user2code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code

David Azcona
Insight Centre for Data Analytics
Dublin City University
Dublin, Ireland
david.azcona@insight-centre.org

Piyush Arora
ADAPT Centre
Dublin City University
Dublin, Ireland
parora@computing.dcu.ie

I-Han Hsiao
Computing Systems & Informatics
Arizona State University
Tempe, AZ, USA
Sharon.Hsiao@asu.edu

Alan Smeaton
Insight Centre for Data Analytics
Dublin City University
Dublin, Ireland
Alan.Smeaton@dcu.ie

ABSTRACT

In this work, we propose a new methodology to profile individual students of computer science based on their programming design using a technique called embeddings. We investigate different approaches to analyze user *source code* submissions in the Python language. We compare the performances of different source code vectorization techniques to predict the correctness of a code submission. In addition, we propose a new mechanism to represent students based on their code submissions for a given set of laboratory tasks on a particular course. This way, we can make deeper recommendations for programming solutions and pathways to support student learning and progression in computer programming modules effectively at a Higher Education Institution. Recent work using Deep Learning tends to work better when more and more data is provided. However, in Learning Analytics, the number of students in a course is an unavoidable limit. Thus we cannot simply generate more data as is done in other domains such as FinTech or Social Network Analysis. Our findings indicate there is a need to learn and develop better mechanisms to extract and learn effective data features from students so as to analyze the students' progression and performance effectively.

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**; **Machine learning**;

KEYWORDS

user2code2vec, code2vec, Code Embeddings, Distributed Representations, Representation Learning for Source Code, Machine Learning, Computer Science Education

ACM Reference Format:
David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. user2code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code. In *The 9th International Learning Analytics & Knowledge Conference (LAK19), March 4–8, 2019, Tempe, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3303772.3303813

1 INTRODUCTION

Online learning tools and platforms including MOOCs provide a rich mechanism to engage and interact with educational material based on individuals' knowledge and development. Such tools also provide a mechanism to support personalised learning effectively using customized recommendations. These recommendations are developed based on users' understanding, effort and interaction with the systems by interpreting historical data from previous cohorts of students. Use of students' digital footprints and, particularly, interactions on VLE systems have been rising in last decade because of their advantage to better support individualized learning. However, developing a richer *representation* for student digital footprints effectively and efficiently is still a challenging problem which has been an area of recent interest in research, and is the focus of this work.

Learning richer distributed representations of words has shown to be quite effective for Natural Language Processing tasks [10, 14]. An embedding is a mapping from discrete objects to real numbers vectors. Such mappings constitute a dimension which may not always be meaningful or easily explainable in Machine Learning. However, the patterns of location and distances between vectors derived from embeddings may uncover numerous latent factors among the embeddings. One of the main objectives of this work is to explore the latent Learning Analytics by building high dimensional and distributional representations of student profiles and their programming codes.

In this paper we present our work on how to effectively represent and compare students' source code submitted on an internal online platform at a University, described in Section 3. We investigate different techniques to represent *students' code* (code2vec) and evaluate the performance of different representations to predict the correctness of a code solution. Furthermore, after investigating different representations of user code (code2vec), we propose a mechanism to represent students using their code submissions for given programming exercises for a course as a matrix (user2code2vec). This methodology can be used to effectively compare students in a class, cluster students who show similar behaviour and perform class-based analytics over them.

The research questions that we investigate in this work are stated as the following:

- **RQ1 (code2vec):** How can student programming submissions be encoded into vectors?
- **RQ2 (user2code2vec):** By leveraging the vectorization of code submissions for a given course, how can we represent students based on their programming work?

The main contributions of this work are as follows:

(i) Several mechanisms to represent user source code are investigated by describing the merits associated with each approach;
(ii) The performance of different source code representations for predicting the correctness of student's source code are evaluated;
(iii) A mechanism for representing student programming profiles is proposed based on their vector representations and leveraging the code submission for a given course.

Next, we discuss the outline of this paper. Section 2 describes the prior work related to work presented in this paper. Section 3 discusses the nature of the data we explored and section 4 describes different methods for code representation and vectorization that we investigated. Section 5 describes different ways for representing tokenized source code into real vectors to be used for predicting the correctness of a program using different code representations. Section 6 then describes our proposed method to represent a user using all their code submitted for the tasks or exercises in a course. In section 7 we discuss the results of our task on predicting the correctness of a program. Section 8 presents an analysis of users for two courses based on richer user representation, using all their program submission for a course. Finally, section 9 describes the main conclusion and directions for future work.

