View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Heriot Watt Pure

Automatically Correcting Semantic Errors in Programming Assignments

Ben Trevett, Donald Reay, Nick Taylor

Heriot-Watt University, Edinburgh, UK,
bbt1@hw.ac.uk, d.s.reay@hw.ac.uk, n.k.taylor@hw.ac.uk,

Abstract. MOOCs allow thousands of students to be taught how to program at scale and, using automatic error correction, personalized feedback can also be provided at scale. Current NLP techniques have been used to correct errors in programming assignments, however, these mainly focus on syntactic errors and do not take advantage of NLU techniques to understand the desired semantics. By understanding semantics, there is a potential that the accuracy of error corrections can be vastly improved as focus shifts to correcting errors specific to the desired problem, rather than errors common across all programs. Each step of the process also gives an opportunity to do surrounding work on NLU techniques applied to source code.

Keywords: automatic error correction, natural language processing

1 Introduction

Massive online open courses (MOOCs) have become popular in recent years, due to their ability to offer education at scale. The downside of MOOCs is that the feedback offered to students does not scale. In order to alleviate this problem in the domain of programming assignments, there are *test suites* - a set of desired input-to-output mappings. However, test suites only indicate that a program contains an error and not the location of the error. The ability to highlight the location of the error is a valuable tool in providing feedback for novice programmers.

Natural language processing (NLP) techniques have proven a great success in analysing human languages for tasks such as speech recognition and machine translation. One highly effective NLP technique is using *n-grams*, a Markovian model used to predict the next *symbol* (such as a word or character) in a sequence from the last $(n-1)$ symbols. Hindle et al. [1] show that n-grams are also effective for analysing programming languages. They hypothesise that this is because even though programming languages are ”complex, flexible and powerful”, they are written by humans to be read by other humans, thus can be analysed similarly to natural languages.

Recent advances in *deep learning*, due to the availability of data and improved computing power (via GPUs), have shifted the focus of NLP techniques towards *neural network* based models, especially those using recurrent neural networks

(RNNs). White et al. [2] show that RNNs are consistently superior to n-grams for predicting the next *token* in a corpus of code. This is because n-grams are unable to use any information outside of the $(n-1)$ previous symbol *window* which prevents them from learning long term dependencies, such as remembering to close a bracket. Conversely, RNNs have a *hidden state* which allows them to build a current *context* of the code, allowing them to remember long term dependencies.

2 Related Work

RNNs have been used in program repair in systems such as SynFix [3], which learns a *language model* - a probability distribution of the following token from the previous tokens and context - from correct code. When presented with an erroneous program containing a syntax error, SynFix steps through the code to find areas of high *perplexity*. These are areas where the program under question differs from the learned model of correct code, and thus potentially are the location of errors. SynFix can then suggest a replacement token to reduce the perplexity, potentially fixing the error.

sk_p [4] also uses RNNs that are only trained on correct programs. However, the system is trained to predict a statement from the previous and following statements, rather than a sequence of tokens, and instead of greedily selecting the token to give the lowest perplexity it uses sequence-to-sequence networks [5] to generate the statement one token at a time.

Attention mechanisms [7], used by DeepFix [6], allow the RNN to use the context of the whole program, rather than the surrounding statements, to allow the system to output both the line number of the statement containing the error as well as the correct statement.

It can be argued that using deep learning for syntax errors, although useful, is not entirely necessary as integrated development environments (IDEs) and compilers have robust syntax error checking due to their rigorous parsing which has been built on for decades. Semantic errors should be the focus of error correction as these cause the compiler to run the code without any indication of error, yet produce the incorrect output. However, semantic errors prove a considerable challenge compared to syntax errors, due to semantic errors being unique to the problem under consideration whereas syntax errors are independent of the problem being solved.

It can also be argued that the systems simply learn common syntax errors made by students that appear across all programming assignments and do not take the desired semantics of the program into account. This desired semantics will contain important information for correcting errors, and is currently not explicitly being used by these systems.

3 Aims & Objectives

The aim of this project is to take advantage of natural language understanding (NLU) techniques in order to understand the desired semantics of a submitted program, calculate if and where the program deviates from this and then suggest changes to the program in order for it to achieve the desired semantics. Thus, achieving a way to detect and correct semantic errors.

