Towards Demystifying Dimensions of Source Code Embeddings

Md Rafiqul Islam Rabin  
University of Houston  
mrabin@central.uh.edu

Arjun Mukherjee  
University of Houston  
amukher6@central.uh.edu

Omprakash Gnawali  
University of Houston  
odgnawal@central.uh.edu

Mohammad Amin Alipour  
University of Houston  
maalipou@central.uh.edu

1 INTRODUCTION

The availability of a large number of mature source code repositories has fueled the growth of "Big Code" that attempts to devise data-driven approaches in the analysis and reasoning of the programs [4, 15] by discovering and utilizing commonalities within software artifacts. Such approaches have enabled a host of exciting applications e.g., prediction of data types in dynamically typed languages [20], detection of the variable naming issues [5], or repair of software defects [18].

Deep neural networks have accelerated innovations in Big Code and have greatly enhanced the performance of prior traditional approaches. The performance of deep neural networks in cognitive tasks such as method name prediction [6] or variable naming [5] has reached or exceeded the performance of other data-driven approaches. The performance of neural networks has encouraged researchers to increasingly adopt the neural networks in processing source code.

Source code representation is the cornerstone of using neural networks in processing programs. Numerous work on devising representations for code in certain tasks [4, 15]. In such representations, the code is represented by a vector of numbers, called embeddings, resulted from training on millions of lines of source code or program traces. The current state of practice in devising such representations includes decisions about the length of code embeddings, code features included in learning, etc. The current approach is highly empirical and tedious; moreover, the analysis and evaluation of the source embeddings are nontrivial.

While there are an increasing number of work on the interpretation and analysis of neural networks for source code, e.g., [14], [33], [28], and [32], to the best of our knowledge there is no work to look at the internal of source code embeddings. In addition to facilitating the interpretation of the behavior of neural models, understanding the source code embeddings would enable researchers and practitioners to optimize neural models, and potential can provide methodologies to objectively compare different representations.

In this work, we report our initial attempts for demystifying the dimensions of source code embeddings, which is aimed at a better understanding of the embedding vectors by analyzing their values and comparing them with understandable features. In particular, we report the results of our preliminary analysis of code2vec embeddings [11], a popular code representation for method name prediction task. More specifically, we use the code2vec embeddings to build SVM models and compare them with SVM models trained on the handcrafted features. We analyze the statistical characteristics of the dimensions in the embeddings.
Our results suggest that the handcrafted features can perform very close to the highly-dimensional code2vec embeddings, and the information gains are more evenly distributed in the code2vec embeddings compared to the handcrafted features. We also find that the code2vec embeddings are more resilient to the removal of dimensions with low information gains than the handcrafted features.

Contributions. This paper makes the following contributions.

- It provides an in-depth analysis of dimensions in code2vec source code embeddings in a small number of methods.
- It compares the performance of handcrafted features with naive representations and code2vec embeddings.

2 BACKGROUND

The code2vec [11] source code representation uses bags of paths in the abstract syntax tree (AST) of programs to represent programs. The model encodes the AST path between leaf nodes and uses an attention mechanism to compute a learned weighted average of the path vectors in order to produce a single code vector of 384 dimensions for each program.

The code2vec [11] was initially introduced to predict the name of method [6], given the method’s body. Figure 1 depicts an example of this task wherein a neural model based on code2vec correctly predicts the name of the method in the Figure as swap.

![Image of method name prediction by code2vec][1]

Figure 1: An example of method name prediction by code2vec[11].

3 METHODOLOGY

To evaluate the code2vec code representation we follow the workflow in Figure 2. We first select a few methods in which we are interested in the analysis of their representations. We then manually select features that best can predict their names. Next, we create binary classifiers for predicting the name of those methods with code2vec embeddings and handcrafted features. Finally, we evaluate and compare the performance of the trained classification models. In the rest of this section, we will describe dataset and selection of methods, feature extraction, classifier creation, baseline classifiers, and evaluation metrics.

3.1 Dataset and Method Selection

Table 1: Top-Ten method name and feature list.

| Name of Method | Feature List |
|----------------|-------------|
| equals         | Instance, Boolean, equals, This |
| main           | println, String |
| setUp          | super, setup, New, build, add |
| onCreate       | bundle, onCreate, setContentView, R |
| toString       | toString, format, StringBuilder, append, + |
| run            | handler, error, message |
| hashCode       | hashCode, TernaryOperator |
| init           | init, set, create |
| execute        | CommandLine, execute, response |
| get            | return, get |

Table 2: Additional code complexity features.

| LOC, Block, Basic Block, Parameter, Local Variable, Global Variable, Loop, Jump, Decision, Condition, Instance, Function, TryCatch, Thread |

GitHub. Overall, it contains about 16M methods where almost 3.5M methods have a unique name. We chose ten most-frequent method names in the JAVA-LARGE dataset and the corresponding method bodies to create a new dataset, Top-Ten, for further analysis.

