NL+SE Workshop Final Report

Prem Devanbu, Dana Movshovitz-Attias, and Chris Quirk
April 21, 2016

Introduction

A workshop on the application of Statistical NLP Methods to Software Artifacts was held at Microsoft Research during Oct 25, 26, and 27th. The goals of the workshop were to explore current research and future directions on the following topics:

1. "Big Code" data, such as Gigatoken Code Corpora (with change history, bug reports, Q&A, comments) are now available. The statistics in these corpora resemble those of NL Corpora. How can this be exploited?

2. How can NLP approaches aimed at generating formal representations of natural language be exploited in software?

The expected outcomes of the workshop were a report summarizing the discussions, and a "wish list" of things to help impulse research in the area, including datasets, competitions, collaborations, etc.

While there has been considerable interest in applying NLP methods to software in the past, to our knowledge, this was the first time that a significant, large number of NLP researchers with a strong statistical focus had a chance to interact closely with software engineering researchers, and hear first-hand the problems that are faced in this area.

This workshop was jointly sponsored by Microsoft Research and the U.S. National Science Foundations. There were 43 attendees, including 6 students, and 8 international participants.

Warm-up: Tutorials

This was a "first contact" workshop. For many of the Software Engineering and NLP researchers, this was the first time that they directly interacted with researchers from the other discipline. To facilitate the interaction, we arranged two broad, introductory tutorials. One tutorial covered methods in statistical language processing and language modeling by Dr. Ashish Vaswani from USC/ISI. The other, on Software data and software mining, was presented by Prof. Tao Xie from University of Illinois. The

tutorials were very well attended. Slides from both Tao Xie's and Ashish Vaswani's lectures are available. Videos (Vaswani, Xie) are also available.

Software Tools and Processes

(Organizers: Premkumar Devanbu & Chris Quirk)
(Scribe: Jennifer D'Souza)

This early session was aimed at setting the theme for the workshop; it began with a keynote presentation by Prof. Charles Sutton, a leading researcher in this area. Slides and video for this presentation are kindly made available by Prof. Sutton, under the auspices of Microsoft Research. The presentation covered the beginnings of research in the area, as well as a round-up of the current research, and presented a vision for future work. This was followed by a discussion, which was scribed by Dr Jennifer D'Souza. There were several main topics touched on in the discussion.

Talk Summary

Prof Sutton's tak introduced the area with the claim that "Source code is a means of human communication": code is an intentional act of one developer to communicate with another, about design, rationale, usability etc. With this "speech act", a developer aims to make their code more maintainable, reusable etc. Certainly, developer-to-developer communication occurs over a variety of online media, including version control systems, email, chat-groups, on-line fora, social coding websites, and such.

The talk then listed some analogies between a wide range of NLP-related tasks and corresponding software engineering talks. A full list is best viewed in the video, but representative analogies include NLP translation to Code porting, and spelling and grammar correction to code patching.

The middle part of the talk surveyed a range of current research achievements in the application of NLP methods to code, including research at Edinburgh (Sutton, Allamanis), UCL (Earl Barr), UC Davis (Devanbu), ETH (Vechev) and Alberta (Hindle).

The talk concluded with a call for a more empirical, data-driven perspective on software engineering problems: rather than exclusively focusing on formal abstraction as a means of retreating from the "undecidability bottleneck" of deriving sound & complete program properties, Sutton suggests a "statistical retreat" based on models estimated from large corpora, that use inductive bias to capture the typical, "natural" patterns latent in large corpora. He also suggested another view of "naturalness": that code in which latent, deep, semantic properties are evident from surface-level

properties are easier for humans to read and understand, and perhaps also are more amenable to hybrid statistical-formal methods analysis.

Following the keynote, there was a wide-ranging discussion on the general area, which we group into topics as follows.

Problems Yet to be Explored

The discussion began with a call for interesting future topics for study. A number of topics arose.

1. **Surpassing Humans.** Tao Xie speculated whether (statistical) AI methods could surpass human performance, specifically, in the area of constructing revealing test cases that could expose defects.

2. **Program Repair/Fault Localization.** Abram Hindle mentioned that language models can locate the precise source of syntax errors in programs, which compilers can often find difficult to diagnose. There was also discussion about how language models might promote the discovery of more "natural" repairs to errors, both syntactic and otherwise.

3. **The "Essence" of Programs.** Zhendong Su asked if statistical models be used to identify the "essence" of programs, by enabling the culling of those parts which are repetitive, formulaic, or redundant (such as variable names). The relevance of the "*sloppy programming*" project was discussed, where the scaffolding of programs could be automatically generated around some human-supplied essentials.

4. **Specification-conformant programs.** Graham Neubig wondered whether a program generated from a natural language specification could somehow be checked against the natural language specification for correctness, perhaps by also generating a formal specification from the natural language description.

The Creation of Datasets

There was considerable interest in the challenges and availability of datasets. There were two main topics that were discussed: *parallel code/NL datasets,* and the *use of Mechanical Turk* to create data.

Parallel, aligned corpora of Code & Natural Language. There was strong interest in the possible availability of aligned code/natural language corpora, where *"the code*

does exactly what the natural language says" (Graham Neubig). Several possible sources of such data were suggested:

1. RosettaCode. A "chrestomathy" of code, in different languages, accompanied by english descriptions.
2. JPetStore. A particularly well-documented and commented 3-tier system.
3. iPython notebooks. These are a very popular platform for literate programming, where code is accompanied by descriptions of its function, and sample inputs and outputs.

Mechanical Turk for dataset creation. While the utility of some these corpora was acknowledged, there was interest in a broader set of more closely aligned sources. This gave rise to a discussion on creating/curating datasets with manual effort; thus Mechanical Turk arose as a natural option, and considerable discussion ensued.

1. Several researchers (Dawn Lawrie, Collin McMillan, Nate Kushman) described difficulties with using Turkers, specially in finding people qualified for code-related tasks.
2. Kushman noted that the direction of the task (code->English vs. English->Code) matters. The summarizing task (Code->English) gave rise to unnatural annotations, whereas the reverse (Code from english descriptions) proved quite amenable to this setting; they had good luck with the ODesk Platform in this setting, relative to Amazon's mTurk. Movshovitz-Attias also reported good experience with using mTurk for relation labeling
3. Luke Zettlemoyer noted that task-definition when involving humans to produce a supervisory dataset is critical: *e.g.,* for the code->English task, the word-count budget given to human Turker would be critical.
4. The question of qualifying Turkers was viewed as critical, to avoid spamming; Chris Quirk speculated that educational settings might be a better way for low-cost supervisory data creation. Nate Kushman observed that providing enhanced incentives might motivate people to the aquire a higher-level of skill required for high-fidelity task completion.

