An Evaluation of code2vec Embeddings for Scratch

Benedikt Fein, Isabella Graßl, Florian Beck, Gordon Fraser
University of Passau
Passau, Germany

ABSTRACT
The recent trend of embedding source code for machine learning applications also enables new opportunities in learning analytics in programming education, but which code embedding approach is most suitable for learning analytics remains an open question. A common approach to embedding source code lies in extracting syntactic information from a program’s syntax tree and learning to merge these into continuous distributed vectors (e.g., CODE2VEC). CODE2VEC has been predominantly investigated in the context of professional programming languages, but learning analytics are particularly important in the context of educational programming languages such as SCRATCH. In this paper, we therefore instantiate the popular embedding approach CODE2VEC for SCRATCH programs, create three different classification tasks with corresponding datasets, and empirically evaluate CODE2VEC on them. Our experiments demonstrate that a transfer of CODE2VEC to the educational environment of SCRATCH is feasible. Our findings serve as a basis to apply code embeddings to further educational tasks such as automated detection of misconceptions of programming concepts in SCRATCH programs.

Keywords
code2vec, Scratch, programming education.

1. INTRODUCTION
The application of natural language processing (NLP) and machine learning (ML) methods in the field of software engineering (SE) is gaining popularity in research and industry [27]. A central prerequisite for such machine learning applications on source code is to represent semantically similar code as similar continuously distributed vectors (e.g., CODE2VEC). CODE2VEC has been predominantly investigated in the context of professional programming languages, but learning analytics are particularly important in the context of educational programming languages such as SCRATCH. In this paper, we therefore instantiate the popular embedding approach CODE2VEC for SCRATCH programs, create three different classification tasks with corresponding datasets, and empirically evaluate CODE2VEC on them. Our experiments demonstrate that a transfer of CODE2VEC to the educational environment of SCRATCH is feasible. Our findings serve as a basis to apply code embeddings to further educational tasks such as automated detection of misconceptions of programming concepts in SCRATCH programs.

Programming education research frequently relies on analysis of learners’ programs, for example to automatically detect incorrectly used programming concepts and bugs [1][11][25][26]. Code embeddings bring the promise of novel applications also in the educational domain [9][10][24]; e.g., continuously distributed vectors make it possible to monitor learner trajectories or to detect outliers and anomalous behavior. However, code embeddings are predominantly generated from syntactic features of the source code. For example, CODE2VEC considers the relation of pairs of textual tokens in the context of the syntax tree that results from parsing the source code. Most code embedding approaches are designed for textual programming languages such as Java or Python. Programming education, however, is frequently based on simplified block-based programming languages such as SCRATCH [22]. These programming languages are intentionally designed to reduce the syntactic overhead for learners, and may thus affect the same syntactic properties of programming languages that make them amenable to code embedding models. This may in turn affect the applicability of these models in an education context.

The aim of this paper is to adapt and investigate the popular CODE2VEC code embeddings for the educational programming language SCRATCH. We implement an analysis for SCRATCH programs that extracts the path context information on which CODE2VEC is built. We then create three different classification tasks with corresponding datasets to study the suitability of the resulting embeddings:

- • Girls and boys are known to implement different project types and programming concepts [13][16]; we explore whether code embeddings can capture these nuances.
- • A major characteristic of SCRATCH programs with educational implications [1] is their type (e.g., game, animation, etc.). We explore whether code embeddings enable the prediction of project types from code.
- • The original evaluation of CODE2VEC explored the ability of embeddings to capture semantic content by predicting names of methods. We adapt this task to SCRATCH by predicting names of sprites.

Although SCRATCH code differs from text-based code in important ways affecting code embeddings, such as the structure or size of syntax trees, or the organisation into sprites and scripts rather than classes and methods, we find that CODE2VEC nevertheless performs well at these tasks.
2. BACKGROUND AND RELATED WORK

To understand the application of code2vec to the introductory programming language SCRATCH, this section outlines the concepts and their use cases.

