MINING SOURCE CODE REPOSITORIES AT MASSIVE SCALE USING LANGUAGE MODELING

COMP 5900 X

ERIC TORUNSKI

DEC 1, 2016

OVERVIEW

- Miltiadis Allamanis, ***Mining Source Code Repositories at Massive Scale using Language Modeling***, Mining Software Repositories (MSR), 2013
- Variable and function names in source code have meaning. Looking for patterns in names should give clues about the code.
- Language Models (LM) are probability distributions over strings. Training the LMs with more data should give better results.
- Looking at how common the names are should give a clue as to how reusable the code is. Also, it should give a clue as to which domain the code belongs to (financial, medicine, science, etc)

SOURCE CODE

- Source code is not only used to give instructions to a processor, but names in the code convey meaning to other programmers.
- The goal is to train a LM from a body of source code to assign high probability to strings that a human user is likely to write, and low probability to strings that are awkward or unnatural (autocomplete).
- Code that is more natural should be more understandable to other programmers.

METRICS USED

- N-gram log probability (NGLP). In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech.
- Identifier Information Metric (IIM). This has direct applications to code reuse because code that is less domain specific is more likely to be reused.

N-GRAM PROBABILITY

- The probability of a token t_m depends on the previous n -1 tokens
- The problem to avoid is that the probability of a token will be 0 if it is not in the training set. Source code normally introduces new words so there should be some minimal default probability.
- From an information theory perspective, a series of tokens has a measure of surprise, or predictability :

$$Q(t_0, t_1, \ldots t_m) = -\log_2(P(t_0, t_1, \ldots t_m))$$

- This is called the n-gram log probability

CODE ANALYSIS

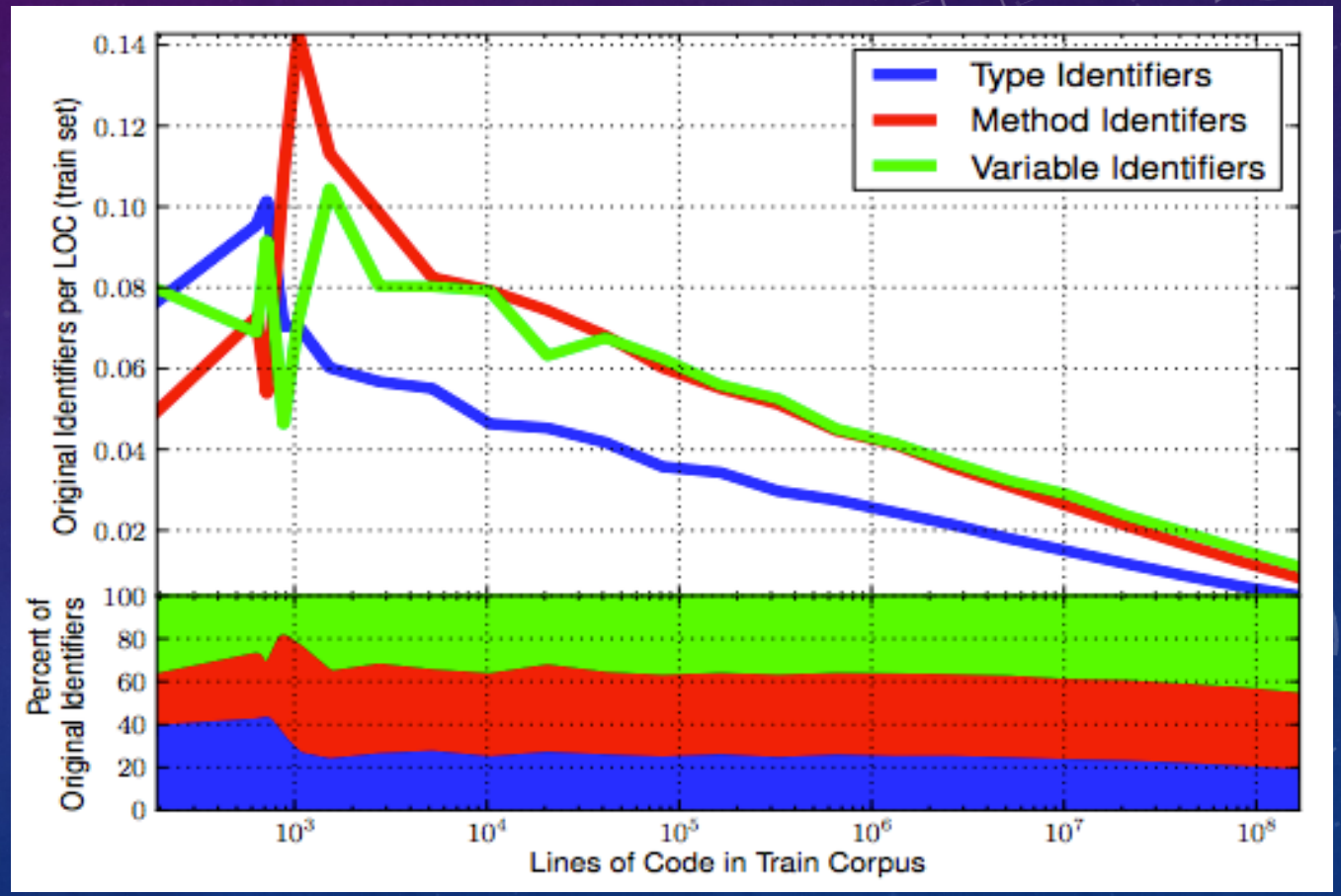
- The authors wanted to use only “high-quality” code projects. They used GitHub’s social fork system to see how many times a project was forked, meaning that people want to use it. The assumption is that low quality projects are rarely forked.
- They also removed duplicate code (imported source code libraries)
- Ended up with 14807 projects, with 352,312,696 lines of code in 2,130,264 files. They used Eclipse JDT to get AST of code to remove whitespace and comments. This gave them information about whether names were variables, types, or methods.