Software Dataset Challenges. Danny Tarlow noted the challenges non-experts in software engineering face when accessing richer forms of representation of code. He mentioned that if one knew how to query static analysis tools, compilers and such other tools that could produce different representations of software, the resulting data could be used to facilitate machine learning and the building of machine learning models. However, he mentioned that the know-how around using these tools was one area of entry that kept non-experts in software engineering from attacking software engineering problems and from using more sophisticated methods than just token-level information. So he said that along with releasing the source code, it might

also be useful to consider releasing other kinds of information along with it. It was observed that the BOA data server might help with these challenges.

The Challenges and Value of Competitions.

The criticality of competitions (a well-curated dataset, a well-defined task, and a challenging baseline performance criterion) to rapid advances was noted by the NLP colleagues; the relative unpopularity of such competitions in software engineering was recognized by the SE colleagues. Chris Quirk noted that substantial benefits could be gained with introducing competitions in software engineering. For instance, researchers could gain access to a large language modeling dataset and all systems could be compared on a benchmark evaluation such as perplexity. Potentially, the best model for predicting the naturalness of Java code could result from an organized competition. Tao Xie noted that the task should be considered practically relevant by software engineers. A few categories of tasks were discussed.

1. *Code cloning:* a few datasets already exist; although no "golden" set is available, precision could be estimated by sampling. (Abram Hindle)
2. *Code completion:* arbitrarily many benchmarks could be created by sampling from existing code bases, pull requests etc. (Hindle)
3. *Traceability:* There was considerable discussion on this topic, concerning the difficulties of creating datasets at scale. Cleland-Huang's TraceLab is a noteworthy example of such a traceability dataset.
4. *Bug Localization:* This is the task of localizing defects given passing and failing test sets.

Summary

The need for datasets, benchmarks and competitions is seen as vital to the vibrancy of this community; there is considerable incentive for creating benchmarks and competition datasets, since these will have a great deal of impact.

Data repositories (Boa briefing)

(Scribe: Zhilin Yang zhiliny@cs.cmu.edu)
(Organizers: Premkumar Devanbu & Chris Quirk)
(Presenter: Dr. Tien N. Nguyen, Iowa State University)

A general introduction to Boa was provided, slides and video are available. Mining software repositories (MSR) at a large-scale is important for more generalizable research results. Therefore, a number of recent studies in the MSR area have been conducted using corpus sizes that are much larger compared to the corpus size used by studies in the previous decade. Such a large collection of software artifacts is openly available for analysis, e.g., SourceForge has 350k+ projects, GitHub has 10M+ projects, and Google Code has 250K+ projects. This is an enormous collection of software and software-related metadata.

Using this vast amount of information to conduct MSR studies can be challenging. Specifically, large scale MSR studies (*e.g.*, finding instances of bugs and bug fixes at scale) requires expertise in programmatic APIs for version control systems, database management, data mining, and parallelization. These four requirements significantly increase the cost of scientific research in this area. Moreover, building analysis infrastructure to process such ultra-large-scale data efficiently can be very difficult. Efficiency is another issue. Due to the large amount of available software repositories, it is nontrivial to set up parallel architecture for data processing and repository mining.

A domain specific language and infrastructure for code mining, ***Boa***, was presented in the workshop. The fundamental goal of Boa is usability and simplicity. Boa hides the low-level details of repository mining from the users. Boa provides a parallelization framework and all user queries are executed on Hadoop transparently, which makes it easy to write scalable and efficient user programs.

Boa Infrastructure: The Boa infrastructure is designed to diminish the barrier to entry for ultra-large scale MSR studies. Boa consists of a domain-specific language, its compiler, a data set that contains almost 700k open-source projects as of this writing, a backend based on map-reduce to effectively analyze this dataset, and a web-based frontend for writing code for MSR-related research.

Boa downloads and replicates the software repositories from various source code hosts such as SourceForge and GitHub. It translates the data into a custom format necessary for efficient querying. The translated data is then stored as a cache onto Boa's cluster of servers. This forms the data infrastructure for Boa and abstracts many of the details of how to send, store, update, and query such a large volume of data. Boa transforms

the original data into structured representation such as abstract syntax trees, and stores the the data . Users send queries to the system by writing user programs in the Boa query language. The user programs are compiled to Hadoop programs by Boa. Boa further deploys and executes the Hadoop programs on the cluster, and returns the results to the users.

Boa currently supports user queries through the Web interface on http://Boa.cs.iastate.edu. Users submit a query written in Boa's domain-specific query language to the website. User programs in Boa are concise and do not require external libraries. Once a user writes their query, they then select the dataset to use as input. Boa provides snapshots of the input data, marked with a timestamp. Boa periodically produces these datasets (at least yearly, in the future perhaps even monthly). Once a dataset is created it, is immutable and permanently available. This enables researchers to easily reproduce previous research results, by simply providing the same query and selecting the same input dataset.

For each submitted query, Boa creates a job. All jobs have a unique identifier and allow users to control them, such as stopping the job, resubmitting the job, and viewing the results of the job. The servers compile that query and translate it into a Hadoop map-reduce program. This program is then deployed onto the cluster and executes in a highly parallel, distributed manner. All of this is transparent to the users. The job page will show if compilation passed and any error messages. It will also show the status of executing the query. Once finished, it provides information about how long it took to execute and links for viewing and downloading the output. Once a job has completed without error, the output of the Boa program is available from the job's page. There are two options: users may view up to the first 64k of the output online or they may download the results as a text file.

When Boa executes a program, it first instantiates a separate program for each code project. Statistics are computed on each node of the cluster, and results are sent back to the aggregator with an aggregator function. The aggregator defines the operation to apply on the collected results, including sum, mean, top, bottom, and set operations.

Boa Language: Since Boa is a domain-specific language, it defines various domain-specific types, including types for projects, code repositories, and abstract syntax tree roots. Besides the predefined domain specific types, Boa supports user-defined functions. User can define custom functions, either in Boa language or Java. Boa also provides a type called `time` to represent unix-like timestamps. All date/time values are represented using this type. There are many built-in functions for working with time values, including obtaining specific date-related parts (such as day of month, month, year, etc), adding to the time by day, month, year, etc, and truncating to specific granularities. Strings in Boa are arrays of Unicode characters. Strings can be indexed to retrieve single characters. Strings can be concatenated together using the

plus (+) operator. There are also many built-in functions for working with strings to upper/lowercase them, get substrings, match against regular expressions, etc. Boa also provides several compound types. These types include arrays, maps, stacks, and sets and are composed of elements of basic type. Arrays can be initialized to a set of comma-delimited values surrounded by curly braces.

