

On the Naturalness of Software

By Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu

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 regular, 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 completion capability. We conclude the paper by laying out a vision for future research in this area.

1. INTRODUCTION

Of all that we do, *communicating* is the one action that is arguably the most essentially human. Honed by millions of years of cultural and biological evolution, language and communication are an ordinary, instinctive part of everyday life (see Pinker³⁹). Although “natural” languages such as English and Tamil accommodate exquisitely complex forms of expression, everyday human communication evolved in settings impaired by noise and impelled by need; thus, it is most often simple, expedient, even repetitive. This “naturalness” of quotidian human linguistic behavior, together with large online corpora of utterances, modern computing resources, and statistical innovations, has led to a revolution in *natural-language processing*, whose fruits we enjoy every day in the form of speech recognition in mobile devices and automated language translation in the cloud, *inter alia*.

However, the field of natural language processing (“NLP,” see Sparck-Jones⁴⁶ 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 1960's. In the 1970s and 1980s, 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*, that is, 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,^a 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.^b

Can we apply these techniques *directly* to software, with its strange syntax, awash with punctuation, and replicate this success? The funny thing about programming is that it is as much *an act of communication*, from one human to another, as it is a way to tell computers what to do. Knuth said as much, 30 years ago:

*Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*²³

If one, then, were to view programming as an act of communication, is it driven by the “language instinct”? Do we program as we speak? Is our code largely simple, repetitive, and predictable? *Is code natural?*

This, precisely, is our central hypothesis:

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

We believe that repetitiveness occurs in code corpora at *lexical*, *syntactic*, and *semantic* levels, both locally and globally, and we believe that this phenomenon can be captured and exploited using modern methods of computational

^a This included the Canadian Hansard (parliamentary proceedings), and similar outputs from the European parliament.

^b Indeed, a renowned pioneer of the statistical approach, Fred Jelinek, 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.

The original version of this paper was published in the *Proceedings of the 34th International Conference on Software Engineering* (2012), 837–847. All five authors were at UC Davis at the time.

data analysis. We believe 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 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 above model is indeed capturing 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 already 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.

2. MOTIVATION & 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 in the paper.

Consider a speech recognizer, receiving a (noisy) radio broadcast, such as *In Berlin today, Chancellor Angela <radio buzz> announced...* A good speech recognizer might guess that the buzzed-out word was *Merkel* 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 of this style, it is quite possible to fill in the blanks. Thus, if we hear the word “Chancellor Angela,” we can expect that, *in most cases*, the next word is “Merkel.” This fact is well-known and exploited by speech recognizers, natural language translation devices, and even some optical character recognition (OCR) 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,¹³ 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. Our intuition essentially amounts to the following: *We*

can use a code corpus to estimate the probability distribution of code fragments. If this distribution has low-entropy, and we can model it well: then, given a partial fragment of code (*qua* partial token sequence, partial AST, or partial semantic abstraction) we can often, with high confidence, guess the rest.

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.” Language models in NLP can be lexical (token sequences), syntactic (grammatical structures), or semantic models (probabilistic or vector-space representations of meaning). Markovian models of token sequences are the simplest, and we use them as an illustration.

2.1. Language models

A language model assigns a probability to an utterance. For us, “utterances” are programs. More formally, consider a set of allowable program tokens^c \mathcal{T} , and the (over-generous) 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 $p(\cdot)$ over systems $s \in S$, that is,

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

In practice, given a corpus C of programs $C \subseteq S$, and a suitably chosen parametric distribution $p(\cdot)$, we attempt to calculate a maximum-likelihood estimate of the parameters of this distribution; this gives us an estimated language model. A great many language models, based on phenomena at lexical, syntactic and semantic levels have been developed (see the Manning and Schütze³¹ book). The choice of a language model is usually driven by practicalities: how easy is it to estimate and how useful it is. For these reasons, the most ubiquitous is the n -gram model, which we now use to illustrate our thesis.

Illustration: n -gram models Consider $a_1 a_2 \cdots a_i \cdots a_n$, the sequence of tokens in a document (in our case, source code for a system s). We can statistically model how likely tokens are to follow other tokens. Thus, we can estimate the probability of a document based on the product of a series of conditional probabilities:

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

These n -gram models assume a *Markov property*, that is, token occurrences are influenced only by a limited prefix of length $n-1$, thus for 4-gram models, we assume

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

These models are estimated from a corpus using simple maximum-likelihood based frequency-counting of token sequences. Thus, if “*” is a wildcard, we ask, how relatively often are the tokens a_1, a_2, a_3 followed by a_4 :

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

^c Here we use a token to mean its lexeme.

In practice, estimation of n -gram models is quite a bit more complicated. The main difficulties arise from data sparsity, that is, 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; but even the biggest systems only have $O(10^8)$ tokens. Thus, some trigrams may never occur in one corpus, but may in fact occur elsewhere. This can cause technical difficulties; when we encounter a previously unseen n -gram, we are, in principle, “infinitely surprised,” because an “infinitely improbable” event x estimated from the previously seen corpus to have $p(x) = 0$ actually occurs. *Smoothing* is a technique to handle cases where we have not seen the n -grams yet and still produce usable results with sufficient statistical rigor. 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 Kneser-Ney smoothing (e.g., Koehn,²⁴ Section 7) gives good results for software corpora. However, we note that these are very early efforts in this area, and new software language models^{35, 49} and estimation techniques have improved on the results presented below.

