On the Naturalness of Software

Abram Hindle, Earl T. Barr, Zhendong Su
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
{ajhindle,barr,su}@cs.ucdavis.edu

Mark Gabel
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75080 USA
mark.gabel@utdallas.edu

Premkumar Devanbu
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
devanbu@cs.ucdavis.edu

***Abstract*—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.**

We begin with the conjecture that most software is also natural, in the sense that it is created by humans at work, with all the attendant constraints and limitations—and thus, like natural language, it is also likely to be repetitive and predictable. We then proceed to ask whether a) code can be usefully modeled by statistical language models and b) such models can be leveraged to support software engineers. Using the widely adopted n-gram model, we provide empirical evidence supportive of a positive answer to both these questions. We show that code is also very repetitive, and in fact even more so than natural languages. As an example use of the model, we have developed a simple code completion engine for Java that, despite its simplicity, already improves Eclipse's built-in completion capability. We conclude the paper by laying out a vision for future research in this area.

***Keywords*-language models; n-gram; natural language processing; code completion; and code suggestion**

I. Introduction

The word "natural" in the title of this paper refers to the fact that code, despite being written in an artificial language (like C or Java) is a *natural product of human effort*. This use of the word *natural* derives from the field of *natural language processing*, where the goal is to automatically process texts in natural languages, such as English and Tamil, for tasks such as translation (to other natural languages), summarization, understanding, and speech recognition.

The field of natural language processing ("NLP", see Sparck-Jones [1] for a brief history) went through several decades of rather slow and painstaking progress, beginning with early struggles with dictionary and grammar-based efforts in the 1960s. In the '70s and '80s, the field was reanimated with ideas from logic and formal semantics, which still proved too cumbersome to perform practical tasks at scale. Both these approaches essentially dealt with NLP from first principles—addressing *language*, in all its rich theoretical glory, rather than examining corpora of actual *utterances*, *i.e.*, what people actually write or say. In the 1980s, a fundamental shift to *corpus-based, statistically rigorous* methods occurred. The availability of large, on-line corpora of natural language text, including "aligned" text with translations in multiple languages,[1] along with the computational muscle (CPU speed, primary and secondary storage) to estimate robust statistical models over very large data sets has led to stunning progress and widely-available practical applications, such as statistical translation used by `translate.google.com`.[2] We argue that an essential fact underlying this modern, exciting phase of NLP is *natural language may be complex and admit a great wealth of expression, but what people write and say is largely regular and predictable*.

Our *central hypothesis* is that the same argument applies to software:

> *Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*

We believe that this is a general, useful and practical notion that, together with the very large publicly available corpora of open-source code, will enable a new, rigorous, statistical approach to a wide range of applications, in program analysis, error checking, software mining, program summarization, and code searching. This paper is the first step in what we hope

Abram Hindle is now with University of Alberta, Edmonton.
Mark Gabel was at UC Davis when this work was done.

[1]This included the Canadian Hansard (parliamentary proceedings), and similar outputs from the European parliament.

[2]Indeed, a renowned pioneer of the statistical approach, Fred Jelenik, is reputed to have exclaimed: "Every time a linguist leaves our group, the performance of our speech recognition goes up!!!" See http://en.wikiquote.org/wiki/Fred_Jelinek.

will be a long and fruitful journey. We make the following contributions:

1) We provide support for our central hypothesis by instantiating a simple, widely-used statistical language model, using modern estimation techniques over large software corpora;
2) We demonstrate, using standard cross-entropy and perplexity measures, that the model indeed captures the high-level statistical regularity that exists in software at the n-gram level (probabilistic chains of tokens);
3) We illustrate the use of such a language model by developing a simple code suggestion tool that substantially improves upon the existing suggestion facility in the widely-used Eclipse IDE; and
4) We lay out our vision for an ambitious research agenda that exploits large-corpus statistical models of natural software to aid in a range of different software engineering tasks.

II. Motivation and Background

There are many ways one could exploit the statistics of natural programs. We begin with a simple motivating example. We present more ambitious possibilities later.

Consider a speech recognizer, receiving a noisy signal corresponding to *"In Brussels today, the European Central* <radio phizz> *announced that interest rates remain unchanged..."*. A good speech recognizer might guess that the noisy word was *"Bank"* rather than *"fish"* from context. Likewise, consider an integrated development environment (IDE) into which a programmer has typed in the partial statement: "`for(i=0;i<10`". In this context, it would be quite reasonable for the IDE to suggest the completion "`;i++){ `" to the programmer.

Why do these guesses seem so reasonable to us? In the first case, the reason lies in the highly predictable nature of newscasts. News reports, like many other forms of culturally contextualized and stylized natural language expression, tend to be well-structured and repetitive. With a reasonable prior knowledge (*viz.*, a good statistical model) of this style, it is possible to rank-order likely utterances. Thus, if we hear the words *"European Central"*, the next word is more likely to be *"Bank"* rather than *"fish"*. This fact is well-known and exploited by speech recognizers, natural language translation devices, and even some OCR (optical character recognition) tools. The second example relies on a lesser-known fact: *natural programs are quite repetitive*. This fact was first observed and reported in a very large-scale study of code by Gabel and Su [2], which found that code fragments of surprisingly large size tend to reoccur. Thus, if we see the fragment `for(i=0;i<10` we know what follows in most cases. In general, if we know the most likely sequences in a code body, we can often help programmers complete code. What this essentially amounts to is *using a code corpus to estimate the probability distribution of code sequences*. With the ability to calculate such a distribution *and* if this distribution has low-entropy, we will often be able to guess with high confidence what follows the prefix of a code sequence.

What should the form of a such a distribution be, and how should we estimate its parameters? In NLP, these distributions are called "language models".

A. Language Models

A language model essentially assigns a probability to an utterance. For us, "utterances" are programs. More formally, consider a set of allowable program tokens[3] $\mathcal{T}$, and the (overgenerous) set of possible program sequences $\mathcal{T}^*$; we assume the set of possible implemented systems to be $S \subset \mathcal{T}^*$. A language model is a probability distribution $p(.)$ over systems $s \in S$:

$$\forall s \in S\ [0 < p(s) < 1]\ \wedge\ \sum_{s \in S} p(s) = 1$$

In practice, given a corpus C of programs $C \subseteq S$, and a suitably chosen parametric distribution $p(.)$, we attempt to calculate a maximum-likelihood estimate of the parameters of this distribution; this gives us an estimated language model. The choice of a language model is usually driven by practicalities: how easy is it to estimate and use. For these reasons, the most ubiquitous is the *n-gram* model, which we now describe.

