	Precision	Recall	F-score
SVM	No special processing	0.312977	0.672483	0.312977	0.246029
	Removing stopwords, low frequency words	0.293333	0.700265	0.293333	0.242265
Naive Bayes	No special processing	0.259843	0.577479	0.259843	0.181176
	Removing stopwords, low frequency words	0.271318	0.753100	0.271318	0.188976
XGBoost	No special processing	0.166667	0.606909	0.166667	0.184742
	Removing stopwords, low frequency words	0.304569	0.791608	0.304569	0.370414
<i>k</i> -NN	No special processing	0.230769	0.738754	0.230769	0.194529
	Removing stopwords, low frequency words	0.312500	0.809010	0.312500	0.311697

Considering only the first stage classification task and not using, the results were much better results with 89% accuracy using SVM, although only for a limited number of classes. The full classification, while also presenting good results, was skewed from the imbalance of the testing set, most of the accuracy was attributed to three labels.

The results for the classification using the expanded dataset can be viewed in Tables 7, 8, and 9. The expanded dataset yielded considerably better results for all the tasks examined in this work.

The first stage classification had an increase of almost 6% from the original data set with SVM outperforming the others. Surprisingly the second stage classification almost doubling the performance achieved with the original dataset. Accuracy improved to 58% from 29% and f-score increased to 52% from 24%, when using an SVM. With the augmented dataset it was possible to fine-tune BERT for the second classification and it proved to be the best performing model for this task.

For the full classification using SVM, the accuracy reported was of 89% with 90% f-score, an increase of 10p.p. from the dataset. Using BERT for this task returned similar results to the SVM noting the approximated 20p.p. increase in accuracy from the original dataset, from 58% to 76%. Having a bigger training pool was essential in improving the BERT model.

6 Conclusion and Future Work

Similar to the results achieved by Silva et al. [9], the concise versions of the complaints proved to be enough for a reasonable and usable classification result, in the data explored removing stopwords and frequent words had low gains. In this case, since the tokens were already part of a short corpus, the additional processing (removing stopwords, the most highest and low frequency words) showed only marginal improvements. The classes with more representatives were the ones that achieved a higher score in precision, as the complaints were more similar and mostly relating to a more frequent issue. The categories with lower representatives were generally harder to classify as the texts and issues featured little resemblance.

Due to the low number of complaints for some classes, the more traditional models had a better performance than the deep learning models, especially when using the original dataset. When using the expanded dataset, deep learning models yielded similar results to more traditional models. For the second stage classification BERT was able to outperform

■ **Table 5** Results for the full classification using the original dataset, k -NN and SVM have similar scores and present decent performance although SVM has a higher precision and f-score.

Model	Processing	Accuracy	Precision	Recall	F-score
Naive Bayes	No special processing	0.750374	0.907025	0.750374	0.810925
	Removing stopwords, low frequency words	0.770553	0.92491	0.770553	0.832805
SVM	No special processing	0.786996	0.913812	0.786996	0.837782
	Removing stopwords, low frequency words	0.793722	0.924617	0.793722	0.845285
KNN	No special processing	0.796712	0.887542	0.796712	0.833793
	Removing stopwords, low frequency words	0.803438	0.883222	0.803438	0.837429
XGBoost	No special processing	0.601644	0.842656	0.601644	0.685857
	Removing stopwords, low frequency words	0.633782	0.894219	0.633782	0.730709
Multilingual-Bert	No special processing	0.58071	0.82905	0.58071	0.67277
	Removing stopwords, low frequency words	0.550822	0.84441	0.550822	0.65165

■ **Table 6** Metrics by class for SVM using the original dataset.

	Precision	Recall	F-Score	Support
Alheio	0.88	0.64	0.74	247
CAAJ	0.14	0.83	0.24	6
DGAJ	0.05	0.43	0.10	7
DGRSP	0.97	0.90	0.94	228
IGFEJ	0.11	0.75	0.19	8
INMLCF	0.13	0.83	0.22	6
IRN	0.98	0.82	0.89	802
Others	0.11	0.57	0.18	7
Courts	0.27	0.52	0.35	27

all of the others. The expanded dataset led to considerably better results when compared to the original data set, especially for the second stage classification task and for the full classification. Boosting the number of representatives for each class, especially the least represented ones, greatly improved the performance of the models. The new sentences and the tokens introduced were essential to improve the BERT performance. Even though further testing is needed to confirm this, these experiments seem to indicate that balancing the data played a crucial role in the performance gains and could be considered as technique for improving datasets with lower cardinality.

For future work, the fine tuning of a multistage classification method should be explored while also considering more classification stages and using binary classification [1]. More techniques for expanding the corpus and classes examples could also be explored, as an example wordnets could be used to further diversify the dataset. Producing hand made examples for a class could also provide better representatives for each class as the machine made translations that can only reach a certain limit. It should also be noted that combining different models for the various stages of the classification could provide additional insights.

9:10 Classification of Public Administration Complaints

■ **Table 7** Results for the first stage classification using the augmented dataset, SVM outperform all the other models for this task.

Model	Processing	Accuracy	Precision	Recall	F-score
Naive Bayes	No special processing	0.930493	0.932668	0.930493	0.931215
	Removing stopwords, low frequency words	0.933483	0.937128	0.933483	0.934630
SVM	No special processing	0.934230	0.941662	0.934230	0.937099
	Removing stopwords, low frequency words	0.940957	0.945378	0.940957	0.942697
<i>k</i> -NN	No special processing	0.911809	0.914773	0.911809	0.911943
	Removing stopwords, low frequency words	0.904335	0.908641	0.904335	0.905653
XGBoost	No special processing	0.904335	0.918122	0.904335	0.909718
	Removing stopwords, low frequency words	0.912556	0.924713	0.912556	0.917385
Multilingual-Bert	No special processing	0.93	0.93025	0.93	0.92831
	Removing stopwords, low frequency words	0.936472	0.93789	0.936472	0.93710

■ **Table 8** Results for the second stage classification using the expanded dataset, BERT outperform all the other models for this task.

Model	Processing	Accuracy	Precision	Recall	F-score
SVM	Removing stopwords, low frequency words	0.578947	0.767832	0.578947	0.525620
	No special processing	0.488095	0.772963	0.488095	0.405475
Naive Bayes	Removing stopwords, low frequency words	0.447368	0.480623	0.447368	0.335068
	No special processing	0.472222	0.432330	0.472222	0.370383
XGBoost	Removing stopwords, low frequency words	0.534884	0.716058	0.534884	0.535191
	No special processing	0.455556	0.718620	0.455556	0.447290
<i>k</i> -NN	Removing stopwords, low frequency words	0.487500	0.751803	0.487500	0.405868
	No special processing	0.432432	0.775594	0.432432	0.330564
Multilingual-Bert	No special processing	0.65217	0.64684	0.65217	0.59163
	Removing stopwords, low frequency words	0.55384	0.7719	0.55384	0.483812

■ **Table 9** Results for the full classification using the expanded dataset, SVM outperform all the other models for this task.

Model	Processing	Accuracy	Precision	Recall	F-score
Naive Bayes	No special processing	0.870703	0.931320	0.870703	0.891887
	Removing stopwords, low frequency words	0.878924	0.931459	0.878924	0.896761
SVM	No special processing	0.884155	0.921298	0.884155	0.897685
	Removing stopwords, low frequency words	0.890882	0.924622	0.890882	0.902759
<i>k</i> -NN	No special processing	0.835575	0.908580	0.835575	0.860282
	Removing stopwords, low frequency words	0.837818	0.909102	0.837818	0.863215
XGBoost	No special processing	0.774290	0.877428	0.774290	0.814150
	Removing stopwords, low frequency words	0.791480	0.881701	0.791480	0.825520
Multilingual-Bert	No special processing	0.76233	0.9231	0.76233	0.819744
	Removing stopwords, low frequency words	0.84679	0.93258	0.84679	0.88095

References

- 1 Fernando Batista and Ricardo Ribeiro. Sentiment analysis and topic classification based on binary maximum entropy classifiers. *Proces. del Leng. Natural*, 50:77–84, 2013. URL: <http://journal.sepln.org/sepln/ojs/ojs/index.php/pln/article/view/4662>.
- 2 André Fazendeiro. Automatic correspondence distribution for a public institution. Master’s thesis, Instituto Superior Técnico, 2021.
- 3 Ana Catarina Forte and Pavel B. Brazdil. Determining the level of clients’ dissatisfaction from their commentaries. In João Silva, Ricardo Ribeiro, Paulo Quaresma, André Adami, and António Branco, editors, *Computational Processing of the Portuguese Language*, pages 74–85, Cham, 2016. Springer International Publishing.
- 4 Hugo Gonçalo Oliveira, João Ferreira, José Santos, Pedro Fialho, Ricardo Rodrigues, Luisa Coheur, and Ana Alves. AIA-BDE: A corpus of FAQs in Portuguese and their variations. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 5442–5449, Marseille, France, May 2020. European Language Resources Association. URL: <https://aclanthology.org/2020.lrec-1.669>.
- 5 Henrique Lopes-Cardoso, Tomás Freitas Osório, Luís Vilar Barbosa, Gil Rocha, Luís Paulo Reis, João Pedro Machado, and Ana Maria Oliveira. Robust complaint processing in portuguese. *Information*, 12(12), 2021. doi:10.3390/info12120525.
- 6 Pedro Henrique Luz de Araujo, Teófilo Emidio de Campos, and Marcelo Magalhães Silva de Sousa. Inferring the source of official texts: Can svm beat ulmfit? In Paulo Quaresma, Renata Vieira, Sandra Aluísio, Helena Moniz, Fernando Batista, and Teresa Gonçalves, editors, *Computational Processing of the Portuguese Language*, pages 76–86, Cham, 2020. Springer International Publishing.
- 7 Luis Neto. Cia: Citizen contact center agent assistant. Master’s thesis, Instituto Superior Técnico, January 2021.
- 8 Vilma Neves. Automatic classification of correspondence from a public institution. Master’s thesis, Instituto Superior Técnico, 2021.
- 9 Sara Silva, Ricardo Ribeiro, and Rúben Pereira. Less is more in incident categorization. In Pedro Rangel Henriques, José Paulo Leal, António Menezes Leitão, and Xavier Gómez Guinovart, editors, *7th Symposium on Languages, Applications and Technologies, SLATE 2018, June 21-22, 2018, Guimarães, Portugal*, volume 62 of *OASICs*, pages 17:1–17:7. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/OASICs.SLATE.2018.17.

9:12 Classification of Public Administration Complaints

- 10 Alberto Simões, Xavier Gómez Guinovart, and José João Almeida. Enriching a portuguese wordnet using synonyms from a monolingual dictionary. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Sara Goggi, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Helene Mazo, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France, May 2016. European Language Resources Association (ELRA).
- 11 Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. Bertimbau: Pretrained bert models for brazilian portuguese. In Ricardo Cerri and Ronaldo C. Prati, editors, *Intelligent Systems*, pages 403–417, Cham, 2020. Springer International Publishing.
- 12 Xiaobo Tang, Hao Mou, Jiangnan Liu, and Xin Du. Research on automatic labeling of imbalanced texts of customer complaints based on text enhancement and layer-by-layer semantic matching. *Scientific Reports*, 11(1):11849, June 2021. doi:10.1038/s41598-021-91189-0.
- 13 Guoyin Wang, Chunyuan Li, Wenlin Wang, Yizhe Zhang, Dinghan Shen, Xinyuan Zhang, Ricardo Henao, and Lawrence Carin. Joint embedding of words and labels for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2321–2331, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi:10.18653/v1/P18-1216.

Comparing Different Approaches for Detecting Hate Speech in Online Portuguese Comments

Bernardo Cunha Matos ✉

INESC-ID Lisboa, Portugal
Instituto Superior Técnico, Lisbon, Portugal

Raquel Bento Santos ✉

INESC-ID Lisboa, Portugal
Instituto Superior Técnico, Lisbon, Portugal

Paula Carvalho ✉ 

INESC-ID Lisboa, Portugal

Ricardo Ribeiro ✉ 

INESC-ID Lisboa, Portugal
Iscte – University Institute of Lisbon, Portugal

Fernando Batista ✉ 

INESC-ID Lisboa, Portugal
Iscte – University Institute of Lisbon, Portugal

Abstract

Online Hate Speech (OHS) has been growing dramatically on social media, which has motivated researchers to develop a diversity of methods for its automated detection. However, the detection of OHS in Portuguese is still little studied. To fill this gap, we explored different models that proved to be successful in the literature to address this task. In particular, we have explored transfer learning approaches, based on existing BERT-like pre-trained models. The performed experiments were based on CO-HATE, a corpus of YouTube comments posted by the Portuguese online community that was manually labeled by different annotators. Among other categories, those comments were labeled regarding the presence of hate speech and the type of hate speech, specifically overt and covert hate speech. We have assessed the impact of using annotations from different annotators on the performance of such models. In addition, we have analyzed the impact of distinguishing overt and covert hate speech. The results achieved show the importance of considering the annotator's profile in the development of hate speech detection models. Regarding the hate speech type, the results obtained do not allow to make any conclusion on what type is easier to detect. Finally, we show that pre-processing does not seem to have a significant impact on the performance of this specific task.

2012 ACM Subject Classification Computing methodologies → Transfer learning; Social and professional topics → Hate speech; Computing methodologies → Supervised learning; Computing methodologies → Machine learning approaches; Information systems → Clustering and classification

Keywords and phrases Hate Speech, Text Classification, Transfer Learning, Supervised Learning, Deep Learning

Digital Object Identifier 10.4230/OASICS.SLATE.2022.10

Supplementary Material *Software (Code, Data and Models)*: https://hate-covid.inesc-id.pt/?page_id=200

Funding This research was supported by Fundação para a Ciência e a Tecnologia, through the Projects HATE Covid-19 (Proj.759274510), MIMU (FCT PTDC/CCI-CIF/32607/2017), and UIDB/50021/2020.



© Bernardo Cunha Matos, Raquel Bento Santos, Paula Carvalho, Ricardo Ribeiro, and Fernando Batista;

licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 10; pp. 10:1–10:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Social media environments are fertile to the dissemination of aggressive and harmful content [12, 18]. Some factors that contribute to this dissemination are the platforms' easy access, the users' potential anonymity, and the increased willingness of people to express their opinions online [8, 12]. The development of methods and tools for automatically detecting offensive and abusive language and hate speech (HS) has recently gained traction in the Natural Language Processing (NLP) and Artificial Intelligence (AI) research communities [8, 21]. The non-existence of a unique and consensual definition of HS [8] makes it difficult to clearly distinguish HS from other related phenomena, such as offensive speech, either for humans or algorithms [14]. For the purpose of this work, HS is defined according through the following coexisting conditions [4]:

- HS has a specific target that can be mentioned explicitly or implicitly, which corresponds to vulnerable or historically marginalized groups or individuals targeted for belonging to those groups;
- HS typically spreads or supports hatred, or incites violence against the targets, by disparaging, humiliating, discriminating, or even threatening them based on specific identity factors (e.g., religion, ethnicity, nationality, race, color, descent, gender, sexual orientation);
- HS can be expressed both explicitly (or overtly) and implicitly (or covertly).

The major difficulty of this task is related to the fact that most of hateful comments in social media are covert or implicit, and their interpretation requires information on the sociopolitical and pragmatic context [3]. Moreover, demographic features such as the first language, age, education, and social identity can result in subjective and biased annotations in the corpora used to both train and test OHS detection systems [1].

Recently, the Commissioner for Human Rights, in a memorandum related to Portugal, has noted that, despite the information being provided by civil society organisations indicates low rates of reporting of HS, there is a rise in the number of racially motivated hate crimes and HS [19]. This highlights the need of monitoring the spread of OHS, which is only possible at a large scale by the use of computational methods.

Despite the great popularity of OHS detection, few studies have specifically dedicated to the analysis and detection of European Portuguese OHS. In fact, there is a lack of resources (particularly annotated corpora) specifically designed to support OHS detection in Portuguese [8, 9].

This work uses the recently created CO-HATE (Counter, Offensive and HS) Corpus [4], which is composed by comments on YouTube videos that tackle topics that could potentially generate hatred content. In particular, this corpus focus on the expression of afrophobia, romaphobia, and LGBTQIphobia by the Portuguese online community, since the Afro-descendant, Roma, and LGBTQI communities are among the most commonly reported targets of both offline and online HS in Portugal [11]. The corpus was labeled by five annotators, who followed detailed guidelines developed for this purpose. As expected, the Inter-Annotator Agreement (IAA) among the annotators is relatively low, achieving a Krippendorff's alpha value of 0.478, which reflects the plurality of subjective views on the HS concept. This motivated us to investigate how does the selection of different perspectives for training affects the results of HS detection.

Recent work on OHS detection relies mostly on the use of Deep Learning (DL) methods for both feature extraction and training of classifiers [2, 10, 13, 20]. As reported in literature [17], the use of models such as Convolutional Neural Networks (CNN) and Long Short-Term

Memory Networks (LSTM), among others, suffers from the lack of labelled data. Transfer learning approaches can overcome this issue since they do not require large amounts of labelled data to train models. Furthermore, they are not so time-consuming, and can outperform all the remaining approaches [17, 22, 25]. In this work, we focus on different transfer learning approaches based on an existing pre-trained model called BERT (Bidirectional Encoder Representations from Transformers) [6]. In particular, we compare the results of three different models – BERT-LinearLayer, BERT-CNN, and GAN-BERT – on the automated detection of HS in Portuguese YouTube comments, based on the CO-HATE corpus. We study the impact of using data annotated by different sets of annotators on the performance of the models and also performed some experiments related to the impact of preprocessing. We also study the differences in detecting Covert and Overt HS.

This document is organised as follows: Section 2 presents an overview of the most relevant related literature; Section 3 describes the dataset used; Section 4 describes the different models; Section 5 presents the experimental setup; Section 6 presents the results; finally, Section 7 presents the major conclusions and pinpoints future directions.

2 Related Work

Recently, HS detection has gained particular relevance in areas such as NLP and AI, and different approaches, mostly relying on deep learning methods, have been proposed in the literature.

Kamble et al. [13] explored HS detection in Hindi-English code-mixed tweets. They developed domain specific word embeddings from 255,309 Hindi-English tweets conveying HS and non-hate. They used the Word2Vec algorithm [16] to train the word embeddings model. Using those embeddings as features, they conducted classification experiments with Deep Learning algorithms such as one-dimensional CNN (CNN-1D), LSTM, and BiLSTM. According to the results reported by the authors, CNN-1D resulted in the highest precision, F1-score, and accuracy, while BiLSTM achieved the best recall. Their models were able to better capture the semantics of HS along with their context which resulted in an improvement of about 12% in F1-score over a past work that used statistical classifiers. For offensive language detection, Ong [20] experimented CNN, LSTM, BiLSTM, GRU, BiGRU, and combinations based on these models. The author used GloVe word vectors, some pre-trained using Twitter with 100 and 200 dimensions and others pre-trained with Common Crawl with 300 dimensions. The author concluded that the architecture that had the highest macro average F1-Score was BiLSTM-CNN.

New architectures, namely the BERT-based ones, seem to outperform all the remaining approaches. Ranasinghe et al. [22] presented a multilingual Deep Learning model to identify HS and offensive language in social media, in their submission to the sub-task A of the HASOC 2019 shared task. To make the system portable to all the languages in the dataset, they used only minimal preprocessing methods, such as removing usernames, removing urls, and, depending on the architecture, converting all tokens into lowercase. They experimented multiple DNNs architectures: pooled GRU, LSTM+GRU+attention, two-dimensional CNN (CNN-2D)+Pooling, GRU+capsule, and LSTM+capsule+attention, using FastText as word embeddings. Furthermore, they also experimented with fine-tuned BERT, which outperformed every above mentioned DNN for German, English, and Hindi. Mozafari et al. [17] used two Twitter datasets that were annotated for racism, sexism, hate, and offensive content. They experimented different combinations of BERT with other models, such as CNN and LSTM. The evaluation results indicated that BERT-CNN outperformed previous works by

10:4 Detecting Online Hate Speech in Portuguese

profiting from the syntactical and contextual information embedded in different transformer encoder layers of the BERT using a CNN-based fine-tuning strategy. In OffensEval 2019, a shared task that focus on the detection of offensive language, Zampieri et al. [25] mentioned that among the top-10 teams, seven used BERT with variations in the parameters and in the preprocessing steps. The top-performing team [15] achieved a macro average F1-Score of 0.829, using pre-trained BERT with fine-tuning on the OLID dataset, and hashtag segmentation and emoji substitution as preprocessing.

An improvement to these BERT-based models was proposed by Croce et al. [5]. The authors implemented GAN-BERT using Generative Adversarial Learning. In this model, the generator is trained to produce a sample, and the discriminator to distinguish between generated samples or samples belonging to the training data. BERT is used to encode the input and as the discriminator. The model was tested with a variety of datasets for multiple tasks (topic classification, question classification, and sentiment analysis) obtaining an increase in performance for all of them when compared to BERT.

With these successful approaches in mind, we will use similar models for the classification of HS in social media text, in particular we will fine-tune BERT, and test combinations of BERT with CNN and GAN.

3 Data

The dataset used in our experiments, the CO-HATE Corpus [4], is composed by the comments retrieved from 39 YouTube videos: 20,590 written comments (795,111 tokens), posted by 8,485 different online users. The corpus was annotated by five recruited annotators, who are currently enrolled in a bachelor's or a master's degree in Communication or in Political and Social Sciences. The average age of the annotators is 23 (ranging from 21 to 27 years old) and the annotation team is composed by both individuals belonging to the communities monitored in this study, and by annotators that do not belong to any potentially marginalized group. More specifically, the annotation team includes Portuguese individuals as follows: a female of African descent, a White male who identifies himself as part of the LGBTQ+ community, a female of Roma descent, a White cisgender hetero male, and a White cisgender hetero female. The corpus is subdivided into five parts, each containing approximately 4,000 messages, on average. Each part was randomly assigned to a different annotator. Additionally, all the annotators were assigned to a common part comprehending 534 messages, which was used to measure the agreement between the annotators, and assess the reliability of the annotations assigned to the entire corpus.

These 534 messages are also used as the test set. Given the task subjectivity, and assuming that the profile of human annotators may influence the data annotation, all the messages labeled as conveying HS by at least two annotators will be considered as hatred content. With this voting type the test set is composed of 50% HS messages. We did not consider the messages containing at least one vote in order to discard unintentional errors introduced by the annotator; the possibility of two annotators making a mistake would be a more unlikely scenario. Table 1 presents the distribution of hatred messages used in training, considering both the annotations performed by each annotator and the ones performed by different groups of annotators that were selected following the criteria defined in Section 5.

4 Model Description

In this section, we describe the models we use for detecting OHS, based on the data previously described. We decided to test such models because they have already shown good performance in similar tasks [5, 17, 23].

■ **Table 1** Class distribution for the training data.

Set of annotators	Number of messages	HS (%)
A+B+C+D+E	20,056	35
A+B+D+E	16,039	37
A+B+C	12,036	30
D+E	8,020	43
A	4,008	25
B	4,011	36
C	4,017	29
D	4,014	39
E	4,006	48

4.1 BERT-LinearLayer

BERT [6] is a multi-layer bidirectional transformer encoder. In our experiments, we used BERT base, which contains an encoder with 12 layers (transformer blocks), 12 self-attention heads, and 110 million parameters. As the BERT model is pre-trained on general corpora, we had to fine-tune BERT using our annotated dataset. BERT-LinearLayer is inspired in [23] and [17]. In this architecture, the [CLS] token output of the 12th transformer encoder, a vector of size 768, is given as input to a fully connected network W/o hidden layer. Then, the Sigmoid activation function is applied to the hidden layer in order to make the prediction.

4.2 BERT-CNN

This model is inspired in [23] and consists of two main components. The first one is the BERT model, in which the text is passed through 12 layers of self-attention to obtain contextualized vector representations. The other one is a CNN, which is used as a classifier.

First, the text is given as input to BERT, then the output of the last four hidden layers of the pre-trained BERT are concatenated to get vector representations. Next, these embeddings are passed in parallel into 160 convolutional filters of five different sizes (768x1, 768x2, 768x3, 768x4, and 768x5), 32 filters for each size. Each kernel takes the output of the last four hidden layers of BERT as 4 different channels and applies the convolution operation on it. After that, the output is passed through the ReLU Activation function and a Global Max-Pooling operation. Finally, the output of the pooling operation is concatenated and flattened to be later on passed through a dense layer and a Sigmoid function to get the final binary label.

4.3 GAN-BERT

The GAN-BERT approach is based on GAN-BERT [5]. The input is encoded by a BERT model. The GAN is composed by a generator, trained to produce a sample, and a discriminator to distinguish between generated samples or samples belonging to the training data. The generator is a multi-layer perceptron (MLP) that transforms an input into a vector representation being the [CLS] token used as a sentence embedding. The discriminator is another MLP with a last layer with SoftMax as an activation function to classify the received embedding. The training process consists of optimizing both generator and discriminator losses. The generator loss considers the error induced by the generated examples correctly identified by the discriminator. The discriminator loss considers the error induced by wrongly

classifying the labeled data and by not being able to recognize generated samples. The BERT weights will be updated when updating the discriminator. After training, the generator is discarded.

5 Experimental setup

The maximum sequence length of each text sample was set to 350 tokens to avoid overloading the GPU. A substantial amount of messages does not exceed that length and it does not degrade performance. All the models were trained for 15 epochs and the model with the best positive class F1-Score on the development set was saved. The training data was split into 80% and 20% for training and development sets, respectively, preserving the same proportions of examples in each class.

The first experiment was done using the entire training data, i.e., the 20,056 messages annotated by annotators A, B, C, D, and E. Considering the subjectivity of this topic, we have also experimented using the corpus annotated by each user independently (A, B, C, D, and E). Since annotator C was the one having the worse IAA, when compared to the remaining annotators (0.23 on average, while the others have at least 0.531), we tested the combination A+B+D+E. Also, the annotators D and E were the ones that achieved better results independently and the highest agreement rate between them, so we also have tested the combination D+E. Since annotators D and E do not belong to any potential historically marginalized group, we decided to test also the combination A+B+C, composed by annotators that belong to the target communities. This may help us to understand how the annotators' social identity may affect the performance of OHS detection.

For all models, we used two different pre-trained BERT models, namely **Multilingual BERT** (mBERT),¹ and **BERTimbau** (brBERT) [24]. When using the entire corpus, BERTimbau had an overall performance better than mBERT, so for all the other experiments we only tested with BERTimbau and we only report the results obtained using it.

We report both macro and positive class F1-scores, but when assessing the models performance we give particular importance to the positive class F1-score, since it evaluates the models performance on the class that we want to detect.

6 Results and Analysis

The results of all models for the different sets of training data are represented in Table 2. In these experiments, the text was not pre-processed and in the test set we considered a minimum of two votes to decide if a comment contains or not HS. We present the results of our baseline, a dummy classifier that classifies all instances as HS. Considering the positive class F1-score as the benchmark metric, we can see that using the data of A, B, C, and A+B+C always obtains worse results than the baseline model. Also, when using the data of all annotators, A+B+C+D+E, the best result (Positive Class F1-Score of 0.667) was not able to outperform the baseline model. On the other hand, when using the data of D, E, D+E, and A+B+D+E, the results outperformed the baseline.

Comparing the different model architectures, the results suggest that BERT-LinearLayer and BERT-CNN can attain better results than GAN-BERT. In particular, the best result was obtained using the data of D+E and the BERT-CNN model with an F1-score of 0.721.

¹ <https://github.com/google-research/bert/blob/master/multilingual.md>

■ **Table 2** Performance of all models with different data used for train.

Training Data	Model	Positive Class			Macro Avg		
		Prec	Rec	F1	Prec	Rec	F1
	Baseline	0.500	1	0.667	0.250	0.500	0.333
A	BERT-LinearLayer	0.604	0.629	0.617	0.609	0.609	0.608
	BERT-CNN	0.601	0.648	0.623	0.609	0.609	0.608
	GAN-BERT	0.578	0.390	0.465	0.559	0.552	0.540
B	BERT-LinearLayer	0.721	0.532	0.612	0.675	0.663	0.657
	BERT-CNN	0.695	0.581	0.633	0.667	0.663	0.661
	GAN-BERT	0.625	0.468	0.535	0.600	0.594	0.587
C	BERT-LinearLayer	0.611	0.629	0.620	0.614	0.614	0.614
	BERT-CNN	0.649	0.547	0.593	0.629	0.625	0.623
	GAN-BERT	0.615	0.562	0.587	0.606	0.605	0.604
D	BERT-LinearLayer	0.603	0.809	0.691	0.657	0.639	0.628
	BERT-CNN	0.589	0.790	0.675	0.636	0.620	0.608
	GAN-BERT	0.623	0.682	0.651	0.636	0.635	0.634
E	BERT-LinearLayer	0.665	0.663	0.664	0.665	0.665	0.665
	BERT-CNN	0.702	0.697	0.699	0.700	0.700	0.700
	GAN-BERT	0.631	0.633	0.632	0.631	0.631	0.631
D+E	BERT-LinearLayer	0.645	0.768	0.701	0.679	0.672	0.669
	BERT-CNN	0.625	0.850	0.721	0.696	0.670	0.659
	GAN-BERT	0.630	0.682	0.655	0.641	0.640	0.640
A+B+C	BERT-LinearLayer	0.649	0.610	0.629	0.641	0.640	0.640
	BERT-CNN	0.683	0.614	0.647	0.666	0.665	0.664
	GAN-BERT	0.618	0.442	0.515	0.592	0.584	0.576
A+B+D+E	BERT-LinearLayer	0.712	0.622	0.664	0.688	0.685	0.684
	BERT-CNN	0.679	0.697	0.688	0.684	0.684	0.683
	GAN-BERT	0.656	0.622	0.638	0.648	0.648	0.648
A+B+C+D+E	BERT-LinearLayer	0.636	0.700	0.667	0.651	0.650	0.649
	BERT-CNN	0.688	0.618	0.651	0.670	0.669	0.668
	GAN-BERT	0.648	0.509	0.570	0.622	0.616	0.612

Intuitively, it was expected that BERT-CNN would yield better results, since it uses information that contains both syntactical and contextual features from the last four layers of BERT, which encode more information than the output of the top layer [6], while BERT-LinearLayer and GAN-BERT models only use the [CLS] token of the last transformer encoder. We also believe that the convolutions of the CNN architecture highlighted features related to different writing patterns, which may occur in HS samples, and therefore provide better detection for the aforementioned category.

The low IAA obtained between all the annotators, a Krippendorff’s alpha of 0.478, suggested the difficulty of this task even for humans. Nevertheless, reasonable results were obtained when considering the multiplicity of perspectives given by all the annotators: a Positive Class F1-Score of 0.667 (and a macro averaged F1-score of 0.649, considerably better than the one achieved by the baseline). The annotators belonging to the communities targeted in this study (A, B, and C) tended to disagree more with each other than the annotators not belonging to these communities (D and E) [4]. This aspect is also reflected in the models trained with the information provided by such annotators. Every single Positive Class F1-Score obtained by D+E is greater than the Positive Class F1-Scores obtained by A+B+C.

actual values	Non-HS	131	136
	HS	40	227
		Non-HS	HS
		predicted values	

■ **Figure 1** Confusion matrix of the best performing model: BERT-CNN with D+E training data.

We also find important to put into perspective the results we had with the results reported by other researchers on the same task. We did not find any works using Portuguese pre-trained BERT models, so we had to compare our results with works focusing on other languages. Safaya et. al [23] achieved, on average, a macro average F1-score of 0.851 with their BERT-CNN model, using language-specific pre-trained models for Arabic, Greek, and Turkish languages. Although we did not test all the architectures they have tested, just like them, in our work, BERT-CNN yielded better results than using BERT-LinearLayer. Dowlagar et. al [7] achieved a macro average F1-score of 0.883 by fine-tuning an English-specific pre-trained BERT model, the biggest macro F1-score when compared to other machine learning approaches. To the best of our knowledge, no experiences were performed in the supervised case using GAN-BERT. The best macro average F1-score we had was 0.7, which is not in line with the values that we have now reported. Several factors may help understanding this difference. For example, most corpora used by researchers are often created through the use of generic lexical-based approaches, used to retrieve content containing words or expressions with negative polarity. This selection method leaves out an immense set of potentially relevant hatred content, including covert (indirect or implicit) HS, often resorting to rhetorical figures, and apparently neutral words and constructions used to attack or humiliate the HS targets. CO-HATE data was selected using a different approach, allowing the inclusion of messages conveying either overt or covert HS. In fact, covert HS surpasses the frequency of overt HS in this corpus. Given covert HS is much harder to detect than overt HS, we believe that this may rend the task more difficult to the classifiers.

The use of BERTimbau, a Brazilian Portuguese model trained on the BrWaC (Brazilian Web as Corpus), with European Portuguese text extracted from the YouTube platform may also influence the results, since there are lexical differences between these variants of Portuguese that are not covered by BERTimbau Tokenizer. More general aspects such as hyper-parameter optimization can also impact the performance. For instance, varying values of batch size, learning rate, dropout rate (when applicable) and different max length values for text sequences are all variables that may affect the results.

6.1 Error Analysis

This section discusses the major classification errors derived from our best model, BERT-CNN, trained with the data from annotators D and E. In Figure 1, we present the confusion matrix of our best model in order to better visualize the classification errors. As we had already shown in the Table 2, we have interesting results in terms of recall, but the precision is not at the same level. We have inspected the 136 messages that were incorrectly classified as HS and found that a high proportion contain counter-speech. Hence, this suggest that future experiments should include other related HS categories, such as 'Counter-speech', instead of considering only a dichotomous classification. Also, some of these messages include words and expressions that are also highly frequently used in messages classified as conveying hate speech (e.g., *Rendimento Social de Inserção (RSI)*, *abonos*, and *subsidiodependência*; “Social Integration Income”, “subsidies”, and “subsidy dependence”), which may influence

the classifier in the training phase. We have also manually inspected the 40 messages that were incorrectly classified as Non-HS. This analysis suggest that most errors are associated with messages that, out-of-context, are difficult to interpret, as illustrated in Examples 1-4.

- (1) *Adoro Portugal*
I love Portugal
- (2) *Sim :)*
Yes :)
- (3) *Correto !*
Correct !
- (4) @João Francisco 🍌🍌🍌

6.2 Detecting Overt and Covert HS

We conducted additional experiments on the identification of Overt HS and Covert HS. This is particularly relevant because Covert HS is quite frequent in our data, even more frequent than Overt HS. In fact, our training data contains about 16% of Overt HS and 19% of Covert HS, while the test set contains about 22% of Overt HS and 34% of Covert HS. All the experiments were conducted using BERT-CNN, the model that achieved the best performance in the binary classification of HS, and all the training data were used for trained. The results are presented in Tables 3 and 4, where the baseline consists of classifying all instances as the positive class. When considering the positive class F1-Score, the detection of Overt HS (0.495) was more successful than the detection of Covert HS (0.488). This is an expected result, since covert HS usually tends to use linguistic phenomena, such as sarcasm or irony, much more difficult to detect with an automatic approach. In fact, the lower frequency of Overt HS in the data makes the detection of Overt HS more challenging due to the unbalanced data, but our model was able to still attain a good performance, specially considering the baseline. The detection of Covert HS proved to be difficult to perform using the proposed model.

■ **Table 3** Classification of Overt Hate Speech, using the BERT-CNN model trained with all data.

Model	Overt HS			Macro Avg		
	Prec	Rec	F1	Prec	Rec	F1
Baseline	0.221	1	0.362	0.110	0.500	0.181
BERT-CNN	0.580	0.432	0.495	0.715	0.672	0.687

■ **Table 4** Classification of Covert Hate Speech, using the BERT-CNN model trained with all data.

Model	Covert HS			Macro Avg		
	Prec	Rec	F1	Prec	Rec	F1
Baseline	0.335	1	0.502	0.168	0.500	0.251
BERT-CNN	0.648	0.391	0.488	0.696	0.642	0.650

6.3 Pre-processing Impact

We tested the impact of applying pre-processing, which consisted of removing processing errors generated in the data retrieval; anonymizing users' mentions by replacing a user tag with "@UserID" to represent a username with a single word and keep the context and

10:10 Detecting Online Hate Speech in Portuguese

removing repetitions of three or more punctuation signals and emojis. We compiled in Table 5 the best result of each model for each case. We were expecting that linguistic clues just like the repetitions of punctuation signals and emojis could aid the models in capturing HS but their removal accompanied by some other pre-processing techniques led to better results for BERT-LinearLayer and BERT-CNN models, even though they are minor improvements (+0.003 for BERT-LinearLayer and +0.005 for BERT-CNN). On the other hand, the results of GAN-BERT met our expectations and got worsen (-0.008), but again the differences were not significant.

■ **Table 5** Positive Class F1-Scores of the best models on the test data with and without pre-processing.

Model	W/o preproc	W/ preproc
BERT-LinearLayer	0.701	0.704
BERT-CNN	0.721	0.726
GAN-BERT	0.655	0.647

7 Conclusion

In this paper, we compared three different models based on BERT on the task of identifying HS in Portuguese YouTube comments: BERT-LinearLayer, BERT-CNN, and GAN-BERT. BERT-CNN achieved the highest Positive Class F1-Score, taking advantage of both the syntactical and contextual information embedded in the last four transformer encoder layers of the BERT model and the convolutions of the CNN architecture. We have shown how the data chosen to train the models might affect the results. Our best model, BERT-CNN, achieves a Positive Class F1-Score of 0.72, surpassing the baseline in almost 6 p.p. The best result was obtained by combining the data of the annotations from the annotators with the best IAA, which suggests that if we intent to have a good performance using data annotated by different annotators, we should consider the aggregation of annotations from annotators sharing similar perceptions. Although our best macro average F1-score was reasonable, it was not at the same level of other HS detection works that involve other languages. Factors such as the high frequency of Covert HS in our dataset, the use of BERTimbau with European Portuguese, and hyper-parameter optimization may explain these differences in the performance. We performed an Error Analysis of the results provided by our best model and found that some of the messages misclassified as Non-HS do not provide the necessary context for a good classification. Also, a high proportion of messages misclassified as HS correspond, in fact, to counter-speech, and contain words that are frequently used in hatred content. We conducted some experiments for the binary classification of Overt HS and the binary classification of Covert HS in order to investigate if the HS type could affect the performance of our approaches. As expected, the detection of Covert HS proved to be slightly harder than the detection of Overt HS. We also tested the impact of pre-processing the text, and the results show that pre-processing does not seem to have a significant impact on the performance of this specific task.

