Applying Statistical Language Modeling to Genetic Programming

H. Chase Stevens
s1107496

Supervisor: Dr. I. Stark
University of Edinburgh

M.Sc. Project Proposal

April 2016

1 Introduction

2 Background and related work

2.1 Application of language modeling to formal languages

2.2 Genetic programming

3 Methodology

3.1 Compilation of Python corpus

The creation of a language model for Python will require a large and representative corpus of Python source code. While previous work has employed small collections of hand-chosen open-source Python projects for use as a corpus [1] [2], and other work has resulted in the compilation of large corpora for languages such as Java [3], there does not appear to be a suitably large Python source code corpus publicly available for use in this project at present; ergo, the first objective of this project will be to compile one. To do so, source code will be downloaded from open-source projects hosted on the popular code repository host GitHub.

Ideally, the Python corpus compiled for this project will consist of idiomatic, useful, and well-written Python source code; for this project, good-quality code is of special importance, as inexecutable code may still be syntactically valid and parsable. In compiling a similar corpus for the Java programming language, Allamanis and Sutton [3] sought to maintain a minimum level of quality in the GitHub repositories used by filtering out those that had not been forked at least once; in this project, for which I am planning to create a smaller corpus, repositories will instead be used if they exceed some minimum number of stars or forks. To find repositories matching these criteria, the GitHub search API will be used, which allows filtering on not only number of stars and forks but also on project language.

3.2 Creation of language model from Python corpus

While much work has been done on modeling programming languages through the use of n-gram models (e.g. [4], [3]), genetic programming is most naturally applied to tree-like structures as opposed to sequences, and, as such, the language model constructed for this project will be a probabilistic grammar over Python's Abstract Syntax Tree (AST).

When considered as sequences of tokens, programming languages are purported to contain less information per token than natural languages [4]. For example, under a trigram model, Java source code has an estimated average per-token cross-

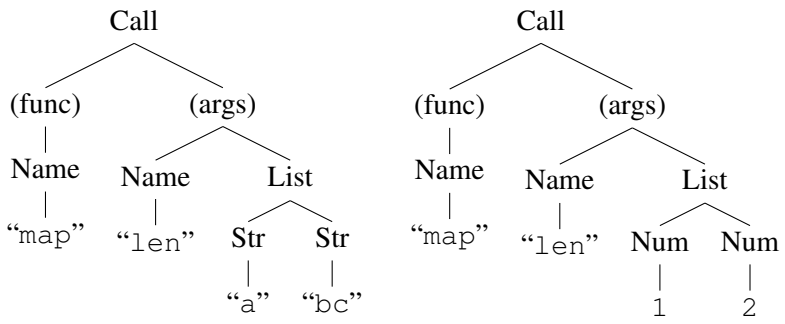

Figure 1: *Comparison of the respective Python AST representations of the semantically valid expression "`map(len, ['a', 'bc'])`" (left) and the invalid expression "`map(len, [1, 2])`" (right). Note that across both cases the types of the direct descendant nodes of "Call" (i.e. Name, Name, and List) are identical.*

entropy of 4.9 bits [3], compared to an estimated cross-entropy value of nearly 8 bits per word in English [5]. However, despite this, an individual node in an AST can provide relatively little semantic information or information about non-children descendant nodes within the tree, easily allowing for the generation of unrunnable programs. Consider the example presented in Figure 1, from which it can be clearly seen that a Call node in Python's AST contains no information about e.g. the types of the arguments with which the specified function is to be called. In order to combat this problem and encourage the generation of valid Python code, I plan to annotate AST nodes with variables learned through expectation maximization (EM), an unsupervised technique which has been shown to reduce perplexity measurements of probabilistic grammars over natural languages without the need for careful manual feature selection [6] [7].

