Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Studying the Difference Between Natural and Programming Language Corpora

Casey Casalnuovo · **Kenji Sagae** · **Prem Devanbu**

Received: date / Accepted: date

Abstract Code corpora, as observed in large software systems, are now known to be far more repetitive and predictable than natural language corpora. But why? Does the difference simply arise from the syntactic limitations of programming languages? Or does it arise from the differences in authoring decisions made by the writers of these natural and programming language texts? We conjecture that the differences are *not* entirely due to syntax, but also from the fact that reading and writing code is *un*-natural for humagins, and requires substantial mental effort; so, people prefer to write code in ways that are familiar to both reader and writer. To support this argument, we present results from two sets of studies: 1) a first set aimed at attenuating the effects of syntax, and 2) a second, aimed at measuring repetitiveness of text written in other settings (*e.g.* second language, technical/specialized jargon), which are also effortful to write. We find find that this repetition in source code is not entirely the result of grammar constraints, and thus some repetition must result from human choice. While the evidence we find of similar repetitive behavior in technical and learner corpora does not conclusively show that such language is used by humans to mitigate difficulty, it is consistent with that theory. This discovery of "non-syntactic" repetitive behaviour is *actionable*, and can be leveraged for statistically significant improvements on the code suggestion task. We discuss this finding, and other future implications on practice, and for research.

Keywords Language Modeling · Programming Languages · Natural Languages · Syntax & Grammar · Parse Trees · Corpus Comparison

Casey Casalnuovo
Department of Computer Science, University of California, Davis, CA, USA
E-mail: ccasal@ucdavis.edu

Kenji Sagae
Department of Linguistics, University of California, Davis, CA, USA
E-mail: sagae@ucdavis.edu

Prem Devanbu
Department of Computer Science, University of California, Davis, CA, USA
E-mail: ptdevanbu@ucdavis.edu

1 Introduction

Source code is often viewed as being primarily intended for machines to interpret and execute. However, more than just an interlocutory medium between human and machine, it is also a form of communication between humans - a view advanced by Donald Knuth:

> *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 (Knuth, 1984).*

Software development is usually a team effort; code that cannot be understood and maintained is not likely to endure. It is well known that most development time is spent in maintenance rather than *di novo* coding (Lehman, 1980). Thus it is very reasonable to consider source code as a form of human communication, which, like natural languages, encodes information as sequences of symbols, and is amenable to the sorts of statistical language models (LM) developed for natural language. This hypothesis was originally conceived by Hindle et al. (Hindle et al, 2012), who showed that LM designed for natural language were actually *more effective* for code, than in their original context. Hindle *et al* used basic ngram language models to capture repetition in code; subsequent, more advanced models, tuned for modular structure (Tu et al, 2014; Hellendoorn and Devanbu, 2017), and deep learning approaches such as LSTMs (Hochreiter and Schmidhuber, 1997) (with implementations such as (White et al, 2015; Khanh Dam et al, 2016)) yield even better results. Fig 1 demonstrates this difference on corpora of Java and English, using the standard entropy measure (Manning and Schütze, 1999) over a held-out test set. A lower entropy value indicates that a token was less surprising for the language model. These box plots display the entropy for each token in the test set, and show that (regardless of model) Java is more predictable than English[1].

But why is code more predictable? The difference could either arise from a) *inherent syntactic* differences between natural and programming languages or b) the *contingent authoring* choices made by authors. Source code grammars are unambiguous, for ease of parsing; this limitation might account for the greater predictability of code. But there may be other reasons; perhaps source code is more domain-specific; perhaps developers deliberately limit their constructions to a smaller set of highly reused forms, just to deal with the great cognitive challenges of code reading and writing. Recent work on human processing of natural languages has shown that the entropy of natural language text is correlated with cognitive load (Frank, 2013), with more surprising language requiring greater effort to interpret. In code, this suggests the intuitive notion that, in general, the use of more familiar and less surprising source code is expected to reduce cognitive load requirements.

Finally, we note that prior studies on the differences between natural language and code have typically aimed at exploring one programming language and one natural language (Hindle et al, 2012; Tu et al, 2014). Though this paper will focus primarily on syntactic differences between English and Java, we do wish to confirm that the differences seen between English and Java apply across a variety of programming and natural languages.

This raises 3 questions of interest:

1. Do the differences in repetition seen between English and programming languages like Java generalize to other programming and natural languages?
2. How much does programming language *syntax* influence repetitiveness in coding? and
3. What are the *contingent* factors (not constrained by syntax) that play a role in code repetitiveness?

[1] Precise details on the datasets and language models will be presented later their respective sections.

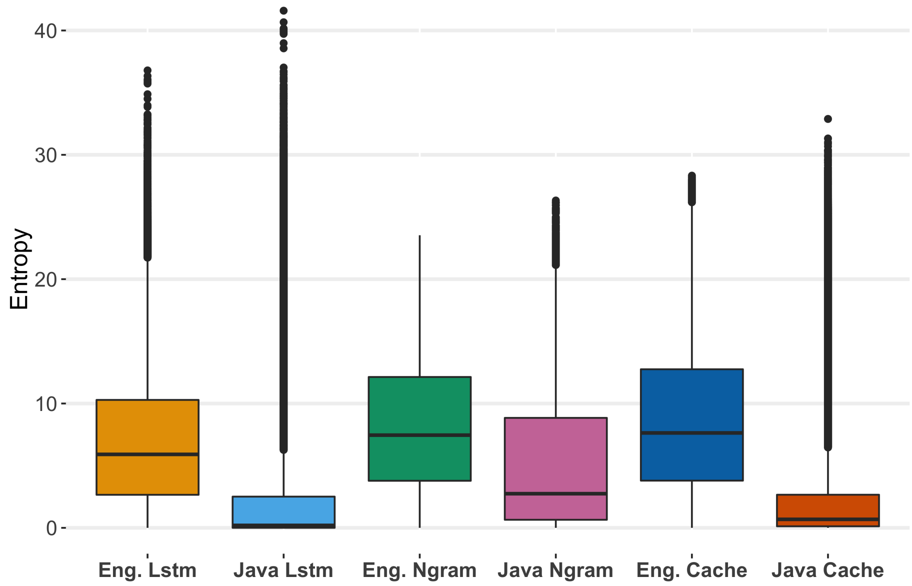


Fig. 1 Entropy comparisons of English and Java corpora from 3 different language models

We address the first question, with experiments breaking down the syntactic differences between source code and natural language. We study the second question using pre-parsed English and Code data, to account for the effects of syntax. The third question is very open-ended; to constrain it, we consider a variant thereof:

3. *Is repetitiveness observed in code also observed in other natural language corpora that similarly required significant effort from the creators*?

We address this question, with corpora of text that are similarly "effortful" for the writers (or readers, or both) or have potentially higher costs of miscommunication: we consider English as a second language and in specialized corpora such as legal or technical writing. To summarize our results, we find:

– The differences between source code and English, observed previously in Java hold true in many different programming and natural languages.
– Programming language corpora are more similar to each other than to English.
– Even when accounting for grammar and syntax in different ways, Java is statistically significantly more repetitive than English.
– ESL (English as a Second language) corpora, as well as technical, imperative, and legal corpora, do exhibit repetitiveness similar to that seen in code corpora.
– Our findings on syntax have practical consequences: they help significantly improve code suggestion on open category tokens in Java, which are harder for language models to predict but useful for programmers.

These suggest that differences observed between natural and programming languages are not entirely due to grammatical limitations, and that code is also more repetitive due to contingent facts – i.e. humans *choose* to write code more repetitively than English. Our experiments with bodies of text (other than code) that require greater effort indicate that people choose to write these corpora quite repetitively as well; this suggests that the greater repetitiveness in code could also arise from a desire to reduce effort. We conclude the paper with some discussion on the practical actionability of this scientific study, including specifically on code suggestion. A partial replication package for the data, source code, and experiments in this paper can be found at `https://github.com/caseycas/CodeNLPReplication`.

2 Theory

We provide a few definitions used throughout this paper. First, by *syntax*, we mean the aspects of language related to structure and grammar, rather than meaning. Both code (an artificial language) and natural language have syntactic constraints. Code has intentionally simplified grammar, to facilitate language learning, and to enable efficient parsing by compilers. Human languages have evolved naturally; grammars for natural languages are imperfect models of naturally occurring linguistic phenomena, and in general, are more complex, non-deterministic, and ambiguous than code grammars.

A language's syntax constrains the set of valid utterances. The more restrictive the grammar, the less choice in utterances. Thus, it is possible that the entropy differences between code and NL arise entirely out of the more restrictive grammar for code. If so, the observed differences are not a result of conscious choice by humans to write code more repetitively; it is simply the grammar.

However, if we could explicitly account for the syntactic differences between English and code, and *still* find that code is repetitive, then the unexplained difference could well arise from deliberate choices made by programmers. Below, we explore a few theories of why the syntax of source code may be more repetitive than the syntax of natural language.

Second, to explain another bit of terminology briefly: by *corpus*, we mean a body of text assembled with a specific experimental goal in mind, such as: a collection of tweets, a collection of Java source code, a collection of EU parliamentary speeches, or a very broad collection of different kinds of text (*e.g.,* the Brown Corpus).

2.1 Syntactic Explanations

2.1.1 Open And Closed Vocabulary Words

As languages evolve, vocabularies expand; with time, certain word categories expand more rapidly than others. We can call categories of words where new words are easily and frequently added *open category* (*e.g.*, nouns, verbs, adjectives). As the corpus grows, we can expect to see more and more open category words. *Closed category* vocabulary, however, is limited; no matter how big the corpus, the set of distinct words in these categories is fixed and limited[2]. In English, closed category words include conjunctions, articles, and pronouns. This categorization of English vocabulary is well-established (Bradley, 1978), and we adapt this analogously for source code.

In code, reserved words, like *for*, *if*, or *public* form a closed set of language-specific keywords which help organize syntax. The arithmetic and logical operators (which combine elements in code like conjunctions in English) also constitute closed vocabulary. Code also has *punctuation*, like ";" which demarcates sequences of expressions, statements, etc. These

[2] While this category is very rarely updated, there could be unusual and significant changes in the language – for instance a new preposition or conjunction in English.

categories are slightly different from those studied by Petersen et al (2012) who consider a kernel or core vocabulary, and an unlimited vocabulary to which new words were added. Our definitions are tied to syntax rather than semantics, hingeing on the type of word (e.g. noun vs conjunction or identifier vs reserved word) rather than how core the meaning of the word is to the expressibility of the language. Closed vocabulary words are necessarily part of the kernel lexicon they describe, but open category words will appear in both the kernel and unlimited vocabulary. For example, the commonly used iterator *i* would be in the kernel vocabulary in most programming languages, but other identifiers like *registeredStudent* could fall under Petersen's unlimited lexicon.

Closed vocabulary tokens relate most to syntactic form, whereas open vocabulary tokens relate more to semantic content. As long as grammars are stable, a small number of closed category tokens is sufficient. In contrast, new nouns, verbs, adverbs, and adjectives in English, or types and identifiers in Java are constantly invented to express new ideas in new contexts. Thus, one can expect that the corpus that only contains these words, (*viz.,* the open-category corpus) would be more reflective of content, and less of the actual syntax. Thus analyzing the open-category corpus (for code and English) would allow us to judge the repetitiveness that arises more from content-related choices made by the authors, rather than merely from syntax *per se*. Removal of closed category words, to focus on content rather than form, recapitulates the removal of *stop words* (frequently occurring words that are considered of no or low value to a particular task) in natural language processing. Thus, our first experiment addresses the question:

RQ1. *How much does removing closed category words affect the difference in repetitiveness and predictability between Java and English?*

2.2 Ambiguity in Language

Programming language grammars are intentionally unambiguous, whereas natural languages are rife with grammatical ambiguity. Compilers must be able to easily parse source code; syntactic ambiguity in code also impedes reading & debugging. For example, in the C language, there are constructs that produce undefined behavior (See Hathhorn et al. (Hathhorn et al, 2015)). Different compilers might adopt different semantics, thus vitiating portability.

Various theories for explaining the greater ambiguity in natural language have been proposed. One camp, led by Chomsky, asserts that ambiguity in language arises from NL being adapted not for purely communicative purposes, but for cognitive efficiency (Chomsky et al, 2002).

Others have argued that ambiguity is desirable for communication. Zipf (Zipf, 1949) argued that ambiguity arises from a trade off between speakers and listeners: ambiguity reduces speaker effort. In the extreme case if one word expressed all possible meanings then ease of speaking would be minimized; however, listeners would prefer less ambiguity. If humans are able to disambiguate what they hear or read more easily, then some ambiguity could naturally arise. Others argue ambiguity could arise from memory limitations or applications in inter-dialect communication (Wasow et al, 2005). A variant of Zipf's argument is presented by Piantadosi et al. (Piantadosi et al, 2012): since ambiguity is often resolvable from context, efficient language systems will allow ambiguity in some cases. They empirically demonstrated that words which are more frequent and shorter in length, tend to possess more meanings than infrequent and longer words.

Ambiguity is widely prevalent in natural language, both in word meaning and in sentence structure. Words like “take” are *polysemic*, with many meanings. Syntactic structure (even without polysemic words) can lead to ambiguity. One popular example of ambiguous sentence structure is that of prepositional attachment. Consider the sentence:

They saw the building with a telescope.

There are two meanings, depending on where the phrase *with a telescope* attaches: did they see using the telescope, or is the telescope mounted on the building? Both meanings are valid, where one or the other may be preferred based on the context.

Such ambiguous sentences can be resolved using a *constituency parse tree* or *CPT* – representing natural language in a way similar to how an *AST* represents source code. A *CPT* is built from nested units, building up to a root node that represents the whole sentence (typically represented with *S* or *ROOT*). The terminal nodes are the words of the original sentence, and the non-terminals include parts of speech (nouns/verbs) and phrase labels (noun phrases, verb phrases, prepositional phrases, etc). While there is no definitive set of non-terminals used of labeling English sentences, some sets are very commonly used, such as the one designed for the Penn Treebank (Marcus et al, 1993).

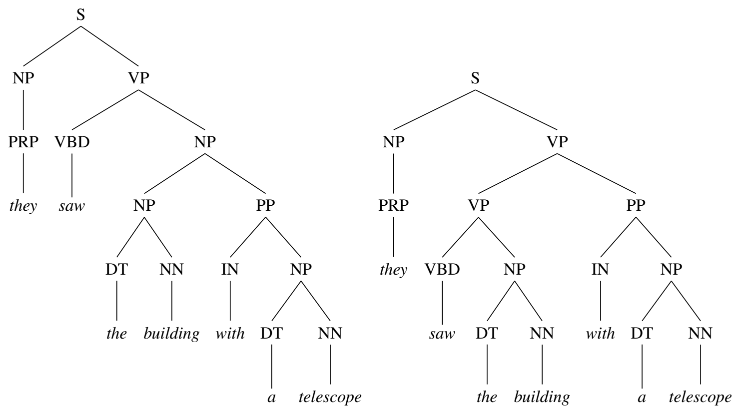


Fig. 2 Two parse trees for the sentence *They saw the building with a telescope*. The tree on the left corresponds to the the reading that the telescope is part of the building; on the right, to the reading that the viewing was done with a telescope

A *CPT fully* resolves syntactic ambiguities: *e.g.,* consider Fig. 2, which shows the two possible *CPTs* for our example sentence. While the raw text is ambiguous, each of the *CPTs* fully resolve and clarify the different possible meanings; only one meaning is possible for a given *CPT*. In source code, however, the syntactic structure is unambiguous, given the raw tokens.