CODE ANALYSIS - LIMITATIONS

- Since this is mining GitHub, the source code will be skewed towards open source projects. It is unlikely to find source code on banking, defense, cybersecurity projects, etc.

IDENTIFIER INTRODUCTION RATE

- Measuring new names: initially, Type identifiers (String, Vector, HashMap) are found at the highest rate (instances / kloc), with method and variable names tied for second.
- After learning more code, the orders are reversed with type identifiers being least likely

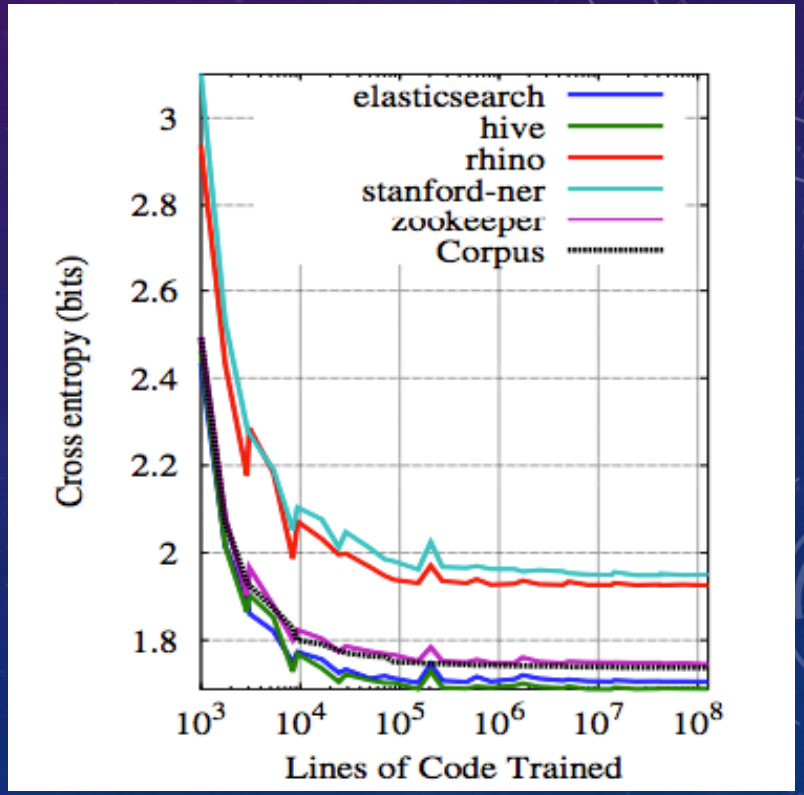


TESTING OF GIGA-SCALE LANGUAGE MODELS

- Using a large body of training data reduces the cross-entropy of tokens to 4.9 bits, as opposed to 6 bits as originally reported by Hindle et al. This means that using large data sets makes identifiers more predictable in unknown source code. The algorithm is said to be “Data-hungry”.