The reason for restricting our analysis to these methods is twofold. First, the sheer number of method names in JAVA-LARGE prohibits a scalable manual inspection and analysis for all methods. Second, the distribution of method names in JAVA-LARGE conforms to the power-law; that is, relatively few method names appear frequently in the dataset while the rest of method names appear rarely in the dataset. Therefore, the performance of any classifier on JAVA-LARGE heavily relies on its performance on the few frequent method names. Column “Name of Method” in Table 1 lists the names of ten most-frequent methods that we chose for our analysis.

Table 1: Top-Ten method name and feature list.

Deduplication of the Top-Ten dataset. As noted by [1], the JAVA-LARGE dataset suffers from duplicate methods that can inflate the results of the prediction. We removed duplicate methods in the dataset following the steps outlined in [1] and used the same parameters for deduplication thresholds: key-jaccard-threshold, \( t_0 = 0.8 \) and jaccard-threshold, \( t_1 = 0.7 \).

Dataset for each Top-Ten method. For each method \( M \) in Top-Ten, we create a training set that constitutes from 1000 randomly selected positive examples (methods with name \( M \)), and 1000 randomly selected negative examples (any method but \( M \)) from the deduplicated Top-Ten training set. For the validation set and test set, we select all the positive examples and the same number of randomly selected negative examples from the deduplicated Top-Ten validation set and test set, respectively. Table 3 shows the size of the dataset for each Top-Ten method.

![Image of method name prediction by code2vec][1]
Table 3: Size of the dataset for each Top-Ten method.

| Name of Method | #Training | #Validation | #Test |
|----------------|-----------|-------------|-------|
| equals         | 2000      | 1212        | 1778  |
| main           | 2000      | 1220        | 2032  |
| setUp          | 2000      | 1220        | 1424  |
| onCreate       | 2000      | 1876        | 1484  |
| toString       | 2000      | 586         | 1278  |
| run            | 2000      | 876         | 1558  |
| hashCode       | 2000      | 534         | 770   |
| init           | 2000      | 892         | 2504  |
| execute        | 2000      | 498         | 702   |
| get            | 2000      | 780         | 670   |

3.2 Extracting Handcrafted Features

Method-only features. For each method, two authors do their best effort to draw discriminant features by inspecting the training dataset. Table 1 shows the handcrafted features for each method, in total, 33 features for ten methods.

Code complexity features. An important metric of interest might be adding code complexity features. Similar methods may have certain patterns such as the number of lines, variables, or conditions. Therefore, we further extend the handcrafted features with an additional 14 code complexity features shown in Table 2. Thus, the handcrafted features become a union of 47 features including the code complexity features. Note that we only focus on the simpler code complexity features shown in Table 2 as our study is limited to the methods, and thus the class level or project level code complexity metrics do not apply to our study.

Feature Extraction. We use the JavaParser [37] tool to parse the methods in the dataset and extract the handcrafted features. We consider the 33 handcrafted features of methods (Table 1) as (a) binary vectors, and (b) numeric vectors. For the binary vectors, we use 1 and 0 that denote the presence or absence of individual features in the method, respectively. For the numeric vectors, we count the number of occurrences of features in the program, and in the end, we normalize them using StandardScaler [31] to map the distribution of values to a mean value of 0 and a standard deviation of 1. The 14 complexity features (Table 2) are always considered as numeric values.

3.3 Classification Models

Support Vector Machines. Support Vector Machines (SVM) are one of the most popular traditional supervised learning algorithms that can be used for classification and regression on linear and non-linear data [13, 17, 24]. SVM uses the concept of linear discriminant and maximum margin to classify between classes. Given the labeled training data points, SVM learns a decision boundary to separate the positive points from the negative points. The decision boundary is also known as the maximum margin separating hyperplane that maximizes the distance to the nearest data points of each class. The decision boundary can be a straight line classifying linear data in a two-dimensional space (i.e. linear SVM using linear kernel) or can be a hyperplane classifying non-linear data by mapping into a higher-dimensional space (i.e. non-linear SVM using RBF kernel).