Source code syntax is represented using a similar hierarchical construction: the *abstract syntax tree* or *AST*. However, *ASTs* differ from *CPTs* in that they exclude some tokens of the original text, that are inferable from context. Both trees unambiguously represent structure in natural language and source code. In section 3.4, we will discuss how we modified these slightly to further improve their comparability.

Using such trees, we can revisit the question of whether the greater repetitiveness and predictability of source code arises merely from simpler, unambiguous syntactic structure.

Once converted to a tree based form, code and NL are on equal footing, with all ambiguity vanquished; the syntactic structure is fully articulated. On this equal footing, then, is code *still* more repetitive and predictable than English? This leads us to our next research question:

RQ2. *When parse trees are explicitly included for English and Java, to what degree are the differences in predictability accounted for?*

2.3 Explanations From Contingent Factors

After accounting for the *inherent* explanations for the greater repetitiveness of code, like syntax and vocabulary, we consider *contingent* explanations, that is, whether code is more repetitive because human *choose* to communicate in code differently.

We theorize that *humans communicate differently when the effort of communication, and/or the cost of mis-communication is high*. We clarify these factors with a few examples. In some settings, the *effort* required to communicate is higher than others. Settings requiring specialized language *e.g.,* intricate and technical language, like legal arguments or mathematical proofs, or unfamiliar settings *e.g.,* speaking in a foreign language—require greater human effort. In such settings, we might expect people to have lower flexibility, and thus show less variation in how they choose to communicate. Likewise in some settings, the cost of *mis*-communication is very high, *e.g.,* in legal documents, or instruction manuals. In such settings, we might expect that humans just to be very clear, would resort to very common, well-understood constructions, to have greater confidence that the language would be familiar and unambiguous to most readers.

These ideas are consistent with psycholinguistic findings that higher entropy in natural language incurs greater cognitive load in human language processing (Levy, 2008; Demberg and Keller, 2008), and that the use of less surprising or more predictable word choice reduces processing effort (Frank, 2013). Since systematic repetition is associated with lower entropy, it is plausible that repetitiveness is employed as a strategy to manage cognitive load in situations where the level of effort required for effective communication is high. Additionally, existing research suggests that developers process software using the same brain machinery used for natural language, but do so with less fluency. Prior work does suggest (Siegmund et al, 2014) that some of the parts of the brain used in natural language comprehension are shared when understanding source code.

However, despite the overlap in brain regions used, eye-tracking studies have shown that the way in which humans read source code and natural language differ in interesting ways (Busjahn et al, 2015; Jbara and Feitelson, 2017). Natural language tends to be read in a linear fashion. For English, normal reading order would be largely left-to-right, top-to-bottom. While source code is typically read left-to-right at the statement level, it involves a greater degree of non-linear reading behavior overall. People's eyes jump around the code while reading, following function invocations to their definitions, checking on variable declarations, and assignments, following control-flow paths *etc.* Busjahn et al. (Busjahn et al, 2015) found this behavior in both novices and experts. Although there is no experimental evidence[3] to directly support the claim that code (as a communication medium) is more difficult for humans than natural language, available evidence and intuition suggests, at the very least, that code is a type of medium that presents special challenges for humans.

[3] Indeed, it is not clear how to even design such an experiment.

Though establishing differences in difficulty between natural language and code is challenging, some research in the areas of programming language design and CS education has touched on the difficulty *between* programming languages for novices. Programming languages such as Quorum (Stefik and Ladner, 2017) have leveraged research on what parts of syntax programming language learners struggle with (Stefik and Siebert, 2013). Languages such as Ruby, Python, and Quorum were found to be more intuitive than Java or Perl, which did not better than a language with random keywords, and that static typing was a hurdle for new programmers to learn. Likewise, alternative schemes such as block-based language were found to be advantageous in teaching programming language constructs, if not at overall program comprehension (Weintrop and Wilensky, 2015). However, these studies focus on learning difficulty, rather than an inherent difficulty of communication in natural and programming languages by humans with fluency in these languages.

Finally, Code is actually also inherently a *machine*, with a highly-specific, and carefully designed function, that must be maintained; the consequences of mis-communication concerning code is very high. If a maintainer misunderstands the intent of the original developer, and makes inappropriate changes, the results could well be catastrophic. Practical code typically stays in use for a good long while, and is maintained by large teams; so developers have a strong incentive to ensure that their code is readily understood by the maintainers.

We hypothesize that these factors cause humans to write code with a very high level of repetitiveness. This hypothesis concerns the motivations of programmers, and is difficult to test directly. We therefore seek corpus-based evidence in different kinds of natural language. Specifically, we would like to examine corpora that are more difficult for their writers to produce and readers to understand than general natural language. Alternatively, we also would like corpora where, like code, the cost of miscommunication is higher. Would such corpora evidence a more repetitive style? To this end, we consider a few specialized types of English corpora: *1) corpora produced by non-fluent language learners, presumably with a great deal of effort* and 2*) corpora written in a technical style or imperative style, with the intent that readers need to understand the content precisely, without confusion*.

2.3.1 Native vs Language Learners

Attaining fluency in a second language is difficult. If humans manage greater language difficulty by deploying more repetitive and templated phrasing, then we might find evidence for this in English as a Foreign language *(EFL)* corpora.

Use of templated and repetitive language appears in linguistic research through the concept of *formulaic sequences* (Schmitt and Carter, 2004). These are word sequences that appear to be stored and pulled from memory as a complete unit, rather than being constructed from the grammar. Such sequences come in many forms, one of the most common being concept of *idioms*, but the key point is that they are intended to convey information in a quick and easy manner (Schmitt and Carter, 2004). This theory is backed by empirical evidence, as both native and non-native readers have been found to read such phrases faster than non-formulaic language constructs (Conklin and Schmitt, 2008). Several studies have found that language learners acquire and use these sequences as a short hand to express themselves more easily, and thus use them more excessively than native speakers (Schmitt and Carter, 2004; De Cock, 2000; Paquot and Granger, 2012). We can see such use as an adaption for novices increased difficulty with the language. If we can statistically capture the patterns in written corpora of language learners and see similar trends as in source code, it would be consistent with the hypothesis that source code is more repetitive because it is more cognitively difficult. Therefore we ask the following questions:

RQ3. *Do english foreign language learners produce writing that resembles code patterns more closely than general English?*

2.3.2 Technical and Imperative Style

Tied into alternative cognitive explanations for the observed differences between programming and natural languages is the question of style. Source code is a technical production; if writing in a technical style is more difficult, we would expect other technical corpora to be more repetitive and predictable.

While differences between general and technical language use have long been a focus of linguists (Gotti, 2011), the attempts to categorize the differences between the two (Gotti, 2011) run into somewhat contradictory forces. Gotti cites Hoffman who gives 11 properties desirable in technical language, including unambiguousness, objectivity, brevity, simplicity, consistency, density of information, etc. The desire for a lack of ambiguity contradicts with the desire for a concise and informative text, as the meaning is also intended to be clear (Hoffmann, 1984; Gotti, 2011). Moreover, technical language is also heavily decided by the intended audience, ranging a spectrum from communication to laypeople (either for educational or general public dissemination) to communication between experts, which often includes highly unambiguous mathematical formulations (Gotti, 2011). Expert to expert communication is characterized by usage of unexplained terminology, or jargon, which can be efficient (Varantola, 1986; Gotti, 2011). Moreover, technical language is marked by compound noun phrases, which may be easier for language models to detect. Salager et al. found that compared to the 0.87% rate of compounds in general English, technical language had them appear at a rate of 15.37% (Salager, 1983). Once learned, these instances jargon and compound phrases may act to reduce cognitive load for experts who recognize them, allowing for easier reference of complex ideas.

Additionally, longer sentences are associated with technical language, especially legal language, with increased length sometimes suggested as arising from a need for greater precision (Gotti, 2011). However, this claim of precision in legal language is disputed, as Danet points out that for being supposedly precise, laws often require extensive interpretation (Danet, 1980). Though there is evidence of political gamesmanship making the language overly verbose and complex, legal language and technical language in general are still driven in part by the need for precision and reduced ambiguity. Such language can be seen as more difficult or labored than general language, and we would expect it to feature more code-like properties.

Finally, if we consider language transactions as an optimization of cognitive effort between speaker and listener (Zipf, 1949; Piantadosi et al, 2012), then it is useful to consider how the type of language will shift the balance in one direction or the other. In fact, psycholinguistic research suggests that a reader's or listener's cognitive load increases when faced with certain types of ambiguity and increased entropy in language (Hale, 2003). In language where there is a high cost when the listener misinterprets the speaker, then these theories would predict the language would become less ambiguous, which would be reflected in language models. In code, there is a very high cost of misinterpretation, and thus the grammar does not typically permit ambiguity (barring undefined behavior in languages like C). Thus, in theory, contexts in natural language with a high cost will also more closely resemble code. Technical language is one such area where clear communication is important, but imperative language is another. When humans write instructions or give commands, if the reader or listener misinterprets the commands, there is presumably a higher cost than in the case of merely descriptive language. Therefore, we would also expect such corpora to exhibit more code-like behavior.

RQ4. *Do technical and imperative language, seemingly more difficult and with higher cost of misinterpretation than general and domain specific language, exhibit more code-like properties?*

2.4 Measuring Repetition in Language

When studying repetition and comparing between our programming and natural languages we apply two general techniques - language modeling and Zipf frequency plots. This section provides some background on these methods. Specific details on how we extend and apply these method in our experiments can be found in sections 3.2 and 3.5.

2.4.1 Statistical Language Models and Entropy

A *Statistical Language Model* assigns a probability to utterances in a language. These models are estimated on a representative training corpus, and typically work by by estimating the probabilities of a token in a given context. Let us define an utterance as a sequence of tokens $S = t_1, t_2, ..., t_n$. For each token t_i in the sequence, we have a corresponding *context* $C(t_i)$. The exact definition of the context will depend on what language model is being used. In *ngram* models, the context is defined as the preceding n tokens; in neural models such as an forward LSTM, all previous tokens are available as potential context[4]. Then, we can define the probability of the sequence relative a language model L_M as:

$$P(S;L_M) = \prod_{i=1}^{n} P(t_i|C(t_i);L_M) \tag{1}$$

Eq. 1 defines the probability of the sequence as the product of the probabilities of each token in the sequence, given the context of the token and the language model. Typically, instead of using the raw probabilities, Eq. 1 is represented in the form of *entropy*. Formally, the average entropy per token in S, $\bar{H}$ is defined as:

$$\bar{H}(S;L_M) = -\frac{1}{\|S\|} * \sum_{i=1}^{n} log(P(t_i|C(t_i);L_M)) \tag{2}$$

Originally proposed by Shannon (Shannon, 1948), who later used it to predict the next letter in a sequence of English (Shannon, 1951), entropy models the amount of information conveyed by a message. That is, if the message where to be translated to binary, what is the fewest number of bits required to encode it in the language model? The fewer the bits are needed encode the message, the less information (and thus more repetitive/predictable) the message. In the context of language models, entropy indicates how *unexpected* a token is, and acts as measure of how successful the language model is in capturing the underlying relevant features that characterize the grammar, vocabulary usage, and ideas of the text.

Different types of models capture different kinds of repetitiveness, so considering the entropy of a text under multiple language models gives greater insight into the features of a

[4] Bidirectional LSTMs can make use of context both before and after a token.

text. We thus explore predictability and repetition using basic ngram models, ngram cache models that focus on capturing local repetition, and LSTM models capable of capturing long term dependencies in the text.

N-gram models are the simplest: here, the context $C(t_i)$ is equivalent to the past n tokens in the sequence. For example, the probability of a sentence in a trigram model would be:

$$P(S) = \prod_{i=3}^{n} P(t_i|t_{i-2}, t_{i-1}) \tag{3}$$

Note that we can pad the start of a sequence with buffer tokens in order to produce a probability value for the initial tokens. Thus, in the above example t_3 would be the actual first token in the sequence.

Ngram models capture the global repetitiveness of a corpus, but source code has additional *local* repetitiveness. These local patterns are modeled in a local *cache*, and this type of model as an *ngram cache model*. Tu et al. originally observed this effect in Java code (Tu et al, 2014), and Hellendoorn et al. have recently extended the idea of a cache to have multiple layers of nesting (Hellendoorn and Devanbu, 2017). It is notable the ngram cache models do not show improvement over ngram models in English. Formally, Eq. 4 shows the basic cache model as described by Tu et al.

$$\begin{aligned} P(t_i|h, cache) = \lambda * P_{ngram}(t_i|h) + (1-\lambda) * P_{cache}(t_i|h) \\ 0 <= \lambda <= 1 \end{aligned} \tag{4}$$

The cache model interpolates between two ngram models P_{ngram} and P_{cache}. The first is the regular ngram model as described in 3. The second ngram model is built using counts built from the local *cache*. Details on this model and how λ is selected can be found in Zhaopeng et al (Tu et al, 2014).

Finally, we also use Long Short Term Memory Network, or LSTM (Hochreiter and Schmidhuber, 1997). Unlike traditional feedforward models, these recursive neural networks (RNNs) allow models to leverage variable-length contexts (Mikolov et al, 2010). LSTMs are RNNs, with the ability to choose to remember some of the prior elements of the sequence[5]. This "selective memory" allows LSTMs to learn longer contexts than the fixed ngram models.

LSTMs and RNNs have been applied to both natural (Mikolov et al, 2010; Sundermeyer et al, 2012) and programming languages (White et al, 2015; Khanh Dam et al, 2016). We include LSTMs to compare and contrast their ability to learn natural and programming languages, but also to leverage their greater context when modeling our linearized parse trees. Much larger ngram models are needed to capture the text of these trees, but the selective learning of the LSTM is greater able to capture the repetition in them. We provide more details on these in sections 3.4 and 3.5.

2.4.2 Zipfian Distributions in Natural Language and Code

Zipf famously observed that the distribution of the vocabulary of natural language is made up of a few highly frequent words with a long tail of very rare words (Zipf, 1949). The original

[5] A good explanation of the details of LSTM cell structure can be found at: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

formula indicates a power-law relationship between the rank of a word and its frequency. By rank, we mean that the most frequent word receives rank 1 (or 0), then the next most frequent gets rank 2, and so on. Then, the frequencies of this words following roughly this formula:

$$f \approx \frac{C}{r^{\alpha}} \tag{5}$$

Here, f represents the frequency of the word, C is a constant, r is the word rank, and α is the power used to fit the line (originally observed as being close to 1) to the data. This law was improved slightly to better fit very frequent and very rare words by Mandelbrot soon after (Mandelbrot, 1953). He proposed an additional constant b, which was better able to account for high frequency words in natural language texts:

$$f \approx \frac{C}{(r+b)^{\alpha}} \tag{6}$$

However, the power law only approximates the frequency patterns of language unigrams. More precise models of word frequency include a bipartite function known as the Double Pareto; this plot of the distribution has an observable bend in log-log plots of natural language data (Ferrer i Cancho and Solé, 2001; Gerlach and Altmann, 2013; Piantadosi, 2014; Mitzenmacher, 2004), between two slopes. The first slope is associated with the most frequent words, called a kernel lexicon, and a second rate of decrease among the less common words, belonging to an unlimited lexicon. When new vocabulary is added to a natural language, they are added to the unlimited lexicon at a decreasing rate over time (Petersen et al, 2012). By accounting for these two vocabularies in modeling, very accurate simulations of natural language vocabulary frequency and growth can be captured (Gerlach and Altmann, 2013). Notably, the decreasing need for additional words observed in natural language (Petersen et al, 2012), is not true in source code, as developers make up new identifiers for new files, which is why cache models are so much more effective in code (Hindle et al, 2012; Tu et al, 2014). Finally, while these two lexicons are similar to the notion of open and closed vocabularies, they are slightly different. The kernel lexicon consists of the closed category words that build the structure of the language, as well as very frequent open category words. The rest of the open category words fall into the unlimited lexicon.