Consider the sequence of tokens in a document (in our case, a system s), $a_1 a_2 \ldots a_i \ldots a_n$. N-gram models statistically estimate how likely a token is to follow other tokens. Thus, we can estimate the probability of a document based on the product of a series of conditional probabilities:

$$p(s) = p(a_1)p(a_2 \mid a_1)p(a_3 \mid a_1 a_2) \ldots p(a_n | a_1 \ldots a_{n-1})$$

N-gram models assume a *Markov property*, *i.e.*, token occurrences are influenced only by the $n-1$ tokens that precede the token under consideration, thus for 4-gram models, we assume

$$p(a_i | a_1 \ldots a_{i-1}) \simeq p(a_i \mid a_{i-3} a_{i-2} a_{i-1})$$

These models are estimated on a corpus using maximum-likelihood based frequency-counting of token sequences. Thus, if "$*$" is a wildcard, we can estimate the probability that a_4 follows the tokens a_1, a_2, a_3 with:

$$p(a_4 | a_1 a_2 a_3) = \frac{count(a_1 a_2 a_3 a_4)}{count(a_1 a_2 a_3 *)}$$

In practice, estimation of n-gram models is quite a bit more complicated. The main difficulties arise from data sparsity, *i.e.*, the richness of the model in comparison to the available data. For example, with 10^4 token vocabulary, a trigram model must estimate 10^{12} coefficients. Some trigrams may never occur in one corpus, but may in fact

[3]Here, we use "token" to mean its lexeme.

occur elsewhere. This leads to technical difficulties; when we encounter a previously unseen n-gram, we are in principle "infinitely surprised", because an "infinitely improbable" event x, which did not occur in the training corpus and was therefore estimated to have $p(x) = 0$, actually occurs. This leads to infinite entropy values, as will become evident below. *Smoothing* is a technique to handle such cases while still producing usable results with sufficient statistical rigour. Fortunately, there exist a variety of techniques for smoothing the estimates of a very large number of coefficients, some of which are larger than they should be and others smaller. Sometimes it is better to back-off from a trigram model to a bigram model. The technical details are beyond the scope of this paper, but can be found in any advanced NLP textbook. In practice we found that Modified Kneser-Ney smoothing (*e.g.*, Koehn [3], §7) gives good results for software corpora, compared to plain Kneser-Ney, Lidstone/add-one smoothing and maximum likelihood. For instance, maximum likelihood is often infinitely surprised and Lidstone smoothing tends to overemphasize surprises. However, we note that these are very early efforts in this area, and new modeling and estimation techniques, tailored for software, might improve on the results presented below.

How do we know when we have a good language model?

B. What Makes a Good Model?

Given a repetitive and highly predictable corpus of documents (or programs), a good model captures the regularities in the corpus. Thus, a good model, estimated carefully from a representative corpus, will predict with high confidence the contents of a new document drawn from the same population. Such a model can guess the contents of the new document with very high probability. In other words, the model will not find a new document particularly surprising, or "perplexing". In NLP, this idea is captured by a measure called *perplexity*, or its log-transformed version, *cross-entropy*.[4] Given a document $s = a_1 \dots a_n$, of length n, and a language model $\mathcal{M}$, we assume that the probability of the document estimated by the model is $p_{\mathcal{M}}(s)$. We can write down the cross-entropy measure as:

$$H_{\mathcal{M}}(s) = -\frac{1}{n} \log\ p_{\mathcal{M}}(a_1 \dots a_n)$$

and by the formulation presented in Section II-A:

$$H_{\mathcal{M}}(s) = -\frac{1}{n} \sum_{i=1}^{n} \log\ p_{\mathcal{M}}(a_i \mid a_1 \dots a_{i-1})$$

This is a measure of how "surprised" a model is by the given document. A good model has low entropy for target documents. It gives higher probabilities, (closer to 1, and thus lower absolute log values) to more frequent words, and lower probabilities to rare ones. If one could manage to deploy a (hypothetical) truly superb model within an IDE to help programmers complete code fragments, it might be able to guess with high probability most of the program, so that most of the programming work can be done by just hitting a tab key! In practice of course, we would probably be satisfied with a lot less.

But how good are the models that we can actually build for "natural" software? Is software is really as "natural" (*i.e.*, unsurprising) as natural language?

III. Methodology and Findings

To shed light on this question, we performed a series of experiments with both natural language and code corpora, first comparing the "naturalness" (using cross-entropy) of code with English texts, and then comparing various code corpora to each other to further gain insight into the similarities and differences between code corpora.

Our natural language studies were based on two very widely used corpora: the Brown corpus and the Gutenberg corpus.[5] For code, we used two corpora, a collection of Java projects and a collection of applications from Ubuntu, broken up into application domain. All are listed in Table I

After removing comments, the projects were lexically analyzed according to language syntax to produce token sequences that were used to estimate n-gram language models. Most of our corpora are in C and Java. Extending to other languages is trivial.

The Java projects were our central focus; we used them both for cross-entropy studies, and some experiments with an Eclipse plug-in for a language-model-based code-suggestion task. Table I describes the 10 Java projects that we used. The *Version* indicates the date of the last commit to the master branch in the Git repository when we cloned the project. *Lines* is calculated using Unix `wc` on each file within each repository, and tokens are extracted from each of these files. *Tokens* counts the total tokens extracted; *Unique Tokens* counts the distinct tokens. The Ubuntu domain categories were quite large in some cases, ranging up to 9 million lines, 41 million tokens (one million unique). The number of unique tokens is interesting and relevant, as they give a very rough indication on the potential "surprisingness" of the project corpus. If these unique tokens were uniformly distributed throughout the project (highly unlikely), we could expect a cross-entropy of $log_2(1.15E6)$, or approximately 20 bits. A similar calculation for the Java projects ranges from 13 to 17 bits.

A. Cross-Entropy of Code

Cross-entropy is a measure of how surprising a test document is to a distribution model estimated from a corpus.

[4] http://en.wikipedia.org/wiki/Cross_entropy; see also [4], §2.2, page 75, equation 2.50.

[5] We retrieved these corpora from http://www.nltk.org/.