Classifiers. For each method \( M \), we create two SVM classification models: SVM-HANDCRAFTED and SVM-CODE2VEC. SVM-CODE2VEC uses the code2vec embeddings of the programs in training the SVM model, which is a single fixed-length code embedding (384 dimensions) that represents the source code as continuous distributed vectors for predicting method names. SVM-HANDCRAFTED uses the vector of the handcrafted features (33 dimensions without complexity features, and 47 dimensions with complexity features) to train an SVM model.

Training. We use the SVMlight 1, an implementation of Support Vector Machines (SVMs) in C [27], to train the classification models in the experiments.

Since the performance of SVM depends on its hyper-parameters, we run the grid search algorithm [12] for hyper-parameter optimization. We train SVMs with tuned parameters on handcrafted features and code2vec embeddings for each method name.

3.4 Naive Sequence-based Neural Baselines

We also create two sequence-based baselines to compare our handcrafted features: (a) CharSeq where the program is represented by

---

1http://svmlight.joachims.org/
a sequence of characters in the program, and (b) TokenSeq where a sequence of tokens in the program represent the program.

**CharSeq.** For character-based representation, we first remove comments from the body of the method and save the body as a plain string. Then we create a list of ASCII characters by filtering out all non-ASCII characters from the string of body. After that, we create a character-based vocabulary with the unique ASCII characters found in the training+validation set of the Top-Ten dataset (the character-based vocabulary stores 94 unique ASCII characters). Finally, we encode the method body by representing each character with its index in the character-based vocabulary.

**TokenSeq.** For token-based representation, we modify the JavaTokenizer tool [1] to get the sequence of Java tokens from the body of the method. After that, we create a token-based vocabulary with the unique tokens found in the training+validation set of the Top-Ten dataset (the token-based vocabulary stores 108106 unique tokens). Finally, we encode the method body by representing each token with its index in the token-based vocabulary.

**Training Naive Models.** We train 2-layer bi-directional GRUs [16] on character-based representation (CharSeq) and token-based representation (TokenSeq) for predicting the method name. The classifier on CharSeq and TokenSeq are referred to as GRU-CharSeq and GRU-TokenSeq, respectively.

### 3.5 Evaluation Metrics

We use the following metrics as commonly used in the literature [5, 11] to evaluate the performance of handcrafted features. Suppose, \( tp \) denotes the number of true positives, \( tn \) denotes the number of true negatives, \( fp \) denotes the number of false positives, and \( fn \) denotes the number of false negatives in the results of the classification of a method on the test data.

**Accuracy** indicates how many predicted examples are correct. It is the ratio of the correctly predicted examples to the total examples of the class.

\[
\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}
\]

**Precision** indicates how many predicted examples are true positives. It is the ratio of the correctly predicted positive examples to the total predicted positive examples.

\[
\text{Precision} = \frac{tp}{tp + fp}
\]

**Recall** indicates how many true positives examples are correctly predicted. It is the ratio of the correctly predicted positive examples to the total examples of the class.

\[
\text{Recall} = \frac{tp}{tp + fn}
\]

**F1-Score** is the harmonic mean of precision (P) and recall (R).

\[
F_1\text{-Score} = \frac{2 \cdot P \cdot R}{P + R}
\]

### 4 RESULTS

In this section, we will describe the experimental results including choice of handcrafted features, comparison of classifiers, and visualization. Each classifier is trained on the corresponding training set, tuned on the validation set, and later evaluated on a separate test set. In this section, the classifiers on CharSeq, TokenSeq, HC(Binary)+CX(Norm), and code2vec feature vectors are referred to as GRU-CharSeq, GRU-TokenSeq, SVM-Handcrafted, and SVM-code2vec, respectively.