Power laws and other related distributions (exponential, lognormal, etc) have been examined in regards to many source code features of interest: class methods and fields, dependency and function call graphs, etc (Louridas et al, 2008; Concas et al, 2007; Baxter et al, 2006). Of closest interest to our work are two papers (Zhang, 2008; Pierret and Poshyvanyk, 2009) source code lexical tokens against Zipf law's in the same manner as natural language. Both find that source code unigrams do largely follow Zipfian patterns, both in Java (Zhang, 2008) and in several additional languages (Pierret and Poshyvanyk, 2009). Zhang explores dividing Java tokens into five categories and remarks on the similarity of java keywords to the natural language concept of stop words. However, neither paper explores the the Zipf curves of programming languages directly with natural language for comparison purposes. We will use Zipf curves in addition to language models so that some language features can be confirmed in a environment agnostic to the choices of a particular language model.

Table 1 Summary of the size and vocabulary of the programming language corpora

Language	# of Tokens	# of Unique Tokens	Projects
Java	16797357	283255	12
Haskell	19113708	473065	100
Ruby	17187917	862575	15
Clojure	12553943	563610	561
C	14172588	306901	10

3 Materials and methods

3.1 Data

We collected many different kinds of natural and programming language corpora. Below, we shall describe how each were collected, along with any modifications made to them for our experiments.

3.1.1 Programming Language Corpora

We focus most on Java and English; however, we empirically confirm that the Java/English difference also applies to several programming languages, including some functional languages. We gather source from OSS projects written in Java, Haskell, Ruby, Clojure, and C. We chose Java and Ruby due to their popularity on GitHub and Java in particular due to its past use as a research subject for ngram models (Hindle et al, 2012; Allamanis and Sutton, 2013). We also add C as well due to its historical significance as a procedural language. Haskell and Clojure are among the most popular functional languages on Github. Two requirements were used when selecting projects for our corpora: (1) the combined size of the projects chosen for each language were roughly equivalent and (2) the projects did not overlap too much in shared domain or source code.

Due to differences between the more and less popular languages, we cannot adopt exactly the same selection criteria for each language. On Github, developers mark projects they want to follow with stars.[6] These stars are a proxy for popularity (Tsay et al, 2014), which we use to choose projects in very popular languages like Java, Ruby, and C. For these languages, we manually selected the projects by examining the list of most starred projects and carefully reading the project descriptions. We chose projects such that they were both popular, and that their descriptions indicated that the project purpose did not overlap in domain.[7]

The functional languages, Haskell and Clojure, are not as popular. After the few most popular projects, the code size of each new project drops drastically. As having significantly smaller training data can negatively affect model performance, we decided that having corpora be roughly equivalent in size was more important than domain diversity. Many more projects are needed to provided sufficient data. This makes manually selecting diverse projects unfeasible, especially as the smaller projects often lack meaningful descriptions.

We thus use an automated process that focuses first on collecting a sufficient amount of data, but still apply some constraints to filter out less meaningful projects and avoid projects that share code. First, we use GHTorrent (Gousios and Spinellis, 2012) to obtain a list of all

[6] `https://help.github.com/articles/about-stars/`

[7] One exception for Java is the Eclipse project, which was not hosted on GitHub, but is selected for significance within the Java community

Table 2 Summary of the size and vocabulary of the English and other natural language corpora

Category	Corpus	# of Tokens	# Unique Tokens
General English	Brown	1209052	48675
	1-Billion Sample	16444921	186864
Specialized English Texts	NASA	302582	10965
	US Code	2800633	29752
	Commit Messages	1933678	74280
	Scifi	1541467	40700
	Shakespeare	1027515	27218
	Recipes	1388875	14328
English as a Foreign Language	Gachon	3063661	40116
	Teccl	2108397	35806
Other Natural Languages	German	17007990	710301
	Spanish	16955041	453133

non forked projects in the language on Github, and select those with over 100 commits. Any project whose name directly contains the name of another project on the list is removed. We then parsed the git logs to verify the GHTorrent results and remove any projects under the commit threshold or with only 1 contributor.

Finally, as we wish to avoid projects including that share significant amounts of exactly copied code, we remove projects that share overly similar directory structures. For each project, we build a set of names, where the each name is a source code file and the directory immediately above it. Then, we use the Jaccard index to compare these sets of names. This index takes the intersection of the two sets and divides it by their union. Any pair of projects that share more than 10% of the their names are thus excluded. In deciding which of the two projects to keep, we remove one if it is an obvious fork of the the other, or if it conflicts with several projects. Otherwise we pick whichever project is larger in bytes, or if they are the same, delete one arbitrarily.

Then, for the projects selected for each programming language, we selected all files associated with the primary file type for that language. We took *.java*, *.clj*, *.hs*, *.rb*, and *.c/.h* files for Java, Clojure, Haskell, Ruby, and C respectively. We use the Pygments syntax highlighting library[8] in python to divide the code in tokens, and separate them with spaces, ignoring comments and removing indentation and other whitespace. Additionally, we treat the content of strings as three units, giving a token to the opening and closing quotes, but removing all spacing within the string and count it as one individual token. For example of what one tokenized line looks like, the line *return EpollSocketTestPermutation.INSTANCE.socket();* is represented as *return EpollSocketTestPermutation . INSTANCE . socket ( ) ;*.

Table 1 shows the size in tokens and projects of the resulting corpora. We see that all the language sizes fall in roughly the same order of magnitude, though the number of projects needed to achieve the size varies.

3.1.2 English Corpora

We drew on a variety of natural language corpora to capture general characteristics of writing, those specific to writing produced by English language learners, and the differences in English technical and non-technical language. We will describe each corpus in turn below; summaries of all English (and other natural language) corpora are located in Table 2.

[8] `http://pygments.org/`

First, for general purpose English, we began with the topically balanced Brown Corpus (Kučera and Francis, 1967), provided by the NLTK project (Bird, 2006). While well balanced, this corpus is small for modern statistical language modeling, so we also used as a general English corpus a 1 billion token benchmark corpus (Chelba et al, 2013). As noted previously, it is important that the language models are explored to roughly equivalent sized training sets, and 1 billion tokens is orders of magnitude larger than our code corpora. Thus, we select a random sample of this corpus, ending up with approximately 17 million tokens – about the same size as the programming language corpora.

Although English is our primary example of natural language, we also consider two other natural language corpora, German and Spanish, to verify that our results are not specific to English and rather apply to other natural languages. These are only used to in the initial experiment, aimed at seeing how well the comparison of the differences in repetitiveness of Java and English holds across various programming and natural languages. The German and Spanish corpora were selected from a sample of files from the unlabeled datasets from the ConLL 2017 Shared Task (Ginter et al, 2017), which consist of web text obtained from CommonCrawl.[9] Like the 1 billion token English corpus, we selected a random subsample to make these corpora size comparable with our other corpora. In this sample, we excluded files from the Wikipedia translations, as we observed Wikipedia formatting mixed in with some of the files. Summaries of the vocabulary token counts of these corpora are also in Table 2.

To test hypotheses about language difficulty and repetition, we used two english language learner corpora. The Gachon (Carlstrom and Price, 2013) corpus is a collection of primarily Korean, but also some Chinese and Japanese English language learners. The Gachon corpus covers a range of just over 25K 100 to 150 words answers to 20 essay questions. It has meta information including the years a student has studied the language, the their native language, and their TOEIC language proficiency score[10]. While this corpus contains explicit information about the writer's language proficiency, it does suffer from a confounding effect of being limited in domain to merely 20 topics. Domain specificity is known to make corpora more predictable and repetitive (Hindle et al, 2012). Therefore, we include the Teccl Corpus (Ten-thousand English Compositions of Chinese Learners) (Xue, 2015) as another example of *EFL* for robustness. Unlike the Gachon corpus, Teccl covers a much wider range of topics (the authors estimate around 1000). It consists of a wide range of writers in both geographically and in current education level.

The question of technical and imperative language is also confounded with the possibility of restricted domain. Therefore we selected six corpora, three technical corpora, two non-technical corpora with potentially restricted domain, and a corpus of instructions in the form of cooking recipes. The two non-technical corpora came from literature: a corpus of Shakespeare's works (Norvig, 2009), restricted in domain by having the same author, and a corpus complied of 20 classic science fiction novels from the Gutenberg corpus[11], which all fall under the same literary genre.

For the technical and imperative language corpora, we selected a corpus of NASA directives, a corpus of legal language, a corpus of commit messages, and a corpus of cooking recipes. The NASA directives were scraped from the NASA website. Directives share similarities with source code requirement documents, a written English equivalent to source code. Source code requirements explain in detail what is expected from a software applica-

[9] `http://commoncrawl.org`
[10] `https://www.ets.org/toeic`
[11] `https://www.gutenberg.org/`

tion, and the requirements documents of the NASA CM1 and Modis projects have been used in many requirements studies (Hayes et al, 2005; Sundaram et al, 2005). However, the requirements documents for the two NASA projects often used in these studies are only about 1.2K words for Modis, and 22K words for CM1. Language models typically require far more words, we mined the more general NASA directives, creating a corpus approximately 245K words long.

Among technical corpora, one type of corpus of special interest are English documents surrounding the source code process. In addition to their technical and domain limited nature, these documents, like code, are also written by developers. One could argue that programmers as authors may simply be more likely to use repetitive patterns in all of their writing, whether text or source code. If such language demonstrates the same repetitiveness as code, then it would support the idea that the repetition comes from the type of author of the text. Therefore, we considered several sources of texts likely to be written by those in the development community, including stack overflow posts, GitHub issues and pull requests, and commit messages. Ultimately, we selected commit messages as our example as we observed the other corpora more frequently had a dual language problem - they included both English and source code in the text. As separating the two languages is often non-trivial, commit messages effectively fulfill the corpus requirements.

For our corpus of commit messages, we began from a sample of 200 of the top 900 most starred GitHub projects, coming from a dataset mined for a study by Kavaler et al. (Kavaler et al, 2017). Initial exploration into this corpus lead us to observe the frequent presence of URLs along with some automatically generated segments. To normalize these commit messages, we replaced URLs with a special tag, and then removed all lines starting with ”git-svn-id”, as they represented a highly repetitive automated pattern not representative of real programmer written English. We then took a random sample of these commit messages to obtain a corpus of roughly equivalent size to all of our other specialized English corpora.

Legal language, like code tends to be *prescriptive* and *precise*. Just as code variables and functions regularly reference other parts of the code, so to do references within legal text. For this purpose, we downloaded the US Legal Code [12]. The US legal code consists of 54 major title sections relating to the general permanent federal law of the United States.

Finally, we use a recipe corpus as a study of relatively precise, purposeful imperative language usage. This corpus comes from the text found in the million recipe corpus (Salvador et al, 2017). Like source code, recipes are instruction sequences, though the degree of precision required in the writing is lower. In order to make this corpus comparable to our technical corpora, we selected a random sample of the recipes with total textual size of about 1 million tokens. The full corpus contains images, ingredients, and instructions associated with each recipe. For our purposes, we only considered the *instructions* text for each recipe as input into our models.

3.1.3 Parse Tree Corpora

For our parse tree comparison experiment, we needed to extract an abstract syntax tree for a software corpus, and represent it in a similar fashion to natural language constituency trees (as described below). This experiment was limited to our Java and English data. When comparing the parse trees, we first selected constituency parse trees for written English from the Penn Treebank (Marcus et al, 1993), which includes sections from the Brown Corpus and the Wall Street journal corpora. Then, we used a modified version of the Eclipse Abstract

[12] `http://uscode.house.gov/download/download.shtml`

Table 3 Summary of corpora token counts and vocabulary for the modified English and Java parse trees

	Java Trees	English Trees
All Tokens	11267469	11354764
Terminal Tokens	2191014	1740902
Simplified Non-Terminal Vocabulary Size	81	93

Syntax Tree parser to transform all the files in our Java corpus to English. Since the Java trees could be automatically created, we randomly sampled from these Java files in order to make the corpora roughly size equivalent in token count to the Penn Treebank trees. Details on the modifications made to make the two trees more comparable are described in section 3.4.

Table 3 shows the sizes of the resulting corpora. We see that the trees have roughly the same number of non-terminal tokens, but that the number of distinct rules is much larger in English than in Java. Likewise, the Java trees have about half as many terminal tokens.

3.2 Comparing Language Repetition and Predictability

As introduced in section 2.4, we use two general methods for measuring the repetition of language corpora. The first involves the reporting the entropy per token as described in section 2.4.1. The details of the modeling and the representation of the results can be found in section 3.5.

The redundancy of corpora can also be modeled using a variant of the Zipf plot (Zipf, 1949). In a standard Zipf plot, we count all occurrences of a word in a text and assign each word a rank based on frequency. The x-axis is the rank of the word, and the y-axis is its frequency. When plotted in log-scale, this relationship appears roughly linear. We modify this plot in two ways. First, we normalize the frequencies on the y-axis to a percentage to make different corpora more comparable. Second, we extend the idea of a Zipf plot beyond merely individual word frequencies to *word sequence* frequencies. Counts bigrams, trigrams, or higher order ngrams, helps make the distribution of phrase usage more apparent. In more repetitive texts the most frequent phrases constitute proportionately more of the text. On a log-log plot, we can visualize this effect (roughly) as the power law slope of the data. More repetitive texts begin higher on the y-axis and descend more steeply. Once normalized, corpora with steeper slopes demonstrate a greater frequency of repetitive phrase use; those with shallower slopes are show greater innovation.

Using Zipf plots to assess corpus repetition averts some of threats from using language models. To use an LSTM, the vocabulary size must be limited by removing infrequent words, which would artificially affect results for these words. There is no such limitation in the Zipf plots, which increases the robustness of the overall observations.

3.3 Measuring Open Category Words

To test the hypothesis that differences in closed category words account for most differences between source code and English, we remove the closed vocabulary words from a corpus, and leave behind just sequences of open vocabulary words. Removed are elements most closely tied to the language syntax; arguably, what remains are *content words*. These most closely model the sequence of *ideas* expressed by the text.

How do we determine what tokens qualify? For English, we use a list of 196 words and contractions, along with a list of 30 punctuation markers, derived from a published NLTK stop word list (Bird, 2006). For our programming languages, we use the Pygments type categorization (implemented with regular expressions) to remove non-identifier words, keeping references to types, classes (when applicable), variables, and function names. Specifically, we labelled as open category tokens that Pygments had marked as Token.Name (but not the subtype Token.Name.Builtin), Token.Keyword.Type, Token.Literal.String, or Token.Number with a few modifications. These modifications involved some small changes to keyword lists and are intended to make the closed category words more consistent across the different programming languages. For example, we classified the boolean (*true/false* and *null* literal values as closed category. We also extended the list of what Pygments considered keywords in Haskell[13], Ruby[14], and Clojure[15]. These labels only approximate the open category words, but they do remove operators, separators, punctuation, and most keywords. If these sequences of content words are more repetitive in source code than in natural language, this would be consistent with the theory that the repetition in code is not wholly due to syntactic constraints. Below are examples of what part of these filtered sequences would look like in Java and English respectively:

... InputStream in FileInputStream file ByteArrayOutputStream out ByteArrayOutputStream byte buf byte 8192 ...

... Now 175 staging centers volunteers coordinating get vote efforts said Obama Georgia spokeswoman Caroline Adelman ...

One consideration for these open category words in code is the question of how to handle literal values. In the case of strings, an argument could be made that many of them would qualify as being natural language, leading to a dual language corpus. We compared the code corpus open category words both with and without the literal values included, but found little difference in the overall trends from our language models and Zipf models, though the exact size of the differences changed. Presented in this paper are the results of the corpora with the literal values included.

