Studying the Differences Between Natural and Programming Languages

Casey Casalnuovo
ccasal@ucdavis.edu
UC Davis Computer Science
2063 Kemper Hall, One Shields Avenue
Davis, CA 95616

Abstract

Programming languages are more repetitive and predictable than natural languages, and this difference could arise from either syntactic limitations or the semantic choices made by the writers of these texts. We present one experiment that provides evidence against the hypothesis that this difference is caused only by syntactic restrictions by removing closed vocabulary words and showing the remaining content words in Java are still more repetitive than those in English. We also summarize ongoing experiments aimed at further understanding the causes of the difference between natural and programming languages.

Introduction & Background

Source code is sometimes conceived as primarily being a product for machines to interpret and execute. This view however misses a critical point, that source code is not only a intermediate form of communication between human and machine, but also a form of communication between humans - a view advocated by Donald Knuth (Knuth 1984).

Few widely used software applications are developed by a single person, and applications with code that cannot be understood and maintained will be unlikely to succeed. Indeed, it has long been known that most development time is spent maintaining existing code rather than creating new code (Lehman 1980). Therefore, it is entirely reasonable to consider source code as much of a form of human communication as written text or speech, and like these it is also amenable to the same sorts of statistical language models used in natural language processing.

This observation about the potential adaptability of language models to source code was originally made by Hindle et al. (Hindle et al. 2012), who showed that not only were language models developed for natural language effective at capturing features of code, but in fact *more effective* than in their original context. While Hindle's work focused on using basic ngram language models to capture repetition in source code, this observation holds true for various cache ngram models (Tu, Su, and Devanbu 2014; Hellendoorn and Devanbu 2017), or even language models leveraging deep neural networks such as Long Short Term Memory Networks (LSTMs) (Hochreiter and Schmidhuber 1997). Figure 1 demonstrates this difference on a corpus of Java and English, using the standard evaluation method of

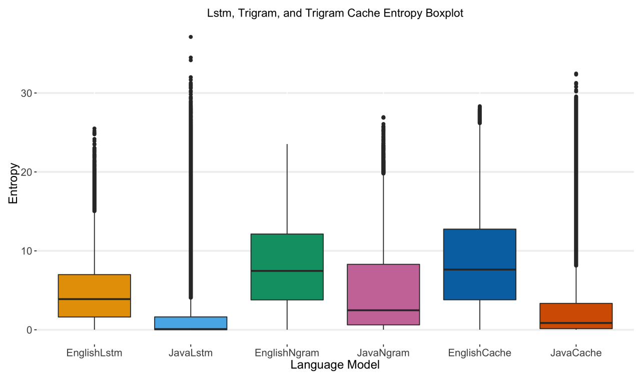

Figure 1: Entropy comparisons of English and Java corpora from 3 different language models.

entropy[1]. Lower entropy values indicate a token was less surprising for the language model. These boxplots display the entropy for each token in the test set, and show that regardless of the model chosen, Java tokens are more predictable than English tokens. For details on the datasets and language models, see the Data and Language Models sections respectively.

This repetition can also be visually represented without any need for a language model by 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 most frequent word receives rank 1 (or 0), then the next most frequent gets rank 2, and so on. These data are plotted with ranks on the x axis and the frequencies on the y axis. Zipf observed that these frequencies followed a roughly a power law distribution, an observation famously referred to as Zipf's Law. When plotted in log-scale, this relationship appears roughly linear.

We modify this plot in two ways. First, while Zipf put frequency on the y-axis, we normalize the values to a percentage to make corpora more comparable. Secondly, we extend the idea of a Zipf plot beyond merely individual words. Instead of just looking at the frequencies of individual words, we can also look at the frequencies of sequences of words. Figure 2 shows Zipf plots for unigrams and trigrams. The

[1]https://en.wikipedia.org/wiki/Cross_entropy

increased repetition of source code over English can be seen in the widening gap between the slopes of the two lines as we increase the size of the sequence. The most frequent trigrams in Java are more frequent relative to the rest of the corpus than is true in English. In comparison, the most frequent unigrams are behave much more similarly in both Java and English. While plots are not displayed here due to space limitations this trend of an increasing gap persists for sequences of 2, 3, 4 and 5, and this increased repetition over English exists in several other programming languages (C, Haskell, Clojure, Ruby).