2 RELATED WORK

Neural language model-based distributed representations of text as proposed by Bengio et al. [11] and further developed by Mikolov et al. [14, 15], learn distributed word representations using Neural Network based methods which are trained over large collection of texts. These representations, commonly referred to as *embeddings*, embed an entire vocabulary into a comparatively low-dimensional vector space, where dimensions are real values. These embedding models have been shown to perform well on semantic similarity between words and on word analogies tasks [10, 14].

In the area of computer programming, predicting code properties or extracting meaningful features from vast amounts of code data has experienced tremendous progress recently [1, 3, 20]. Predicting code properties without compiling or running is used for name prediction of program entities [4], code generation [23], code completion [24] and code summarization [2]. In addition, embeddings-based techniques have been recently applied to learning effective code representations, comparing source codes and recommending approaches to students.

Mou et al. [16] recently proposed how to successfully develop program vector representations to be used in conjunction with Deep Learning models for the task of classifying computer programs. The vector representations learned used the nodes from Abstract Syntax Trees (ASTs) which are a tree representation of the abstract syntactic structure of source code [18]. The authors explored other granularity levels for representations such as characters, tokens or statements. In our work, we also explore tokens as a way to vectorize code submissions by leveraging the Python Tokenizer library.

Even more recently, Alon et al. [3] developed a code2vec neural attention network that collects AST paths and aggregates them to extract syntactic information from code snippets. Their objective was to predict semantic properties such as method names by representing snippets of code as continuous distributed vectors, also known as Code Embeddings. In our work, we build similar higher-level distributed vectors to predict the correctness of code solutions to verify patterns and meaningful information is then extracted.

Piech et al. [20] leveraged Code Embeddings to give feedback to students in MOOCs. First, they captured functional and stylistic elements of student submissions and, then, they learned how to give automatic feedback to students. This was done by developing functionality matrices at each point of the syntax tree of the submission.

In terms of giving feedback, Paaßen et al. [19] demonstrated a continuous hint approach can predict what capable students would do in solving a multi-step programming task and that the hints built using embeddings can match the edit hints that human tutors would have given. Also, Gross et al. [12] proposed feedback strategies and automatic example assignments using structured solution spaces. More recently, Proksch et al. [21] collected a dataset of rich events streams. Intead of studying artifacts after they happened, they build FeedBaG, a general-purpose interaction tracker for Visual Studio that monitors development activities and collected data from software developers.

Finally, Mou et al. [17] proposed a tree-based Convolutional Neural Network, denoted as TBCNN, using a convolution kernel designed over programs' ASTs to capture structural information. They also used this technique to classify programs based on functionality and detecting code snippets with particular patterns. In addition, developing a dataset of syntax trees can be used for recommendations as Proksch et al. [22] did for C# using solutions taken from GitHub.

In our work, code solutions from students are transformed into continuous distributed vectors, Code Embeddings, to be used as a representation of their programming submissions (code2vec). These vectors are leveraged to construct a matrix that represents each user in a comparable way (user2code2vec). Sahebi et al. [25] proposed a Tensor Factorization approach for modelling learning and predicting student performance that does not need any prior knowledge. This work outperformed state-of-the-art approaches for measuring learning and predicting performance such as Bayesian Knowledge Tracing and other tensor factorization approaches. We were inspired by this work [25] to develop a similar representation for users who learn coding at our University and we use embeddings to learn higher level representations of that information.

3 DATA

In our Higher Education Institution, students learn how to code by taking a variety of programming modules. Students develop code algorithms for problems proposed by Faculty. Many of these courses or modules are delivered through a custom Virtual Learning Environment (VLE) built for the purpose of teaching and learning computer programming. This custom VLE enables students to access course information, material and slides for each module. In addition, our system integrates an automatic grading platform where students can verify their code submissions for programming exercises. Students typically develop solutions locally for laboratory sheets for the computer programming courses. Then, they submit their programs online to the automatic grading platform which runs a number of testcases specified by the lecturer on each exercise. This provides instant feedback to students based on the suite of testcases run and ultimately tells the student whether the program is considered correct or incorrect if any of the testcases fail. This information is invaluable to their learning and such a platform is needed to verify their programs work as expected.