Boa provides a notion of output variables. Output variables declare an output aggregation function to use on the output. All aggregators can optionally take indices. Indices act as grouping operators. All output is sorted and grouped by the same index. Then the aggregation is applied to each group. It is also possible to have multiple indices in which case the grouping is performed left to right. The collection aggregator provides a way to simply collect some output without applying any aggregation to the values. A value emitted to this aggregator will appear directly in the results.

The *VISITOR* design pattern is a built-in feature in Boa. Users can perform depth-first traversal through abstract syntax trees by calling the *VISITOR* APIs. Boa also supports custom traversals over trees. Users can customize the stop criteria and traversal order.

Current Status and Future Work: An Eclipse plug-in will be released soon after the workshop, which aims to support integrated debugging and testing in Eclipse IDE. Boa is backed by 8 million projects, 23 million revision and 146 million unique files downloaded from the Web and stored on the cluster. Boa parses the source code and produces 71 billion abstract syntax tree nodes.

Boa now has more than 300 users from over 20 countries. In the future, Boa plans to include more advanced features, such as domain specific types for security, integration improvement of Eclipse plug-in, query reuse from the crowd, more advanced support for debugging and testing, and result sharing and collaboration.

Boa is going to support code mining research of studying the naturalness of software, including language modeling in code, code completion, recommendation, code synthesis, and statistical machine translation for code (language migration).

Code and Program Modeling

(Organizers: Charles Sutton & Tien Nguyen)
(Scribe: Vincent Hellendoorn, vhellendoorn@live.nl)

The goal of the session (Videos and slides) was to discuss models of code and programs that are based on data. The session had two invited speakers, one coming from a software engineering perspective and from a machine learning perspective. The software engineering speaker was Prof. Earl Barr, an associate professor at University College London, and the machine learning speaker was Dr Daniel Tarlow, a research at Microsoft Research in Cambridge, UK. Both speakers have done some of the early influential work in this area.

Talk by Earl Barr, University College London: Inference Problems in Software Engineering

1. Inferring programmer's intents and traceability

Inferring intent is a core problem in SE. Given finished project, we have some requirements, and a codebase. Traceability between the requirements and the codebase is important because it can tell us what the code is intended to do. Most large projects have thousands of unfixed bugs, because they only have finite resources. If we knew which requirements were bound to which bugs, we could explore this space more effectively, because they allow us to check the implementation.

A Code base is just a snapshot of version history. Requirements come from somewhere else: stakeholders. Many problems in projects happen when obtaining requirements from stakeholders. Complete requirements may be difficult to obtain. Thus, both requirements themselves, and links between requirements and artifacts, are partial (and noisy). There is additional data: issue trackers, mailing lists, documentation, logs. Problem of SE is: requirements are great and we'd like them, but mostly we have no requirements available.

Machine learning can help in relation extraction. Where could this be useful? You can recover these links and thus give features that can be used to build tools to check whether the features hold or not on new data. Furthermore, this can be interesting to the ML and NLP communities because all of this data is relatively structured (source code; requirements with semi-formal English/mixed); i.e., more structured than arbitrary text on the web. This structure may be used to improve performance (accuracy, resistance to noise, performance) of tools that attempt to recover traceability links.

2. Testing

Test suites are often informed by requirements, and are potentially a good way to obtain traceability links; but there are many complications. They tend to capture some under-approximation of the true intent thereof. The relationship can be ambiguous; we often do not know what aspect of requirements a given test case is supposed to test. NL people might think that strange; just look at the execution path. The problem with this is: in order to test some code, a test case may for instance need to traverse the file system of the OS, traverse code that opens/closes files. It does not intend to, but it must. Finding what paths a test is specifically intended to test is non-trivial.

Developers tend to create tests. We do have lots of tests; modern, Agile methods tend to emphasize testings, thus increasing test availability.

There are two lines of work in SE that exploit test suites. Test suites are normally designed for regression testing – give confidence that you can make changes without breaking existing functionality. Another area is automated program repair, which requires something that localizes the bug. Here, the goal is to synthesize a fix that passes test suite. The underlying assumption is that there is a correct test suite for a buggy program.

Another possible problem: although programmers do write test suites, they do not write enough (writing them is tedious & difficult). It would be nice if we could help them write test-suites. If you just randomly sample input domain, most test cases will be useless and redundant. We also do not know if the output is correct for samples test. This problem is often neglected in automated test generation. This may be a good place for machine learning: Given a test, we may use the observed behaviour and the query program's other concrete behavior to infer likely correct behavior, and perhaps even (if the answer is numerical) confidence intervals for correctness. This technique may help address Oracle problem.

3. Natural language, invariants, program by contract

We can think about other ways of recording and capturing programmer intent: programming by contract. No requirements are written in stakeholder format, but at a much lower level: embedded in code. Example: Eiffel, which supports design-by-contract. The code specifies preconditions and post-condition given parameters and returns.

Let us consider simple assertions: just having code annotated with a couple of asserts allows us to verify some properties, at least locally; it enables program validation. Problem with this is a context switch: not an easy shift to make. You first think empirically, as a flow of values, then you must think declaratively, in a universal,

quantified way, to come up with assertions. Loop invariants is a hard problem. Recently, work by Alex Aiken's group using PAC learning (learning geometric concepts) makes good progress on that.

Code contracts are used in OSS; developers tend to write pre-conditions, which are easier to specify; postconditions are more difficult. Our take: it's like writing unit tests, but writing specifications.

The challenge here, where NLP might help, is in suggesting asserts, to alleviate the annotation task. Daikon was mentioned a few times (a dynamic invariant Detector): It invalidates invariants through dynamic executions. For this tool, invariant templates were created by experts. We may use ML to learn possible invariants that are bespoke for code base, and thus more likely to be interesting. A problem with Daikon is that it finds many vacuous invariants (irrelevant to actual behavior of function). Another issue is to suggest where to place them. They are probes to the state space; there may be more natural places to put them than others. Places that are dominators in CFG for instance; not necessarily obvious to human developers, but easy to find by compilers.

Talk by Daniel Tarlow - Microsoft Research in Cambridge, UK Textual models of code: neural network probabilistic models

There is a specific class of language models that Tarlow had been trying on code that seem to strike a balance between being relatively simple (hopefully) and flexible and powerful enough to provide the benefits of NNLMs (neural network language models), and also allow us to input some information specifically from source code (e.g. tree-structure).

So why build models of source code? To create more "natural-looking" code. There are often a great many solutions to a given programming problem, and we want the most natural one. Any time you want to generate code that people will look at, you care about generating natural code (obey conventions). We want to build models that produce probability distributions of code that match real code.

To build good models, we should combine sample signals from programming text, associated natural language, and dynamic program executions (a la Daikon).