References

- 1 Hala Al Kuwatly, Maximilian Wich, and Georg Groh. Identifying and measuring annotator bias based on annotators' demographic characteristics. In *Proceedings of the Fourth Workshop on Online Abuse and Harms*, pages 184–190, Online, November 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.a1w-1.21.
- 2 Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep learning for hate speech detection in tweets. In *Proceedings of the 26th international conference on World Wide Web companion*, pages 759–760, 2017.
- 3 Fabienne Baidier. Covert hate speech, conspiracy theory and anti-semitism: Linguistic analysis versus legal judgement. *International Journal for the Semiotics of Law-Revue internationale de Sémiotique juridique*, pages 1–25, 2022.
- 4 Paula Carvalho, Danielle Caled, Cláudia Silva, Fernando Batista, and Ricardo Ribeiro. The expression of Hate Speech against Afro-descendant, Roma and LGBTQ+ communities in YouTube comments. *submitted*, 2022.
- 5 Danilo Croce, Giuseppe Castellucci, and Roberto Basili. GAN-BERT: Generative adversarial learning for robust text classification with a bunch of labeled examples. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2114–2119, Online, July 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.acl-main.191.
- 6 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1423.
- 7 Suman Dowlagar and Radhika Mamidi. Hasocone@fire-hasoc2020: Using BERT and multilingual BERT models for hate speech detection. *CoRR*, abs/2101.09007, 2021. arXiv:2101.09007.
- 8 Paula Fortuna and Sérgio Nunes. A survey on automatic detection of hate speech in text. *ACM Computing Surveys (CSUR)*, 51(4):1–30, 2018.
- 9 Paula Fortuna, João Rocha da Silva, Juan Soler-Company, Leo Wanner, and Sérgio Nunes. A hierarchically-labeled Portuguese hate speech dataset. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 94–104, Florence, Italy, August 2019. Association for Computational Linguistics. doi:10.18653/v1/W19-3510.
- 10 Björn Gambäck and Utpal Kumar Sikdar. Using convolutional neural networks to classify hate-speech. In *Proceedings of the first workshop on abusive language online*, pages 85–90, 2017.
- 11 European University Institute. *Monitoring media pluralism in the digital era: application of the media pluralism monitor in the European Union, Albania and Turkey in the years 2018 2019: country report Portugal*. Publications Office, 2020. doi:10.2870/292300.
- 12 Md Saroar Jahan and Mourad Oussalah. A systematic review of hate speech automatic detection using natural language processing. *arXiv preprint*, 2021. arXiv:2106.00742.
- 13 Satyajit Kamble and Aditya Joshi. Hate speech detection from code-mixed hindi-english tweets using deep learning models. *arXiv preprint*, 2018. arXiv:1811.05145.
- 14 György Kovács, Pedro Alonso, and Rajkumar Saini. Challenges of hate speech detection in social media. *SN Computer Science*, 2(2), February 2021. doi:10.1007/s42979-021-00457-3.
- 15 Ping Liu, Wen Li, and Liang Zou. Nuli at semeval-2019 task 6: Transfer learning for offensive language detection using bidirectional transformers. In *Proceedings of the 13th international workshop on semantic evaluation*, pages 87–91, 2019.
- 16 Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. doi:10.48550/ARXIV.1301.3781.
- 17 Marzieh Mozafari, Reza Farahbakhsh, and Noel Crespi. A bert-based transfer learning approach for hate speech detection in online social media. In *International Conference on Complex Networks and Their Applications*, pages 928–940. Springer, 2019.

10:12 Detecting Online Hate Speech in Portuguese

- 18 Karsten Müller and Carlo Schwarz. Fanning the flames of hate: Social media and hate crime. *Journal of the European Economic Association*, 19(4):2131–2167, 2021.
- 19 Council of Europe. Portugal should act more resolutely to tackle racism and continue efforts to combat violence against women. <https://www.coe.int/en/web/commissioner/-/portugal-should-act-more-resolutely-to-tackle-racism-and-continue-efforts-to-combat-violence-against-women>, June 2021.
- 20 Ryan Ong. Offensive language analysis using deep learning architecture. *arXiv preprint*, 2019. [arXiv:1903.05280](https://arxiv.org/abs/1903.05280).
- 21 Fabio Poletto, Valerio Basile, Manuela Sanguinetti, Cristina Bosco, and Viviana Patti. Resources and benchmark corpora for hate speech detection: a systematic review. *Language Resources and Evaluation*, 55(2):477–523, 2021.
- 22 Tharindu Ranasinghe, Marcos Zampieri, and Hansi Hettiarachchi. Brums at hasoc 2019: Deep learning models for multilingual hate speech and offensive language identification. In *FIRE (Working Notes)*, pages 199–207, 2019.
- 23 Ali Safaya, Moutasem Abdullatif, and Deniz Yuret. KUISAIL at SemEval-2020 task 12: BERT-CNN for offensive speech identification in social media. In *Proceedings of the Fourteenth Workshop on Semantic Evaluation*, pages 2054–2059, Barcelona (online), December 2020. International Committee for Computational Linguistics. URL: <https://www.aclweb.org/anthology/2020.semeval-1.271>.
- 24 Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. Bertimbau: pretrained bert models for brazilian portuguese. In *Brazilian Conference on Intelligent Systems*, pages 403–417. Springer, 2020.
- 25 Marcos Zampieri, Shervin Malmasi, Preslav Nakov, Sara Rosenthal, Noura Farra, and Ritesh Kumar. Semeval-2019 task 6: Identifying and categorizing offensive language in social media (offenseval). *arXiv preprint*, 2019. [arXiv:1903.08983](https://arxiv.org/abs/1903.08983).

Semi-Supervised Annotation of Portuguese Hate Speech Across Social Media Domains

Raquel Bento Santos ✉

INESC-ID Lisbon, Portugal

Instituto Superior Técnico, Lisbon, Portugal

Bernardo Cunha Matos ✉

INESC-ID Lisbon, Portugal

Instituto Superior Técnico, Lisbon, Portugal

Paula Carvalho ✉ 

INESC-ID Lisbon, Portugal

Fernando Batista ✉ 

INESC-ID Lisbon, Portugal

Iscte – University Institute of Lisbon, Portugal

Ricardo Ribeiro ✉ 

INESC-ID Lisbon, Portugal

Iscte – University Institute of Lisbon, Portugal

Abstract

With the increasing spread of hate speech (HS) on social media, it becomes urgent to develop models that can help detecting it automatically. Typically, such models require large-scale annotated corpora, which are still scarce in languages such as Portuguese. However, creating manually annotated corpora is a very expensive and time-consuming task. To address this problem, we propose an ensemble of two semi-supervised models that can be used to automatically create a corpus representative of online hate speech in Portuguese. The first model combines Generative Adversarial Networks and a BERT-based model. The second model is based on label propagation, and consists of propagating labels from existing annotated corpora to the unlabeled data, by exploring the notion of similarity. We have explored the annotations of three existing corpora (CO-HATE, ToLR-BR, and HPHS) in order to automatically annotate FIGHT, a corpus composed of geolocated tweets produced in the Portuguese territory. Through the process of selecting the best model and the corresponding setup, we have tested different pre-trained embeddings, performed experiments using different training subsets, labeled by different annotators with different perspectives, and performed several experiments with active learning. Furthermore, this work explores back translation as a mean to automatically generate additional hate speech samples. The best results were achieved by combining all the labeled datasets, obtaining 0.664 F1-score for the *Hate Speech* class in FIGHT.

2012 ACM Subject Classification Social and professional topics → Hate speech; Theory of computation → Semi-supervised learning; Computing methodologies → Transfer learning

Keywords and phrases Hate Speech, Semi-Supervised Learning, Semi-Automatic Annotation

Digital Object Identifier 10.4230/OASICS.SLATE.2022.11

Supplementary Material *Software (Code, Data and Models)*: https://hate-covid.inesc-id.pt/?page_id=200

Funding This research was supported by Fundação para a Ciência e a Tecnologia, through the Projects HATE Covid-19 (Proj.759274510), MIMU (FCT PTDC/CCI-CIF/32607/2017), and UIDB/50021/2020.



© Raquel Bento Santos, Bernardo Cunha Matos, Paula Carvalho, Fernando Batista, and Ricardo Ribeiro;

licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 11; pp. 11:1–11:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A growing number of people have reported that have already been exposed to hate speech on social media [29]. Due to the anonymity allowed on the Internet, people feel more ease at expressing themselves and engaging in hostile behaviors [12]. Therefore, it urges to develop models able to detect online hate speech automatically.

The non-existence of a unique and consensual definition of hate speech [12] makes its detection more difficult, either for humans or algorithms. For the purpose of this work, hate speech is defined according to the following coexisting conditions [6]:

- Hate speech has a specific target that can be mentioned explicitly or implicitly, which corresponds to vulnerable or historically marginalized groups or individuals targeted for belonging to those groups;
- Hate speech typically spreads or supports hatred, or incites violence against the targets, by disparaging, humiliating, discriminating, or even threatening them based on specific identity factors (e.g., religion, ethnicity, nationality, race, color, descent, gender, sexual orientation);
- Hate speech can be expressed both explicitly (or overtly) and implicitly (or covertly).

Most hateful comments are implicit in text, making use of several rhetorical strategies, such as irony and rhetorical questions [33], turning them even harder to identify. Furthermore, hate speech is often context-dependent, meaning that specific words or expressions may have different interpretations, depending on the linguistic and pragmatic context where they are used [16]. Moreover, the personal experiences, knowledge, and beliefs of the ones studying it, as well as demographic features such as the first language, age, education, and social identity, can also introduce personal bias into the classification process [1, 32].

Robust models typically rely on large-scale annotated language resources, which have been created following different annotation guidelines. However, the existing resources – mostly for English – cannot be easily transferred to other languages due the linguistic disparities even within the same language, and the multiplicity of hate speech targets being considered in those studies [27]. Even within the same language, models tend to have generalization problems, dropping in performance when applied to a distinct dataset [34]. Besides, with few exceptions, existing corpora do not usually cover implicit hate speech [3, 15, 17]. In fact, the data comprising Hate Speech (HS) corpora is often retrieved by using negative polarity words and expressions, which are not usually found in implicit hate speech. In addition, most corpora available are imbalanced, and the majority class often correspond to neutral speech, i.e., not offensive nor hateful speech. This asymmetry may deteriorate the performance of the classification models.

Being aware that creating manually annotated corpora is a very time-consuming and expensive task, requiring linguistic and pragmatic knowledge, we propose an ensemble of two semi-supervised models to create annotated corpora representative of the hate speech present on social media platforms in Portugal. The first model combines Generative Adversarial Networks and a BERT-based model. The second one is based on label propagation, assigning labels to the unlabeled data based on their similarity with the annotated corpus. Both models are combined in a semi-supervised self-training approach to obtain an automatically annotated corpus.

The rest of this document is organized as follows. Section 2 presents the related work, focusing particularly on the hate speech datasets available for Portuguese and the most relevant semi-supervised learning models for this task. Section 3 describes the datasets used in the experiments performed, and Section 4 describes our model and the pre-processing applied to the corpora. Section 5 presents the results, and, finally, Section 6 highlights the main conclusions.

2 Related Work

Hate Speech in social media is a recent research topic that has been evolving with the increased use of these platforms. To the best of our knowledge, there are only two datasets covering Portuguese hate speech publicly available.¹

Leite et al. [18] developed ToLR-BR, a corpus composed of 21,000 tweets, retrieved by applying a list of offensive keywords and considering keywords related to influential Brazilian users that could be targets of hate speech or abuse. The messages were classified as *Homophobia*, *Obscene*, *Insult*, *Racism*, *Misogyny*, and *Xenophobia* by three annotators. Around 44% of the messages were classified as offensive by at least one annotator, 21% by two, and 7% by the three annotators.

Fortuna et al. [13] presented a Hierarchically-Labeled Portuguese Hate Speech Dataset (HPHS) of 5,670 Brazilian Portuguese tweets from 115 users. The messages were retrieved using a list of offensive keywords and by considering users that typically post hateful comments. The tweets were manually classified by three annotators in a binary scheme (hate speech or not). The hatred messages were then classified according to their target, following a hierarchical scheme including 81 hate speech categories. Around 22% of the tweets correspond to hate speech.

Given this lack of resources, semi-supervised learning surges as a solution for hate speech classification. This approach considers a small amount of labeled data and makes use of a large amount of unlabeled data.

Alsafari and Sadaouia [2] use semi-supervised self-training to classify Arabic tweets in *Clean* or *Offensive/Hate*. This approach consists of re-applying the classifier to its most confident predictions [31]. To ensure a good learning ability and good performance, it is required a sufficiently large initial training dataset [2] considering that the performance depends on the accuracy of the pseudo-labels [19]. The tweets are represented with Word2Vec SkipGram embeddings to capture their semantic and syntactic information. The model consists of one classifier based on N-Grams and two deep neural network classifiers. The authors performed multiple experiments with Support Vector Machines (SVM), Convolutional Neural Networks (CNN), AraBERT, and DistilBERT. The classifiers were evaluated according to their accuracy, model size, and inference speed, being the best results achieved by the CNN approach. This model was then used to perform fifteen iterations reusing the predictions with higher confidence. AraBERT and DistilBERT were not used due to their complexity. With the increase in the number of iterations, the model started to associate a hashtag with the tag *Offensive/Hate* so hashtags were ignored. However, the models still perform poorly when classifying implicit hate and in the presence of rare terms. Besides, tweets with counterspeech and abusive words are wrongly classified as *Offensive/Hate*. As expected, the authors also show that increasing the size of the labeled dataset led to a performance increase.

Croce et al. [8] propose GAN-BERT. In Generative Adversarial Learning (GAN), the generator is trained to produce a sample and the discriminator to distinguish between generated samples or samples belonging to the training data. With Semi-Supervised Generative Adversarial Networks (SS-GAN), the discriminator will also classify the sample. BERT is used to encode the input and as the discriminator. The generator is a multi-layer perceptron that transforms an input into a vector representation being the [CLS] token used as a sentence embedding. The discriminator is another multi-layer perceptron with a last layer with

¹ <https://hatespeechdata.com/>

SoftMax as an activation function to classify the received embedding. The training process consists of optimizing both generator and discriminator losses. The generator loss considers the error induced by the generated examples correctly identified by the discriminator. The discriminator loss considers the error induced by wrongly classifying the labeled data and by not being able to recognize generated samples. The BERT weights will be updated when updating the discriminator. After training, the generator is discarded. The model was tested with a variety of datasets for multiple tasks obtaining an increase in performance for all of them when compared to BERT. Furthermore, the authors have proved that less than 200 annotated examples obtain similar results to the supervised approach. More recently, Breazzano et al. [5] extended this model to multi-task learning and applied it to hate speech classification with similar performance.

D'Sa et al. [10] represent tweets as a pre-trained sentence embedding, using the Universal Sentence Encoder (USE). The authors use a Multilayer Perceptron (MLP) to transform this generic representation into a task-specific representation using a small amount of labeled data. After training with the labeled data, the MLP classifier receives as input the pre-trained representations of a labeled sample and an unlabeled sample. The outputs of the activation function of the two hidden layers correspond to two different task-specific representations. Then, label propagation is performed to obtain the labels for the unlabeled sample. Label propagation is a graph-based semi-supervised technique where the data is represented as a graph. The vertices correspond to the data points and the edges represent the similarity between two nodes. The data points close to each other tend to have a similar label so, the labels are propagated from the labeled points to the unlabeled ones [10, 21]. Finally, the pre-trained embeddings and the labels are used to train the MLP classifier. Comparatively to the MLP classifier trained only with the labeled set and without label propagation, training using label propagation on pre-trained representations performs worse. However, the two representations from the hidden layers capture class information and have better results. In some cases, the label propagation using the representation after the first hidden layer performed better so fully fine-tuned representation may not always be the best approach.

Considering that most of the interactions present in social media do not correspond to hate speech, and given the difficulty to extract them, the percentage of hate speech present in hate speech corpora is low (around 8%). Data augmentation allows to expand an existing training dataset by implementing transformations to the already labeled data or by creating synthetic examples from this data [19, 22]. This can reduce the data scarcity by generating new instances for the minority classes [1], balancing the dataset labels, and reducing the overfit [28]. It can also help the model to better generalize to unseen data, increasing its overall performance [19]. However, data augmentation in NLP tasks is limited since most operations can distort the meaning of the sentence and the number of synonyms of a word is not very high. Considering these limitations, we opted to use back translation since the paraphrases generated by this approach tend to preserve the semantics of the message [4].

This work will follow a self-training approach with an ensemble of two models to reduce the bias of each one. Considering the good results of the previous two models, our proposal will consist of an adaptation of both.

3 Data

We use CO-HATE [6] and FIGHT [7], two corpora recently created from Portuguese online data containing potential hate speech. CO-HATE is composed of comments retrieved from YouTube, and has been manually annotated. FIGHT is composed of tweets, lacks from

annotations, and our goal is to provide such annotations. Two additional datasets, focusing on Brazilian Portuguese, were also considered as additional labeled sources, as described in Section 3.3.

3.1 CO-HATE Corpus

The CO-HATE (**C**ounter, **O**ffensive and **H**ate speech) corpus [6] is composed of 20,590 written messages, posted by 8,485 different online users on 39 YouTube videos covering topics and events targeting, directly or indirectly, three specific focus groups: African descent, Roma, and the LGBTQ+ communities. The first two communities correspond to the most representative minorities in Portugal. The LGBTQI community was reported as the most targeted group in terms of online hate speech [11, 23, 25]. The CO-HATE corpus was manually annotated by five annotators, each being responsible for annotating approximately 4,000 messages. Additionally, all annotators were assigned to a common part consisting of 534 messages to assess the inter-annotator agreement (IAA) and the reliability of the annotations.

The annotators are currently enrolled in a bachelor’s or a master’s degree in Communication or in Political and Social Sciences. The average age of the annotators is 23 (ranging from 21 to 27 years old), and three annotators are female. The annotators A, B and C belong to the communities monitored in this study. More specifically, the annotation team includes Portuguese individuals as follows: a female of African descent, a White male who identifies himself as part of the LGBTQ+ community, a female of Roma descent, a White cisgender hetero male, and a White cisgender hetero female [6].

The final labels for the messages are obtained considering the majority of the annotations. The IAA (using Krippendorff’s alpha) between all the annotators was considerably low (0.478), despite providing the annotators with detailed guidelines. This demonstrates the subjectivity (and difficulty) of this task, even for humans [6]. Table 1 shows the percentage of messages classified as conveying hate speech by each annotator individually, and the group of annotators (ABCDE).

■ **Table 1** Proportion of messages containing hate speech in CO-HATE corpus, by annotator.

Annotators	Number of messages	HS (%)
A	4,008	25
B	4,011	36
C	4,017	29
D	4,014	39
E	4,006	48
Total	20,590	35

3.2 FIGHT Corpus

The FIGHT (**F**Indin**G** **H**ate Speech in **T**witter) corpus [7] is composed of 56,546 geolocated tweets in the Portuguese territory. This corpus was obtained with two retrieval methods: selecting tweets that include non-ambiguous words that may be used to mention one of the aforementioned target groups (54,352 tweets); and selecting tweets containing a potential mention to the target group, and at least one offensive or insulting word or expression (9,796 tweets). The second approach prevents from retrieving a multiplicity of hate speech forms,

including implicit or covert hate speech, but allows retrieving potential offensive or hatred content [7]. In order to evaluate the performance of our models, we have manually annotated a sample of 300 tweets, which is used as our test set.

3.3 Additional Datasets

Since the previously mentioned corpora are focused only on three specific hate speech targets, we decided to consider two additional hate speech Brazilian Portuguese datasets, ToLR-BR and HPHS, covering other HS targets. Taking into account the subjectivity of this task and the personal bias that can be introduced in the annotation process, only the messages labeled as hate speech by the majority of the annotators will be considered as such in order to select only clear cases of hate speech and considering that it is the standard approach in the literature. Regarding ToLR-BR corpus [18], we assumed as hate speech all tweets with one of the labels *Homophobia*, *Racism*, *Misogyny*, and *Xenophobia* given by the majority of the annotators. Of the 21,000 tweets, 403 were classified as hate speech. From these, 192 correspond to *Homophobia*, 96 correspond to *Racism*, 158 to *Misogyny* and 60 to *Xenophobia*. For the HPHS dataset [13], we considered as hate speech the tweets classified as *Hate Speech* by at least two out of the three annotators. From the 5,670 tweets, 1,788 correspond to hate speech.

4 Modeling Approaches

The goal of this work is to present a model capable of automatically classifying hate speech, aiming at contributing to solve the scarcity of annotated hate speech corpora in Portuguese. The model should be able to transfer knowledge from the CO-Hate corpus in order to annotate the FIGHT corpus. This is a particularly complex task considering the different nature of the two corpora. While CO-Hate is composed by YouTube comments contextualized by the videos, FIGHT is composed by individual tweets that are published without a context. Besides, YouTube comments can have an arbitrary size while tweets are limited to 280 characters.

The proposed model corresponds to an ensemble of two semi-supervised models, to reduce the bias of each model. The first model combines Generative Adversarial Networks and a BERT-based model, based on GAN-BERT [8]. The goal is to find the distribution of classes for the labeled data and update it with the unlabeled data. The second model, label propagation, uses the similarities between the instances (points) of the datasets to propagate the existing labels to the unlabeled data. The label of a given point is determined by the labels of the closest points (the implementation used the scikit-learn library [24]). Both models have been recently tested for the hate speech domain, obtaining performance improvements when compared to other previously developed models [5, 10].

Both classifiers are trained with a sample of labeled data. Then, at each iteration, both classifiers classify a subset of unlabeled data. The most confident predictions are added to the labeled set and the models are fine-tuned with them. The maximum sequence length of each message was defined as 350 tokens to ensure the efficiency of the model without losing too much information. The GAN-BERT model was trained for 15 epochs with 5 patience, considering the model with the best F1-score for the positive class. The training data was randomly split into 80% for the train set and 20% for the development set. The label propagation model used the k -nearest neighbors algorithm with a maximum of 100 iterations and neighbors between 3 and 50.

Considering the GAN-BERT model, we fine-tuned two different pre-trained BERT-based models: **Multilingual BERT**² and **BERTimbau** [30] to find which performed better. Similarly, for the label propagation model, we tested representing the sentences with **Doc2Vec** and **Universal Sentence Encoder (USE)**.³

5 Results

This section describes three different types of experiments. Section 5.1 starts by exploring different embeddings for each model, and assessing the impact of different types of pre-processing. Section 5.2 presents experiments performed by each model individually, considering several subsets of training data. Lastly, Section 5.3 presents the results of the ensemble model, and reveals the impact of using additional labeled datasets and back translation.

5.1 Different Embeddings and Pre-processing Experiments

The experiments here described use CO-HATE as training data, and a sample of 300 tweets from FIGHT corpus that were manually annotated for this purpose, as testing data.

We have started by combining our modeling approaches with different embeddings. The results achieved are summarized in Table 2, where the baseline consists of a dummy classifier that classifies all examples as *Hate Speech*. Results show that BERTimbau achieves an overall better performance when combined with GAN-BERT. This was expected considering that BERTimbau was trained using web corpora that are more likely to include toxicity than the Google Books corpus used for Multilingual BERT. For the label propagation model, we have adopted USE, considering that it has a higher recall and F1-score for the positive class, two relevant metrics when detecting hate speech.

■ **Table 2** Performance of different embeddings for each model.

		Acc	HS Class			Macro Average		
			Prec	Rec	F1	Prec	Rec	F1
Dummy Classifier (all HS)		0.177	0.177	1.000	0.300	0.088	0.500	0.150
GAN-BERT	Multilingual	0.633	0.315	0.133	0.187	0.450	0.430	0.458
	BERTimbau	0.647	0.188	0.302	0.232	0.508	0.511	0.501
Label Propagation	Doc2Vec	0.707	0.260	0.358	0.302	0.555	0.569	0.557
	USE	0.653	0.248	0.472	0.325	0.553	0.582	0.546

In order to understand the impact of pre-processing, for each model, we have also performed experiments without any pre-processing, and with two levels of pre-processing. The partial pre-processing is composed of the following steps:

- Noise removal: remove processing errors in the data retrieval;
- Removal of repetitions of three or more punctuation signals and emojis. This step may remove some noise and shorten the message to fit the maximum sequence length. However, it may lose some of the meaning of the sentence;
- Anonymization of users' mentions: replace a user tag with "@UserID" to represent a username with a single word.

² <https://github.com/google-research/bert/blob/master/multilingual.md>

³ <https://tfhub.dev/google/universal-sentence-encoder-multilingual/3>

11:8 Semi-Supervised Annotation of Hate Speech

The full pre-processing approach is composed of the previous steps plus:

- Removal of user’s mentions;
- Removal of links.

As presented in Table 3, the best results for GAN-BERT were obtained with the full pre-processing, contrarily to what we expected. This pre-processing puts the emphasis on the message. However, some of the meaning of the messages can be lost by removing the repetitions of punctuation signals and emojis, and some context can be removed by deleting the user’s mentions and links. For the label propagation model, the best results were obtained without any pre-processing, potentially because the pre-processing removes too much context from the messages. The following section use GAN-BERT with the pre-processed training set, and the label propagation model with the original data, which are the most promising combinations. Our goal is to obtain the most promising model, so, for each experiment, we will select the options that result in the best performance.

■ **Table 3** Impact of pre-processing.

	Pre-processing	Acc	HS Class			Macro Average		
			Prec	Rec	F1	Prec	Rec	F1
GAN-BERT	Without	0.647	0.188	0.302	0.232	0.508	0.511	0.501
	Partial	0.680	0.228	0.340	0.273	0.535	0.546	0.534
	Full	0.707	0.294	0.472	0.362	0.580	0.546	0.586
Label Propagation	Without	0.653	0.248	0.472	0.325	0.553	0.582	0.546
	Partial	0.633	0.234	0.472	0.313	0.544	0.570	0.531
	Full	0.623	0.222	0.453	0.298	0.536	0.556	0.520

5.2 Considering Different subsets, from Different Annotators

As previously mentioned, the CO-HATE corpus annotation process involved five annotators, with the achieved low values of IAA evincing the difficulty and subjectiveness of the task. Therefore, in order to assess the perspective of each annotator in the hate speech classification, we have tested several combinations of data subsets. We have used the corpus annotated by each user independently, the corpus composed of messages labeled by all the annotators, and multiple combinations taking into consideration the annotators that have shown best inter-annotator agreement results. Table 4 presents the results for GAN-BERT, using subsets annotated by each annotator and by combined datasets of the most relevant associations. The following experiments were carried out with the two samples that achieved better results, namely the combination of data annotated by annotators B, C, and D, and the data annotated by all the annotators. The results for the label propagation models are shown in Table 5. For this second model, we opted to use the data annotated by annotators A, B, and C, and by all the annotators.

As mentioned by Carvalho et al. [6], annotators A, B, and C belong to the target groups considered in the corpus. Comparing the IAA between this group and the one composed by annotators D and E, who do not belong to any potential marginalized group, we observe that hate speech is perceived differently by individuals from both groups. In fact, the agreement rate was lower among the individuals of the target groups for almost all dimensions considered in the guidelines. Considering the classification of hate speech, the annotators A, B, and C had an IAA of 0.360, while the annotators D and E had an IAA of 0.735. This corroborates the idea that hate speech identification is a very subjective task, and that the annotators’

■ **Table 4** Impact of the perspective of annotators in the performance of GAN-BERT model.

	Acc	HS Class			Macro Average		
		Prec	Rec	F1	Prec	Rec	F1
A	0.667	0.169	0.226	0.194	0.495	0.494	0.492
B	0.530	0.225	0.679	0.338	0.552	0.589	0.487
C	0.477	0.201	0.660	0.308	0.529	0.529	0.444
D	0.687	0.275	0.472	0.347	0.570	0.602	0.571
E	0.463	0.190	0.623	0.291	0.515	0.526	0.430
BD	0.597	0.254	0.660	0.366	0.571	0.622	0.535
DE	0.677	0.266	0.472	0.340	0.565	0.596	0.563
ABC	0.667	0.169	0.226	0.194	0.495	0.494	0.492
BCD	0.700	0.287	0.472	0.357	0.578	0.610	0.581
ABDE	0.620	0.161	0.280	0.282	0.492	0.493	0.484
ABCDE	0.707	0.294	0.472	0.362	0.580	0.610	0.586

■ **Table 5** Performance of the label propagation model based on the perspective of annotators.

	Acc	HS Class			Macro Average		
		Prec	Rec	F1	Prec	Rec	F1
A	0.730	0.259	0.283	0.270	0.551	0.554	0.552
B	0.637	0.225	0.434	0.297	0.537	0.557	0.526
C	0.707	0.267	0.358	0.302	0.555	0.570	0.558
D	0.537	0.179	0.453	0.257	0.502	0.504	0.460
E	0.463	0.186	0.604	0.284	0.511	0.518	0.428
AC	0.697	0.250	0.358	0.295	0.549	0.564	0.551
DE	0.583	0.223	0.547	0.317	0.541	0.569	0.509
ABC	0.693	0.259	0.396	0.313	0.557	0.577	0.558
BCE	0.563	0.213	0.547	0.307	0.533	0.557	0.494
ABCD	0.677	0.250	0.415	0.312	0.552	0.574	0.550
ABCDE	0.643	0.240	0.472	0.318	0.549	0.576	0.538

social identity may influence the perception of HS. Taking this into account, we tried to investigate the impact of each group on the performance of the models and assess whether higher IAA lead to better performance. For GAN-BERT, from Table 4, it is clear that the sample composed by annotators D and E obtained globally better results. For the label propagation model, from Table 5, although using the data from annotators A, B and C led to higher accuracy and precision, the F1-score for the positive class is slightly higher for D and E.

In order to understand the potential of GAN-BERT using the FIGHT corpus, similarly to what was done by Croce et al. [8], each message was labeled as *Unknown* and added to the CO-HATE corpus. The final label given to each point was the one with the highest confidence between *Hate Speech* and *Non Hate Speech*. This increased the accuracy of the model, but reduced the remaining metrics so the idea was discarded.

5.3 Ensemble Model with Additional Labeled Resources

After assessing the potential of both models individually, they were combined in order to produce the labels for the FIGHT corpus. Each individual model used the best training set, i.e., the pre-processed CO-HATE corpus for GAN-BERT and the original one for the label propagation model. Table 6 shows the corresponding results for six different experiments, all of them considering an iterative training over five epochs.

Experiment 1 consisted in adding the most confident predictions (above 0.9) given simultaneously by both models to the training set of the following epoch. Considering that the majority of the labels were *Non Hate Speech* (non-HS), we observed that the training set was getting too unbalanced (only around 20% *Hate Speech*) and the performance of the models was decreasing. To overcome this issue, Experiment 2 consisted of adding only the *Hate Speech* labels, thus obtaining around 42% of hate speech. Experiment 3 consisted of adding all the most confident predictions, including the ones given by only one model. Although the recall increased due to the higher number of hate speech instances, the accuracy and precision decreased, which is possibly explained by the lower confidence associated with these labels.

In order to increase the amount of hate speech present in the training set, we have also added ToLR-BR and HPHS. This solution significantly increased the performance of the model, as reported in Experiments 4 and 5. However, similarly to the previous Experiments 1 and 2, adding only the positive examples turned out to be a better approach (Experiment 5). Additionally, Experiment 6 used back translation to generate more annotated examples from the additional datasets. For this, the hate speech sentences were translated from Portuguese into English and then, back to Portuguese, using the Google translate API – Googletrans.⁴ The results reveal a significantly lower performance, possibly due to loss of context during the translation process.

■ **Table 6** Impact of data additions to the training set in label propagation model.

Setup	Acc	HS Class			Macro Average		
		Prec	Rec	F1	Prec	Rec	F1
Baseline	0.650	0.245	0.472	0.323	0.552	0.580	0.543
1) labels in common, HS+non-HS	0.672	0.602	0.583	0.592	0.643	0.667	0.659
2) labels in common, HS	0.693	0.618	0.647	0.632	0.655	0.692	0.684
3) all labels, HS	0.669	0.568	0.789	0.660	0.665	0.669	0.668
4) additional datasets, HS+non-HS	0.713	0.676	0.573	0.620	0.693	0.602	0.695
5) additional datasets, HS	0.708	0.629	0.693	0.659	0.687	0.723	0.702
6) additional datasets, HS, back translation	0.644	0.579	0.472	0.520	0.628	0.618	0.619

Considering all the experiments performed, the final results were obtained using as initial training set the CO-HATE corpus with the data classified by the corresponding annotators and the instances corresponding to hate speech of the two additional datasets. Table 7 reports the results of these experiments after five iterations, revealing that GAN-BERT requires more data in order to obtain better results. Using the data from the annotators B, C, and D, the majority of the metrics decreased when adding the additional dataset, especially precision. This corroborates the theory that GAN-BERT is more susceptible to noise, as seen when assessing the impact of pre-processing. However, with the entire corpus, we can obtain better results than the previous baseline. For the label propagation model,

⁴ <https://py-googletrans.readthedocs.io/en/latest/>

using the corpus correspondent to the annotators A, B, and C led to an increase in precision but a decrease in the remaining metrics. Considering the entire dataset, there is a clear increase in terms of recall and F1-score.

■ **Table 7** Models’ performance after 5 iterations.

		Acc	HS Class			Macro Average		
			Prec	Rec	F1	Prec	Rec	F1
GAN-BERT	Baseline	0.707	0.294	0.472	0.362	0.580	0.546	0.586
	BCD	0.317	0.194	0.906	0.319	0.549	0.548	0.317
	ABCDE	0.693	0.600	0.743	0.664	0.691	0.683	0.673
Label Propagation	Baseline	0.708	0.629	0.693	0.659	0.687	0.723	0.702
	ABC	0.672	0.657	0.413	0.507	0.667	0.632	0.631
	ABCDE	0.692	0.601	0.743	0.664	0.667	0.748	0.690

5.4 Final Considerations

In an attempt to compare our results with other work reported literature, we considered the work of Breazzano et al. [5] and D’Sa et al. [10] involving Italian and English, respectively, and a similar task, since we did not find any other previous similar work for Portuguese. However, it is important to stress that results can not be directly compared, not only because of the different languages and cultural aspects, but mostly because the testing datasets are different. Breazzano et al. [5] applied GAN-BERT to several Italian hate speech. Both the HaSpeeDe⁵ and the DANKMEMES [20] datasets were used in a binary classification task, with the best model achieving a macro average F1-score of 0.633 and 0.584 and an accuracy of 0.693 and 0.562, respectively. D’Sa et al. [10] applied a label propagation model to two English datasets from Founta et al. [14] and Davidson et al. [9] to distinguish hate speech from offensive and normal speech, obtaining a macro average F1-score around 0.670 and 0.710, respectively. Considering that our task corresponds to a cross-domain scenario, we expected this would negatively impact the results. Additionally, since BERTimbau was trained with Brazilian Portuguese, the GAN-BERT model can have been impacted by vocabulary used only in European Portuguese. Besides that, for the label propagation model, the comparison is done with English datasets, so we expected lower results due to the existence of more morphological variations in Portuguese [26]. However, with GAN-BERT we obtained a macro average F1-score of 0.673, and 0.702 for the label propagation model, which are in line with the above mentioned results, reinforcing the potential of this approach.

6 Conclusions and Future Directions

In the literature, several semi-supervised learning methods have been applied in the field of text classification and adapted to hate speech detection. However, this task is extremely complex and subjective, and its success often depends on the creation of robust and large-coverage language resources, which are still scarce for Portuguese. To address this gap, we have implemented an ensemble of two semi-supervised models. The first one employs a GAN in combination with a BERT-based model. The second model is based on label propagation,

⁵ <https://github.com/msang/haspeede/>

which propagates labels based on similarities. The two models were combined to extract the most confident predictions, which were added to the training data of the next iteration, in an active-learning fashion.

We have explored the annotations of three existing corpora (CO-HATE, ToLR-BR, and HPHS) in order to automatically annotate FIGHT, a corpus composed of geolocated tweets produced in the Portuguese territory. Several pre-processing strategies were tested, demonstrating good results, particularly for the GAN-BERT model. Back translation was also tested in an attempt to generate more hate speech examples, but no performance increase was obtained. The best results were obtained using all the corpora. Specifically, we obtained an F1-score of 0.664 for the *Hate Speech* class for both models. The label propagation approach proved to be more stable and less susceptible to noise, with similar performance to the existing models, besides being a less complex model, and hence faster to train with larger amounts of data. However, both models obtained good performance, especially considering the different nature of the corpora.

In terms of future directions, we plan to manually annotate the entire FIGHT corpus, in an semi-automatic way, and to perform further extensive cross-domain experiments involving CO-HATE and FIGHT, using the proposed models.

References

- 1 Hala Al Kuwaty, Maximilian Wich, and Georg Groh. Identifying and measuring annotator bias based on annotators' demographic characteristics. In *Proceedings of the Fourth Workshop on Online Abuse and Harms*, pages 184–190. Association for Computational Linguistics, November 2020. doi:10.18653/v1/2020.a1w-1.21.
- 2 Safa Alsafari and Samira Sadaoui. Semi-Supervised Self-Training of Hate and Offensive Speech from Social Media. *Applied Artificial Intelligence*, pages 1–25, October 2021. doi:10.1080/08839514.2021.1988443.
- 3 Fabienne Baider and Maria Constantinou. Covert hate speech: A contrastive study of greek and greek cypriot online discussions with an emphasis on irony. *Journal of Language Aggression and Conflict*, 8(2):262–287, 2020.
- 4 Djamila Romaiisa Beddiar, Md Saroar Jahan, and Mourad Oussalah. Data expansion using back translation and paraphrasing for hate speech detection. *Online Social Networks and Media*, 24:100153, 2021. doi:10.1016/j.osnem.2021.100153.
- 5 Claudia Breazzano, Danilo Croce, and Roberto Basili. MT-GAN-BERT : Multi-Task and Generative Adversarial Learning for sustainable Language Processing. In *Proceedings of the Fifth Workshop on Natural Language for Artificial Intelligence (NL4AI 2021)*. CEUR Workshop Proceedings, November 2021. URL: <http://ceur-ws.org/Vol-3015/>.
- 6 Paula Carvalho, Danielle Caled, Cláudia Silva, Fernando Batista, and Ricardo Ribeiro. The expression of Hate Speech against Afro-descendant, Roma and LGBTQ+ communities in YouTube comments. *Discourse and Society*, submitted.
- 7 Paula Carvalho, Bernardo Matos, Raquel Santos, Fernando Batista, and Ricardo Ribeiro. Hate Speech Dynamics Against African descent, Roma and LGBTQI Communities in Portugal. *LREC*, 2022.
- 8 Danilo Croce, Giuseppe Castellucci, and Roberto Basili. GAN-BERT: Generative adversarial learning for robust text classification with a bunch of labeled examples. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2114–2119, Online, July 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.acl-main.191.
- 9 Thomas Davidson, Dana Warmisley, Michael Macy, and Ingmar Weber. Automated hate speech detection and the problem of offensive language. *Proceedings of the 11th International Conference on Web and Social Media, ICWSM 2017*, pages 512–515, 2017. arXiv:1703.04009.