Table I
10 JAVA PROJECTS, C CODE FROM 10 UBUNTU 10.10 CATEGORIES, AND 3 ENGLISH CORPUS USED IN OUR STUDY. ENGLISH IS THE CONCATENATION OF BROWN AND GUTENBERG. UBUNTU 10.10 MAVERICK WAS RELEASED ON 2010/10/10; THE NUMBER OF PROJECTS IN EACH CATEGORY IS IN PARENTHESES.

			Tokens	
Java Project	**Version**	**Lines**	Total	Unique
Ant	20110123	254457	919148	27008
Batik	20110118	367293	1384554	30298
Cassandra	20110122	135992	697498	13002
Eclipse-E4	20110426	1543206	6807301	98652
Log4J	20101119	68528	247001	8056
Lucene	20100319	429957	2130349	32676
Maven2	20101118	61622	263831	7637
Maven3	20110122	114527	462397	10839
Xalan-J	20091212	349837	1085022	39383
Xerces	20110111	257572	992623	19542
			Tokens	
Ubuntu Domain	**Version**	**Lines**	Total	Unique
Admin (116)	10.10	9092325	41208531	1140555
Doc (22)	10.10	87192	362501	15373
Graphics (21)	10.10	1422514	7453031	188792
Interp. (23)	10.10	1416361	6388351	201538
Mail (15)	10.10	1049136	4408776	137324
Net (86)	10.10	5012473	20666917	541896
Sound (26)	10.10	1698584	29310969	436377
Tex (135)	10.10	1405674	14342943	375845
Text (118)	10.10	1325700	6291804	155177
Web (31)	10.10	1743376	11361332	216474
			Tokens	
English Corpus	**Version**	**Lines**	Total	Unique
Brown	20101101	81851	1161192	56057
Gutenberg	20101101	55578	2621613	51156

Thus, if one tests a corpus against itself, one has to set aside some portion of the corpus for testing, and estimate (train) the model on the rest of the corpus. In all our experiments, we measured cross-entropy by averaging over a 10-fold cross-validation: we split the corpus 90%–10% (in lines) at 10 random locations, trained on the 90% and tested on 10%, and measured the average cross-entropy. We used a open-vocabulary model: tokens unseen in the training text were smoothed to a small probability allocated to "unknown" tokens. A further bit of notation: when we say we measured the cross-entropy of X to Y, Y is the training corpus used to estimate the parameters of the distribution model $\mathcal{M}_Y$ used to calculate $H_{\mathcal{M}_Y}(X)$.

First, we wanted to see if there was evidence to support the claim that software was "natural", in the same way that English is natural, *viz.*, whether regularities in software could be captured by language models.

RQ1: *Do n-gram language models capture regularities in software?*

To answer this question, we estimated n-gram models for several values of n over both the English corpus and the 10

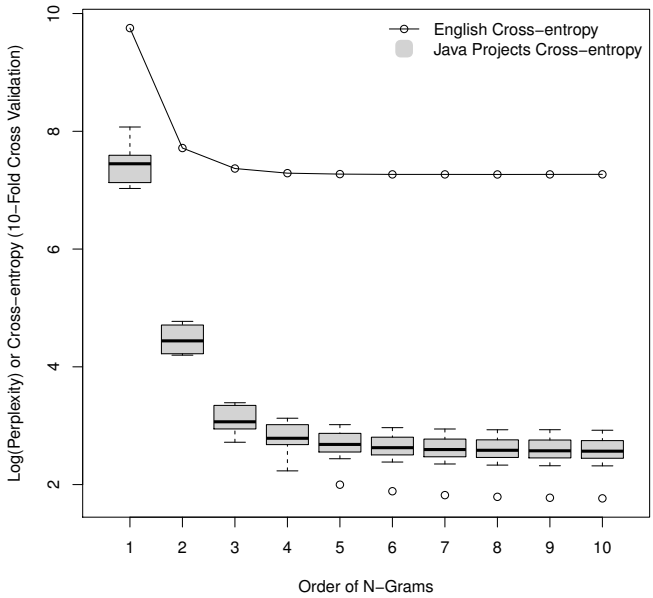


Figure 1. Comparison of English cross-entropy versus the code cross-entropy of 10 Java projects.

Java language project corpora, using averages over 10-fold cross-validation (each corpus to itself) as described above. The results are in Figure 1. The single line at the top is the average, over the 10 folds, of the English corpus, beginning at about 10 bits for unigram models, and trailing down to under 8 bits for *10*-gram models. When you build a model on one project, we call the computation of cross-entropy on test data from that same project *self cross-entropy*. The average self cross-entropy for the 10 Java projects are shown below in boxplots, one for each order from unigram models to 10-gram models. Several observations can be made. Each project was concatenated and viewed as a single document.

First, software unigram entropy is much lower than might be expected from a uniform distribution over unique tokens, because token frequencies are obviously very skewed.

Second, *cross-entropy declines rapidly with n-gram order*, saturating around tri- or 4-grams. The similarity in the decline in English and the Java projects is striking. This decline suggests that *the language model captures as much repetitive local context* in Java programs, as it does in English corpora. We take this to be highly encouraging: the ability to model the regularity of the local context in natural languages has proven to be extremely valuable in statistical natural language processing; we hope (and provide some evidence to support the claim) that this regularity can be exploited for software tools.

Finally, *software is far more regular than English* with entropies sinking down to under 2 bits in some cases.

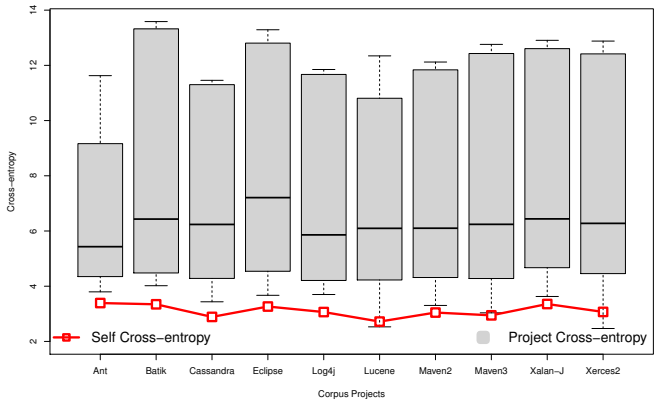


Figure 2. Cross-entropy versus self cross-entropy of the 10 Java projects studied.