The computer programming grading system has been used for several years on a variety of programming courses at our University. This allowed researchers and Faculty to gather a fine-grained digital footprint of students learning programming at our University [5]. Recently, research in Learning Analytics has focused on Predictive Modelling and identifying those students having difficulties with course material, also in programming courses [9], and offering remediation, personalized feedback and interventions to students using Machine Learning techniques [6, 8]. Prior work has reported that customized notifications sent to students regarding their performance and offering resources such as further learning material, code solutions from peers in their class and university support services helped students to increase their differential performance and engagement on these programming courses [7]. However, there is a limit to this prior work where most of the models use little or no programming work as features for the learning algorithms or feedback sent to students. In this work we explore different mechanisms to represent students' code to predict its correctness and to better analyze students' progress using their interactions which can be exploited to provide effective feedback and support better recommendations.

Every time a student submits a code solution for verification, the system stores the code submission, the student identifier, the IP used on the network for the upload, the results of the testcases run with inputs and outputs, the course the submission belongs to, the exercise and the task name the student is attempting by using the submission's filename. In total, we collected more than half a million programming submissions (591,707) for 666 students from 5 Python programming courses over 3 academic years.

4 RESEARCH METHODOLOGY: CODE VECTORIZATION

In order for a learning algorithm, like a Logistic Regression Model or a Support Vector Machine, to understand text, it needs to be converted into vectors of numbers. In short, text has to be encoded as numbers to be used as input or output for Machine Learning and Deep Learning models. For that, a suite of NLP techniques are employed. In our case, code submissions or programs cannot be considered as natural language and need to be parsed and analysed in a different way. We explored the following representations of programming submissions by tokenizing the code:

(1) Code as Word Vectors
(2) Code as Token Vectors
(3) Code as Abstract Syntax Tree Vectors

In the following sections we dig deeper into vectorized representations using student code. For that, we support our narrative with Listings 1, 2, and 3. These examples are code snippets similar to the students' submissions in our programming courses.

Listing 1: Hello World Example

```
#!/usr/bin/env python
print "Hello,␣World!"
```

Listing 2: Call a Function Example

```
#!/usr/bin/env python
def say_hello():
    print("Hello,␣World!")
say_hello()
```

Listing 3: Sum of Two Variables Example

```
#!/usr/bin/env python
# read from input
a = int(raw_input()) # first
b = int(raw_input()) # second
print a + b
```

4.1 Program Code as Word Vectors

A straightforward approach to representing programs is to leverage the words from the code solutions as input to a machine learning algorithm. For each submission, we split the submission only using the space, tabular (\t) and new line (\n) characters. A typical tokenizer uses characters such as exclamation marks and other operands and operators. In a coding scenario, these characters play a key role in code submissions and we do not use them as filters for our tokenizer.

Listings 4, 5 and 6 show how such word vectors are extracted, prepared and made ready to use for some of our snippets.

Listing 4: Array of Words for Hello World Example

```
['print', '"hello,', 'world"']
```

Listing 5: Array of Words for Call a Function Example

```
['def', 'say_hello():', 'print("Hello,',
 'World!")', 'say_hello()']
```

Listing 6: Array of Words for Sum of Two Variables Example

```
['a', '=', 'int(raw_input())',
 'b', '=', 'int(raw_input())',
'print', 'a', '+', 'b']
```

These word vectors may not represent a programming submission in a very comparable way to other submissions that have, for instance, different variable names. Even though the special characters like operands carry important information regarding these code programs, splitting the words only using spaces may not give a useful representation.

4.2 Code as Token Vectors

In addition, and as we are working with the Python programming language, we leverage Python's Tokenizer[1] library for source code analysis. This module provides a lexical scanner for Python source code and is itself also implemented in Python. For instance, Listings 7, 8 show operators and delimiter tokens are clearly identified and are assigned the generic OP token category in our example snippets. We hypothesize this fine-grained tokenization such as the generalization of operands (OP), strings or names can help determine a more representable vectorization of a code submission.