- 10 Ashwin Geet D'Sa, Irina Illina, Dominique Fohr, Dietrich Klakow, and Dana Ruiter. Label Propagation-Based Semi-Supervised Learning for Hate Speech Classification. In *Proceedings of the First Workshop on Insights from Negative Results in NLP*, pages 54–59. Association for Computational Linguistics, November 2020. doi:10.18653/v1/2020.insights-1.8.
- 11 Tim Fitzsimons. Nearly 1 in 5 hate crimes motivated by anti-LGBTQ bias, FBI finds. *NBC News*, November 2019. URL: <https://www.nbcnews.com/feature/nbc-out/nearly-1-5-hate-crimes-motivated-anti-lgbtq-bias-fbi-n1080891>.
- 12 Paula Fortuna and Sérgio Nunes. A survey on automatic detection of hate speech in text. *ACM Computing Surveys*, 51(4):1–30, July 2019. doi:10.1145/3232676.
- 13 Paula Fortuna, João Rocha da Silva, Juan Soler-Company, Leo Wanner, and Sérgio Nunes. A hierarchically-labeled Portuguese hate speech dataset. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 94–104, Florence, Italy, August 2019. Association for Computational Linguistics. doi:10.18653/v1/W19-3510.
- 14 Antigoni Maria Founta, Constantinos Djouvas, Despoina Chatzakou, Ilias Leontiadis, Jeremy Blackburn, Gianluca Stringhini, Athena Vakali, Michael Sirivianos, and Nicolas Kourtellis. Large scale crowdsourcing and characterization of twitter abusive behavior. In *Twelfth International AAAI Conference on Web and Social Media*, 2018.
- 15 Akshita Jha and Radhika Mamidi. When does a compliment become sexist? analysis and classification of ambivalent sexism using twitter data. In *Proceedings of the second workshop on NLP and computational social science*, pages 7–16, 2017.
- 16 György Kovács, Pedro Alonso, and Rajkumar Saini. Challenges of hate speech detection in social media. *SN Computer Science*, 2(2):1–15, 2021.
- 17 Ritesh Kumar, Atul Kr Ojha, Shervin Malmasi, and Marcos Zampieri. Benchmarking aggression identification in social media. In *Proceedings of the first workshop on trolling, aggression and cyberbullying (TRAC-2018)*, pages 1–11, 2018.
- 18 Joao A Leite, Diego F Silva, Kalina Bontcheva, and Carolina Scarton. Toxic language detection in social media for brazilian portuguese: New dataset and multilingual analysis. *arXiv preprint*, October 2020. arXiv:2010.04543.
- 19 Changchun Li, Ximing Li, and Jihong Ouyang. Semi-Supervised Text Classification with Balanced Deep Representation Distributions. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5044–5053. Association for Computational Linguistics, August 2021. doi:10.18653/v1/2021.acl-long.391.
- 20 Martina Miliani, Giulia Giorgi, Ilir Rama, Guido Anselmi, and Gianluca E Leboni. DANKM-EMES@ EVALITA 2020: The Memeing of Life: Memes, Multimodality and Politics. In *EVALITA*, 2020.
- 21 Yassine Ouali, Céline Hudelot, and Myriam Tami. An Overview of Deep Semi-Supervised Learning. *arXiv:2006.05278*, pages 1–43, June 2020. arXiv:2006.05278.
- 22 Maria Papadaki. Data Augmentation Techniques for Legal Text Analytics. Master's thesis, Athens University of Economics and Business, October 2017. URL: <http://nlp.cs.aueb.gr/theses.html>.
- 23 Haeyoun Park and Iaryna Lyshyn. L.G.B.T. people are more likely to be targets of hate crimes than any other minority group. *The New York Times*, June 2016. URL: <https://www.nytimes.com/interactive/2016/06/16/us/hate-crimes-against-lgbt.html>.
- 24 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL: <https://scikit-learn.org/>.
- 25 Wyatt Ronan. New FBI Hate Crimes Report Shows Increases in Anti-LGBTQ Attacks. *Human Rights Campaign*, November 2020. URL: <https://www.hrc.org/press-releases/new-fbi-hate-crimes-report-shows-increases-in-anti-lgbtq-attacks>.

11:14 Semi-Supervised Annotation of Hate Speech

- 26 Diana Santos and Alberto Simões. Portuguese-English word alignment: some experiments. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, May 2008. European Language Resources Association (ELRA). URL: http://www.lrec-conf.org/proceedings/lrec2008/pdf/760_paper.pdf.
- 27 Sheikh Muhammad Sarwar and Vanessa Murdock. Unsupervised Domain Adaptation for Hate Speech Detection Using a Data Augmentation Approach. *arXiv:2107.12866*, July 2021. [arXiv:2107.12866](https://arxiv.org/abs/2107.12866).
- 28 Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1), July 2019. doi:10.1186/s40537-019-0197-0.
- 29 Alexandra A. Siegel. Online hate speech. In Joshua A. Tucker Nathaniel Persily, editor, *Social Media and Democracy*, chapter 4, page 67. Cambridge University Press, August 2021. doi:10.1017/9781108890960.
- 30 Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. BERTimbau: Pretrained BERT Models for Brazilian Portuguese. In *Brazilian Conference on Intelligent Systems*, pages 403–417. Springer, 2020.
- 31 Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine Learning*, 109:373–440, November 2020. doi:10.1007/s10994-019-05855-6.
- 32 Zeerak Waseem. Are You a Racist or Am I Seeing Things? Annotator Influence on Hate Speech Detection on Twitter. In *Proceedings of the First Workshop on NLP and Computational Social Science*, pages 138–142. Association for Computational Linguistics, November 2016. doi:10.18653/v1/w16-5618.
- 33 Michael Wiegand, Josef Ruppenhofer, and Elisabeth Eder. Implicitly Abusive Language – What does it actually look like and why are we not getting there? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 576–587. Association for Computational Linguistics, June 2021. doi:10.18653/v1/2021.naacl-main.48.
- 34 Wenjie Yin and Arkaitz Zubiaga. Towards generalisable hate speech detection: a review on obstacles and solutions. *PeerJ Computer Science*, 7:e598, 2021.

Large Semantic Graph Summarization Using Namespaces

Ana Rita Santos Lopes da Costa ✉ 

Department of Computer Science, Faculty of Science, University of Porto, Portugal

André Santos ✉ 

CRACS & INESC Tec LA / Faculty of Sciences, University of Porto, Portugal

José Paulo Leal ✉ 

CRACS & INESC Tec LA / Faculty of Sciences, University of Porto, Portugal

Abstract

We propose an approach to summarize large semantics graphs using namespaces. Semantic graphs based on the Resource Description Framework (RDF) use namespaces on their serializations. Although these namespaces are not part of RDF semantics, they have intrinsic meaning. Based on this insight, we use namespaces to create summary graphs of reduced size, more amenable to be visualized. In the summarization, object literals are also reduced to their data type and the blank nodes to a group of their own. The visualization created for the summary graph aims to give insight of the original large graph. This paper describes the proposed approach and reports on the results obtained with representative large semantic graphs.

2012 ACM Subject Classification Information systems → Summarization; Information systems → Resource Description Framework (RDF)

Keywords and phrases Semantic graph, RDF, namespaces, reification

Digital Object Identifier 10.4230/OASICS.SLATE.2022.12

Supplementary Material *Software (Python Module)*: <https://pypi.org/project/rdf-summarizer/>
Software (Source Code): <https://github.com/ritasantos11/RDF-Summarizer>
archived at `swh:1:dir:a5f50e27fbce52accaec26736daa124a0d0fe728`

Funding This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

André Santos: Ph. D. Grant SFRH/BD/129225/2017 from Fundação para a Ciência e Tecnologia (FCT), Portugal.

1 Introduction

Semantic graphs based on the Resource Description Framework (RDF) are frequently massive, with more than a billion triples. This order of magnitude raises several problems when processing these RDF graphs: they are difficult to load into memory; querying them is time-consuming, and visualization tools are virtually useless.

Graph summaries are an approach to deal with this issue. Summaries may be a smaller sub-graph retaining only some nodes and edges; or a document in another format which summarizes the graphs content, such as statistics on the number of nodes and edges, or its structural features.

A common approach in graph summarization is to group nodes and edges of similar kind. The challenge is to identify a reduced number of groups of nodes and edges in an RDF graph so that the graph summary is both meaningful and understandable. On the one hand, the groups of nodes and edges must retain part of the meaning of their elements. On the other hand, the reduced graph must be small enough to be easily understood.



© Ana Rita Santos Lopes da Costa, André Santos, and José Paulo Leal;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 12; pp. 12:1–12:9
OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Most nodes and edges in RDF have an associated Internationalized Resource Identifier (IRI), typically an Uniform Resource Locator (URL). RDF serialization formats, such as RDF/XML and Turtle, shorten these IRIs using namespaces. Although namespaces exist only in serializations and are not part of the semantics of RDF, they have intrinsic meaning.

Namespaces reflect a common purpose or a common origin of IRIs. They are more than just alias to prefixes shared by a large set of IRIs used in a semantic graph. Similar resources described in a semantic graph typically share a common namespace. For instance, book resources in DBpedia have a common namespace¹. Also, each vocabulary used in semantic graphs has its particular namespace, as is the case of Dublin Core or Friend-of-a-friend.

The semantics of an RDF graph has two equivalent representations: either as a set of triples whose elements are IRIs or strings; or as a labeled multi-graph whose labels are IRIs or strings. In a strict sense, namespaces are not part of RDF semantics. However this does not entail that they are devoid of meaning. Namespaces reflect a commonality of resource identifiers and vocabularies and are assigned by semantic graphs authors, not automatically generated by an algorithm.

Based on this insight, this work explores the use of namespaces and their intrinsic meaning to summarize RDF graphs. Certainly, we must also consider literals and blank nodes since IRIs are not the only kind of element in RDF triples. Thus, the proposed approach maps IRIs, literals, and blank nodes to a reduced set of identifiers to produce summary graphs retaining part of the semantics of the original graph.

Our goal is to generate meaningful summaries from semantic graphs with hundreds of millions of triples and produce them in a few hours. We want to understand the information stored in the graph that otherwise we would not be able to given the volume of the data and to visualize it in an easy way. We applied the proposed summarization method to two large graphs, with more than a million triples: the KBpedia knowledge graph and the Linked Movie Database. The first has over one million triples and the second over three million triples. We were able to produce a summary graph, from which we were able to extract visualizations and statistics.

The remainder of this paper is organized as follows. Section 2 provides background on semantic graphs and RDF. Section 3 surveys related work on semantic graph summarization. Section 4 details the proposed approach to using namespaces for semantic graph summarization. Section 5 reports on the use of this approach to summarize two RDF graphs: KBpedia and LinkedMDB (Linked Movie Database). Finally, Section 6 identifies the main contributions of this research and opportunities for future work.

2 Background

Semantic graphs store information about concepts in the nodes and the semantic relations between them in the edges. They are associated with ontologies, formal and explicit specifications of shared formalizations characterized by high semantic expressiveness required for increased complexity [11].

These type of graphs are expressed in RDF where the knowledge about a given domain is represented by triples $[subject, predicate, object]$. Each triple states that the *subject* is connected to the *object* through a relation described by the *predicate* [7]. RDF itself contains properties for relating subjects to objects, such as the *rdf:type* property, and it is also possible to create other specific properties for the domain in question. RDF graph nodes can be an Internationalized Resource Identifier (IRI), a literal or a blank node (an anonymous resource

¹ <http://purl.org/NET/book/vocab#>

for which an IRI or literal is not given). The literals can only appear in the object position of the triples and have associated a datatype that indicates what is the type of the content of that literal. It can be, for example, strings, integers or dates [16]. Blank nodes can only be used in the subject or object positions.

In some serialization formats the IRIs that identify nodes and edges can be associated with a namespace. Common prefixes of IRIs are given smaller names to identify them, allowing not to use the full IRI when referring to a node or edge. For example, in the RDF Turtle serialization format, a prefix definition would be:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
```

With this declaration, instead of using the RDF property *type* with the full IRI², it can be shortened to `rdf:type`.

These alias definitions are internal to each serialized document and have no impact on the semantics of the graph. Nevertheless, common namespaces (from widely used vocabularies or ontologies) are usually shortened to the same prefixes. The *prefix.cc* website³ contains a crowd-sourced mapping of namespaces and prefixes [9], and it includes a namespace lookup.

Often in RDF there is the need to make statements about other statements. RDF*, an extension to RDF is being developed to address this issue, but it is currently only a draft [1]. RDF Reification is another approach which provides the ability to make RDF statements to express something in a language using the language, so it becomes treatable by the language [4]. In practice, it allows building a triple in which the subject is another triple. As an example, if we want to represent information about the source of the triple [*ex:Gardener ex:hasKilled ex:butler*], we need to construct a RDF statement about it, as in Figure 1. This way, the reified triple can be referenced by others triples.

```
ex:statementExample rdf:type rdf:Statement .
ex:statementExample rdf:subject ex:Gardener .
ex:statementExample rdf:predicate ex:hasKilled .
ex:statementExample rdf:object ex:Butler .
ex:statementExample ex:saidBy ex:Nurse
```

■ **Figure 1** Reification example.

KBpedia is a medium-sized open-source knowledge graph that combines leading public knowledge bases (Wikipedia, Wikidata, schema.org, DBpedia, GeoNames, OpenCyc, and the UNSPSC products and services) into an integrated and computable structure [3]. LinkedMDB is a RDF graph that contains linked-data-collection of movies, actors, directors and the relationships between them [10].

SPARQL Protocol and RDF Query Language (SPARQL) is a language for querying RDF graphs. Through this language, it is possible, for example, to return the properties associated with a certain class, query the instances that are part of a certain class and count the numbers of triples [15].

Graphs can be described using the DOT language. DOT is a text based graph representation which can be transformed into a diagram using tools like Graphviz [8]. Graphviz is an open source graph visualization software that allows to create nodes and edges with different sizes and colors, for example. It provides eight layout engines to draw the graph [2].

² <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>

³ <http://prefix.cc>

A trie or prefix tree is a tree data structure that stores strings. It allows to retrieve a string from a set in an efficient way. It supports operations of insertion and lookup. For this work, we adapted the definition of trie given in [12].

3 Related Work

Liu et al. [14] present several kinds of graph summarization methods. They distinguish between methods for static graphs that do not consider additional information to their structure and those for static graphs that contain attributes in nodes and arcs.

Static graphs with attributes can use the following summarization kinds of methods: aggregation-based, bit-compression-based, and influence-based. Aggregation methods group several nodes that share properties into a *supernode*, making the summary graph contain specific properties. Others apply clustering to map each densely connected cluster into a supernode. Bit-compression-based methods decrease the number of bits needed to store a graph. Influence-based methods discover the description of the spread of influence by formulating the summarization problem as an optimization process in which some information about the influence is kept [14].

Static graphs without attributes can also use methods based on simplification or sparsification. These methods remove less important nodes or arcs, resulting in a sparsified graph [14].

Bonifati et al. [5] present statistical and goal-oriented method types. The former is based on quantitative measurements and the occurrence count. The latter optimizes the memory footprint by producing a concise representation that fits in memory.

Specifically for semantic graphs, many of the available summarization methods are either structural or statistical. Structural methods consider structural features such as paths, graph patterns, or frequent nodes. Some of these approaches extract the most frequent graph patterns, sub-graphs that share common types and properties. Statistical methods rely on graph statistics, such as node type frequencies. We can combine these method types to obtain better results [7].

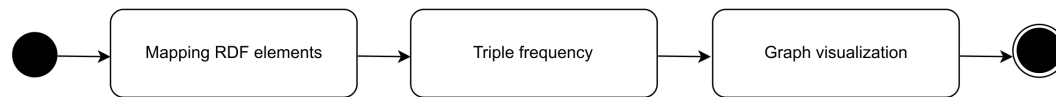
A particular type of semantic graph summarization is schema extraction. If the graph itself does not have an associated ontology, it is possible to extract a schema that acts as a summary of the graph [7]. The extracted schema provides information on the content of the graph, that is, what types and properties exist, and helps with its exploration [13].

Kellou-Menouer and Kedad [13] propose a schema with types and edge definitions based on a density clustering algorithm. This approach generates types through grouping and the description of each one of them, where each property will have an associated probability. It then generates the edges between types by analyzing the type descriptions. The resulting summary is a small graph showing the relationships between the different types of the original one. The first phase of this approach is to generate the types and their description, called type profiles, with each property having an associated probability. The following step is to generate semantic and hierarchical edges between types by analyzing type profiles. In the first phase, a density-based clustering algorithm is applied to group similar entities and create a profile for each class. These profiles are used to find semantic and hierarchical edges between types and to generate overlapping types. Type profiles are vectors of properties, where each property has an associated probability that indicates its relative importance.

Bouhamoum et al. [6] adapt the previous approach to massive graphs. They transform the graph into a concise representation that contains all the patterns of combined properties. These patterns are collections of properties that appear together on one or more nodes. The scheme results from a clustering algorithm applied to these patterns. The result is the same if the algorithm is applied to the original graph, but is faster if applied to patterns.

4 Approach

The UML diagram in Figure 2 depicts the three main activities in summarization process. We start by mapping the RDF elements of each triple into groups, producing reduced triples. Then, we count repeated reduced triples and produce a summary graph. We finish with a creation of a visualization of the summary graph.



■ **Figure 2** UML activity diagram.

The following subsections detail the graph summarization workflow. Subsection 4.1 describes how RDF triple components are reduced to group identifiers. Subsection 4.2 explains how a summary graph is produced from triples with a reduced number of identifiers. Finally, Subsection 4.3 describes how the graph is visualized.

4.1 Mapping RDF elements into groups

An arc linking two nodes in a semantic graph corresponds to a triple. The subject and object are the source and target nodes and the predicate is the arc. The elements of RDF triples can be IRIs, literals, or blank nodes. We deal with each of these differently:

IRIs: are reduced to their namespace;

Literals: are reduced to their datatype;

Blank nodes: are reduced to a particular group.

Literals occur only as objects in triples and represent unstructured data; for instance, a date or a number. This kind of element is reduced to the literal datatype. Blank nodes occur either as subjects or objects in a triple and correspond to unidentified resources that cannot be referenced from another graph. This kind of element tends to be less frequent and is reduced to a particular group. The most frequent kind of element in a triple is the IRI and reducing them is the core of the proposed approach.

Figure 3 shows a triple from a large graph and how its IRIs are shortened using namespaces. The central column presents the original IRIs with the namespace prefix underlined. The right column presents the short version of the same IRI with the prefix replaced by an alias. The alias is also underlined and separated from the suffix by a colon.

Triple	Original IRI	Short IRI
Subject	<u>http://kbpedia.org/kko/rc/Abbey</u>	<u>kko</u> :Abbey
Predicate	<u>http://www.w3.org/2000/01/rdf-schema#subClassOf</u>	<u>rdfs</u> :subClassOf
Object	<u>http://dbpedia.org/ontology/Abbey</u>	<u>dbo</u> :Abbey

■ **Figure 3** Example of a triple and its corresponding triple with namespaces.

IRIs are mapped to their namespaces. This is done by

1. checking explicit prefix declarations in the RDF file;
2. checking the Prefix.cc prefix database;
3. pattern matching.

12:6 Large Semantic Graph Summarization Using Namespaces

If the RDF file being processed has any prefix declarations, these are used as the preferred method for finding prefixes: any IRIs matching these prefix declarations are reduced to the corresponding alias defined in the file. IRIs not reduced by the previous step are looked up on a database of prefixes downloaded from the Prefix.cc website⁴, which contains a mapping between prefixes and their common alias for almost 3000 prefixes. These prefixes are inserted into a trie where they can be efficiently searched. For all the remaining IRIs we attempt to find namespace prefixes by deleting trailing chunks of the IRIs and searching for common patterns. Namespace prefixes usually end in either '#' or '/', so we repeatedly delete the portion of the IRI after the last trailing '#' or '/', and add to the prefix list any truncated IRI appearing multiple times.

4.2 Graph summary creation

Triples in the original graph are converted into reduced triples, with each of its elements (subject, predicate, object) reduced to a group identifier, as detailed in the previous subsection. Then, the reduced triples are condensed into a summary graph.

The summary is itself an RDF graph and each statement corresponds to a reduced triple. These triples are reified RDF statements: each is attributed its own identifier, allowing assertions about it to be made. All resources in the summary graph are identified with IRIs on the `ngs` namespace⁵. These include triples identifications and references to groups. The namespace is also used for a predicate that states the number of occurrences of reduced triples, counted during triple conversion.

```
@prefix ngs: <https://www.dcc.fc.up.pt/~up201605706/ngs#> .
ngs:t1 rdf:type rdf:statement
ngs:t1 rdf:subject ngs:kko
ngs:t1 rdf:predicate ngs:rdfs
ngs:t1 rdf:object ngs:dbo
ngs:t1 ngs:num_occurrences 530
```

■ **Figure 4** Example of reduced triples represented in the summary graph.

Listing 4 shows the set of triples in the summary that represent a set of reduced triples. It corresponds to 530 triples in the original graph that are mapped to the single triple *[kko rdfs dbo]*. That is, those 530 original triples had subjects that were reduced to `kko`, predicates reduced to `rdfs` and objects reduced to `dbo`.

4.3 Graph visualization

The graph summary is then processed to be visualized. This requires the conversion of the graph to a DOT file, using the following visual features to represent nodes and edges characteristics:

1. **node size:** nodes appearing more frequently in the graph summary are represented in a larger size;
2. **node color:** groups corresponding to literal data types are represented in a different color;

⁴ <http://prefix.cc>

⁵ Corresponding to the prefix <https://www.dcc.fc.up.pt/~up201605706/ngs#>.

3. **edge thickness:** triples with a higher number of occurrences in the graph summary are represented with thicker edges;
4. **edge color:** edges are drawn with different colors, each corresponding to a different group.

These visual features highlight the relative frequency of namespaces used in nodes and edges and the connections between them. Additionally, it makes it easy to detect other graph features such as strongly connected components, or disconnected graphs.

5 Validation

To validate the proposed semantic graph summarizing technique we applied it to KBpedia knowledge graph. We ran the program in two ways: converting the object literals to their data types (option 1 in the table of results) and not converting them (option 2).

Table 1 shows the number of triples that the produced smaller graph has, the approximated time that the program took to finish to produce a graph and to write the RDF statements, the percentage of reduction that was obtained like this: $\frac{\#triples - \#reduced_triples}{\#triples} * 100$ and the number of triples that are processed per second: $\frac{\#triples}{time}$.

■ **Table 1** Table of results.

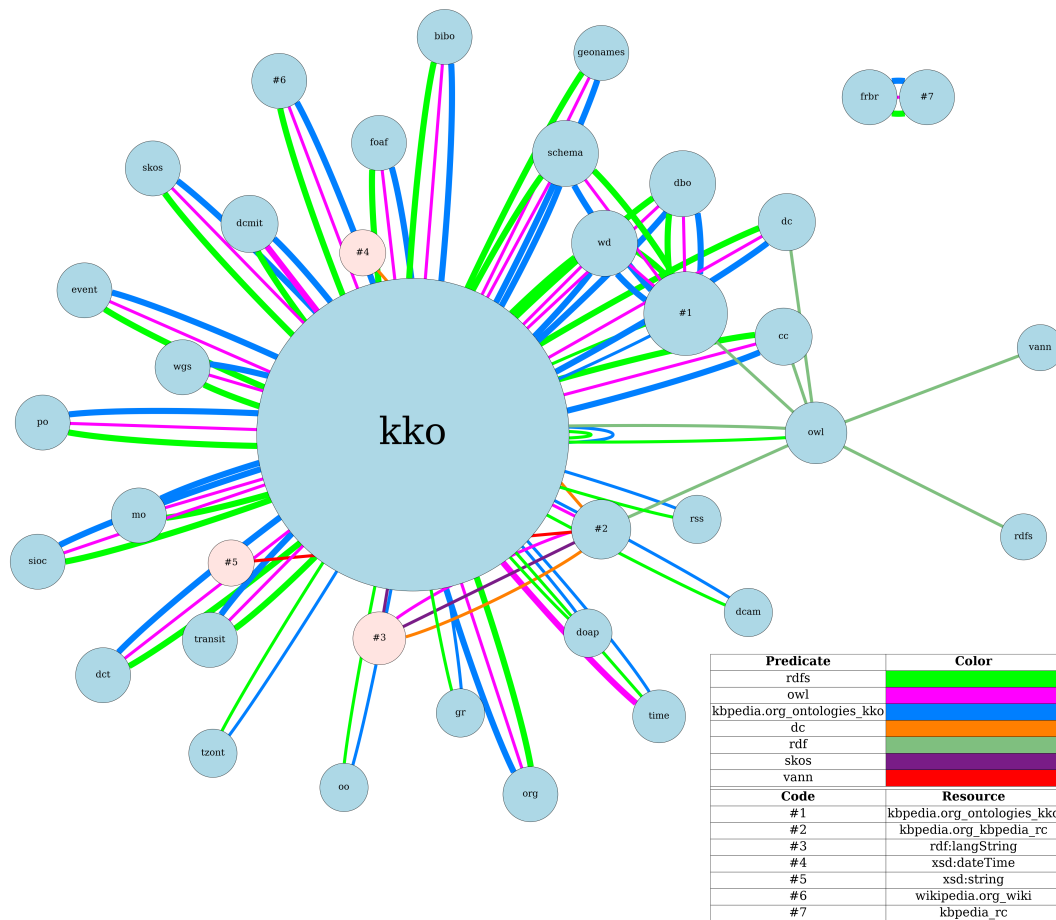
Graph	#Triples	Option	# Reduced triples	% Triple reduction	Time	#Triples/s
KBpedia	1 175 147	1	159	99.99	0h07	2 798
		2	442 290	62.36	01h10	280

Figure 5 shows a visual representation of the KBpedia's summary graph. From this summary we are able to discover several Kbpedia features.

- KBpedia's namespace for individuals, <http://kbpedia.org/kko/rc/>), represented in the figure by the **kko** node, is both the most central node (most of the other nodes are connected to it) and the most frequently mentioned.
- The second largest node (#1) corresponds to the <http://kbpedia.org/ontologies/kko#> namespace, used in KBpedia to specify ontology elements.
- KBpedia uses the IRI <http://kbpedia.org/kbpedia/rc#> to identify the graph itself. It is the only resource in its namespace (#2), and it connects to **kko** and three literal groups, **rdf:langString**, **xsd:dataTime** and **xsd:string**.
- Most of the predicates in KBpedia come from three different groups. The most frequently used are **rdfs** and KBpedia's **kbpedia.org_ontologies_kk** (corresponding to the <http://kbpedia.org/ontologies/kko#> namespace, followed by **owl**).
- The namespace <http://kbpedia.org/kbpedia/rc/> (ending in '#' instead of '/', corresponding to group #7 in the figure) defines a set of classes connected only to the **frbr** group (which corresponds to the <http://purl.org/vocab/frbr/core#> namespace).
- Triples of these groups are disconnected from the core of KBpedia. This is an interesting and unexpected insight provided by the summary.

6 Conclusion and future work

Massive semantic graphs are increasingly hard to understand and visualize. A remedy is to summarize them into smaller graph. In most graph summarizing techniques nodes are grouped into supernodes, and edges between them are grouped in superedges, thus reducing



■ **Figure 5** Visualization of KBpedia graph summary.

graph sizes. Our ongoing research explores the use of namespaces to obtain these supernodes and edges for RDF graphs. The work presented in this paper contributes with a summarizing technique for RDF graphs and a Python package implementing it.

In the proposed approach, RDF triple elements are mapped into groups according to their kind: IRIs are mapped into their namespaces, literals into their data types, and blank nodes into a particular group. The result is a collection of triples of the mapped elements, where most triples are repeated several times. The summary graph is also an RDF graph, where reduced triples are reified, and the nodes representing each triple record the number of repetitions. Finally, the summary graph is converted into the DOT graph description language for visualization.

The proposed approach was applied to large RDF graphs, one with over 1 million triples and another with over 3 million triples. For the smaller one, the resulting summary graphs were produced in a short time and provided meaningful information. For larger graphs, such as LinkedMDB, results could not be obtained in a timely manner. Hence, the current implementation must be optimized to handle larger semantic graphs.

The work presented in this paper is still in progress, and we have planned several improvements. We will explore this approach with even larger graphs to improve the algorithm efficiency. We will formalize graph summaries using RDF Schema, defining the

reification of mapped triples. Finally, we will contribute a package for summarizing RDF graphs to PyPI, the repository of software for the Python programming language, which will be available at <https://pypi.org/project/rdf-summarizer/>.



References

- 1 Dörthe Arndt, Jeen Broekstr, Bob DuCharme, Ora Lassila, Peter F. Patel-Schneider, Eric Prud'hommeaux, Jr. Ted Thibodeau, and Bryan Thompson. RDF-star and SPARQL-star. URL: https://w3c.github.io/rdf-star/cg-spec/editors_draft.html.
- 2 Graphviz authors. Graphviz. URL: <https://graphviz.org>.
- 3 Mike Bergman and Frédéric Giasson. KBpedia. URL: <https://kbpedia.org>.
- 4 Tim Berners-Lee. Reifying rdf (properly), and n3. URL: <https://www.w3.org/DesignIssues/Reify.html>.
- 5 Angela Bonifati, Stefania Dumbrava, and Haridimos Kondylakis. Graph summarization. *arXiv preprint arXiv:2004.14794*, 2020.
- 6 Redouane Bouhamoum, Kenza Kellou-Menouer, Stephane Lopes, and Zoubida Kedad. Scaling up schema discovery for rdf datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 84–89. IEEE, 2018.
- 7 Šejla Čebirić, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. Summarizing semantic graphs: a survey. *The VLDB journal*, 28(3):295–327, 2019.
- 8 Dinis Cruz. Dot language (graph based diagrams). URL: <https://medium.com/@dinis.cruz/dot-language-graph-based-diagrams-c3baf4c0decc>.
- 9 Richard Cyganiak. prefix.cc. URL: prefix.cc.
- 10 Linked data. Linked movie database. URL: <https://data.world/linked-data/linkedmdb>.
- 11 Christina Feilmayr and Wolfram Wöß. An analysis of ontologies and their success factors for application to business. *Data & Knowledge Engineering*, 101:1–23, 2016.
- 12 Elshad Karimov. *Trie Data Structure*, pages 67–75. Apress, Berkeley, CA, 2020. doi: 10.1007/978-1-4842-5769-2_9.
- 13 Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in rdf data sources. In *International Conference on Conceptual Modeling*, pages 481–495. Springer, 2015.
- 14 Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM computing surveys (CSUR)*, 51(3):1–34, 2018.
- 15 Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. URL: <https://www.w3.org/TR/rdf-sparql-query/>.
- 16 David Wood Richard Cyganiak and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. URL: <https://www.w3.org/TR/rdf11-concepts/>.

Metaobject Protocols for Julia

Marcelo Santos  

Instituto Superior Técnico, University of Lisbon, Portugal

António Menezes Leitão  

INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal

Abstract

Metaobject Protocols enable programmers to extend programming languages without the need to understand the lower level details of their implementation. However, designing these protocols comes with two challenges: allow programmers to limit their concerns to higher level concepts and minimize performance penalties in programs. In this work, we propose metaobject protocol for the programming language Julia. Julia’s object system is very limited, when compared to languages following the Object-Oriented paradigm. However, Julia’s compilation approach allows for a considerable degree of code optimization through the exploration of runtime type information. Through the usage of Julia’s run-time optimizations, we propose a metaobject protocol that combines user-extensibility with limited performance penalties. This paper focuses on the development of a multiple inheritance method dispatch and method combination mechanisms with zero runtime overhead.

2012 ACM Subject Classification Software and its engineering → Language features

Keywords and phrases Julia, Metaobject Protocols, Object-Oriented Programming, Performance

Digital Object Identifier 10.4230/OASICS.SLATE.2022.13

Supplementary Material *Software (Source Code)*: <https://github.com/tosmarcel/julia-mop>
archived at `swh:1:dir:b1f9259d39045b00963846ba7d4b8a3dd3829971`

Funding This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017.

1 Introduction

Traditionally, there has been a separation between programmers and language designers. Programmers treat languages as black boxes with their own rules, as established by the language designers. In this model, the semantics of a language is seen as unchangeable, thus imposing limits to its expressiveness.

Metaobject Protocols (MOPs) come to blur the distinction between programmers and language designers, by providing programmers with an interface to modify the language. By treating the language itself as a mutable object-oriented program, one can alter the semantics and introduce new behaviours to the language through the abstractions made available by the OOP paradigm. We provide two examples as motivation for the use of MOPs:

- As shown in [8], a programmer might be discontent with the performance of the implementation of slot accessing for objects of class A, but an implementation satisfying performance requirements for class A could damage performance for slot accessing for objects of class B. This, in turn, demonstrates the impossibility of the creation of an implementation satisfying every programmer’s needs. With the use of MOPs, one can define a specific implementation for only one class and leave the default implementation for the remaining classes.
- With MOPs, it becomes fairly straightforward to add new behaviour to existing code without directly modifying it. Suppose that it becomes a requirement to serialize objects, i.e., save their data to disk, each time they’re modified. One can leverage the interfaces provided by MOPs to intercept each time object’s slots are changed and thus serialize the object’s state as desired. One could accomplish this task with plain OOP strategies but it



© Marcelo Santos and António Menezes Leitão;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 13; pp. 13:1–13:15
OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

would require the creation of frontends for slot modification to be able to track changes. This would, in turn, result in the mixture of code relevant to the object's nature and code responsible for serialization, two very distinct concerns. With MOPs, the handling of these concerns would trivially be separated.

A prime example on the implementation of MOPs is the one present in the Common Lisp Object System (CLOS).

1.1 The Julia Programming Language

One of the main problems faced in the implementation of the CLOS metaobject protocol was guaranteeing good performance alongside the flexibility of the MOP [9]. This is a recurring problem in the implementation of higher-level languages, known as the Paradox of High-Level Languages [9]. The main premise of high-level languages is that they allow programmers to better formulate what their programs do through more expressive methodologies. Consequently, compilers for these languages should be able to exploit this knowledge and output faster programs than their low-level counterparts. But reality lies on the opposite side, meaning that there are concepts which are, in fact, not being expressed more clearly, instead requiring programmers to provide sufficient detail to enable efficient execution. We can further divide languages in two categories: static and dynamic languages. Dynamic languages are seen as higher level than static languages, as they allow us to express ideas harder to write in static languages. In the most recent years we've seen a fast growth in popularity of languages featuring object-oriented and dynamic properties [11, 5]. This is due to the increased productivity felt by developers. Python, one of these languages, is used extensively in the fields of numerical and scientific computing, which often require large-scale computations to be executed. One of the major complaints regarding Python is that it is slow [12], so, to resolve that problem, a few strategies are applied:

- Create libraries relying on faster languages to process more intensive operations (e.g. Numpy, a numerical computing library for python which relies on calling C code);
- Optimize the underlying compiler/interpreter (e.g. Pypy, an alternative python implementation featuring Just-in-Time (JIT) compilation);
- Prototyping in this high-level language and then translating the code to a more performant and often lower level language like C, C++, or Fortran. This last route is the one yielding the best results performance-wise, but it falls short for relying on a dichotomy of languages, requiring more knowledge and work from the programmer.

The Julia programming language [2], a dynamic language with a focus on performance, tries to solve this problem. By employing strategies like JIT compilation and code specialization on run-time types, it achieves outstanding results compared to other dynamic languages like Python, while being close to very performant languages like C++ [1].

Multiple dispatch is used extensively in Julia. It is the dynamic dispatch of methods based on the run-time information of all the arguments. Multiple dispatch is applied when calling generic functions to choose which method to dispatch. "A generic function is a function whose behavior depends on the classes or identities of the arguments supplied to it. The methods define the class-specific behavior and operations of the generic function"[3]. This allows for extension of the language by creating more methods for a generic function.

Julia features a very basic object system. Subtyping can only be accomplished from abstract types, the equivalent of abstract classes in Java. So, inheritance must be planned ahead, if one were intending to inheriting behaviour from a struct (concrete class).

1.2 Objectives

Throughout computer language history we have seen a demand for the development of Object Systems atop existing languages. The Lisp community saw the development of CommonLoops [4], which joined Lisp’s procedure-oriented paradigm with object-oriented programming. The same evolutionary strategy was applied to Objective-C, which stemmed from the C language and became a prime example of an OOP language by just adding a small number of syntactic features taken from the Smalltalk language while keeping compatibility with the remaining aspects of C [7].

In this work, we focus on the Julia Programming Language. Currently, the Julia community uses composition as a mechanism to express the sharing of functionality between concrete types. We will design and implement, in Julia, a performant MOP alongside a more expressive Object System.

2 Related Work

The Common Lisp Object System is an object-oriented extension to the Common Lisp[13] language.

CLOS decouples methods from objects through the usage of generic functions. Generic functions are functions whose behaviour depends on the types of the arguments supplied to them. A method defines the behaviour of a generic function for a set of arguments. This object system allows the extension of methods through the definition of method combinations. A method combination results in the application of auxiliary methods triggered by the calling of primary methods, the methods to which auxiliary methods connect to.

CLOS also allows the implementation of mixins, which, like inheritance, allow the sharing of functionality, but without the relationship of subclass. This pattern avoids ambiguity issues related to multiple-inheritance.

CLOS provides a way of changing objects structure through the redefinition of classes at run-time. When a class is redefined, changes are propagated to all instances and the instances of its subclasses[3].

2.1 Metaobject Protocol Implementations

Although many languages leverage MOPs to extend their functionality, we will only focus on the most expressive one, the CLOS MOP, and another one which stands out for its different implementation.

2.1.1 CLOS

The motivation for the development of the CLOS MOP came from the need to give developers the ability to modify the language from a high-level perspective, i.e., without low-level knowledge of the inner workings of the language. The CLOS language provides ways to incrementally change the behaviour of fundamental language constructs. This is possible through the reification of elements such as **class**, **generic-function**, and **method**. These elements are said to be metaobjects and their exposure and consequent modification is what allows for the manifestation of changes to the language semantics. As is often the case, applying global changes to an already existing system without proper testing can lead to unforeseen results. Furthermore, a programmer’s intent might be to just change a portion of the language for a specific section of the program and not its entirety. This problem is

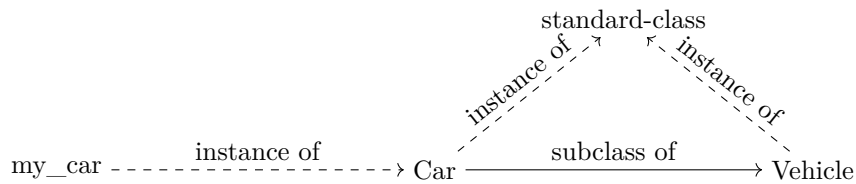
13:4 Metaobject Protocols for Julia

solved by CLOS by exposing metaobjects as part of an object-oriented hierarchy capable of being extended. As such, one can look at the semantics of the language as an object-oriented program as well.

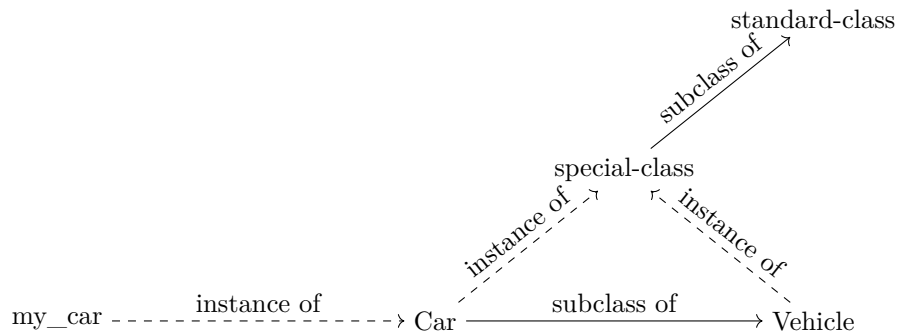
In a traditional class-based object-oriented language, objects, e.g. a car, and classes, e.g., Car and Vehicle, are related through the instance-of and subclass-of relations, as follows:



The implementation of a MOP integrates metaobjects in the system through the reification of classes. Classes are now themselves objects, instances of another class:



The class **standard-class** is said to be a **class metaobject class**, or a **metaclass**. It defines the behaviour of its instances, class metaobjects (**Car** and **Vehicle** in this example), which in turn define the behaviour of regular objects (**my_car** in this example). We can define a subclass of **standard-class** to specify new semantics for the classes **Car** and **Vehicle**, rendering the relationship between the objects as such:



Because **standard-class** still exists and is left unmodified, all other class metaobjects will exhibit the same behaviour as before the creation of **special-class**. Furthermore, only classes explicitly told to have **special-class** as a class metaobject class will follow its defined rules. This MOP allows one to only change a subset of class behaviour, as the remaining functionality will be provided by parent metaclasses, i.e., **standard-class** will be responsible for handling functionality not overridden by **special-class**, as is usual in the inheritance relation of object systems. The selective nature of metaclass specialization is what allows for the incremental extension of programming languages.

Listing 1 presents a concrete CLOS example of metaclass specialization from [9]: We explain how the MOP was used to achieve a modification on the semantics of object instantiation:

- We begin by creating a new metaclass **counted-class** whose main goal is to give classes the ability to track how many times they have been instantiated. It is a subclass of **standard-class**, the default metaclass. We give it a slot, **counter** initialized to zero to track this number, which will be present in all classes that are instances of counted-class.

■ **Listing 1** CLOS Metaclass Example.

