Report of Dagstuhl Seminar 15472

Programming with "Big Code"

Edited by

William W. Cohen[1], Charles Sutton[2], and Martin T. Vechev[3]

1 **Carnegie Mellon University, US,** `wcohen@cs.cmu.edu`
2 **University of Edinburgh, GB,** `csutton@inf.ed.ac.uk`
3 **ETH Zürich, CH,** `martin.vechev@inf.ethz.ch`

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 15472 *Programming with "Big Code"*. "Big Code" is a term used to refer to the increasing availability of the millions of programs found in open source repositories such as GitHub, BitBucket, and others.

With this availability, an opportunity appears in developing new kinds of statistical programming tools that learn and leverage the effort that went into building, debugging and testing the programs in "Big Code" in order to solve various important and interesting programming challenges.

Developing such statistical tools however requires deep expertise across multiple areas of computer science including machine learning, natural language processing, programming languages and software engineering. Because of its highly inter-disciplinary nature, the seminar involved top experts from these fields who have worked on or are interested in the area.

The seminar was successful in familiarizing the participants with recent developments in the area, bringing new understanding to different communities and outlining future research directions.

Seminar November 15–18, 2015 – http://www.dagstuhl.de/15472
1998 ACM Subject Classification D.2.3 [Software Engineering] Coding Tools and Techniques, I.2.2 [Artificial Intelligence] Automatic Programming, I.2.5 [Artificial Intelligence] Programming Languages and Software
Keywords and phrases machine learning, natural language processing, programming languages, software engineering, statistical programming tools
Digital Object Identifier 10.4230/DagRep.5.11.90

1 Executive Summary

Martin T. Vechev
William W. Cohen
Charles Sutton

License Creative Commons BY 3.0 Unported license
© Martin T. Vechev, William W. Cohen, and Charles Sutton

The main objective of the seminar was to bring together several research communities which have so far been working separately on the emerging topic of "Big Code" and to foster a new community around the topic. Over the last 4–5 years there have been several developments and interesting results involving "Big Code" all spanning a wide range of fields and conferences: the seminar brought these communities together and enabled them to interact for the first time.


Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license
Programming with "Big Code", *Dagstuhl Reports*, Vol. 5, Issue 11, pp. 90–102
Editors: William W. Cohen, Charles Sutton, and Martin T. Vechev

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

The program was structured as a series of talks interspersed with discussion. Almost all of seminar participants gave a talk on their latest research. Even though the initial plan was to include special discussion sessions, each talk triggered so much discussion, both during the talk itself, and also after, that there was no need for specific discussion slots. We believe the seminar was successful in setting the right atmosphere for open ended discussion and obtained the desired affect of triggering much organic interaction.

Only the last day (morning) included a short wrap-up discussion session focusing on the future of the area, defining common data sets and future challenges the community can address. That discussion is summarized in the working group report.

The seminar was highly inter-disciplinary involving experts from programming languages, software engineering, machine learning and natural language processing. Further, it brought together research groups from Europe, Asia and U.S., all working on the topic of "Big Code", and raised awareness and familiarity with what different research groups are working on.

The talks and discussions spanned several topics including: the kinds of statistical methods used (e.g., n-gram models, recurrent neural networks, graphical models, probabilistic grammars, etc), new programming applications that can benefit from these models (e.g., code completion, code search, code similarity, translating natural language to code, etc), and the interaction between these. Some of the presentations were more of an introductory/overview nature while others focused on the more technical aspects of particular programming tools and machine learning models.

After two days of presentations and discussions, we used the last day of the seminar (before lunch) to summarize the discussions and to outline a future research direction. A suggestion enthusiastically embraced by everyone was to create a web site which lists the current data sets, challenges, tools and research groups working on the topic. The view was that this will not only enable existing groups to compare their tools on common problems and data sets but will also make it much easier for other research groups and graduate students to get into the area and to start contributing. It also serves as a useful instrument for raising awareness about the topic:

We have now created this web site and have made it available here: http://learnbigcode.github.io/.

In a short time, several groups have started contributing by uploading links to tools, data sets and challenges.

Overall, the seminar was successful both in terms of stimulating new and fruitful interaction between research communities that were working in the area but were separated so far, but also in setting a common agenda moving forward. Due to the high interest and feedback from this seminar, we anticipate that in a year or two from now, we will be ready to propose a larger seminar on the topic.