But how do we know when we have a good language model?

2.2. 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 good confidence the contents of a new document drawn from the same population. A better model can guess the contents of the new document with higher probability. In other words, the model will find a new document with “typical” content not particularly surprising, or very “perplexing.” In NLP, this idea is captured by a measure called *perplexity*, or its log-transformed version, *cross-entropy*.^d 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 2.1:

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

This is a measure of how “surprised” a model is by the given document. A good model estimates higher probabilities (closer to 1, and thus lower absolute log values) to most words in the document. 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 as “natural” (i.e., unsurprising) as natural language?

3. METHODOLOGY & FINDINGS

To explore these questions, 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 gain further 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.^e For code, we used several sets of corpora, including a collection of Java projects, as well a collection of applications from Ubuntu, broken up into application domains. All are listed in Table 1.

After removing comments, the C and Java project source code was lexically analyzed to produce token sequences that were used to estimate n -gram language models. While

^e We retrieved these corpora from <http://www.nltk.org/>.

Table 1. The 10 Java Projects, C code from 10 Ubuntu 10.10 Categories, and 3 English Corpora we used in our study.

Java Project	Version	Lines	Tokens	
			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
Ubuntu Domain	Version	Lines	Tokens	
			Total	Unique
Admin	10.10	9092325	41208531	1140555
Doc	10.10	87192	362501	15373
Graphics	10.10	1422514	7453031	188792
Interpreters	10.10	1416361	6388351	201538
Mail	10.10	1049136	4408776	137324
Net	10.10	5012473	20666917	541896
Sound	10.10	1698584	29310969	436377
Tex	10.10	1405674	14342943	375845
Text	10.10	1325700	6291804	155177
Web	10.10	1743376	11361332	216474
English Corpus	Version	Lines	Tokens	
			Total	Unique
Brown	20101101	81851	1161192	56057
Gutenberg	20101101	55578	2621613	51156

^d http://en.wikipedia.org/wiki/Cross_entropy; see also Ref. 31, Section 2.2, p. 75, Equation (2.50).

English is the concatenation of Brown and Gutenberg. Ubuntu 10.10 Maverick was released on 2010/10/10.

our experiments were in C and Java, extending to other languages is trivial, and indeed subsequent work with Python yielded similar results.

The Java projects were our central focus; we used them both for cross-entropy studies and experiments with a language model-based code suggestion plug-in for Eclipse. Table 1 describes the 10 Java projects that we used. *Version* indicates the date of the last revision in the Git repository when we cloned the project. *Lines* is calculated using Unix *wc* on all files within each repository; tokens are extracted from each of these files. *Unique Tokens* refers to the number of distinct tokens that make up the total token count given in the *Tokens* field. For English, the unique token count corresponds to the vocabulary size; for code, this count comprises all the identifiers, keywords and operators.

The Ubuntu application domains 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, as it gives a rough indication of the potential “surprisingness” of the project corpus. If these unique token were uniformly distributed throughout the project (which would be very odd indeed), we could expect a cross-entropy of $\log_2(1.15E6)$, or approximately 20 bits. A similar calculation for the Java projects ranges from about 13 bits to about 17 bits.

3.1. Cross-entropy of code

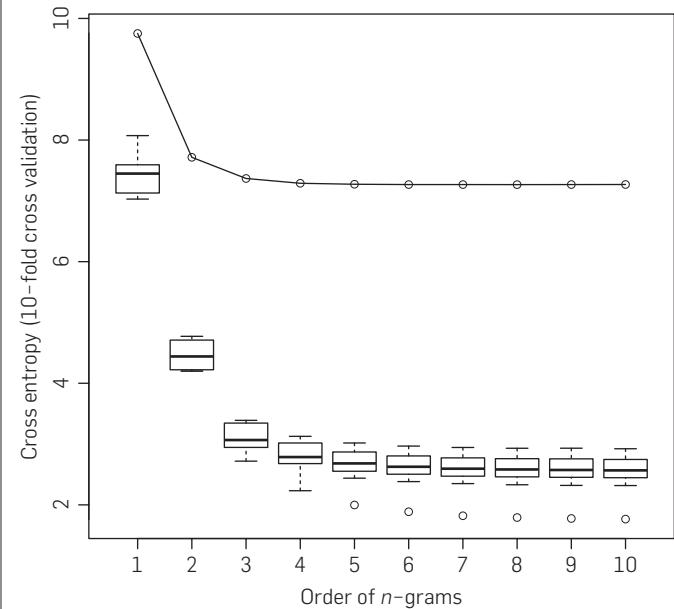
Cross-entropy measures how surprising a test document is to a language model estimated from a corpus. To test a corpus against itself, one must set aside some portion of the corpus for testing, and estimate (train) the model on the rest of the corpus. In all our experiments, we measured cross-entropy by averaging over a 10-fold cross-validation: we split the corpus 90–10% (in lines) at random, trained on the 90%, tested on 10%, and measured the average cross-entropy. 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 M_Y used to calculate the cross-entropy, denoted $H_{M_Y}(X)$.