2.1 The Scratch Programming Language

SCRATCH is a block-based programming environment that is particularly designed for learners due to its ease of use through the arrangement of visual blocks. In SCRATCH, the behavior of graphical objects, the sprites, is controlled by means of code blocks, which are assembled to scripts. The code blocks have particular shapes so that they can only be assembled in syntactically valid ways, without the need for the syntactic overhead of text-based programming languages (such as indentation, braces, semicolons, etc.) Code blocks control the appearance and behavior of sprites, as well as interactions with the user.

Besides the intuitive programming user interface, the popularity of SCRATCH is also supported by a rich ecosystem of users sharing their programs publicly and interacting around them. In addition to accessing this information through the user interface, it is also possible to use a REST-API to programmatically access all publicly available data conveniently, which is helpful to enable data mining applications.

SCRATCH programs are categorized into one or more project types: games, stories, animations, music, art, and tutorials. It has been established that some project types require certain programming concepts more than others. Furthermore, it has been repeatedly observed that there are gender-dependent preferences regarding the project type and thus in the programming concepts: While girls mainly prefer programs with storytelling elements, boys implement more programs with game structures.

2.2 Analyzing Scratch Programs

The source code of programs in text-based programming languages is represented using plain text files. In contrast, block-based programs require an intermediate format to describe the program blocks. In particular, SCRATCH programs are represented using JavaScript Object Notation (JSON) format. These JSON files organize programs in terms of their “targets” (stage and sprites), and for each target the JSON file lists its name, its procedures (i.e., custom blocks), scripts, variables, lists, messages, sounds, costumes, and blocks. The blocks are organized as lists, where each element contains a unique identifier as well as the identifiers of the parent and successor blocks, we well as any parameter blocks. Whereas text-based programs are often used directly as input for machine learning approaches, this JSON format is intuitively less suitable for NLP-based approaches.

Static program analysis is usually not conducted on the raw text representation, but the abstract syntax tree (ASTs) intermediate representation, which results from parsing the source code. An AST-like representation is used by the SCRATCH virtual machine in order to interpret SCRATCH programs. The LitterBox analysis framework provides a Java API to parse SCRATCH programs and apply static analysis. Figure 1 shows a publicly shared example project implementing a flappy bird game (Fig. 1a). Figure 2 shows the code of one of its sprites: This sprite represents the ground and the script implements the scrolling motion to simulate movement of the “flappy Mario” character. Figure 1c shows the AST representing the same script. Although this AST is slightly simplified for space reasons, it is noteworthy that this AST is less “abstract” than an AST for other languages would be. For example, while a text-based programming language would likely define an abstract token type for binary operators, with the actual operator as a leaf, this AST contains two leaves together with the path context consists of two leaves together with the path

2.3 Code2vec Code Embeddings

Code2vec learns code embeddings from the syntactical representation of programs through a neural network, where semantically similar code snippets, which are implemented differently but serve the same purpose, represent vectors with a small distance to each other in the vector space. As a basis, code2vec extracts path contexts from the AST: A path context consists of two leaves together with the path context consists of two leaves together with the path

---

Footnotes:

2. For example, https://javaparser.org/
that connects them. For example, consider the two variable tokens in Fig. 1C which are connected by a path that ascends from the leaf node up to the abstract StmtList node, which is the least common ancestor of the two leaves, and then descends to the other leaf:

Code2vec extracts the path contexts for all pairs of leaves in the AST [17]. Then, a neural attention model is used to combine the path contexts to a single vector representation, i.e., the code embedding [6]. The attention mechanism learns to assign weights to path contexts depending on their importance to the semantics of the code snippet, which is assumed to be captured by method names. Consequently, code2vec is applied to individual functions; note that, except for custom blocks, Scratch scripts are not named. The final single vector that represents the code is calculated as a weighted sum over the learned individual vectors for the path contexts [6]. When given an unseen code snippet, the network can then use the learned weights of the paths to calculate such a weighted sum again and therefore assigns a similar vector to semantically similar program code.