2 Table of Contents

3 Overview of Talks

3.1 Miltos Allamanis

Miltos Allamanis (University of Edinburgh, GB) and Charles Sutton (University of Edinburgh, GB)

License Creative Commons BY 3.0 Unported license
© Miltos Allamanis and Charles Sutton

We briefly discuss recent work on mining code idioms from codebases. A code idiom is a syntactic code fragment that recurs frequently across software projects and has a single semantic purpose. Although, we know that developers write idiomatic code it is not clear why idioms arise in source code. In this talk, I discuss potential reasons for the prevalence of idiomatic code among developers.

3.2 Loop-Invariant Synthesis using Techniques from Constraint Programming

Jason Breck (University of Wisconsin – Madison, US)

License Creative Commons BY 3.0 Unported license
© Jason Breck

In this talk, I describe a loop invariant synthesis technique inspired by constraint programming. In particular, the technique uses an abstract domain from constraint programming: the abstract domain consists of sets of boxes, where a box is a collection of interval constraints, one for each program variable. The technique synthesizes inductive loop invariants for programs that manipulate real-valued variables. It works by iteratively splitting and deleting boxes until the set of boxes becomes an inductive loop invariant, or a failure condition is reached. I describe an extension to the technique that uses an abstract domain of octagons instead of boxes. I also describe a series of experiments that test our technique on programs taken from the literature on numeric loop invariant synthesis.

3.3 An overview of the Pliny project

Swarat Chaudhuri (Rice University – Houston, US) and Christopher M. Jermaine (Rice University – Houston, US)

License Creative Commons BY 3.0 Unported license
© Swarat Chaudhuri and Christopher M. Jermaine

Formal methods is the science of mechanized formal reasoning about complex systems. Data mining is the science of extracting knowledge and insights from large volumes of data. In this talk, I will describe Pliny, a Rice-led DARPA project that seeks to bridge these two disciplines. The vision of Pliny is to develop a wide range of formal reasoning tools that aim to make software more reliable, faster, and more easily programmed. The difference between Pliny and existing formal methods approaches is that Pliny complements automated logic-based analysis with statistical mining of "Big Code", i.e., large corpora of open-source software. The logical and statistical techniques are unified under a Bayesian framework where logical techniques are guided by data-driven insights and data mining happens on artifacts generated through automated reasoning.

3.4 Studying the Naturalness of Software

Premkumar T. Devanbu (University of California – Davis, US)

License Creative Commons BY 3.0 Unported license
© Premkumar T. Devanbu

Natural languages have evolved to serve immediate, natural human purposes: survival, nourishment, reproduction; while languages per se are rich in vocabulary and grammatical flexibility, most human utterances are simple and repetitive, reflecting the origins of the medium. At UC Davis, we discovered in 2011 that large software corpora show even greater repetitive nature, despite the considerable power and flexibility of programming languages. Since studies since then, we have studied this repetitive structure in detail, and shown that the phenomenon persists (and indeed strengthens) even if differences between programming language corpora and natural language corpora are accounted for. Thus, although programming languages have much greater vocabulary (arising from variable names) the vocabulary, when names are split, show an even greater degree of repetition. Likewise, the simplicity of programming code is not just an artifact of simpler structure: when compared on an equal basis (programming corpora without keywords and operators, language corpora without function words) software in fact becomes even more repetitive. IN ongoing work, we are finding that when software code is non-repetitive (or surprising), it is in fact much likelier to be defective.

3.5 Doing Software Analytics Research – Incorporating Cross-Domain Expertise

Shi Han (Microsoft Research – Beijing, CN)

License Creative Commons BY 3.0 Unported license
© Shi Han

This talk introduces four selected research projects in the past six years at the Software Analytics group of Microsoft Research, demonstrating te importance and challenges of incorporating cross-domain expertise for successful learning/mining tasks against software artifects such as code or log.

- StackMine – performance debugging in the large via mining millions of stack traces
- DriverMine – comprehending OS performance issues in the software ecosystem scope
- JSweeter – uncovering JavaScript performance code smells relevant to type mutation
- Codeology – program understanding via mining big code

3.6 Can big code find errors?

Abram Hindle (University of Alberta – Edmonton, CA)

License Creative Commons BY 3.0 Unported license
© Abram Hindle

Syntax errors, misspellings, and misunderstandings are detriment to programmers everywhere. But naturalness is here help, by treating software source code as natural language utterances we can leverage tools used in the NLP domain to help debug software. Specifically in this

work we demonstrate that smoothed n-gram models of source code tokens trained on a large corpus of code can easily determine if code doesn't belong: code that doesn't belong, code that is surprising to a model is usually code that contains syntax errors, misspellings, or awkward semantics not usually employed. We evaluate this conjecture on code from 2 fundamentally different programming languages: Java and Python. We find that in both cases asking a model trained on good source code, "does this potentially bad source code surprise you?" allows us to identify syntax errors, misnamed identifiers, and even missing code tokens. We find that in the case of Python, a dynamic language, that the lack of oracle is a huge impediment, such that programmers may ship python code that seems to work but actually contains syntax errors.