Ontologies and understanding of software semantics

(Scribe: Zhilin Yang, (zhiliny@cs.cmu.edu))
(Organizers: Dana Movshovitz-Attias & Tao Xie)

Brief of Topic

The goal of this session (Slides, Video) was to discuss methodology for understanding software semantics, with an emphasis on building structured data repositories for the software domain, such as ontologies. The aim was to understand both the needs of the Software Engineering community for structured understanding of software semantics, as well as the available Machine Learning methods and their relevance to the SE research needs. The long-term vision is that the multitude of existing data sources in the software domain can benefit from structured semantic resources, and ultimately also contribute to the development of improved structured learning methods.

The session was opened with a talk by Prof. Jane Cleland-Huang from DePaul University, who introduced an approach for building an ontology which was used to improve a traceability tool, DoCIT. The talk was followed by a discussion, which was scribed by Zhilin Yang. There were several main topics touched on in the discussion.

The main talk presented a natural language interface for software questions, TiQi, a traceability tool, DoCIT, and an ontology building method. TiQi processes natural language and classifies tokens into predefined lexicons, and synthesizes SQL queries based on the lexicons. DoCIT uses a transmissive-receptive heuristic to determine whether a trace link exists between two software artifacts. The ontology building approach leverages a trace matrix as distant supervision between software artifacts and domain documents, and uses a classifier to combine results from various extraction tools including topic models and association rules.

Topics from general discussion

Current use of ontologies in SE. Ontologies are currently being using in the SE domain as a means of constraining the space search for specially-tailored SE tasks. The ontologies that are commonly in use are very basic, for example, they do not include an entity hierarchy, and are only used to assess pairwise, entity-to-entity subclass relations. No other relations are currently explored. In comparison, richer ontologies have been explored by NLP researchers, which include entity and concept hierarchies

and a varied set of entity and concept pairwise relations, as well as general statistics of the use of entities in relevant domain corpora.

Motivation for using ontologies. We had a discussion on the motivation for using ontologies in the SE domain. Participants presented a preference for using flat probabilistic relations (such as subclass, or similarity), rather than hierarchical ones. However, while probabilistic relations can be used for finding statistically related information, when we try to answer questions or provide decision support, we need relations and formalism. This is where ontology building can be helpful. This motivation needs to be reiterated and better demonstrated through task-based examples. In other words, a collaboration between NLP and SE researchers is required, in order to better demonstrate where ontologies can be helpful for SE tasks.

Paraphrasing. A critical challenge of ontology building is that people usually tend to use different words even if they are referring to the same meaning. To address this challenge during ontology building, we can apply paraphrasing between different ontologies and expressions. Fuzzy matching can also be helpful. Word mismatch is a common problem. For example, when we do question answering based on Freebase, we need to reason over different representations of the same meaning.

Evolving terminology. A claim was made that code repositories and terminology are evolving faster than natural language. Additionally, people in different organization, in different contexts, use different terms for the same meaning. An important resource is the discussion in forums such as StackOverflow, which we can use to track the evolving terminology. We can look at the new terms to capture the underlying evolved nature of software terminology.

Using NELL as an SE ontology. NELL operates on top of a manually-built input ontology, and adds new facts to the existing ontology. The original ontology must be initiated by a human. We need a good ontology to start, which requires expertise in defining the relevant SE entities and relations. Once we have that, it is possible to use NELL for growing the ontology. It is also very likely, that a single SE ontology will not be relevant for all SE tasks.

Possible future directions. It is important to develop tools for ontology building for the software domain. Since software engineering is a highly technical domain, a lot of future tasks can benefit from the ontology. However, at this point in time, work on automatic ontology development is only being done as part of NLP research. Ontologies as a whole are not commonly used in SE and therefore no specific future tasks has been enumerated, that can accommodate the future use on richer ontological structures.

Summary and Recommendations

There is currently minimal use of ontologies for addressing SE tasks. The ontologies being used are, in fact, pairwise subclass relations identified over software entities. Meanwhile, NLP research provides the means for constructing richer ontologies, including entity and concept hierarchies, a multitude of software relations, and additional domain relevant statistics.

Importantly, there is a gap between the SE and NLP community in terms of the motivation for using ontologies. Unfortunately, in this workshop, we did not fully determine what are SE tasks that can directly benefit from the use of ontologies. This seems to be an important starting point, necessary for moving this area forward. Once specific tasks can be enumerated, then the needs and requirement of SE ontologies will surface additional challenges in ontology construction and use. These can be a basis for collaborations that will further both SE and NLP future research in semantic understanding through ontologies.

Open ended challenges. The following is a summary of challenges that have been identified with regards to ontology construction for the software domain:

- Identify SE tasks that can benefit the use of an ontology.
- Learning a set of rich entity-to-entity relations, beyond subclass relations.
- Using an automatically built ontology.
- What are the domains for which it is more appropriate to use ontologies over SE entities (such as short nouns and noun phrases) versus longer phrases and terms.
- Can NELL be used to build an SE ontology?
- Paraphrasing and soft-matching of ontology entities, during ontology generation and use.
- Using StackOverflow to determine the evolving nature of software terminology.

Information Retrieval

(Organizers Denys Poshyvanyk & Dana Movshovitz-Attias)
(Scribe: Martin White, mgwhite@email.wm.edu)

Brief of Topic

The goal of this session (Slides, Video) was to discuss the use of Information Retrieval (IR) technology in Software Engineering. The aim was to understand both the needs of the Software Engineering community for handling unstructured data (e.g., text) embedded in a multitude of software artifacts as well as the available Information Retrieval and NLP methods and their relevance to the SE research needs. The long-term vision is that the multitude of existing data sources in the software domain can benefit from Information Retrieval and NLP techniques.

The session was opened with a talk by Prof. Andrian Marcus from the University of Texas at Dallas, titled as "Using Text Retrieval in Software Engineering: An Overview". The talk overviewed major SE tasks currently supported by text retrieval (TR) methods as well as the detailed description on how TR methods are currently used in SE context. The talk concluded with an overview of crosscutting research problems and open challenges. The talk was followed by a discussion, which was scribed by Martin White.

