Homework Assignment # 3

Yulia Belyakova `belyakova.y@northeastern.edu`
Artem Pelenitsyn `pelenitsyn.a@northeastern.edu`

Challenging `code2vec` to a Functional Language

CS 6140 (ML): Project Proposal

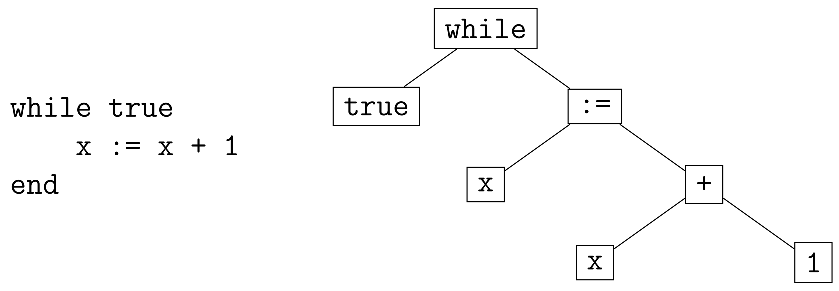
1 Objectives and Significance

Descriptive and consistent names of routines serve at least two goals: (1) improve code readability and maintainability, and (2) facilitate better public APIs[1], as the library users often search for a routine by a name. But coming with a good name agreed upon by other programmers can be hard. Thus, programmers would benefit from an automatic tool that suggests names according to the best practices and naming conventions.

As there are loads of open source projects, there is an opportunity to use machine learning for extracting and predicting descriptive method names [1, 2, 3]. To the best of our knowledge, so far research in this area has been focused on object-oriented languages, including Java, Python, and JavaScript. We are interested in testing the idea in a different context — that of a functional language.

The goal of our work is to challenge the `code2vec` framework, a neural model for predicting method names, published at POPL 2019 [3]. We want to find out whether `code2vec` can accurately predict function names (instead of method names) in the context of a language with higher-order functions, algebraic datatypes, and pattern matching. We pick the Haskell programming language as a representative example of functional languages.

[1]Application Programming Interface



```
while true
    x := x + 1
end
```

Figure 1: An Example of AST

2 Background

(a) Concepts

Routine (function, method) is a *named*, reusable code snippet designed to perform a specific task. Different software projects can define similar or even identical routines. Ideally, all routines that serve the same goal should have the same name. Therefore, the best name for a routine would be the one used the most for the same problem. In general, it is impossible to know whether two routines solve the same problem (i.e. have the same semantics), but if the code is well-structured and short enough, a human can answer this question by simply inspecting the code; a *name-predicting tool* should be able to do the same on a large scale.

AST (Abstract Syntax Tree) is a tree representation of program code where each node denotes a construct from the program (see example in Fig. 1). Given a programming language and a text of a program written in it, it is relatively straightforward to transform the text into a tree (parse the program). Building AST is the first step in most compilers.

Higher-order function is a function that accepts another function as an argument or returns a function as a result. For example, the function `double(f, x) = f(f(x))` is higher-order: its first argument `f` is itself a function, and `double` calls `f` twice.

(b) Related Work and Motivation

There have been multiple attempts to apply machine learning techniques to computer programs. The major challenge here is to represent a code snippet in a way that is both tractable by learning algorithms but also meaningful. One approach is to treat code as text in a natural language and apply natural languages processing (NLP) techniques — the, so called, naturalness hypothesis [4]. The advantage of the approach is its simplicity, but on the downside, the information about the hierarchical nature of the code is totally lost. Therefore, instead of representing a program as a sequence of tokens, one can alternatively look at its abstract syntax tree (AST) [5, 4]. Recently, Alon et al. [2] introduced the *path-based* approach, which considers leaf-to-leaf paths on the program AST. Based on that, Alon et al. [3] developed a method of aggregating the paths into a single fixed-length *code vector*, thus making a **code embedding** [1].

Code embeddings enable the application of various machine learning techniques in the field of programming languages. In particular, the `code2vec` framework [3] implements a neural model that uses code vectors to predict method names for the Java and C`#` programming languages, with extensions of `code2vec`[2] targeting Python and C/C`++`. The authors' previous work [2] had support for predicting variable names and types of expressions in Java, C`#`, Python, and JavaScript, though this functionality is not supported by `code2vec`.

While the variance of the names prediction accuracy across the *supported* languages is about only 10%, to the best of our knowledge, no one has attempted to implement code embeddings for *functional* languages. They support features not typical for languages such as Java or Python, for instance, algebraic data types and pattern matching, so we have to find a way to turn them into paths efficiently. Furthermore, functional programs tend to use higher-order functions more than object-oriented programs use higher-order methods. Therefore, we suspect that this might affect the performance of the `code2vec` approach.