A potential threat to this experiment results from the fact that English open and closed category words are fairly well defined, but far less so for programming languages. Pygments provides a good approximation (which we try to further improve), there are some corner cases. Some language elements are very common and difficult to extend without strictly being on the official list of reserved words, or could be construed as part of a larger category that *is* open category, such as primitive types like *int* in Java can be seen as belonging to the larger open category of *types*[16]. We argue that these edge cases are infrequent enough and the size of the effects observed in our experiments are large enough that drawing the boundaries between open and closed differently would only slightly impact our results[17].

[13] We add \, *proc*, *forall*, *mdo*, *family*, *data*, and *type*.

[14] We add *__ENCODING__*, *__END__*, *__FILE__*, and *__LINE__*.

[15] We add *recur*, *set!*, *moniter-enter*, *moniter-exit*, *throw*, *try*, *catch*, *finally*, and */*, along with some operators Pygments had classified as Token.Names

[16] In particular, we called these primitives types open category to be consistent with how other programming languages like Haskell treat their types.

[17] Additionally, in our experience, tweaking the boundaries of these categories results only in slight changes in repetition.

3.4 Creating Equivalent Parse Trees in Java and English

While the syntax of Java and English strings can be unambiguously represented with a tree data structure, the trees themselves are quite different. First, Java grammar is explicitly defined, where as English grammar is at best an imprecise model of an evolving reality. Second Java parse trees are also *abstract*, and omit some tokens present in the original text: *punctuations* (*e.g.* 1`"{ }", ";", ".", "+", "-"`) and some reserved keywords. In contrast, the constituency parse trees of English are *concrete*, comprising all tokens in the original text. Thus, the vocabulary size differences could confound the interpretation of comparisons of repetition: lower vocabulary, more chance of repetition. Finally, the syntax trees in Java and English represent different granularities. In Java a complete *AST* describes an entire file; in English, the tree describes a sentence. Thus, the code *ASTs* are both encompass for tokens and have longer paths from the root to the leaves.

How can we account for some of these differences and create a more fair comparison? First, we use a highly reliable English constituency parse – that from the Penn Treebank (Marcus et al, 1993) (PTB). This includes about 200 files of the 500 file Brown corpus, with an additional text from the Wall Street Journal. All parses have been manually corrected by linguists to ensure accuracy; Indeed, PTB is a standard choice for training/evaluating other automated syntax parsers for English (De Marneffe et al, 2006; Petrov et al, 2006; Andor et al, 2016). Automated methods for creating parses of English exist (De Marneffe and Manning, 2008; Petrov, 2016), but they are not always accurate. To focus on the actual grammatical structure rather than an approximation, we choose the human annotated parse trees as our corpus.

Second, we modify both trees to make them more comparable. For Java, we modify the tree to be *concrete* instead of abstract. We created a new category, called *PUNCTTERMINAL* for all terminal tokens typically missing from an *AST*, giving a total of 81 non-terminal tags. These new nodes are inserted into the syntax tree such that during preorder traversal, the terminals will appear in the same order as in the original text – a feature that is already true of the constituency parse trees. In the English parse trees, we consider the effect of reducing the size of set of non-terminal tags to be closer in size to the set of Java nonterminals. The PTB includes tags with multipart labels indicating both constituent and function (for example *NP-2*, *PP-TMP*, *ADVP-TMP-PRD*[18]). We reduce this set by retaining only the grammatical *category* label, such as *ADVP*, of an adverbial phrase, leaving out additional tags such *TMP* that reflect grammatical *function*. After this reduction, we have a total of 93 syntactic categories for English. To verify whether this reduction could unfairly penalize the language models ability to learn the grammar, we consider results on both the original unmodified tags and on the simplified tags. In our plots, we will refer to the modified English trees as *simplified*. That the English trees capture sentences and the Java trees capture files remains an intrinsic difference between them and a possible threat, but these changes at least make the trees contain similar organization and content.

Figures 3 and 4 display examples of what each of these trees look like for English and Java respectively. Note how the changes to the Java tree ensure that both trees produce the original text in the same left to right order. The tags used for English are described by the Penn TreeBank (Marcus et al, 1993), and the tags for the Java AST come from the eclipse ASTNode class.[19]

[18] (e.g. *ADVP-TMP* reflects that the adverbial phrase serves a temporal function).

[19] `https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTParser.html`

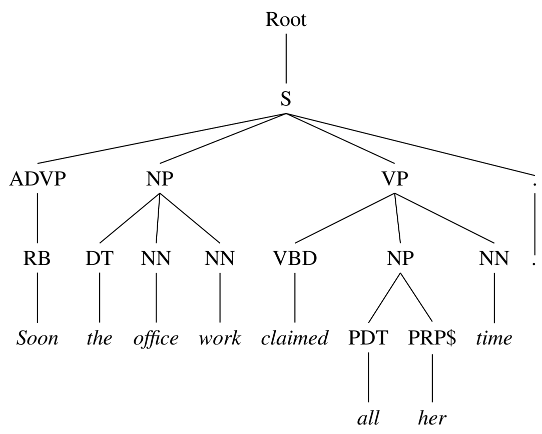


Fig. 3 A example CPT from one of the sentences from the Penn Treebank along with the reduced tag sets

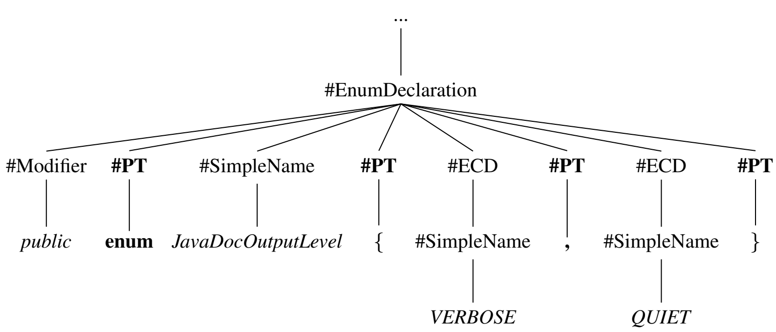


Fig. 4 An example of part of a modified AST capturing a single line of Java. The bolded tags correspond to nodes added to ensure the tree contains all tokens from the original text (PT = PunctTerminal, ECD = EnumConstantDeclaration)

To measure the entropy of the terminal tokens, we linearize the tree using preorder traversal; this presents the non-terminals as context for the terminal symbols and also retains the order of the words originally in the text. Then, we apply our language models to this linearized parse tree. Prior work indicates that LSTMs can capture the syntax of the grammar in this form(Vinyals et al, 2015). Importantly, by examining *only* the entropy values of the terminal tokens, we account for the differences in the complexity of the grammars[20]. Given the extra information the grammar provides, we can see how the differences in terminal entropy between Java and English changes. The more the gap reduces, the more the differences in the language can be attributed to the grammar instead of some other contingent factor. Finally, while the theoretical grounding for capturing the grammar's of the trees has only been found with neural models like our LSTM, we also include results from the simpler ngram and cache models for completeness.

[20] Indeed, when running LSTMs over just the nonterminals, we see that the Java grammar is more predictable than the English grammar.

3.5 Modeling Details

Our ngram models were estimated using KenLM (Heafield, 2011) with modified Kneyser-Ney smoothing (Kneser and Ney, 1995), based off of the code used by Tu et al. (Tu et al, 2014). For the raw texts of all English and programming language corpora, we use a trigram model as the base. When comparing the parse trees, we use instead a 7-gram model to capture more information about the sparser context. This was determined empirically by modeling parse trees with ngram models from 2 to 9-grams, and observing no further improvement after the 7-gram level. In our cache models, we use a 5000 token window cache with 10 tokens of context. Our LSTM models are implemented in Tensorflow (Abadi et al, 2016), with a mini-batch size of 20, 1 hidden layer of 300 units, a maximum of 13 training epochs, no dropout, and a learning rate of 1.0. Additionally, to see the effect of scaling the LSTM models to a larger one for our parse tree experiment, we also used a model with 2 hidden layers of size 650, a dropout rate of .5, and a maximum of 39 training epochs. We will prefer to these models as the *small* and *medium* sized LSTM models going forward. These settings are similar to those used by Hellendoorn et al (Hellendoorn and Devanbu, 2017).

Our corpora tend to have large vocabularies, which need to be limited in order for the LSTM models to complete within a reasonable timeframe. Likewise, new unseen tokens can always appear in the test set. This is especially true in source code, where new variable names can be easily created and used in new localized contexts (Tu et al, 2014). Theefore, ngram language models use smoothing (Chen and Goodman, 1998), which reserves some probability mass for unseen words. We limit our vocabulary size to the most frequent 75000 distinct tokens, replacing the least frequent words with with a special “unknown” token (*UNK*).

For the LSTM models, we split each code corpus at the file level with 70% of files in the training set, and 15% each in the validation and test sets. We do that same for the natural language corpora if they come with files; otherwise, we divide them into small chunks which are combined into training, validation, and test sets with the same splits. The ngram models do not use a validation set, so we combine the validation and training sets when training them. While we tried to use consistent training and testing sets across our language models, we had a few instances where Kenlm crashed during training due to errors estimating the smoothing discounts. In the open category experiment, we had to select subsets of the training data for Ruby and Haskell in order for the models to train correctly. We selected the largest continuous segments of the training data that completed successfully, ending up with 5.7 million tokens and 8.1 million tokens for Ruby and Haskell respectively. The test sets for these corpora were unaffected. The other exception was that giving the vocabulary capped version of the Java parse tree to the KenLM model caused an error. Therefore, the training and test sets for the LSTM and ngram models for the Java parse tree are not exactly comparable. As we are primarily concerned with the LSTM results and cross language comparison, this is does not have an impact on our results. The English parse tree did not need to be capped as its vocabulary was below 75000, so these comparisons are unaffected.

When comparing the results of the language models, we report the per-token estimated entropy values. This forms a distribution of entropy values, which we compare visually with box plots and quantitatively with two sample statistical tests. The distributions of entropy are often long tailed, violating the assumptions of the t-test, so we instead us the non-parametric Mann Whitney U Test (also commonly referred to as a Wilcox test), to compare the distributions. We report the significance of the test, a 99% confidence interval for the true difference in the median value of the distributions, and a effect size r, which can be interpreted similarly to a Cohen's-d value (Field, 2009). These tests and confidence intervals where implemented in R using the *coin*(Hothorn et al, 2006) package, and plots were created using *ggplot*(Wickham, 2009).

There are several potential threats to the validity to consider in our modeling choices. While we have used several language models and tried to use random sampling to make each corpora comparable, we cannot say how the results might change with a much larger corpus. For some corpora, like general Java or English, one can easily get billion token corpora. But for more specialized corpora or less popular programming languages, the pool of what is available is much smaller, and limits how much we can use from the larger corpora. Otherwise, the effects observed in the models could simply result from larger amounts of training data. We selected training, validation, and test sets randomly, but a different split could produce different results. A more robust method would be to use 10-fold cross validation, but given the large number of corpora and the training time necessary to train the LSTM models, this was not feasible.

4 Results

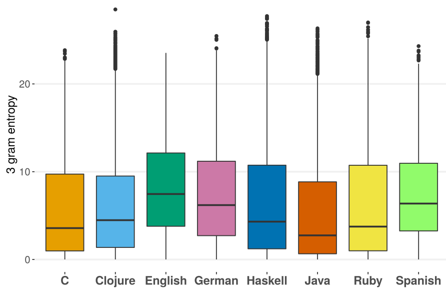
We now present results, structured as follows.

(i) We examine if the Java-English difference is consistent in other programming languages and natural languages.
(ii) We compare the repetitiveness of open-category words of each programming language with those in English.
(iii) We explore the syntactic structure of Java and English to see what parts of the structure of each contributes to differences in repetition.
(iv) Finally, we compare source code with English language learner and technical corpora to see if the expected characteristics of each make them more code-like.

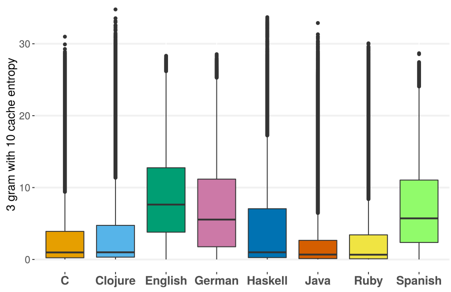
4.1 Repetition in Natural Languages and Various Programming Languages

Table 4 Summary of non-parametric effect sizes and 99% confidence intervals (in bits) comparing each code and natural language corpus with English a baseline. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a Mann Whitney U test

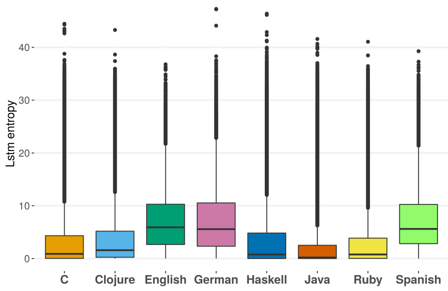
Language <English	Ngram	Cache	LSTM
German	(-0.921, -0.897) 0.088***	(-1.585, -1.56) 0.145***	(-0.182, -0.161) 0.02***
Spanish	(-0.662, -0.639) 0.064***	(-1.38, -1.355) 0.127***	(-0.055, -0.035) 0.005***
Java	(-2.974, -2.951) 0.285***	(-5.422, -5.398) 0.562***	(-4.292, -4.272) 0.564***
C	(-2.586, -2.559) 0.242***	(-4.93, -4.901) 0.488***	(-3.581, -3.557) 0.44***
Clojure	(-2.138, -2.115) 0.203***	(-4.755, -4.728) 0.479***	(-3.075, -3.053) 0.372***
Ruby	(-2.338, -2.314) 0.219***	(-5.12, -5.095) 0.516***	(-3.691, -3.671) 0.469***
Haskell	(-2.059, -2.036) 0.191***	(-4.148, -4.139) 0.405***	(-3.443, -3.423) 0.431***



(a) 3 grams



(b) 3 grams with cache



(c) LSTM (Small)

Fig. 5 Entropy score distributions for each of our programming and natural language corpora, using ngram, ngram-cache, and lstm models. Each data point used in the box plot is the entropy score for one of the tokens in the test set

Fig. 5 shows entropy distributions over all tokens from various language models for Java, Haskell, Ruby, Clojure, C, English, German, and Spanish. First, we clearly replicate the prior results comparing Java to English (e.g. (Hindle et al, 2012)), across many programming and natural languages. Regardless of the language model used, all of the programming languages are more predictable than English and the other natural language corpora. Second, Table 4, shows that these differences are significant. Indeed, programming languages are usually several bits more predictable than English. The other natural languages, German and Spanish, are somewhat more predictable than English with ngram models, but about the same with the LSTM model. The non-parametric effect sizes of the differences between programming languages and English vary from *small* to *medium*.