The main talk started with a brief history of IR applications for SE tasks. Then the talk reviewed a number of applications of text retrieval techniques for SE problems such as restructuring and refactoring, software categorization, licensing analysis, clone detection, effort estimation, use case analysis, traceability link recovery, feature location, code reuse, bug triage, program comprehension, test case generation and others. The talk highlighted two main uses of text retrieval techniques for SE tasks: (1) formulate the SE task as a TR problem and (2) formulate the SE task as a text analysis problem. The first option includes the following steps: (a) building a corpus from software artifacts; (b) indexing a corpus using a given TR model; (c) formulating a query (manual or automatic); (d) computing similarities between the query and the documents in the corpus; (e) ranking the documents based on the similarities; (f) returning the top N as the result list; (g) inspecting the results; (h) going back to (c) if needed or stop. The second model, which involves formulating SE task as a text analysis problem, includes the following steps: (a) building a corpus, (b) indexing a corpus using a given TR method, (c) computing similarities between the documents in the corpus, (d) defining the measures/metrics based on the similarities; and (e) performing the analysis based on the obtained measures. The talk also presented detailed examples of applying TR for SE tasks, such as concept location and conceptual

cohesion measurement. The talk concluded with the discussion of crosscutting research problems related to corpus building (identifier splitting, synonyms, abbreviation expansion, using n-grams, term boosting), comparing and selecting TR models (VSM, LDA, LSI, BM25, as well as configuration selection), incremental indexing and vocabulary analysis, query formulation/reformulation, results presentation as well as information integration with other sources of information such as structural (e.g., dependencies), dynamic (e.g., execution traces), process information. There were several main topics touched on in the discussion.

Topics from general discussion

Information Retrieval (IR) metrics in SE research. The metrics are something that we need to consider as a community rather than continue to see patterns of papers that simply say "we used these metrics because these three papers used them too." There generally needs to be more context when settling on metrics for a particular approach.

IR versus information extraction in SE research. There was a comment that what we call IR in Software Engineering (SE) research is really information extraction. Moreover, viewing SE problems as information extraction problems rather than retrieval problems may yield new insights. Two possible explanations for this were provided. First, the confusion between retrieval and extraction was due in part to being ill-educated which the group generalized as a bias in SE research because SE researchers have not had formal education in natural language processing (NLP). Incidentally, this was a theme throughout the workshop. The second problem was because SE researchers initially were concerned with distinguishing search and retrieval when IR was first applied in SE contexts 15 years ago. There was no consensus on whether the type of extraction we do in SE research is equivalent to retrieval in the true NLP understanding.

Canonical methods versus formal methods. There was a question as to whether IR in SE research as we know it is even the right problem formulation. Do we need a new formalism, e.g., formal methods? However, the participants did not suggest any alternative problem formulations.

FDA rules and regulations. The discussion was also steered toward understanding specific SE problems where NLP can be brought to bear. One example was the FDA and their rules and regulations for fusion pumps. How do they know whether they extracted all of them? How do they know if two of them contradict each other? Can we look at our arsenal to approach this research? These questions sparked the discussion on the topics outlined below.

Requirements. A claim was made that requirements are still declarative and that requirements and code are two different classes of text. Code describes the running system, which is completely different text than requirements. Methods, which may be a property of requirements analysis, may not be good for analyzing text.

Requirements documentation. Commenting on specific tasks, there was a suggestion on the problem of ambiguity in requirements documents, however, it is still unknown how statistical NLP would be able to tell when something in the requirements documents does not mean what you think it means. A follow up discussion highlighted that this is vagueness rather than ambiguity and then it was emphasized that the issues are with interpreting different requirements. Another question was posed about informal/formal requirements: Have theorem provers been applied to requirements documentation to build a formal representation such that a prover can be run over it? However, general discussion was that writing requirements in formal ways just does not happen in industry; in other words industry prioritizes talking to the customer over documentation. A follow up question about whether we can leverage formal specifications so we can approximate the supervision signal was left open-ended.

Operational data. A claim was made that documentation is essentially operational data. This is data that was left behind and not intended to support things like formal methods whatsoever. In fact, we are in the text retrieval domain because we are dealing with scraps. Our text is not necessarily for any of these tasks.

Grounded language. A remark was made that it is useful to think about grounded language. Most NLP techniques will understand a sentence in isolation. It is worth considering what a sentence means in a particular context. There are a lot of opportunities between natural language understanding and the structure that code provides.

What propagates to industry? A comment was made that performance is key but understandability is also important. Moreover, statistical NLP is not completely handcuffed in its ability to ease understanding. For a programmer, it may not be necessary. However, most of the existing applications are generally tool-chains that not only tell you what to change but also tell you how to change in the case of an impact analysis problem. In other words, statistics will get you in the neighborhood and static analysis will get you to where you need to go.

Summary and Recommendation

There is currently a serious interest in adapting Information Retrieval approaches for solving SE problems. We had an active discussion that spanned a number of issues and open-ended questions. The following is a summary of challenges and opportunities

that have been identified with regards to applications of Information Retrieval for Software Engineering tasks:

- Combination with NLP;
- Matching tools/techniques to the task;
- Better evaluation mechanisms/research community standards;
- Data availability;
- Tool integration, optimization and user studies;
- Education and Infrastructure building.

Language Generation from Code

(Organizers: Dawn Lawrie & Graham Neubig)
(Scribe: Vincent Hellendoorn, vhellendoorn@live.nl)

Brief of Topic

This session (slides, video) covered methods to generate natural language descriptions of source code. These methods have been the subject of study within the SE community, and are often called "code summarization" methods, as they can create a concise natural language description of a body of code for more efficient developer perusal. Recently there have also been a few methods based on machine learning-based natural language processing techniques. This session discussed the current state of these methods, and where they should be going in the future.

The session was opened with a talk by **Graham Neubig** from the Nara Institute of Science and Technology, who did a survey of the state of the art in this field, followed by an open discussion. The session was scribed by **Vincent Hellendoorn.**

State of the Art in Code->NL Generation

The main talk presented a survey of the state of the art in the field, and the **slides** and **video** are openly available online. It covered approaches in the scientific literature to the generation of natural language from code and included a number of distinctions that can be made between the various methods.

- **Type of generated language:** Most natural language generation focuses on "code summarization," attempting to generate concise summaries of code blocks to make them easier to understand quickly. There is also work on pseudocode generation for beginner programmers, or method name generation.
- **Level of granularity of code:** There are methods to generate descriptions for variables, lines of code, functions, classes, commits, and multiple-file code concerns.
- **Generation methods:** Many methods are rule-based, but there are also data-driven methods based on information retrieval, machine translation, or neural networks.
- **Evaluation methods:** We can evaluate how good the summary is itself (intrinsic evaluation), or how much it helps with an external task.
- **Available data sources:** For data-driven methods, data is necessary. There have been some works harvesting data from stack overflow, or from comments

within code. There is also a set of line-by-line code/comment pairs created by hand.

Discussion Topics

Task-specificity: It is very important to think of what we'll use a code summary for when generating it. Collin McMillan noted that descriptions written for programmers and descriptions written for end-users will be very different, and Dawn Lawrie noted that even for programmers there are different roles such as testers or API users.