> Corpus-based statistical language models capture a high level of local regularity in software, even more so than in English.

This raises a worrying question: is the increased regularity we are capturing in software merely a difference between the English and Java languages themselves? Java is certainly a much simpler language than English, with a far more structured syntax. Might not the lower entropy be simply an artifact of Java's artificial, simple syntax? If the statistical regularity of the local context captured by the language model were simply arising from the simplicity of Java, then we should find this uniformly across all the projects; in particular, if we train a model on one Java project, and test on another, we should successfully capture the local regularity in the language. Thus, we sublimate this anxiety-provoking question into the following:

RQ2: *Is the local regularity that the statistical language model captures merely language-specific or is it also project-specific?*

This is a simple experiment. For each of the 10 projects, we train a trigram model, and evaluate its cross-entropy against each of the 9 others, then compare the result with the average 10-fold self cross-entropy. We chose trigrams because they do not use much memory and use minimal context to produce low cross-entropy. This plot is shown in Figure 2. The x-axis lists all the different Java projects, and, for each, the boxplot shows the range of cross-entropies with the other nine projects. The red line at the bottom shows the average self cross-entropy of the project against itself. In this figure, each document was the concatenation of a project. As can be seen, the self-entropy is always lower. Even for small projects, like Log4J and Maven, the self cross-entropy is low; because it is also obtained by 10-fold cross-validation, there is no risk of over-fitting. Thus, to evaluate self cross-entropy with 10-fold cross-validation, 10% of the lines act as a test document and the corpus is the other 90% of the lines. This suggests that *useful language models can be built even for small code corpora*.

> Language models capture significant levels of local regularity that are *not* an artifact of the programming language syntax, but rather arise from "naturalness" or repetitiveness specific to each project. Furthermore, we have captured this regularity in projects with only about 62K lines of code.

This is noteworthy: it appears each project has its own type of local, non-Java-specific regularity that the mode is capturing; furthermore, the local regularity of each project is *special unto itself*, and different from that of the other projects. Clearly, each project has its own vocabulary, and specific local patterns of iteration, field access, method calls, *etc.* Language models are, therefore, capturing non-Java-specific project regularity beyond the differences in unigram vocabularies. In Section IV, we discuss the application of the multi-token local regularity captured by the models to a completion task. As we demonstrate in that section, the models are able to successfully suggest non-linguistic tokens (tokens that are not Java keywords) about 50% of the time; this also provides evidence that the low entropy produced by the models are not just because of Java language simplicity.

But projects do not exist in isolation; the entire idea of *product-line engineering* rests on the fact that products in similar domains are quite similar to each other. This raises the interesting question:

RQ3: *Do n-gram models capture similarities within and differences between project domains?*

We approached this question by studying categories of applications within Ubuntu, listed in Table I. For each category, we calculated the *self* cross-entropy within the category (red box) and the *other* cross-entropy, the cross-entropy against all the other categories (boxplot), shown in Figure 3. Here again, as in Figure 2, we see that there appears to be a lot of local regularity repeated *within* application domains, and much less so across application domains. Some domains, *e.g.* the Web, appear to have a very high-level of regularity (and lower self-entropy); this is an interesting phenomenon, requiring further study. While larger projects (*i.e.* more data) is better, these results suggest that even new projects can leverage corpora in the same or similar domains.

B. Concluding Discussion

A high degree of local repetitiveness, or regularity, is present in code corpora and captured by n-gram models.

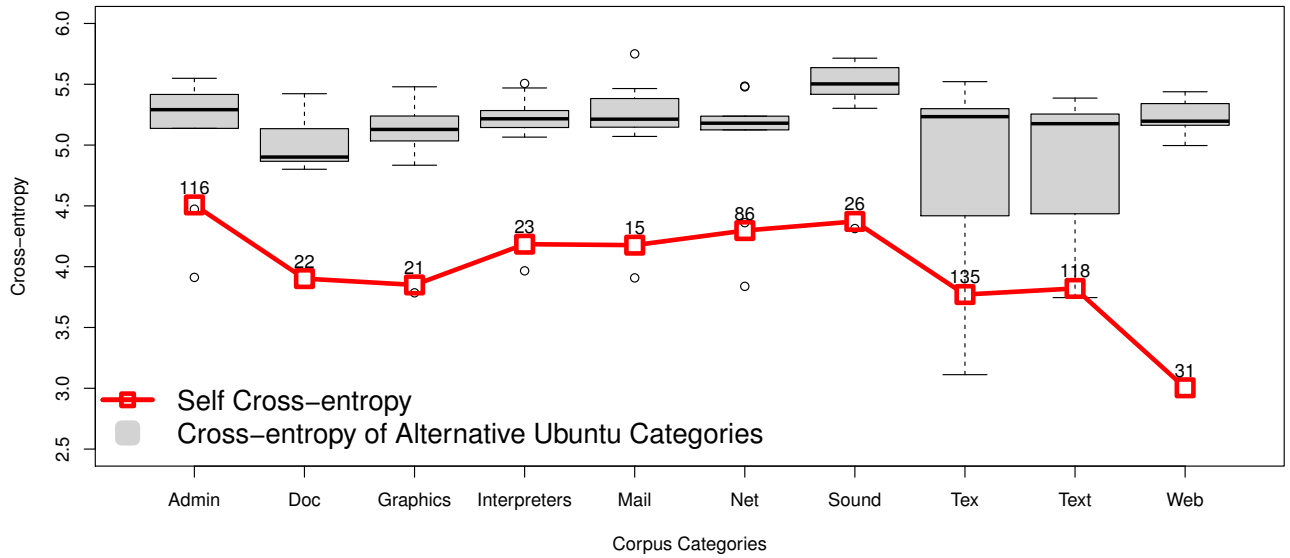


Figure 3. 10 categories of Ubuntu applications cross-entropy (10 categories, 593 total packages).

The data suggest that these local regularities are specific to both projects and to application domains. The data also indicate that these regularities are not simply due to the more regular (when compared to natural languages) syntax of Java, but arise from other types of project- and domain-specific local regularities that exist in the code. Next, we show that these project-specific regularities are actually useful. We exploit project-specific models to extend the Eclipse suggestion engine; we also show that the n-gram models quite often (about 50% of the time) provide suggestions that are project-specific, rather than merely suggesting context-relevant Java keywords.