Tab. 5 also shows the improvement when a cache model is used to capture the *locality* of the corpus. As expected, the basic trigram models perform the worst on all the code corpora. The cache improves all of the programming languages, albeit to various degrees. For natural language, the cache has no effect in English, as previously reported (Tu et al, 2014). However, in German and Spanish, there is a small cache effect, much smaller than seen in any programming language. Our small 1-layer LSTM model dominates the basic ngram models significantly, but their improvement over the cache models are variable. The LSTM is better for English, but not for German or Spanish, and among the programming languages Ruby and Clojure see almost no difference. Haskell, gains the most from the longer context of the

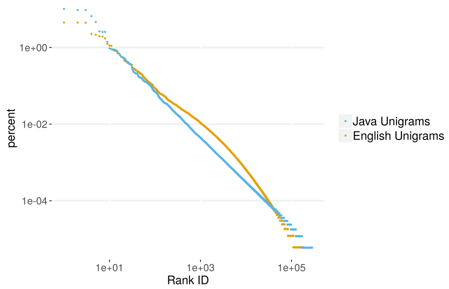
Table 5 Summary of non-parametric effect sizes and 99% confidence intervals of the difference (in bits) of language. The columns compare how many bits higher the entropy of model on the left is from the one on the right. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a Mann Whitney U test

Language	Ngram > Cache	Cache > LSTM
English	(-0.218, -0.193) 0.020***	(1.459, 1.484) 0.140***
German	(0.488, 0.511) 0.051***	(-0.01, 0.003) 0.001
Spanish	(0.582, 0.604) 0.060***	(-0.049, -0.027) 0.004***
Java	(1.240,1.255) .275***	(0.148, 0.152) .178***
Haskell	(1.484, 1.504) .220***	(.265, .272) .158***
Ruby	(1.770, 1.795) 0.327***	(0.004, 0.005) 0.033***
Clojure	(2.006, 2.023) .318***	(-0.038, -0.032) .024***
C	(1.418, 1.442) .271***	(0.091, 0.096) .082***

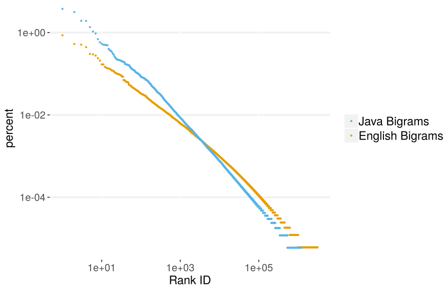
LSTM model, suggesting that its syntax may have less localized repetitiveness compared to the other languages, which lines with common beliefs about the language's conciseness. We note however, that these are smaller LSTM models - larger more expensive models would likely perform better. However, this is not the focus of our paper, and a detailed look at the question of deep models versus ngram cache models in Java was performed by Hellendoorn and Devanbu (Hellendoorn and Devanbu, 2017).

Fig. 6 contains Zipf curves for only Java and English for unigrams, bigrams, and trigrams. The increased repetition of source code over English widens the gap between the slopes as the length of the n-gram increases; longer sequences are repeated even more in Java than in English. However, the English curve exhibits a noticeable bend that the Java unigram curve lacks. This behavior agrees with past studies of such curves in English (Ferrer i Cancho and Solé, 2001; Gerlach and Altmann, 2013; Piantadosi, 2014; Mitzenmacher, 2004), as is better modeled with a bipartite double pareto curve as previously described in 2.4.2. Theoretically, this bend results from a decreasing need for new vocabulary in English, whereas in code new identifiers can be created with every file - the basis of the effectiveness of cache models (Tu et al, 2014). However, this double-pareto behavior is not pertinent to our main experimental question of comparing repetitiveness, so we do not delve into it here.

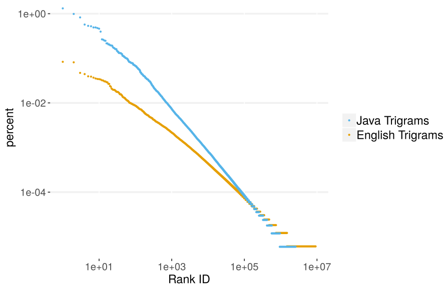
We extend the Fig. 6 Zipf plots to cover all our programming and natural languages in Fig. 7, and a range of behaviors are observed. All of the programming languages have steeper slopes than the natural language corpora, but not all exhibit the same level of repetition. Of the source code corpora, the Haskell bigrams and trigrams fall midway between the natural languages and the other source code languages. This aligns with the behavior we saw in the cache models - suggesting that Haskell's syntax may have less local repetition that other programming languages. The other programming languages are more closely grouped together, with no clear distinction between them. From here on, while comparing programming *vs* natural languages, we use English as a proxy for other natural languages.



(a) Unigrams



(b) Bigrams



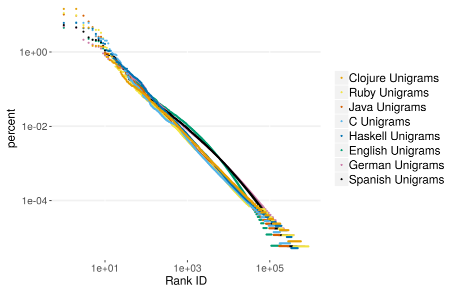
(c) Trigrams

Fig. 6 Comparison of slopes for Zipf plots of Java and English unigrams, bigrams, and trigrams. The axes are in log scale. Higher percentages in low ranks indicate a more repetitive corpus, as can be seen by the diverging slopes between Java and English

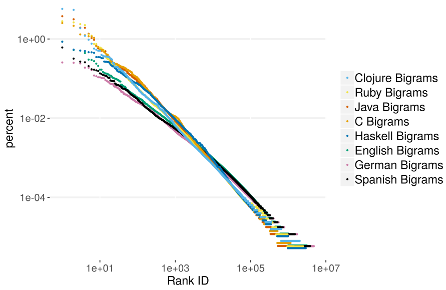
4.2 Modeling just the Open Vocabulary Words

Table 6 shows the size of two corpora after tokenization before and after closed category word removal. Three of the programming language corpora (Haskell, Ruby, and Clojure) exhibit a similar amount of closed category word usage as English, with C and Java having about 10-12% less proportionately. Existing work by Allamanis et al. has shown closed category tokens in code to be much more predictable than identifiers (Allamanis and Sutton, 2013), But since English does not have proportionately more open category words than code, we cannot attribute the additional ease of predicting programming languages simply to an increased amount of closed category tokens. However, the difference could still result if these closed category tokens are far more predictable in code than in English. As we shall see shortly, this is not the case.

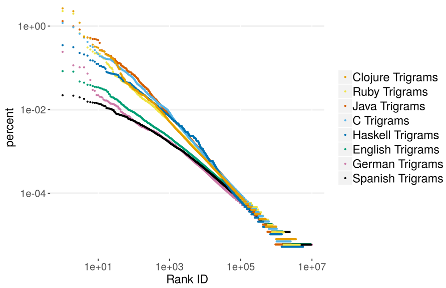
Fig. 8 shows the Zipf slopes of the of the open category-only unigrams, bigrams, and trigrams. The unigrams in code are roughly equivalent to that of English, except for the curved nature of the Zipf line. This is again explained by the theory of kernel and unlimited lexicons discussed in section 2.4.2 - natural language open category words appear in both the kernel and unlimited lexicon in English but there is a decreasing need for them as the vocabulary grows. As we move from unigrams to bigrams and then trigrams, we see a similar separation in the Zipf plots lines as was seen in the full texts. In all programming languages,



(a) Unigrams



(b) Bigrams



(c) Trigrams

Fig. 7 Unigram, bigram, and trigram Zipf Slopes for all 5 of our different programming languages as compared to our 3 natural language corpora. The other programming and natural languages exhibit similar behavior to Java and English

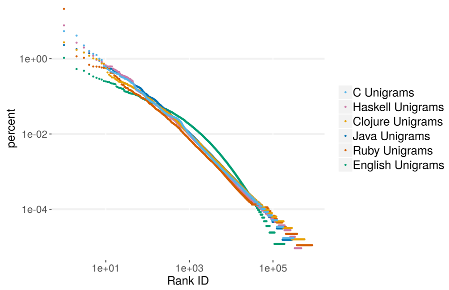
Table 6 Summary of the fraction of open category tokens to all tokens in English and programming languages

	All Tokens	Open Category Tokens
English	16444921	8340320 (50.7%)
Java	16797357	6469474 (38.5%)
Haskell	19113708	10803544 (56.5%)
Ruby	17187917	8992955 (52.3%)
Clojure	12553943	6286549 (50.1%)
C	14172588	5846097 (41.2%)

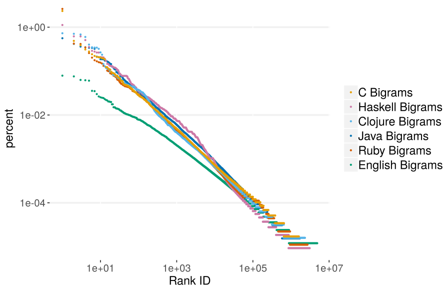
the open category word-sequences are more repetitive than English, though the amount of repetition varies.

Fig. 9 confirms this intuition of content word repetition in source code; the open category words of English are more predictable than those in programming languages. Table 7 quantifies these differences with Wilcox tests, showing that the difference for all distributions is significant and varies from a few *small* to mostly *medium* effect sizes. Java, Haskell, and Ruby open category words tend to be more predictable, while C and Clojure names are more difficult to predict.

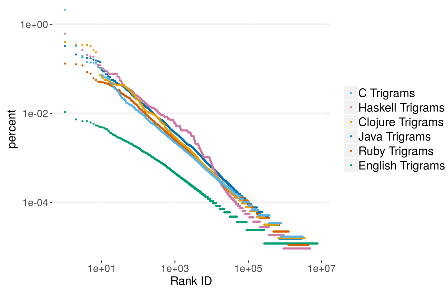
When contrasting the median difference in entropy, all of the programming language open category words are at least 4 bits more predictable than the English ones, and the difference is often substantially higher. In fact, the median difference between the program-



(a) Unigrams



(b) Bigrams



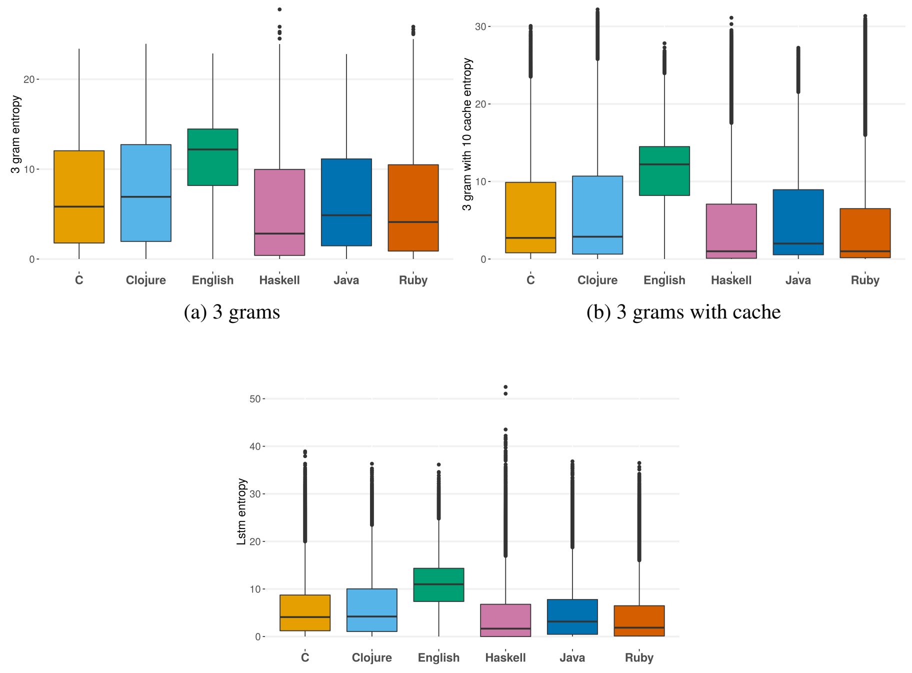
(c) Trigrams

Fig. 8 Unigram, bigram, and trigram Zipf plots comparing English open category words with programming language open category words

Table 7 Summary of non-parametric effect sizes and 99% confidence intervals (in bits) comparing the median of the entropy distribution of open category English words with those of several programming languages. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a Mann Whitney U test

Language <English	Ngram	Cache	LSTM
Java	(-5.507, -5.462) 0.403***	(-7.377, -7.335) 0.5***	(-6.618, -6.58) 0.505***
C	(-4.715, -4.673) 0.335***	(-6.858, -6.811) 0.446***	(-5.826, -5.784) 0.435***
Clojure	(-4.112, -4.065) 0.306***	(-6.641, -6.594) 0.444***	(-5.463, -5.42) 0.397***
Ruby	(-6.021, -5.98) 0.437***	(-8.543, -8.507) 0.57***	(-7.518, -7.485) 0.567***
Haskell	(-6.823, -6.785) 0.463***	(-8.427, -8.392) 0.552***	(-7.555, -7.522) 0.557***

ming languages content words and English context words is larger than when considering all tokens, though the size of this increase varies. Note that when compared to the distributions of entropy of the full corpora seen in Fig 1, the the open category words are less predictable, as expected from existing research (Allamanis and Sutton, 2013). Finally, if we exclude the literal values in the code corpora from open category words, we get similar re-



(a) 3 grams

(b) 3 grams with cache

(c) LSTM (Small)

Fig. 9 Entropy distribution comparisons of English and the programming language open category words from an ngram, cache, and LSTM model

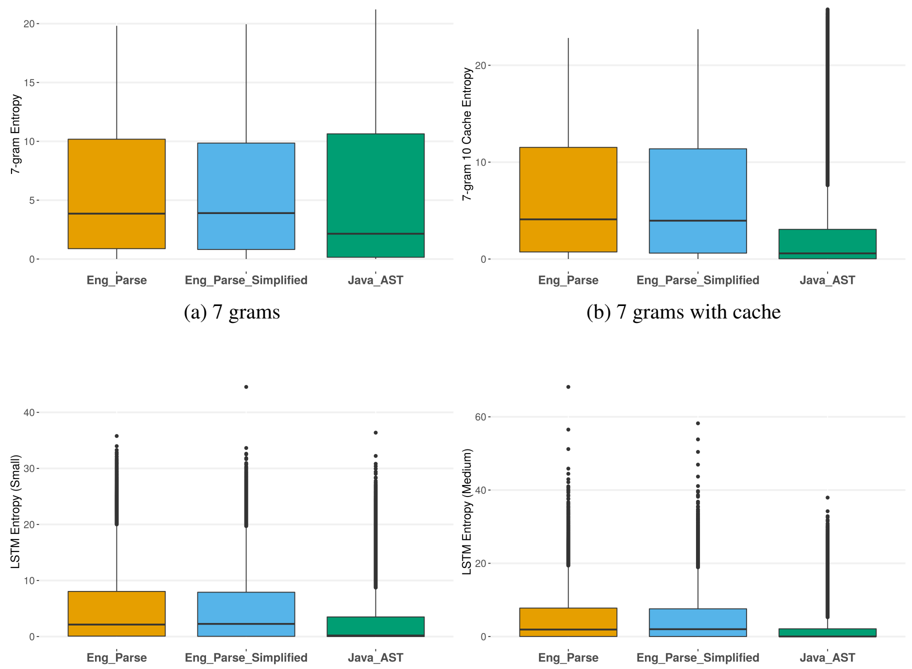
sults[21]. Therefore, in answering RQ 1, we see that while content words are in general less predictable, code content words not only easier to predict than English content words, but also the difference in predictability is accentuated!

4.3 Parse Tree Results

Fig. 10 shows the entropy comparisons of the *terminal* token distribution for both Java and English when parse trees are taken into account. Though we focus primarily on the entropy distributions of the LSTM models, as they can capture well the linearized tree structure, we will mention the ngram and cache model results briefly. With the ngram model the difference between Java and English drops substantially, albeit not completely. In contrast, the cache model is able to capture proportionally more of the grammar of Java. However, neural models are better able to learn the grammar, and in both the smaller 1 layer LSTM and in the larger 2 layer LSTM Java remains more predictable than English.