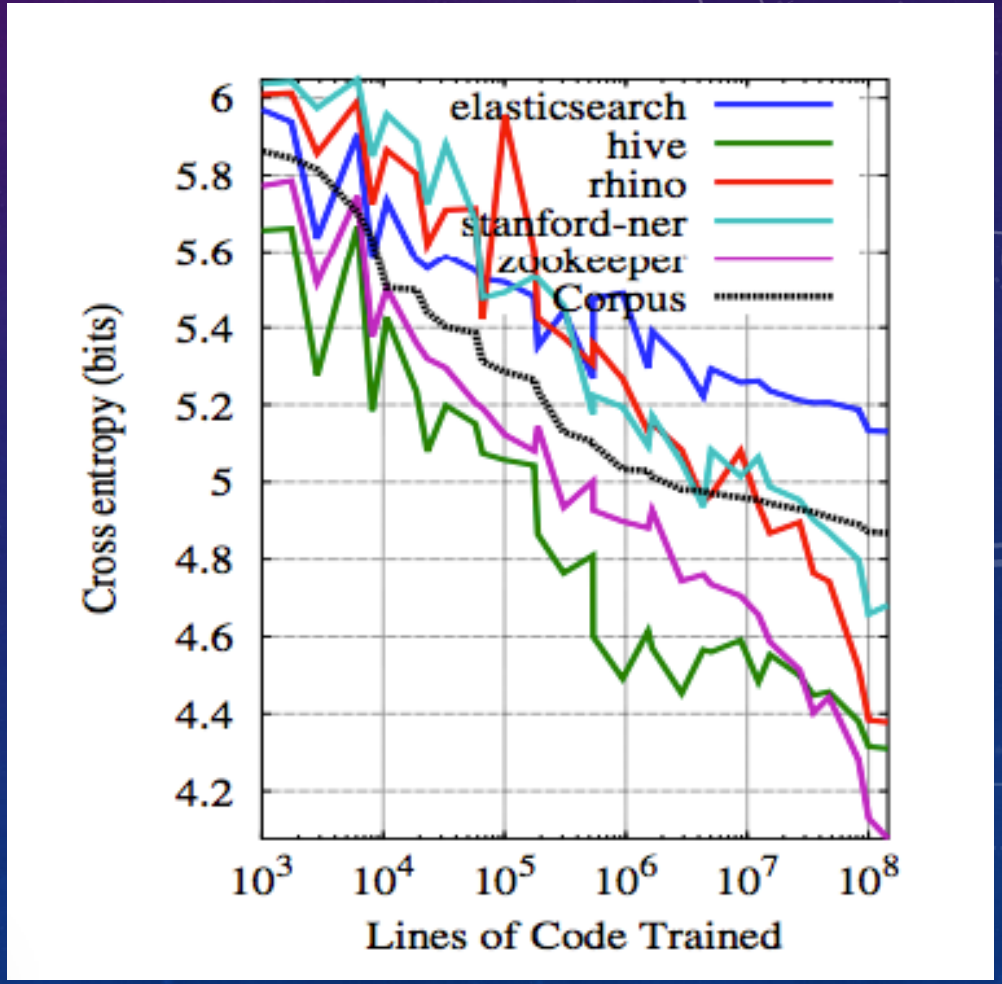
LEARNING SOURCE CODE STRUCTURE

- To understand how n-gram models learn source code, the authors created a new “collapsed model” to see the difference between learning code structure, and identifier names.
- When reading code, the tokenizer outputs only “Identifier”, “Literal”, “IF”, “ FOR”, etc, instead of actual variable names. After only 100, 000 lines of code in the training set, the cross entropy levels off, meaning that it stops seeing new tokens, and that it stops learning new facts.