Example tasks: Some specific uses of code summaries were discussed. Moving beyond a line-by-line summary, there was discussion of generating summaries for developers versus end-users. The first line of a git commit can be an example of a natural language summary that describes the differences contained in the commit, as noted by Abram Hindle. An example of a summary where models do not currently exist is a tool-tip for method summaries in an IDE like Eclipse. Earl Barr notes that summaries could help developers build mental models of code more quickly. Automatically generating comments for code is also important because, as noted by Denys Poshyvanyk, code is modified much more frequently then the associated comments; thus, they decay quickly. Turning to the end-user, one type of useful summary is release notes. Andrian Marcus noted that new features, permissions, and licensing might all be summarized. Baishakhi Ray commented that code summarization could have a big effect on security policy, referencing a 2015 ICSE paper on information flow, which tracked Android applications to see how processes communicate and whether they may violate policy.

Other methods for achieving similar goals: Prem Devanbu noted that Charles Sutton's group has proposed methods for auto-folding uninteresting methods in source code, Collin McMillan noted methods for generating API usage patterns, and Abram Hindle noted code diffs. These do not require natural language, but can achieve similar goals and are interesting in their own right. Also, Sol Greenspan noted that in MDSD environments, if the model is specified properly comments may not be necessary.

Can we discriminate between good and bad natural language descriptions?: There was some interested on the software engineering side expressed by Collin McMillan about ways to discriminate whether a software description was a good one or a bad one. Nate Kushman said that our standard tool here would be language models, and Graham Neubig and Chris Quirk noted that there are many types of features you can use in language models depending on what you want to capture. Denys Poshyvanyk noted that one problem is that comments tend to go stale, so it would be nice if we could verify them.

What kinds of context do we need to condition on?: Sol Greenspan noted that domain knowledge is extremely important when generating documents.

How could we get data for training machine learning models?: Andrian Marcus noted that there are many source for data, including descriptions of methods, stack overflow, and release notes. Many people noted that some projects have excellent comments, such as the aTunes project, parts of the Linux Kernel, and Google projects that have been open sourced. Graham Neubig notes that the data often needs to be "clean", in the sense that most of the content must be reflected in both the code and the natural language, and Dana Movshovitz-Attias noted that "clean" is still difficult to define and perhaps task specific. Sonia Haduc noted that there is a new stack overflow project "Warriors of Documentation" that may be very useful for this.

Can we handle edge/corner cases?: Nate Kushman noted that cases that are unusual may actually be of more value to document, but it's not easy to generate these.

Summary and Recommendations

In short, the main topics of discussion focused on "why we generate language from code," "how we obtain data for training models," "whether there are other methods to achieve the same goal," and "whether existing natural language descriptions can be verified." Based on the discussion, we can make the following recommendations:

- It is important to keep the task in mind when generating data from code, in particular whether the descriptions will be generated for developers or end users, and adjust the strategy based on this.
- We can also consider other methods for achieving the same goal, such as hiding pieces of code that are not relevant, or showing usage patterns.
- Many noted the potential of stack overflow or comments to generate data, but many other forms of developer activity are interesting as potential targets.
- It will be best to think about not only language generation, but also language verification, particularly in the case of stale or incomplete comments.

Natural Language Programming and Semantic Parsing

(Organizers: Ray Mooney & Chris Quirk)
(Scribe: Gagan Bansal bansalg@cs.washington.edu)

Brief of Topic

This session (Slides, Video, Second Part) discussed progress and future work on the challenging problem of translating user instructions in natural language to executable computer code. In NLP, the subarea of *semantic parsing* has studied the problem of mapping natural language to a formal representation of its meaning. The majority of the work in this area has been on understanding database queries; however, there has been relatively little work on other types of programs. The session was organized as a series of short 5 minute talks by all panel members, including Ray Mooney, Chris Quirk, Gagan Bansal, Yoav Artzi, Percy Liang, Srini Iyer, Nate Kushman, and Yi Wei.

Mooney briefly introduced the NLP work on semantic parsing, focusing on the role of using machine learning methods to induce semantic parsers from corpora of natural language sentences paired with their formal language translations. Quirk described a recent project with Mooney and Michel Galley on mapping natural language descriptions of simple "If This Then That" scripts to code, using a corpus of over 100K examples collected from ifttt.com. Bansal discussed his on-going work with Dan Weld on this same corpus, exploiting text from the IFTTT API documentation to improve the interpretation of these short program descriptions. Artzi and Liang briefly described their work on semantic parsing, which has mostly focussed on interpreting natural language queries to FreeBase. Iyer discussed approaches to generate natural language descriptions of SQL queries. Kushman described his work on interpreting operating system instructions and regular-expression descriptions, focusing on the importance of handling non-compositionality, in which there is no direct mapping from words and phrases to program constructs. Wei discussed methods for searching over code snippets and synthesizing code snippets given natural language descriptions.

Other items from discussion

There was discussion of what application domains were most suited for natural language programming. End-user development of short programs such as scripts for spreadsheets or data analytics was considered a good application. For for more professional programmers, areas such as model-driven engineering and generation of program assertions for use in verification and debugging were discussed.

Programmers may write more assertions and other annotations useful for program analysis if they are provided with a natural language input mechanism.

Training natural language programming systems requires relevant large “parallel corpora” of NL descriptions and programs, which can be difficult to find or construct. Unsupervised approaches such as Unsupervised Semantic Parsing may be useful in reducing the amount of supervision required. Alternate supervision signals such as input/output examples may also help; NL programming can be combined with programming by demonstration.

Most current approaches use single user inputs. Often a single turn is insufficient to describe program intent. Using dialog strategies for correcting and refining intent will likely be important, but poses difficulties for data driven settings as the problem space becomes larger. On a related note, users must often check and correct the resulting code. Systems must provide some representation of the program for the user to inspect, and likely provide debugging tools.

To address the issue that programs may not be directly compositional from natural language, it might be best to first map natural language into a domain-independent formal logical language, and then separately learn to map this intermediate representation into a final program. One candidate language is Abstract Meaning Representation (AMR), a recent focus in the NLP community.

Automatic evaluation is a major consideration. NL approaches like BLEU are attractive for speed of evaluation and tool availability, but do not measure syntactic well-formedness, much less semantic correctness. That said, fast but imperfect measure helped drive some of the largest progress in a number of NLP areas over the past decade.

Summary and Recommendations

Research in mapping language to code (and code to language) is made more difficult by the lack of annotated data resources. Syntactic parsing, semantic parsing, and machine translation all made great strides as more data became available. That said, it is not obvious which domains will be tractable and see the greatest gains in the short to medium term.

We encourage the creation of curated resources that are suitable for pilot studies. Rather than annotating millions of instances, many projects can be explored with only hundreds or thousands of training instances. The GEOQUERY dataset spurred a broad range of research even though it had approximately one thousand natural and formal language pairs. As a starting point, the community could construct parallel sets of code

assertions and their descriptions, or natural language descriptions of formal requirements.