```
* (defclass counted-class (standard-class)
  ((counter :initform 0)))

* (defclass counted-rectangle () ()
  (:metaclass counted-class))

* (defmethod make-instance :after
  ((class counted-class) &key)
  (incf (slot-value class 'counter)))

* (slot-value (find-class 'counted-rectangle) 'counter)
0
* (make-instance 'counted-rectangle)
#<COUNTED-RECTANGLE {10042DAC03}>
* (slot-value (find-class 'counted-rectangle) 'counter)
1
```

- Then, we define a class, **counted-rectangle**. This macro also defines it as a metaobject and sets **counted-class** as its metaobject class. The relationship between the classes is as follows:

counted-rectangle $\overset{\text{instance of}}{\dashrightarrow}$ counted-class $\xrightarrow{\text{subclass of}}$ standard-class

- We define a method combination for the **make-instance** generic function. Method combinations are a mechanism through which one appends functionality to existing generic functions. In this case, we are adding instructions to be executed after the default implementation runs. This version of **make-instance** is specialized for arguments which are instances of **counted-class**. This method specifies what happens when a new instance is created. In this example, we are incrementing the **counter** slot. This step illustrates the essence of incremental modification - create methods that specialize for the desired metaclasses.
- The remaining instructions demonstrate how the **counter** from the **counted-rectangle** class has been updated after creating an instance of **counted-rectangle**.

The CLOS MOP exposes a reflection interface that enables the program to understand itself while it is running. In the example above, we used **find-class** to retrieve a class metaobject from its name. This mechanism is essential to achieve the reification of language constructs, which is needed for the modification of the language. Besides instantiation, the protocol allows for the inspection and control of the following concepts:

- **Class precedence lists** - the modification of these structures permits the reordering of the priority of superclasses. Given that some languages possessing multiple-inheritance exhibit different ordering schemes, this allows, for example, to port libraries from languages similar to CLOS while maintaining inheritance priority compatibility.
- **Slot Access** - this defines what actions are taken when attempting to access an object's slots.
- **Instance Allocation** - how instances are allocated. The motivation example given in Section 1 would have to resort to this interface in order to accomplish its goal.

13:6 Metaobject Protocols for Julia

■ **Listing 2** Julia Generic Functions and Methods.

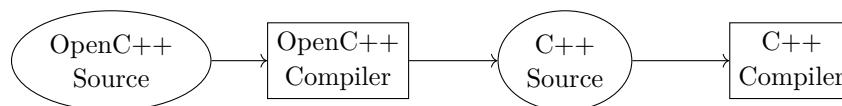
```
f(x::Integer) = x + 1
f(x::Real) = x + 2
```

The reference [9] provides an extensive list of capabilities and methods of the CLOS MOP.

2.1.2 OpenC++

One of the main problems of the CLOS MOP is the performance hit taken. The cause for this problem is the existence of metaobjects at run-time, which are needed to change the semantics of the language. Although this allows for greater flexibility, it increases the level of additional run-time logic.

OpenC++[6] is a metaobject protocol extension to C++[14] similar to CLOS but with a very different approach. It moves all the logic of the MOP to compile time. The purpose of this move is to incur zero run-time speed or space overhead, while still allowing the compiler to perform optimizations. The basic system architecture is as follows: The OpenC++



compiler generates the metaobjects responsible for executing the protocol when compiling OpenC++ to C++. These metaobjects intercept the parsing of regular C++ entities (e.g., class definitions, member access, object creation, virtual function invocation) and take control of their compilation. In essence, these metaobjects modify the program's semantics through the manipulation of parse trees. The resulting modifications are then passed as a regular C++ source to a C++ compiler which doesn't have knowledge of the MOP. With this strategy, no metaobjects exist at run-time, thus saving time and memory. One big disadvantage of this approach is the fact that since the protocol only acts at compile time, it becomes impossible to apply the same changes at run-time. Another minor problem is the increase of time and complexity of compilation.

This solution, which tries to solve the performance issues from metaobject protocols, seems to hint the existence of a tradeoff between execution speed and flexibility, similar to the aforementioned Paradox of High-Level Languages.

2.2 Julia

In this section, we present features of Julia that we consider relevant for our proposal. These include patterns which allow us to build an object system and performance optimizations capable of minimizing the overhead of our implementation.

Julia provides the concepts of generic functions and methods just like in CLOS[3]. Generic functions are implicitly declared through the definition of methods.

We now concern ourselves with method specialization. It is an aggressive mechanism employed by the Julia run-time which, combined with JIT compilation, generates compiled code and caches functions based on their arguments. In Listing 2, we create a generic function **f** with two method implementations: one for **Real** and its subtypes and another for **Integer** and its subtypes. In Listing 3 we perform the respective method calls.

■ **Listing 3** REPL evaluation of generic function `f`.

```
julia> f(1)
2
julia> f(1.0)
3.0
```

■ **Listing 4** Inspection of methods' IR.

```
julia> @code_llvm f(1)
define i64 @julia_f_163(i64 signext %0) {
    %1 = add i64 %0, 1
    ret i64 %1
}

julia> @code_llvm f(1.0)
define double @julia_f_182(double %0) {
    %1 = fadd double %0, 2.000000e+00
    ret double %1
}
```

The Julia programming environment allows us to inspect the Intermediate Representation (IR) of language constructs. IR is a higher level representation of assembly used by LLVM[10], the section of Julia's implementation responsible for generating architecture-specific machine code and applying lower level optimizations. Listing 4 shows that although methods have the same name, they exist in different sections of memory and possess distinct behaviour, as one can see through the IR output by the `@code_llvm` macro.

As a dynamically typed language, Julia allows the programmer to omit types of parameters in function definitions. We can take the example in Listing 4 and generalize it in Listing 5. Intuitively, this method would result in the generation of generic code capable of receiving arguments of any type. By handling a broader set of types, we lose the opportunity to exploit type-specific information that would allow the compiler to optimize this method. Luckily, the approach taken by the developers of the language is far different from this.

We can observe in Listing 6 that each method call executes different code. What this shows is that Julia implicitly employs a JIT strategy to compile at run-time type specific versions of a method according to its arguments. Consequently, exploiting type information becomes possible.

2.2.1 Method redefinition

Immutability, of which Julia takes advantage of, comes as a decisive factor when it comes to compiler optimization. The more structures that are guaranteed to stay constant, the more

■ **Listing 5** Dynamic Typing.

```
g(x) = x + 1

g(1)      # 2
g(1.0)    # 2.0
```

13:8 Metaobject Protocols for Julia

■ **Listing 6** Specialization of generic function.

```
julia> @code_llvm g(1)
define i64 @julia_g_167(i64 signext %0) {
    %1 = add i64 %0, 1
    ret i64 %1
}

julia> @code_llvm g(1.0)
define double @julia_g_186(double %0) {
    %1 = fadd double %0, 1.000000e+00
    ret double %1
}
```

optimizations the compiler can perform. A function accessing variables from the outside will have completely different assemblies depending on the mutability of those variables. If we can assert the immutability of our system, the faster it will run. However, given the nature of run-time metaobject systems, the opposite happens. The ability of changing program structure at run-time is one of the main benefits of using MOPs. Luckily, we can exploit another feature of Julia: function redefinition.

The redefinition of a method changes its behaviour and triggers the recompilation of methods which depend on it. Redefinition allows for behaviour changes without relying on data structures to hold mutable data. This also provides fast access to data without compromising flexibility.

Julia is thus inserted in the group of languages allowing the modification of a program with zero downtime, which opposes the traditional model of shutting down, recompiling, and rerunning programs.

2.3 Problem

This section has shown two Metaobject Protocol systems and exposed a dilemma between them: either opt for a flexible system or a performant one. The nature of these two implementations focuses on the time-frame in which metaobjects exist: run-time or compile-time. Our goal is to bridge the gap between these two architectures and create a performant run-time metaobject protocol on top of the Julia language and its optimizations.

3 Solution

In this section we will describe our implementation proposal for multiple-inheritance method dispatch and method combination mechanisms with zero run-time overhead for the Julia language, without changing its implementation or semantics.

3.1 Class Definition

We begin by describing our class definition mechanism. One can define a new class by calling the `@defclass` macro. It takes the name of the class to be defined and a list of superclasses from which it inherits behaviour. In Listing 7, we define four classes: **A**, which has no superclasses, **B** and **C** which both inherit from **A**, and **D** which inherits from **B** and **C**. This macro is responsible for ensuring the following:

■ **Listing 7** Class definition.

```
@defclass A ()
@defclass B (A,)
@defclass C (A,)
@defclass D (B, C)
```

■ **Listing 8** Defining and calling methods (following Listing 7).

```
> @defmethod bar(a::A) = 0
> bar(D())
0
> @defmethod bar(b::B) = 1
> @defmethod bar(c::C) = 2
> bar(D())
1
```

- Create a Julia structure with the name of the class. This structure has no supertypes, even if it is defined with superclasses.
- Define the method **superclasses** which takes the Julia type for the class and returns the types corresponding to the superclasses. If the class is defined without superclasses, its only superclass is **Any**, the type in Julia of which every type is a subtype of.
- Define the method **prelist** which takes the Julia type for the class and returns the precedence list for the class. A precedence list is an ordered list of classes which determines which are more specific than the others. This is necessary to determine from what classes objects inherit behaviour. By default this order is determined by applying a topological sort to the class hierarchy, similar to CLOS.
- Define the method **classof**, which when receiving an instance of the class, returns the class itself. This method is analogous to Julia's **typeof**.

These mechanisms for retrieving class information rely on methods and not mutable data structures or constants because:

- Although constants allow for the optimization of generated code, they forbid future changes, which goes against the purpose of metaobject protocols.
- Even though mutable data structures would allow for changes in the class system, they prevent the same optimizations done to constants.

Methods whose only purpose is to return data give us the best of both worlds: mutability and optimizations. The only cost for mutability is recompilation time.

3.2 Method Dispatch

We now describe how we use Julia's language feature to build a multiple-inheritance method dispatch mechanism. We provide the **@defmethod** macro, that, just like regular Julia method definitions, takes a method name, the list of arguments and optionally their types, and the method body, as we can see in Listing 8.

This macro is responsible for:

- Store an anonymous function taking generic arguments with the same method body as the one passed to **@defmethod** macro. This is stored in a key-value manner, in which the key is the list of the types of the parameters and the value is the anonymous function.

13:10 Metaobject Protocols for Julia

■ Listing 9 Method combination.

```
> @defmethod foo(d::D) = println("Primary method")
> foo(D())
Primary method
> @defmethod foo::before(d::D) = println("Calling before")
> @defmethod foo::after(d::D) = println("Calling after")
> foo(D())
Calling before
Primary method
Calling after
```

- Create a julia method, the method computer, taking generic arguments with the same number of arguments as the one specified with **@defmethod** whose purpose is to select the appropriate method to apply given the types of the arguments. If the method computer already exists, it is redefined to include the update of the available anonymous functions.

3.2.1 The Method Computer

The method computer is where most of the dispatch work takes place. It executes the following steps:

- Gather a list of the types of the arguments supplied to the method.
- Get the list of methods whose parameters are compatible with the arguments.
- Sort this list of methods in order to obtain the most specific method.
- Call the most specific method by passing it the arguments received and return its return value.

We say that a list of parameters P is compatible with a list of arguments A if:

- The length of P is equal to the length of A .
- For each pair (P_k, A_k) where P_k is the k^{th} parameter and A_k is the k^{th} argument, P_k is in the precedence list of A_k .

A method M is more specific than some method N , with respect to a list of arguments A if M has the smallest k for which M_{P_k} comes before N_{P_k} in the precedence list of A_k , where M_{P_k} is the k^{th} element of the parameters of M , N_{P_k} is the k^{th} element of the parameters of N , and A_k is the k^{th} argument.

3.3 Method Combination

Currently we support the same method combination implemented by the CLOS Standard Method Combination: before, after and around methods. See Listing 9 for an example.

Besides selecting the most specific method, the Method Computer must also take into account method combination. To do so, it must:

- Separate the method lists into four groups: before, after, around and primary. Primary methods are those defined without any method combination specifier.
- Apply the same filtering and ordering from the arguments for each group.
- Generate and call an effective method, which comes from applying all valid methods from the combination.

■ **Listing 10** Joining anonymous functions with `join_lambdas`.

```
function join_lambdas(around::Tuple, b::Tuple, p::Tuple, a::Tuple)
  (x...) -> begin
    next = join_lambdas(around[2:end], b, p, a)
    callnextmethod() = next(x...)
    callnextmethod(y...) = next(y...)
    hasnextmethod() = true
    around[1](x..., callnextmethod, hasnextmethod)
  end
end
```

3.3.1 Computing the Effective Method

The effective method is an anonymous function returned by the recursive method `join_lambdas`. `join_lambdas` receives four arguments, one for each method group. Each call to `join_lambdas` attaches one method from one group, while processing one group at a time. The order for processing groups is the same as the one specified by CLOS' method combination semantics: `around`, `before`, `primary`, and `after`.

The method `join_lambdas` has a different way of joining methods depending on which group it is currently processing:

- While processing a method from the `around` group, return an anonymous function which calls the method being processed. Besides receiving the arguments passed to the effective method, the method being processed also receives two functions as arguments: `nextmethod` and `hasnextmethod`. `nextmethod` is the next recursive call to `join_lambdas`. `hasnextmethod` returns true if there are more methods following in the combination and false otherwise. The `join_lambdas` method for processing the `around` group is shown in Listing 10.
- Processing the `before` group is much simpler. The only concern of the returned anonymous function is calling the current method and recurring into the `join_lambdas` call.
- Handling the `primary` group is very similar to the `around` group, given that it must allow calling `hasnextmethod` and `nextmethod`. The biggest difference from the `around` group processing is storing the return value from the call and returning it only after the `after` group.
- Processing the `after` group is identical to what is done to the `before` group.

We follow these steps instead of an iterative method calling approach because the JIT can better optimize method call chains. The resulting effective method of `foo` in Listing 9 can be seen in Listing 11.

3.4 Integrating Julia Types

Since the classes created by our system are Julia types, the preexisting Julia types are automatically partially supported. To fully integrate them, we need to define for the regular types the same methods we define for classes and their instances:

- `superclasses` is equivalent to calling the built-in function `supertype` and returning a single element list;
- `preclist`, the precedence list of a type, is equivalent to returning a list of every supertype until the `Any` type is reached;
- `classof` is the same as calling the built-in `typeof`.

With all of these equivalences in place, we can create methods through our `@defmethod` macro with regular Julia types.

13:12 Metaobject Protocols for Julia

■ **Listing 11** Resulting effective method of `foo` from Listing 9 (simplified).

```
(x1...) -> begin
    ((d) -> println("Calling before"))(x1...)

    ((x2...) -> begin
        res = ((d) -> println("Primary method"))(x2...)

        ((x3...) -> begin
            ((d) -> println("Calling after"))(x3...)
        end)(x2...)

        return res
    end)(x1...)
end
```

■ **Listing 12** IR code from calling `bar(D())`.

```
define i64 @julia_foo_1266() #0 {
top:
    ret i64 1
}
```

4 Evaluation

We now proceed to analyse the performance of our solution. We will take two approaches. First, to look at the LLVM Intermediate Representation (IR) language, a higher-level assembly, generated by the JIT. Second, to compare execution times with CLOS. The following tests were executed on an Intel Core i5-8250U 3.4GHz PC with 16GiB of RAM running the Linux operating system.

4.1 Generated IR

We can analyse the performance of our method dispatch mechanism by taking the code from Listing 8 and reading its IR in Listing 12. As we can see, no computation from method dispatch is present in the end result. Only the relevant method body remains. We can thus conclude that our multiple-inheritance method dispatch is just as performant as Julia's single-inheritance method dispatch.

The overhead resulting from computing the effective method in Listing 9 can be seen in Listing 13. Just like in the previous example, no overhead is present. Not only that, but there are also no intermediate method calls being made from the chain produced by `join_lambdas`. The bodies from each method were joined into a single body. Only the code relevant for printing is displayed. This makes the use of our method combination mechanism equivalent to creating a regular Julia method with a body as the concatenation of the bodies of the methods defined for the combination. This regular Julia method, defined in Listing 14 yields the same IR as the method combination in Listing 9, shown in Listing 13, thus having the same execution times.

■ **Listing 13** IR code from calling the method combination of `foo(D())` (simplified).

```
define void @julia_foo_1172() #0 {
top:
    %0 = alloca {}, align 8

    store {}* inttoptr (i64 139884251139280 to {}*), {}** %0, align 8
    %1 = call nonnull {}* @j1_println_1178(...)

    store {}* inttoptr (i64 139884243053904 to {}*), {}** %0, align 8
    %2 = call nonnull {}* @j1_println_1179(...)

    store {}* inttoptr (i64 139884260369712 to {}*), {}** %0, align 8
    %3 = call nonnull {}* @j1_println_1180(...)

    ret void
}
```

■ **Listing 14** Regular Julia method equivalent to method combination in Listing 9.

```
function foo_effective(d::D)
    println("Calling before")
    println("Primary method")
    println("Calling after")
end
```

4.2 Execution time against CLOS

The Steel Bank Common Lisp (SBCL) is a high-performance Common Lisp implementation. In this section we will compare an ad-doc example of method dispatch between the SBCL CLOS and our Julia solution. For this example, we iterate a list of objects to force method dispatch. We have the Julia version in Listing 15 and the SBCL CLOS in Listing 16. Both versions assume a class hierarchy like the one specified in Listing 7. Each iteration example was ran once. The results for calling `foo` with `arr` as an argument yields times of **0.409013** seconds for Julia and **0.229447** seconds for CLOS, which means SBCL is more optimized. However, Julia only exhibits a worse time because it cannot infer the types of the elements of `arr`. If we specify `arr` as being an array of `D`, then we get a much better result: **0.000012** seconds.

■ **Listing 15** Julia example.

```
@defmethod baz(b::B, n) = n+1
@defmethod baz(c::C, n) = n*2
arr = []
for i in 1:10000000 push!(arr, D()) end
foo(a) =
    let y = 0
        for e in a
            y += baz(e, 10)
        end
    y
end
```

■ **Listing 16** CLOS example.

```
(defmethod baz ((b B) n) (+ n 1))
(defmethod baz ((c C) n) (* n 2))
(defparameter arr (make-list 10000000
                             :initial-element (make-instance 'D)))
(defun foo (a)
  (let ((y 0))
    (loop for e across a do
      (incf y (baz e 10)))
    y))
```

5 Future Work

The work presented in this paper is an element of a metaobject protocol being implemented in Julia. Our future work is to continue adding features to our implementation in order to get closer to levels of expressiveness of CLOS.

6 Conclusion

In this paper, we discussed metaobject protocols. We analysed two different implementations choosing different trade-offs in terms of performance and expressiveness. Through the described optimizations of the Julia language, we proposed a zero run-time overhead solution to two main pieces of metaobjects protocols: multiple-inheritance method dispatch and method combinations.

References

- 1 S Borağan Aruoba and Jesús Fernández-Villaverde. A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58:265–273, 2015.
- 2 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.
- 3 Daniel G Bobrow, Linda G DeMichiel, Richard P Gabriel, Sonya E Keene, Gregor Kiczales, and David A Moon. Common lisp object system specification. *ACM Sigplan Notices*, 23(SI):1–142, 1988.
- 4 Daniel G Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *ACM Sigplan Notices*, 21(11):17–29, 1986.
- 5 Stephen Cass. The 2015 top ten programming languages.
- 6 Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, 1995.
- 7 Brad J Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- 8 Gregor Kiczales, J Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G Bobrow. Metaobject protocols: Why we want them and what else they can do. *Object-Oriented Programming: The CLOS Perspective*, pages 101–118, 1993.
- 9 Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. *The art of the metaobject protocol*. MIT press, 1991.
- 10 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

- 11 Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.
- 12 Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3136014.3136031.
- 13 Guy Steele. *Common LISP: the language*. Elsevier, 1990.
- 14 Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

The Visual Programming Environment ROBI for Educational Robotics

Gustavo Galvão ✉

Centro ALGORITMI, Departamento de Informática, Universidade do Minho, Braga, Portugal

Alvaro Costa Neto ✉ 

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, Barretos, Brazil

Cristiana Araújo ✉ 

Centro ALGORITMI, Departamento de Informática, Universidade do Minho, Braga, Portugal

Pedro Rangel Henriques ✉ 

Centro ALGORITMI, Departamento de Informática, Universidade do Minho, Braga, Portugal

Abstract

This paper presents the outcomes of a research project focused on the training of Computational Thinking, resorting to a block-based visual programming language created to program an Arduino Uno based robot. To support the design and implementation of the visual programming environment Robi, we start discussing the relevance of Educational Robotics to motivate and engage children in programming activities. Students usually face great difficulties to learn computer programming and it is nowadays accepted that young people shall be trained in Computational Thinking to acquire the skills necessary to easily solve problems within and beyond the realm of Computer Science and Engineering. The resolution of obstacles imposed by the costs and reduced availability of typical Educational Robotics kits, in combination with the benefits of existing block-based programming languages, like simplicity and intuitiveness, motivated the project here reported and analyzed. We aim at showing that Robi, a visual block-based programming language and robot programming environment, provides an easy, accessible and intuitive platform to learn how to solve problems programming a computer and support the training of Computational Thinking.

2012 ACM Subject Classification Software and its engineering → Compilers; Social and professional topics → Computing education

Keywords and phrases Programming Languages, Visual Languages, Computer Programming, Educational Robotics

Digital Object Identifier 10.4230/OASICS.SLATE.2022.14

Supplementary Material *Software (Web Application)*: <https://robi.di.uminho.pt/>

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

Computational Thinking was presented by Wing in 2006, stating that it is a method for solving problems, or designing systems and understanding human behavior, based on the fundamental concepts of Computer Science [25]. According to Wing, training in Computational Thinking is not just for Computer Scientists but for all people, and should be applied from an early age. Computational Thinking promotes the development of problem solving strategies, logical reasoning, problem decomposition, knowledge relationship, abstract thinking, among others [25]. It is believed that if children develop these skills from an early age, they will be able to solve problems more easily, whether in mathematics, physical chemistry, computer science or even in their daily lives.



© Gustavo Galvão, Alvaro Costa Neto, Cristiana Araújo, and Pedro Rangel Henriques; licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 14; pp. 14:1–14:15

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One way to develop Computational Thinking skills is to solve problems using block-based programming environments. By using these environments, in addition to solving problems, children also acquire Computer Programming abilities. Block-based programming environments leverage a programming primitive-as-puzzle-piece metaphor that provides visual cues to the user on how and where commands can be used as their means of constraining program composition. This type of visual programming consists of dragging blocks into a scripting area, and fitting them together to form scripts [24]. Some examples of block-based programming environments platforms used to introduce programming are: *Scratch*¹, *mBlock*², *VEXcode VR*³, *Alice*⁴. They provide a fun and engaging introduction to programming concepts without the need to deal with complex syntactic constructs, commonly considered obstacles to students that recently started learning text-based programming [14]. The *Scratch* and *mBlock* Programming environments allow integration with robot kits, such as *LEGO Mindstorms* and *mBot - Makeblock*, respectively.

The integration with robot kits provides more motivation and greater student engagement. However, despite the benefits of block-based programming environments, their integration with robot kits is absent or too expensive and complex for younger children.

Motivated by the benefits that robotics tools bring, which lead students to acquire not only Computational Thinking skills but also other necessary skills such as collaboration, cognitive capabilities, self-confidence, spatial perception and understanding [18], among others, we decided to develop a new block-based programming language integrated with an Arduino Uno⁵.

Arduino Uno is an open source micro-controller that can be easily programmed and reprogrammed at any time. The Arduino platform was launched in 2005 for hobbyists, students or professionals to create devices that interact with their environments using sensors and actuators in a simple and low-cost way [19]. One of the reasons why the Arduino platform is a low-cost product is due to the independence of the suppliers of parts and components. This is in contrast to robot kits like *LEGO*, *Makeblock* and other manufacturers, which have their own standard components, making them more expensive [17]. The programming language created aims to be simple, intuitive and as iconic as possible.

This article has six sections. In Section 2, Computational Thinking and Educational Robotics are presented. An overview of Robi Environment is provided in Section 3. Section 4 presents the visual programming language of Robi Environment. Section 5 discusses how to use robot Robi and make it to work controlled by our programming environment. Section 6 presents the system assessment. Finally, in Section 7, conclusions and future work are discussed.

2 Computational Thinking and Educational Robotics

Advances in technology made computing devices present in almost every aspect of daily lives. Essential to this development, Computer Programming presents itself both as a main topic and as one in which students face several difficulties to properly learn. By leveraging Computational Thinking (CT) it is possible to better prepare younger students and consequently improve their academic success later in life [22]. Being an important subject

¹ Accessible at: <https://scratch.mit.edu/>

² Accessible at: <https://mblock.makeblock.com/en-us/>

³ Accessible at: <https://vr.vex.com/>

⁴ Accessible at: <https://www.alice.org/>

⁵ Accessible at: <https://store.arduino.cc/>

of national education programs in many countries, to the extent of receiving classifications as national objectives [16], CT is recognized as a fundamental competency for success in the fields of Science, Technology, Engineering and Mathematics (STEM).

The International Society for Technology in Education & Computer Science Teachers Association [10], define Computational Thinking as a process of problem solving that includes the following characteristics: (1) formulate problems in a way that allows us to use a computer or similar to help solve them; (2) logically organize and analyze the data; (3) represent the data through abstractions such as models and simulations; (4) automate solutions through algorithmic thinking; (5) identify, analyze and implement possible solutions more effective; and (6) generalize and transfer this process to a wide variety of problems.

Teaching CT from an early age benefits students not only to overcome difficulties in programming courses, but also to improve skills that can be crucial in other areas, such as the development of critical thinking, abstract thinking, logical reasoning, problem solving strategies and persistence [15, 1]. To this intent, block-based programming environments are commonly used to keep children both interested and motivated when learning computer programming. As an example, the Hour of Code campaign, which provides online introductory programming activities, reaches millions of students through the use of several tools, including visual block-based programming environments such as Scratch [8].

A resource that has stood out to train students in Computational Thinking is Educational Robotics. Educational Robotics is a powerful and flexible teaching and learning tool, encouraging students to build and control robots using specific programming languages. The robot is a tangible object that students can interact with [9]. Recent reviews of the literature on Educational Robotics in the mandatory stages distinguish its potential for training CT and understanding concepts related to STEM areas [7, 12, 6, 2, 3, 4]. The use of robots in the classroom, in addition to being motivating, allows the design of activities that promote both CT and skills related to scientific and mathematical skills. In addition, it also enhances the development of skills: cognitive, digital, social, collaborative and teamwork, creativity [13, 5, 3, 4].

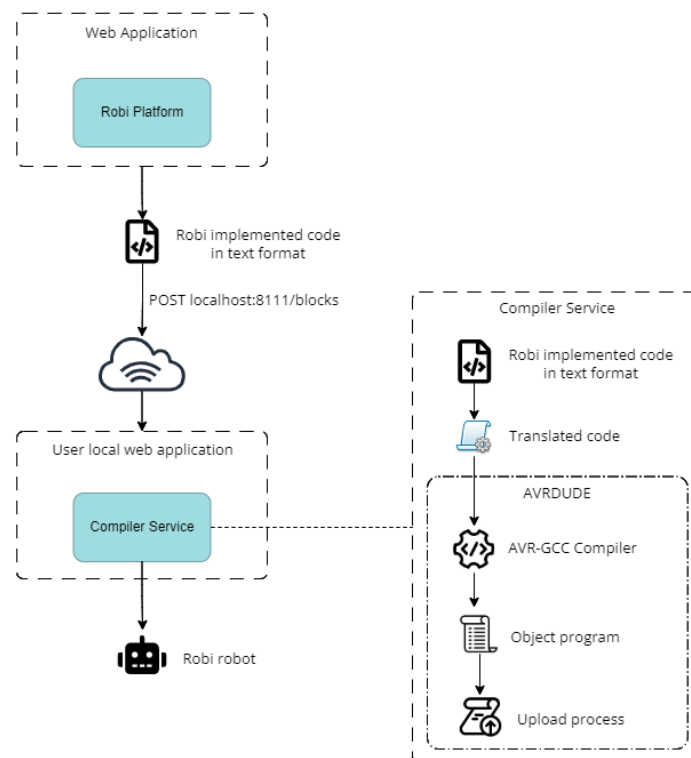
3 Robi, the Environment

Robi environment is composed of two web applications: one in the cloud, and one local to the user's computer⁶. The cloud-based web application is responsible for providing the visual programming environment to the user and for providing downloads to the local web application. The local web application contains a compiler service, that translates the intermediary code developed translated by the cloud-based web application to an Arduino compatible binary using the AVRDUDE tool, the same used by the official Arduino IDE. Figure 1 shows the system architecture as explained above.

3.1 System Requirements and Implementation Details

Some requirements were identified for the proper design of the visual programming language, as well as to develop the system architecture, both cloud-based and local. These included: an IDE for programming Robi using the block-based language, the hardware upload mechanism to send the downloaded script to an Arduino Uno based robot, and the definition of both the block-based language and its intermediary textual representation.

⁶ The official Robi website may be found at the following address: <https://robi.di.uminho.pt/>



■ **Figure 1** System architecture of the Robi platform.

Among the many languages, packages and tools that could be used to develop this project, given the authors' experiences in similar projects, the following were chosen:

- Angular for the development of the front-end;
- Java, Gradle and Spring boot for the development of the back-end;
- JUnit, WireMock and RestAssured for testing purposes;
- Spring Rest Docs and AsciiDoctor for documentation;
- AVRDUDE for the final compilation and upload to the robot's hardware.

4 Robi, the Visual Programming Language

Robi programming language was designed with the purpose of programming an Arduino-based robot using only blocks that represent desired robot actions. For this reason, this language is mainly composed of robot movement and sensor blocks. Besides these robot-oriented blocks, Robi also contains classic programming-oriented blocks, such as repetition and conditional structures, math operators, variable declaration and assignment, and a delay operation. The intent was to represent the robot behavior programming in a high level language, directly mapping the actions that users would expect to be executed by the robot itself.

While there would be advantages in adopting a known visual language, such as Blockly⁷, it was decided that Robi should have a visual identification specifically designed to its purposes. This decision was made based on the fact that Robi's language is directly aimed at controlling the robot and trying to reduce a general purpose language to fit its specifications could lead to a series of adaptation problems.

⁷ Accessible at: <https://developers.google.com/blockly/>

■ **Listing 1** Formal definition of the Robi Programming Language.

```

Program      : Variables INIT Block
Variables    : & | Variables VARIABLE
Block        : Instruction | Block Instruction
Instruction   : Condition | Assignment | Loop | MoveOp Expression
Conditional  : IF Condition THEN "[" Block "]"
              | IF Condition THEN "[" Block "]" ELSE "[" Block "]"
MoveOp       : FORWARD | BACKWARDS | TURN_LEFT | TURN_RIGHT
Loop         : DO Expression TIMES "[" Block "]"
              | DO WHILE Condition "[" Block "]"
Assignment   : UPDATE VARIABLE TO Expression
Condition    : Expression RelationalOp Expression
              | Condition LogicOp Condition
LogicOp      : AND | OR
RelationalOp : GREATER_THAN | LESS_THAN | EQUALS
Expression   : NUMBER | VARIABLE | SENSOR
              | Expression MathOp Expression
MathOp       : PLUS | MINUS | TIMES | DIVISION

```

4.1 Formal Definition

Being an actual programming language (albeit a domain-specific one), the Robi programming language was formally defined using a list of terminal symbols and a Backus-Naur Form (BNF) description of its grammar. The terminal symbols mainly represent the blocks available to the user: INIT, VARIABLE, IF, THEN, ELSE, FORWARD, BACKWARDS, TURN_LEFT, TURN_RIGHT, DO, TIMES, WHILE, UPDATE, TO, AND, OR, GREATER_THAN, LESS_THAN, EQUALS, NUMBER, SENSOR, PLUS, MINUS, TIMES and DIVISION.

A program is formally defined by the first rule (for symbol 'Program') in Listing 1. The INIT symbol corresponds to the *Init* block, always present in the visual code. The other lines, after the first one referred, describe the rest of the grammar rules that define Robi Language (see Listing 1). The symbols, both terminal and non-terminal, are self-explanatory as to their uses. The rules form a lambda complete language that allows for the construction of entire programs that correctly control the robot with known statements, such as variable declaration, conditional and repetition structures and mathematical and logical operators.

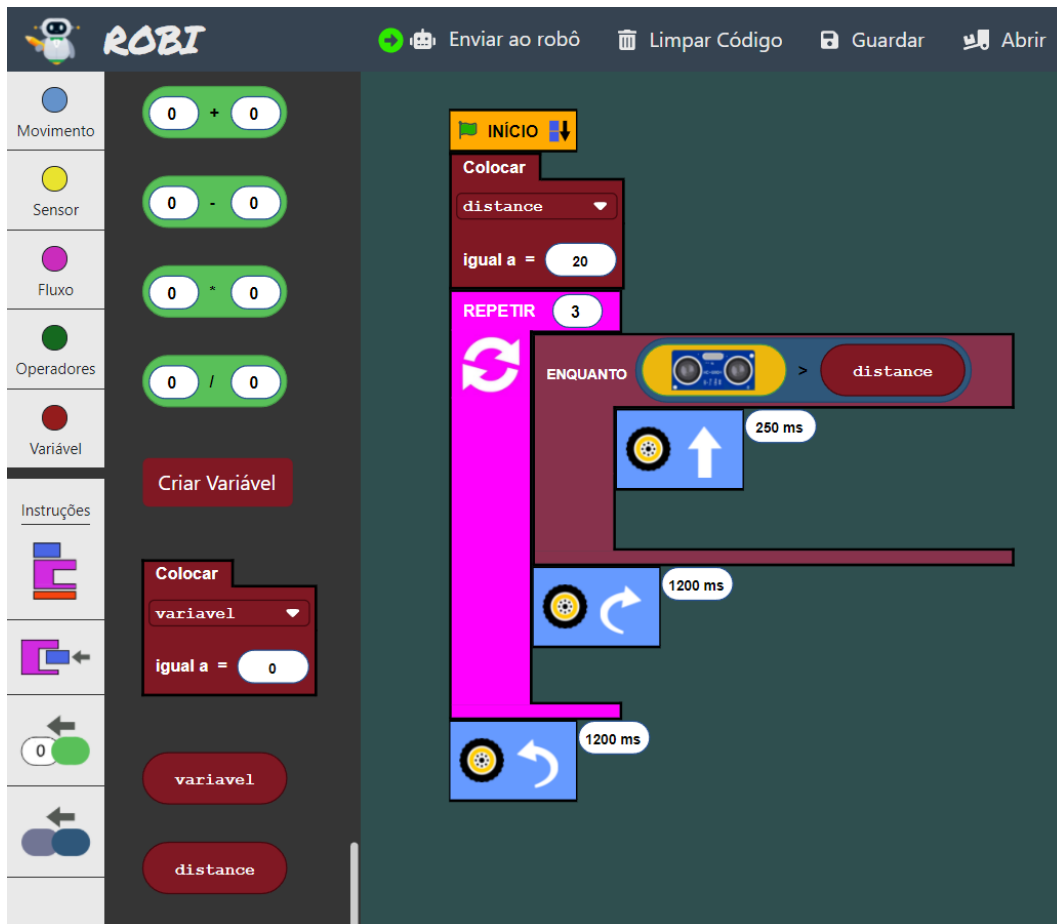
Listing 2 presents an example of a program formally derived from the formal grammar to move the robot in a specific pattern: move forward until an obstacle is detected, then turn right; repeat three times. Listing 2 resembles an ordinary source code written in a high level programming language, with the use of variables, execution flow statements and expressions. While it wasn't the purpose to have users to program directly using the formal definition of Robi's language, it was important to construct this definition in order to formally validate the structure of the visual language, which follows the same rules. The visual programming interface for the Robi environment and the block-based programming language equivalent of the example described in Listing 2 are shown in Figure 2. The interface itself and the graphic symbols (blocks) used to represent the language statements make heavy use of iconography, in order to facilitate the understanding and use of Robi by younger children.

The blocks are divided into five categories. The visual programming interface aims to facilitate its use by grouping blocks through similar colors (which may be seen on the left side of Figure 2). This technique provides visual cues to children as to what the block represents in a general sense. The five categories for the blocks are:

14:6 Robi, Visual Programming Environment for Educational Robotics

■ **Listing 2** Example of a concrete program formally written in Robi's formal language.