Listing 7: Token Categories for Hello World Example

```
Characters      Category     Token
1,0-1,5:        NAME         'print'
1,6-1,20:       STRING       '"Hello,␣World"'
2,0-2,0:        ENDMARKER    ''
```

Listing 8: Token Categories for Call a Function Example

```
Characters      Category     Token
1,0-1,3:        NAME         'def'
1,4-1,13:       NAME         'say_hello'
1,13-1,14:      OP           '('
1,14-1,15:      OP           ')'
1,15-1,16:      OP           ':'
1,16-1,17:      NEWLINE      '\n'
2,0-2,4:        INDENT       '␣␣␣␣'
2,4-2,9:        NAME         'print'
2,9-2,10:       OP           '('
2,10-2,25:      STRING       '"Hello,␣World!"'
2,25-2,26:      OP           ')'
2,26-2,27:      NEWLINE      '\n'
3,0-3,1:        NL           '\n'
4,0-4,0:        DEDENT       ''
4,0-4,9:        NAME         'say_hello'
4,9-4,10:       OP           '('
4,10-4,11:      OP           ')'
4,11-4,12:      NEWLINE      '\n'
5,0-5,0:        ENDMARKER    ''
```

These token categories or types have an associated identifier that can also be used for vectorization, see Listing 9.

Listing 9: Token IDs for Call a Function Example

```
Characters      Category     Token
1,0-1,3:        1            'def'
1,4-1,13:       1            'say_hello'
51,13-1,14:     51           '('
51,14-1,15:     51           ')'
51,15-1,16:     51           ':'
4,16-1,17:      4            '\n'
5,0-2,4:        5            '␣␣␣␣'
1,4-2,9:        1            'print'
51,9-2,10:      51           '('
3,10-2,25:      3            '"Hello,␣World!"'
51,25-2,26:     51           ')'
4,26-2,27:      4            '\n'
54,0-3,1:       54           '\n'
6,0-4,0:        6            ''
1,0-4,9:        1            'say_hello'
51,9-4,10:      51           '('
51,10-4,11:     51           ')'
0,0-5,0:        0            ''
```

[1]For Python 3: https://docs.python.org/3/library/tokenize.html

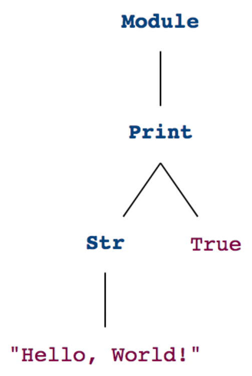


Figure 1: AST for Hello World Example

Although these tokens appear to represent code solutions more meaningfully than word vectors, information regarding the structure, design and flow of the program is still not captured and this requires a more complex representation, as we shall see in the next sub-section.

4.3 Code as Abstract Syntax Tree Vectors

In order to preserve the structure of the source code in a student submission, we also analyze the code submissions using Abstract Syntax Trees (ASTs). An AST is a tree representation of the abstract syntactic structure of source code, no matter what programming language the code is written [18]. An AST is an abstract representation as there are no details regarding the correctness of the implementation but only the structure and content. For instance, operands are implicit in the AST and IF or While expressions are denoted with a tree node. Figures 1, 2 and 3 are custom visualizations[2] after recursively traversing the nodes from the AST trees generated for our example code snippets. Green nodes represent terminal nodes or leaves. Nodes that have children are colored in blue.

[2]https://github.com/hchasestevens/show_ast

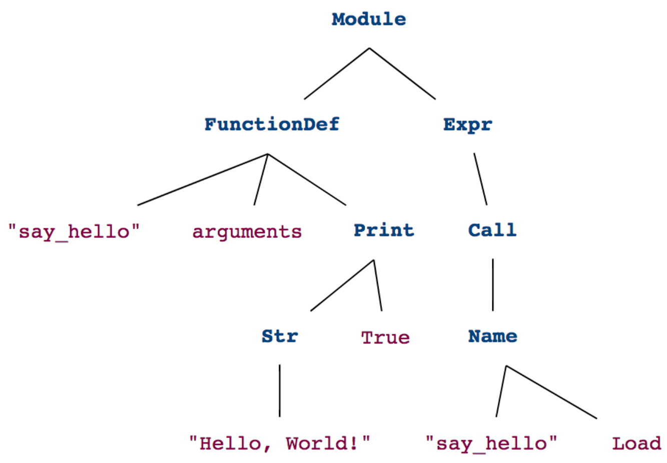


Figure 2: AST for Call a Function Example

After traversing the ASTs, nodes can be represented using their parents in a pair-wise way. See Listings 10 and 11 for two of the example snippets. The ASTs are traversed using a Breadth-first search (BFS) approach.