In natural language, these local regularities have proven to be of profound value for tasks such as translation. It is our belief that these simple local regularities can be used for code summarization and code searching. We also believe that deeper, semantic properties will also, in general, manifest themselves in these same local regularities. These are discussed further in future work, Section VI.

IV. Suggesting the Next Token

The strikingly low entropy (between 3 and 4 bits) produced by the smoothed n-gram model indicates that even at the local token-sequence level, a high degree of "naturalness" obtains. With just 8–16 tries (2^3–2^4) we may very well guess the right next token!

A. Eclipse Suggestion Plug-In

We built an Eclipse plug-in to test this idea. Most modern IDEs, have a built-in *suggestion engine* that suggests a next token whenever it can. Typically, suggestions are based on type information available in context. We conjectured that

Algorithm 1 $\mathcal{MSE}$(esugg, nsugg, maxrank, minlen)

Require: esugg and nsugg are ordered sets of Eclipse and N-gram suggestions.

elong := $\{p \in \text{esugg}[1..\text{maxrank}] \mid \texttt{strlen}(p) > \text{minlen}\}$

if elong $\neq \emptyset$ **then**
  return esugg[1..maxrank]
end if
return esugg$[1..\lceil \frac{\text{maxrank}}{2} \rceil] \circ$ nsugg$[1..\lfloor \frac{\text{maxrank}}{2} \rfloor]$

corpus-based n-gram models suggestion engine (for brevity, $\mathcal{NGSE}$) could enhance Eclipse's built-in suggestion engine (for brevity, $\mathcal{ECSE}$) by offering tokens that tend to *naturally* follow from preceding ones in the relevant corpus.

The $\mathcal{NGSE}$ uses a trigram model built from a project corpus. After each token, $\mathcal{NGSE}$ uses the previous two tokens (the test document), already entered into the text buffer, and attempts to guess the next token, currently based on a static corpus of source code. The language model estimates the probability of a specific choice of next token; this probability can rank order the likely next tokens. Our implementation produces rank-ordered suggestions in less than 0.2 seconds on average. Both $\mathcal{NGSE}$ and $\mathcal{ECSE}$ produce many suggestions, too many to present, so we use a heuristic to merge the lists from the two groups: given an admissible number n of suggestions to be presented to the user, choose n candidates from both $\mathcal{NGSE}$'s and $\mathcal{ECSE}$'s offers.

In general, $\mathcal{NGSE}$ was good at recommending *shorter* tokens, while $\mathcal{ECSE}$ was better at longer tokens (we discuss

the reasons for this phenomenon later in this section). This suggested the simple merge algorithm $\mathcal{MSE}$, defined in Algorithm 1. In our experiments, 7 is the break-even length, after which Eclipse outperforms our n-gram model, so we set minlen $= 6$. Whenever Eclipse offers long suggestions within the top n, we greedily pick all the top n offers from Eclipse; otherwise, we pick half from Eclipse and half from n-grams.

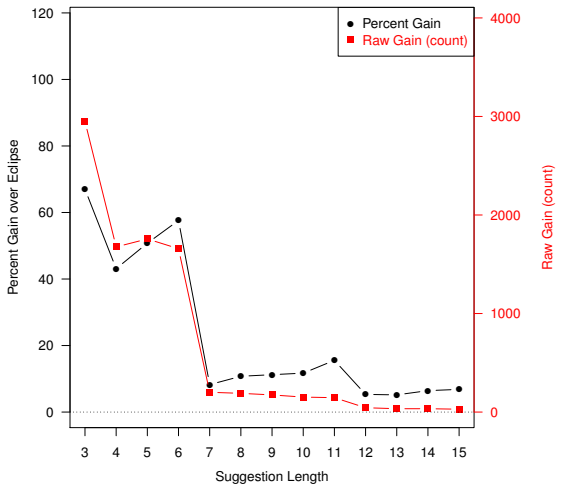
The relative performance of $\mathcal{MSE}$ and $\mathcal{ECSE}$ in practice might depend on a great many factors, and would require a well-controlled, human study to be done at scale. A suggestion engine can present more or fewer choices; it may offer all suggestions, or only offer suggestions that are long. Suggestions could be selected with a touch-screen, with a mouse, or with a down-arrow key. Since our goal here is to gauge the power of corpus-based language models, as opposed to building the most user-friendly merged suggestion engine (which remains future work) we conducted an automated experiment rather than a human-subject study.

We controlled for 2 factors in our experiments: the string length of suggestions l, and the number of choices n presented to the user. We repeated the experiment varying n, for $n = 2, 6, 10$ and l, for $l = 3, 4, 5, \ldots 15$. We omitted suggestions less than 3 characters, as not useful. Also, when merging two suggestion lists, we chose to pick at least one from each, and thus $n \geq 2$. We felt that more than 10 choices would be overwhelming—although our findings do not change very much at all even with 16 and 20 choices. We choose 5 projects for study: *Ant, Maven, Log4J, Xalan*, and *Xerces*. Each project was a mature Apache foundation Java project used adopted by many Java open-source projects.

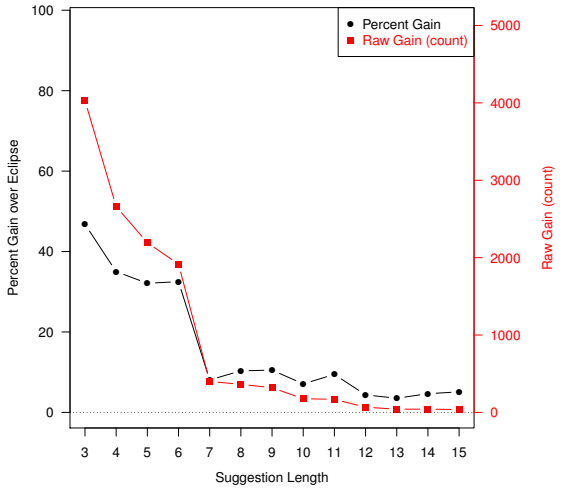
In each project, we set aside a test set of 40 randomly chosen files (200 set aside in all) and built a trigram language model on the remaining files for each project. Trigrams are chosen because they use sufficient context (2 tokens), they use less memory than 4-grams or 5-grams to represent, and trigrams represent an inflection point where increasing the order of n for n-grams results in a decreasing reduction of cross-entropy (see Figure 1). We then used the $\mathcal{MSE}$ and $\mathcal{ECSE}$ algorithms to predict every token in the 40 set-aside files, and evaluated how many *more* times the $\mathcal{MSE}$ made a successful suggestion, when compared to the basic $\mathcal{ECSE}$. We do not report precision or recall since there is usually only one correct suggestion. 40 files were chosen to reduce run-time while maintaining necessary statistical significance and power. In each case, we evaluated the advantage of $\mathcal{MSE}$ over $\mathcal{ECSE}$, measured as the (percent and absolute) gain in number of correct suggestions at each combination of factors n and l.