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.

RQ 1: *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 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 above is the average over the 10 folds for the English corpus, beginning at about 10 bits for unigram models, and trailing down to under 8 bits for 10-gram models. The average self cross-entropy for the 10 projects are shown in boxplots, one for each order from unigram models to 10-gram models. Several observations can be made. First, software unigram entropy is much lower than might be expected from a uniform distribution over unique tokens, because token frequencies are

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



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 Java corpora, like English corpora, contain a good deal of *locally repetitive text*, and this repetition is *effectively captured by the language model*. We take this to be highly encouraging: the ability to model local repetitions in natural languages has proven to be extremely valuable in statistical NLP; we hope that this regularity can be exploited for software tools (and build an NLP-based code suggestion tool whose success supports the claim).

Last, but not least, *software is far more regular than English* with entropies sinking down to under 2 bits in some cases.

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

This raises an important question: *Is the increased regularity we are capturing in software merely a difference between English and Java?* 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 the artificially simple syntax for Java? If the statistical regularity of the local context being captured by the language model were arising solely 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 reformulate the above question into the following:

RQ 2: *Is the local regularity that the statistical language model captures merely language-specific or is it also project-specific?*

For each of the 10 projects, we train a trigram model, and

evaluate its cross-entropy with each of the 9 others, and compare the value with the average 10-fold cross-entropy against itself. 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. As can be seen the self-entropy is always lower.

The language models capture a significant level of local regularity that is *not* an artifact of the programming language syntax, but rather arising from “naturalness,” or regularity, specific to each project.

This is a noteworthy result: it appears each project has its own type of local, non-Java-specific regularity that is being captured by the model; furthermore, the local regularity of each project is *special unto itself*, and different from that of the other projects. Most software engineers will find this intuitive: each project has its own vocabulary, and specific local patterns of iteration, field access, method calls, *etc.* It is important to note that the models are capturing non-Java-specific project regularity beyond simply the differences in unigram vocabularies. In Section 4, 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 another interesting question:

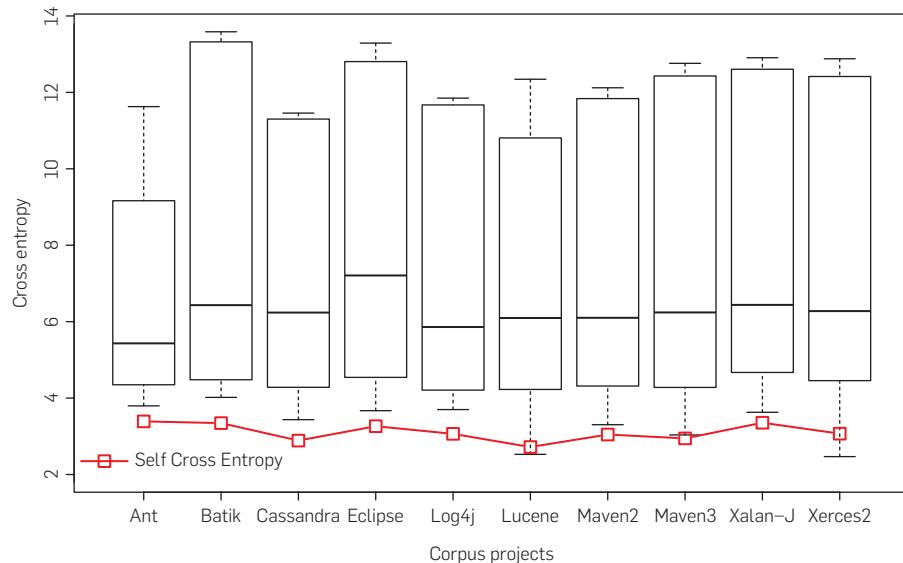
RQ 3: *Do n*-gram models capture similarities within and

differences between application domains?

We approached this question by studying application domains within Ubuntu. Table 1 lists the 10 the application domains we selected; these domains each contained a sizeable number of projects. The domains (with number of projects in each domain) are Administration (116), Documentation (22), Graphics (21), Interpreters (23), Mail (15), Networking (86), Sound/Audio (26), Tex/Latex related (135), Text processing (118), and Web (31). For each domain, we calculated the *self* cross-entropy within the domains, and the *other* cross-entropy, the cross-entropy of the domain against all the others. Cross-domain entropy (median ranging from 5 to 5.5 bits) is consistently and strongly higher, about a bit in most cases, than within-domain entropy (3–4.5 bits). Here again, as in the within-project versus between-project case, 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, for example, the Web, appear to have much greater with-domain regularity (over 2 bits lower cross-entropy); this phenomenon merits further study.