3.7 Mining and Understanding Software Enclaves

Suresh Jagannathan (Purdue University – West Lafayette, US)

License Creative Commons BY 3.0 Unported license
© Suresh Jagannathan

The modern-day software ecosystem is a messy and chaotic one. Among other things, it includes an intricate stack of sophisticated services and components, susceptible to frequent (and often incompatible) upgrades and patches; emerging applications that operate over large, unstructured, and noisy data; and, an ever-growing code base replete with latent defects and redundancies. Devising novel techniques to tame this complexity, and improve software resilience, trustworthiness, and expressivity in the process, is a common theme actively being explored by several ongoing DARPA programs. This talk gives an overview of one such effort – MUSE (Mining and Understanding Software Enclaves) –, which aims to exploit predictive analytics over large software corpora to automatically repair and synthesize programs. It seeks to realize this vision of "Big Code" by developing foundational advances in programming language design, analysis, and implementation, and has as its overarching goal, revolutionizing the way we think about software construction and reliability.

3.8 Learning to Search Large Code Bases

Christopher M. Jermaine (Rice University – Houston, US)

License Creative Commons BY 3.0 Unported license
© Christopher M. Jermaine

In the Pliny project, our goal is to use a large code repository to help perform tasks such as synthesis, bug finding, and repair. One of the key subtasks for each of these tasks is searching a large database for all codes that are "close to" a query specification. Unfortunately, "close to" is not defined a priori. We can extract features from code (including syntactic and semantic information about the code) and we can define distance metrics over those features, but we do not know beforehand which are going to be most useful for solving a given task. In this talk, I will describe how we plan to use the question-answer posts from Stack Overflow to help bootstrap the process of figuring out what features and metrics are potentially useful to power search. We describe a model that is similar to Canonical Correlation Analysis, and automatically chooses features and metrics that are useful for linking code to natural language text. Our hypothesis is that those same features and metrics will be useful for finding codes that are close to a query in a synthesis, debugging, or repair task.

3.9 Big Code for Better Code

Sebastian Proksch (TU Darmstadt, DE)

License Creative Commons BY 3.0 Unported license
© Sebastian Proksch

The goal of this talk was to stimulate discussion. Because of a very restricted time slot for the presentation, the talk was very focussed on two points:

1. It introduced an extensible inference engine for recommender systems in software engineering that is based on feature vectors generated from structural context. Researchers can freely exchange the underlying pattern detection mechanisms and get a recommender and the necessary evaluation pipeline for free.
2. State of the art evaluation of recommender systems in software engineering are typically based on artificial evaluations. The talk stresses the fact that this does not provide any insights about the perceived usefulness of a tool by the developer. To improve evaluations, we collected real developer feedback by instrumenting the Visual Studio IDE and use this as a ground truth for evaluation instead.

3.10 Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes

Christopher Quirk (Microsoft Corporation – Redmond, US)

License Creative Commons BY 3.0 Unported license
© Christopher Quirk

Using natural language to write programs is a touchstone problem for computational linguistics. We present an approach that learns to map natural-language descriptions of simple "if-then" rules to executable code. By training and testing on a large corpus of naturally-occurring programs (called "recipes") and their natural language descriptions, we demonstrate the ability to effectively map language to code. We compare a number of semantic parsing approaches on the highly noisy training data collected from ordinary users, and find that loosely synchronous systems perform best.

3.11 Learning Programs from Noisy Data

Veselin Raychev (ETH Zürich, CH) and Martin T. Vechev (ETH Zürich, CH)

License Creative Commons BY 3.0 Unported license
© Veselin Raychev and Martin T. Vechev

We present a novel technique for constructing statistical code completion systems. These are systems trained on massive datasets of open source programs, also known as "Big Code". The key idea is to introduce a domain specific language (DSL) over trees and to learn functions in that DSL directly from the dataset. These learned functions then condition the predictions made by the system. This is a flexible and powerful technique which generalizes several existing works as we no longer need to decide a priori on what the prediction should be conditioned. As a result, our code completion system surpasses the prediction capabilities of existing, hard-wired systems.