While source code is more repetitive and thus more predictable, it is not clear what source of these differences are. Broadly, the difference could be explained from theories about the either *syntactic* or the *semantic* differences between natural and programming languages. Syntactically, additional repetition could arise from the more restricted grammar of source code. Source code grammars are unambiguous, for ease of parsing; this limitation might account for the observed repetition.

It's also possible that the repetitiveness of source code transcends syntactic limitations: perhaps source code has greater domain specificity; perhaps programming is so cognitively challenging for humans, that developers deliberately limit their constructions to a smaller set of highly reused forms.

The existence of design patterns (Johnson et al. 1995) and programming idioms[2] offer some evidence for this claim. They recommend answering well known engineering problems with commonly used and tested patterns. For the purposes of human to human communication, use of a documented design pattern can clearly indicate code structure between experts, and reduce cognitive effort. Moreover, the popularity of sites like Stack Overflow[3], shows that solutions to existing problems are often reused. In contrast, however, programming language adoption is often driven by the availability of libraries(Meyerovich and Rabkin 2013), which minimize source code repetition. Likewise, copy pasted code is often targeted for refactoring, again potentially reducing source code duplicates.

This leaves us with 2 questions: 1) how much does programming language *syntax* influence repetitiveness in coding? and 2) what are the *semantic* factors that influence repetitiveness? Here, we will present the results of an experiment to address the former in detail and summarize some preliminary results from other experiments trying to address both questions in our Discussion section. Our experiment answers the following question:

To what degree does removing closed vocabulary words more closely associated with the syntactic structure of language, and examining only the open vocabulary words associated with the content of a text explain the difference between Java and English?

As presented below, we find that ignoring the non-content words and comparing only the content words *does not* eliminate all of the differences between the two corpora.

[2]See `http://www.programming-idioms.org/`.

[3]https://stackoverflow.com/

Experiment

Languages evolve with time. As they evolve, certain word categories expand more easily than others. We can call categories of words where new words are easily and frequently added *open category* words (*e.g.*, nouns, verbs, adjectives). The vocabulary in open categories can grown without limit; as more text is added to the corpus, we would expect to continuously see new open-category words. Other categories are *closed* - no matter how much text is incorporated into the corpus, the set of distinct words in these categories is fixed and limited.[4]

So what are the closed vocabulary words in English and Java? In English, closed category words are often called *stopwords*, and include conjunctions, articles, and pronouns. In Java, elements such as reserved words, like *for*, *if*, or *public* form a small list of language specific keywords that cannot be used outside well defined contexts. The arithmetic and logical operators, which combine elements in code, resemble conjunctions in English, and also constitute closed vocabulary. Additionally, closely related is the notion of *punctuation*, which appears in both natural and programming languages, and respectively divides them into clauses and sentences or expressions and statements.

Closed vocabulary tokens relate to the underlying structure and grammar of a language, whereas open vocabulary tokens relate more to the content. A fixed number of (closed category) markers defining how to structure the content of a message in a language are 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.

Suppose we remove the closed vocabulary words from a corpus, and leave behind just sequences of open vocabulary words. Gone are the elements most closely tied to the language syntax; arguably, what remains are *content words*, that most closely model the sequence of ideas *expressed* by the text. Below is are examples of what part of these filtered sequences would look like in Java and English respectively:

... String lines data split response setContentType response setCharacterEncoding int batchCount String s lines s s trim ...

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

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.

Language Models

For greater robustness, we compare corpora using three different language models, two which are variants on the simpler ngram model, and one LSTM model.

For our two ngram models, we use a simple trigram model, and a trigram model with a 10-token cache as de-

[4]While this category could very rarely be updated, this would encompass unusual and significant changes in the language - for instance a new preposition or conjunction in English.