We confirm the intuition provided in the box plots in the upper part of Table 8. Each of the differences between the English and Java terminals are significant, and have a *small* effect size in the more capable LSTM and cache models. The effect size in the ngram model

[21] The size of the entropy difference between English and Code open category words is less, though still larger than between all tokens



(a) 7 grams

(b) 7 grams with cache

(c) LSTM (Small)

(d) LSTM (Medium)

Fig. 10 Entropy comparisons of the terminal tokens in the parse trees using ngram and LSTM models

is very small, but it is questionable how well such a simple model can capture the tree syntax; the LSTM results are the most reliable. The median difference between Java and English is roughly 1.5 or 1.6 bits for the cache model, and 0.6 or 0.7 in our LSTM models. The concerns about the effect of simplifying the types effecting the comparison of grammar were unfounded. Using Wilcox tests to compare the simplified and the full non-terminal set revealed no significant difference in the more reliable LSTM models, and a significant but extremely small effect in the ngram and cache models. Finally, to ensure a fair comparison between these languages as parse trees and them as raw text, Table 8 has a column *Original Text*. These are the same set as the terminal tokens in the parse tree[22], but with all tree information removed before language model processing. We see that in the original text, the effect sizes and confidence intervals are all larger, with almost *medium effect sizes* and gaps far greater than 1 bit of difference. Therefore, we can conclude that eliminating the ambiguity of English grammar explains *some*, but not all of the difference in repetition of the language compared to Java.

Additionally, with our medium LSTM 23.4% (small LSTM had 9.9%) of Java terminals had entropy 0, meaning the choice was completely determined by the grammar. In contrast, in the medium LSTMs only about 5.1/5.0% (for the simplified and unsimplified tree) of English terminals had 0 entropy. The small LSTMs had .9%/1.7% tokens that were completely predictable in the English simplified/unsimplified trees. These tokens primarily consisted of

[22] In the simplified parse tree in the case of English.

Table 8 Summary of non-parametric effect sizes and 99% confidence intervals (in bits) comparing the difference in the median of the entropy distributions of the terminal tokens in parse trees from Java and the Penn Treebank. The differences indicate how much smaller the Java distributions are compared to English. Rows labelled with simplified are comparing English trees with simplified non-terminals to the Java trees, and rows without it use the original Treebank tags. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a Mann Whitney U test

Model	Terminal Tokens in Tree	Original Text
Ngram Simplified	(-0.351, -0.293) 0.078***	(-3.411, -3.336) 0.316***
Ngram	(-0.440, -0.400) 0.088***	
Cache Simplified	(-1.557, -1.499) 0.264***	(-5.200, -5.116) 0.479***
Cache	(-1.654, -1.586) 0.277***	
LSTM Simplified (Small)	(-0.621, -0.574) 0.248***	(-4.0680, -3.985) 0.413***
LSTM (Small)	(-0.616, -0.581) 0.257***	
LSTM Simplified (Medium)	(-0.746, -0.695) 0.320***	(-3.441, -3.375) 0.414***
LSTM (Medium)	(-0.706, -0.661) 0.328***	

the punctuation of each language, with occasionally stop words or reserved words in Java. In English, the largest contributor to low-entropy tokens were commas, and in Java it was open parentheses, the dot operator, open brackets, and closing parentheses in decreasing order. The other tokens only made much smaller portions of the 0 entropy tokens.

Thus, we answer RQ 2 somewhat positively - the ambiguity accounted for in the grammar by parse trees does explain some but not all of the difference between natural language and source code. Both this experiment and the previous one suggest that the differences seen between source code and English consist of more than simply syntactic differences. This leaves the possibility that at least some of the difference comes from human choices independent from the grammar.

4.4 Comparing Code with Effortful English Corpora

While humans may *choose* to write more repetitively for various reasons, we present find evidence that greater repetition arises when a) the text more effortful for the writer, or b) when the cost of miscommunication is higher. For the former we focus on English language learner texts, and for the latter we use various technical corpora combined with some non-technical corpora to control for effects of domain specificity. We limit the presentation of our results from these corpora to Java, but we found similar results were found when comparing the other programming language corpora as well.

4.4.1 Comparing English Language Learner Corpora to Code

Fig. 11 shows Zipf plots comparing English with our ESL (English as a second language) corpora and Java. ESL is certainly more repetitive than general purpose English; however, it is not as repetitive as source code. This behavior is confirmed with the language models

Table 9 Summary of non-parametric effect sizes and 99% confidence intervals (in bits) of the median entropy comparing the English Language Learner corpora with Java and the balanced English Brown corpus. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a Mann Whitney U test

Brown >Language	Ngram	Cache	LSTM
Gachon	(-1.729, -1.657) 0.147***	(-1.643, -1.564) 0.132***	(-4.194, -4.106) 0.309***
TECCL	(-1.673, -1.595) 0.152***	(-1.558, -1.475) 0.133***	(-3.767, -3.674) 0.294***
Language >Java (Small)			
Gachon	(-1.501, -1.429) 0.138***	(-4.046, -3.972) 0.403***	(-2.043, -1.988) 0.296***
TECCL	(-1.575, -1.501) 0.153***	(-4.079, -4.004) 0.434***	(-2.461, -2.398) 0.341***

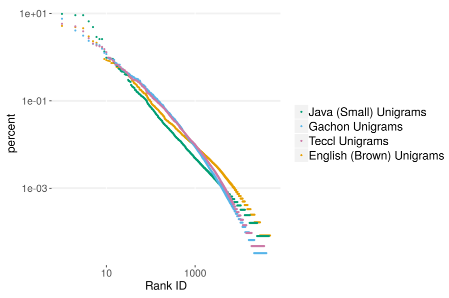
displayed in Fig 12. Regardless of where the more basic trigram model or the increasing the context with the LSTM model, the entropy, like the Zipf slope lines, fall in between source code and general native language written corpora. Table 9 reports p-values, confidence intervals, and effect sizes and confirms that the english language learner texts fall fairly evenly between native English and Java. The one exception is the when using the cache model, where code gains comparatively over both fluent and learner english. Neither exhibits the locality needed to benefit from this model's assumptions.

Thus, we can answer RQ 3 positively. The language learner corpora more closely resemble the repetition in code than does general English. This is consistent with the the hypothesis that less fluency and therefore greater difficulty for writers would result in more repetitive corpora.

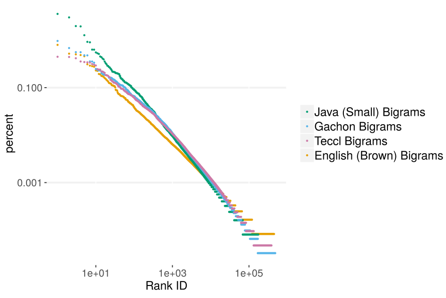
4.4.2 Comparing Technical and Non-Technical Corpora to Code

Now, we compare technical and imperative English (such as law, recipes, or high-level requirements) with non-technical English such as novels and plays. We expect our technical and imperative English to be more code-like due to the greater need for precision, and consequently the potentially higher cost of a miscommunication. Fig. 13 displays the unigram, bigram, and trigram Zipf curves for all of these corpora, the Brown corpus balanced from diverse English sources, and our smaller sample of the Java corpus. Interestingly, in the unigram plots, while Brown exhibits the expected curvature, while the specialized English corpora do not curve to the same degree. As the ngram length increases the slopes of the Java corpus, the technical corpora, and the non-technical corpora separate. The science fiction novels and Shakespeare's plays behave very similarly to the balanced Brown corpus. The technical corpora fall between these nontechnical English corpora and the Java code corpus, as expected from our hypothesis.

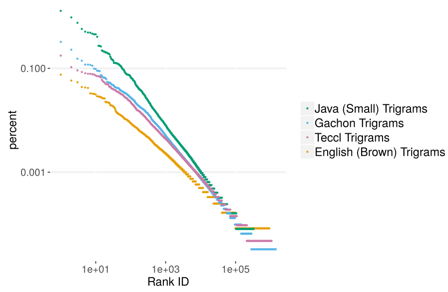
We note that of the technical corpora, the commit messages written by developers have some divergent behavior in their most frequent trigrams. While we filtered this corpus for obviously automated patterns, we note that in several projects commit messages follow a strong template. This leads to a couple extremely frequent patterns (*pull request #* and *Merge pull request* being the two outlying trigram patterns). However, once outside these few most frequent ngrams, the repetition in the corpus drops off much more sharply than in code. The technical and imperative corpora of NASA directives, recipes, and US Code corpora exhibit more code-like behavior than the Shakespeare, Science Fiction, and Brown corpora.



(a) Unigrams



(b) Bigrams



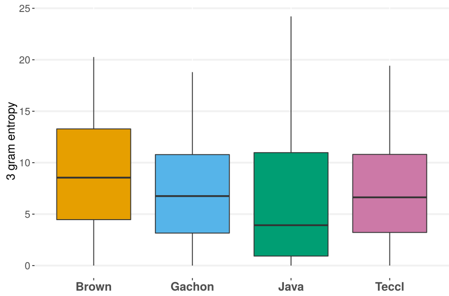
(c) Trigrams

Fig. 11 Zipf plots for the unigrams, bigrams, and trigrams of the general English, Java, and English language learner corpora

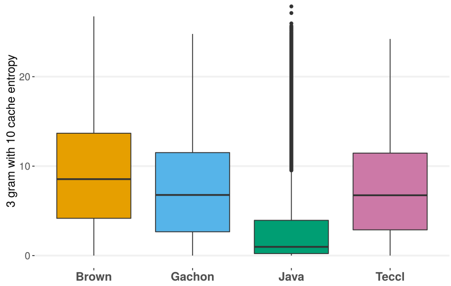
In Fig. 14, we verify these results with the ease of prediction via language model. With the exception of commit messages [23], the technical corpora are easier to predict than the non-technical corpora, but not as easy as the Java corpus, regardless of which language model is used. We validate these distributions with Wilcox tests and effect sizes, shown in Table 10, which compare the effect size between brown and our other corpora, and Java and our other corpora. We see that all corpora are more predictable than Brown, but that the commit messages and the non-technical corpora are proportionately much closer to the balanced Brown corpus than the other technical and imperative corpora.

To understand why the commit message entropy might behave more closely to non-technical English, we note that previous studies have pointed out the often poor quality of software documentation (Bachmann and Bernstein, 2009), and which has long been a concern in the software community(Zhi et al, 2015). That software documentation is often of poor quality conflicts with previously theorized need for more repetition in technical corpora - that when greater precision is needed and the cost of miscommunication is higher, more repetitive language is used. However, if developers do not consistently treat commit messages as an environment for precise communication, this may explain why its behavior diverges from the other technical corpora.

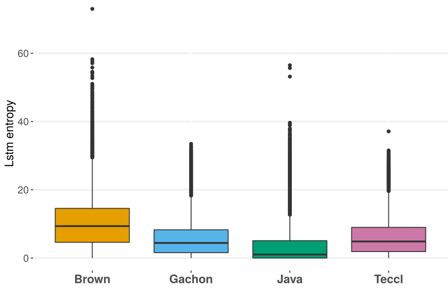
[23] We note that an independent study on commit message entropy and build failure found similar ranges (Santos and Hindle, 2016)



(a) 3 grams



(b) 3 grams with cache

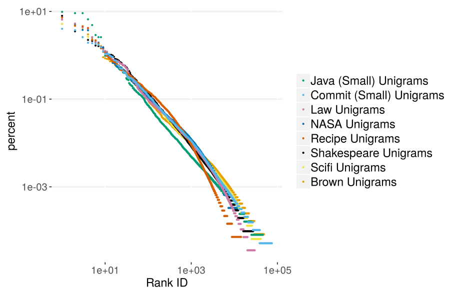


(c) LSTM (Small)

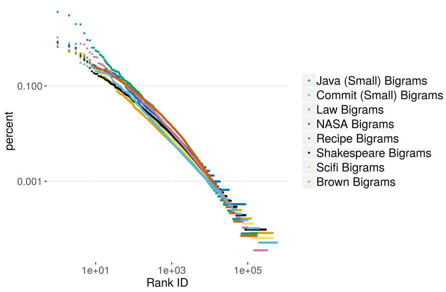
Fig. 12 Entropy comparisons of the of the English language learners corpora with Java and English Corpora using the LSTM and best trigram models

Likewise, the entropy of Java is significantly smaller than all corpora, but this effect size of this difference is sometimes small between it and the technical english corpora. In fact, with the best language models, the size of the difference between the median entropy of Java and both the corpus of US law and the corpus of recipes is only slightly over 1 bit. In terms of confidence intervals, when using a cache or LSTM model, Java is about as twice as predictable as the these corpora.

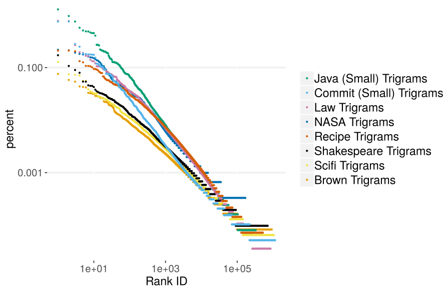
We also checked to see if there was any effect of a cache for the technical and non-technical corpora. If technical language behaves like code, we would expect more local repetition, and hence improvements when moving from an ngram to an ngram-cache model. Table 11, demonstrates confidence intervals and effect sizes for the cache improvements, with positive confidence intervals indicating an improvement over a basic ngram model. For our non-technical corpora, there is a extremely small negative effect on predictability when using a cache, and no significant effect on the Brown corpus. In comparison, the small Java corpus, commit messages, the legal language corpus, and the NASA directive corpus all have significant increases in entropy when not including the cache, though there is no cache effect in the recipe corpus. The effect size is extremely tiny in the NASA corpus, but somewhat larger for the US code. This agrees with the notion of the restrictiveness of technical language, and especially that of legal language as the most restrictive technical language, as its local repetitiveness allows a cache to improve about twice as much over the raw ngram score. However, the cache effect in the legal corpus is still not as large as with



(a) Unigrams



(b) Bigrams



(c) Trigrams

Fig. 13 Unigram, bigram, and trigram Zipf plot comparisons between the technical and imperative English corpora in comparison to the non technical English corpora and Java

the Java corpus. In commits, the cache effect may be strengthened by temporal locality - commits close in time could involve similar changes.

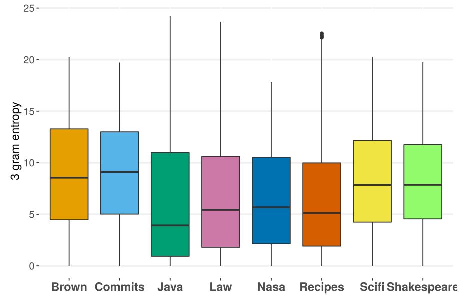
So the locality effects in specialized technical and imperative corpora are mixed. The technical corpora appear more code-like, but the imperatively styled recipe corpus does not. However, a more focused study would be needed to better establish the role of locality in technical language. Nevertheless, overall, we can answer RQ 4 mostly positively. Other than the commit message corpus which exhibits unique behavior in the Zipf plots and entropy distributions, the technical and imperative corpora resemble code more closely than other domain limited non-technical corpora.

5 Discussion

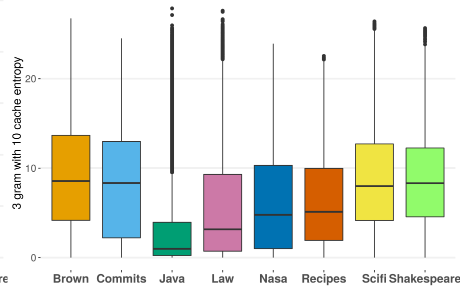
5.1 Practical Impacts

We note that Naturalness *per se* has tremendous practical impact; and has proven to be highly actionable. Hundreds of papers from dozens of different authors have explored applications ranging from code suggestion, to defect finding, to software porting, to automatic repair synthesis[24]. The wealth of applications of this phenomenon begs the question: *Why is code*

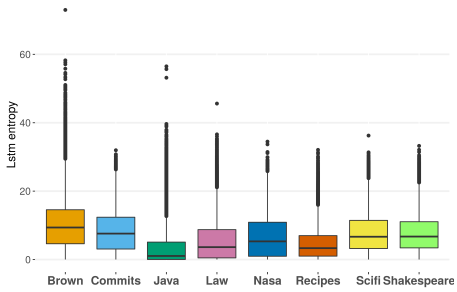
[24] Allamanis et al (2017) have extensively surveyed such applications.