2.4 Code Embeddings in CS Education

Various approaches to create code embeddings have recently been considered in an education context. Pich et al. [21] created embeddings for programs written in a text-based educational language by executing unit tests; these embeddings were shown to be useful for predicting which students would benefit from instructor feedback. Azcona et al. [7] demonstrated that code2vec embeddings on Python code are particularly promising on learner’s code when compared to word embeddings applied directly to tokens. Cleuziou et al. [9] proposed a two-step embedding approach where first the AST paths executed by predefined test cases are extracted, and embeddings are created using document embedding techniques. This approach was applied to Python code for the task of propagating teacher feedback. Shi et al. [23] evaluated the two code embedding techniques code2vec and ASTNN [28] for the supervised learning task of bug prediction on Java programs. Paassen et al. [20] introduced the Ast2vec approach for embedding Python programs, with the aim to also support transformations back from embeddings to source code. Finally, Bazzocchi et al. [8] proposed to bypass the embedding problem by using an encoder-decoder architecture directly on Python source code. All of these approaches have in common that they are applied to text-based programming languages.

To the best of our knowledge, there is only one prior investigation of code2vec on block-based programs: Shi et al. [24] applied code2vec to SNAP [14] and clustered the embedded programs to identify clusters representing common misconceptions. Shi et al. demonstrated that for this application code2vec embeddings are superior to other models of the code, such as Bags of Words. In this paper we aim to shed more light on how code2vec generalizes to other tasks. Although SNAP represents programs using XML files that are closer in their structure to regular programs, the resulting ASTs are similar to those of Scratch, and so we expect our findings to generalize also to SNAP.

3. METHOD

To evaluate the code2vec code embeddings for Scratch programs, we investigate the following research questions:

RQ 1 Gender: How accurately can code2vec assess a binary classification task on Scratch programs?
RQ 2 Category: How accurately can code2vec assess a multi-class classification task on Scratch programs?
RQ 3 Sprite naming: How accurately can code2vec assess a classification task on Scratch programs?

3.1 Datasets

The RQs require different datasets for their classification tasks: predicting gender, project type, and sprite names.

RQ1. To answer RQ1, we use a dataset of 317 Scratch programs [13], of which 171 were created by 64 (self-identified) girls and 146 by 68 boys in the range of 8–10 years. The programs are the result of the final task of a multi-day introductory programming course; the children were tasked to implement a Scratch program based on a topic of their own choice. The resulting programs were then manually labelled with the students’ genders. The programs of both genders are comparable in block size (on average: boys 27, girls 22) and number of sprites (on average: girls 6.10, boys 4.78) although the types of blocks and sprites differ [13].

RQ2. To answer RQ2, we sampled 216000 Scratch programs publicly shared between March 2021 and June 2021. Since the REST API of the Scratch website [4] does not provide information about project types, we downloaded programs from each category individually by using GET requests containing certain category names. To create a balanced dataset we subsampled these programs to create a uniform distribution of labels; each program can belong to one or more categories. Since users often use hashtags with all category keywords to gain more visibility, the dataset contains a high percentage of misclassifications. To mitigate these misclassifications, we applied several filtering steps: First, we excluded duplicates and remixed programs. We then also excluded programs tagged as games from the music and tutorial categories, as users often incorrectly add the hashtag music to their game programs simply because they contain background music. In addition, we removed programs in the tutorial, art, music categories that contain their category keyword in the notes and credits section, as users would state credits to the music they included. We evaluated the effectiveness of these filtering steps by manually classifying 10 randomly selected programs from each category, which confirms a decrease of the misclassification rate to 20 % or less in every category. The final dataset consists of 50560 multi-labelled Scratch programs in 40 categories representing various combinations of the six base-categories.

RQ3. To answer RQ3, we created a randomized sample of 530696 Scratch programs publicly shared between April 2007 and April 2020. The data mining was realized by retrieving the 10000 most recently publicly shared Scratch programs each day using the REST API of the Scratch website in the mentioned period.