B. How Does the Language Model Help?

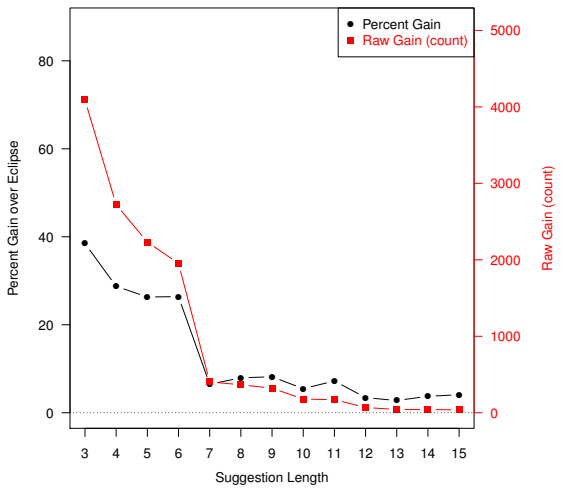
Figure 4 shows the results. Note the two y-scales: the left side (black circle points) is percent additional correct



(a) Gain using top 2 suggestions.



(b) Gain using top 6 suggestions.



(c) Gain using top 10 suggestions.

Figure 4. Suggestion gains from merging n-gram suggestions into those of Eclipse.

suggestions, and the right side (red square points) are the raw counts. Since the raw count of successful suggestions from the Eclipse engine $\mathcal{ECSE}$ also declines with length, both measures are useful. As can be seen $\mathcal{MSE}$ provides measurable advantage over $\mathcal{ECSE}$ in all settings of both factors, though the advantage generally declines with l. The gains up through 6-character tokens are quite substantial, in the range of 33–67% additional suggestions from the language model that are correct; between 7 and 15 characters, the gains range from 3–16%.

The additional suggestions from $\mathcal{NGSE}$ run the gamut, including methods, classes and fields predictable from frequent trigrams in the corpus (*e.g.*, `println`, `iterator`, `transform`, `IOException`, `append`, `toString`, `assertEquals`), package names (*e.g.*, `apache`, `tools`, `util`, `java`) as well as language keywords (*e.g.*, `import`, `public`, `return`, `this`). An examination of the tokens reveals why the n-grams approach adds most value with shorter tokens. The language model we build is based on *all* the files in the system, and the most frequent n-grams are those that occur frequently in all the files. In the corpus, we find that more frequently used tokens have shorter names; naturally these give rise to stronger signals that are picked up by the n-gram language model. Note that a significant portion, *viz.*, 50% of the successful suggestions are **not** Java keywords guessed from language context—they are project-specific tokens. Statistical language models thus capture *non-language-specific, local regularity* in each project.

In the table below, we present another view of the benefit of $\mathcal{MSE}$; the total number of keystrokes saved by using the base $\mathcal{ECSE}$, (first row) the $\mathcal{MSE}$ (second row) and the percent gain from using $\mathcal{MSE}$.

	Top 2	Top 6	Top 10
$\mathcal{ECSE}$	42743	77245	95318
$\mathcal{MSE}$	68798	103100	120750
Increase	61%	33%	27%

Finally, here we used *one specific* language model to enhance *one specific* software tool, a suggestion engine. With more sophisticated language models, specifically ones that combines syntax, scoping and type information, we expect to achieve even lower entropy, and thus better performance in this and other software tools.

V. Related Work

A. Code Completion and Suggestion

By *completion* we mean the task of completing a partially typed-in token; by *suggestion* we mean suggesting a complete token. The discussion above concerned *suggestion* engines.

Modern IDEs provide both code completion and code suggestion, often with a unified interface. Two notable Java-based examples are Eclipse and IntelliJ IDEA. Both draw possible completions from existing code, but they operate quite differently from our completion prototype.

Eclipse and IDEA respond to completion requests (a keyboard shortcut such as ctrl+space) by conservatively deducing what tokens "might apply" in the current syntactic context. Eclipse and IDEA implement dozens of "syntactic and semantic completion rules" that are primarily guided by the Java Language Specification. For example, both Eclipse and IDEA first parse the surrounding code and infer the current *context*. They then create a short list of *expected token types*. If this list contains, say, a reference type, the tools use the rules of the type system to add a list of applicable type-names to the list of completions. Similarly, if a variable is expected, the tools list visible names from the symbol table. As a final step, both tools rank the completions with a collection of apparently hand-coded heuristics.

We complement this approach. Rather than using language semantics and immediate context to guess what *might* apply, our n-gram model captures what *most often does* apply. Our approach is much more flexible, since it is language-independent and tolerates inchoate code. It also has the potential to be much more precise: the space of commonly used completions is naturally far smaller than the space of "language-allowed" completions. Note that our approach *complements* the current IDE approach: language-based guesses can be enhanced (or ordered) using corpus statistics. It is noteworthy that perhaps some of the strongest evidence for the "naturalness" of software is how much our n-gram-based suggestion engine improves Eclipse's language-based engine (Section IV).

There are approaches arguably more advanced than those currently available in IDEs. The BMN completion algorithm of Bruch *et al.* [5] is focused on finding the most likely *method calls* that could complete an expression by using frequency of method calls and prior usage in similar circumstances. We propose a broad vision for using language models of code corpora in software tools. Our specific illustrative completion application has a broader completion goal, completing all types of tokens, not just method calls. Later, Bruch *et al.* [6] lay out a vision for next-generation IDEs that take advantage of "collective wisdom" embodied in code bodies and recorded human action. We enthusiastically concur with this vision; our specific approach is that "natural software" has statistical regularities that allow techniques from statistical NLP to be profitably applied in the endeavor to make this vision a reality.