Discussion We have seen that a high degree of local regularity is present in code corpora and, furthermore, that *n*-gram models effectively capture these local regularities. We find evidence suggesting that these regularities are specific to both projects and to application domains. We also find evidence that these regularities arise not merely from the relatively more regular (when compared to natural languages) syntax of Java, but also arise from other types of project- and domain-specific local regularities that exist in the code. In the next section, we demonstrate that these project-specific regularities are useful, by using project-specific models to extend the Eclipse suggestion engine; we also show that the *n*-gram models often (about 50% of the time) provide suggestions that are project-specific, rather than merely suggesting Java keywords guessable from context.

Figure 2. Ten projects cross-entropy versus self cross-entropy.



In natural language, these local regularities have proven to be of profound value, for tasks such as translation. It is our belief that these regularities can be used for code summarization and code searching; we also believe that deeper, semantic properties will also often manifest themselves in local regularities. These are discussed further in future work section (Section 6).

4. 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, there is a high degree of “naturalness.” If we make 8–16 guesses (2^3 – 2^4) as to what the next token is, we may very well guess the right one!

Eclipse Suggestion Plug-in We built an Eclipse plug-in to test this idea. Eclipse, like many other IDEs, has a built-in *suggestion* engine that suggests a next token whenever it can. Eclipse (and other IDEs) suggestions are typically based on type information available in context. We conjectured that 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 the corpus of a lexed project. At every point in the code, \mathcal{NGSE} uses the previous two tokens, already entered into the text buffer, and attempts to guess the next token. The language model built from the corpus gives maximum likelihood estimates of the probability of a specific choice of next token; this probability can be used to rank order the likely next tokens. Our implementation produces rank-ordered suggestions in less than 0.2 s on average. Both \mathcal{NGSE} and \mathcal{ECSE} produce many suggestions. Presenting all would be overwhelming. Therefore we defined a heuristic to merge the lists from the two groups: given an admissible number n of suggestions

Algorithm 1. \mathcal{MSE} (eprops , nprops , $\mathit{maxrank}$, minlen)

Require: eprops and nprops are ordered sets of Eclipse and N-gram proposals.

```

elong := { $p \in \mathit{eprops}[1..\mathit{maxrank}] \mid \text{strlen}(p) > 6$ }
if elong ≠  $\emptyset$  then
    return  $\mathit{eprops}[1..\mathit{maxrank}]$ 
end if
return  $\mathit{eprops}\left[1..\left\lceil \frac{\mathit{maxrank}}{2} \right\rceil\right] \circ \mathit{nprops}\left[1..\left\lceil \frac{\mathit{maxrank}}{2} \right\rceil\right]$ 

```

to be presented to the user, choose n candidates from both \mathcal{NGSE} ’s and \mathcal{ECSE} ’s offers.

We observed that in general, \mathcal{NGSE} was good at recommending *shorter* tokens, and \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. Whenever Eclipse offers long tokens (we picked 7 as the break-even length, based on observation) within the top n , we greedily pick all the top n offers from Eclipse. Otherwise, we pick half from Eclipse and half from our n -gram model.

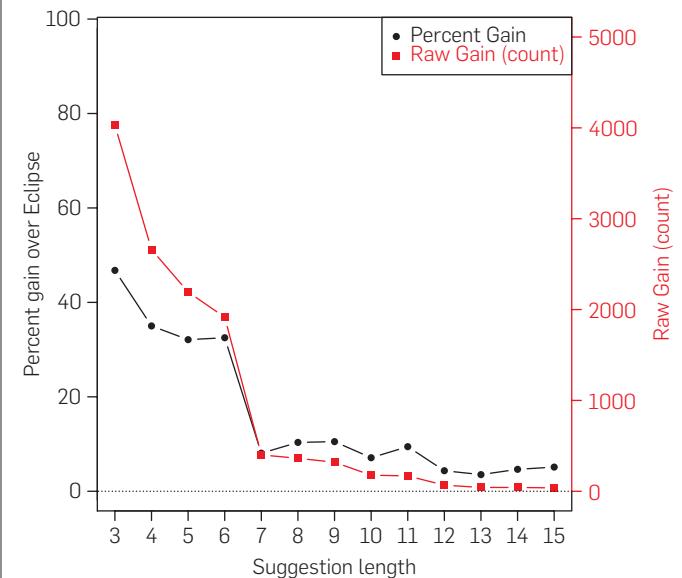
The relative performance of \mathcal{MSE} and \mathcal{ECSE} in actual practice might depend on a great many factors, and would

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 $l = 3, 4, 5, \dots, 15$. 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. 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. 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} . 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 .

How does the language model help? Figure 3 presents the results for the 6-suggestion case. The figure has two y-scales: the left side (black circle points) is percent additional correct 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. \mathcal{MSE} provides measurable advantage

Figure 3. Suggestion gains from merging n -gram suggestions into those of Eclipse.



over $ECSE$ for 2, 6, as well as 10 suggestions, and for all token string lengths; however, the advantage in general declines with token length. 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% to 16%. When compared to the 6 suggestions case (Figure 3), the gains are stronger with 2 suggestions and lower for 10 suggestions.