[https://github.com/LLK/scratch-rest-api/wiki](https://github.com/LLK/scratch-rest-api/wiki)
[https://scratch.mit.edu/explore/programs/all/](https://scratch.mit.edu/explore/programs/all/)
3.2 Data Analysis

Each dataset is divided into training, validation and test dataset with a ratio of 80:10:10. For RQ1, the training set contains 253 programs, the test and validation set 32 programs each; for RQ2 the training set contains 34,639 programs, the test and validation set 4335 programs each.

To answer RQ3, we use a classification task to identify the names of sprites based on their code, thus resembling the method name prediction task[6]. In contrast to RQ1/RQ2, this task considers the ASTs of individual sprites, rather than entire programs. The training set contains 504,503 programs with 4,487,940 sprites, the test set 15,000 programs with 137,429 sprites and the validation set 15,000 programs with 132,875 sprites. The training dataset contains 247,317 different names with 90,802 of them appearing more than once. The 100 most frequent names are used for 580,544 sprites. We use accuracy, precision, recall and F1-score to quantify the performance of the generated models. To better understand the contribution of the code structure versus the literals used in programs, we conduct a small ablation study with a model for each task where literal values are replaced with abstract tokens for their type (string or number).

3.3 Data Preprocessing

The SCRATCH programs must first be processed to extract the path contexts in an appropriate format for the code2vec model. SCRATCH programs are saved as .sb3 files, containing image and audio files as well as the JSON program code. We use LitterBox[12] to parse these JSON files into their AST representation. We extended LitterBox with the extraction and cleanup of the path contexts, such that no additional intermediate representations of the graph structure are needed. The extraction of path contexts ignores non-code related aspects of the AST, such as the positions of blocks in the code editor or post-it style comments.

For RQ1 and RQ2, the entire AST of the program, starting with the Program root node, is considered when extracting path contexts, and the labels are included in the dataset. For RQ3, we extract the path contexts per sprite from their sub-trees (ActorDefinition nodes in LitterBox), as well as the sprite name as the label for the classification task. Similar to how code2vec treats method names, sprite names are split on special characters into subtokens, and the subtokens are normalized to only contain lowercase letters. The final sprite name is then obtained by joining the non-empty subtokens back together with a vertical bar “|” as separating character to support manual interpretation. Additionally, there can be sprites that have the default name (depending on the language settings, e.g., “sprite”) after this normalization step. These are sprites that were not named by the user, and therefore the name cannot be assumed to describe the code. We excluded these sprites from the dataset.

3.4 Neural Network Structure

For all experiments we used the network structure as described by Alon et al.[6] and their implementation[13]. Even after extensive hyperparameter tuning by rerunning the experiment while iteratively changing the parameters one at a time, most of the values as used by Alon et al. for their analysis on Java code[6] also perform best on SCRATCH code (see Table 1). Consequently, we mainly re-used the default or similar values for common hyperparameters. We adapted batch sizes for the different experiments based on the dataset sizes: For the small dataset for RQ1 we reduced the batch size to 16; for RQ2 we used a batch size of 512.

Of the additional hyperparameters specific to the domain of code embeddings, the maximum considered path length and the number of path contexts used for the representation require particular consideration: Increasing the maximum path length allows the model to learn about related elements that are further apart in the source code. However, this also increases the number of generated path contexts. Due to the limited amount of memory available to us during the training phase, a random sample of those has to be chosen. By generating too many path contexts, the chance of missing semantically important ones during sampling increases.