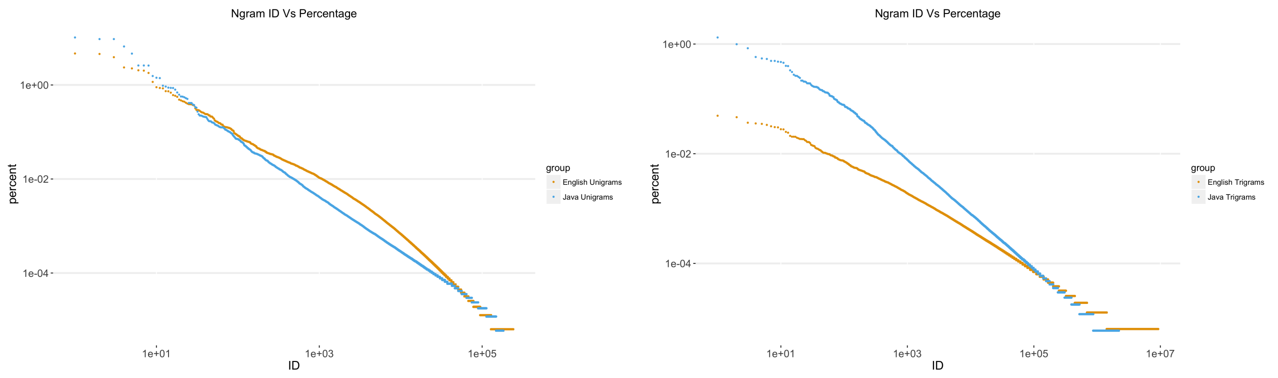

Figure 2: Comparison of slopes for Zipf unigram and Zipf trigram plots.

scribed in Tu et al. (Tu, Su, and Devanbu 2014). The underlying ngram model for both is implemented by KenLM (Heafield 2011).

Our LSTM models are implemented in Tensorflow (Abadi et al. 2016), with a minibatch size of 20, 1 hidden layer of 300 units, 13 training epochs, and a learning rate of 1.0. For the LSTM we divide each corpus so it is split at file level with 70% of files in the training set, and 15% each in the validation and test sets. The ngram models do not use a validation set, so we combine the validation and training sets when training them. Otherwise, the test sets for all models are equivalent for each corpus.

Data

To perform this experiment, we remove the closed category or stopwords from our Java and English corpora. For English, we use a list of 196 words and contraction stems, along with a list of 30 punctuation markers, derived from a published NLTK stopword list (Bird 2006). In Java, we use the Pygments syntax highlighting library[5] to remove non-identifier words, keeping only references to types, classes, variables, and function names.

Table 1 shows the size of our two corpora after tokenization before and after stopword removal. The Java corpus consists of a diverse set of 12 projects including popular GitHub projects and part of the Eclipse project, whereas the English corpus comes from a random sampling of a 1 billion token benchmark corpus (Chelba et al. 2013). We see that removing these closed category words removes a greater percentage of the Java corpus (only 33% left for Java vs 53% left for English). Existing work by Allamanis et al has shown closed category tokens in code to be much more predictable than identifiers(Allamanis and Sutton 2013), so if this fact holds in English too, that fact that Java has proportionally more could explain why it is more repetitive. However, as we will show shortly, this difference in amount of stopwords is insufficient to explain the differences seen between the two texts.

[5]http://pygments.org/

Table 1: Summary of the corpora sizes for English and Java.

	Java	English
All Tokens	16852300	15708917
Open Category	5577507	8340284

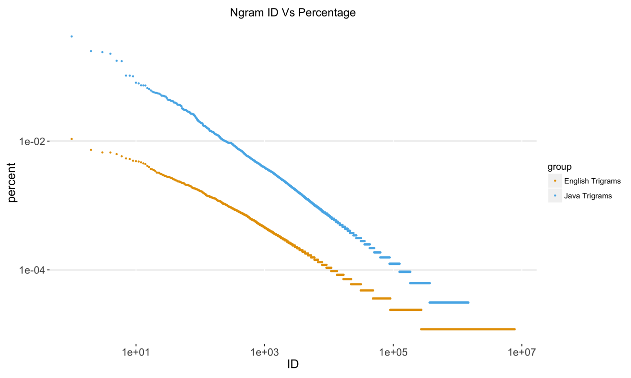

Figure 3: Trigram Zipf plots show slopes for English and Java open category words that indicate greater repetition in Java.