The additional suggestions from $NGSE$ run the gamut, including methods, classes, and fields predictable from frequent trigrams in the corpus (e.g., `println`, `iterator`, `IOException`, `append`, `toString`, `assertEquals`, `transform`), 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 coders tend to choose shorter names for entities that are used more widely and more often; naturally these give rise to stronger signals that are picked up by the n -gram language model. It is worth repeating here that a significant portion, *viz.*, 50% of the successful suggestions are not Java keywords guessed from language context—they are project-specific tokens. This reinforces our claim that the statistical language model is capturing a significant level of local regularity in the project corpus.

In the table below, we present a different way of looking at the benefit of MSE ; the total number of keystrokes saved by using the base $ECSE$, (first row) the MSE (second row) and the percent gain from using MSE . Here, we used one specific language model to enhance one specific software tool, a suggestion engine. With more sophisticated techniques, that model syntactic, scoping, type, and semantic phenomena, we expect to achieve even lower entropy, and thus better performance in this and other software tools. We do invite readers to download our Eclipse Suggestion Plugin, CACHECA^{f,11} which based on a cache-enhanced locality-sensitive n -gram model,⁴⁹ and improved merging heuristics.

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

5. RELATED WORK

There are a few related areas of research, into which this line of work could be reasonably contextualized.

Code Completion and Suggestion *Completion* is the task of completing a partially typed-in token; *suggestion* suggests a complete token. Section 4 concerns *suggestion* engines.

Modern, mature software development environments (SDEs) provide both code completion and code suggestion, often with a unified interface. Two notable open-source Java-based examples are Eclipse and IntelliJ IDEA.

As in our work, these tools draw possible completions from existing code, but the methods of suggestion are fundamentally different.

Eclipse and IntelliJ IDEA respond to a programmer’s completion request (a keyboard shortcut such as `ctrl+space`) by deducing what tokens “might apply” in the current syntactic context. Here, the tools are primarily guided by Java programming language semantics. For example, both Eclipse and IntelliJ IDEA respond to a completion request by parsing available source code and creating 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 currently defined type-names to the list of completions. Similarly, if a variable is expected, the tools use names visible in the symbol table. Eclipse and IDEA implement dozens of these “syntactic and semantic rules” for various classes of syntactic constructs. As a final step, both tools rank the completions with a collection of apparently hand-coded heuristics.

Our approach is complementary. Rather than using language semantics and immediate context, to guess what *might* apply, our n -gram language model captures what *most often* does apply. Our approach offers a great deal more flexibility and 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. Further, our approach can enhance (or order) semantically informed suggestions and completions. It is noteworthy that perhaps some of the strongest evidence for the “naturalness” of software, though, is how well our completion prototype functions *in spite of not using semantic information*.

There are approaches arguably more advanced than those currently available in IDEs. The BMN completion algorithm of Bruch et al.⁸ focuses on finding the most likely *method calls* likely to 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.⁷ lay out a vision for next-generation IDEs that take advantage of “collective wisdom” embodied in corpora and recorded human action. We enthusiastically concur, and suggest the use of language models from NLP to capture this wisdom.

Robbes and Lanza⁴² 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 can complement history-based approaches by adding a language model. Han et al.¹⁴ uses Hidden Markov Models (HMM) to infer likely tokens from abbreviations. They make use of a dynamic programming trellis approach for back-tracking and suggestion. Their HMM is, in fact, a language model, but the paper does not describe how effective it is or how well it would perform for completion tasks without user-provided abbreviations.

Jacob and Tairas¹⁹ used n -gram language models for a different application: to find matching code clones relevant to partial programming task. Language models were built over

^f <http://macbeth.cs.ucdavis.edu/CACHECA>.

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 Pletcher¹⁷ used context-specific API information in order to sort Eclipse code completions using static types. Users can filter out future completion proposals.

The “Naturalness” of Names in Code A line of work has been automatically evaluating the quality of entity names in code, asking “Do the names reflect the meanings of the artifacts, and if not, how could they be improved^{6, 27?}” Work by Høst and Østvold^{15, 16} 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, and they use it to detect names whose usage is inconsistent with the corpus.

Summarization and Concern Location A line of work on code summarization^{9, 47, 48} uses semantic properties inferred by static analysis, rather than the “natural” regularities of software corpora capturable in statistical models; it is complementary to ours: properties derived by static analysis (as long as they can be done efficiently, and at scale) could be used to 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,⁴⁴ facts mined from code,⁴⁰ or co-occurrence of related words in code.⁴⁵

NLP and Requirements Researchers have studied the use of natural language in requirements engineering: “Can we utilize natural language specifications to automatically generate more formal specifications, or even code?”^{25, 26, 43} Some of these approaches make use of NLP tools like parsers and part-of-speech taggers. Our approach considers instead the “naturalness” of *code*, as embodied in statistical models built from large corpora of artifacts; however we also consider (in future work) the possibility of using aligned (code/natural language) corpora for tasks such as code summarization or code search, which could be viewed as translation tasks.

Software Mining Work in this very active area⁵¹ 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,^{12, 30} patterns of errors,^{21, 29} topic extraction,²⁸ guiding changes,⁵² 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 that *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 6.3 and 6.4).