Listing 10: AST Pairs for Hello World Example

```
Parent Node     Child Node
'Module'        'Print'
'Print'         'Str'
'Print'         'True'
'Str'           'Hello\tWorld!'
```

Listing 11: AST Pairs for Call a Function Example

```
Parent Node     Child Node
'Module'        'FunctionDef'
'Module'        'Expr'
'FunctionDef'   'say_hello'
'FunctionDef'   'arguments'
'FunctionDef'   'Print'
'Expr'          'Call'
'Print'         'Str'
'Print'         'bool'
'Call'          'Name'
'Str'           'Hello\tWorld!'
'Name'          'say_hello'
'Name'          'Load'
```

5 EXPERIMENT: CODE2VEC

We investigate how student code submissions can be transformed into meaningful vectors as a form of representation. As mentioned earlier, computers do not understand text data and text needs to be represented and encoded into vectors of numbers as the input to a Machine Learning algorithm. For that, we use the following two approaches:

Table 1: Count Occurrence Matrix for Listings 1, 2

UNK	'('	')'	'print'	'"Hello, World!"'	'say_hello'	'def'	':'
0	1	1	1	1	0	0	0
0	3	3	1	1	2	1	1

(1) Code BOW (bag-of-words)
(2) Code Embeddings

The number of words extracted after running the tokenizer on our data are 231,659 which was fitted with 591,707 code submissions. A lot of memory is required to generate these large sparse matrices for learning code2vec and user2code2vec representations. Although our experiments are run on a GPU for faster computation, running a classification algorithm for more than half a million source code files is computationally expensive, hence we set a limit to the number of Words, Python Categories, Python Tokens Words and AST Nodes to 2,000. Overall, there are less Token Words than Words.

5.1 Code BOW (bag-of-words)

The bag-of-words (BOW) model, also called the vector space model, is a simple representation used in NLP and Information Retrieval [26]. According to this model, a text (such as a sentence or a document) is represented as a bag of its words, disregarding grammar and even word order but keeping multiplicity. In our work, we leverage the BOW model to represent code submissions by looking at either:

(a) Words
(b) Python Token Categories
(c) Python Token Words
(d) AST Nodes

The order or these items is ignored and only their frequency is stored in a large sparse matrix. This matrix can be populated using one of the following operations:

- Count: count of each word in the document.
- Frequency: frequency of each word as a ratio of words within each document.
- Binary: presence, whether or not each word is present in the document.
- TF-IDF: Text Frequency times Inverse Document Frequency (TF-IDF) scoring for each word in the document.

Table 1 shows a simple BOW example using the count of each Token Word for Listings 1 and 2 as the corpus. This BOW approach can be used for classification methods where the count, frequency, presence or TF-IDF of occurrence of each item (Word, Token Category, Token Word or AST Node) is used as a feature for training a classifier.

5.2 Code Embeddings

The BOW model provides an order-independent source code representation, only the counts of either (a) Words, (b) Python Categories, (c) Python Tokens Words or (d) AST Nodes matter. In contrast, embeddings are a different type of feature learning technique in NLP where items, words (or even phrases) from the vocabulary are