Results

In Figure 3 we see the slopes of these content-word trigram sequences sustain a significant divergence between each other, indicating that Java content-word sequences are more repetitive than the English content-words. Therefore, our intuition should be that the language models should also have an easier time predicting the Java sequences. Figure 4 confirms this intuition. Regardless of which language model is used, the content words of Java remain easier to predict than the content words of English. Additionally, note that when compared to the average entropy of the entire corpora seen in Figure 1, the open category entropy values are higher, as expected from existing research (Allamanis and Sutton 2013). So, while content words are in general less predictable, Java content words are easier to predict than English content words.

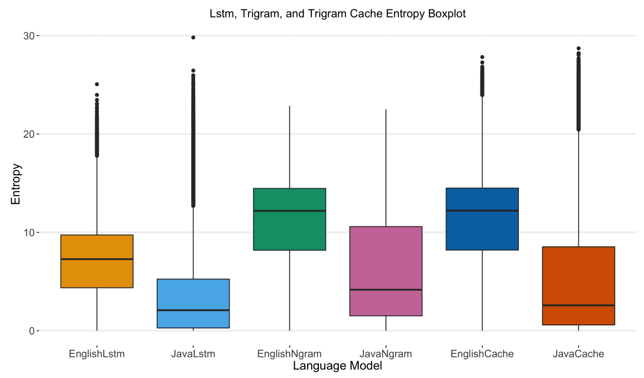

Figure 4: Entropy comparisons of English and Java corpora with closed category word excluded from three different language models.

Discussion and Ongoing Work

Testing theories about the differences between natural and programming languages while controlling for all possible confounding factors is difficult. Comparing the effects of the syntactic or the semantic differences in isolation is much easier than modeling them together. This paper presented an experiment on the syntactic elements of Java and English, which provides some evidence that the content words of source code are more repetitive than those in English, and that previously observed effects are not entirely due to language structure, but more work is needed to conclusively establish this.

We have tried several other experiments to answer *why* programming and natural languages differ, and summarize the preliminary results here. Further considering the problem from a syntactic perspective, we have built comparable parse trees for Java and English to evalauate how much of the difference is due to the greater ambiguity in natural language grammar. Though the effect sizes differ, the results from this experiment also suggest that the differences between code and English cannot be explained just by syntax.

Therefore, it is likely that the *content* presented in source code is indeed more repetitive than English content. As mentioned previously this could arise from a variety of causes, including increased difficulty causing developers to artificially restrict how they code to more limited but reliable choices, or perhaps software projects being more domain specific and limited in scope. We gathered several corpora of varying domain specificity, style, and language proficiency in English to test these hypotheses. While these studies have not yet provided any definitive evidence, our results comparing generic English with the writing of novice English language learners has shown novice English learners write more repetitively and predictably than what is observed from native English writers. This behavior is consistent with the hypothesis that when language is more cognitively difficult, humans rely on a smaller set of restricted forms to compensate. Also, when comparing technical domain specific corpora with non-technical domain specific corpora, we found only the technical corpora were more repetitive.

While these experiments form a starting point, we hope to continue to research in this vein and invite others to contribute so that we can achieve a better understanding of how humans process and produce natural and programming languages.

$$f(x_j) = \frac{e^{\frac{-||x_i - x_j||^2}{2\sigma_i^2}}}{\sigma_i\sqrt{2\pi}}$$

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

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

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

Chelba, C.; Mikolov, T.; Schuster, M.; Ge, Q.; Brants, T.; and Koehn, P. 2013. One billion word benchmark for measuring progress in statistical language modeling. *CoRR* abs/1312.3005.

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

Hellendoorn, V. J., and 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*, 763–773.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, 837–847. Piscataway, NJ, USA: IEEE Press.

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

Johnson, R.; Gamma, E.; Helm, R.; and Vlissides, J. 1995. Design patterns: Elements of reusable object-oriented software. *Boston, Massachusetts: Addison-Wesley*.

Knuth, D. E. 1984. Literate programming. *The Computer Journal* 27(2):97–111.

Lehman, M. M. 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* 68(9):1060–1076.

Meyerovich, L. A., and Rabkin, A. S. 2013. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices* 48(10):1–18.

Tu, Z.; Su, Z.; and Devanbu, P. 2014. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 269–280. New York, NY, USA: ACM.

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.