Robbes and Lanza [7] compare a set of different method-call and class-name completion strategies, which start with a multi-*letter* prefix. They introduce an approach based on history and show that it improves performance. Our

approach is complementary to theirs: it can provide full-token completion of any token and is based a language model that exploits regularities in program corpora. Han *et al.* [8] uses Hidden Markov Models (HMM) to infer likely tokens from short form tokens. They make use of a dynamic programming trellis approach for backtracking and suggestion. Their HMM is in fact a language model, but the paper does not describe how effective a model it is or how well it would perform for completion tasks without user-provided abbreviations.

Jacob and Tairas [9] used n-gram language models for a different application: to find matching code clones relevant to partial programming task. Language models were built over clone groups (not entire corpora as we propose) and used to retrieve and present candidate clones relevant to a partially completed coding task.

Hou and Pletcher [10] propose and evaluate several strategies for improving Eclipse's standard code completions. They focus their effort on one specific class of Eclipse's completions, *method calls*, and they find that ranking calls by *frequency of past use* is effective. Both this work and our own completion prototype drive completions with usage data, but our work is more general: we use a much more general language model to predict and complete arbitrary code, not solely method calls, and we propose many other potential applications as well.

B. The "Naturalness" of Names in Code

This line of work aims to automatically evaluate if "the names reflect the meanings of the artifacts, and, if not, how could they be improved [11, 12]?" Work by Høst and Østvold [13, 14] also concerns method naming: they combine static analysis with an entropy-based measure over the distribution of simple semantic properties of methods in a corpus to determine which method names are most discriminatory, then use it to detect names whose usage are inconsistent with the corpus. This work does not use language models to capture repetition in code.

C. Summarization and Concern Location

This line of work aims to generate natural language descriptions (summaries) of code [15–17]. This work uses semantic properties of code derived by *static analysis*, rather than using statistical models of the "natural" regularities of code. It is complementary to ours: properties derived by static analysis (as long as they can be done efficiently, and at scale) could enrich statistical models of large software corpora. Another line of work seeks to locate parts of code relevant to a specified concern (*e.g.* "place auction bid"), which could be local or cross-cutting, based on fragments of code names [18], facts mined from code [19], or co-occurrence of related words in code [20].

D. Software Mining

Work in this very active area [21] aims to mine *useful information* from software repositories. Many papers can be found in MSR conference series at ICSE, and representative works include mining API usages [22, 23], patterns of errors [24, 25], topic extraction [26], guiding changes [27] and several others. The approaches used vary. We argue that the "naturalness" of software provides a *conceptual perspective* for this work, and also offers some novel *implementation* approaches. The conceptual perspective rests on the idea useful information is often manifest in software in uniform, and uncomplicated ways; the implementation approach indicates that the uniform and uncomplicated manifestation of useful facts can be determined from a large, representative software corpus in which the required information is already known and annotated. This corpus can be used to estimate the statistical relationship between the required information and readily observable facts; this relationship can be used to reliably find similar information in new programs similar to the corpus. We explain this further in future work (Sections VI-C and VI-D).

VI. Future Directions

We present now possible applications of corpus-based statistical methods to aid software engineering tasks.

A. Improved Language Models

In this paper, we exploited a common language model (n-grams) that effectively captures local regularity. There are several avenues for extension. Existing, very large bodies of code can be readily parsed, typed, scoped, and even subject to simple semantic analysis. All this data can be modeled using enhanced models to capture regularities that exist at syntactic, type, scope, and semantic levels.

There is a difficulty here: the richer a model, the more data is needed to provide good estimates for the model parameters; thus the risk of data sparsity grows as we enrich our models. Ideas analogous to the smoothing techniques used in n-gram models will have to be adapted and applied to build richer models of software corpora. Still, if these models do capture regularities, they may then be employed for software engineering tasks, some of which we discuss below.

B. Language Models for Accessibility

Some programmers have difficulty using keyboards, because of RSI or visual impairment. There has been quite a bit of work on aiding such programmers using speech recognition (*e.g.*, [28–30]). However, these approaches suffer from fairly high recognition error rates and are not widely used [31]. None of the published approaches make use of a statistical language model trained on specific code corpora. We hypothesize that the use of a language model

can significantly reduce the error rates; they certainly play a crucial role in conventional speech recognition engines. Because a large proportion of development work occurs in a maintenance or re-engineering context, language models derived from existing code should improve the performance of these speech recognition systems. Even when only a small amount of relevant code exists, language model adaptation techniques [32] could be applied, using corpora of similar code.

C. Summarizing and/or Retrieving Code

Consider the task of summarizing code fragments or code changes in English. Consider also the approximate reverse task: finding/retrieving a relevant fragment of code (*e.g.* method call) given an English description. We draw an analogy between these two problems and *statistical natural language translation*, ($\mathcal{SNLT}$). Code and English are two languages, and essentially both the above are *translation* tasks. $\mathcal{SNLT}$ relies on access to an *aligned corpus*, which is a large set of sentences simultaneously presented in two or more languages (*e.g.*., proceedings of parliaments in Canada and Europe). Consider the problem of translating a Tamil sentence T to an English sentence E. The translation process is primed using an aligned English-Tamil corpus: one estimates, using the aligned corpus of $E-T$ pairs, the conditional distribution (using a Bayesian formulation) of English output sentences E given Tamil sentences T. The translation process calculates the most likely sentence E given a specific T.

We propose to tackle the summarization/retrieval task using statistical estimates derived from several corpora. First, we use an aligned (English/Code) corpora built from multiple sources: one source arises from the version history of a program. Each commit in a typical project offers a matched pair of a log message (English), and some changes (Code). Another source of aligned examples are in-line comments that are clearly matchable with nearby code [33]. Second, we can use any available English language text associated with a given project, including code comments, code, design documents, bug reports, discussions on mailing lists, to build a relevant English corpus. Finally, models of the code and the associated English can be used select "most likely" translations.

D. Software Tools

We hypothesize that the "naturalness" of software implies a "naturalness" of deeper properties of software, such as those normally computed by powerful but expensive software tools; as programmers tend towards repetitive use of code idioms, we hypothesize that deeper, more semantic properties of programs *also* manifest themselves in programs in superficially similar ways. More specifically, we hypothesize that semantic properties *usually* manifest themselves in *superficial* ways that are computationally cheap to detect, particularly when compared to the cost (or even infeasibility) of determining these properties by sound (or complete) static analysis.