Since EM as an optimization technique is not guaranteed to converge on a global optimum and relies on randomized initializations which have demonstrable effects on the quality of the final grammar obtained [6], I plan to generate several candidate probabilistic grammars using EM, as well as probabilistic grammar without annotations for comparison. Considering that cursory initial investigation suggests that several thousand Python projects on GitHub will meet the proposed criteria for inclusion in the aforementioned source code corpus, generating these language models might prove to be quite computationally intensive. Ideally, I would like to make use of a Hadoop cluster to distribute and parallelize the creation of each language model, thereby reducing overall time spent in model creation. Failing this, an alternative would be to sample a small subset of the corpus for use in training the language models, however, as corpus size has been shown to greatly impact language model efficacy [3], I would be reticent to pursue this course of action.

3.3 Application of language model to genetic programming system

The use of probabilistic grammars in genetic programming has been well established, with many variations having been presented in the literature [**?**]. The key difference between previous work and the system proposed here is that, while previous approaches have assumed initial uniform probability distributions over competing production rules which are then updated in accordance with the relative success of the solutions produced incorporating each production [**?**] [**?**] [**?**], my system will use the aforementioned Python language model to supply initial probabilities, which will similarly be updated as appropriate to meritorious solutions identified for specific tasks.

The direction of genetic programming's search strategy is not alone sufficient to ensure solutions are found in reasonable time: a phenomenon known as "bloat", in which solutions incorporate increasingly large portions of code which do not contribute to their fitness to the task at hand, can have serious deleterious effects on the run-time of the search, as each individual within the genetic programming population becomes more expensive to evaluate. To combat this, I plan to use the parsimony techniques introduced in Poli [**?**] and Poli & McPhee [**?**], which introduce pressures against excessive solution growth both during the fitness evaluation and individual selection phases of genetic programming.

Another important consideration when allowing for the generation of arbitrary code is the safety with which such code can be run; code may exhibit side-effects that harm or render inoperable the underlying system on which it is executed. While not a perfect solution, `pysandbox`[1] offers a reasonable level of protection against undesirable side-effects provided the code has not been malicious written so as to intentionally subvert the sandbox's constraints.

4 Evaluation

In the literature, many genetic programming approaches are evaluated using grammars tailored to particular domains [8]. However, in this project, by necessity, the genetic programming system to be developed will be operating on the general grammar of the Python programming language. Therefore, a suitable task will be one in which solutions lend themselves to representation and manipulation as Python ASTs.

While, often, genetic programming is used to generate code from scratch, recent work has successfully applied genetic programming to the automated repair of pre-existing programs [9]. As in this case manipulation is done on the program's AST, this task would be an ideal evaluative measure for the proposed genetic programming system. Although prior work in this domain has been evaluated on snapshots

[1] https://pypi.python.org/pypi/pysandbox/

of open source software during instances in which the software failed to pass a suite of automated tests, given the scope of this project, evaluation will instead either be performed on test-compliant software which has had bugs deliberately introduced through random AST mutation, or on hand-written examples, dependent on the feasibility of the former given the project's time constraints.

The primary means of evaluating the proposed genetic programming system will be demonstrating how many (if any) software repair tasks it is able to solve. Time permitting, the system's performance, both in terms of number of solutions found and speed with which solutions are identified, will be compared against a baseline genetic programming system which does not incorporate an initial language model as induced using the Python corpus. A small-scale qualitative analysis of the naturalness or idiomaticity of code produced by both systems would also be possible.

5 Timeline

6 Conclusions

References

[1] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 269–280.

[2] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 255–270.

[3] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 207–216.

[4] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[5] P. F. Brown, V. J. D. Pietra, R. L. Mercer, S. A. D. Pietra, and J. C. Lai, "An estimate of an upper bound for the entropy of english," *Computational Linguistics*, vol. 18, no. 1, pp. 31–40, 1992.

[6] T. Matsuzaki, Y. Miyao, and J. Tsujii, "Probabilistic cfg with latent annotations," in *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2005, pp. 75–82.

[7] S. Petrov, L. Barrett, R. Thibaux, and D. Klein, "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, 2006, pp. 433–440.

[8] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong *et al.*, "Genetic programming needs better benchmarks," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 2012, pp. 791–798.

[9] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.