```
VARIABLE distance
INIT
UPDATE distance TO 20
DO 3 TIMES [
  DO WHILE SENSOR GREATER_THAN distance [
    FORWARD 250
  ]
  TURN_RIGHT 1200
]
TURN_LEFT 1200
```



■ **Figure 2** Program created in the Robi visual programming interface, equivalent to Listing 2.

Movement contains blocks for controlling the movement of the robot, such as moving forward or turning right (blue boxes with wheels in Figure 2);

Sensor the only block present in this category reads data from the ultrasonic sensor (yellow pill shaped block in Figure 2);

Flow controls program execution flow with encompassing blocks, such as the pink and dark rose containers in Figure 2;

Operators contains blocks for common mathematical, relational and logical operators, such as the dark blue pill shaped block in Figure 2;

Variables deal with declaration and assignment of variables, which may be named with a limit of twelve letter or digits (red blocks in Figure 2).

The interface works via drag and drop operations to add blocks to the programming area, while connections are made between blocks by snapping actions whenever they are in the proximity. In order to remove a block, a simple drag and drop operation from the programming area back to the list of blocks is necessary.

5 Robi, the Robot

The intention behind building a robot and connecting it to the programming interface is to provide children with a physical representation of the outputs that their programs actually produce. This tactile, real-world feedback is crucial in the learning process for younger children and greatly improves understanding for older students, specially when mediated by an instructor or teacher [20, 23].

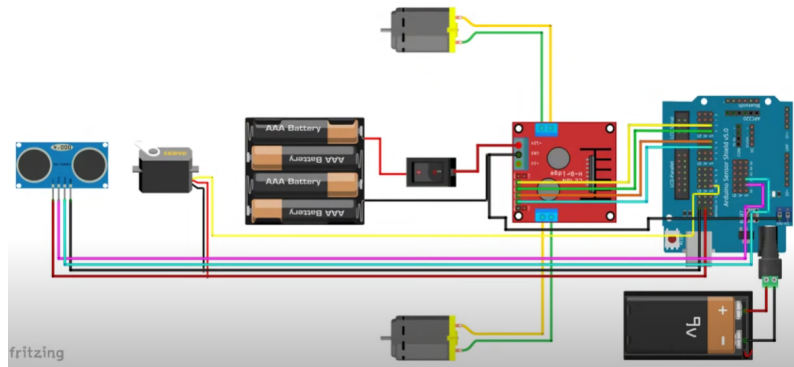
5.1 Components and the Platform

As said above, the robot was built with on an Arduino Uno platform, which not only has extensive on-line documentation, but also enables the easier creation of electronic and interactive objects. The Arduino ecosystem also contains a high amount of modules and sensors that can be easily integrated. The components used to build the robot are available in the official Robi website.

The set of electronic and mechanic components was purchased as a low-cost kit in a wholesale website both for economic and logistic reasons. By basing the robot around a readily available kit instead of building a custom platform, the whole project becomes more affordable and easier to complete, since it may be bought worldwide. Any compatible kit would also be suitable to use, even a custom built, as long as the same connections and configurations are made to the individual components or the low-level definitions are adapted to the new platform.

Figure 3, copied from [21], corresponds to the circuit diagram used as basis to create the robot electronics and hardware. The robot uses two different batteries: one set of four 1.5V batteries for the motors and a 9V battery for the Arduino Uno. The platform is robust, easy to build, and works almost immediately. One caveat of its current construction is the precision loss of simple DC motors, which could be improved using stepper or servo motors. This choice was made to simplify construction and lower the total cost of the robot, since more precise motors would probably be the most expensive part of the build.

The main component of the robot system is the Arduino Uno which receives the program from the computer and controls the rest of the circuit. The DC motor driver board is connected to both motors and the Arduino Uno (GPIO pins 2, 3, 4 and 5) and allows the rotation of each motor in both directions independently. The ultrasonic sensor is connected



■ **Figure 3** Electronic connections diagram for the Robi robot.

directly to the Arduino Uno through ports A1 and A2. These components and ports definitions are important for the configuration code that is used by the Robi environment to program the robot behavior.

5.2 Code Setup

There is a predefined configuration code for the robot to work as planned. In that code, some basic system initialization is done (Arduino ports are defined to connect the motors and the ultrasonic sensor, libraries are imported, and the ultrasonic sensor is calibrated) and some support functions are defined. Listing 3 presents the code that Robi adds to any program to bootstrap the Arduino system initialization and avoid having users worried about this part of the process.

In order to aid the translation process from the block-based programming language to standard Arduino source code, eight support functions are also predefined:

```
readPing returns a ping from the ultrasonic sensor measured in centimeters;
moveStop stops any movement;
moveForward moves forward indefinitely;
moveBackwards moves backwards indefinitely;
moveForwardCustom moves forward for a specific interval measured in milliseconds;
moveBackwardsCustom moves backwards for a specific interval measured in milliseconds;
turnLeft turns left for a specific interval measured in milliseconds;
turnRight turns right for a specific interval measured in milliseconds.
```

These functions assume a similar role to an instruction set of a classic target architecture in a compilation process. They are used in the translation service as basis for the intermediary textual representation of the block-based code. When compiled, some visual language blocks are translated to calls to these functions inside the main `loop` function of the Arduino platform, while others are translated to actual Processing directives, such as conditional and repetition statements.

5.3 Translation and Upload

The translation from the block-based code composed in the visual programming interface to the final binary instructions that can be uploaded to the the robot follows three steps:

■ Listing 3 Basic Arduino configuration code.

```
#include <Servo.h>
#include <NewPing.h>

const int LeftMotorForward = 5;
const int LeftMotorBackward = 4;
const int RightMotorForward = 3;
const int RightMotorBackward = 2;

#define trig_pin A1
#define echo_pin A2
#define maximum_distance 200

boolean goesForward = false;
int distance_from_ultrasonic_sensor = 100;

NewPing sonar(trig_pin, echo_pin, maximum_distance);
Servo servo_motor;

void setup() {
    pinMode(LeftMotorForward, OUTPUT);
    pinMode(LeftMotorBackward, OUTPUT);
    pinMode(RightMotorForward, OUTPUT);
    pinMode(RightMotorBackward, OUTPUT);

    servo_motor.attach(11);
    servo_motor.write(90);
    delay(2000);
    distance_from_ultrasonic_sensor = readPing();
    delay(100);
    distance_from_ultrasonic_sensor = readPing();
    delay(100);
    distance_from_ultrasonic_sensor = readPing();
    delay(100);
    distance_from_ultrasonic_sensor = readPing();
    delay(100);
}
}
```

1. The block-based visual code is evaluated in the cloud-based web application and translated into a JSON object – the equivalent of an intermediary representation in classic compilation terms – which is then sent to the local web application;
2. The local web application receives the JSON object (see Listing 4 for an example) and translates it into Arduino compatible source code by mapping the action nodes to either equivalent statements or predefined support functions. It then aggregates it to the configuration source code shown in Subsection 5.2 for final processing;
3. The local web application calls its compiler service to compile the Arduino compatible source code from the previous step into bytecode using the AVRDUDE tools. The compiler service then uploads the bytecode to the robot's Arduino Uno via USB.

With regards to the JSON object, every block represents an action. That action is the identifier that the compiler service uses to identify the original visual block and then map it to the correct code translation. If the action corresponds to one of the predefined support

■ Listing 4 JSON object for the example in Figure 2.

```

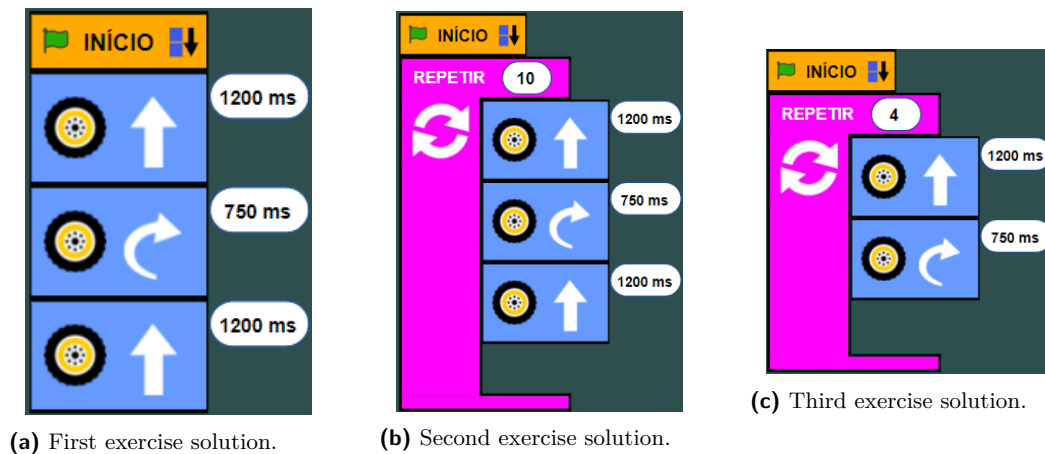
"blocks": {
  "0": {
    "action": "set_variable",
    "operatorExpression": undefined,
    "value": 20,
    "variableName": "distance"
  },
  "1" : {
    "action": "repeat",
    "blocksInside": {
      "0": {
        "action": "while",
        "blocksInside": {
          "0": {
            "action": "move_forward",
            "operatorExpression": undefined,
            "value": 250
          },
          "operatorExpression": "((readPing())>(distance))",
          "value": "0"
        }
      },
      "1": {
        "action": "turn_right",
        "operatorExpression": undefined,
        "value": 1200
      }
    }
  },
  "2" : {
    "action": "turn_left",
    "operatorExpression": undefined,
    "value": 1200
  }
},
"portCOM": "4",
"variablesInScript": ["variavel"]

```

functions listed in Subsection 5.2 the resulting mapping is just a call to that function. If, otherwise, the block represents another kind of statement (flow control structures, variable declaration *etc.*) than an appropriate snippet of code is mapped, taking into account necessary grammatical analysis of expressions. The last two lines of the JSON object represent the communication port used to upload the bytecode to the Arduino Uno and the list of variables used in the program.

6 Robi, Assessment

Robi's platform was tested by four children: two 7 years old, one 9 years old, and one 12 years old. Three exercises were proposed to be implemented by them in the Robi platform:



■ **Figure 4** Expected solutions for the three assessment exercises.

1. An introductory and tutorial exercise. The students should program the robot to move forward, turn approximately 90 degrees, and then move forward again. The purpose of this exercise was to introduce the basic concepts of block-based programming and the Robi visual programming interface. The expected solution for this exercise is presented in Figure 4a;
2. An intermediate exercise. The students should build upon the previous solution and order the robot to repeat the first exercise movement ten times. It was intended to teach students the importance of constructive and progressive learning, while introducing the concept of loops. The expected solution for the second exercise is shown in Figure 4b;
3. A final and more advanced exercise. It was requested from the students to teach the robot to draw a rectangle (or square). While it seems very similar to the previous exercise, considering only its solution, this exercise was described in a way to force the children to reason about the real world problem (moving the robot to draw something specific) and think about strategies to transform their solutions into software. The expected solution for the final exercise is presented in Figure 4c.

All four children were able to find a solution for the first exercise. One of the children (7 years old) took more time than the others, since he did not have practice using a computer. This fact reinforces the need for taking students background into account when teaching any kind of subject [11]. The other three children were able to solve the exercise without major difficulties.

Although the second exercise was solved by all children without any difficulties, none of them tried to search for a block that could simplify the script that they did not use in the first exercise. After explaining how loop works and showing that among the available blocks in the platform there were a few that could repeat a portion of code, they found and used the *Repeat* block. The 7 years old child that did not have practice with computer took a little more than the others at trying to connect the *Movement* blocks inside the *Repeat* blocks, meaning that the use of a computer graphical interface may pose a substantial obstacles to children that are not used to computers.

The last exercise was proposed in a way that the children had the possibility to come up with different solutions. A 7 years old child, different from the one without computer practice, asked about a way to simplify the act of writing the same values on all the blocks he added. The concept of variables was explained to him and he came up with the solution presented in



■ **Figure 5** Third exercise alternative solution.

Figure 5. The 9 years old child also solved this exercise without the use of the *Repeat* block, and the 12 years old child did use the *Repeat* block implying that more complex subjects such as repetition structures may pose a heavier cognitive burden on younger children. Every child had his or her own solution and none of them delayed solving this exercise neither had major difficulties finding a solution.

Based on the solutions and feedback given by the children, the Robi environment seemed to be intuitive and simple to use. The children were able to learn the basics of educational robotics and consequently the main programming concepts themselves, while having fun in using and teaching a robot to drive around.

7 Conclusion

A research work based on literature review evaluated if a new platform to support Educational Robotics would contribute to the current state of the art. After analysing some platforms in use throughout the world, it was noticed that simplicity, intuitiveness of use, and cost-effectiveness could be improved. We then concluded that developing Robi's language and programming visual platform was worthwhile, both to increase the user-friendliness of the platform and the ease of controlling a cheap but effective Arduino-based robot, as well as to secure the flexibility to adapt or upgrade the system in any direction.

Since the aim of this project was to develop a new programming language, the design of a grammar was necessary to formally guide the process. The creation of a grammar has the purpose of supporting the development of the new programming language. The grammar rules and symbols helped with the decision of blocks that Robi system would have and the conditions that the system needed in order to connect the blocks.

We decided to integrate the Robi platform with the Arduino Uno and its chassis car kit, because the Arduino Uno is a well-known open source microcontroller board and has a great amount of documentation and forums easily found all over the internet. Besides the documentation, the Arduino Uno and the electronic components that can be connected to it, offer a cheap approach if the desire is to build a robot. Arduino Uno can be programmed using the Arduino IDE, which uses the AVR-GCC compiler and the AVRDUDE tool. In order to compile the code the user writes using the visual programming language, Robi translates the visual script to adequate Processing source code and makes use of that AVR-GCC compiler. After the compilation is done, the bytecode is uploaded through AVRDUDE. It was decided early on that there would be no simulators for this project, because it would largely increase the project's scope and time requirements, and other similar systems already provide simulation tools.

Robi's interface was developed from scratch. All blocks were designed using the SVG technology, with the support of the Angular framework, which helped with the maintainability, readability and with the code reuse. The advantage of developing Robi from scratch is that the chosen approach provides flexibility to add, change, update or remove anything as the developer desires. Robi features require a flexible and powerful development, such as the visual interface reflecting the robot, or the interface that shall be as simpler as possible.

Different from the LEGO programming environment, Robi provides a web application solution, which opens opportunities to create, in the future, a class management system or any other advantage that a cloud solution provides.

The assessment of the Robi platform was carried out with 4 children, two of them 7 years old. The system showed to be intuitive and simple as expected. After the exercises done in Robi environment, the children were asked to do the same in the Scratch environment. Children said they found it more difficult to find out where to start the program in Scratch and felt the system was less intuitive, due Scratch being less visual and iconic than Robi. At the end of the platform assessment, the children confirmed that Robi was a very intuitive and simple environment to work with.

As future work, we consider that it is necessary to design and to carry out an experiment with a larger number of children and to create and apply questionnaires to evaluate the experiment. Since the experiment carried out was focused on ease of programming, it is important in the future to design formal usability tests. Another important aspect is to integrate Bluetooth in the Robot to make it more convenient and practical to load the code to the Robot (this avoids the cable connection). In addition, other sensors or modules must

be integrated (example: color sensor, sound emitting module, display module, among others), so that the Robot is more complete and has more potential. Currently, Robi only has got the platform to program the Robot. We also consider relevant to develop in the future a support system for managing students and the exercises they solved to monitor their progress and identify their difficulties.

References

- 1 Cristiana Araújo, Lázaro Lima, and Pedro Rangel Henriques. An Ontology based approach to teach Computational Thinking. In Célio Gonçalo Marques, Isabel Pereira, and Diana Pérez, editors, *21st International Symposium on Computers in Education (SIIE)*, pages 1–6. IEEE Xplore, November 2019. doi:10.1109/SIIE48397.2019.8970131.
- 2 Soumela Atmatzidou and Stavros Demetriadis. Advancing students' computational thinking skills through educational robotics: A study on age and gender relevant differences. *Robotics and Autonomous Systems*, 75:661–670, 2016. doi:10.1016/j.robot.2015.10.008.
- 3 Nicholas Alexander Bascou and Muhsin Menekse. Robotics in k-12 formal and informal learning environments: A review of literature. In *2016 ASEE Annual Conference & Exposition*, New Orleans, Louisiana, June 2016. ASEE Conferences. <https://peer.asee.org/26119>. doi:10.18260/p.26119.
- 4 Fabiane Barreto Vavassori Benitti. Exploring the educational potential of robotics in schools: A systematic review. *Computers & Education*, 58(3):978–988, 2012. doi:10.1016/j.compedu.2011.10.006.
- 5 Christina Chalmers. Robotics and computational thinking in primary school. *International Journal of Child-Computer Interaction*, 17:93–100, 2018.
- 6 Guanhua Chen, Ji Shen, Lauren Barth-Cohen, Shiyang Jiang, Xiaoting Huang, and Moataz Eltoukhy. Assessing elementary students' computational thinking in everyday reasoning and robotics programming. *Computers & Education*, 109:162–175, 2017. doi:10.1016/j.compedu.2017.03.001.
- 7 Morgane Chevalier, Christian Giang, Alberto Piatti, and Francesco Mondada. Fostering computational thinking through educational robotics: a model for creative computational problem solving. *International Journal of STEM Education*, 7, August 2020. doi:10.1186/s40594-020-00238-z.
- 8 Tomáš Effenberger and Radek Pelánek. Towards making block-based programming activities adaptive. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, pages 1–4, 2018.
- 9 Francesc Esteve, Jordi Adell, M^a Ángeles Llopis, Gracia Valdeolivas, and Julio Pacheco. The development of computational thinking in student teachers through an intervention with educational robotics. *Journal of Information Technology Education: Innovations in Practice*, 18:139–152, October 2019. doi:10.28945/4442.
- 10 International Society for Technology in Education (ISTE) & Computer Science Teachers Association (CSTA). Operational definition of computational thinking for K-12 education. <http://www.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf>, 2011. Accessed: 2021-12-16.
- 11 P. Freire. *Pedagogia da Autonomia: Saberes necessários à prática educativa*. Paz e Terra, 2011.
- 12 Anaclara Gerosa, Víctor Koleszar, Leonel Gómez-Sena, Gonzalo Tejera, and Alejandra Carboni. Educational robotics and computational thinking development in preschool. In *2019 XIV Latin American Conference on Learning Technologies (LACLO)*, pages 226–230, 2019. doi:10.1109/LACLO49268.2019.00046.
- 13 Carina Soledad González-González. State of the art in the teaching of computational thinking and programming in childhood education. *Education in the Knowledge Society*, 20:1–15, 2019.

- 14 Shuchi Grover and Satabdi Basu. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 267–272, 2017.
- 15 Shuchi Grover, Satabdi Basu, Marie Bienkowski, Michael Eagle, Nicholas Diana, and John Stamper. A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments. *ACM Transactions on Computing Education (TOCE)*, 17(3):1–25, 2017.
- 16 Ting-Chia Hsu, Shao-Chen Chang, and Yu-Ting Hung. How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education*, 126:296–310, 2018.
- 17 Luiz A Junior, Osvaldo T Neto, Marli F Hernandez, Paulo S Martins, Leonardo L Roger, and Fatima A Guerra. A low-cost and simple arduino-based educational robotics kit. *Cyber Journals: Multidisciplinary Journals in Science and Technology, Journal of Selected Areas in Robotics and Control (JSRC), December edition*, 3(12):1–7, 2013.
- 18 Eija Karna-Lin, Kaisa Pihlainen-Bednarik, Erkki Sutinen, and Marjo Virnes. Can robots teach? preliminary results on educational robotics in special education. In *Sixth IEEE International Conference on Advanced Learning Technologies (ICALT'06)*, pages 319–321. IEEE, 2006.
- 19 Leo Louis. working principle of arduino and u sing it. *International Journal of Control, Automation, Communication and Systems (IJACCS)*, 1(2):21–29, 2016.
- 20 J. Piaget, M. Piercy, and D.E. Berlyne. *The Psychology of Intelligence*. Routledge classics. Routledge, 2001.
- 21 MERT Arduino & Tech. How to make Arduino Obstacle Avoiding Robot Car | Under \$20. https://www.youtube.com/watch?v=4CF00MiS1M8&ab_channel=MERTArduino%26Tech, 2018. Accessed: 2021-12-16.
- 22 Salete Teixeira, Diana Barbosa, Cristiana Araújo, and Pedro Rangel Henriques. Improving Game-Based Learning Experience Through Game Appropriation. In Ricardo Queirós, Filipe Portela, Mário Pinto, and Alberto Simões, editors, *First International Computer Programming Education Conference (ICPEC 2020)*, volume 81 of *OpenAccess Series in Informatics (OASISs)*, pages 27:1–27:10, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASISs.ICPEC.2020.27.
- 23 L.S. Vygotsky, E. Hanfmann, G. Vakar, and A. Kozulin. *Thought and Language*. The MIT Press. MIT Press, 2012.
- 24 David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–25, 2017.
- 25 Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, March 2006. doi:10.1145/1118178.1118215.

Down-Translating XML: The Python Way

Alberto Simões  

2Ai – School of Technology, IPCA, Barcelos, Portugal

José João Almeida  

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Abstract

Nowadays, the most used approach to process an XML file is based on the processing of a DOM structure and a set of operations that collects or edits information in the model using some kind of selectors (usually CSS-like or XPath).

Nevertheless, the process of performing a depth-first walk through the DOM, and synthesizing values, is a simple way to traverse and transform an entire XML document.

In this document we discuss the details on the implementation and usage of a Python package for XML document processing based on this structure. Given the existence of similar tools for other programming languages, we will mainly focus on the used approach, that takes advantage of the Python style guides and development patterns.

2012 ACM Subject Classification Software and its engineering → Extensible Markup Language (XML); Software and its engineering → Scripting languages

Keywords and phrases XML, Python, Depth-First Processing

Digital Object Identifier 10.4230/OASICS.SLATE.2022.15

Supplementary Material *Software (Python Package)*: <https://pypi.org/project/xmltd/>
Software (Source Code): <https://natura.di.uminho.pt/svn/main/python/XML-DT/>

Funding *Alberto Simões*: This project was funded by Portuguese national funds (PIDDAC), through the FCT – Fundação para a Ciência e Tecnologia and FCT/MCTES under the scope of the project UIDB/05549/2020.

José João Almeida: This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

The Down-Translation is the name used in other libraries, for Perl [1] and JavaScript [7], to refer to the approach of processing an XML Document Object Model (DOM) tree doing a depth-first traversal, and synthesizing values.

The idea is simple: define a mashup of a dispatch table and visitor patterns [3]. The programmer can define a function to process each type of node in the XML file. This function returns (or synthesizes) the value of processing its children. This returned value is used by its parent, and the values are synthesized up to the root of the document object model tree.

While the approach is known, in this article we describe the implementation of such a processor for the Python programming language. This implementation differs from the other two mentioned above as it does not use an explicit dispatch table, but uses Python's own symbol table for that purpose. This makes it easy to define XML processors as simple classes, that can even inherit from other XML processors, allowing more specific approaches to take advantage of other, more general, processors.

The article is structured as follows: Section 2 motivates for the use of Down-Translation approaches for XML Processing. Section 3 discusses the implementation decisions and presents some details on how Python features were used to provide an interesting programming interface. Section 4 shows some examples of DT Processors. The article concludes with an analysis of the obtained tool and future steps.



© Alberto Simões and José João Almeida;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 15; pp. 15:1–15:9

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 DOM vs DT processing

Currently, most libraries for processing XML documents focus on the construction of the DOM and allowing the programmer to search in the tree for specific structures. These structures are usually encoded using the Cascading Style Sheets (CSS) [4] syntax for element selectors, or by using the XPath [6] language path specification.

The programmer instructs the XML library to find the set of elements in the desired positions, and to extract the required information. Most of the time, these elements are not processed directly, but the children are visited following some other path, in order to find the relevant information. This process is repeated until all the desired information is obtained.

Whenever a selector or a XPath path is used, the complete DOM tree needs to be processed. Thus, every time a new query is performed, the library requires to process the tree and find the relevant elements.

With a Depth-First approach, the DOM tree is still required, but the goal is to traverse it only once, from its leaves to its root. During this process the programmer is able to construct whatever he needs as an answer. It can be the creation of other XML document, and therefore, returning a string or XML objects in each visitor, or to return a specific data structure that represents the desired information. As a third option, each visitor can store the desired information in any data structure, as side effects.

3 Python Implementation

In this section we will describe the details on the `XmlDt` module implementation. First we will describe how the dispatch table of visitors is implemented, as well as the visitors for special elements, like comments or textual data. Then follows the list of special methods that the programmer can use on each visitor. Finally, a discussion on the support of hooks and the chaining of visitors.

3.1 XML Element Dispatch Table

The Perl and JavaScript versions of the Down-Translation approach are based on dispatch tables. These are implemented using dictionaries (hash tables or objects, to use the terminology for each language). The dictionaries store a relation between each XML element names and an anonymous functions (or lambda expression) that is responsible for processing those elements. While this is a compact method to define such processors in these two languages, for Python it is not so clean, as Python lambda expressions is limited to a single statement, as the typical way of using lambda expressions in Python that require more than one statement is to implement a named function that is, then, called by the lambda expression. This approach would require the programmer to implement a named function for each element being processed.

Taking this into consideration, the module was implemented using Python's reflection capabilities. The `XmlDt` module offers a super-class that the user should derive to create each processor. The methods on this subclass are automatically assigned for processing the XML elements with the same name. This is performed during run-time, when `XmlDt` looks-up each XML element name in the class symbol table.

The programmer's work is just to define a subclass for `XmlDt`, and implement methods for each XML element needs processing. These methods are visitors of the XML element nodes. They receive the processor object (as they are methods) as well as the element they are visiting.

■ **Listing 1** Simple XmlDt to replace `` and `<i>` elements by `<emph>` elements.

```
class ReplaceBoldItalic (XmlDt):
    def i(self, element):
        element.tag = "emph"
        return element.xml

    def b(self, element):
        element.tag = "emph"
        return element.xml

processor = ReplaceBoldItalic()
print(processor(filename="something.xml"))
```

While interesting, this approach has two main drawbacks, that needed to be considered:

- The super-class should not pollute the symbol table, as any name used for a method might be used in an XML element. To minimize this problem, all methods inside the XmlDt package are prefixed with an underscore. While those are valid names for elements on XML files, they are not common.

Note that Python has two distinct use of the underscore in method named. When using only one underscore in the beginning of the method name, it is used to denote that the method is private (while this is not enforced, it is the common *pythonic* way of describing that property). There is also the possibility of prefixing the methods with two underscores, and add two underscores at the end of the method name. These methods are special methods that Python defines, and allow the implementation of special functionalities, like the definition of a constructor or an index-based accessing function.

XmlDt uses these two types of methods. The first for internal methods, that are not expected to be sub-classed. The second, for internal methods that include basic implementation/behavior, but that can be extended by the programmer (thus, following the same *pythonic* approach for other similar situations).

- From the W3C standard, the valid characters for XML element names are all valid as method names, except the period and the dash/hyphen. To address this problem, a Python decorator¹ was implemented. Python decorators are special methods invoked at compile time. They decor or annotate methods, and allow to add extra information to the method. This decorator allows the programmer to implement the visitor with any valid name, and assign it to another tag name. It can be used for any tag, but it mostly useful when tag names include any of these two characters.

A simple example of a XmlDt processor, that processes an XML document and replaces all `` and `<i>` element names into `<emph>` elements is presented in Listing 1

When requiring to process an element whose name is not a valid Python method name, the decorator mentioned before should be used. The decorator is named `dt_tag` and must precede a method. This decoration specifies the XML element name that will be processed by the visitor described. Listing 2 shows a simple example of how this decoration can be used, to process the `<TEI.2>` root element.

¹ <https://peps.python.org/pep-0318/>

■ **Listing 2** Method decorated with the name of the XML element the visitor should process.

```
class processTEI (XmlDt):
    @XmlDt.dt_tag("TEI.2")
    def process_tei_two(self, element):
        # do something
        return element.xml
```

3.2 Special element handlers

There are some other handlers that are useful, but that are not related directly to XML elements, like the processing of text content of elements, processing comments, or even, the possibility to define a default visitor (that will be called whenever a visitor is not found for a specific XML element).

These specific methods have default behaviors defined in the `XmlDt` module, just like other double underscore methods that are available builtin in Python. That is the reason we chose this syntax for visitors that have a default behavior, but that can be overridden by the programmer.

PCData the `XmlDt` module implements the method `__pcdata__` that processes text nodes in the DOM tree. While there is a default visitor, that returns the text with no change, the programmer can implement its own visitor. This visitor receives the processor object and the text, as a string.

CData by default `XmlDt` does not distinguish between PCData and CData elements.² Nevertheless, it is possible to configure `XmlDt` to distinguish them. In this case, the `__cdata__` visitor is called whenever such an element is found.

Comments it is not common to process an XML document and take actions accordingly with its comments, but given that the underlying library has access to them, `XmlDt` allows the programmer to perform actions on such elements, overriding the `__comment__` method. By default, this method has no action.

Default when processing large XML documents, it is common that a specific set of elements have specific visitors that need to be hand tailored, but for the majority of the elements, their visitors are always the same. With this in mind, the `__default__` method can be overridden to define the default behavior for elements that do not have a visitor.

As an example, consider a processing tool that processes an XML document replacing each word by its stem³. Listing 3 shows how this can be performed Using the NLTK library [2] and `XmlDt` using two special methods, one to process text contents, and the other to construct back XML elements from each node in the original tree.

3.3 XmlDt methods

There are some specific functionalities that can be useful for the programmer. While one of the goals was not to pollute the name space for the user module, we still decided to define standard methods, making them start with the `dt_` prefix. If by some reason these names are required as element names, the user can still use the `@dt_tag` decorator referred previously.

² In this sense, follows the same approach of the `lxml` (<https://pypi.org/project/lxml/>) library, that is the module used by `XmlDt` to construct the DOM tree.

³ A stem is a kind of root for similar words [5].

■ **Listing 3** Stemming the contents of an XML file.

```

from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
from xmldt import XmlDt

ps = PorterStemmer()

class XmlStemmer(XmlDt):
    def __pcdata__(dt, txt):
        words = word_tokenize(txt)
        return " ".join([ps.stem(w) for w in words])

    def __default__(dt, el):
        return el.xml

```

It might be important to stress here why we have two distinct naming conventions. These are methods offered by the library, that are not supposed to be overridden by the programmer. The double underscore methods are offered as default methods that can be overwritten.

The most relevant methods and getters are:

dt_in_context This method is used to check the current visitor's path, allowing the user to define custom behaviors accordingly with the element context. It receives one or more element names and return a boolean value if the current node's path include those elements.

dt_parent This is an accessor to the current parent's element, allowing the addition of further information in that element. There is a similar method to access to grand-parent element: **dt_gparent**.

dt_root Similar to the accessor shown above, this allows the direct access to the document root element.

dt_path Returns the current path of elements (does not include the element itself). This is not a list of element names, but a list of the elements themselves.

3.4 Hooks and Chaining

To allow sub-classes to expand the way methods are searched in the module symbol table, `XmlDt` uses the concept of a hook list, where the programmer can include new code to decide which method to call for processing a specific element. This is useful, for instance, in the `HtmlDt` class (described in section 4.2), where this feature is used to allow the programmer to define methods to handle specific element classes (the `class` attribute from HTML).

While this feature is quite useful, it raises a problem: what visitor to call when an element has more than one class, or if there is a visitor function for the specific class and for the element name itself. To allow multiple visitors to be called, there is a special method, `dt_chain(el)`, that, whenever is called with the current element, makes `XmlDt` to call the next available visitor.

The order of processing elements is as follows: first all hook methods are checked. After hooks, the main processor symbol table is searched. Finally, the default processor is used. When more than one hook method is registered, the used order is relevant to understand how processors are searched.

■ **Listing 4** Implementation of a XML grep command line tool.

```

from xmldt import XmlDt
import sys
import re

if len(sys.argv) != 3:
    print("Usage: dtgrep pattern filename")
    sys.exit(1)

_, pattern, filename = sys.argv

class Grep(XmlDt):
    def __pcdata__(self, text):
        if re.search(pattern, text):
            xpath = "/".join([e.tag for e in self.dt_path])
            print(f"{xpath}: {text}")
        return text

grep = Grep()
grep(filename=filename)

```

4 Using XmlDt

In this section we present three different usages of XmlDt:

1. A simple **grep** command line tool, that searches for a pattern in elements contents, and present the user with information about the hit;
2. A subclass of XmlDt, named **HtmlDt**, for processing HTML files, and using some functionalities especially useful for this kind of documents;
3. A tool to help the process of creating a XmlDt processor based on sample XML files.

4.1 XML Grep

This is a straightforward and typical use of XmlDt. The user specified a pattern and a XML file, and the elements which content match the supplied pattern are presented as output.

The program presented in Listing 4 includes the relevant parts of dealing with XmlDt. The details on processing command line arguments were simplified: the first argument for the command line tool (thus, `argv[1]`, is the regular expression, and the second is the file to process.

This solution uses the `__pcdata__` handler to perform the search and, whenever a match is found, to print the path to that element as well as the element contents.

4.2 Processing HTML

One of the benefits of implementing the tool as a class is the possibility to create sub-classes that inherit the default behavior of the main tool and add new features.

This allowed the creation of the **HtmlDt** module, that inherits the default functionalities of XmlDt and adds some specific tools to easy the parsing of HTML documents.

The implementation of `HtmlDt` defines:

- the constructor for the parser including the attribute required by `lxml` to process HTML files;
- a decorator, named `html_class`, that allows the programmer to specify a function that will process HTML elements that belong to a specific HTML *class*;

While the first item has a very simple implementation, just overriding the default constructors, the second requires some Python *reflection*, that is, the ability to query and change details on the program while it is executing. In this case, a hook is added to `XmlDt` that searches in the Python symbol table for methods decorated with the `html_class` property. The hook runs before any other method from the dispatch table, and checks if the *class* attribute is present in the element being currently processed. Whenever an element with a known class is found that specific processor is called.

Listing 5 shows an example of using the `html_class` decorator, to specify that, whenever an element is found with the *bold* class, the element should be changed to the `` element.

■ **Listing 5** Example of usage of the `HtmlDt` module.

```
class AddBold (HtmlDt):

    @HtmlDt.html_class("bold")
    def set_bold(self, e):
        e.tag = "b"
        return e.xml
```

While the `HtmlDt` class has less than 40 lines of code, the details of handling the decorators is not simple, and are outside of the scope of this article. The most interesting part is the hook that processes classes, and is shown in Listing 6. This hook is registered in the constructor of the `HtmlDt` module. The hook, itself, receives an element and looks for the *class* attribute. Existing, its value is replaced by a list of all the used classes (splitting the class list by the space character, as defined by the HTML standard). Then each class is visited, once at a time, following the order of use. If there is a specific method for that class, then it is added to a list, that is returned at the end of the hook. This list is, at last, the sequence of functions to be called by the `XmlDt` class.

■ **Listing 6** Handler for class-based hooks.

```
def _class_hook(self, element):
    handlers = []
    if "class" in element:
        element["class"] = re.split(r'\s+', element["class"])
        for c in element["class"]:
            if c in self._classes:
                handlers.append(self._classes[c])
    return handlers
```

4.3 Base Processor Generator

The `XmlDt` package distribution includes a small utility that allows to create bare processors based on specific XML instances. The tool receives a list of XML files, processes them, and generates a Python script with the dispatch table for all the required visitors.

15:8 Down-Translating XML: The Python Way

The tool has a diverse set of options, that will not be discussed here. Nevertheless, its basic structure is quite simple. Given the ability to describe processors based on generic XML objects (namely the `__default__` handler), it is possible to gather information about every tag element present on the supplied XML files, as well as what attributes are used for each of these elements. Listing 7 shows a simplified version of the default element handler.

■ **Listing 7** Simplified `__default__` handler for processor generation.