Additionally, we might curate a small set of comment-code pairs. The relationship between the code and the comment could be identified (is the comment explanatory, redundant, inconsistent, etc.). Building accurate classifiers of comment intent is an interesting problem itself, and could also lead to better parallel code/comment data.

Given a few benchmark tasks, the community next needs to standardize on evaluation metrics. Ideally there would be both fast automatic metrics amenable to optimization, and slow but accurate metrics potentially with a human in the loop. This combination allows for fast daily progress and incremental research while retaining a long term focus on utility.

Deeper annotation of these resources could also be useful. The natural language utterances could be annotated with gold standard syntactic or even semantic parses (according to the Abstract Meaning Representation specification, for instance). Performing this annotation on a few thousand sentences would allow domain adaptation and evaluation work. Likewise the code could be annotated with abstract syntax trees, program analysis information, or whatever resources are readily available.

Finally, we need a good means of disseminating information to the community. There are already several important resources, such as the Stack Overflow data. We should maintain a website or wiki that allows community members to identify and update datasets, benchmarks, and key implementations.

Closing Session

The closing session was dominated by a discussion of available, as well as desirable resources, datasets and challenges that are available to researchers interested to explore this area further.

The full spreadsheet is publicly available.

Conclusion and Acknowledgements

Recent research has exposed the repetitive, predictable nature of software, and the utility of statistical models in both improving the performance of traditional software tools, and in facilitating new types of tools. This interdisciplinary research program requires deep skills in both software engineering and statistical natural language processing. This workshop was organized to bring together two communities that in the past have had limited interactions. The workshop began with two tutorials, one by an SE researcher on “big-data” approaches in software engineering, and another, by an NLP researcher, on the foundations and achievements of statistical NLP. Over two following days, a variety of topics were discussed by the attendees over several sessions. Each session began with a “mood setting” talk, and was followed by active and vigorous discussions. The document above provides links to slides and videos (graciously made available by the speakers, and recorded and made available through the auspices of Microsoft Research).

The organizers of the workshop gratefully acknowledge support from Microsoft Research for the use of their excellent conference facilities, and also generous support provided for meals, local transportation, and refreshments for attendees. We gratefully acknowledge support from the National Science Foundation (Award Number: 1551318) which provided travel support for all attendees.

Bibliography of the Area

[1] A. Aiken. MOSS. Accessed 2015/05/31.

[2] Y. Al-Onaizan, J. Curin, M. Jahr, K. Knight, J. Lafferty, D. Melamed, F.-J. Och, D. Purdy, N. A. Smith, and D. Yarowsky. Statistical machine translation. In Final Report, JHU Summer Workshop, volume 30, 1999.

[3] M. Allamanis and E. AC. Bimodal modelling of source code and natural language. In Proceedings of The 32nd International Conference on Machine Learning, pages 2123–2132, 2015.

[4] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.

[5] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, pages 207–216. IEEE, 2013.

[8] M. Allamanis and C. Sutton. Mining idioms from source code. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM.

[9] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014), Hong Kong, volume 16, 2014.

[10] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. The Journal of Machine Learning Research, 3:1137–1155, 2003.

[11] S. Bird, E. Klein, and E. Loper. Natural language processing with Python. ” O’Reilly Media, Inc.”, 2009.

[12] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. Computational linguistics, 19(2):263–311, 1993.

[13] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE ’09, pages 213–222, New York, NY, USA, 2009. ACM.

[14] J. C. Campbell, A. Hindle, and J. N. Amaral. Python: Where the mutants hide or, corpus-based coding mistake location in dynamic languages.

[15] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren’t natural: improving error reporting with language models. In Proceedings of the 11th Working Conference on Mining Software Repositories, pages 252–261. ACM, 2014.

[16] D. Cer, M. Galley, D. Jurafsky, and C. D. Manning. Phrasal: A statistical machine translation toolkit for exploring new model. Proceedings of the NAACL HLT 2010 Demonstration Session, pages 9–12, 2010.

[17] C. Chelba and F. Jelinek. Exploiting syntactic structure for language modeling. In Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1, pages 225–231. Association for Computational Linguistics, 1998.

[18] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In Proceedings of the 34th Annual Meeting on Association for Computational Linguistics, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

[19] D. Chollak. Software bug detection using the n-gram language model. 2015.

[20] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In Proceedings of the 25th international conference on Machine learning, pages 160–167. ACM, 2008.

[23] A. Dekhtyar and J. H. Hayes. Good benchmarks are hard to find: Toward the benchmark for information retrieval applications in software engineering. 2006.

[24] S. Della Pietra, V. Della Pietra, R. L. Mercer, and S. Roukos. Adaptive language modeling using minimum discriminant estimation. In Proceedings of the workshop on Speech and Natural Language, pages 103–106. Association for Computational Linguistics, 1992.

[26] J.-M. Favre, D. Gasevic, R. L¨ammel, and E. Pek. Empirical language analysis in software linguistics. In Software Language Engineering, pages 316–326. Springer, 2011.

[28] J. Fowkes, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton. Autofolding for source code summarization. arXiv preprint arXiv:1403.4503, 2014.

[29] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. Cacheca: A cache language model based code suggestion tool. ICSE Demonstration Track, 2015.

[30] M. Gabel and Z. Su. A study of the uniqueness of source code. In SIGSOFT FSE, pages 147–156, 2010.

[31] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In Reverse Engineering (WCRE), 2010 17th Working Conference on, pages 35–44. IEEE, 2010.

[33] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on, pages 332–343. IEEE, 2009.

[34] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.

[35] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. MSR, 2015,
[36] R. Hill and J. Rideout. Automatic method completion. In Automated Software Engineering, 2004. Proceedings. 19th International Conference on, pages 228–235. IEEE, 2004.
[37] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In Proceedings of ICSE 2012 (34th International Conference on Software Engineering), pages 837–847, 2012.
[38] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In Proceedings of the 27th international conference on Software engineering, pages 117–125. ACM, 2005.
[39] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, pages 49–65, New York, NY, USA, 2014. ACM.
[40] F. Jacob and R. Tairas. Code template inference using language models. In Proceedings of the 48th Annual Southeast Regional Conference, page 104. ACM, 2010.
[41] F. Jelinek, J. D. Lafferty, and R. L. Mercer. Basic methods of probabilistic context free grammars. Springer, 1992.
[42] D. Jurafsky and H. James. Speech and language processing an introduction to natural language processing, computational linguistics, and speech. 2000.
[43] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, pages 173–184. ACM, 2014.
[44] G. Karypis. Evaluation of item-based top-n recommendation algorithms. In Proceedings of the tenth international conference on Information and knowledge management, pages 247–254. ACM, 2001.
[45] B. Kitchenham. Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004):1–26, 2004.
[46] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering a systematic literature review. Information and Software Technology, 51(1):7 – 15, 2009. Special Section - Most Cited Articles in 2002 and Regular Research Papers.
[47] B. A. Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. In Proceedings of the 26th international conference on software engineering, pages 273–281. IEEE Computer Society, 2004.
[48] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In Acoustics, Speech, and Signal Processing, 1995. ICASSP- 95., 1995 International Conference on, volume 1, pages 181–184. IEEE, 1995.
[49] K. Knight. A statistical mt tutorial workbook, 1999.
[50] P. Koehn. Statistical machine translation. Cambridge University Press, 2009.