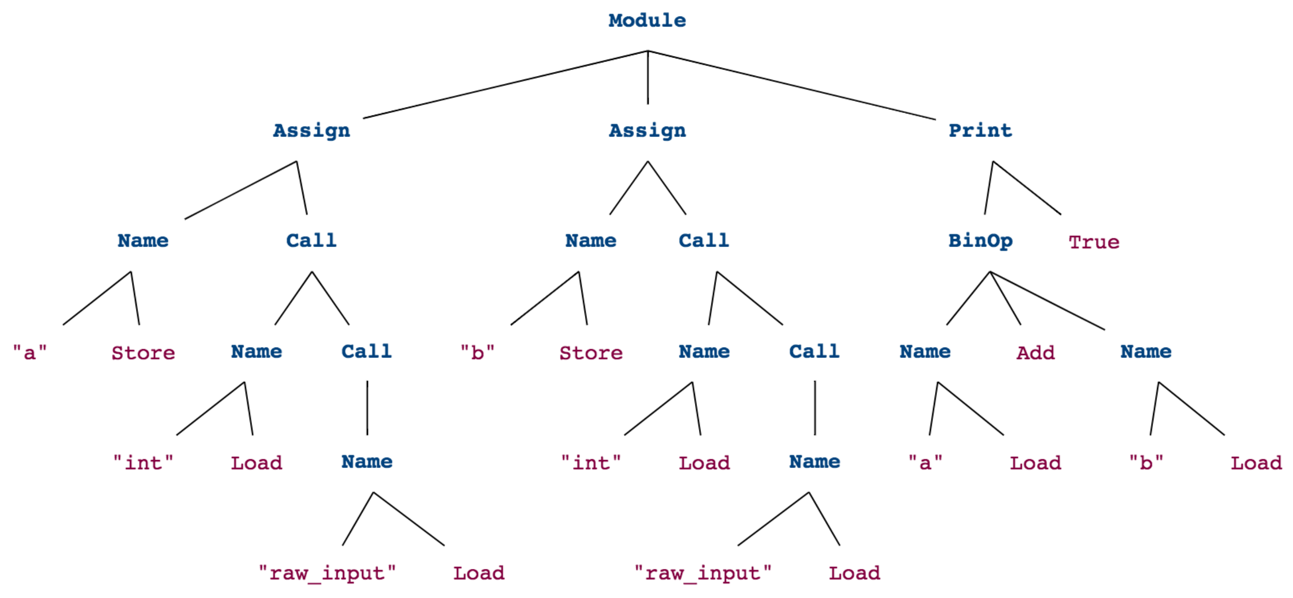


Figure 3: AST for Sum of Two Variables Example

mapped to vectors of real numbers. It involves a mathematical embedding from a space with one dimension per word to a continuous vector space with a much lower dimensionality.

We generate embeddings for code submissions of our students by transforming them into vectors in a continuous vector space. In a similar manner, we leverage the vectorization of the code solutions proposed in Section 4. We hypothesize that embeddings extract patterns using contextual information and in combination with a Neural Network can predict the correctness of the code solutions more effectively. Embeddings typically uncover really interesting properties between items or words such as neighborhoods of items or classes, relationships between items or constant vector differences, which we describe in the next section.

6 EXPERIMENT: USER2CODE2VEC

Students typically submit programming versions of the same exercise proposed in the labsheets to the grading platform until they either get it correct, or they give up. There is no limit on the number of student submissions per exercise. Then, for each user and each proposed exercise or task, the grading platform contains a set of versions. In our work, we only leverage the latest version per task for the vectorized representation of the user. In a similar manner, if we wanted to keep all versions, we would add another dimension and develop a tensor with all the submissions.

For each course and academic year, a **User Representation Matrix** is constructed for each student using the code vectors of the submissions to the proposed labsheets by the Lecturer. Having a vector representation of code submissions allows researchers to generate a higher-level representation for each student or user. This User Representation Matrix is built by vectorizing the submissions. Submission are vectorized using either:

(1) Word Tokenizer
(2) Token Word Python Tokenizer

This results in a User Representation Matrix of shape (number_tasks, MAX_LENGTH). MAX_LENGTH is the limit for each sequence that we use for padding the code submission after tokenization. MAX_LENGTH is set to 50. The User Representation Matrix for each student is flattened out as a long vector. Principal Component Analysis (PCA) [28] is leveraged as the dimensionality reduction technique to visualize the 100-dimension vectors or user embeddings into 2 dimensions. In short, a student is represented as a vector of her submissions.

7 RESULTS: CODE2VEC

In this section, the results of the code2vec technique will be discussed for both approaches: BOW and Embeddings. We train the models and learn the representations using all the Python programs submitted by students in our University over a number of years, and we use these representations to predict the correctness of a student's code.

7.1 Code BOW (bag-of-words)

First, we build four tokenizers constructed and fitted with the code submissions using either (a) Words, (b) Python Categories, (c) Python Token Words or (d) AST Nodes respectively. The dictionary of items and their counts are shown in Table 2. It is interesting to see the differences between the top occurrences for each tokenization, where Token Words are a generalization of Words, Token Categories are a generalization of Token Words and the AST nodes are at an abstract level which contain items regarding the structure of the code submission.