```
def __default__(self, element):

    # register tag
    if element.tag not in self.data:
        self.data[element.tag] = {"count": 0}

    # count number of occurrences for this tag
    self.data[element.tag]["count"] += 1

    # save information about existing attributes
    for key in element.attrs.keys():
        if key not in self.data[element.tag]:
            self.data[element.tag][key] = 0

    # count attribute usage
    self.data[e.tag][key] += 1
```

After processing the set of sample files, and storing all the information in the `data` property, the tool uses a text template to generate the processor. It also includes details on the number of occurrences of each element and attribute in the XML files as Python comments. This information can be useful for the programmer to structure the processor.

5 Conclusion

In this short paper we describe a specific Python tool for processing XML documents. It is based on two distinct patterns from software development: the dispatch table and the visitor patterns.

The usage of these patterns with the Down-Translation approach, and the power of the Python language, namely in terms of reflection, allows the creation of elegant XML processors.

The tool is available both at PyPi⁴ and is open source⁵.

While the current status of the tool makes it usable and useful, a set of improvements are in place for development:

- Add support to process URL directly;
- Add support to handlers based on the element identifier and in the element language, taking advantage of the `@id` and `@xml:lang` attributes;
- Allow proper debug of the order handlers are called when more than one can be used for a specific element (for instance, an element might be processed by a handler based on tag name, or HTML class, or identifier, or language);
- Add support for CSS selectors for handlers.

⁴ <https://pypi.org/project/xmltd/>

⁵ <https://natura.di.uminho.pt/svn/main/python/XML-DT/>

References

- 1 José João Almeida and José Carlos Ramalho. XML::DT a perl down-translation module. In *XML-Europe'99, Granada - Espanha*, May 1999.
- 2 Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.
- 3 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. URL: <https://books.google.pt/books?id=6oHuKQe3TjQC>.
- 4 Ian Hickson, John Williams, Daniel Glazman, Peter Linss, Erika Etamad, and Tantek Çelik. Selectors level 3. W3C recommendation, W3C, November 2018. URL: <https://www.w3.org/TR/2018/REC-selectors-3-20181106/>.
- 5 Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.
- 6 Jonathan Robie, Josh Spiegel, and Michael Dyck. XML path language (XPath) 3.1. W3C recommendation, W3C, March 2017. URL: <https://www.w3.org/TR/2017/REC-xpath-31-20170321/>.
- 7 Alberto Simões. XML Parsing in JavaScript. In Ricardo Queirós, Mário Pinto, Alberto Simões, José Paulo Leal, and Maria João Varanda, editors, *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*, volume 56 of *OpenAccess Series in Informatics (OASISs)*, pages 9:1–9:10, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASISs.SLATE.2017.9.

Determining Programming Languages Complexity and Its Impact on Processing

Gonçalo Rodrigues Pinto ✉

Department of Informatics, University of Minho, Braga, Portugal

Pedro Rangel Henriques ✉ 

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Daniela da Cruz ✉ 

Checkmarx, Braga, Portugal

João Cruz ✉

Checkmarx, Braga, Portugal

Abstract

Tools for Programming Languages processing, like Static Analysers (for instance, a Static Application Security Testing (SAST) tool), must be adapted to cope with a different input when the source programming language changes. Complexity of the programming language is one of the key factors that deeply impact the time of giving support to it.

This paper aims at proposing an approach for assessing language complexity, measuring, at a first stage, the complexity of its underlying context-free grammar (CFG). From the analysis of concrete case studies, factors have been identified that make the support process more time-consuming, in particular in the stages of language recognition and in the transformation to an abstract syntax tree (AST). In this sense, at a second stage, a set of language characteristics is analysed in order to take into account the referred factors that also impact on the language processing.

The principal goal of the project here reported is to help development teams to improve the estimation of time and effort needed to cope with a new programming language. In the paper a tool is proposed, and its prototype is presented, that allows the evaluation of the complexity of a language based on a set of metrics to classify the complexity of its grammar, along with a set of properties. The tool compares the new language complexity so far determined with previously supported languages, to predict the effort to process the new language.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Complexity, Grammar, Language-based-Tool, Programming Language, Static code analysis

Digital Object Identifier 10.4230/OASICS.SLATE.2022.16

Supplementary Material *Software (Web Application)*: <https://lce.di.uminho.pt/>
archived at `swb:1:dir:ec41f17cb7b247b4615a92cf8fc37b82b3fc972c`

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

Acknowledgements We want to thank the reviewers for the input and suggestions on the paper.

1 Introduction

A SAST tool analyses source code written in a programming language and finds its security vulnerabilities. While this solution satisfies the need (detecting software vulnerabilities), there are other factors that need special attention in this type of tool, one of which is the maintenance required.



© Gonçalo Rodrigues Pinto, Pedro Rangel Henriques, Daniela da Cruz, and João Cruz;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 16; pp. 16:1–16:15

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

16:2 Determining PL Complexity

Several new practices have emerged in recent years that can improve software maintenance. The major consideration is how to balance the enormous complexity of software with its cost, effort, and time required for maintenance. For that, it must be adapted to handle different inputs when the source programming language varies.

To do this, one of the first steps towards supporting a new programming language in this tool is to create a new parser to analyse the relevant language.

The complexity of the programming language is one of the key factors that affects the time to provide support for it. This limitation raises the need to evaluate whether the complexity of a programming language is related to the complexity of its context-free grammar.

Thus, given the difficulties associated with the SAST engine in analysing and supporting a new programming language, it is motivating to create a tool that selects and implements a set of metrics and analyses a set of properties that allow us to assess the complexity of a language.

The primary purpose of the study described here is to assist language support teams in better estimating the time and effort required to support a new programming language. Along the paper, we propose and present a tool for evaluating the difficulty of supporting a language based on a collection of metrics to classify the complexity of its grammar, as well as a set of properties. To forecast the work required to process the new language, the program compares the new language difficulty so far identified with previously supported languages.

This Section 1 discussed the significance of maintenance, what a SAST tool is and its limits, how complexity is to be measured, and why the provided tool was developed. In Section 2, it is intended to focus on the main points to characterize the concepts of software, language, and grammar in determining the complexity of programming languages and their impact on processing. After the concepts have been introduced, Section 3 follows, in which the DSL created for this purpose is presented in order to represent the extra-grammatical characteristics that have to be described by those who know the language. Introduced and described the language intended for this particular problem domain, it is fundamental to talk about the proposal to be developed, showing its architecture and the results already obtained to produce a quantitative and qualitative report of the language, this information is described in the Section 4. Finally, Section 5 is the summary of the document, some conclusions and results achieved, and a description of future work.

2 Software, Grammar, Language Complexity and the impact on processing

Section 2 begins by introducing the concept of software complexity and its impact on the timing of support. After that, one of the tools that allows to evaluate the complexity of a language and grammars, is presented, explaining its relation with languages and how grammatical complexity is defined. Afterwards, the way to measure this grammatical complexity, by metrics, is presented. Finally, the subject of this project, complexity of programming languages, is introduced.

2.1 Software Complexity

Knowledge about the properties of entities is obtained through measurement. In order to relate and compare properties between entities, rules are used. Nevertheless, measurement is not something clear or easy to define, because it is always open to subjective interpretation. Every time we effectively measure something that was not measurable at first glance, we expand the power of software engineering, as is done in other disciplines in this area.

There is no theory that shows whether a set of metrics is valid. We only know that there is a structure based on objectives for software measurement, which can improve software engineering practices.

This structure is based on three principles: categorizing the entities to be investigated, determining relevant measuring targets, and determining the maturity level attained.

In recent years, software complexity has been the subject of much interest in order to define measures for measuring it. Complexity is the characteristic associated with a system or model whose state is composed of many parts and is difficult to understand or find an answer for. Understanding and measuring the software complexity is not something simple and obvious.

However, measuring the complexity of the problem associated with this software is useful, as it may prevent the effort or resources needed for the project. By comparing the problems and considering the solutions found for the problems already solved, it is possible to predict the properties of the new solution to the latest problem, such as cost or time.

Size along with structure are the main internal properties in measuring software complexity, according to Fenton and Pfleeger in 1998 [6].

- **Size Complexity** – the traditional attribute to measure in software, because it is advantageous, accessible to measure without having to run the system, and because software development is a physical entity.
- **Structure Complexity** – determines the level of project productivity, as it has been proven that a larger module does not always take longer to specify, design, code, and test than a small one. The structure of the product affects its maintenance and development effort.

Therefore, complexity can be assessed by quantifying a subset of software metrics that are based on static analysis. In this way, we can better understand the language in some aspects, such as the size and structure.

2.2 Grammar Complexity

Since any grammar characterizes a language and gave a premise for determining elements of that language, a grammar might be considered as both a program and a specification. Grammars formally specify languages, so the complexity of languages depends on the complexity of grammars, even if the complexity of grammars does not fully imply the complexity of language analysis.

In this context, the use of grammars is proposed to define the languages and support their recognition, which leads to a strong relationship between grammar and the language that is defined by that grammar [7]. Therefore, grammar will be one tool to assess the complexity of a language.

Considering what has been previously presented to show the relationship between grammars and languages, supporting a new programming language in a static analysis tool is faster and requires less effort, the less complex the grammar is.

The complexity of a grammar as a characterizer and producer of a language that directs the recognition of sentences in that language concerns how the symbols depend on each other, i.e., the number of symbols on the right-hand side of a production for a given symbol on the left-hand side, or how many symbols that symbol intervenes in.

Considering this, the need to evaluate the complexity of a grammar arises, since it will allow us to evaluate the complexity of the language defined by it. Thus, the use of grammatical metrics is relevant to the study in question.

2.2.1 Measuring Grammar Complexity

The metrics for evaluating the complexity of a well-formed context-free grammar are presented, dividing them into the previously mentioned criteria:

- Size metrics that measure the number of symbols (terminals or non-terminals) and productions used to write the grammar. As the grammar is the basis to recognize the sentences of the language defined by itself, it is reasonable to state that the size of the grammar has a direct impact on the time and effort necessary to support that language

Size Metrics

■ **Table 1** Metrics for evaluating the Size of Context-Free Grammars.

Metric	Definition
#P	Number of productions
#N	Number of non-terminals
#T	Number of terminals
#UP	Number of unit productions
RHS-Max	Maximum number of symbols on an RHS
RHS	Average number of symbols in the RHS
ALT	For the same left sides, average size of alternative productions
MCC	McCabe cyclomatic complexity

- Structure metrics that measure the dependency among the symbols of a grammar induced by its productions. Once again, we can state that the more intricate are the interrelations among the symbols, the harder it is to support the grammar and to recognize the sentences of the generated language. To compute those metrics, a grammar is represented as a graph.

Structure Metrics

■ **Table 2** Metrics for evaluating the Structure of Context-Free Grammars.

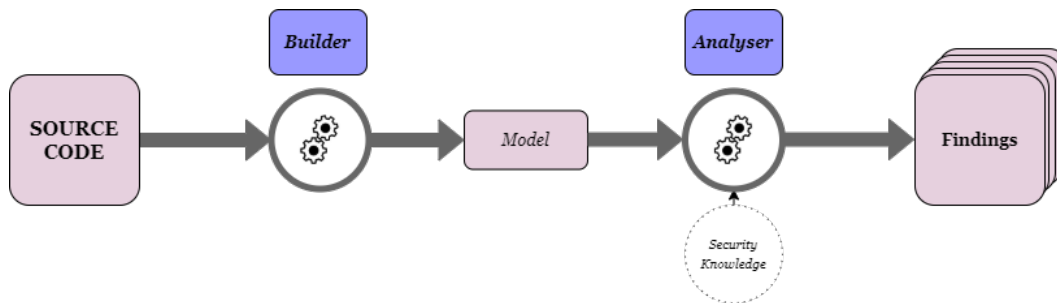
Metric	Definition
#R	Number of recursive symbols
FanIn	Average number of branches of the input nodes (non-terminals) of the DGS
FanOut	Average number of branches of the output nodes of the DGS
TIMP	Tree Impurity
CLEV	Normalized Counts of Levels
NSLEV	Number of Non-Singleton Levels
DEP	Size of The Largest Level

2.3 Language Complexity

Software security is turning into an inexorably significant differentiator for IT organizations. Therefore, methods for forestalling software vulnerabilities during software development are turning out to have increasing significance. The longer it takes to find the vulnerabilities, the more costly it will be to fix, and making an already difficult situation even worse.

In order to identify existing vulnerabilities, Static Application Security Testing, abbreviated as SAST and often alluded to as “White-Box Testing”, is used. The tool performs a security test that examines the source code of applications.

The idea behind this type of analysis is to identify in the code the use of language constructions that are vulnerable and can facilitate external attacks on the SW system. Static analysis examines the text of a program statically, without running it. In Figure 1, a high-level description of the scanning pipeline that allows for the aforementioned static analysis of the source code will be presented.



■ **Figure 1** An abstract, high-level model of a SAST tool.

In a comprehensive way, a SAST tool basically comprises two processes.

1. **Model Builder** by ingesting the source code and transforming it into a normalized, understandable model for the analyser to decipher. A typical model is the development of an AST, a simplified representation of the structure of the source code, where each node in the tree is associated with a constructor of the code. In this way, the syntax is abstract since it does not represent every detail, as it appears in real syntax.
2. **Analyser based on existing security knowledge**, using a series of rules to figure out what the tool should evaluate within the source code. It is critical to customize and calibrate these rules to suit a specific application, as this allows for more reliable and worthwhile results. The feasibility of these methods is the main inner component of a SAST tool, from a customer's point of view.

The analyser based on current security knowledge process is irrelevant to the goal project since the challenge of supporting a new programming language resides in the model's builder process, because the language in which the source code is produced influences the static analysis process.

Taking this into consideration, the complexity of a language is defined by the complexity of the grammatical structure and the semantic properties, these two factors thus highlight the time and effort needed for support.

In this sense, the complexity of supporting a new language in this case study is strongly related to the AST generated [8]. For this reason, for the support to be more productive, at the level of less time and effort, the programming language in which the code under analysis is written, should generate an AST with the following characteristics:

- **Dense:** No pointless nodes;
- **Convenient:** Patterns in the tree are straightforward and quick to discern;
- **Significant:** Emphasize operators, operands, and the connection between them.

The initial two focuses infer that it ought to be simple and quick to identify patterns in the tree. In order to gather as much useful information as possible about each node, it is usually necessary to make multiple passes over the tree.

The last point infers that the tree structure ought to be insensitive to changes in the grammar.

16:6 Determining PL Complexity

Besides this strong relationship, there is another aspect that the complexity of the programming language influences, since when the tree is generated there are properties of the nodes that have not been assessed so far and are necessary in order to detect as many vulnerabilities as possible.

In general, it is intended that each node of the AST contains the following properties:

- **Name:** Symbols are identifiers like x and y , yet they can be operators as well, as for example, the symbol known as the addition operator (+);
- **Category:** The thing like the symbol is. Is it a class, method, variable, name, etc. To approve a method called $x + y$, for instance, we want to realize that $+$ is a method to add 2 or more values, not a variable or class.
- **Type:** When the symbol is a variable, there is an interest in knowing what type it concerns, in order to be able to subsequently approve certain operations. Normally, the software engineer needs to explicitly distinguish between each type of symbol (in some languages, the compiler assumes this).
- **Scope:** The scope of a symbol restricting is the part of a program where the symbol is valid, that is, the place where the symbol can be utilized to refer to the element.

The cost of supporting a programming language in a SAST tool is a function of many of its properties. The following is one of the main characteristics¹ present in a real programming language that influences the complexity of support in this type of application.

Declaration in a programming language is a statement that determines the properties of a symbol. A declaration introduces a program Entity identified by a unique name (an identifier), its category (function, variable, constant, etc.), its type (in case it is a variable or constant) or if it is a function what it accepts as input and output, it can also declare things like the size of a type.

► **Example 1.** Consider for example the same code written in different programming languages, Python, Java, JavaScript, respectively, to declare the same variables and the same function.

```
def main():
    x = 13
    y = "Python!"
```

```
public class Main {
    public static void main(String[] args) {
        int x = 13;
        String y = "Java!";
    }
}
```

```
function main() {
    let x = 13;
    var y = "JavaScript!"
}
```

A declaration conveys the “meaning” of a symbol, which highlights that this property is related to semantics and not syntax. However, there is interest in analysing this because of the extra effort needed in the Builder process.

¹ In order to make the document as compact as possible no further features will be described, but the reader is warned that there are other relevant properties.

The fact that a declaration is used to communicate the presence of an entity to the compiler means that, in dynamically typed languages such as Python, it is unnecessary to specify the variables; the runtime interpreter does the verification work. Considering the present case study, as shown, a static analysis tool does not execute any code. For that reason, in order to detect as many vulnerabilities as possible, it needs to do the work of the compiler.

In comparison, for example, to the extreme that a language that is strictly typed like Java, C# or C++ explicitly defines the data type when creating a symbol, which makes vulnerability detection easier than most things will be known, allowing reasoning and analysis with confidence.

There is also the case of the JavaScript language, where the variable declaration is dynamic but allows you to write using explicit types, getting the advantages of a strongly typed language. This makes the static analysis of vulnerabilities easier than dynamically modifying them.

3 A DSL to describe the properties of a Programming Language

As mentioned and demonstrated, there are aspects of programming languages that grammar metrics cannot measure or capture, but which have a huge impact on the support, particularly when it comes to a static analysis tool.

In this sense, a domain-specific language (DSL) was created with the goal of describing the features of a programming language that bring positive or negative impact to the moment of support. The Properties Language is used in programming languages to define the substance of linguistic features and paradigms.

This language was not created with the intention of acting as documentation for a specific language, but it does include in its design various characteristics and notions that allow for an assessment of its complexity.

The formal definition of the DSL to describe the Properties of a Language is shown in Listing 1.

■ **Listing 1** Formal Definition of the Properties Language.

```
language -> header properties
header -> name (version)?
name -> ID
version -> VERSION
properties -> paradigms
    (SEMICOLON propertyDeclaration)?
    (SEMICOLON propertyIndentation)?
    (SEMICOLON propertyMemory)?
    (SEMICOLON propertySemiColon)?
    (SEMICOLON feature (SEMICOLON feature)*)?
    DOT
paradigms -> PARADIGM COLON paradigm (COMMA paradigm)*
paradigm -> nameParadigm (weight)?
weight -> DOLLAR NUM
propertyDeclaration -> DECLARATION COLON typeDeclaration (weight)?
typeDeclaration -> STATIC | DYNAMIC | BOTH
propertyIndentation -> INDENTATION COLON useIndentation (weight)?
useIndentation -> YES | NO
propertyMemory -> MEMORY COLON typeMemory (weight)?
typeMemory -> MANUAL | AUTOMATIC
```

16:8 Determining PL Complexity

```
propertySemiColon -> STRSEMICOLON COLON isMandatory (weight)?
isMandatory -> YES | NO | OPTIONAL
feature -> featureName COLON LPAR observations RPAR (weight)?
featureName -> STR
observations -> observation (COMMA observation)*
observation -> STR

NUM -> '-'? [0-9]+
STR -> '"' ( ~('"' ))* '"'
VERSION -> [0-9]+ (('.' [0-9]+) )?
ID -> [a-zA-Z]+
```

The language concepts presented are quite explicit, in that the terminology used literally means that name. The phrases in this language are precisely the features of a programming language. Briefly, a reference about the name of the paradigms in which it can be any within a short list created and selected by the author.

Regarding the content of each list is one or more items within a paradigm, either a predefined feature or a free one that the user wants to describe. For each of these items, an associated weight (positive or negative) is optionally defined, thus translating the possibility of translating the impact of this element on the processing complexity.

► **Example 2.** *The following is an example of the description of the properties of the Python programming language, according to the DSL presented above.*

```
Python 3.10.4

PARADIGMS : object-oriented $3, procedural, functional,
           structured, reflective;

DECLARATION : dynamic ;

INDENTATION : yes $-1 ;

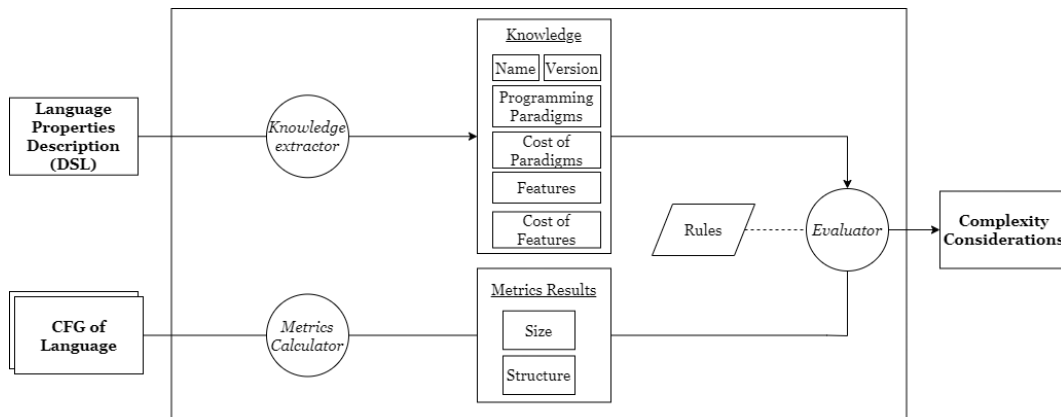
MEMORY : automatic $1 ;

SEMI-COLON : no ;

"Anonymous functions" : [
    "Implemented using lambda expressions;
     however, there may be only one expression in each body"
];
"String interpolation": [
    "The process of evaluating a string literal containing 1..*
     placeholders, yielding a result in which the placeholders are
     replaced with their corresponding values."
] $20;
"Triple-quoted": [
    "Beginning and ending with three single or double quote marks.",
    "May span multiple lines and function like documents in shells"
]
.
```


4 A Tool to determine Programming Language Complexity

Although there are some similar tools, such as the SynQ tool [9, 10], the SdfMetz, SdfCoverage [1, 2] and the gMetrics tool [3, 4] or even the GQE tool [5], that provide information about the complexity and quality of a grammar, this tool differs in its ultimate goal, to measure the complexity of a language rather than the grammar. However, these tools already demonstrate that grammar metrics are important for understanding the complexity of a grammar, therefore their implementation. Furthermore, this tool brings innovation, as it allows for an evaluation of linguistic properties through a new DSL.



■ **Figure 2** Language Complexity Evaluator, architecture.

The diagram of the grammatical representation of the tool under development is depicted in Figure 2. The tool, Language Complexity Evaluator (LCE), is an application that reads any ANTLR grammar (that is, any grammar written in the ANTLR metalanguage) and any text written in the metalanguage built by the author, generating values for each of the metrics under consideration and extracting knowledge from the properties provided. The user has the ability to analyse these values, as well as, observe a series of automatically generated considerations from a series of predefined rules. In this way, the user can easily predict whether the language is of low or high complexity.

Currently, the LCE tool is available as an open source project at <https://lce.di.uminho.pt/> and supports a variety of operations, the most notable of which are:

- Recognize a grammar in the ANTLR format and the description of features in the previously presented DSL format;
- Compute the size and structure metrics previously presented;
- Extract the knowledge from the features described using the DSL and their associated weights;
- Draw conclusions from the results obtained in the two previous phases.

4.1 Main Results

This subsection seeks to demonstrate the real application (using as a case study a well-known programming language) of the Language Complexity Evaluator tool, described above, in its present state publicly available.

The purpose is to demonstrate the advantages of utilizing this tool to estimate the time and effort needed to handle a new programming language in an SAST tool.

16:10 Determining PL Complexity

To do this, an actual grammar of a programming language will be utilized as a case study to demonstrate the various outcomes offered by the tool.

Python will be the programming language utilized as a case study. This programming is a general-purpose language, adaptable, and powerful. Because it is brief and easy to read, it is a good first language. Python can accomplish just about everything, from web development to machine learning.

The grammar that specifies the Python programming language, developed for ANTLR v4², and the description of the same programming language's features, shown in Example 2, are then considered as input files.

The outcome of the LCE tool's evaluation of the complexity of this language will be reported in two stages: first by the results of the metrics acquired from the grammar, and subsequently by knowledge extraction using the linguistic attributes supplied. Finally, some conclusions obtained from these two phases will be provided.

This tool effectively completes and automates the entire procedure, it also includes visual representations of the grammar structure and the Dependency Graph between symbols (DGS), as can be seen in the Figures 3 and 4.

■ Metrics Results

Tables 3 and 4 show the calculated values for each previously established grammatical metric in Tables 1 and 2.

Size Metrics

■ **Table 3** Results of metrics for evaluating the Size of Context-Free Grammars.

Metric	Result
#P	122
#N	86
#T	89
#UP	6
RHS-Max	7
RHS	3.877
ALT	1.419
MCC	2.721

Structure Metrics

■ **Table 4** Results of metrics for evaluating the Structure of Context-Free Grammars.

Metric	Result
#R	48
FanIn	5.5
FanOut	2.703
TIMP	43.917
CLEV	46.512
NSLEV	2
DEP	35

² The lexer and parser grammar can be found in the grammars-v4 repository of ANTLR.

■ Knowledge

Example 2 offers a brief overview of some of Python's linguistic features. The knowledge extraction received from LCE may be used to confirm the main characteristics of this language, such as:

- It includes five programming paradigms, one of which, object-oriented, has a weight of three units. The tool also creates the following assignments: the functional paradigm received a value of three units, the procedural paradigm received a value of two, and the others received a value of one (reflective and structured). This takes the overall weight associated with the paradigms to 10 units.
- As for the linguistic features themselves, seven were indicated, and four of them did not have an associated weight. The tool assigned a support weight to these features (declaration type, whether it enforces the use of semicolons, Triple-quoted feature, and the anonymous functions feature). Regarding the declaration type of variables and the use of semicolons, since both properties have a great impact on static vulnerability analysis, a weight of 9 and 10 units respectively was assigned. As for the other characteristics, a weight of 5 was assigned to each. This takes the overall weight attributed to the features to 49 units.

■ Complexity Considerations

The considerations, derived automatically by the tool, are actually helpful to the users. It gives complexity assumptions, letting the end user to reason about the language's support.

In terms of grammar, LCE tool produces the following complexity issues:

- A high size of the largest level indicates an uneven distribution of the non-terminals among grammatical levels;
- The CFG is very large, in terms of productions and symbols, with many unitary productions and a low percentage of non-terminal recursive symbols, which contributes to its ease of understanding, manipulation, and maintenance.

In terms of features, LCE tool produces the following complexity issues:

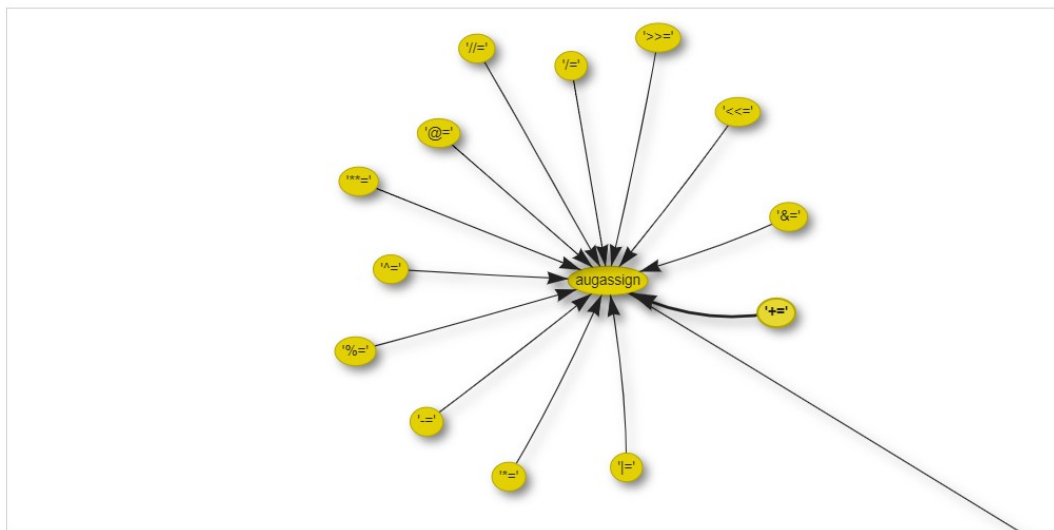
- In a language that does not require the use of semicolons, it makes the support process in a static analysis tool extremely complex, because it involves additional effort to figure out where an instruction begins and ends.
- This language by allowing a type declaration as dynamic does not require explicitly defining the data type when creating a symbol, making vulnerability detection more complex, as it is necessary to perform the compiler's work beforehand in order to know the information inferred by it, thus requiring more support effort.
- This programming language requires its scopes to be delimited by their indentation, so the lexical analyser needs to do some work beforehand, since it needs to count the indentation, detect whether the block opens or closes, or whether it is the same, thus making the support process more time-consuming.

Grammar

-	grammarSpec
-	grammarDecl
-	grammarType
	grammar
-	identifier
	Python3
	;
+	prequelConstruct
+	rules

■ **Figure 3** Visual representation of the grammar.

Dependency Graph



■ **Figure 4** Visual representation of the DGS.

4.2 Predicting the effort to process a Programming Language automatically

In order to predict the effort to process a programming language automatically, MongoDB is used to store the previous languages already analysed for long-term storage, the information stored and used in the comparison are the metrics calculated, and the knowledge extracted.

MongoDB is a NoSQL document database that uses JSON-like documents to store data. MongoDB is used to store all language information.

This information comprises headers, paradigms, features, size metrics, and structure metrics. From the paradigms of a given language, a similarity search is performed with the languages already stored, and then the corresponding languages are presented with their respective values.

Following the same format as the previous part, a new case study will be introduced and analysed, this time to compare the complexity of programming languages.

► **Example 3.** *The following is an example of the description of the properties of the JavaScript programming language, according to the DSL presented above.*

```
JavaScript

PARADIGMS : event-driven, functional, imperative,
            procedural, object-oriented ;

DECLARATION : dynamic $7 ;

INDENTATION : no ;

SEMI-COLON : optional $9 ;

"Scoping" : [
    "Function scoping with 'var'",
    "Block scoping with the keywords 'let' and 'const'"
] $13;

"Weakly typed" : [
    "Means certain types are implicitly cast depending
     on the operation used"
] ;

"Anonymous function" : [
    "A function definition that is not bound to an identifier.",
    "Anonymous functions are often arguments being passed to
     higher-order functions or used for constructing the result of
     a higher-order function that needs to return a function"
] ;

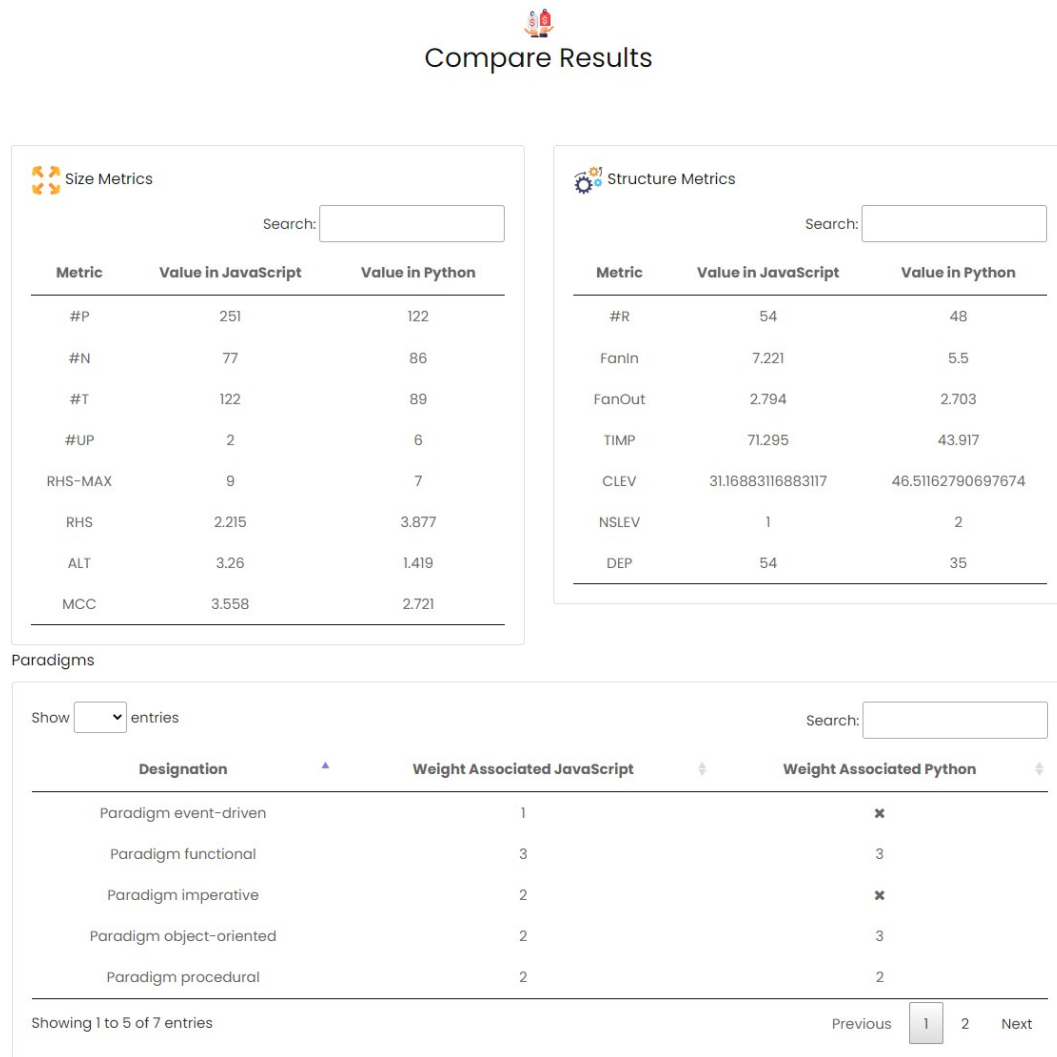
"Run-time environment" : [
    "typically relies on a run-time environment (e.g., a web browser)
     to provide objects and methods by which scripts can interact
     with the environment (e.g., a web page DOM)."
] ;

"First-class functions" : [
    "Function is considered to be an object",
    "A function may have properties and methods",
    "Created each time the outer function is invoked"
] ;