3 Proposed Approach

(a) Data Acquisition

To train a `code2vec` model for the Haskell language, we need to obtain a large amount of code on that language. For this, we will use Hackage[3], a centralized curated set of Haskell packages. The set consists of ∼16 thousand packages with source code available. Hackage has both web interface and a command-line tool to fetch its packages. We are going to implement a Shell script that would query the whole list of packages and download every package from that list.

The enormous amount of code duplication in public code repositories [6] is a clear threat to ML applications to big code. We conjecture that a curated set of packages, such as Hackage, should be less of a victim to the threat. This line of reasoning is applied in [3] too: the authors take top GitHub projects in the hope that "popular" projects are not (as much) susceptible to the duplication problem.

Preliminary experiments and anecdotal evidence suggest several gigabytes of source code on Hackage if consider only the latest versions of the packages (to avoid duplication). While

[2]`https://github.com/tech-srl/code2vec/blob/master/README.md#extending-to-other-languages`
[3]`http://hackage.haskell.org/`

the latest `code2vec` work [3] operated on tens of gigabytes of Java files, Allamanis et al. [4] used about 5 gigabytes of code when working with other languages. Thus, we hope that the code from Hackage will allow us to build a useful model.

(b) Proposed Method and Implementation

Our goal is to add the support for Haskell into the `code2vec` framework. The paper on `code2vec` has been published along the artifact freely available on GitHub[4]. In the README file of the GitHub repository, the authors left several notes on extending their project to analysing different programming languages. In a nutshell, adding support for a new language amounts to writing an *extractor* — a tool capable of extracting path information out of source code. Typically, an extractor will use a parser for corresponding programming language to get an AST of a code snippet. Then, the AST is used by the extractor to generate paths and dump them into the simple textual format expected by `code2vec`.

One challenge in building an extractor is to find a parser for the language of interest. Fortunately, in the case of Haskell, we have one major compiler for the language, the GHC compiler, which supplies its parsing engine in the form of a package (located on Hackage).

Another challenge for path-based code embedding is the amount of paths encoded in AST. Even a simplest program written on any programming language can hold a multitude of paths. The authors of the `code2vec` work identified a feasible cutoff point for discarding paths; this is done using particular properties of the paths. We need to come up with reasonable threshold values too, and make sure that those numbers allow for teaching a useful model given limited computational resources.

Provided the paths are successfully generated for the training set, the `code2vec` framework can be configured to use these data for training a model. The next step is evaluation of the model.

(c) Evaluation Strategy

Given a Haskell code snippet, the model yields a label — the proposed name for the snippet. If the snippet originated from a real function, we know the true label for it, and, therefore, can determine true and false positives. Given those outcomes, the accuracy of labelling can be measured using several well-known statistics, namely: *precision* (number of true positives divided by the total number of elements labeled), *recall* (the number of true positives divided by the total number of elements that actually belong to the positive class) and a combination

[4]`https://github.com/tech-srl/code2vec/`

of them called *F1 score*:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

In fact, the original `code2vec` work [3] used all three to asses the quality of their model. We are going to reuse the metrics for evaluating our project. This enables us to compare the performance of our model to the one built for Java in [3].

The experiment will require dividing the data set into training, validation and testing subsets, as is accustomed. The validation subset is used in [3] to tune hyperparameters concerning the cutoff value for the paths to be considered. Following [4], we plan to randomly pick 80–90% of the data set for training, and divide the rest between validation and testing sets evenly, as in [3].

(d) Expected Outcomes and Fallback Option

Our goal is to obtain a model for predicting function names for Haskell code snippets. The model should achieve reasonable performance by the measures described above. As already mentioned, our project has several threats to validity:

- smaller raw data set (compared to [3]),
- less of computational resources,
- less time to experiment with cutoff values,
- unique features of functional languages not targeted by the initial work.

Given the challenges we face, we might not achieve the goal. We think that it would be an interesting outcome on its own, suggesting that functional languages require some substantial changes in the `code2vec` approach. In that scenario, we are willing to explore a simpler application of code embedding, namely, finding similar code snippets. We conjecture that proximate code vectors could suggest code clones (or near-clones). Using clustering, we hope to detect code duplicates in automated fashion.

4 Individual tasks

We decided to divide the work as follows:

- getting the data from Hackage — Artem;
- cleaning and preparing the data (filtering and shuffling Haskell source files) — Julia;

- extracting ASTs using GHC's parser — Artem;
- extracting paths from ASTs — Julia;
- piping extracted paths to `code2vec` and training the model — Artem;
- validation: experimenting the resulting model, tuning hyperparameters — together;
- testing: measuring accuracy — together;
- writing final report — each person covers their corresponding job.

References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, 2015. doi: 10.1145/2786805.2786849.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, 2018. doi: 10.1145/3192366.3192412.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019. doi: 10.1145/3290353.

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, July 2018. doi: 10.1145/3212695.

[5] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 731–747, 2016. doi: 10.1145/2983990.2984041.

[6] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, October 2017. doi: 10.1145/3133908.