3.12 Clio:Digital Code Assistant for the Big Code Era

Armando Solar-Lezama (MIT – Cambridge, US)

License Creative Commons BY 3.0 Unported license
© Armando Solar-Lezama

The talk describes our efforts in leveraging information from big code in order to support synthesis tasks. The first part of the talk first provides a brief overview of our DemoMatch effort which allows user to demonstrate the use of a framework on an existing application and then generates the code necessary to use the framework in that way. The second part of the talk describes a tool called Swapper that automatically generates formula simplification routines to be used inside solvers. Swapper takes as input a corpus of formulas from synthesis problems and produces simplifiers tailored to those formulas. Automatic generation of such simplifiers is a first step towards automatic generation of domain specialized solvers.

3.13 Statistical Analysis of Program Text

Charles Sutton (University of Edinburgh, GB)

License Creative Commons BY 3.0 Unported license
© Charles Sutton

Billions of lines of source code have been written, many of which are freely available on the Internet. This code contains a wealth of implicit knowledge about how to write software that is easy to read, avoids common bugs, and uses popular libraries effectively.

We want to extract this implicit knowledge by analyzing source code text. To do this, we employ the same tools from machine learning and natural language processing that have been applied successfully to natural language text. After all, source code is also a means of human communication.

We present three new software engineering tools inspired by this insight, that learn local coding conventions (naming and formatting), syntactic idioms in code, and API patterns.

3.14 Graph-structured Neural Networks for Program Verification

Daniel Tarlow (Microsoft Research UK – Cambridge, GB)

License Creative Commons BY 3.0 Unported license
© Daniel Tarlow

An open problem in program verification is to verify properties of computer programs that manipulate memory on the heap. A key challenge is to find loop invariants – formal descriptions of the data structures that are instantiated – which are used as input to a proof procedure that verifies the program. We describe a machine learning-based approach, where we execute the program and then learn to map the state of heap memory (represented as a labelled directed graph) to a logical description of the instantiated data structures. In the process of working on this problem, we developed a new general purpose neural network architecture that is suitable for learning mappings from graph-structured inputs and sequential outputs. We describe this model and speculate that it could be generally useful for a range of problems that arise in the space of "big code".

3.15 Learning from Big Code

Martin T. Vechev (ETH Zürich, CH)

License Creative Commons BY 3.0 Unported license
© Martin T. Vechev

An overview presentation covering the recent research advancements on the topic of "Big Code" at ETH Zürich. The presentation covers several probabilistic models and how they are used (e.g., recurrent networks, CRFs, etc), for instance, JSNice. The talk also discusses various open questions and challenges that the community can explore further.

3.16 Estimating Types in Binaries using Predictive Modeling

Eran Yahav (Technion – Haifa, IL)

License Creative Commons BY 3.0 Unported license
© Eran Yahav

Reverse engineering is an important tool in mitigating vulnerabilities in binaries. As a lot of software is developed in object-oriented languages, reverse engineering of object-oriented code is of critical importance. One of the major hurdles in reverse engineering binaries compiled from object-oriented code is the use of dynamic dispatch. In the absence of debug information, any dynamic dispatch may seem to jump to many possible targets, posing a significant challenge to a reverse engineer trying to track the program flow.

We present a novel technique that allows us to statically determine the likely targets of virtual function calls. Our technique uses object tracelets – statically constructed sequences of operations performed on an object – to capture potential runtime behaviors of the object. Our analysis automatically pre-labels some of the object tracelets by relying on instances where the type of an object is known. The resulting type-labeled tracelets are then used to train a statistical language model (SLM) for each type. We then use the resulting ensemble of SLMs over unlabeled tracelets to generate a ranking of their most likely types, from which we deduce the likely targets of dynamic dispatches. We have implemented our technique and evaluated it over real-world C++ binaries. Our evaluation shows that when there are multiple alternative targets, our approach can drastically reduce the number of targets that have to be considered by a reverse engineer.

3.17 App Mining

Andreas Zeller (Universität des Saarlandes, DE)

License Creative Commons BY 3.0 Unported license
© Andreas Zeller

How do we know what makes behavior correct? In the absence of detailed specifications, one alternative could be to analyze large bodies of existing software to determine which behaviors are common and thus normal. We have mined thousands of popular Android apps from the Google Play store to determine their normal behavior with respect to their API usage and their information flows. After clustering apps by their description topics, we identify outliers in each cluster with respect to their behavior. A "weather" app that sends messages

thus becomes an anomaly; likewise, a “messaging” app would not be expected to access user account data. The approach is very effective in identifying novel malware even if no malware samples are given, and Google is currently adopting the approach for its store. In the long run, we expect mined patterns of normal behavior to well complement explicit specifications.