Generally, the maximum path length that should be considered in the case of SCRATCH is higher than for the original Java method name experiment: Even a single sprite encapsulates the full behavior of a figure in a game and can contain multiple scripts, each controlling different aspects of behavior. Therefore, a sprite can be seen as comparable to a class in Java with scripts corresponding to methods. This results in long paths especially for connections between AST leaves placed in different scripts or sprites. Figure 2 shows the average program sizes for the three different datasets, showing that the RQ2 and RQ3 datasets have substantially larger programs than the gender classification task (RQ1). As the project categorization task (RQ2) considers entire programs, only 2% of all paths would be retained when pruning at the maximum of eight, as used in the original Java study (Fig. 3). Consequently, for RQ2 we increased the length to 12, resulting in 18,200 path contexts.

Table 1: Hyperparameters used for Java code[6] compared to the ones for SCRATCH experiments.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>RQ1</th>
<th>RQ2</th>
<th>RQ3</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of contexts</td>
<td>200</td>
<td>200</td>
<td>1000</td>
<td>200</td>
</tr>
<tr>
<td>embedding size</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>max path length</td>
<td>8</td>
<td>8</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>dropout keep rate</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>batch size</td>
<td>1024</td>
<td>16</td>
<td>512</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table source: [6]
Table 2: Top-1 and top-5 accuracy, precision, recall, and F1-score for code2vec when replacing literal values with abstract tokens (AT) and when keeping them.

<table>
<thead>
<tr>
<th>Task</th>
<th>Metric</th>
<th>Prec.</th>
<th>Recall</th>
<th>F1</th>
<th>Acc.</th>
<th>Top-5 Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1</td>
<td>(AT)</td>
<td>78.1</td>
<td>78.1</td>
<td>78.1</td>
<td>78.1</td>
<td>—</td>
</tr>
<tr>
<td>RQ1</td>
<td></td>
<td>90.6</td>
<td>90.6</td>
<td>90.6</td>
<td>90.6</td>
<td>—</td>
</tr>
<tr>
<td>RQ2</td>
<td>(AT)</td>
<td>63.3</td>
<td>59.2</td>
<td>61.2</td>
<td>57.9</td>
<td>93.4</td>
</tr>
<tr>
<td>RQ2</td>
<td></td>
<td>64.1</td>
<td>60.0</td>
<td>62.0</td>
<td>58.9</td>
<td>93.6</td>
</tr>
<tr>
<td>RQ3</td>
<td>(AT)</td>
<td>45.4</td>
<td>41.6</td>
<td>43.4</td>
<td>41.5</td>
<td>51.9</td>
</tr>
<tr>
<td>RQ3</td>
<td></td>
<td>57.4</td>
<td>53.5</td>
<td>55.3</td>
<td>53.8</td>
<td>61.2</td>
</tr>
</tbody>
</table>

4. RESULTS

To evaluate the code2vec embeddings for Scratch, Table 2 shows the performance of the code2vec model on three different classification tasks.

4.1 RQ1: Gender Classification

The gender classification task shows a very high accuracy of 90.6%, suggesting that the projects are quite homogenous within the two gender groups. Grassl et al. [13] observed structural differences between the projects of the two genders, which is reflected by the high accuracy. For example, boys tend to produce interactive projects using event handling blocks and loop control structures, while girls produce more sequential programs. We observe a sharp drop in accuracy when ignoring literals [Table 3]; we conjecture that this is also related to the reported sequential nature of the girls’ projects: Girls tend to produce story-like projects where sprites speak more, thus using more string literals.

RQ1 Summary. code2vec is able to predict the gender based on code with a high accuracy of 90.6%.

4.2 RQ2: Project Type Classification

Compared to the gender classification task, the project category classification task shows a substantially lower accuracy of 58.9% (Table 2). The lower accuracy is likely influenced by the more challenging multi-class classification task, more noise in the data compared to the small gender dataset, and the generally larger projects used in this dataset.