| Method | Feature Vectors | Precision | Recall | F1-Score |
|--------|-----------------|-----------|--------|----------|
| equals | HC(Binary)      | 98.54     | 98.88  | 98.71    |
|        | HC(Norm)        | 98.20     | 97.98  | 98.09    |
|        | HC(Binary)+CX(Norm) | 99.21     | 98.54  | 98.87    |
|        | HC(Norm)+CX(Norm) | 98.99     | 98.76  | 98.87    |
| main   | HC(Binary)      | 94.62     | 96.85  | 95.72    |
|        | HC(Norm)        | 91.70     | 94.59  | 93.12    |
|        | HC(Binary)+CX(Norm) | 94.72     | 97.15  | 95.92    |
|        | HC(Norm)+CX(Norm) | 91.04     | 94.98  | 92.97    |
| setUp  | HC(Binary)      | 87.70     | 86.10  | 86.89    |
|        | HC(Norm)        | 78.90     | 90.87  | 84.46    |
|        | HC(Binary)+CX(Norm) | 90.26     | 93.68  | 91.94    |
|        | HC(Norm)+CX(Norm) | 87.53     | 92.70  | 90.04    |
| onCreate| HC(Binary)     | 100.00    | 92.99  | 96.37    |
|        | HC(Norm)        | 100.00    | 92.86  | 96.30    |
|        | HC(Binary)+CX(Norm) | 99.86     | 93.13  | 96.38    |
|        | HC(Norm)+CX(Norm) | 100.00    | 92.45  | 96.08    |
| toString| HC(Binary)    | 93.41     | 97.65  | 95.48    |
|        | HC(Norm)        | 93.56     | 95.46  | 94.50    |
|        | HC(Binary)+CX(Norm) | 95.57     | 94.52  | 95.04    |
|        | HC(Norm)+CX(Norm) | 94.81     | 94.37  | 94.59    |
| run    | HC(Binary)      | 62.03     | 61.87  | 61.95    |
|        | HC(Norm)        | 60.51     | 75.74  | 67.27    |
|        | HC(Binary)+CX(Norm) | 69.24     | 66.75  | 67.97    |
|        | HC(Norm)+CX(Norm) | 69.55     | 70.09  | 69.82    |
| hashCode| HC(Binary)   | 97.06     | 94.29  | 95.65    |
|        | HC(Norm)        | 96.85     | 95.84  | 96.34    |
|        | HC(Binary)+CX(Norm) | 98.95     | 97.92  | 98.43    |
|        | HC(Norm)+CX(Norm) | 98.19     | 98.44  | 98.31    |
| init   | HC(Binary)      | 74.73     | 94.25  | 83.36    |
|        | HC(Norm)        | 73.55     | 92.17  | 81.81    |
|        | HC(Binary)+CX(Norm) | 77.72     | 90.58  | 83.66    |
|        | HC(Norm)+CX(Norm) | 75.43     | 91.69  | 82.77    |
| execute| HC(Binary)     | 76.25     | 86.89  | 81.22    |
|        | HC(Norm)        | 63.60     | 94.59  | 76.06    |
|        | HC(Binary)+CX(Norm) | 80.67     | 82.05  | 81.35    |
|        | HC(Norm)+CX(Norm) | 76.36     | 92.02  | 83.46    |
| get    | HC(Binary)      | 86.76     | 95.82  | 91.07    |
|        | HC(Norm)        | 84.96     | 91.04  | 87.89    |
|        | HC(Binary)+CX(Norm) | 89.89     | 95.52  | 92.62    |
|        | HC(Norm)+CX(Norm) | 88.54     | 92.24  | 90.35    |
**Table 5: Type of feature vectors.**

| Feature Vectors | Definition |
|-----------------|------------|
| CharSeq         | A sequence of ASCII characters represented by its index in a character-based vocabulary. |
| TokenSeq        | A sequence of Java tokens represented by its index in a token-based vocabulary. |
| HC(Binary)      | The 33 handcrafted features of methods as binary vectors. |
| HC(Norm)        | The 33 handcrafted features of methods as numeric vectors. |
| HC(Binary)+CX(Norm) | HC(Binary) with the additional 14 complexity features as numeric vectors. |
| HC(Norm)+CX(Norm) | HC(Norm) with the additional 14 complexity features as numeric vectors. |
| code2vec        | The code vectors of 384 dimensions from code2vec model. |

**Table 6: Average results on the Top-Ten dataset.**

| Feature Vectors | Accuracy | Precision | Recall | \(F_1\)-Score |
|-----------------|----------|-----------|--------|---------------|
| CharSeq         | 38.65    | 26.02     | 38.65  | 30.57         |
| TokenSeq        | 70.58    | 60.38     | 70.59  | 63.37         |
| HC(Binary)      | 88.32    | 87.11     | 90.56  | 88.64         |
| HC(Norm)        | 86.27    | 84.18     | 92.11  | 87.58         |
| HC(Binary)+CX(Norm) | 90.14    | 89.61     | 90.98  | 90.22         |
| HC(Norm)+CX(Norm) | 89.36    | 88.04     | 91.77  | 89.73         |
| code2vec        | 93.73    | 95.54     | 91.38  | **93.24**     |

### 4.1 Choice of Handcrafted Features

**Table 4** shows the detailed result of handcrafted features on the Top-Ten dataset where the **bold** values represent the best results and the underlined values represent the second-best result. In this table, "HC" stands for handcrafted features. "HC(Binary)" and "HC(Norm)" denote the handcrafted features as binary vectors and numeric vectors, respectively. Similarly, "CX(Norm)" is to indicate the additional complexity features as numeric vectors.