"Promises and Async/await" : [
    "Supports promises and Async/await for
     handling asynchronous operations."
]
.
```

16:14 Determining PL Complexity

The Example 3 shows a description of the properties of another programming language, JavaScript, according to the DSL presented above. With its grammar available in the same ANTLR grammars-v4 repository mentioned above, it is possible to make an automated and readable comparison between the two languages presented through the tool.



■ **Figure 5** Comparison of results between Python and JavaScript.

Figure 5 compares the metrics collected, and the various paradigms introduced, along with their associated weights. Aside from this information, feature descriptions with weights for each language may be found independently.

In summary, the grammar used to express JavaScript is far bigger than Python's since it contains more productions and terminals; nevertheless, at the structural level, the JavaScript grammar is more like a graph in which the symbols are connected, whereas Python's is more like a tree. When it comes to attributes, the tool always depends on what the user describes, but based on these two samples, we can deduce that Python total was 49 and JavaScript total was 55. There are three common paradigms, and only the languages differ in the weight they place on the Object-Oriented Paradigm.

Therefore, the user may simply compare the material supplied by the LCE tool to the languages already supported, allowing them to forecast the work required to support a new programming language in its static analysis tool.

5 Conclusion

Along the paper, we examined ways to assist language support teams in improving their assessment of the time and effort necessary to deal with a programming language that is not yet supported in a Tool for Programming Languages processing, like Static Analysers.

The approach followed for assessing the language's complexity starts by first evaluating the difficulty of its underlying CFG. Factors causing the support process to take longer were found through the investigation of real case studies, particularly during the phases of language recognition and transformation to an AST. In this regard, in a second phase, a collection of linguistic traits is assessed in order to account for the aforementioned elements that also have an influence on language processing.

This research led to the development of a tool, available at <https://lce.di.uminho.pt/>, that allows for the evaluation of a language's difficulty based on a set of metrics to rate the complexity of its grammar with a set of properties. Furthermore, the tool analyses the difficulty of the new language as identified thus far with previously supported languages in order to forecast the effort required to process the new language.

5.1 Future Work

An important task, to be done in future work, is to improve the grammar considerations and the description of the provided features, using real grammars and linguistic properties of programming languages.

References

- 1 Tiago L. Alves and Joost Visser. Metrication of sdf grammars. Technical report, Universidade do Minho, 2005.
- 2 Tiago L. Alves and Joost Visser. A case study in grammar engineering. In *SLE*, 2008.
- 3 Julien Cervelle, Matej Crepinsek, Rémi Forax, Tomaz Kosar, Marjan Mernik, and Gilles Roussel. On defining quality based grammar metrics. *2009 International Multiconference on Computer Science and Information Technology*, pages 651–658, 2009.
- 4 Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, and Gilles Roussel. On automata and language based grammar metrics. *Comput. Sci. Inf. Syst.*, 7:309–329, 2010.
- 5 João Cruz. Qge – An attribute grammar based system to assess grammars quality. Master's thesis, Universidade do Minho, December 2015.
- 6 Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Boston : PWS Publishing Company, 2 edition, 1997. ISBN: 0-534-95425-1.
- 7 Pedro Rangel Henriques. Brincando às linguagens com rigor: Engenharia gramatical. Technical report, Departamento de Informática da Escola de Engenharia da Universidade do Minho, November 2013.
- 8 Terence John Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009. ISBN: 978-1-934356-45-6.
- 9 J.F. Power and Brian A. Malloy. Metric-based analysis of context-free grammars. *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 171–178, 2000.
- 10 J.F. Power and Brian A. Malloy. A metrics suite for grammar-based software. *J. Softw. Maintenance Res. Pract.*, 16:405–426, 2004.

Reasoning with Portuguese Word Embeddings

Luís Filipe Cunha ✉ 

Department of Informatics, University of Minho, Braga, Portugal

J. João Almeida ✉ 

Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Alberto Simões ✉ 

2Ai – School of Technology, IPCA, Barcelos, Portugal

Abstract

Representing words with semantic distributions to create ML models is a widely used technique to perform Natural Language processing tasks. In this paper, we trained word embedding models with different types of Portuguese corpora, analyzing the influence of the models' parameterization, the corpora size, and domain. Then we validated each model with the classical evaluation methods available: four words analogies and measurement of the similarity of pairs of words. In addition to these methods, we proposed new alternative techniques to validate word embedding models, presenting new resources for this purpose. Finally, we discussed the obtained results and argued about some limitations of the word embedding models' evaluation methods.

2012 ACM Subject Classification Computing methodologies → Natural language processing; Computing methodologies → Machine learning

Keywords and phrases Word Embeddings, Word2Vec, Evaluation Methods

Digital Object Identifier 10.4230/OASICS.SLATE.2022.17

Funding *J. João Almeida*: This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

Alberto Simões: This project was funded by Portuguese national funds (PIDDAC), through the FCT – Fundação para a Ciência e Tecnologia and FCT/MCTES under the scope of the project UIDB/05549/2020.

1 Introduction

The use of word embedding models to create words' numerical representations is a widely used technique in natural language processing tasks. In this paper, an analysis of the word embedding models' training and validation was performed where we observed the influence of training parameterization and the classical methods of evaluating these models and measuring their quality.

In this paper, to train word embeddings, three corpora were used: a thematic corpus with reviews from the Portuguese trip advisor; The AC/DC corpus from Linguatca that contains cultural content, literature, news, etc. and corpora from Opus Projec; A corpus of an encyclopedic nature extracted from the Portuguese Wikipedia.

Firstly, we created several models with the same corpora with different hyper-parameters such as vector dimensions and epochs and even created models with different architectures: Skip-gram and CBOW. Then, to make some assumptions about these models, we validated them with the usual evaluation methods, four-word analogies, and measuring two-word distance. Furthermore, during the validation process, we also observed the differences in creating word embeddings with different types of corpora with different sizes, analyzing the results of each model.



© Luís Filipe Cunha, J. João Almeida, and Alberto Simões;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 17; pp. 17:1–17:14
OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In addition to these, we also propose new validation methods and resources to make better judgments about the model's behavior. The created resources are based on the minimum distance classification to a set of class classifiers allowing us to calculate words' polarity, classes and similarities to multiple concepts. During the creation of these resources, we tried to use a domain-specific language, so they were easy to use and adapt to new problems.

We also identified some problems that we faced during this work associated with the word embeddings' evaluation methods and the limitations of this word representation method.

2 Related Work

Recently, interest in word embedding models has been increasing since they have played an essential role in improving the performance of several natural language processing tasks. In English, there is already a wide variety of word embedding models trained in different contexts, however, for Portuguese, it is not always easy to find pre-trained models that satisfy our requirements.

In recent years, several papers were published where word embeddings created from Portuguese corpora were trained and validated. In [15] they trained word embedding models in Portuguese corpora with 536 805 758 tokens and validated them with a set of resources generated in [9, 7, 8, 1] and [2], later translated into Portuguese in [13].

In [6] 31 word embedding models were trained on a Portuguese corpus with 1 395 926 282 tokens using four different techniques: FastText, GloVe, Wang2Vec and Word2Vec. In this paper they evaluated the word embeddings using semantic analogies and task-specific evaluations such as POS and other sentence semantic similarity tasks. Here they concluded that word analogies were not appropriate for word embedding evaluation and that using task-specific evaluations appeared to be a better option.

In [11] they created a new Portuguese test set for Portuguese word embeddings validation. This validation method is focused on lexical-semantic relations and was compared to previous word embeddings evaluation methods.

Other works have been done recently where pre-trained word embeddings have been used to initialize task-specific models in order to transfer their vocabulary knowledge to other classification tasks. An example of this is [3] where several ML models were initialized with pre-trained Portuguese word embeddings and then trained to perform Named Entity Recognition on Portuguese archival documents.

3 Models

In order to train our word embeddings, we used the Word2vec technique with two different models, Continuous Bag-of-Words(CBOW) and Skip-Gram [9]. A CBOW model uses the word's context to predict the current word, i.e., using the N words after and before the token to calculate its representation. Then the model uses a sliding window to go through all the words of the training corpus. Skip-Gram models are similar to CBOW, however, instead of using the word's context to predict the hidden word, given a word, it predicts other words in the sentence, its neighboring words. Training models with both methods allowed the creation of a high dimensional semantic vector space where words with similar semantic meaning are close in distribution.

To train word embedding models, we used the Gensim library [14] that allowed us to fine-tune the model hyper-parameters. In this paper, we trained several different models with different configurations of vector dimensions, epochs and architectures. To keep the models

■ **Table 1** Corpora used for the training of word embeddings.

Corpus	Sentences	Tokens	Characters
Wikipedia-PT	13 551 925	298 505 778	1 905 522 756
TripAdvisor	98 778	1 040 654	6 486 772
LinguOpus	30 346 096	638 513 424	3 670 306 994

organized, we used a notation to name our models depending on the training hyper-parameters: corpus-dimensions-epochs-architecture. For example the model `wiki-300-20-cbow` is a CBOW model trained with 300 dimensions and 20 epochs.

4 Datasets

In this paper, we created several word embedding models using three different datasets (Table 1): Natcorpus-TripAdvisor PT – TripAdvisor reviews, Portuguese Wikipedia and LinguOpus corpus.

Wikipedia-pt

The Portuguese Wikipedia corpus was obtained from a Wikipedia dump made on 2022-04-01, which contained approximately 152 million sentences made up of 903 million tokens. Despite this, the raw format used by Wikipedia contains a lot of non-linguistic elements, so we had to perform some data cleaning. The resultant dataset is constituted only by the textual elements of the original wiki corpus.

Natcorpus-TripAdvisor PT

The Natcorpus-TripAdvisor PT is composed of 13 158 reviews of hotels, near 1 million words, written in Portuguese, from TripAdvisor . These reviews also contain a assessment (1..5). It includes Portuguese from Brazil, Portugal, Angola, Moçambique, Cabo Verde and Timor. The reviews contain emojis abbreviations, errors.

LinguOpus

LinguOpus includes texts in European Portuguese obtained from two main sources: the Linguateca¹ corpus [16] and the Opus Project² [17]. From the first, all corpora tagged as European Portuguese were used. From the second, the Portuguese part of European parallel corpora was used. In addition to these two main sources, it also includes some Portuguese parts from the Per-Fide³ project corpora ([4]), namely Le Monde Diplomatique, The Vatican Corpus and the European Central Bank corpus.

¹ <https://www.linguateca.pt/ACDC/>

² <https://opus.nlpl.eu/>

³ <http://per-fide.di.uminho.pt/>

5 Evaluation Methods and Resources

The classic evaluation method of embeddings models consists of computing word analogies such as $\text{vector}(\text{"King"}) - \text{vector}(\text{"Man"}) + \text{vector}(\text{"Woman"})$ expecting that the resulting vector is closest to the $\text{vector}(\text{"Queen"})$. Another way to validate these models is to calculate the proximity between pairs of words and compare the result with a value estimated by humans, normally between 0 and 10.

5.1 Available Resources

In English, there are several resources that can be used for evaluation purposes, such as [9] and [1]. Some of these resources were later translated to Portuguese, enabling to validate models trained with Portuguese corpora. In this paper, we use the following test corpus to validate our models: LX-4WAnalogies, LX-WordSim-353, LX-SimLex-999 and LX-Rare-Word from [13] and NRC-EIL [10].

- **LX-SimLex-999** – Portuguese translation of [7]. In this dataset, the proximity between pairs of words was calculated on a scale between 0 and 10. During the translation of this dataset, the proximity value was recalculated to adjust to the Portuguese language. In total, it contains 999 pairs of words: 666 pairs of noun-noun, 222 pairs of verb-verb and 111 pairs of adjective-adjective.
- **LX-Rare Word Similarity** – Created from [8]. It contains about 1 017 pairs of words extracted from Wikipedia and WordNet and the proximity score associated with each pair. During the translation of this dataset, the proximity value was recalculated and adjusted to the Portuguese language.
- **LX-WordSim-353** – Created from WordSim-353 [1] containing 353 pairs of words and their proximity scores. All these pairs received a human judgment on a proximity scale from 0 to 10. In this case, the proximity scores were preserved during translation.
- **LX-4WAnalogies** – Created from the Portuguese translation of a test dataset used in [9]. This dataset contains analogies represented by sets of four words, for example, "Brussels Belgium Lisbon Portugal". In total, this dataset contains five sections with 8 869 semantic analogies and nine sections with 10 675 syntactic analogies.
- **NRC-EIL (PT-BR)** – The NRC Emotion Intensity Lexicon [10] contains a list of words paired with eighth basic emotions (anger, anticipation, disgust, fear, joy, sadness, surprise, and trust) and their proximity score. In this paper, we used the Portuguese version of this resource that was translated automatically using Google Translate in 2018.
- **LX-Battig** – Portuguese translation of an English dataset [2]. This resource contains 83 terms associated with ten different categories: mammals, birds, fish, vegetables, fruit, trees, vehicles, clothes, tools and kitchenware.
- **TALES** – *Teste para Analogias Lexico-Semanticas* (TALES) [11] is a Portuguese testset focused on lexical-semantic relations, that uses the analogies method, however, accepting more than one acceptable answer for each analogy.

The creation of this resource was inspired by [5] but created from scratch for the Portuguese Language, containing relations such as Hypernymy, Meronymy, Synonymy and Antonymy. It is a balanced testset with a total of 14 test files with 50 entries each.

5.2 Created Resources

In addition to using several already available resources to validate the quality of our word embedding models, we decided to create our own resources and methods in order to have more information about our models' behavior. Thus, we present Class-Sim (Classifier by Similarity) and Analogy-DD (Analogy Declarative Description).

■ **Listing 1** NAT-WordSim-1 dataset (YAML) used for Class-Sim method.

```

- section: polarity
  clas:
    - [bom]
    - [mau,horrivel,fujam]
  testes:
    - [ acolhedor , 0 ]
    - [ adorei , 0 ]
    - [ ridiculo , 1 ]
    - [ amavel , 0 ]
    - [ ruidoso , 1 ]
    - [ horror , 1 ]
- section: class-animais
  clas:
    - [ave]
    - [mamifero, cao, gato]
    - [peixe]
  testes:
    - [ melro , 0]
    - [ aguia , 0]
    - [ macaco , 1 ]
    - [ elefante , 1 ]
    - [ tubarao , 2 ]

```

Classifier by Similarity

The first method that we created was a classifier by similarity (Class-Sim), which tests whether the model is able to associate words with its corresponding class. For that, we created a new YAML dataset **NAT-WordSim-1** (Listing 1) that is divided into sections. Each section contains multiple classes and a set of words associated with one of those classes. The principle behind this method is for the model to measure the cosine similarity of each word with all the classes of the section. Then it assigns each word to a class by choosing the minimum distance value of the word to all the available classes. Note that each class can be made up by more than one word. Then, we validate if the words get assigned with the correct classes.

The resources used to perform this evaluation method are publicly available ⁴ containing a dataset with three classes: Geo (27 samples), class of Animals (22 samples) and Polarity (86 samples) with a total of 135 word samples. The polarity samples were extracted from the TripAdvisorPT corpus.

Finally, the corpus LX-Battig was also parsed into the YAML format present in Listing 1 to use it with the Class-Sim evaluation method.

Analogy-DD

Analogies are a very elegant way to show / see if a word-embedding captured a relation (semantic, syntactic, ...) between terms.

The analogy tests, previously described, focus in a set of relations that related to encyclopedic knowledge (country-capital, country-currency, city-in-state), grammatical relations, and family relation.

With Analogy-DD initiative, we intend:

- to discuss analogies, their underlying relations and properties;
- to define a declarative notation to describe facts and analogy types;
- to create a sets of facts based on relations;
- to build a set of tools that generates analogies from the previous facts. The generated analogies tests

The problem of analogies and relations is in fact complex. Relations have different signatures and properties.

⁴ <https://github.com/lfcc1/slate22-wemb>

17:6 Reasoning with Portuguese Word Embeddings

Consider the following situations:

- multiple answers. Example:
 - country-people: *England : English :: Poland : x* \in {*polaco, polonês*}
- transitive relation. Example “is-a” (classification) is a transitive relation; cat is a feline, but also a mammal and an animal:
 - is-a: *ant : insect :: cat : x* \in {*feline, mammal, animal*}
- multi word elements. Example:
 - capital-country: *Lisboa : Portugal :: Londres : x* \in {*Reino Unido, Inglaterra*}

Consider the following extract from a Analogy-DD in Portuguese:

```
# Relações para uso na geração de Analogias (e outros)
# V0.3 2022-05-01

# $1(ferramenta) é_usado_para $2(actividade,V)

martelo :: pregar, martelar
verruma :: furar
chave de fendas :: aparafusar
berbequim :: furar
lixa :: lixar
serrote, serra :: serrar
```

Notes:

- line 1,2 – metadata
- line 3 – defines a schema of a new section:
 - `$1(ferramenta)` – a value selected from the the first column (this value is a *ferramenta* (tool),
 - `é_usado_para` – (is used to) name or the relation,
 - `$2(actividade,V)` – value selected from second column (a activity, POS=verb)
- line 4..9 – tuples. Example: *lixa :: lixar* can be read *lixa é_usada para lixar*.
- An analogy is built by joining 2 different tuples. (Ex: *serra : serrar :: verruma : ?furar*).
- Some assessment tools may need explicit constrains (no multi-word allowed on the elements (like *chave de fendas*), or no variants allowed on the answer (like *pregar* or *martelar*) – .

Analogy-DD may have:

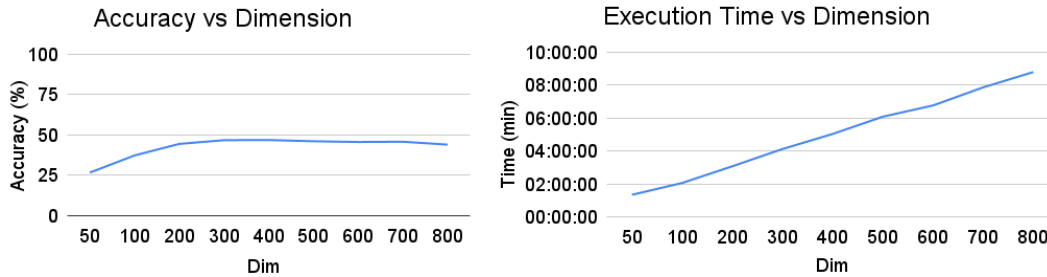
- several sections;
- with several relation schema;
- tuples may have 2 or more fields;
- each field may have several values (separated by “,”).

The current version of the Portuguese Analogy-DD has 37 sections (near 37 relations), 410 tuples, and generates more then 5000 analogy lines⁵.

⁵ This number may change according to the selected options.

6 Results

In this section, the results of the experiments made in this work are presented. Initially, we trained several models with vectors of different dimensions in order to analyze the impact of this hyper-parameter on the model's performance and computational cost.

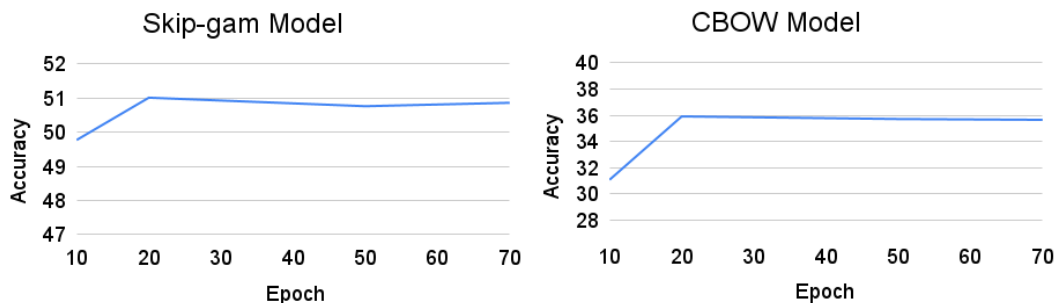


■ **Figure 1** Comparison of vector dimension with accuracy and execution time on LX-4Analogies task.

In Figure 1 we have a performance analysis of models trained on the Portuguese Wikipedia corpus, where several models with different word embeddings dimensions were trained and then validated with the LX-4Analogies dataset. As we can see in Figure 1 (Left), the model trained with 50 dimensions obtained an accuracy of 25%. As we increased the dimensions of the word embeddings, the model's accuracy also increased, reaching values of approximately 50% with 300 dimensions vectors. However, models trained with dimensions number greater than 300 were not able to obtain higher results. In fact we registered an accuracy decrease in models trained with 700 and 800 vector dimensions.

On the other hand, in Figure 1 (Right), we performed an analysis at each model's validation execution time. This Figure shows that the execution time is proportional to the number of dimensions used to train the models. Thus, since models trained with dimensions greater than 300 did not obtain superior performances, we decided to use models with 300 dimensions to perform the tests in this paper. This dimensional value is consistent with most of the related work done in this field where several researches [12],[9],[6] concluded that the models achieved the best performance by using 300 dimensions.

Then we also analyzed the models' behavior with different epoch values.



■ **Figure 2** Comparison of models epochs with accuracy on LX-4Analogies task.

■ **Table 2** Models' accuracy results with Class-Sim method and Analogies.

Model	NAT-WordSim-1	LX-Battig	Analogies-DD	LX-4WAnalogies
TripAdvisorPT	95.06	oov	5.19	4.13
LinguOpus	79.55	41.77	21.04	29.86
wiki-300-70-sg	86.36	66.67	23.91	50.86
wiki-300-50-sg	86.36	62.82	24.29	50.76
wiki-300-20-sg	84.78	67.95	22.74	51.01
wiki-300-10-sg	84.85	66.67	22.57	49.77
wiki-300-70-cbow	75	38.46	21.26	35.65
wiki-300-50-cbow	71.21	38.46	21.43	35.70
wiki-300-20-cbow	74.24	39.74	21.65	35.91
wiki-300-10-cbow	75	34.62	20.35	31.08

In Figure 2 we can see the accuracy of the model trained with different epoch numbers, validated on the LX-4Analogies dataset. Increasing the model epochs from 10 to 20, we see that the models increased their performance, especially the CBOW model from 32% to 36%, which is a considerable performance gain. However, going upwards with the epoch number, the models' performance remained stable in this concrete task.

6.1 Analogies and Class-Sim

Then we validated the models with our evaluation resources, starting with the Class-Sim and analogies resources. In Table 2 we can observe the results of the Class-Sim and analogies evaluations on eight models, six of them trained on the Portuguese Wikipedia corpus with the Skip-gram (sg) and CBOW (cbow) architectures, vector dimensions of 300 trained with 10, 20, 50 and 70 epochs. Note that some models were not able to perform some validation tests due to the excess of out of vocabulary words (oov).

Firstly we have the NAT-WordSim-1 test corpus evaluated with the Class-Sim method. In this evaluation, we can see that the TripAdvisorPT was the model that obtained higher results. In fact, this evaluation dataset contains a high number of polarity test samples extracted from the TripAdvisor reviews. One could say that this model was the one to create better polarity associations between the words used in this test resource. This is expected since the TripAdvisorPT model was trained with the same data as this evaluation test.

We can also see that the models trained on the Wikipedia corpus with Skip-gram architecture could obtain better results than those trained with CBOW.

Then we have another Class-Sim evaluation test with the LX-Battig. In this case, the TripAdvisor model was not able to produce any results because it was trained in a very specific domain (reviews from Trip Advisor). As such, it cannot recognize most of this test resource vocabulary. The model that obtained higher results in this test case was the wiki-300-20-sg.

Secondly, we have analogies evaluation resources. Again, the TripAdvisorPT model obtained low results in these test cases due to its limited vocabulary. On the other hand, some models were able to achieve results of 51.01% in the LX-4WAnalogies dataset and 24.29% on Analogies-DD. As stated in Section 5, the Analogies-DD resource was created in

■ **Table 3** LX-4WAnalogies accuracy results by section.

Section	wiki-300-20-sg	wiki-300-20-cbow	TripAdvisorPT	LinguOpus
capital-common-countries	83.55	47.4	oov	25.97
capital-world	67.65	30.48	oov	19.09
currency	28.92	6.59	oov	5.58
city-in-state	51.01	13.94	oov	4.29
family	59.29	57.62	13.33	48.57
gram1-adjective-to-adverb	9.85	13.05	0	10.47
gram2-opposite	23.19	24.82	3.57	19.38
gram3-comparative	56.67	63.33	16.67	43.33
gram4-superlative	8.18	14.55	3.57	9.52
gram5-present-participle	69.57	72.15	0	65.44
gram6-nationality-adjective	76.46	61.87	0	54.19
gram7-past-tense	58.11	59.32	1.39	45.31
gram8-plural	47.7	29.6	3.33	31.01
gram9-plural-verbs	39.15	46.3	4.17	50.49
Total accuracy	51.01	35.91	4.13	29.86

order to generate a higher variety of analogies. We consider that the generated analogies are harder to predict since most of them would accept multiple words as a correct answer. Considering this property, the models are expected to obtain lower validation values.

After this, we decided that it was crucial to create a view of these evaluations that was able to show the analogies' results by section. In Table 3, we can analyze the results of the models' validation on the LX-4WAnalogies resource, being possible to observe the model's accuracy associated with each domain section. A view that allows us to analyze the results of each section instead of total accuracy gives us more accurate information about where our model fails, allowing us to make better judgments about its behavior.

Analyzing the models trained on Wikipedia, the Skip-Gram model obtained better overall results than the CBOW model, achieving an accuracy of 51.01% and 35.91%, respectively. In fact, the Skip-Gram model managed to obtain results above 80% in the section of capitals and common countries, 67.65% in the remaining capitals of the world, and 76.46% in the nationality-adjective section. Since the Portuguese Wikipedia corpus is encyclopedic, it contains information about all countries and the relationships of their corresponding capitals and nationalities, making this type of evaluation strongly favorable to this model. However, when we look at the grammatical sections, we see that this model obtains lower results, reaching accuracy values of 9.85% and 8.18% in the gram1-adjective-to-adverb and gram4-superlative sections. These results may occur because Wikipedia is not a grammatical corpus and does not contain enough information about the words' morphology.

To learn the word's grammatical relations the model would need a larger variety of morphological and grammatical samples. These are more frequent in literary and everyday texts, and less frequent in encyclopedic corpora. For example, the adjectives' superlatives are rarely present in Wikipedia texts.

On the other hand, we verified that the model trained with the Portuguese TripAdvisor corpus obtained lower results in this test, 4.13% of accuracy overall. Again, the reason for this could be that this model was trained with fewer data in a specific context, limiting

■ **Table 4** Models' word similarity validation results with Pearson/Spearman correlation coefficient.

Model	LX-WordSim-353	LX-SimLex-999	LX-Rare Word	NRC-PT
TripAdvisorPT	28.70 / 25.27	22.84 / 21.73	33.08 / 29.80	14.85 / 12.57
LinguOpus	42.49 / 41.69	25.65 / 22.74	49.67 / 48.34	22.93 / 21.97
wiki-300-70-sg	52.53 / 56.00	34.16 / 32.41	50.69 / 49.79	28.16 / 26.63
wiki-300-50-sg	52.53 / 56.00	34.16 / 32.41	50.76 / 49.75	28.16 / 26.63
wiki-300-20-sg	52.50 / 55.67	34.90 / 33.06	51.63 / 50.67	28.28 / 26.70
wiki-300-10-sg	51.97 / 55.15	35.63 / 33.96	51.89 / 50.70	28.77 / 27.39
wiki-300-70-cbow	41.26 / 41.57	26.74 / 24.29	48.16 / 46.46	26.08 / 24.42
wiki-300-50-cbow	41.63 / 42.42	26.86 / 24.45	48.02 / 46.28	26.15 / 24.57
wiki-300-20-cbow	41.77 / 42.50	27.91 / 25.54	47.65 / 45.65	25.81 / 24.38
wiki-300-10-cbow	41.73 / 42.56	26.02 / 23.80	44.63 / 42.80	24.03 / 22.79

its acquired knowledge to the domain of its training data. We also verified that it did not obtain a classification in some sections. This was because the model's vocabulary is not extensive enough to represent words of some domains, such as the countries, capitals, and nationalities present in the LX-4WAnalogies dataset. In this way, we could not perform some of the validation tests.

6.2 Word Similarity Results

With the Class-Sim and analogies tests validated, we proceeded with another evaluation method, Word similarity.

In Table 4 we have the validation results of the models trained with different epochs. We performed word similarity tests in this validation, measuring two different metrics to validate the word embedding models, the Pearson and Spearman correlation coefficient. As can be seen, contrary to tests with analogies, in the word similarity tests, CBOW models are able to obtain results similar to Skip-gram models, sometimes even superior.

Looking at the results' variation based on the number of epochs of the models, we can say that this parameter did not have significant relevance to the results, mainly from 20 epochs upwards.

Again the TripAdvisorPT was the model that obtained the worst results for the same reason we already discussed in the analogies validation. On the other hand, the Wikipedia model was the one that obtained the higher correlation coefficients, followed by the LinguOpus model.

Looking at the validation tests, we can see that the NRC-PT was the test where the models produced the worst results. One possible reason could be that this dataset was translated from English to Portuguese using Google Translate, keeping the word proximity scores. However, the semantic similarity between pairs of words may change across different languages due to linguistic and cultural differences, negatively affecting the models' results.

■ **Table 5** TALES accuracy results by relation category.

Relations	wiki-300-20-sg	Correct Questions	Total Questions
Synonym-of-N	12.01	3824	31852
Synonym-of-V	12.64	4337	34302
Synonym-of-ADJ	10.47	3079	29402
Antonym-of-ADJ	20.20	495	2450
Purpose-of-D	12.27	601	4900
Purpose-of-Inv	11.12	817	7350
Part-of-D	10.54	2583	24500
Part-of-Inv	9.90	2668	26950
Hypernym-of-abstract	11.07	1085	9800
Hypernym-of-concrete	12.20	1495	12250
Hypernym-of-ACCAO	12.69	1866	14700
Hypernym-of-ACCAO-Inv	13.09	2245	17150
Hypernym-of-Inv-abstract	12.03	2357	19600
Hypernym-of-Inv-concrete	11.53	2542	22050
Total accuracy	10.62	27326	257256

6.3 TALES

In order to use TALES resource we had to use the Vecto package⁶. Vecto was used with default parameters i.e., accuracy for the performance measurement and vector offset (3CosAdd) [9] for the analogy solving method. Only one model was used in the TALES validation method, which was the `wiki-300-20-sg` model. In this validation test we can observe the performance of our model in 14 different lexical-semantic relations subcategories, Table 5.

In this test we were able to achieve 10.62% of accuracy, with our best score being the Antonym-of-ADJ relations with 20.20% and the Part-of-Inv relations being the validation test with lower results, 9.9% accuracy. Comparing this results with the original TALES paper, in [11], using the word2vec-SkipGram architecture they were able to achieve 10% of accuracy which is almost the same value achieved by our model.

7 Problems during validation

During the word embeddings validation, we encountered some problems that could negatively affect the models' performance measurement.

Starting with the already available datasets used in this paper, in the case of the words similarity evaluation, we have seen that some of these resources are translations from English to Portuguese, preserving the similarity score between pairs of words. Despite some of the translations being made carefully involving more than one human translator, keeping

⁶ <https://github.com/vecto-ai>

17:12 Reasoning with Portuguese Word Embeddings

the same similarity value can be problematic due to the cultural and semantic linguistics differences between different languages. Sometimes it is impossible to map words between different languages while keeping the exact same meaning.

For example, in the resource LX-WordSim-353 (English version) we have the following pairs of words, "football" and "soccer" with a similarity score of 9.03, which makes sense since football and soccer are both sports and can share much semantic meaning between them. However, looking at the Portuguese version of this resource, this entry was translated to `futebol futebol 9.03`. In Portugal, soccer is one of the most influential national sports, and when the word football is mentioned, it usually refers to the English word soccer. In this example, both "football" and "soccer" were translated to the same word, "futebol", however, the similarity score remained the same. In this case, the model will measure the distance between the same word vector, which will return a proximity score of 10 (maximum proximity value), however, the validation expects it to return a proximity score of 9.03.

Another problem found in these resources during the translation process was that some words were translated into multiple words. For example the the entry "gem jewel 8.96" was translated to "*pedra preciosa joia 8.96*" where the word "gem" is translating to "pedra preciosa". This evaluation entry could be invalid for evaluating word embedding models that use vectors to represent single words.

Finally, when we validated our models with the LX-Battig resource with the Class-Sim method, we encountered a limitation in the word embeddings mechanism. This evaluation classified each sample class by minimizing the distance between the words and all the classes. Using the Wikipedia model, we were able to achieve results of 67.95%. Then we decided to analyze the samples that the model failed to classify to draw conclusions about the model's behavior. Some words the model was unable to classify included "cat", "lion" and "tiger", which the model classified as fish instead of mammal. In this case, we found out that the model considers the words cat, lion, and tiger closer to the class fish because it is association these tokens with the cat fish, lion fish, and tiger fish which are all described in detail in the Portuguese Wikipedia.

Other words the model failed to identify were "*Pereira*" and "*Oliveira*" (pear and olive tree), which were identified as person names instead of tree names. In Portugal, due to historical reasons, some tree names are common names of people, which confuses the model. In this case, using only one vector to represent a word can cause ambiguity when a word can be interpreted differently depending on its context domain.

8 Conclusions and future work

Some final claims:

- Word embeddings' results are very sensitive to changes on the configuration parameters used in the creation.
- The preprocess stage is very important.
- The available public resources for testing have several important issues, and often are concerned with irrelevant questions. Nevertheless, they are very useful for tuning and improving models. In some of these evaluation tests, 50% of their entries test whether a successful model can determine the capital of a country. We believe that this type of test may not be an optimal indicator to generalize the performance of the model.
- It is crucial to analyze specific examples of the discordant results in order to understand what is being tested and is being learned (surprises are frequent).

- Designing new tests and new use cases proved to be challenging and rich. We end up building a set of functions that will probably constitute a python toolkit in the near future.

For future work, one experiment that would be interesting is to use all the corpora used in this paper to train only one final model and re-evaluate it to understand how the merge of data would affect its performance. Another experiment would be to create contextualized word embeddings, generating multiple word vectors for the same word depending on its context domain. This would allow overcoming some problems we observed in this paper, where a word with different meanings could be hard to disambiguate.

As for our validations resources, we intend to increase the size of the Analogias-DD and NAT-WordSim-1 datasets, adding more samples from a wider variety of domains. By doing so, we would be improving the quality of this evaluation method.

References

- 1 Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Paşca, and Aitor Soroa. A study on similarity and relatedness using distributional and WordNet-based approaches. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 19–27, Boulder, Colorado, June 2009. Association for Computational Linguistics. URL: <https://aclanthology.org/N09-1003>.
- 2 Marco Baroni, Brian Murphy, Eduard Barbu, and Massimo Poesio. Strudel: A corpus-based semantic model based on properties and types. *Cognitive Science*, 34(2):222–254, 2010. doi:10.1111/j.1551-6709.2009.01068.x.
- 3 Luís Filipe da Costa Cunha and José Carlos Ramalho. Ner in archival finding aids: Extended. *Machine Learning and Knowledge Extraction*, 4(1):42–65, 2022. doi:10.3390/make4010003.
- 4 Idalete Dias, Sílvia Araújo, Alberto Simões, José Almeida, Nuno Carvalho, Ana Oliveira, and André Santos. *The Per-Fide Corpus: A New Resource for Corpus-Based Terminology, Contrastive Linguistics and Translation Studies*, pages 177–200. Bloomsbury, April 2014.
- 5 Aleksandr Drozd, Anna Gladkova, and Satoshi Matsuoka. Word embeddings, analogies, and machine learning: Beyond king - man + woman = queen. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3519–3530, Osaka, Japan, December 2016. The COLING 2016 Organizing Committee. URL: <https://aclanthology.org/C16-1332>.
- 6 Nathan Hartmann, Erick Fonseca, Christopher Shulby, Marcos Treviso, Jessica Rodrigues, and Sandra Aluisio. Portuguese word embeddings: Evaluating on word analogies and natural language tasks, 2017. doi:10.48550/ARXIV.1708.06025.
- 7 Felix Hill, Roi Reichart, and Anna Korhonen. SimLex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695, December 2015. doi:10.1162/COLI_a_00237.
- 8 Thang Luong, Richard Socher, and Christopher Manning. Better word representations with recursive neural networks for morphology. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 104–113, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL: <https://aclanthology.org/W13-3512>.
- 9 Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 2013.
- 10 Saif M. Mohammad. Word affect intensities. In *Proceedings of the 11th Edition of the Language Resources and Evaluation Conference (LREC-2018)*, Miyazaki, Japan, 2018.
- 11 Hugo Gonçalo Oliveira, Tiago Sousa, and Ana Oliveira Alves. Tales: Test set of portuguese lexical-semantic relations for assessing word embeddings. In *HI4NLP@ECAI*, 2020.

17:14 Reasoning with Portuguese Word Embeddings

- 12 Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL: <http://www.aclweb.org/anthology/D14-1162>.
- 13 Andreia Querido, Rita Carvalho, Joao Rodrigues, Marcos Garcia, Joao Silva, Catarina Correia, Nuno Rendeiro, Rita Pereira, Marisa Campos, and António Branco. Lx-lr4distsemeval: a collection of language resources for the evaluation of distributional semantic models of portuguese. *Revista da Associação Portuguesa de Linguística*, 3:265–283, September 2017. doi:10.26334/2183.
- 14 Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. URL: <http://is.muni.cz/publication/884893/en>.
- 15 João Rodrigues and António Branco. Finely tuned, 2 billion token based word embeddings for Portuguese. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). URL: <https://aclanthology.org/L18-1382>.
- 16 Diana Santos and Eckhard Bick. Providing Internet access to Portuguese corpora: the AC/DC project. In *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00)*, Athens, Greece, May 2000. European Language Resources Association (ELRA). URL: <http://www.lrec-conf.org/proceedings/lrec2000/pdf/85.pdf>.
- 17 Jörg Tiedemann. Parallel data, tools and interfaces in opus. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Mehmet Ugur Dogan, Bente Maegaard, Joseph Mariani, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, May 2012. European Language Resources Association (ELRA).

ScraPE – An Automated Tool for Programming Exercises Scraping

Ricardo Queirós  

CRACS – INESC-Porto LA, Portugal
uniMAD, ESMAD/P.PORTO, Portugal

Abstract

Learning programming boils down to the practice of solving exercises. However, although there are good and diversified exercises, these are held in proprietary systems hindering their interoperability. This article presents a simple scraping tool, called ScraPE, which through a navigation, interaction and data extraction script, materialized in a domain-specific language, allows extracting the data necessary from Web pages – typically online judges – to compose programming exercises in a standard language. The tool is validated by extracting exercises from a specific online judge. This tool is part of a larger project where the main objective is to provide programming exercises through a simple GraphQL API.

2012 ACM Subject Classification Applied computing → Computer-managed instruction; Applied computing → Interactive learning environments; Applied computing → E-learning

Keywords and phrases Web scrapping, crawling, programming exercises, online judges, DOM

Digital Object Identifier 10.4230/OASlcs.SLATE.2022.18

1 Introduction

Programming courses are part of the curriculum of many engineering and science programs. These courses rely on programming exercises to foster practice, consolidate knowledge and evaluate students. The enrolment in these courses is usually very high, resulting in a great workload for the faculty and teaching assistants. In this context the availability of many and diversified programming exercises from different sources is of great importance [4]. Unfortunately, there are only a few sources to get, in an automatic way, programming exercises. Some notable examples are the online judges, which can be defined as repositories of programming exercises with automatic evaluation capabilities. These systems are often used by students around the world to train for programming contests such as the International Olympiad in Informatics (IOI)¹, for secondary school students; the ACM International Collegiate Programming Contests (ICPC)², for university students; and the IEEEExtreme³, for IEEE student members. Despite their usefulness, these systems do not have a simple mechanism to obtain programming exercises (e.g. an API). In fact, only a few offer interoperability features such as standard formats for their exercises and APIs to foster their reuse in an automated fashion. In this field, the most notable APIs for computer programming exercises consumption are CodeHarbor⁴, FGPE AuthorKit⁵, and Sphere Engine⁶. Still, they are not simple to use and expose a small number of exercises.

¹ <https://ioinformatics.org/>

² <https://icpc.global/>

³ <https://ieeextreme.org/>

⁴ <https://github.com/openHPI/codeharbor>

⁵ <https://github.com/FGPE-Erasmus/authorkit-api>

⁶ <https://sphere-engine.com/>



© Ricardo Queirós;
licensed under Creative Commons License CC-BY 4.0

11th Symposium on Languages, Applications and Technologies (SLATE 2022).

Editors: João Cordeiro, Maria João Pereira, Nuno F. Rodrigues, and Sebastião Pais; Article No. 18; pp. 18:1–18:7

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This poses a big problem for teachers who, due to lack of time, often resort to exercises from previous years. This recurrence hinders diversification and innovation in the practical part of programming courses, which is crucial for their evolution in this area.

This article presents a tool called ScraPE that allows, through a script formalized by a very simple domain-specific language (DSL), to extract data from Web pages (mostly online judges). The script defines a set of steps to navigate, interact and extract data to compose a programming exercise and its direct serialization to a standard language (YAPeXIL [3]). The tool will be used to mitigate the cold-start problem [5] in a larger project where the objective is to provide a simple and flexible GraphQL API for accessing programming exercises that can be consumed by several learning systems.

The remainder of this paper is organized as follows. Section 2 analyzes several of existing online judges to select the most suitable to feed a repository of programming exercises. Section 3 presents an automatic scraping tool to extract programming exercises. Then, in order to evaluate the effectiveness and efficiency of this approach, in Section 4, a report on the use of ScraPE in the TIMUS online judge is presented. The final section summarizes the main contributions of this research and plans future developments of this tool.

2 Online Judges

An Online Judge (OJ) is a system with a set of programming exercises that can be used by anyone to practice for programming contests. These systems can compile and execute your code, and test your code with predefined data. The code being submitted may run with restrictions, including time and memory limit, and other security restrictions. The output of the executed code will be compared with the standard output. The system will then return the result. When the comparison fails, the submission is considered unsuccessful and you need to correct any errors in the code, and resubmit for re-judgement.

Although there are several online judges, they do not provide any kind of API hindering its automatic consumption. In addition, those who provide these API return exercises in disparate formats, which leads to the need to use converters to harmonize formats. With this scarcity of exercises and given the difficulty of creating promptly good exercises, teachers often reuse exercises from previous years, which limits creativity [1].

In this section we survey online judges that present programming exercises. Since there are a large number of online judges, a set of criteria was applied to filter the set and thus obtain those that will be the most suitable to be used as a data source for the system to be implemented.

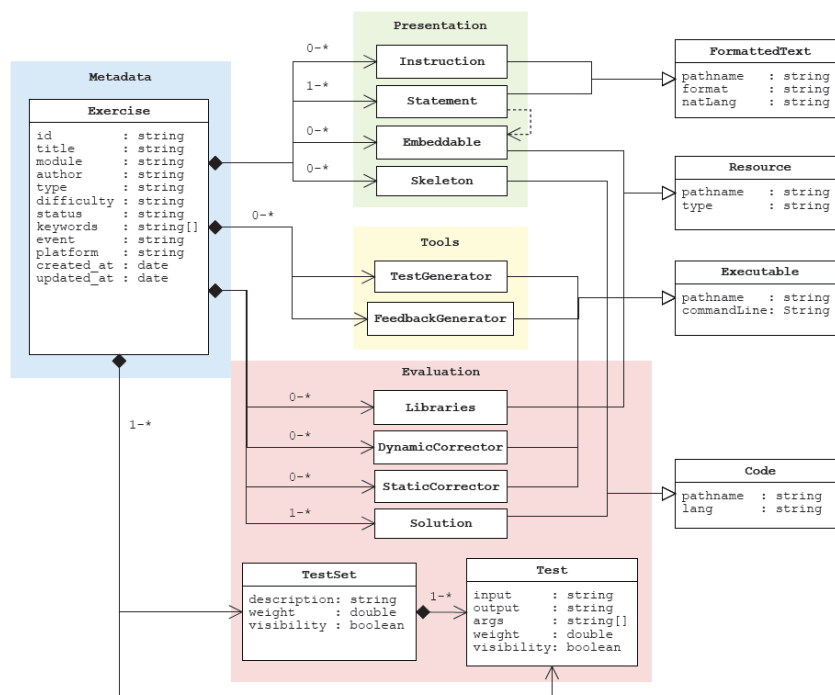
In a first phase we select 72 online judges. Then, in order to narrow the dataset we applied sequentially a set of criteria:

1. Statements in English language;
2. Statements in HTML format;
3. Public problem set (without the need to register/login in the OJ)
4. Minimum number of exercises ($n_{Ex} \geq 1000$)

Based on these filter criteria, only 17 OJs were selected. Then, all OJs were analyzed and validated according to their coverage in the YAPeXIL format [3]. The YAPeXil format is currently the most expressive format to represent a programming exercise [2]. It is formalized by a YAPeXIL JSON Schema (Figure 1) which can be divided into four separate facets:

- **Metadata** – which contains simple properties providing information about the exercise (i.e., a description, the name of the author of the exercise, a set of keywords relating to the exercise, the level of difficulty, the current status, and the timestamps of creation and last modification;

- **Presentation** – which relates to what is presented to the student (i.e. the statement – a formatted text file with a complete description of the problem to solve – embeddables – an image, video, or another resource file that can be referenced in the statement –, and skeleton – a code file containing part of a solution that is provided to the students);
- **Assessment** – which encompass what is used in the evaluation phase (i.e. solution – a code file with the solution of the exercise provided by the author(s), test – a single public/private test with input/output text files, a weight in the overall evaluation, and a number of arguments –, and test_set – a public/private set of tests);
- **Tools** – which includes any additional tools that the author may use in the exercise (i.e. generate the feedback to give to the student about her attempt to achieve a solution and the test cases to validate a solution).



■ **Figure 1** YAPExIL data model.

Each Online Judge was analyzed and its coverage in the 4 facets was verified. Table 1 presents the results of this study.

Based on these results, we can state that LeetCode, CodeChef, TIMUS, URI and Kattis are the OJs with higher YAPExIL coverage values, thus offering a higher guarantee that the exercises provided by the future API are more complete in terms of information for the end user.

3 ScraPE

ScraPE is a basic tool for scraping online judges on data related with programming exercises. The ultimate goal of this tool is to be used as a cold-start facilitator in a bigger system currently being developed which aims to provide a GraphQL API to anyone that want to get free programming exercises. This system will be based on a GraphQL server (Apollo)

18:4 ScraPE – An Automated Tool for Programming Exercises Scraping

■ **Table 1** Online judges comparison based on YAPExIL covereness.

Online Judges	#exercises	YAPExIL facets				TOTAL
		Metadata	Presentation	Assessment	Tools	
UVA	4300	20%	0%	0%	0%	5,00%
TIMUS	1157	95%	50%	0%	0%	36,25%
URI	2296	95%	50%	0%	0%	36,25%
Peking	3054	90%	45%	0%	0%	33,75%
Zhejiang	3179	75%	35%	0%	0%	27,50%
Kattis	3380	95%	50%	0%	0%	36,25%
LeetCode	2262	95%	50%	25%	0%	42,50%
CodeForces	78013	80%	25%	0%	0%	26,25%
DMOJ	4233	75%	25%	0%	0%	25,00%
Dunjudge	1707	80%	25%	0%	0%	26,25%
TopCoder	2122	65%	25%	0%	0%	22,50%
CodeChef	5001	95%	50%	25%	0%	42,50%
E-olymp	8325	85%	25%	0%	0%	27,50%
Toph	1548	90%	25%	0%	0%	28,75%
Hackerearth	1612	75%	25%	0%	0%	25,00%
LightOJ	1025	80%	25%	0%	0%	26,25%
Aizu	3023	85%	25%	0%	0%	27,50%

composed by a GraphQL schema, a resolver, a noSQL database where the exercises will be stored in YAPExIL format and a HTTP client to expose the API. Learning systems and/or individuals will use this API to feed their courses.

3.1 The schema

ScraPE uses a DSL to represent a script which is responsible by all the actions made on web pages from navigating to extracting data. The DSL is formalized as a JSON Schema. Listing 1 presents the action sub-schema, as will be explained below.

■ **Listing 1** Action schema.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "A schema to formalize an Action",
  "type": "object",
  "properties": {
    "page": { "type": "string" },
    "query": { "type": "string" },
    "type": { "type": "string" },
    "output": { "type": "string" },
    "actions": { "type": "array", "items": { "$ref": "#/defs/Action" } }
  },
  "required": ["type", "query", "output"]
}
```

The Action sub-schema is composed by five properties. The **page** property is the web page where the scraper will start extracting data. The **query** property represents a CSS selector that will be used to find the desired DOM nodes. The **type** property is a enumeration of all the action types that can be made in the selected element:

- GET – get DOM element(s) or attribute(s) based on a query;
- FILL – inject text in a selected text box or select an item in a selected combo box;
- CLICK – click in a selected button, radio button or checkbox.