While the accuracy is lower, it is comparable to the performance of the original analysis by Alon et al. [6], which was applied to individual methods. The results on the project category task thus confirm that code2vec can also be applied to whole Scratch programs. We initially assumed that the model requires more path contexts to be able to extract information from the larger scope of the whole program. However, changing the maximum number of contexts to values between 100 and 1000 did not impact the prediction quality. In all cases the accuracy remained between 56.7% and 58.9%. We assume that the model does not actually use the path contexts as such to categorize the programs but instead focuses on the presence or absence of certain block types within the path contexts. For example, the dataset contains the categories “animations”, “games”, and “music”. Games obviously contain many blocks based around the user’s interaction with the program, whereas animations rarely do. Similarly, musical programs can be iden-

---

3 https://github.com/se2p/litterbox
Figure 5: Example prediction with the top-4 paths with the highest attention weights, and the top-3 predictions.

This conjecture is supported by the results without literals (Table 2), which even slightly increases the accuracy. This could be caused by two possible factors: The literal values might not be distinctive for project types; e.g., the movement of sprites in both animations and games relies on similar bounds checks on the visible stage area. Alternatively, some literal values are at least somewhat distinctive for the project type, but the attention mechanism focuses on other more significant differences. In both cases the model uses the attention mechanism to increase the weight for paths that contain project type specific blocks instead of relying on their start and end values. This coincides with our other hypothesis about the required number of path contexts.

RQ2 Summary. The model is able to extract semantic information from whole programs and is able to predict the project type with nearly 60% accuracy.

Table 3: Closest terms in the vector space for the example words “game”, “mario”, “easy” and “sound”.

<table>
<thead>
<tr>
<th>game</th>
<th>mario</th>
<th>easy</th>
<th>sound</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile</td>
<td>luigi</td>
<td>hard</td>
<td>music</td>
</tr>
<tr>
<td>controls</td>
<td>link</td>
<td>medium</td>
<td>music player</td>
</tr>
<tr>
<td>text</td>
<td>sonic</td>
<td>insane</td>
<td>sounds</td>
</tr>
<tr>
<td>word</td>
<td>wario</td>
<td>extreme</td>
<td>audio</td>
</tr>
<tr>
<td>jimmy</td>
<td>yoshi</td>
<td>impossible</td>
<td>sfx</td>
</tr>
</tbody>
</table>

five closest words in the embedding space. All the terms close to “game” clearly have a connection to games themselves: Games tend to have player “profiles”, players interact using “controls”. Similarly, the terms close to “mario” mostly represent other characters from the Super Mario universe.

Unlike the project category task, the literals do contribute to some degree to the performance of the classification (Table 2). For example, Fig. 5 visualizes the most important paths in the “level” sprite from Fig. 1 as determined by the attention mechanism. In particular, the neural network gives the path between the tokens “100” and “480” the most attention; the number 480 represents the width of the stage, and thus is likely to be used in similar contexts.

RQ3 Summary. Code2vec can predict sprite names with a top-5 accuracy of more than 60%, suggesting that semantic information is successfully captured.

5. CONCLUSIONS AND FUTURE WORK

Code embeddings are a trending approach for program analysis, and the computer science education community has recently joined this trend and is exploring novel applications in learning analytics. An important prerequisite for applying machine learning methods is a better understanding of the capabilities and limitations of such approaches.

In order to contribute to such an improved understanding, we evaluated the popular code embedding method Code2vec. This is the first application of Code2vec to the Scratch programming language, and our work has identified a number of important differences between regular, text-based programming languages, and block-based languages like Scratch, such as differences in named entities (e.g., classes or methods) and the overall structure of the resulting AST.

Our experiments on three different classification tasks, predicting gender, project type, and sprite names, suggests that the adaption of Code2vec to the educational domain of Scratch is highly feasible, but there is room for improvement. This suggests that future work should investigate alternative code embedding methods, both those based on syntax (e.g., 28) or graph neural networks 4.

Acknowledgements

This work is supported by the Bayerische Forschungsstiftung (AZ-1520-21, “DeepCode”) and the Federal Ministry of Education and Research (01JA2021, “primary::programming”), as part of the “Qualitätsoffensive Lehrerbildung”, a joint initiative of the Federal Government and the Länder. The authors are responsible for the content of this publication.
6. REFERENCES