#### 4.1.1 Binary vectors vs. Numeric vectors

Figure 3 depicts how the choice of presence (binary vectors) or number of occurrences (numeric vectors) influences the quality of handcrafted features. We compare binary vectors and numeric vectors on (a) method-only features: HC(Binary) vs. HC(Norm), and (b) method+complexity features: HC(Binary)+CX(Norm) vs. HC(Norm)+CX(Norm). In Figure 3(a) and 3(b), the blue line shows the \(F_1\)-Score when the features are considered as binary vectors and the orange line shows the \(F_1\)-Score when the features are considered as numeric vectors. According to Figure 3(a), in most cases, the HC(Binary) are comparatively better than the HC(Norm) except for the ‘run’ and ‘hashCode’ methods where the difference are 5.32% and 0.69%, respectively. Similarly, in most cases, the HC(Binary)+CX(Norm) are comparatively better than the HC(Norm)+CX(Norm) in Figure 3(b) except for the ‘run’ and ‘execute’ methods where the difference are 1.85% and 2.11%, respectively. The average \(F_1\)-Score of Table 6 also shows that the HC(Binary) is almost 1% better than the HC(Norm) and the HC(Binary)+CX(Norm) is almost 0.5% better than the HC(Norm)+CX(Norm). This can suggest that only the presence of features can be used to recognize a method, instead of counting the number of occurrences of features.

**Observation 1:** The presence of a feature can be used to recognize a method instead of counting the number of occurrences of that feature in programs. On average, the choice of binary vectors has increased the \(F_1\)-Score up to 1% than the numeric vectors.

#### 4.1.2 Impact of Additional Complexity Features

Figure 4 depicts the importance of complexity features on the quality of handcrafted features. We compare method-only features and method+complexity features on (a) binary vectors: HC(Binary) vs. HC(Binary)+CX(Norm), and (b) numeric vectors: HC(Norm) vs. HC(Norm)+CX(Norm). In Figure 4(a) and 4(b), the blue line and orange line shows the \(F_1\)-Score of method-only features and method+complexity features, respectively. According to Figure 4(a), in most cases, the HC(Binary)+CX(Norm) are comparatively better than the HC(Binary) except for the ‘toString’ method where the difference is 0.44%. Similarly, in most cases, the HC(Norm)+CX(Norm) are comparatively better than the HC(Norm) in Figure 4(b) except for the ‘main’ and ‘onCreate’ methods where the difference are 0.15% and 0.22%, respectively. The average \(F_1\)-Score of Table 6 also shows that the HC(Binary)+CX(Norm) is almost 1.6% better than the HC(Binary) and the HC(Norm)+CX(Norm) is almost 2.2% better than the HC(Norm). This can suggest that the code complexity features can be useful to better recognize a method, especially for some methods (i.e. ‘setUp’, ‘run’, ‘hashCode’, and ‘execute’) where the improvements for additional code complexity features are almost 3 ~ 7%.

**Observation 2:** The code complexity features can be useful to better recognize a method along with the method-only features. On average, the additional code complexity features have increased the \(F_1\)-Score up to 2.2% than the method-only features.

### 4.2 Comparison of Classifiers

**Table 7** shows the detailed result of different feature vectors on the Top-Ten dataset where the **bold** values represent the best results and the underlined values represent the second-best result. We also draw the commonly used explanatory data plots (harploots over method names in Figure 5a and boxplots over feature vectors in Figure 5b) to visually show the distribution of results on the Top-Ten dataset. As shown in the previous section (Section 4.1), in most cases, the binary vectors perform relatively better than the numeric vectors, and the code complexity features also improve the performance for handcrafted features. Therefore, in this section, we mainly compare the result of HC(Binary)+CX(Norm) from handcrafted features.