The programming assignment setting is unique in that the desired semantics are always known ahead of time, this is either via a known correct *reference submission* and/or a test suite. If given a reference submission, it is possible to use NLP and NLU techniques to analyse this program and calculate an abstract representation of it's semantics. This gives the desired semantics of a submitted program, effectively acting as useful information to determine errors in submitted programs.

From this, a second abstract representation of the submitted program can be formed and compared against the reference submission. The two representations can be used to calculate if they are semantically equivalent, and where they differ if not. The difficulty here lies in the fact that there may be multiple syntactically diverse solutions to the given problem, thus we need to use NLU to understand how syntactically different concepts in code may be semantically equivalent, i.e. a `for` loop and a `while` loop cause the code to be written slightly differently, yet do the same thing.

Finally, the abstract semantic differences between the reference and the student submissions must be transformed into feedback. Feedback can be in the form of an error location (a line number or exact token) and a suggested change (either a single token or an entire statement).

Although the project focus is on automatic error correction, overall it covers a wide variety of NLP and NLU, which has little prior work when applied to source code and uses outside of error correction. For example, generative models when applied to discrete outputs is relatively novel.

4 Approach

The issue with the desired system is that semantic errors are unique to each desired semantic functionality and that each desired semantic functionality can be implemented in various methods. Thus, to train the model it is required to have a dataset containing all possible problems and all possible semantically equivalent yet syntactically diverse solutions to each problem. Therefore, it is required to have both a dataset of infinite size and a model with infinite capacity!

Taking advantage of the programming assignment setting again, as we always have a reference submission available we can take advantage of *generative models* such as generative adversarial networks (GANs) [8]. The generative model is trained to generate code conditioned on a semantically equivalent but syntactically different reference submission. This data for training the generative model (and the error correction model) can be obtained from programming challenge websites.

The generative model can thus be used to generate a examples of semantically equivalent submissions when given a reference submission. Semantic errors can then be inserted into these examples to create a problem-specific dataset. The model, which can initially be trained on a general dataset, can then be *fine-tuned* on this generated problem-specific dataset in order to increase accuracy at the desired task. This moves the issue from requiring an infinite dataset and a model with infinite capacity to requiring a distinct model per programming problem.

Creating the abstract representations of the reference submission and the submitted program can be done using RNNs to generate a *context*, similar to work in [7], but without the *decoder*. This is where an RNN steps through the tokens of the program, embedded into a high dimensional space using word embeddings, and the final hidden state is used as a *context* vector.

The two abstract context vectors can be combined, such as concatenation, and fed through an affine layer to produce a single context vector. This single context vector, hypothetically learns the similarities and differences between the reference submission and the submitted program. This context can be used to indicate the location of the semantic error, by passing through an affine layer that outputs the erroneous token or line number, and the correction, by passing to a *decoder* RNN.

5 Conclusion

This project has a main focus on automatically detecting and correcting semantic errors in programming assignments. However, semantic understanding is a difficult challenge comprised of many parts.

References

1. Hindle et al.: On the Naturalness of Software. International Conference on Software Engineering, 34, 837-847 (2012)
2. White et al.: Toward Deep Learning Software Repositories. Conference on Mining Software Repositories, 334–345 (2015)
3. Bhatia S., Singh, R.: Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. In CoRR: abs/1603.06129, URL: http://arxiv.org/abs/1603.06129 (2016)
4. Pu et al.: Sk_P: A Neural Program Corrector for MOOCs. Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, 39–40,(2016)
5. Sutskever et al.: Sequence to Sequence Learning with Neural Networks. Proceedings of the 27th International Conference on Neural Information Processing Systems, 3104–3112, (2014)
6. Gupta et al.: DeepFix: Fixing Common C Language Errors by Deep Learning. AAAI: Conference on Artificial Intelligence, 31, (2017)
7. Bahdanau et al.: Neural Machine Translation by Jointly Learning to Align and Translate. In CoRR: abs/1409.0473, URL: http://arxiv.org/abs/1409.0473 (2014)
8. Goodfellow et al.: Generative Adversarial Nets Proceedings of the 27th International Conference on Neural Information Processing Systems, 2672–2680, (2014)