(a) 3 grams



(b) 3 grams with cache



(c) LSTM (Small)

Fig. 14 Box plots of the distribution of entropy of the technical and imperative English corpora in comparison to the non technical English corpora and Java

natural? This question, to our knowledge, has not been explored before this work. Although scientific investigation of phenomena often precedes practical application, there are numerous examples where it works in reverse. Lithium *e.g.,* has been used to treat depression for generations; however, the exact bio-chemical mechanisms are only recently becoming clear. This purely scientific advance opens a pathway to "the development of safer, cheaper, or more effective pharmacotherapeutics"[25].

Likewise, while our study is primarily scientific in nature, and succeeded the practical impacts; still even this post-hoc scientific investigation has some practical consequences. We note that in code completion, we can expect that developers find completions on *open* category tokens most helpful; closed category tokens are shorter, and easier to remember. Indeed, most current completion tools (*e.g.,* in Eclipse) are designed for method/member completions - classes of open category tokens. However, these tokens are also the hardest to predict for language models. As we shall see below, our investigations suggest a way to improve the performance for in predicting open category tokens.

Recall that our results indicate that open category tokens in code are still more repetitive than in natural language (See Fig. 8 and Fig. 9). While the repetition persists in the open category tokens, it is clear that the *precise n-gram patterns* would be different. Thus, it is quite possible that the open category n-grams would suggest *different* completions than

[25] See Tobe *et al*, PNAS, 144(22), May 2017

Table 10 Summary of non-parametric effect sizes and 99% confidence intervals (in bits) comparing the tokens for each technical and non-technical corpus with Brown and then Java. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a Mann Whitney U test

Brown >Language	Ngram	Cache	LSTM
NASA	(-2.514, -2.39) 0.197***	(-2.96, -2.826) 0.224***	(-3.374, -3.207) 0.216***
Science Fiction	(-0.514, -0.421) 0.045***	(-0.396, -0.295) 0.031***	(-2.065, -1.949) 0.158***
US Code	(-2.532, -2.456) 0.219***	(-3.736, -3.655) 0.323***	(-4.55, -4.457) 0.343***
Shakespeare	(-0.592, -0.498) 0.053***	(-0.391, -0.287) 0.031***	(-2.157, -2.038) 0.166***
Recipes	(-2.763, -2.683) 0.268***	(-2.737, -2.651) 0.254***	(-5.127, -5.028) 0.426***
Commits	(-0.169, -0.039) 0.008***	(-0.918, -0.789) 0.068***	(-1.841, -1.697) 0.13***
Language >Java (Small)	Ngram	Cache	LSTM
NASA	(-0.636, -0.529) 0.06***	(-2.237, -2.101) 0.253***	(-2.478, -2.332) 0.249***
Science Fiction	(-2.754, -2.67) 0.247***	(-5.35, -5.257) 0.523***	(-4.121, -4.037) 0.457***
US Code	(-0.532, -0.468) 0.061***	(-0.9, -0.868) 0.21***	(-1.152, -1.09) 0.204***
Shakespeare	(-2.801, -2.711) 0.24***	(-5.651, -5.56) 0.519***	(-4.202, -4.117) 0.446***
Recipes	(-0.493, -0.431) 0.063***	(-2.676, -2.603) 0.388***	(-1.135, -1.085) 0.255***
Commits	(-3.355, -3.203) 0.226***	(-5.249, -5.091) 0.377***	(-4.635, -4.508) 0.374***

language models that incorporate all the tokens. Therefore, as an example of applying this paper's theory to a real world application we ask

RQ5. *Does using a language model with only open category tokens improve on the state-of-the-art for the code suggestion task for these most relevant tokens?*

For this sample application experiment, we adapt the framework of Hellendoorn and Devanbu, *SLP-Core*(Hellendoorn and Devanbu, 2017)[26], which is to our knowledge the current best published performer in the Java code completion task. This model extends the basic cache model, using the inherently *hierarchical* namespace scope of code—with several nested caches to get fast and accurate code completion results over nested scopes.

In this experiment we duplicate our Java code corpus, and lex all the files in place in each copy - retaining the directory structure information so the *SLP-core* model can leverage the power of its nested model. In one copy, we retain all tokens; in the other copy, we retain just the open category tokens. We train the model on each corpus, and then using a leave one out approach we iterate over each file in turn - removing it from the training corpus, then using the rest of the files to create a suggestion list for the tokens in this "test" file. For each token and suggestion list, we calculate the *Mean Reciprocal Rank (MRR)*. We then take the two lists of *just the open category word MRR* predictions and average them across each test file.

In Fig. 15 we see the results of this experiment. The box plots show a visible improvement in the average file *MRR* over the open category tokens when using the model that excludes the closed category words as context. To quantify these effects we use paired t-test

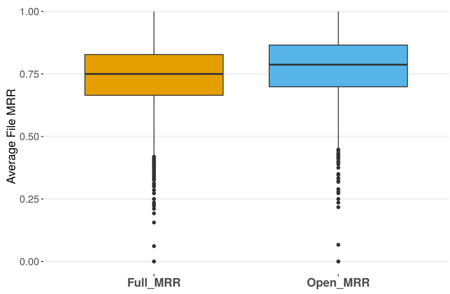
[26] This framework can be found at `https://github.com/SLP-team/SLP-Core`.

Table 11 Summary of non-parametric effect sizes and 99% confidence intervals (in bits) comparing the locality effects of the cache in Java and the various English corpora. Positive values in the intervals indicate an improvement due to the cache, and negative values indicate worse performance compared to the pure ngram model. Numbers are marked with * if $p < .05$, ** if $p < .01$, *** if $p < .001$ from a paired Mann Whitney U test

Language	Ngram > Cache
Brown	(-0.021, 0.049) 0.001
Java	(1.570, 1.631) 0.269^{***}
NASA	(0.375, 0.523) 0.0566^{***}
Recipes	(-0.030, 0.030) 0
Commits	(0.676, 0.829) $.069^{***}$
Science Fiction	(-0.129, -0.028) 0.008^{***}
US Code	(1.123, 1.180) 0.138^{***}
Shakespeare	(-0.255, -0.155) 0.021^{***}

and Cohen's-d effect size (as the distributions of *MRR* are relatively normal), and see that this effect is statistically significant with *medium* effect size (Cohen's-d for paired samples = 0.611). These results indicate a definite improvement, and are actionable for further improvements in code suggestion. Further improvements may be possible through judicious blends of full and open category-only cache models, and is left for future work.

Although we do not pursue them here, there are other possible applications. Our results suggest that the low entropy of source code is contingent, *viz.,* a matter of choice, rather than a syntactic necessity; and furthermore, this low-entropy preference recapitulates similar low-entropy preference exhibited in domains where reading/writing are effortful. This suggests a couple of applications.



(a) Boxplot

Fig. 15 Boxplots comparing the average file *MRR* of the open category code completion model against the full standard model

First, if lower entropy is a matter of preference, then it is possible that tools that lower the entropy of code (without changing the meaning) would be useful to developers. This could be accomplished by applying sequences of meaning-preserving transforms, observing the changes in entropy, and seeking out entropy low-points. Existing work (Liu et al, 2017) has used similar approaches to *increase entropy*, as an effective way to confound code deobfuscators. Second, higher-entropy regions of code might potentially indicate regions that could be restructured, using meaning-preserving transforms, for easier readability; these regions could also be fruitfully suggested to code reviewers as areas worthy of critical attention. This type of entropy-based restructuring or editing is also motivated by prior work in psycholinguistics, that suggests that text with lower entropy is easier to understand (Levy, 2008; Demberg and Keller, 2008; Frank, 2013). Moreover, given that entropy differences appear to capture more than just syntax in code, it would be interesting to see how entropy measures from different and improved language models of code correlate with recent work on confusing code (Gopstein et al, 2017, 2018), and measures of code understandability, where recent work has shown existing metrics may not correlate strongly with human judgement (Scalabrino et al, 2017; Trockman et al, 2018).

5.2 Limitations & Future Work

Given the nature of the cognitive questions this paper seeks to answer from a corpus-based perspective there are several threats we have attempted to minimize. As discussed throughout the paper, natural language differs in domain usage, grammatical constraints, vocabulary creation, in many other ways from programming language. More specifically, for our open and closed category word experiment, these groups are more clearly defined in natural language than source code. We mitigated this threat as much as possible by developing a shared classification across several different languages, and changes along the boundary of this classification resulted in little difference in the results. Likewise, syntax trees for code and English serve different roles and represent different granularities (a file as a unit vs a sentence); nevertheless, we have applied what controls we can to ensure the trees are both concrete, as accurate as possible, and modeled equivalently. Corpora with different constraints and data availability are challenging to model equivalently, but we hope that random sampling and the use of multiple language modeling techniques increases confidence in the validly of our results.

Our studies have focused most heavily on Java and English. While there are good reasons to believe that our findings generalize to other languages, further data analysis would shed a definitive light. In particular, there are indications in our data that Haskell corpora are somewhat different from other languages. The reasons remain unclear; it may be a factor of the language itself–Haskell is a functional higher-order, polymorphic, lazy language, unlike the others we have studied—or it may be a cultural effect. Haskell programmers tend to be very highly trained computer scientists and (in our experience) passionately committed to the power and elegance of lazy, higher-order, polymorphic functional programming. A comparative study of programming language features, and their effects on repetition in software corpora, remains a worthy subject of future study.

Our comparative corpora studies demonstrate English's similarity to code in situations where the language is more challenging to the writer or uses more imperative and technical language. However, given that we have established that some of the additional repetition in code *is* the result of human choice, more work is still necessary to explore the ways *how* and reasons *why* humans choose to write code more repetitively. Below, we highlight a few confounds that can influence these choices, along with avenues to pursue with greater focus on particular factors.

First, our studies suggest that the greater repetition in code may arise from the effort required to read and write code. Some of the tactics used by programmers may certainly contribute this. For example, programmers adopt coding standards to make code easier to read. Programmers often cut and paste code. They prefer familiar ways of coding (*e.g.,* code idioms (Allamanis and Sutton, 2014)). Furthermore, Social Q&A sites like Stack Overflow have become influential stores of coding knowledge, from whence a lot of coding patterns and idioms are disseminated. All these practices certainly contribute to repetitiveness, and the relative degree of influence of these various practices, and the effects on quality and productivity, remains a subject of future study.

Secondly, while our study focuses on *static* corpora, some elements of human choice may become more apparent in corpora that change over time. In natural language, though studies on how natural language changes over time exist (e.g. Petersen et al, 2012), records of documents changed in a manner similar to software versioning are sparse.

However, one of the most compelling aspects of linguistic studies of software is the availability of change histories, which afford the opportunity to conduct time-series studies of software *linguistic evolution*. Software content changes in response to various pressures, including customer demand, changes in platforms and APIs, and social and organizational pressures such as coding standards and code reviews. Our studies were conducted on fairly mature projects, at a fairly advanced stage in the life-cycles. The effects of API change (Dig and Johnson, 2005; Kim et al, 2011) and requirements on software evolution (Harker et al, 1993; Lehman, 1996) have been studied in the past; however, the relationship between software evolution and the linguistic changes arising from time and social factors remain largely unexplored. Some of the questions we would like to explore further include whether each individual's coding patterns change with time, and whether a team's coding patterns start out with greater diversity, and converge over time. Communal discussion patterns and convergence have been studied in open source projects, but only in the natural language of the issue discussion on these projects (Kavaler et al, 2017).

Moreover, social aspects of linguistic changes have been explored in the study of natural language evolution and change (e.g. Bright, 2017; Michael, 2014), and it would be interesting to explore such issues with code. For example, while the way an individual generates source code is certainly shaped by cognitive constraints, there are other factors to consider, such as education, past experience, interaction with other developers, and membership in specific communities. Experience in software development in particular is a complex and nuanced issue, involving not only experience in the programming language, but also in the domain and project to fulfill community expectations. Interestingly, these same factors shape an individual's use of natural language; these are nontrivial issues that are central to the field of sociolinguistics. Just as natural language is shaped at least by both cognitive and social factors, we expect that a sociolinguistic analysis of code would be an interesting direction for future work.

6 Conclusions

Our study starts with the discovery first reported in (Hindle et al, 2012), that software is highly repetitive and predictable. While this is surprising in itself, the real surprise is that it is *far* more predictable than natural language; indeed, using the perplexity measure, it is about 8 to 16 times more predictable. Why is this the case? Is it vocabulary? Syntax? Or something else? Does it depend on programming language? Natural language? The type of corpora? While this paper does not provide a definitive identification of the exact reasons for why code is *so much* more predictable than English, we describe a series of experiments that points *away* from necessary language-based constraints, and *more towards* deliberate *human* choice, as the casual factor.

In this paper we first show that the differences observed between English and Java generalize to other natural and programming languages. Programming language corpora in general are more repetitive than natural language corpora. Next we address the question of whether the greater repetitiveness of code arises mainly from the simpler syntax of code. To begin with, we remove the keywords operators and punctuation from code, and likewise the closed-category words and punctuation from English, and compare the repetitiveness of the remaining content vocabulary, and find that in fact code gets *more repetitive* when these syntactic markers are eliminated; thus suggesting that the additional repetitiveness is not exclusively syntax-based.

Diving deeper and examining the parse tree structure, we do find that some of the differences in predictability derive from differences between programming language and natural language syntax. Once normalized for the number of expansions the grammar allows, writers of English and code choose among their immediate options equivalently in each language. However, when accounting for all the available terminal choices and the long term operations, code still remains more predictable than English. Thus, it seems while a significant portion of the difference between English and Java is determined by grammatical restrictions, these restrictions do not account for all of the difference.

We surmise that the residual differences between Code and English may arise from the greater difficulty of reading and writing code or from a potentially higher cost of miscommunication. Therefore, we compare code with several specialized English corpora that might require greater effort: ESL (english as a second language) corpora, legal corpora, NASA directives, cooking recipes, and developer communication of changes via commit messages. We find most of these corpora are still less predictable than code, but also exhibit more code-like behavior than English, thus constituting an intermediate level of predictability, as expected. Code is a unique form of human expression; as Allamanis et al (2017) observe, it comprises two channels; one from human to human and the other from human to code. This dual-channel nature places special demands on readers and writers. The behavior of these specialized technical and imperative English texts, where the style imposes greater effort and the cost of miscommunication is higher, is consistent with the theory that humans use repetitive but familiar structures to communicate more clearly when under such constraints. This is not conclusive proof nor does it eliminate all other reasons why humans may write code more repetitively, but the lack of such behavior in non-technical domain constrained English is suggestive.

Finally, the experiments in this paper can provide theoretical grounding to choices when designing new languages and finding the right degree of expressiveness. After all, if humans mostly choose from a limited set of possible available constructs in code, then these choices should impact how languages are created, documented, and taught. Highlighting or including language options that are never used may be increase confusion and the potential for mistakes. Likewise, this theory supports the notion that the limitations imposed by style are important for clear communication. For example, existing research shows that pull requests that conform to project style are more readily accepted (Hellendoorn et al, 2015). Our theory can be leveraged to improve code suggestions, as we demonstrated, and further, perhaps also for tools that perform code restructuring, or support code review.