6. FUTURE DIRECTIONS

Statistical models estimated over large code corpora hold tremendous promise in software engineering applications. We list some applications below.

6.1. 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 of these can be modeled using enhanced language models to capture regularities that exist at syntactic, type, scope, and even 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. In follow-on work, researchers have exploited type models,³⁸ syntactic models,^{1, 35} scope,⁴⁹ and simple data flow information,⁴¹ for various tasks. Recent work has also explored deep learning approaches to modeling code.⁵⁰ Regularities in syntax, type, scope, and even semantic properties can be captured in models and exploited. For example, Campbell et al.¹⁰ diagnose syntax error locations by exploiting the fact that language models trained on working code will be surprised by syntax errors. Indeed there are a great many potential applications to software engineering tasks, some of which we discuss further below.

6.2. Language models for accessibility

Some programmers have difficulty using keyboards, because of RSI or visual impairment. Prior work explored using speech recognition (e.g., Refs.^{3, 4, 18}) for assistance. However, due to fairly high recognition error rates, adoption has been limited.³³ None of the published approaches make use of a statistical language model trained on specific code corpora. The use of a code-specific language model will surely reduce error rates, as in conventional speech recognition engines. Much development work occurs in a maintenance or re-engineering context, where ample data is available for model estimation. Even when only a small amount of relevant code exists, language model adaptation,⁵ and cacheing⁴⁹ could be applied.

6.3. Applications of machine translation

Consider the task of summarizing a fragment of code (or change to code) 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 (SNLT)*. Code and English are two languages, and essentially both the above are *translation* tasks. *SNLT* relies on access to an *aligned corpus*, which is a large set of sentences simultaneously presented in two or more languages (e.g., parliamentary proceedings, or Scriptural verses). Consider the problem of translating a Tamil sentence *T* to an English sentence *E*. *SNLT* uses a simple Bayesian formulation to calculate the likeliest English translation; it

maximizes the right hand side of this equation over all possible English sentences E :

$$p(E|T) = \frac{p(T|E)p(E)}{p(T)}$$

For a given T , the denominator is constant, and we just maximize the numerator. Within the numerator, the conditional distribution $p(T|E)$ is estimated using E - T pairs in an aligned corpus; $p(E)$ is provided by a language model estimated over an English corpus. The formulation is clearly also applicable for the reverse translation task. The analogy to code summarization/retrieval tasks is evident: given a code fragment C , produce an English summary E ; or, conversely, retrieve a code fragment C given an English query E . For the former (summarization) task, we need the conditional distribution $p(C|E)$, and a language model $p(E)$.

First, how do we estimate the conditional distribution $p(C|E)$? Several promising sources of aligned (English/Code) corpora exist. Consider the version history of a project. Each commit potentially offers a matched pair of a log message (English), and some changes (Code). Second, many in-line comments could be matched with nearby code.² Stack-overflow posts (both questions and answers) often include closely-related code and text. Next, consider $p(E)$: the relevant (natural language) corpora concerning the code can be found in project-associated English language text, for example, code comments, documents, bug reports, mailing-list discussions, etc. We can thus estimate $p(E)$. These $p(C|E)$ and $p(E)$ models can then select maximally likely code fragments given English descriptions as shown above; the reverse task would use $p(E|C)$ and $p(C)$ models, which are also analogously estimable. This approach could use semantic properties of code, as well; however, unlike Buse and Weimer,⁹ who document the semantics of each change in isolation, we would use statistics of semantic properties over large aligned code/text corpora.

Porting between languages or dialects is another potential application; Nguyen and colleagues^{34,36,37} have exploited the availability of cross-platform applications in Java and C#, with method-level alignments, to train translation models, and learn API mappings. Karaivanov et al.²⁰ have followed the above work in a similar vein.

6.4. 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; we hypothesize that (because programs tend to be repetitive) deeper, more semantic properties of programs are *also* manifest in programs in superficially similar ways. More specifically, we hypothesize that semantic properties are *usually* manifest 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. If feasible, one can leverage this notion to build simple, scalable, and effective approximations in a wide variety of settings, as we now describe.

Consider a very general formulation of a software tool

that computes a function f over a system s , drawn from a domain of systems \mathcal{S} thus:

$$f: \mathcal{S} \rightarrow \mathcal{D}$$

where the range \mathcal{D} represents a fact derived by analysis, for example, a may-alias fact, a mined rule (e.g., an API protocol), a error message (e.g., buffer-overflow warning), or even a new, transformed version of the input system $s \in \mathcal{S}$. Let us assume that the function f is, in general, expensive or even infeasible to calculate, and is thus subject to approximation error and calculation delays in practice. We reformulate this problem by estimating f using an alternative formulation, \hat{f} , which chooses the *most likely* value of $f(s)$, given an easily calculated and readily observed set of *evidence features* $e_1(s), e_2(s), \dots$. Features $e_i(s)$ are “symptoms”; they are not deterministic, conclusive proof that $f(s)$ has some property ϕ , but nevertheless provide varying levels of justification for a probabilistic belief in ϕ .