The `output` property is a string which defines the name of the property to be created in the output file.

The execution of a query can result in multiple nodes. In this case, it is necessary to iterate over all of them and perform certain actions. This is the case, for example, of primary-secondary pages where a Web page has multiple links and where we need to enter and perform a set of actions on each of them. For this particular scenario we could have an `actions` array inside an `action` property.

4 Use Case: Timus Online Judge

Based on the results of Section 2, the effectiveness and efficiency of ScraPE was validated using one of the top-3 online judges as data source. The chosen one was the TIMUS Online Judge.

The first step was the construction of the JSON instance representing the script to be used in the scraping process. Since this process is (still) manual we use the Inspector tab of the browser Developer Tools to get all the desired CSS selectors.

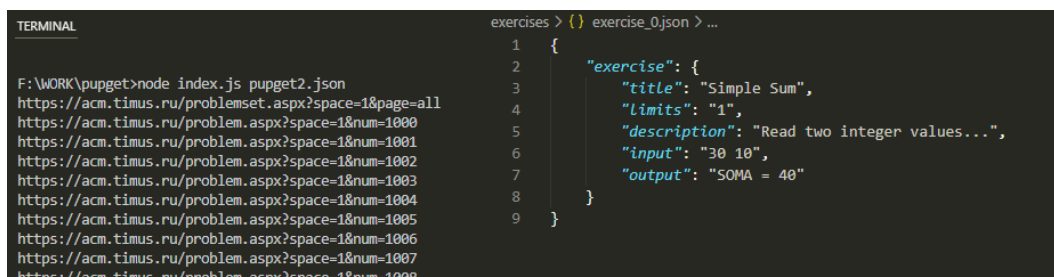
The next step was to refine the script with the action types and the output names for the output files being generated. It should be noted that the script generates an output file in an internal ScraPE format. Afterwards the file will be transformed into the target format (ideally YAPEXIL) using eXtensible Stylesheet Language (XSL) files. This way, it will be easier to scale ScraPE to other desired formats. The final script is presented in Listing 2

Listing 2 Final script.

```
"actions": [{
  "page": "problemset.aspx?space=1&page=all",
  "query": "td.name a",
  "type": "GET",
  "output": "exercise",
  "actions": [{
    "type": "GET",
    "query": "h2",
    "output": "title"
  }, {
    "type": "GET",
    "query": "div.problem_par:nth-child(1)",
    "output": "description"
  }, {
    "type": "GET",
    "query": ".problem_limits",
    "output": "limits"
  }, {
    "type": "GET",
    "query": ".sample td:nth-of-type(1) pre",
    "output": "input"
  }, {
    "type": "GET",
    "query": ".sample td:nth-of-type(2) pre",
    "output": "output"
  }
]}]
```

18:6 ScraPE – An Automated Tool for Programming Exercises Scraping

After the refinement phase, it is possible to run the ScraPE main script in the command line. Figure 2 shows, in the left, the output of the script running in command line and, in the right, one of the exercises extracted in a ScraPE internal format. Currently, exercises are generated in the file system, but the plan is to automatically store them in a specific NoSQL database.



```
TERMINAL                                     exercises > {} exercise_0json > ...
F:\WORK\pupget>node index.js pupget2.json
https://acm.timus.ru/problemset.aspx?space=1&page=all
https://acm.timus.ru/problem.aspx?space=1&num=1000
https://acm.timus.ru/problem.aspx?space=1&num=1001
https://acm.timus.ru/problem.aspx?space=1&num=1002
https://acm.timus.ru/problem.aspx?space=1&num=1003
https://acm.timus.ru/problem.aspx?space=1&num=1004
https://acm.timus.ru/problem.aspx?space=1&num=1005
https://acm.timus.ru/problem.aspx?space=1&num=1006
https://acm.timus.ru/problem.aspx?space=1&num=1007
https://acm.timus.ru/problem.aspx?space=1&num=1008

1 {
2   "exercise": {
3     "title": "Simple Sum",
4     "limits": "1",
5     "description": "Read two integer values...",
6     "input": "30 10",
7     "output": "SOMA = 40"
8   }
9 }
```

■ Figure 2 ScraPE exercises generation.

5 Conclusion

This article introduces an automated scraping tool called ScraPE. The purpose of the tool is not to compete with existing scraping tools, but to feed a database as a cold-start facilitator for an ongoing project. This database will be used in conjunction with an API to serve, in a flexible way, learning systems (or individuals) to get programming exercises in the YAPEXIL format.

Currently, the tool is an ongoing work. In fact several parts of the process are not yet implemented such as: 1) the creation of a GUI editor to facilitate the script creation process; 2) the transformation of the ScraPE internal format to the YAPEXIL format and 3) the storage of the exercises in a database.

After solving these three simple tasks the goal is to create the programming exercises API and integrate the ScraPE tool in order to provide a simple and universal way to everyone get programming exercises with specific filters (for instance, “give me an easy exercise in JAVA with arrays”).

References

- 1 Jackie O’Kelly and J. Paul Gibson. Robocode & problem-based learning: A non-prescriptive approach to teaching programming. *SIGCSE Bull.*, 38(3):217–221, June 2006. doi:10.1145/1140123.1140182.
- 2 José Carlos Paiva, Ricardo Queirós, and José Paulo Leal. Mooshak’s Diet Update: Introducing YAPEXIL Format to Mooshak. In Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira, editors, *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, volume 94 of *Open Access Series in Informatics (OASICs)*, pages 9:1–9:7, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASICs.SLATE.2021.9.
- 3 José Carlos Paiva, Ricardo Queirós, José Paulo Leal, and Jakub Swacha. Yet Another Programming Exercises Interoperability Language (Short Paper). In Alberto Simões, Pedro Rangel Henriques, and Ricardo Queirós, editors, *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*, volume 83 of *OpenAccess Series in Informatics (OASICs)*, pages 14:1–14:8, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASICs.SLATE.2020.14.

- 4 Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003. doi:10.1076/csed.13.2.137.14200.
- 5 Vidhi Singrodia, Anirban Mitra, and Subrata Paul. A review on web scrapping and its applications. In *2019 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–6, 2019. doi:10.1109/ICCCI.2019.8821809.

Analysing Off-The-Shelf Options for Question Answering with Portuguese FAQs

Hugo Gonalo Oliveira ✉ 

CISUC, DEI, University of Coimbra, Portugal

Sara Inacio ✉

CISUC, DEI, University of Coimbra, Portugal

Catarina Silva ✉ 

CISUC, DEI, University of Coimbra, Portugal

Abstract

Following the current interest in developing automatic question answering systems, we analyse alternative approaches for finding suitable answers from a list of Frequently Asked Questions (FAQs), in Portuguese. These rely on different technologies, some more established and others more recent, and are all easily adaptable to new lists of FAQs, on new domains. We analyse the effort required for their configuration, the accuracy of their answers, and the time they take to get such answers. We conclude that traditional Information Retrieval (IR) can be a solution for smaller lists of FAQs, but approaches based on deep neural networks for sentence encoding are at least as reliable and less dependent on the number and complexity of the FAQs. We also contribute with a small dataset of Portuguese FAQs on the domain of telecommunications, which was used in our experiments.

2012 ACM Subject Classification Computing methodologies → Natural language processing

Keywords and phrases Natural Language Processing, Portuguese, Question Answering, FAQs, Information Retrieval, Sentence Encoding, Transformers

Digital Object Identifier 10.4230/OASICS.SLATE.2022.19

Supplementary Material *Software (Source Code)*: https://github.com/NLP-CISUC/PT_QA_Agents

Funding This work was funded by the project POWER (grant number POCI-01-0247-FEDER-070365), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020); and by national funds through FCT, within the scope of the project CISUC (UID/CEC/00326/2020) and by European Social Fund, through the Regional Operational Program Centro 2020.

1 Introduction

As a consequence to recent advances in Artificial Intelligence and, specifically, Natural Language Processing (NLP), there has been more and more interest in the development of systems that one may interact using natural language. These include artificial agents that emulate human-to-human conversations, some capable of providing customer-support and answering complex questions (QA).

When it comes to developing such systems, a popular option is to rely on so-called Natural Language Understanding (NLU) platforms, like Google Dialogflow¹ and Microsoft LUIS², which offer intuitive interfaces for designing dialogue flows, and managing intents, triggered actions and answers to give. However, the configuration and the maintenance of a complete working agent often comes with great manual effort, which tends to escalate with the size and diversity of the target domain, and has to be repeated for every new agent. Moreover,

¹ <https://cloud.google.com/dialogflow>

² <https://www.luis.ai/>



those platforms are generally limited to one or a small set of approaches for matching user request and retrieving answers. In most cases, it will be difficult, if possible, to integrate and test alternative approaches.

A cheaper option is to develop agents that get their answers from lists of Frequently Asked Questions (FAQs), especially when such lists are available *a priori*, in websites or other documentation. The problem is then the selection of a suitable approach for matching user requests with the available FAQs. If such an approach is not dependent on the style, on the domain and on the number of FAQs, creating a new agent becomes a matter of replacing the list of FAQs with a new one.

In order to gather more information on available options for developing a system of the previous kind for Portuguese, we explore and analyse different alternative approaches. We tried to cover a range of approaches, available off-the-shelf, and adaptable to any domain. Some are more traditional (e.g., Information Retrieval, IR), and others are based on more recent transformer neural networks, for sentence encoding (USE [23], BERT [4]), then used for computing semantic similarity, for extractive QA (BERT), and for text generation (GPT2 [16]).

To analyse how the selected approaches adapt to different domains, they were tested in two datasets of FAQs, i.e., question and answer pairs, in Portuguese. Following this experiment, we provide details on the configuration effort required for each approach, on the time required by each to get answers, and on their accuracy. The latter can be computed when question variations, available for both datasets, are used for simulating user requests. One of the datasets, AIA-BDE [7], has questions on a set of subdomains of public administration and was already available for this kind of experiments. The other is on the domain of telecommunications, was created in the scope of this work, and was made available for any interested researcher.

The main conclusions were that a traditional IR approach may be enough for a smaller list of FAQs. However, transformer models for sentence encoding adapt better to larger and more complex lists. This is especially true to the model based on the Universal Sentence Encoder (USE), developed specifically for answering FAQs. Based on all the analysed aspects, results also suggest that a transformer fine-tuned for QA does not suit our goal; and that generation is also not a good option, at least if it relies on fine-tuning the original GPT2, pre-trained mostly on English text. We hope that these insights can guide those willing to build a Portuguese QA agent in Portuguese on a specific domain.

In the remainder of this paper, we overview some related work on FAQ-oriented QA. We then describe the experimentation setup, covering the approaches and the data used. Before concluding, we present and discuss the results of our analysis.

2 **Related Work**

Automatic Question Answering (QA) is a NLP task with the goal of obtaining answers, often out of collections of documents [22], for questions posed in natural language. Traditional approaches were based on Information Retrieval (IR) [10], but more recent approaches rely on fine-tuning transformer neural networks. For instance, when fine-tuned on datasets like SQuAD [17], those models can extract answers from limited contexts [4]. When the answer can be spread across several documents, traditional IR, namely, the BM25 ranking method, can be used for reducing the search space [12].

The problem of getting answers from FAQs is not exactly the same, because the original questions are already formulated, together with their answer. Nevertheless, IR still makes sense in this context, namely for the computing the questions, out of those available, that are most similar to user requests. Once these are identified, their answer can be given.

Work on QA based on FAQs is not recent [2] and has been attempted for different languages [9, 15, 3]. For computing the similarity between user requests and available FAQs, traditional FAQ answering systems have exploited features such as word overlap [9], synonyms [11, 15], or distributional semantic features [9, 6].

Recently, transformers were also adopted for this task. BERT has been fine-tuned for computing the request-question [13] and the request-answer [19, 13] relevance. BM25 has been used as a baseline, but may also be useful for reducing the search space, by making an initial selection of potentially-related question-answer pairs.

For Portuguese, related work resulted in the development of Amaia [20], a FAQ-answering agent that relies on a Semantic Textual Similarity model (STS) for Portuguese, trained in the ASSIN collections [5, 18], which explores a broad range of lexical, syntactic and semantic features. Amaia gets its answers from the AIA-BDE corpus [7], where several additional IR approaches were tested, including BM25 (with the Whoosh library), and others based on static word embeddings or encoding with a pre-trained BERT, multilingual and Portuguese. Reported results show that the best model depends on the kind of variation, with interesting results for BM25, word2vec embeddings, and the Portuguese BERT. More information on AIA-BDE can be found in section 3.3.

More recently, transformer models for English and Portuguese (BERT, RoBERTa, Distillbert) were fine-tuned in the ASSIN 2 collection for encoding full sentences in Portuguese and enabling to answer FAQs [3]. They were tested in a dataset of 72 FAQs, on employee assistance in a telecommunications company, some paraphrased for testing purposes. To our knowledge, this dataset is not provided. The best performance (95% accuracy) was achieved by the models that had been pre-trained for Portuguese.

This work differs from the previous because: instead of single one, it experiments with two public datasets of FAQs in Portuguese and in two different domains; it includes some approaches that had not been tested before in this scenario; and it focuses on off-the-shelf approaches, which require minimal configuration, i.e., no additional training or complex fine-tuning.

3 Experimentation Setup

This work explores a set of approaches for Automatic Question Answering (QA) based in Portuguese FAQs, in any given domain, reflected in a list of FAQs provided. This section starts by introducing the models underlying the selected approaches, then detailing these approaches and, finally, describing the data used for assessing them.

3.1 Explored Models

Explored approaches are based on different technologies, which we present here, before explaining how they were applied for QA from FAQs.

Whoosh³ is a Python library for traditional IR which, by default, implements the BM25F probabilistic ranking function. It can be used for indexing collections of documents, according to a schema that sets available fields and their type, as well as required analysis (e.g., lower-case conversion, stemming). Given an index and a search query, Whoosh will retrieve relevant documents indexed.

³ <https://whoosh.readthedocs.io/>

The Universal Sentence Encoder (USE) [23] is a model for encoding text in high-dimension embeddings. It is currently based on a transformer and covers 16 different languages, including Portuguese⁴. USE-QA uses these embeddings for the task of QA. It encodes pairs of sentences and context and stores them in an index built with the `simpleneighbors`⁵ library. After this, given a request, USE-QA encodes it and queries the index, which returns an ordered list of approximate nearest neighbors in the semantic space.

BERT [4] is one of the most popular models based on transformers, which can be used for obtaining contextualised word or sentence embeddings, and can be further fine-tuned for different tasks like QA or natural language inference (NLI). BERTimbau [21] is a BERT model pre-trained for Portuguese which can be used for embedding words and longer sequences. This model has been fine-tuned for different tasks, including: NLI, on the ASSIN collections (hereafter, BERT-NLI); and extractive QA, in a Portuguese translation of the SQuAD 1.1 corpus (BERT-QA). The pre-trained version and the fine-tuned models are available off-the-shelf from the HuggingFace hub⁶, respectively as: *neuralmind/bert-large-portuguese-cased*, *pierregruillou/bert-base-cased-squad-v1.1-portuguese*, *ricardo-filho/bert-portuguese-cased-nli-assin-assin-2*.

GPT2 [16] is another model based on the transformer architecture. However, in opposition to the previous, which use encoder blocks, GPT2 uses only decoder blocks, and can be used more like a traditional language model, i.e., it generates text from given prompts. It can be fine-tuned with text of different styles and on different domains. Using the right prompts, it has been shown to perform several tasks unsupervisedly, which is also why we decided to explore it for our purpose. Its evolution, GPT3 [1] has ten times more parameters and was pre-trained in more text, in more languages. It is known for performing well in few-shot learning. However, access to GPT3 is controlled by an API⁷ and its free utilisation is limited. So, it was not used in this work.

3.2 Approaches for QA from FAQs

Different approaches were tested for QA based on FAQs, having in mind that they could be applied to any domain, regardless the availability of training data, and with the lowest possible effort. At the same time, we tried to cover approaches relying on different techniques, namely, those described in Section 3.1.

All approaches instantiate a “pipeline” that starts with a list of question-answer pairs (FAQs) and goes through two main stages: adaptation, where approach-specific preparations are performed (e.g., indexation or fine-tuning) once; and execution, where the approach waits for user queries and, upon receiving one, tries to obtain an answer, either by retrieving the most similar question or by generating the following text. We now describe how the selected models instantiate these stages.

The adaptation of Whoosh involves the indexation of the question-answer pairs. Each indexed document will have a field for the question and another for the answer. During execution, a question can be made to the index, and Whoosh will use its terms for retrieving the most similar question(s) and respective answer(s).

⁴ <https://tfhub.dev/google/universal-sentence-encoder-multilingual-qa/3>

⁵ <https://pypi.org/project/simpleneighbors/>

⁶ <https://huggingface.co>

⁷ <https://openai.com/api/>

As it happens with Whoosh, in the adaptation stage of USE-QA, question-answer pairs are indexed: questions are used as the sentences, and the full question-answer pair is used as the context. During execution, given a user query, USE-QA retrieves the most similar question indexed, together with its answer.

BERT models can be used with the help of the HuggingFace `transformers` library⁸, specifically with pipeline objects⁹. For instance, the *feature-extraction* pipeline extracts the hidden states of transformer for a given text, which can be used as its embeddings. It is used with the pre-trained version of BERTimbau (large), in order to get the [CLS] vector. From here, we refer to this approach as BERT-FE.

BERT-NLI, on the other hand, is a sentence transformer, and thus more suitable for encoding full sequences of text, besides having been trained for a task where representing the meaning of sentences is important (i.e., NLI). Although it could be used with the simple `transformers` library, the `sentence-transformers`¹⁰ library is more suitable for this model. It is loaded via the `SentenceTransformer` object, where embeddings can be obtained through the *encode* method.

In the adaptation stage of both BERT-FE and BERT-NLI, the models are loaded and the questions in the list of FAQs are encoded. In the execution stage, given user queries are encoded and the most similar questions, i.e., those maximising the cosine similarity, are computed and retrieved.

BERT-QA is also used with a pipeline object, this time of the *question-answering* type. This model expects a textual context where it will search for answers to given questions. Ideally, for our use case, the full list of FAQs would be used. However, there are limitations on the size of the context (i.e., maximum 512 tokens for BERT-base). So, to use it, the list of FAQs has to be first split into smaller subsets. To do this, and to keep the approach independent of the list of FAQs, its adaptation stage includes the additional step of clustering the FAQs with k-means, hoping to discover groups of related FAQs, when represented by BERT-FE embeddings. To select the number of clusters, we relied on the Elbow [8] method, which returned the optimal number in the 1–100 interval.

For the adaptation of GPT2, the model is fine-tuned with the lists of original questions and their answers. This was done with the `gpt-2-simple` library¹¹. We used GPT2 medium, which has 355M parameters, and set a temperature of 0.2, for avoiding highly random text. For a model like GPT2, fine-tuning does not require the definition of a task. The only requirement is to have data on the style of the text to generate, in this case, the list of FAQs and their answers. Expectations were that, when prompted with a question, GPT2 would generate a suitable answer from what it “had seen” during fine-tuning. Given the format of the lists of FAQs (see Section 3.3), for generation, we used a prefix that started with P:, followed by the user query and by R:. In addition, we define ‘\n\n’ as the truncation token because, before fine-tuning, we force that every question-answer is followed by an empty line.

3.3 Data

Since we also wanted to confirm that the selected approaches would work in different domains, we tested them in two different datasets with Portuguese FAQs and their variations, of different sizes and on different domains.

⁸ <https://huggingface.co/docs/transformers>

⁹ https://huggingface.co/docs/transformers/main_classes/pipelines

¹⁰ <https://huggingface.co/sentence-transformers>

¹¹ <https://github.com/minimaxir/gpt-2-simple>

19:6 Analysing Off-The-Shelf Options for Question Answering with Portuguese FAQs

The AIA-BDE corpus [7] contains 855 question-answer pairs on topics related to public administration. For each pair, variations were produced by different approaches and groups of people. Variations simulate user requests by paraphrasing the original questions, using different words, sometimes with missing information. QA approaches can be tested in the task of retrieving the original question, and the associated answer, given its variations. In AIA-BDE, questions are in lines starting with P:, their answers are in the following line that starts with R:, and variations are between them, in lines starting with VX:, where X depends on the kind of variation. See Figure 1 for an example of a question in AIA-BDE, followed by manually-created variations and the answer.

```
P:Como pedir o Cartão Provisório de Identificação de Pessoa Coletiva?  
VUC:Como posso obter o cartão provisório de identificação de pessoa coletiva?  
VUC:Onde posso pedir o cartão provisório de pessoa coletiva?  
VIN:Como posso pedir o Cartão provisório de identificação de pessoa coletiva?  
VIN:Qual o procedimento para obter o Cartão Provisório de Identificação de  
Pessoa Coletiva?  
R:O Cartão Provisório de Identificação de Pessoa Coletiva deixou de ser emitido, (...) Atualmente, existe apenas o Cartão da Empresa e o Cartão de Pessoa Coletiva, que são emitidos para entidades definitivamente registadas ou inscritas.
```

■ **Figure 1** Example of a FAQ in AIA-BDE, its variations and answer.

A second dataset (hereafter, Telecom) was produced specifically for this work. It is on a different domain, but its creation followed a similar approach to the creation of AIA-BDE. Telecom has 172 question-answer pairs gathered from various online sources, covering instruction manuals of telecommunication equipment and services by the Portuguese telecommunications operator MEO¹². Besides being on a different domain, the style of Telecom FAQs is significantly different than AIA-BDE. For instance, it includes several questions that are, in fact, stating issues (see Figure 2 for examples). For its creation, and similarly to AIA-BDE, at least two question variations were manually created, by two people, for 103 of the FAQs in the dataset. In this case, they were included in a different file (see Figure 3 for the variations of the questions in Figure 2).

```
P: Internet com quebras, com ligação por cabo  
R: Desligue todos os equipamentos da linha telefónica ou da rede sem fios. Depois, ...  
  
P: Sistema Operativo Mac OSX: Como ligar a uma rede sem fios  
R: Procure as redes sem fios Clique no ícone de WiFi e procure as redes sem fios ...  
  
P: Como definir um código pessoal de acesso ao Voice Mail?  
R: Se aceder ao Voice Mail a partir do seu próprio telefone, não é necessário colocar o código pessoal.
```

■ **Figure 2** Example of FAQs in the Telecom dataset.

4 Analysis

The selected approaches were configured for answering questions based on the FAQs in the Telecom and AIA-BDE datasets. This section analyses the configuration effort involved, the accuracy of the given answers, and the time taken for answering. The former and the latter are analysed subjectively, while the available question variations can be used for quantifying accuracy.

¹²<https://www.meo.pt/ajuda-e-suporte/produtos-meo/internet/equipamentos>

Ligação por cabo de internet com quebras
 Internet com falhas com ligação por cabo

Como fazer ligação a uma rede sem fios num Mac OSX?
 Como ligar a uma rede sem fios com o sistema operativo mac OSX?

Como posso definir um código de acesso ao voice mail?
 Como defino um código de acesso ao Voice Mail?

■ **Figure 3** Example of variations for the Telecom dataset.

4.1 Configuration Effort

Selected approaches are quite different and thus require a different configuration effort. This is probably the most subjective aspect analysed but, for those that are not familiar with any of the required technologies, it can be as important as the other two. Our analysis is based on a brief description of the steps required for having each approach ready to work, i.e., answering questions.

Thanks to the Huggingface `transformers` and `sentence-transformers` libraries, transformers are nowadays straightforward to use. In just a few lines, we can start using BERT and computing embeddings. After this, it is just a matter of computing cosines. Therefore, we would say that the BERT-FE and the BERT-NLI approaches are those that require less configuration effort. If it were not for the necessary clustering, BERT-QA would require a similar effort. However, this step increases the complexity of the configuration.

The effort of using USE-QA is also low. Some time was required for better understanding the input format for the list of FAQs. However, every step detailed in documentation¹³.

GPT2 could also be used with the `transformers` library, but the tested approach requires fine-tuning, which is not as straightforward as using an available model. However, in this case we used the `gpt-2-simple` library, which is also well-documented and makes it straightforward to fine-tune the original GPT2 models and use the result.

Using Whoosh requires more steps than the previous models, namely for defining the index schema, creating the index, searching and querying, but every step is also documented¹⁴. In the end, we can say that the required effort is comparable to using GPT2.

4.2 Answering Accuracy

Since both datasets include question variations, which can simulate user requests, each approach was assessed by the answers given for those variations. For this experiment, Tables 1 and 2 report on different measures. Accuracy is the proportion of variations for which the answer was the same as the one for the original question. This is computed for only the variations (Acc-vars)¹⁵ and also when the original questions are included (Acc-all). However, simply comparing the answers is unfair for GPT2, because it may generate answers that are not exactly the same as those expected, even if they might transmit a close meaning. Therefore, we include two additional metrics: BLEU [14], which measures the surface text similarity, and BERTScore [25], which, in an attempt to consider the meaning of text, relies on contextual embeddings, in this case, obtained from BERTimbau.

Figures show that overall accuracies are lower in the AIA-BDE corpus. This was expected because AIA-BDE is larger, meaning that there are many more possibilities when searching for similar questions. Another contribution to this may result from the variations of AIA-BDE,

¹³ https://www.tensorflow.org/hub/tutorials/retrieval_with_tf_hub_universal_encoder_qa

¹⁴ <https://whoosh.readthedocs.io/en/latest/quickstart.html>

¹⁵ For AIA-BDE, only the manually-produced variations were considered.

■ **Table 1** Performance in Telecom FAQs.

Approach	Acc-all	Acc-vars	BLEU	BERTScore
Whoosh	91.53%	88.35%	94.76	98.63
USE-QA	91.53%	88.29%	95.30	99.00
BERT-FE	66.40%	38.83%	79.58	97.38
BERT-NLI	74.60%	54.15%	84.16	97.58
BERT-QA	0.00%	0.00%	0.12	88.71
GPT2	0.00%	0.00%	22.11	90.12

■ **Table 2** Performance in AIA-BDE FAQs.

Approach	Acc-all	Acc-vars	BLEU	BERTScore
Whoosh	70.22%	65.59%	70.45	94.34
USE-QA	84.37%	79.35%	88.63	97.85
BERT-FE	75.05%	68.42%	81.04	96.48
BERT-NLI	79.73%	71.35%	84.44	97.09
BERT-QA	0.00%	0.00%	1.00	74.19

which, at the surface level, can be more different from the original questions. Yet, regardless of the dataset, top scores are always achieved by USE-QA, which seems to be the best option for our purpose.

In the Telecom dataset, the performance of USE-QA is matched by the best configuration of Whoosh. We only show the performance of this configuration, which is based on defaults plus a Portuguese Analyzer. Other configurations were tested (e.g., 3 and 4-grams), but differences were minimal. The performance of Whoosh suggests that, in some cases, traditional IR can be enough. However, we also see that its performance drops for AIA-BDE, a larger dataset and possibly more difficult.

BERT-NLI, fine-tuned for representing the meaning of sentences, performs better than the pre-trained BERT, which was somehow expected. In opposition to the other approaches, the performance of BERT-NLI increases in AIA-BDE, where it has the second best accuracy, confirming that the model representations go beyond surface text, and thus handle well variations using different words.

BERT-QA always extracts spans of text that have nothing to do with the question, which results in 0 accuracy and leads to the conclusion that it is not an option for this task. This is a consequence of both: (i) noise when selecting the cluster of FAQs to use as context; and, especially, (b) the format of the datasets (question-answer), which ends up being different from SQuAD (context-question-answer). While, in the future, we could experiment with a different clustering algorithm, a different numbers of clusters, and a different representation of FAQs, there is not much we can do about the latter, other than fine-tuning BERT on a dataset with a closer style.

We also confirm that GPT2 cannot generate answers that completely match the original ones¹⁶. Yet, even considering BLEU, it is way below the other approaches. GPT2 was fine-tuned in Portuguese data, but its starting point was the original GPT2 (medium), pre-trained mostly on English text, which might have had an impact here. In the future, we should try fine-tuning a GPT2 pre-trained for Portuguese¹⁷, or move on to recent open alternatives of GPT3 (e.g., Meta OPT [24]).

¹⁶Due to lack of memory, it was not possible to compute the figures for GPT2 in AIA-BDE

¹⁷<https://huggingface.co/pierreguillou/gpt2-small-portuguese>

Apart from enabling a comparison with GPT2, BLEU and BERTScore do not add much to our conclusions. BERTScore is always high, even for completely different texts, and should only be considered relatively. It might also be positively biased towards BERT-FE, which can be a consequence of using the same model for computing the scores.

4.3 Time

We analyse the time required for the adaptation and the execution stages. Our experiments were run in Google Colab¹⁸ without hardware acceleration, and average times were measured with the `time` library.

Adaptation, which only has to be done once for each run, includes embedding (BERT, USE-QA) and indexing (Whoosh) the questions, clustering (BERT-QA), or fine-tuning (GPT2). The latter is by far what takes more time in this stage, i.e., more than 1 hour for fine-tuning GPT2 with the Telecom dataset. For all the others, adaptation takes between 1 second (Whoosh) and 2 minutes (BERT-QA) in Telecom, and between 5 seconds (Whoosh) and 9 minutes (BERT-QA) in AIA-BDE.

Regarding the execution stage, we measure the average time taken between giving a question as input and getting its answer. Here, most approaches take less than 1 second in both datasets, with the exceptions being again GPT2 (40 seconds) and BERT-QA (30 to 90 seconds, depending on the number of clusters and size of the selected).

5 Conclusion

We have tested several approaches for automatic QA based in Portuguese FAQs, relying on different techniques, but all easily adapted to any domain. Following the results obtained in two datasets of FAQs and their variations, we can immediately discard two approaches, due to issues on accuracy and configuration effort, namely BERT-QA and GPT2. As for the others, USE-QA, based on the Universal Sentence Encoder, revealed to be the best option for answering FAQs. We should nevertheless highlight the performance of the BM25F traditional IR method, which performs well, especially in the smaller dataset. BERT-NLI is not a bad option either, but BERT would probably benefit from fine-tuning on data in the target style or domain. This was not considered for this work because we wanted to rely, as much as possible, on what was available off-the-shelf. However, it is something to try in the future, as well as further experiments with GPT2, GPT3 and comparable models.

Following the current interest in developing conversational agents and QA systems, often in domains for which FAQs are already available, the reported experiments may help those planning to develop or upgrade such a system. In the meantime, the code used for our experimentation, as well as the new Telecom dataset, are publicly available from a Github repository¹⁹.

References

- 1 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

¹⁸<https://colab.research.google.com/>

¹⁹https://github.com/NLP-CISUC/PT_QA_Agents.

- 2 Robin D Burke, Kristian J Hammond, Vladimir Kulyukin, Steven L Lytinen, Noriko Tomuro, and Scott Schoenberg. Question answering from frequently asked question files: Experiences with the FAQ finder system. *AI magazine*, 18(2):57–57, 1997.
- 3 Nuno Carriço and Paulo Quaresma. Sentence embeddings and sentence similarity for portuguese faqs. *Proceedings of IberSPEECH 2021*, pages 200–204, 2021.
- 4 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
- 5 Erick Fonseca, Leandro Santos, Marcelo Criscuolo, and Sandra Aluísio. Visão geral da avaliação de similaridade semântica e inferência textual. *Linguamática*, 8(2):3–13, 2016.
- 6 Erick R. Fonseca, Simone Magnolini, Anna Feltracco, Mohammed R. H. Qwaider, and Bernardo Magnini. Tweaking word embeddings for faq ranking. In *Proceedings of 5th Evaluation Campaign of Natural Language Processing and Speech Tools for Italian*, volume 1749. CEUR-WS, 2016.
- 7 Hugo Gonçalo Oliveira and Ana Alves. AIA-BDE: um corpo de perguntas, variações e outras anotações. *Linguamática*, 13(2):19–35, December 2021.
- 8 Kalpana D Joshi and PS Nalwade. Modified k-means for better initial cluster centres. *International Journal of Computer Science and Mobile Computing*, 2(7):219–223, 2013.
- 9 Mladen Karan, Lovro Žmak, and Jan Šnajder. Frequently asked questions retrieval for Croatian based on semantic textual similarity. In *Proceedings of the 4th Biennial International Workshop on Balto-Slavic Natural Language Processing*, pages 24–33, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.
- 10 Oleksandr Kolomiyets and Marie-Francine Moens. A Survey on Question Answering Technology from an Information Retrieval Perspective. *Information Sciences*, 181(24):5412–5434, December 2011.
- 11 Govind Kothari, Sumit Negi, Tanveer A. Faruque, Venkatesan T. Chakaravarthy, and L. Venkata Subramaniam. Sms based interface for faq retrieval. In *Proc Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2*, ACL '09, pages 852–860. ACL, 2009.
- 12 Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6086–6096, Florence, Italy, July 2019. ACL.
- 13 Yosi Mass, Boaz Carmeli, Haggai Roitman, and David Konopnicki. Unsupervised FAQ retrieval with question generation and BERT. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 807–812, 2020.
- 14 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- 15 Arianna Pipitone, Giuseppe Tirone, and Roberto Pirrone. ChiLab4It system in the QA4FAQ competition. In *Proceedings of 5th Evaluation Campaign of Natural Language Processing and Speech Tools for Italian*, volume 1749. CEUR-WS, 2016. URL: <http://ceur-ws.org/Vol-1749/>.
- 16 Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- 17 Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, 2016.
- 18 Livy Real, Erick Fonseca, and Hugo Gonçalo Oliveira. The ASSIN 2 shared task: a quick overview. In *Computational Processing of the Portuguese Language - 13th International Conference, PROPOR 2020, Évora, Portugal, March 2-4, 2020, Proceedings*, volume 12037 of *LNCS*, pages 406–412. Springer, 2020.

- 19 Wataru Sakata, Tomohide Shibata, Ribeka Tanaka, and Sadao Kurohashi. FAQ retrieval using query-question similarity and BERT-based query-answer relevance. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1113–1116, 2019.
- 20 José Santos, Luís Duarte, João Ferreira, Ana Alves, and Hugo Gonçalo Oliveira. Developing Amaia: A conversational agent for helping portuguese entrepreneurs — an extensive exploration of question-matching approaches for Portuguese. *Information*, 11(9), 2020.
- 21 Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. BERTimbau: pretrained BERT models for Brazilian Portuguese. In *9th Brazilian Conference on Intelligent Systems, BRACIS, Rio Grande do Sul, Brazil, October 20-23 (to appear)*, 2020.
- 22 Ellen M. Voorhees. The TREC-8 Question Answering track report. In *Proceedings of The Eighth Text REtrieval Conference, TREC 1999, Gaithersburg, Maryland, USA*. NIST, November 1999.
- 23 Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-hsuan Sung, Brian Strope, and Ray Kurzweil. Multilingual Universal Sentence Encoder for semantic retrieval. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 87–94. ACL, July 2020.
- 24 Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint*, 2022. [arXiv:2205.01068](https://arxiv.org/abs/2205.01068).
- 25 Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. BERTScore: Evaluating text generation with BERT. *arXiv preprint*, 2019. [arXiv:1904.09675](https://arxiv.org/abs/1904.09675).

