Applying Statistical Language Modeling to Genetic Programming

H. Chase Stevens
s1107496

Supervisor: Dr. I. Stark
University of Edinburgh

M.Sc. Project Proposal

April 2016

Abstract

This paper serves as a proposal for a project to build and evaluate a genetic programming system which bases its initial search strategy on a large corpus of source code. The project, as outlined, will compile a large Python corpus and learn a statistical language model over this, which will then be incorporated into the genetic programming system. This paper also suggests the problem of automated program repair as a suitable evaluation task for the genetic programming system to be created.

1 Introduction

In the field of genetic programming, much research has been devoted to the invention and application of novel techniques for limiting the search space of genetic programming systems and directing systems' search through this space. The project outlined in this proposal will investigate and evaluate one such technique which, to the best of my knowledge, has not appeared in the literature previously. Specifically, this project will attempt to establish a prior probability distribution over potential solutions and use this distribution to inform the genetic programming system's search. To do so, I propose the application of language modeling techniques from the domain of natural language processing to source code.

If successful, the project will establish not only whether the incorporation of a prior language model into a genetic programming system is feasible and capable of finding adequate solutions to standard tasks, but also evaluate the quality of these solutions and speed with which they are found against a baseline genetic programming system. Further to this, the project will incidentally produce a large corpus of Python source code and a language model over this source code, both of which are of potential value to future work.

2 Aim and objectives

The aim of the project proposed in this paper will be to determine the feasibility and possible benefits of incorporating statistical language models into genetic programming systems. In order to accomplish this, the project will entail accomplishing the following objectives:

1. Compiling a corpus of Python source code
2. Selecting a language modeling technique and applying it to this corpus
3. Evaluating the language model against some baseline measure
4. Creating a genetic programming system that incorporates the language model into its search strategy
5. Applying this system to a suitable genetic programming task

Additionally, time permitting, the project would benefit greatly from the accomplishment of an additional objective, namely:

6. Evaluating the system against a baseline genetic programming system which does not incorporate the language model.

3 Background and related work

3.1 Application of language modeling to formal languages

The application of statistical language modeling techniques to source code as adopted from the field of natural language processing is a relatively recent development in the literature. Seminal work presented in Hindle et al. (2012) [1] applied trigram models to Java in order to establish an upper bound for the language's informational content. The authors were also able to leverage this model to produce a tool for next token prediction, with a great deal of success. Further research has shown that these models become even more effective when trained on larger corpora [2].

Subsequent work has expanded beyond modeling source code as sequences of tokens and utilized more sophisticated techniques to create more powerful models. Context-free grammars, for instance, have been used in identifying and coding idioms [3] and in automatically suggesting code fixes from previous source repository history [4].

3.2 Genetic programming

In the most abstract sense, genetic programming is an iterative technique for creating, selecting, and recombining tree-like structures in order to maximize some objective function [5]. Typically, these structures are executable programs, in which each node is representative of some function and each leaf of some input or fixed value. Unlike other evolutionary computation paradigms such as genetic algorithms, in which solutions are represented as fixed-size genomes, genetic programming often deals with conceptually unbounded structures, resulting in an ill-defined and potentially infinite search space [6]. It's for this reason that, in practice, measures must be taken to partition or bound the search space into feasible and infeasible solutions and to correctly and effectively direct the genetic programming system's search through the space of feasible solutions.

One such method for accomplishing this is via the enforcement of relationships between parent and child nodes in the tree. This is known as "strongly-typed genetic programming" [7]. By implementing a strongly-typed genetic programming system, one can ensure that certain classes of run-time exceptions will not occur when evaluating a solution by certifying that values having the wrong type are not passed to functions. However, one drawback to this approach is that type specifications for each function and value used by the system must be (normally manually) specified, as opposed to more lenient genetic programming approaches in which only function arity is typically specified.

An alternative method for both avoiding run-time exceptions and guiding genetic programming system search is the use of probabilistic grammars. When determining each potential solution's fitness with regards to the objective function, relationships between pairs of nodes in the most successful solutions can be observed and used to learn probability distributions [5]. This technique is capable of supplementing the implicit grammars defined within strongly-typed genetic programming system [8] [9], but can also augment

this grammar [10] or induce grammars in an entirely unsupervised fashion [11].

4 Methodology

4.1 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 [12] [13], and other work has resulted in the compilation of large corpora for languages such as Java [2], 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 [2] 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.