This view has a Bayesian formulation. Denote the probability that d holds of system s , given the observed bits of evidence $e_i(s)$:

$$p(d|s) = \frac{p(e_1(s), e_2(s), \dots, e_n(s)|d)p(d)}{p(e_1(s), e_2(s), \dots, e_n(s))}$$

The distributions on the right-hand side can be estimated over large corpora. Given a specific system, the denominator is a constant. For the numerator, the prior distribution $p(d)$, over the output domain of f , can be estimated from observations in the corpus (e.g., “How often do buffer overflow errors occur?” “How often do different protocol patterns occur?” etc.). In the absence of such information, a uninformative (uniform) prior could be chosen.

Next, we estimate the strength of the association (likelihood) between the conclusion d and the observed features $e_1(s) \dots e_n(s)$ in the corpus. $p(e_1(s), e_2(s), \dots, e_n(s)|d)$ can be viewed as the conditional frequency of the observation of features in the corpus when the output domain property d holds. This term requires the availability of a corpus annotated with property d . Even in cases where d must be manually annotated, we argue that such corpora may be well-worth the investment (by analogy with the Penn Tree Bank³²) and can be constructed using volunteer open-source community, and perhaps market mechanisms like the Mechanical Turk.²²

Finally, assuming suitable models to enable calculation of $p(d|s)$ can be estimated, we write the alternative, probabilistic function \hat{f} to choose the most likely d :

$$\hat{f}(s) = \arg \max_{d \in \mathcal{D}} p(d|s)$$

We hasten to add that this Bayesian formulation is just to provide an intuition on how corpus-based statistics can be used in a principled way to aid in the construction of approximating software engineering tools. In practice, a machine learning formulation (such as decision-trees, support-vector machines, or even simple regression models) might prove more expedient.

Raychev et al.⁴¹ demonstrated this idea, with an application to guessing useful names of variables in obfuscated

code. Variable names are chosen by programmers to represent semantic properties; a corpus of code with clear names can be used for training. Using very simple data-flow properties (readily derivable from syntax) as predictive features, and a log-linear formulation of the above idea, they were able to guess helpful variable names.

7. 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.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant Nos. 1445079, 1247280, 1414172.

Proceedings of the 48th Annual Southeast Regional Conference, 2010.

20. Karaivanov, S., Raychev, V., Vechev, M. Phrase-based statistical translation of programming languages. In *Onward!* ACM, 2014, 173–184.
21. Kim, S., Pan, K., Whitehead, E. Jr. Memories of bug fixes. In *Proceedings, ACM SIGSOFT FSE*. ACM, 2006, 35–45.
22. Kittur, A., Chi, E., Suh, B. Crowdsourcing user studies with mechanical turk. In *Proceedings, CHI*. ACM, 2008.
23. Knuth, D.E. Literate programming. *Comput. J.* 21, 2 (1984), 97–111.
24. Koehn, P. *Statistical Machine Translation*. Cambridge University Press, 2010.
25. Konrad, S., Cheng, B. Real-time specification patterns. In *Proceedings ICSE*, 2005.
26. Körner, S.J., Tichy, W.F. Text to software. In *Proceedings of FSE/SDP Workshop on the Future of Software Engineering Research*. ACM, Nov. 2010.
27. Lawrie, D., Morrell, C., field, H., Binkley, D. What's in a name? A study of identifiers. *Proceedings, ICPC*, 2006.
28. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P. Sourcerer: Mining and searching internet-scale software repositories. *Data Mining Knowl. Discov.* 18, 2 (2009), 300–336.
29. Livshits, B., Zimmermann, T. Dynamine: Finding common error patterns by mining software revision histories. *ACM SIGSOFT Softw. Eng. Notes* 30, 5 (2005).
30. Mandelin, D., Xu, L., Bodík, R., Kimelman, D. Jungloid mining: Helping to navigate the API jungle. In *ACM SIGPLAN Notices*. Volume 40. ACM, 2005.
31. Manning, C., Schütze, H. *Foundations of Statistical Natural Language Processing*. Volume 59. MIT Press, 1999.
32. Marcus, M., Marcinkiewicz, M., Santorini, B. Building a large annotated corpus of English: The Penn treebank. *Comput. Linguist.* 19, 2 (1993), 313–330.
33. Mills, S., Saadat, S., Whiting, D. Is voice recognition the solution to keyboard-based RSI? In *Automation Congress, 2006. WAC'06. World*, 2006.
34. Nguyen, A.T., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N. Statistical learning approach for mining API usage mappings for code migration. In *ASE*, 2014.
35. Nguyen, A.T., Nguyen, T.N. Graph-based statistical language model for code. 2015.
36. Nguyen, A.T., Nguyen, T.T., Nguyen, T.N. Lexical statistical machine translation for language migration. In *FSE*, 2013.
37. Nguyen, A.T., Nguyen, T.T., Nguyen, T.N. Migrating code with statistical machine translation. In *ICSE Companion*, 2014.
38. Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N. A statistical semantic language model for source code. In *ESEC FSE*. ACM, 2013.
39. Pinker, S. *The Language Instinct: The New Science of Language and Mind*. Volume 7529. Penguin, London, UK, 1994.
40. Rastkar, S., Murphy, G.C., Bradley, A. Generating natural language summaries for cross-cutting source code concerns. In *Proceedings, ICSM*, 2011.
41. Raychev, V., Vechev, M., Krause, A. Predicting program properties from big code. In *POPL*. ACM, 2015.
42. Robbes, R., Lanza, M. Improving code completion with program history. *Autom. Softw. Eng.* 17, 2 (2010), 181–212.
43. Rolland, C., Proix, C. A natural language approach for requirements engineering. In *Advanced Information Systems Engineering*. Springer, 1992, 257–277.
44. Shepherd, D., Fry, Z., Hill, E., Pollock, L., Vijay-Shanker, K. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings, AOSD*. ACM, 2007, 212–224.
45. Shepherd, D., Pollock, L., Tourwé, T. Using language clues to discover crosscutting concerns. In *ACM SIGSOFT Software Engineering Notes*. Volume 30. ACM, 2005.
46. Sparck-Jones, K. Natural language processing: A historical review. *Current Issues in Computational Linguistics: In Honour of Don Walker (Ed Zampolli, Calzolari and Palmer)*. Kluwer, Amsterdam, the Netherlands, 1994.
47. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K. Towards automatically generating summary comments for Java methods. In *Proceedings, ASE*, 2010.
48. Sridhara, G., Pollock, L., Vijay-Shanker, K. Automatically detecting and describing high level actions within methods. In *Proceedings, ICSE*, 2011.
49. Tu, Z., Su, Z., Devanbu, P. On the localness of software. In *FSE*. ACM, 2014, 269–280.
50. White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D. Toward deep learning software repositories. In *MSR*, 2015.
51. Xie, T., Thummalapenta, S., Lo, D., Liu, C. Data mining for software engineering. *IEEE Comput.* 42, 8 (2009).
52. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A. Mining version histories to guide software changes. In *ICSE*. IEEE Computer Society, 2004.