As Knuth observed, code is not merely for the machines (Knuth, 1984); while written in an artificial language, it is meant to be read by humans, and thus, exhibits properties inherent to other, naturally-occurring varieties of human-human communication.

Acknowledgements We would like to thank Professors C. Sutton, Z. Su, V. Filkov, and R. Aranovich, along with the UC Davis DECAL and NLP Reading groups for comments and feedback on this research. We also would like to especially thank V. Hellendoorn for his feedback and input on our experiment between parse trees in Java and English. We also acknowledge support from NSF Grant #1414172, Exploiting the Naturalness of Software. Finally we are grateful to the reviewers and editors of this journal for their thoughtful comments, which were very helpful in improving the work presented in this paper.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al (2016) Tensorflow: A system for large-scale machine learning. In: OSDI, vol 16, pp 265–283

Allamanis M, Sutton C (2013) Mining source code repositories at massive scale using language modeling. In: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, pp 207–216, DOI 10.1109/MSR.2013.6624029

Allamanis M, Sutton C (2014) Mining idioms from source code. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2014, pp 472–483, DOI 10.1145/2635868.2635901, URL `http://doi.acm.org/10.1145/2635868.2635901`

Allamanis M, Barr ET, Devanbu P, Sutton C (2017) A survey of machine learning for big code and naturalness. arXiv preprint arXiv:170906182

Andor D, Alberti C, Weiss D, Severyn A, Presta A, Ganchev K, Petrov S, Collins M (2016) Globally normalized transition-based neural networks. arXiv preprint arXiv:160306042

Bachmann A, Bernstein A (2009) Software process data quality and characteristics: A historical view on open and closed source projects. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, ACM, New York, NY, USA, IWPSE-Evol '09, pp 119–128, DOI 10.1145/1595808.1595830, URL `http://doi.acm.org/10.1145/1595808.1595830`

Baxter G, Frean M, Noble J, Rickerby M, Smith H, Visser M, Melton H, Tempero E (2006) Understanding the shape of java software. In: ACM Sigplan Notices, ACM, vol 41, pp 397–412

Bird S (2006) Nltk: the natural language toolkit. In: Proceedings of the COLING/ACL on Interactive presentation sessions, Association for Computational Linguistics, pp 69–72

Bradley DC (1978) Computational distinctions of vocabulary type. PhD thesis, Massachusetts Institute of Technology

Bright W (2017) Social factors in language change. In: The Handbook of Sociolinguistics, Wiley-Blackwell, chap 5, pp 81–91, DOI 10.1002/9781405166256.ch5, URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/9781405166256.ch5`

Busjahn T, Bednarik R, Begel A, Crosby M, Paterson JH, Schulte C, Sharif B, Tamm S (2015) Eye movements in code reading: Relaxing the linear order. In: Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on, IEEE, pp 255–265

Ferrer i Cancho R, Solé RV (2001) Two regimes in the frequency of words and the origins of complex lexicons: Zipf's law revisited? Journal of Quantitative Linguistics 8(3):165–173

Carlstrom B, Price N (2013) Gachon learner corpus

Chelba C, Mikolov T, Schuster M, Ge Q, Brants T, Koehn P (2013) One billion word benchmark for measuring progress in statistical language modeling. CoRR abs/1312.3005, URL http://arxiv.org/abs/1312.3005

Chen SF, Goodman J (1998) An empirical study of smoothing techniques for language modeling. In: Harvard Computer Science Group Technical Report TR-10-98

Chomsky N, Belletti A, Rizzi L (2002) An interview on minimalism. N Chomsky, On Nature and Language pp 92–161

Concas G, Marchesi M, Pinna S, Serra N (2007) Power-laws in a large object-oriented software system. IEEE Transactions on Software Engineering 33(10):687–708

Conklin K, Schmitt N (2008) Formulaic sequences: Are they processed more quickly than nonformulaic language by native and nonnative speakers? Applied Linguistics 29(1):72–89, DOI 10.1093/applin/amm022, URL +http://dx.doi.org/10.1093/applin/amm022, /oup/backfile/content_public/journal/applij/29/1/10.1093_applin_amm022/1/amm022.pdf

Danet B (1980) Language in the legal process. Law and Society Review pp 445–564

De Cock S (2000) Repetitive phrasal chunkiness and advanced efl speech and writing. Language and Computers 33:51–68

De Marneffe MC, Manning CD (2008) The stanford typed dependencies representation. In: Coling 2008: proceedings of the workshop on cross-framework and cross-domain parser evaluation, Association for Computational Linguistics, pp 1–8

De Marneffe MC, MacCartney B, Manning CD, et al (2006) Generating typed dependency parses from phrase structure parses. In: Proceedings of LREC, Genoa Italy, vol 6, pp 449–454

Demberg V, Keller F (2008) Data from eye-tracking corpora as evidence for theories of syntactic processing complexity. Cognition 109(2):193 – 210, DOI https://doi.org/10.1016/j.cognition.2008.07.008, URL http://www.sciencedirect.com/science/article/pii/S0010027708001741

Dig D, Johnson R (2005) The role of refactorings in api evolution. In: null, IEEE, pp 389–398

Field A (2009) Discovering statistics using SPSS. Sage publications

Frank S (2013) Uncertainty reduction as a measure of cognitive load in sentence comprehension. Topics in Cognitive Science 5(3):475–494

Gerlach M, Altmann EG (2013) Stochastic model for the vocabulary growth in natural languages. Physical Review X 3(2):021006

Ginter F, Hajič J, Luotolahti J, Straka M, Zeman D (2017) CoNLL 2017 shared task - automatically annotated raw texts and word embeddings. URL http://hdl.handle.net/11234/1-1989, LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University

Gopstein D, Iannacone J, Yan Y, Delong LA, Zhuang Y, Yeh MKC, Cappos J (2017) Understanding misunderstandings in source code. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 129–139

Gopstein D, Zhou HH, Frankl P, Cappos J (2018) Prevalence of confusing code in software projects: Atoms of confusion in the wild. In: Proceedings of the 15th International Conference on Mining Software Repositories, ACM, pp 281–291, DOI https://doi.org/10.1145/3196398.3196432

Gotti M (2011) Investigating specialized discourse. Peter Lang

Gousios G, Spinellis D (2012) GHTorrent: Github's data from a firehose. In: MSR, IEEE, pp 12–21

Hale J (2003) The information conveyed by words in sentences. Journal of Psycholinguistic Research 32(2):101–123

Harker SD, Eason KD, Dobson JE (1993) The change and evolution of requirements as a challenge to the practice of software engineering. In: Requirements Engineering, 1993., Proceedings of IEEE International Symposium on, IEEE, pp 266–272

Hathhorn C, Ellison C, Roşu G (2015) Defining the undefinedness of c. In: ACM SIGPLAN Notices, ACM, vol 50, pp 336–345

Hayes JH, Dekhtyar A, Sundaram SK (2005) Improving after-the-fact tracing and mapping: Supporting software quality predictions. IEEE software 22(6):30–37

Heafield K (2011) Kenlm: Faster and smaller language model queries. In: Proceedings of the Sixth Workshop on Statistical Machine Translation, Association for Computational Linguistics, pp 187–197

Hellendoorn VJ, Devanbu P (2017) Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE, pp 763–773

Hellendoorn VJ, Devanbu PT, Bacchelli A (2015) Will they like this?: Evaluating code contributions with language models. In: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR ’15, pp 157–167, URL `http://dl.acm.org/citation.cfm?id=2820518.2820539`

Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE ’12, pp 837–847, URL `http://dl.acm.org/citation.cfm?id=2337223.2337322`

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural computation 9(8):1735–1780

Hoffmann L (1984) Seven roads to lsp. Fachsprache 6:1–2

Hothorn T, Hornik K, van de Wiel MA, Zeileis A (2006) A lego system for conditional inference. The American Statistician 60(3):257–263

Jbara A, Feitelson DG (2017) How programmers read regular code: a controlled experiment using eye tracking. Empirical Software Engineering 22(3):1440–1477

Kavaler D, Sirovica S, Hellendoorn V, Aranovich R, Filkov V (2017) Perceived language complexity in github issue discussions and their effect on issue resolution. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, Piscataway, NJ, USA, ASE 2017, pp 72–83, URL `http://dl.acm.org/citation.cfm?id=3155562.3155576`

Khanh Dam H, Tran T, Pham T (2016) A deep language model for software code. arXiv preprint arXiv:160802715

Kim M, Cai D, Kim S (2011) An empirical investigation into the role of api-level refactorings during software evolution. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp 151–160

Kneser R, Ney H (1995) Improved backing-off for m-gram language modeling. In: Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on, IEEE, vol 1, pp 181–184

Knuth DE (1984) Literate programming. The Computer Journal 27(2):97–111

Kučera H, Francis WN (1967) Computational analysis of present-day American English. Dartmouth Publishing Group

Lehman MM (1980) Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68(9):1060–1076, DOI 10.1109/PROC.1980.11805

Lehman MM (1996) Laws of software evolution revisited. In: European Workshop on Software Process Technology, Springer, pp 108–124

Levy R (2008) Expectation-based syntactic comprehension. Cognition 106(3):1126 – 1177, DOI https://doi.org/10.1016/j.cognition.2007.05.006, URL http://www.sciencedirect.com/science/article/pii/S0010027707001436

Liu H, Sun C, Su Z, Jiang Y, Gu M, Sun J (2017) Stochastic optimization of program obfuscation. In: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, pp 221–231

Louridas P, Spinellis D, Vlachos V (2008) Power laws in software. ACM Transactions on Software Engineering and Methodology (TOSEM) 18(1):2

Mandelbrot B (1953) An informational theory of the statistical structure of language. Communication theory 84:486–502

Manning CD, Schütze H (1999) Foundations of statistical natural language processing. MIT press

Marcus MP, Marcinkiewicz MA, Santorini B (1993) Building a large annotated corpus of english: The penn treebank. Comput Linguist 19(2):313–330, URL http://dl.acm.org/citation.cfm?id=972470.972475

Michael L (2014) Social dimensions of language change. In: The Routledge Handbook of Historical Linguistics, Routledge, chap 22, DOI 10.4324/9781315794013.ch22, URL https://www.routledgehandbooks.com/doi/10.4324/9781315794013.ch22

Mikolov T, Karafiát M, Burget L, Cernockỳ J, Khudanpur S (2010) Recurrent neural network based language model. In: Interspeech, vol 2, p 3

Mitzenmacher M (2004) A brief history of generative models for power law and lognormal distributions. Internet mathematics 1(2):226–251

Norvig P (2009) Natural language corpus data. Beautiful Data pp 219–242

Paquot M, Granger S (2012) Formulaic language in learner corpora. Annual Review of Applied Linguistics 32:130–149, URL https://search.proquest.com/docview/1289774805?accountid=14505, copyright - Copyright Cambridge University Press 2012; Last updated - 2015-05-30

Petersen AM, Tenenbaum JN, Havlin S, Stanley HE, Perc M (2012) Languages cool as they expand: Allometric scaling and the decreasing need for new words. Scientific reports 2:943

Petrov S (2016) Announcing syntaxnet: The world's most accurate parser goes open source. Google Research Blog

Petrov S, Barrett L, Thibaux R, Klein D (2006) Learning accurate, compact, and interpretable tree annotation. In: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics, Association for Computational Linguistics, pp 433–440

Piantadosi ST (2014) Zipf's word frequency law in natural language: A critical review and future directions. Psychonomic bulletin & review 21(5):1112–1130

Piantadosi ST, Tily H, Gibson E (2012) The communicative function of ambiguity in language. Cognition 122(3):280–291

Pierret D, Poshyvanyk D (2009) An empirical exploration of regularities in open-source software lexicons. In: Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on, IEEE, pp 228–232

Salager F (1983) Compound Nominal Phrases in Scientific-Technical Literature: Proportion and Rationale. ERIC

Salvador A, Hynes N, Aytar Y, Marin J, Ofli F, Weber I, Torralba A (2017) Learning cross-modal embeddings for cooking recipes and food images. In: Proceedings of the IEEE

Conference on Computer Vision and Pattern Recognition

Santos EA, Hindle A (2016) Judging a commit by its cover: Correlating commit message entropy with build status on travis-ci. In: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR '16, pp 504–507, DOI 10.1145/2901739.2903493, URL `http://doi.acm.org/10.1145/2901739.2903493`

Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2017) Automatically assessing code understandability: how far are we? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, pp 417–427

Schmitt N, Carter R (2004) Formulaic sequences in action. Formulaic sequences: Acquisition, processing and use pp 1–22

Shannon CE (1948) A mathematical theory of communication, part i, part ii. Bell Syst Tech J 27:623–656

Shannon CE (1951) Prediction and entropy of printed english. Bell Labs Technical Journal 30(1):50–64

Siegmund J, Kästner C, Apel S, Parnin C, Bethmann A, Leich T, Saake G, Brechmann A (2014) Understanding understanding source code with functional magnetic resonance imaging. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 378–389

Stefik A, Ladner R (2017) The quorum programming language. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, ACM, pp 641–641

Stefik A, Siebert S (2013) An empirical investigation into programming language syntax. ACM Transactions on Computing Education (TOCE) 13(4):19

Sundaram SK, Hayes JH, Dekhtyar A (2005) Baselines in requirements tracing. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 30, pp 1–6

Sundermeyer M, Schlüter R, Ney H (2012) Lstm neural networks for language modeling. In: Thirteenth Annual Conference of the International Speech Communication Association

Trockman A, Cates K, Mozina M, Nguyen T, Kästner C, Vasilescu B (2018) "automatically assessing code understandability" reanalyzed: Combined metrics matter

Tsay J, Dabbish L, Herbsleb J (2014) Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th international conference on Software engineering, ACM, pp 356–366

Tu Z, Su Z, Devanbu P (2014) On the localness of software. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2014, pp 269–280, DOI 10.1145/2635868.2635875, URL `http://doi.acm.org/10.1145/2635868.2635875`

Varantola K (1986) Special language and general language: Linguistic and didactic aspects. Unesco Alsed-LSP Newsletter (1977-2000) 9(2)

Vinyals O, Kaiser Ł, Koo T, Petrov S, Sutskever I, Hinton G (2015) Grammar as a foreign language. In: Advances in Neural Information Processing Systems, pp 2773–2781

Wasow T, Perfors A, Beaver D (2005) The puzzle of ambiguity. Morphology and the web of grammar: Essays in memory of Steven G Lapointe pp 265–282

Weintrop D, Wilensky U (2015) Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In: Proceedings of the Eleventh Annual International Conference on International Computing Education Research, ACM, New York, NY, USA, ICER '15, pp 101–110, DOI 10.1145/2787622.2787721, URL `http://doi.acm.org/10.1145/2787622.2787721`

White M, Vendome C, Linares-Vásquez M, Poshyvanyk D (2015) Toward deep learning software repositories. In: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, pp 334–345

Wickham H (2009) ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, URL `http://ggplot2.org`

Xue X (2015) Ten thousand english compositions of chinese learners (the teccl corpus), version 1.1

Zhang H (2008) Exploring regularity in source code: Software science and zipf's law. In: Reverse Engineering, 2008. WCRE'08. 15th Working Conference on, IEEE, pp 101–110

Zhi J, Garousi-Yusifoğlu V, Sun B, Garousi G, Shahnewaz S, Ruhe G (2015) Cost, benefits and quality of software development documentation: A systematic mapping. Journal of Systems and Software 99:175–198

Zipf G (1949) Human behavior and the principle of least effort. Addison-Wesley, Cambody Mus Am Arch and Ethnol(Harvard Univ), Papers 19:1–125