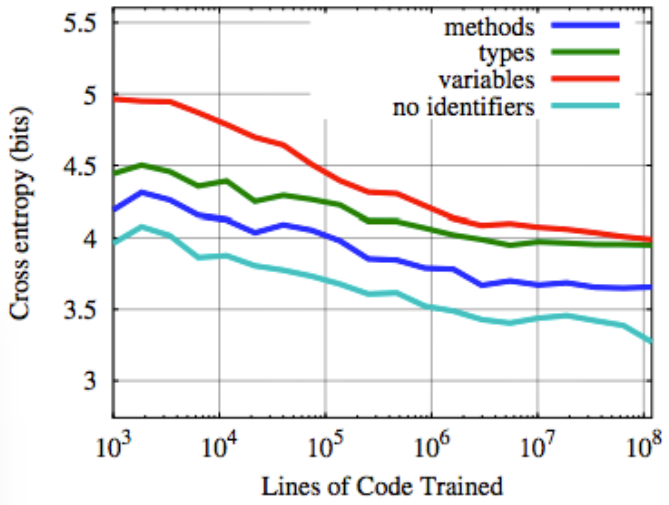
LEARNING SOURCE CODE STRUCTURE

- Using an n-gram model on normal code, the cross entropy continues to decline, suggesting that the extra learning is due to learning identifier names, not code structure.

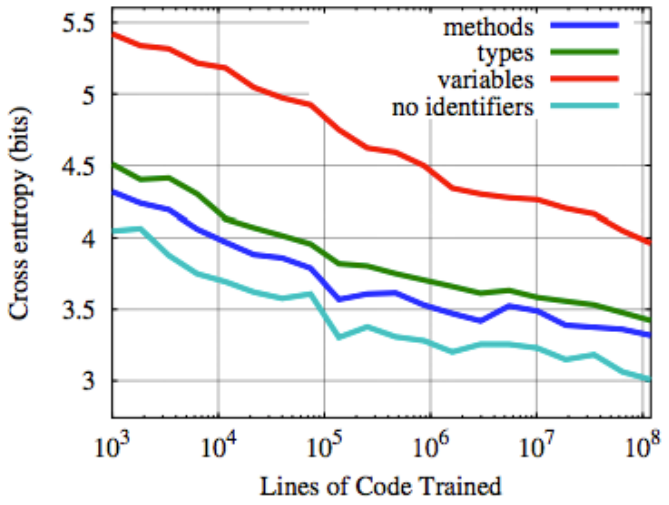


LEARNABILITY OF IDENTIFIERS

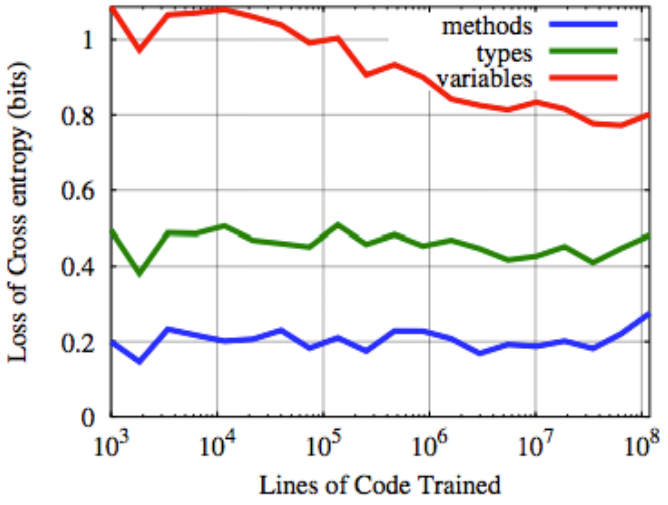
- The collapsed model was modified to output names of types, variables, and method calls independently and then compared.
- Method names are most predictable, then types and variables least



(a) Token learning in elasticsearch



(b) Token learning in rhino



(c) Average cross entropy gap for each token type on five sample projects

VARIABLE NAMES

- In general, variable names are harder to predict, they are more easily learned in large code bases. Therefore, code completion systems will improve as more data are used.
- It makes sense, since you can expect to learn more names (Sarah, Toronto, Airplane) as you read more books / texts.

IDENTIFIER INFORMATION METRIC

- The authors use IIM to determine how domain-specific a file is based on how unpredictable its identifiers are. The intuition is that code that is not domain specific is more reusable. For example, banking software will not be reused in video games, but data structures will be reused in many places.
- The difference in the probability of an identifier and the most probable identifier is the Identifier Information Metric. Some classes had IIM of 1.42 bits UIntMap, NativeDate (1.47), Native Math (1.18). Highly domain specific classes had IIM of 8.68 bits for ByteCode, CompilerEnvirons (4.55), ParserTest (4.52)

CORRELATION WITH OTHER METRICS

- Several metrics exist for calculating code complexity: McCabe's cyclomatic complexity (CC), and lines of code (LOC). The authors compare NGLP to CC and LOC to see if it can be used as a code complexity metric. They don't specify how the NGLP for a file is calculated
- The authors examine the correlation of NGLP to existing metrics in the test code base. Using Spearman's rank correlation gives values of 0.799, and 0.853 for CC and LOC respectively, higher than CC versus LOC (0.778).
- When CC and LOC disagree on complexity, NGLP produces a trade-off between the two metrics.