#### 4.2.1 SVM-Handcrafted vs. Sequence-based Baselines

In this section, we compare our handcrafted features against the following two sequence-based baselines: (a) a sequence of ASCII characters (CharSeq), and (b) a sequence of Java tokens (TokenSeq). According to Table 7 and Figure 5a, for all methods, our SVM-Handcrafted outperforms both GRU-CharSeq and GRU-TokenSeq by a large margin for predicting method name.
(a) Method-only features.  
(b) Method+Complexity features.

Figure 3: Binary vectors vs. Numeric vectors.

(a) Binary vectors.  
(b) Numeric vectors.

Figure 4: Impact of additional complexity features.

(a) Barplots over method names.  
(b) Boxplots over feature vectors.

Figure 5: Comparison of classifiers on Top-Ten dataset.
Table 7: Result of different feature vectors on Top-Ten dataset.

| Method | Feature Vectors | Precision | Recall | \(F_1\)-Score |
|--------|----------------|-----------|--------|---------------|
| equals | CharSeq        | 50.97     | 74.02  | 60.37         |
|        | TokenSeq       | 99.20     | 97.53  | 98.36         |
|        | HC(Binary)+CX(Norm) | 99.21  | 98.54  | 98.87         |
|        | code2vec       | 99.55     | 99.10  | 99.32         |
|        |                |           |        |               |
| main   | CharSeq        | 0.00      | 0.00   | 0.00          |
|        | TokenSeq       | 84.38     | 65.94  | 74.03         |
|        | HC(Binary)+CX(Norm) | 94.72  | 97.15  | 95.92         |
|        | code2vec       | 98.72     | 98.52  | 98.62         |
| setup  | CharSeq        | 26.12     | 59.83  | 36.36         |
|        | TokenSeq       | 42.93     | 89.19  | 65.96         |
|        | HC(Binary)+CX(Norm) | 90.26  | 93.68  | 91.94         |
|        | code2vec       | 99.26     | 94.10  | 96.61         |
| onCreate | CharSeq      | 59.89     | 87.74  | 71.19         |
|         | TokenSeq       | 94.70     | 95.11  | 93.08         |
|         | HC(Binary)+CX(Norm) | 99.86  | 93.13  | 96.38         |
|         | code2vec       | 100.00    | 99.06  | 99.53         |
| toString | CharSeq      | 51.64     | 74.02  | 60.84         |
|         | TokenSeq       | 85.14     | 88.73  | 86.90         |
|         | HC(Binary)+CX(Norm) | 95.57  | 94.52  | 95.04         |
|         | code2vec       | 97.37     | 98.44  | 97.90         |
| run    | CharSeq        | 25.36     | 27.47  | 26.37         |
|        | TokenSeq       | 37.96     | 51.99  | 43.88         |
|        | HC(Binary)+CX(Norm) | 69.24  | 66.75  | 67.97         |
|        | code2vec       | 86.30     | 62.26  | 72.33         |
| hashCode | CharSeq      | 30.18     | 52.99  | 38.45         |
|         | TokenSeq       | 74.70     | 97.40  | 85.55         |
|         | HC(Binary)+CX(Norm) | 98.95  | 97.92  | 98.43         |
|         | code2vec       | 97.74     | 99.74  | 99.74         |
| init   | CharSeq        | 0.00      | 0.00   | 0.00          |
|        | TokenSeq       | 0.00      | 0.00   | 0.00          |
|        | HC(Binary)+CX(Norm) | 77.72  | 90.58  | 83.66         |
|        | code2vec       | 88.74     | 87.54  | 88.11         |
| execute | CharSeq       | 2.44      | 0.28   | 0.51          |
|         | TokenSeq       | 41.04     | 31.34  | 35.54         |
|         | HC(Binary)+CX(Norm) | 80.67  | 82.05  | 81.35         |
|         | code2vec       | 93.44     | 85.19  | 89.12         |
| get    | CharSeq        | 13.55     | 10.15  | 11.60         |
|        | TokenSeq       | 43.77     | 92.24  | 59.37         |
|        | HC(Binary)+CX(Norm) | 89.89  | 95.52  | 92.62         |
|        | code2vec       | 92.33     | 89.85  | 91.07         |

Even in some cases, GRU-CharSeq (i.e. main, init, and execute) and GRU-TokenSeq (i.e. init) fail to predict the method name. The boxplots in Figure 5b indicates that the variance of \(F_1\)-Scores among methods are also very significant for GRU-CharSeq and GRU-TokenSeq. The average \(F_1\)-Score of Table 6 also shows that the SVM-HANDCRAFTED is 59.65% and 26.85% better than the GRU-CharSeq and the GRU-TokenSeq, respectively.