For example, the use of unprotected string functions like `strcat` (as opposed to `strncat`) is evidence for a potential buffer flow, but not conclusive proof. As another example, suppose 3 related methods (wherein the "relatedness" has been detected using a recommender system [34, 35]) `open, access, close` are called together in the same method, with the 3 methods occurring in that textual order in the code, and `access` occurring within a loop. This is evidence (albeit not conclusive) that the 3 methods are to be used with the protocol `open-access*-close`. These are heuristics, analogous to the probabilistic constraints used in Merlin (See Livshits *et al.* [36], Figure 3). But where do they come from? In Merlin, they are hard-coded heuristics based on researchers' intuitions; we argue that they should be derived from corpus-level distribution models, that make use of prior knowledge about protocols already known to be used within those corpora.

This admittedly is a leap of faith; however, if it holds up (and we have found anecdotal evidence that it does, and some prior research implicitly makes a version of this assumption [36]) one can leverage this notion to build simple, scalable, and effective approximations in a wide variety of settings. We contend that the annotation of code corpora, for instance API usage rules, can be automated using data mining techniques. Manually annotated code corpora may be well-worth the investment (by analogy with the Penn Tree Bank [37]) and can be constructed using a volunteer community, or perhaps via market mechanisms like the Mechanical Turk [38].

VII. Conclusion

Although Linguists (sometimes) revel in the theoretical complexities of natural languages, most "natural" utterances, in practice, are quite regular and predictable and can in fact be modeled by rigorous statistical methods. This fact has revolutionized computational linguistics. We offer evidence supporting an analogous claim for software: *though software in theory can be very complex, in practice, it appears that even a fairly simple statistical model can capture a surprising amount of regularity in "natural" software*. This simple model is strong enough for us to quickly and easily implement a fairly powerful suggestion engine that already improves a state-of-the-art IDE. We also lay out a vision for future work. Specifically, we believe that natural language translation approaches can be used for code summarization and code search in a symmetric way; we also hypothesize that the "naturalness" of software implies a sort of "naturalness" of deeper properties of software, such as those normally computed by powerful, traditional software analysis tools. These are challenging tasks, but with potentially high pay-off, and we hope others will join us in this work.

REFERENCES

[1] K. Sparck Jones, "Natural language processing: a historical review," *Current Issues in Computational Linguistics: in Honour of Don Walker (Ed Zampolli, Calzolari and Palmer), Amsterdam: Kluwer*, 1994.

[2] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings, ACM SIGSOFT FSE*. ACM, 2010, pp. 147–156.

[3] P. Koehn, *Statistical Machine Translation*. Cambridge University Press, 2010.

[4] C. Manning, H. Schütze, and MITCogNet, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 59.

[5] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings, ACM SIGSOFT ESEC/FSE*, 2009.

[6] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini, "IDE 2.0: collective intelligence in software development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 53–58.

[7] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[8] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proceedings, ASE*. IEEE Computer Society, 2009, pp. 332–343.

[9] F. Jacob and R. Tairas, "Code template inference using language models," in *Proceedings of the 48th Annual Southeast Regional Conference*, 2010.

[10] D. Hou and D. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion," in *Proceedings, ICSM*, 2011.

[11] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," *Proceedings, ICPC*, 2006.

[12] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," in *Proceedings, MSR*. ACM, 2011.

[13] E. W. Hø st and B. M. Ø stvold, "Software language engineering," D. Gašević, R. Lämmel, and E. Wyk, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. The Java Programmer's Phrase Book.

[14] E. Høst and B. Østvold, "Debugging method names," in *Proceedings, ECOOP*. Springer, 2009, pp. 294–317.

[15] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings, ASE*, 2010.

[16] R. Buse and W. Weimer, "Automatically documenting program changes," in *Proceedings, ASE*. ACM, 2010, pp. 33–42.

[17] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings, ICSE*, 2011.

[18] D. Shepherd, Z. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings, AOSD*. ACM, 2007, pp. 212–224.

[19] S. Rastkar, G. Murphy, and A. Bradley, "Generating natural language summaries for cross-cutting source code concerns," in *Proceedings, ICSM*, 2011.

[20] D. Shepherd, L. Pollock, and T. Tourwé, "Using language clues to discover crosscutting concerns," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–6.

[21] T. Xie, S. Thummalapenta, D. Lo, and C. Liu, "Data mining for software engineering," *IEEE Computer*, vol. 42, no. 8, pp. 35–42, 2009.

[22] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *Proceedings, ACM SIGSOFT FSE*. ACM, 2008, pp. 339–349.

[23] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 48–61.

[24] B. Livshits and T. Zimmermann, "DynaMine: finding common error patterns by mining software revision histories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 296–305, 2005.

[25] S. Kim, K. Pan, and E. Whitehead Jr, "Memories of bug fixes," in *Proceedings, ACM SIGSOFT FSE*. ACM, 2006, pp. 35–45.

[26] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[27] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings, ICSE*. IEEE Computer Society, 2004, pp. 563–572.

[28] S. Arnold, L. Mark, and J. Goldthwaite, "Programming by voice, VocalProgramming," in *Proceedings, ACM Conf. on Assistive technologies*. ACM, 2000, pp. 149–155.

[29] A. Begel, "Spoken Language Support for Software Development," in *Proceedings, VL/HCC*. IEEE Computer Society, 2004, pp. 271–272.

[30] T. Hubbell, D. Langan, and T. Hain, "A voice-activated syntax-directed editor for manually disabled programmers," in *Proceedings, ACM SIGACCESS*. ACM, 2006.

[31] S. Mills, S. Saadat, and D. Whiting, "Is voice recognition the solution to keyboard-based RSI?" in *Automation Congress, 2006. WAC '06. World*, 2006.

[32] J. Bellegarda, "Statistical language model adaptation: review and perspectives," *Speech Communication*, vol. 42, no. 1, pp. 93–108, 2004.

[33] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, pp. 970–983, 2002.

[34] Z. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Proceedings, ACM SIGSOFT ESEC/FSE*. ACM, 2007, pp. 15–24.

[35] M. Robillard, "Automatic generation of suggestions for program investigation," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 11–20.

[36] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," in *ACM SIGPLAN Notices*, vol. 44, no. 6. ACM, 2009, pp. 75–86.

[37] M. Marcus, M. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[38] A. Kittur, E. Chi, and B. Suh, "Crowdsourcing user studies with Mechanical Turk," in *Proceedings, CHI*. ACM, 2008.