LEARNING NATURAL CODING CONVENTIONS

- In a following paper, the authors discuss tools for checking code.
- Miltiadis Allamanis et al, ***Learning natural coding conventions***, FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering
- About 25% of code reviews they examined contained suggestions about changing naming.

NATURALIZE FRAMEWORK

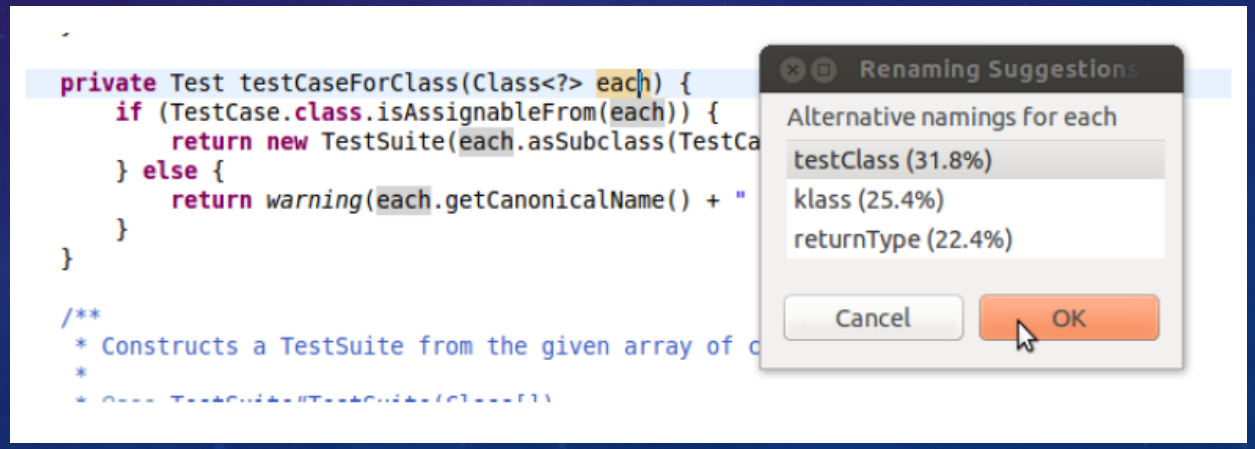
- The authors created the NATURALIZE tools that integrate with Eclipse to check the variable names in a code base. If a variable name is inconsistent with the rest of the code base then the tool will alert the developer.
- The tools look at the current n-grams of code and other probable candidate names from the training code. It only suggests a new name if the difference in probability between the current identifier and the most probable name is above some minimum threshold. This keeps the tool from making suggestions that offer little to no benefit.

NATURALIZE FRAMEWORK

- How to set the threshold? The user specifies a maximum acceptable False Positive Rate α, and keep lowering the threshold until the FPR is equal to α.
- However, you want to keep rare names because they sometimes have important meaning. For rare names, which only happen once or twice in the code, they are replaced by the “Unknown” tag, which then makes them common statistically. Then in certain contexts, an Unknown tag becomes probable as the best suggestion so it accepts a rare token.

CREATING CODE CODE RULES

- Whenever Naturalize is confident enough to prefer one rule over another, the rule can be exported as a code rule for use in Eclipse, Netbeans, etc.
- The number of suggestions to present (k) is a parameter of the algorithm, based on human cognitive preferences.



TESTING NATURALIZE ON GITHUB

- The authors ranked GitHub projects based on the number of Watchers, and Forks to find which are most popular.
- They used leave-one-out cross validation for testing suggestions, using a 5-gram model on the JUnit project.
- They measured a 94% accuracy in its top suggestions for identifier names.

TESTING NATURALIZE ON GITHUB

- Their tool examined several open-source repositories and generated 18 patches giving better names. 14 of the patches were accepted.
- Of the name changes that were rejected, the authors pointed out that the suggested names were used in the code base already. This caused the team to go through the previous code and get rid of all instances of the other variable name.
- “Renaming e to t is no improvement, because we should consistently use e.”

DISCUSSION

- Does common variable names increase readability and reduce bugs?
- What other tools can use this approach of learning common names?
- Could this lead to coding using voice recognition?