[51] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. Software Engineering, IEEE Transactions on, 38(1):54–72, 2012.
[52] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the sphinx speech recognition system. Acoustics, Speech and Signal Processing, IEEE Transactions on, 38(1):35–45, 1990.
[53] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. arXiv preprint arXiv:1401.0514, 2014.
[54] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. RN, 15:01, 2015.
[55] A. Marcus and J. I. Maletic. Recovering documentation-to-sourcecode traceability links using latent semantic indexing. In Software Engineering, 2003. Proceedings. 25th International Conference on, pages 125–135. IEEE, 2003.
[56] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In Companion Proceedings of the 36th International Conference on Software Engineering, pages 492–495. ACM, 2014.
[57] T. Mikolov, M. Karafi´at, L. Burget, J. Cernock`y, and S. Khudanpur. Recurrent neural network based language model. In INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010, pages 1045–1048, 2010.
[58] L. D. Misek-Falkoff. The new field of “software linguistics”: An earlybird view. In Selected Papers of the 1982 ACM SIGMETRICS Workshop on Software Metrics: Part 1, SCORE ’82, pages 35–51, New York, NY, USA, 1982. ACM.
[59] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. arXiv preprint arXiv:1409.5718, 2014.
[60] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang. Building program vector representations for deep learning. arXiv preprint arXiv:1409.3358, 2014. 2014.
[61] D. Movshovitz-Attias and W. W. Cohen. Natural language models for predicting programming comments. In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 35–40, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.
[62] H. Murakami, K. Hotta, Y. Higo, and S. Kusumoto. Predicting next changes at the fine-grained level.
[63] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, pages 457–468, New York, NY, USA, 2014. ACM.
[64] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code.
[65] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code

completion. In Proceedings of the 34th International Conference on Software Engineering, pages 69–79. IEEE Press, 2012.

[66] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 651–654. ACM, 2013.

[67] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 180–190. IEEE, 2013.

[68] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 532–542. ACM, 2013.

[69] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 383–392. ACM, 2009.

[70] E. Nilsson. Abstract syntax tree analysis for plagiarism detection. 2012.

[71] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In Proceedings of the 2013 International Conference on Software Engineering, pages 522–531. IEEE Press, 2013.

[72] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics, pages 311–318. Association for Computational Linguistics, 2002.

[73] A. Pauls and D. Klein. Large-scale syntactic language modeling with treelets. In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1, pages 959–968. Association for Computational Linguistics, 2012.

[74] J. Y. Poon, K. Sugiyama, Y. F. Tan, and M.-Y. Kan. Instructor-centric source code plagiarism detection and plagiarism corpus. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, pages 122–127. ACM, 2012.

[75] M. Post and D. Gildea. Bayesian learning of a tree substitution grammar. In Proceedings of the ACL-IJCNLP 2009 Conference Short Papers, pages 45–48. Association for Computational Linguistics, 2009.

[76] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu. On the “naturalness” of buggy code. 2015 (in submission).

[77] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The uniqueness of changes: Characteristics and applications. Technical report, Microsoft Research Technical Report, 2014.

[78] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In Proceedings of the 42nd Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, pages 111–124. ACM, 2015.
[79] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, page 44. ACM, 2014.
[80] R. Robbes and M. Lanza. How program history can improve code completion. In Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on, pages 317–326. IEEE, 2008.
[81] R. Rosenfeld. A maximum entropy approach to adaptive statistical language modelling. Computer Speech & Language, 10(3):187–228, 1996.
[82] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 76–85. ACM, 2003.
[83] P. Schulam, R. Rosenfeld, and P. Devanbu. Building statistical language models of code. In Data Analysis Patterns in Software Engineering (DAPSE), 2013 1st International Workshop on, pages 1–3. IEEE, 2013.
[84] C. Shah and W. B. Croft. Evaluating high accuracy retrieval techniques. In Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, pages 2–9. ACM, 2004.
[85] Y. Shi, P. Wiggers, and C. M. Jonker. Towards recurrent neural networks language models with linguistic and contextual features. In INTERSPEECH, 2012.
[86] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In Proceedings of the 25th International Conference on Software Engineering, pages 74–83. IEEE Computer Society, 2003.
[87] J. ´ Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In MSR, pages 1–5, 2005.
[88] G. Sridhara, V. S. Sinha, and S. Mani. Naturalness of natural language artifacts in software. In Proceedings of the 8th India Software Engineering Conference, ISEC ’15, pages 156–165, New York, NY, USA, 2015. ACM.
[89] A. Stolcke, C. Chelba, D. Engle, V. Jimenez, L. Mangu, H. Printz, E. Ristad, R. Rosenfeld, et al. Dependency language modeling. 1997.
[90] A. Stolcke et al. Srilm-an extensible language modeling toolkit. In INTERSPEECH, 2002.
[91] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 562–572, New York, NY, USA, 2014. ACM.
[92] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering), pages 269–280. ACM, 2014.
[93] M. Velez, D. Qiu, Y. Zhou, E. T. Barr, and Z. Su. A study of” wheat” and” chaff” in source code. arXiv preprint arXiv:1502.01410, 2015.

[94] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward Deep Learning Software Repositories. In Mining Software Repositories (MSR), 2015 12th IEEE Working Conference on, volume 1, page 1, 2015.

[95] A. T. Ying and M. P. Robillard. Selection and presentation practices for code example summarization. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 460–471. ACM, 2014.

[96] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining api mapping for language migration. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 195–204. ACM, 2010.

[97] Movshovitz-Attias, Dana, and William W. Cohen. "KB-LDA: Jointly Learning a Knowledge Base of Hierarchy, Relations, and Facts."

[98] Oda, Yusuke, et al. "Learning to Generate Pseudo-code from Source Code using Statistical Machine Translation."

[99] Drummond, Anna, et al. "Learning to Grade Student Programs in a Massive Open Online Course." Data Mining (ICDM), 2014 IEEE International Conference on. IEEE, 2014.

[100] Piech, Chris, et al. "Learning program embeddings to propagate feedback on student code." *arXiv preprint arXiv:1505.05969* (2015).