**Observation 3:** The handcrafted features significantly outperform the sequence of characters (by 59.65%) and the sequence of tokens (by 26.85%) for predicting method name.

### 4.2.2 SVM-HANDCRAFTED vs. SVM-code2vec

In this section, we compare our handcrafted features with the path-based embedding of code2vec [11]. According to Table 7 and Figure 5, the SVM-code2vec performs better than the SVM-HANDCRAFTED but the difference is not always significant. When the \(F_1\)-Score of SVM-code2vec is near perfect (i.e., equals, onCreate, and hashCode), the \(F_1\)-Score of SVM-HANDCRAFTED is also higher and very close to the SVM-code2vec. Similarly, for some other methods (i.e., run, init, and execute), they both perform relatively worst. However, there are some cases where the difference between SVM-code2vec and SVM-HANDCRAFTED is significant, for example, SVM-code2vec shows almost 8% improvement over SVM-HANDCRAFTED to classify the 'execute' method. On the other hand, SVM-HANDCRAFTED is 1.5% better than SVM-code2vec to classify the 'get' method. The average \(F_1\)-Score also shows that the SVM-HANDCRAFTED is 1.5% better than SVM-code2vec to classify the 'get' method. The average \(F_1\)-Score of SVM models for methods such as 'main' and 'setUp' by using features with higher dimensional embeddings is symmetric while is highly skewed in handcrafted features.

**Observation 4:** The handcrafted features with a very smaller feature set can achieve highly comparable results to the higher dimensional embeddings of deep neural model such as code2vec.

### 4.3 Information Gains and Importance of Dimensions

Figure 6 depicts the distribution of information gains of each dimension, i.e., feature, in the 'equals' dataset. It suggests that the information gains of features in code2vec embeddings is on average higher than the information gains of features in handcrafted features. However, the distribution of information gains in code2vec embeddings is symmetric while is highly skewed in handcrafted features.

We used the information gains and created new SVM models for methods such as 'main' and 'setUp' by using features with top 25% of information gains. The \(F_1\)-score of SVM models for
binary handcrafted features (HC(BINARY)) with features of top 25% information gains were 93.5% and 80.11%, for ‘main’ and ‘setUp’, respectively, while these value for top 25% code2vec dimensions were 98.62% and 96.28%, respectively. It shows that the handcrafted features suffered a higher loss of performance than their code2vec embeddings counterparts. It may suggest that a large portion of code2vec embeddings might be unnecessary for the acceptable classification, hence, the size of embedding can be reduced.

**Observation 5:** Compare to the handcrafted features, the information gains are more evenly distributed in the code2vec embeddings. Moreover, the code2vec embeddings are more resilient to the removal of dimensions with low information gains than the handcrafted features.

### 4.4 Visualization of Feature Vectors

To better understand how the features separate the positive and negative examples in the dataset we used t-SNE [30] to project the feature vectors in code2vec embeddings and handcrafted features into two-dimensional space. For illustration, we only visualize the method with best performing classifiers (i.e. ‘equals’) in Figure 7 and the method with worst performance classifiers (i.e. ‘run’) in Figure 8, for the CODE2VEC, HC(BINARY) and HC(Binary)+CX(Norm), respectively. Points from the same color (positive examples are in green color and negative examples are in red color) should tend to be grouped close to one another.

**‘equals’ method.** Figure 7 indicates that the data points are generally well grouped for the best method (‘equals’) where the positive points are quite distinct from the negative points. The data points form a cluster of positive points in the middle of Figure 7a and are almost linearly separable in Figure 7b and 7c. All show a good measure of separability as the $F_1$-Scores are nearly 100%.

**‘run’ method.** Figure 8 indicates that the data points are hardly separable. The $F_1$-Score of Figure 8a is around 10% higher than Figure 8b, thus the data points appear more scattered in Figure 8b than in Figure 8a. Similarly, the $F_1$-Score of Figure 8c is around 6% higher than Figure 8b, thus the data points in Figure 8c are relatively less scattered than in Figure 8b.

Although t-SNE plots are not objective ways to compare two embeddings, it may provide an intuition about the separability of methods based on the corresponding feature embeddings. The figures might suggest that the high-dimensional code2vec tends to produce a more complex hypothesis class than necessary, compared to the handcrafted features. Using too complex hypothesis class may increase the chances of overfitting in training the models.