References

1. Allamanis, M., Sutton, C. Mining idioms from source code. In *FSE*. ACM, 2014.
2. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.* 28 (2002), 970–983.
3. Arnold, S., Mark, L., Goldthwaite, J. Programming by voice, VocalProgramming. In *Proceedings, ACM Conference on Assistive Technologies*. ACM, 2000, 149–155.
4. Begel, A. Spoken language support for software development. In *Proceedings, VL/HCC*. IEEE Computer Society, 2004, 271–272.
5. Bellegarda, J. Statistical language model adaptation: Review and perspectives. *Speech Commun.* 42, 1 (2004), 93–108.
6. Binkley, D., Hearn, M., Lawrie, D. Improving identifier informativeness using part of speech information. In *Proceedings, MSR*. ACM, 2011.
7. Bruch, M., Bodden, E., Monperrus, M., Mezini, M. Ide 2.0: Collective intelligence in software development. In *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research*. ACM, 2010, 53–58.
8. Bruch, M., Monperrus, M., Mezini, M. Learning from examples to improve code completion systems. In *Proceedings, ACM SIGSOFT ESEC/FSE*, 2009.
9. Buse, R., Weimer, W. Automatically documenting program changes. In *Proceedings, ASE*. ACM, 2010, 33–42.
10. Campbell, J.C., Hindle, A., Amaral, J.N. Syntax errors just aren't natural: Improving error reporting with language models. In *MSR*. ACM, 2014.
11. Franks, C., Tu, Z., Devanbu, P., Hellendoorn, V. Cacheca: A cache language model based code suggestion tool. *ICSE Demonstr. Track* (2015).
12. Gabel, M., Su, Z. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings, ACM SIGSOFT FSE*. ACM, 2008, 339–349.
13. Gabel, M., Su, Z. A study of the uniqueness of source code. In *Proceedings, ACM SIGSOFT FSE*. ACM, 2010, 147–156.
14. Han, S., Wallace, D.R., Miller, R.C. Code completion from abbreviated input. In *ASE*. IEEE Computer Society, 2009, 332–343.
15. Høst, E.W., Østvold, B.M. *Software Language Engineering. Chapter The Java Programmer's Phrase Book*. Springer-Verlag, Berlin, Heidelberg, 2009.
16. Høst, E., Østvold, B. Debugging method names. In *Proceedings ECOOP*. Springer, 2009, 294–317.
17. Hou, D., Pletcher, D. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Proceedings, ICSM*, 2011.
18. Hubbell, T., Langan, D., Hain, T. A voice-activated syntax-directed editor for manually disabled programme RS. In *Proceedings, ACM SIGACCESS*. ACM, 2006.
19. Jacob, F., Tairas, R. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, 2010.

Abram Hindle (abram.hindle@ualberta.ca), Department of Computing Science, University of Alberta, Edmonton, Canada.

Earl T. Barr (e.barr@ucl.ac.uk), Department of Computer Science, University College London, United Kingdom.

Mark Gabel, Zhendong Su, and Premkumar Devanbu (mark.gabel@gmail.com, [su, ptdevanbu]@cs.ucdavis.edu), Department of Computer Science, UC Davis, CA.