3.18 A User-Guided Approach to Program Analysis

Xin Zhang (Georgia Institute of Technology – Atlanta, US)

License Creative Commons BY 3.0 Unported license
© Xin Zhang

Program analysis tools often produce undesirable output due to various approximations. We present an approach and a system Eugene that allows user feedback to guide such approximations towards producing the desired output. We formulate the problem of user-guided program analysis in terms of solving a combination of hard rules and soft rules: hard rules capture soundness while soft rules capture degrees of approximations and preferences of users. Our technique solves the rules using an off-the-shelf solver in a manner that is sound (satisfies all hard rules), optimal (maximally satisfies soft rules), and scales to real-world analyses and programs. We evaluate Eugene on two different analyses with labeled output on a suite of seven Java programs of size 131–198 KLOC. We also report upon a user study involving nine users who employ Eugene to guide an information-flow analysis on three Java micro-benchmarks. In our experiments, Eugene significantly reduces misclassified reports upon providing limited amounts of feedback

4 Working groups

4.1 Discussion on how to advance the research along this direction and form a community around the topic

Martin T. Vechev (ETH Zürich, CH) and Veselin Raychev (ETH Zürich, CH)

License Creative Commons BY 3.0 Unported license
© Martin T. Vechev and Veselin Raychev

On the last day of the seminar, we dedicated an hour long discussion slot. We discussed several of these questions:

Which tasks should we solve? It was observed that no two groups were solving the exact same problem. To advance the area, the goal is to define a common set of tasks that we believe are important as a community. One comment was the danger of focusing on a single task may be limiting the research. The topic of task diversity came about and that there is no need to try and be very diverse initially. An example of a possible task is: deobfuscation, as there is also a baseline here.

What metrics should we use? Machine learning metrics sometimes make no sense here. End user metrics do matter though. A point was made about algorithmic overfitting: the Siemens benchmark suite for bug localization. Plenty of works that fine-tuned details for these benchmarks, but irrelevant in general. NLP also has experience in overfitting to datasets. A good example of a metric that advanced the area is machine translation with

the BLEU score (before it, all papers did user studies). Ideally, new technique papers in the area should not have to include user studies.

What is a good venue to publish the work? A comment was made that dataset papers should appear, e.g., in EMNLP there is a notable dataset award in addition to a Best paper award. Many good conferences are already accepting works in the area.

Are there expensive to obtain datasets? A question was whether we can agree on a manually annotated dataset? Which datasets are the ones that are costly and we can share the cost.

What will be important for actual programming tools? A comment was that we may actually want to overfit if we solve the actual task requested by the user. Another comment was that the particular datasets are good for our tasks, but useless for programming language tasks. Most programs do not compile. Tasks should be evaluated on how much they save for the user. Refactoring is called rarely, code completion all the time, but how much time is spent/saved for the user?

What are good outcomes of the seminar? An idea was to create a web site where everyone can upload their data set and post a challenge. It was suggested to go with Github and pull requests as opposed to a pure Wiki. This site is now a reality and several groups have already started uploading data sets and challenges:

http://learnbigcode.github.io/

Participants

- Miltos Allamanis
University of Edinburgh, GB
- Earl Barr
University College London, GB
- Jason Breck
University of Wisconsin – Madison, US
- Swarat Chaudhuri
Rice University – Houston, US
- William W. Cohen
Carnegie Mellon University, US
- Premkumar T. Devanbu
Univ. of California – Davis, US
- Shi Han
Microsoft Research – Beijing, CN
- Kenneth Heafield
University of Edinburgh, GB
- Abram Hindle
University of Alberta – Edmonton, CA
- Suresh Jagannathan
Purdue University – West Lafayette, US
- Christopher M. Jermaine
Rice University – Houston, US
- Dongsun Kim
University of Luxembourg, LU
- Dana Movshovitz-Attias
Google Inc. – Mountain View, US
- Tien N. Nguyen
Iowa State University, US
- Sebastian Proksch
TU Darmstadt, DE
- Christopher Quirk
Microsoft Corporation – Redmond, US
- Veselin Raychev
ETH Zürich, CH
- Armando Solar-Lezama
MIT – Cambridge, US
- Charles Sutton
University of Edinburgh, GB
- Daniel Tarlow
Microsoft Research UK – Cambridge, GB
- Martin T. Vechev
ETH Zürich, CH
- Nicolas Voirol
EPFL – Lausanne, CH
- Eran Yahav
Technion – Haifa, IL
- Andreas Zeller
Universität des Saarlandes, DE
- Xin Zhang
Georgia Institute of Technology – Atlanta, US