### 5 RELATED WORK

Many studies have been done on the representation of source code [4, 15] in machine learning models for predicting properties of programs such as identifier or variable names [2, 5, 10, 35], method names [3, 6, 9–11, 19, 40], class names [3], types [10, 21, 35], and descriptions [9, 19]. Allamanis et al. [2] introduced a framework that processed token sequences and abstract syntax trees of code to suggest natural identifier names and formatting conventions on a Java corpus. Allamanis et al. [3] proposed a neural probabilistic language model with manually designed features from Java projects for suggesting method names and class names. Raychev et al. [35] converted the program into dependency representation that captured relationships between program elements and trained a CRF model for predicting the name of identifiers and predicting the type annotation of variables in JavaScript dataset. Allamanis et al. [6] introduced a convolutional attention model for the code summarization task such as method name prediction with a sequence of subtokens from Java projects. Alon et al. [10] used the AST-based representation for learning properties of Java programs such as predicting variable names, predicting method names, and predicting full types. Allamanis et al. [5] constructed graphs from source code that leveraged data flow and control flow for predicting variable names and detecting variable misuses in C# projects. Hellendoorn et al. [21] proposed a RNN-based model using sequence-to-sequence type annotations for type suggestion in TypeScript and plain JavaScript code. Fernandes et al. [19] combined sequence encoders with graph neural networks that inferred relations among program elements for predicting name and description of the method in Java and C# projects. Alon et al. [11] used a bag of path-context from abstract syntax tree to learn the body of method for predicting the method name of Java projects. Alon et al. [9] later used an encoder-decoder architecture to encode the path-context as node-by-node to predict the method name of Java projects and the code caption of C# projects. Liu et al. [29] used similar method bodies to spot and refactor inconsistent method names. Wang and Su [40] embedded the symbolic and concrete execution traces of Java projects to learn program representations for method name prediction and semantics classification.

Apart from that, various deep neural embeddings and models have been also applied to different program analysis or software engineering tasks such as HAGGIS for mining idioms from source code [8], Gemini for binary code similarity detection [43]. Code Vectors for code analogies, bug fining and repair/suggestion [22]. Dynamic Program Embeddings for classifying the types of errors in programs [41]. DYPRO for recognizing loop invariants [39]. Import2Vec for learning embeddings of software libraries [38]. NeurSA for catching static bugs in code [42], and HOPPTY to detect and fix bugs in programs [18]. Researchers have also studied the language model for code completion [23, 34, 36], code suggestion [7], and code retrieval [25] task.

Moreover, Allamanis et al. [4] survey the taxonomy of probabilistic models of source code and their applications, Jiang et al. [26] conduct an empirical study on where and why machine learning-based automated recommendations for method names do work or do not work, and Chen and Monperrus [15] provide a more comprehensive survey that includes embeddings based on different granularities of source code such as tokens, functions or methods, sequences or method calls, binary code, and other for source code embeddings.

### 6 THREATS TO VALIDITY

We have performed a limited exploratory analysis on the ten most frequent methods in the dataset. Therefore, the results should be interpreted in the confinement of the limits of our experiment. The results of SVM-HANDCRAFTED depend on the features that we have extracted. Despite our best effort, it is possible that the handcrafted features can be further improved. Moreover, we only analyzed the
ten most frequent method names. Therefore, our methodology may not generalize on different methods unless we include discriminant features for them. It is possible that experiments on other methods may produce different results.

7 DISCUSSIONS AND CONCLUSION
The code2vec embeddings are highly-dimensional and are the results of training over millions of lines of code. Therefore, it is non-trivial to identify the impacts, if any, of each dimension in storing semantic or syntactic characteristics of a program. Although we really did not understand the actual meaning of each dimension of the code2vec source code embeddings, our results suggest that few handcrafted features could perform very similar to the highly-dimensional code2vec embeddings in our experiments. Compare to the handcrafted features, the information gains are more evenly distributed in the code2vec embeddings. Moreover, the code2vec embeddings are more resilient to the removal of dimensions with low information gains than the handcrafted features.

In this work, we described our preliminary study to demystify the source code embeddings through a comparison of the code2vec embeddings with the handcrafted features. Although preliminary, this work provides some insights into how the features contribute to the classification task at hand. We hope that this paper helps us to design a practical framework to objectively analyze and evaluate dimensions in the source code embeddings. Our source code to extract Top-Ten handcrafted features and train SVM models for method name classification is available at https://github.com/mdrafiqulrabin/handcrafted-embeddings.

REFERENCES
[1] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and