4.2 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. [1], [2]), 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 [1]. For example, under a trigram model, Java source code has an estimated average per-token cross-entropy of 4.9 bits [2], compared to an estimated cross-entropy value of nearly 8 bits per word in English [14]. 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

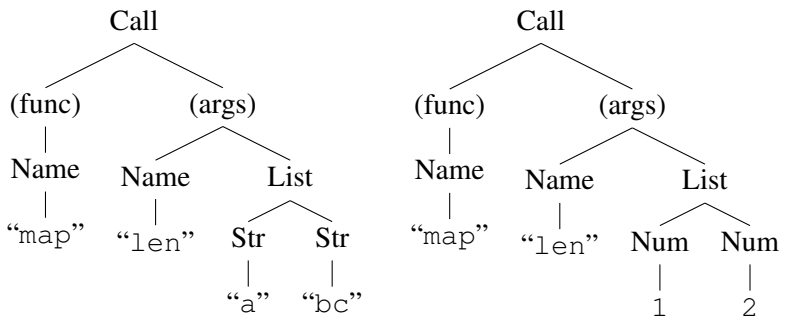

Figure 1: *Comparison of the respective Python AST representations of the semantically valid expression "`map(len, ['a', 'bc'])`" (left) and the invalid expression "`map(len, [1, 2])`" (right). Note that across both cases the types of the direct descendant nodes of "Call" (i.e. Name, Name, and List)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

perplexity measurements of probabilistic grammars over natural languages without the need for careful manual feature selection [15] [16].

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 [15], 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. I therefore plan to make use of the University's Hadoop cluster in order to distribute and parallelize the creation of each language model, thereby reducing overall time spent in model creation. Should this prove an inadequate solution to processing the corpus, 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 [2], I would be reticent to pursue this course of action.

4.3 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 [6]. 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 [9] [8] [17], 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 [18] and Poli & McPhee [19], 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 maliciously written so as to intentionally subvert the sandbox's constraints.

5 Evaluation

In the literature, many genetic programming approaches are evaluated using grammars tailored to particular domains [20]. 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 [21]. 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 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.

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

Date	Milestone	Associated artifact	Time allocated
2/6/2016	Project start date	-	N/A
6/6/2016	Compile corpus	Python corpus	3 days
17/6/2016	Complete language modeling	Language models	9 days
24/6/2016	Evaluate language models	-	5 days
8/7/2016	Implement GP system	GP system	10 days
15/7/2016	Compile evaluation tasks	-	5 days
29/7/2016	Evaluate GP system	-	10 days
19/8/2016	Dissertation deadline	Dissertation	N/A[2]

Table 1: *Proposed timeline of events. Time allocations are given in working days.*

6 Timeline

Table 1 outlines the rough proposed timeline for this project, with corresponding artifacts expected to be generated. A buffer of 3 weeks between the final task's estimated completion date and the dissertation deadline is included, in the event that the completion one or more components exceeds its time allocation.

7 Conclusion

This proposal has outlined a project which will incorporate three novel contributions: the compilation of a large Python source code corpus, the creation of a language model for Python as trained on this corpus, and a genetic programming system which uses this language model to direct its search. This proposal has also presented a task from the literature – program repair – suitable for evaluating the system to be created, on which the system, as equipped with the language model, is expected to perform better than a baseline genetic programming system.

[2]The dissertation will be written incrementally throughout the duration of the M.Sc. project.

References

[1] 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.

[2] 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.

[3] ——, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 472–483.

[4] A. Bomersbach, "Data mining of source code changes," Master's thesis, The University of Edinburgh, 2014.

[5] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu. com, 2008.

[6] Y. Shan, R. I. McKay, D. Essam, and H. A. Abbass, "A survey of probabilistic model building genetic programming," in *Scalable Optimization via Probabilistic Modeling*. Springer, 2006, pp. 121–160.

[7] D. J. Montana, "Strongly typed genetic programming," *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.

[8] A. Ratle and M. Sebag, "Avoiding the bloat with stochastic grammar-based genetic programming," in *Artificial Evolution*. Springer, 2001, pp. 255–266.

[9] P. A. Whigham *et al.*, "Grammatically-based genetic programming," in *Proceedings of the workshop on genetic programming: from theory to real-world applications*, vol. 16, no. 3. Citeseer, 1995, pp. 33–41.

[10] P. A. Whigham, "Inductive bias and genetic programming," in *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*. IET, 1995, pp. 461–466.

[11] I. Tanev, "Implications of incorporating learning probabilistic context-sensitive grammar in genetic programming on evolvability of adaptive locomotion gaits of snakebot," in *Proceedings of GECCO 2004*, 2004.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] C. Keber and M. G. Schuster, "Option valuation with generalized ant programming." in *GECCO*, 2002, pp. 74–81.

[18] R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," in *Genetic programming*. Springer, 2003, pp. 204–217.

[19] R. Poli and N. F. McPhee, "Parsimony pressure made easy," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 2008, pp. 1267–1274.

[20] 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.

[21] 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.