This way, we can construct matrices where each row is a code submission and we count the number of occurrences for each (a)

Table 2: Top-5 Words, Token Categories, Token Words & AST Nodes

Word	Occurrences	Token Category	Occurrences	Token Word	Occurrences	AST Nodes	Occurrences
‘=’	2,440,154	51: ‘OP’	20,368,593	‘)’	3,556,931	‘Name’	10,005,368
‘i’	910,221	1: ‘NAME’	18,075,194	‘(’	3,556,907	‘Load’	9,607,682
‘+’	575,607	4: ‘NEWLINE’	5,886,806	‘=’	2,581,991	‘Store’	2,665,169
‘if’	552,539	2: ‘NUMBER’	2,317,086	‘:’	2,248,901	‘Call’	2,205,672
‘def’	522,536	54: ‘NL’	1,996,531	‘.’	2,011,442	‘Assign’	2,186,523

Word, (b) Token Category, (c) Token Word and (d) AST Node. Figure 4a shows details on the performance of these model combinations (a), (b), (c) and (d) just using the count of items. In addition, we look on the (a) Words (as they work better) and perform a similar analysis looking at the count, presence (binary), frequency and TF-IDF of the Words instead of the pure count only. Figure 4b does not show a meaningful difference between them except that the frequency model works slightly worse than the others. These models are trained using a Naive Bayes classification algorithm [13] holding out 20% of the data as the testset. The models are trained using around half a million code submissions (less for the Tokens or AST Trees as some code submissions could not be tokenized using the Python Tokenizer library or an AST could not be extracted when the programs are incorrectly constructed). The classes for this classification problem are well balanced. For instance, for the model that uses the Words 194,451 submissions were correct and 296,369 were incorrect based on the output of the grading platform. That is the target of our predictions for training the models.

Interestingly, the least generalized model that uses the Words instead of Tokens or AST Nodes is the one that performs slightly better than the rest using BOW. The less generalized the model is, the better it performs, using Words perform better than Tokens and Tokens perform better than AST Nodes.

7.2 Code Embeddings

In a similar manner and in order to feed vectors to a Neural Network we vectorize our code submissions by tokenizing for (a) Words, (b) Python Categories, (c) Python Token Words and (d) AST Nodes. In addition, as a pre-processing step, we pad our sequences up to our limit of 50 words, tokens or nodes. A simple model is developed using an Embeddings layer, flattening the output of that layer on the next one and condensing it on the final one using a softmax function. The embeddings are the representation extracted after learning from the Embeddings layer and contain 100 dimensions. This forces the Neural Network to learn patterns as we are inputting 2,000 words that will be 2,000 dimensions using one-shot encoding.

The performance of the models using (a) Words and (b) Token Words is shown Figure 5. These models are trained using Cross Validation with 20% of the dataset as the holdout set. Utilizing Neural Networks with an Embeddings layer allows us to learn better patterns and representations of the code solutions submitted to the grading platform. The models perform better than the baseline BOW and the Word Tokens are better able to distinguish between correct and incorrect programs. We expect that incorporating the structure of the program using ASTs will create a richer model.

Table 3 shows the results from the different BOW and Embeddings models in a comparable way.

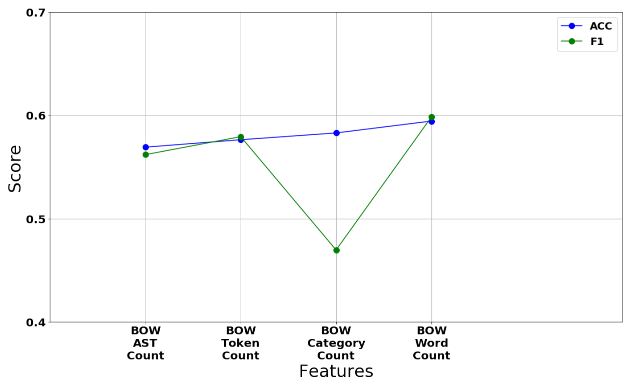
After the models are trained using the code submissions with the correct or incorrect target for each, the learned embeddings can be extracted. Figure 6a shows the embeddings of the top 20 most common words. It is interesting to note how operands are clustered together as are numbers. This confirms that the network is learning efficient representations. Similarly, we can explore the top 20 most common tokens in Figure 6b.
These vectors contain really interesting properties similar to word embeddings. Table 4 shows some cosine similarities between pairs of words that are very close to other pairs. Neighbors of these embeddings can also be checked out, and numbers can be found besides other numbers in String format, but in general, our learned embeddings have noise such as variable names and Strings that prevents us to from seeing other relationships as would be found in word2vec [14].

8 RESULTS: USER2CODE2VEC

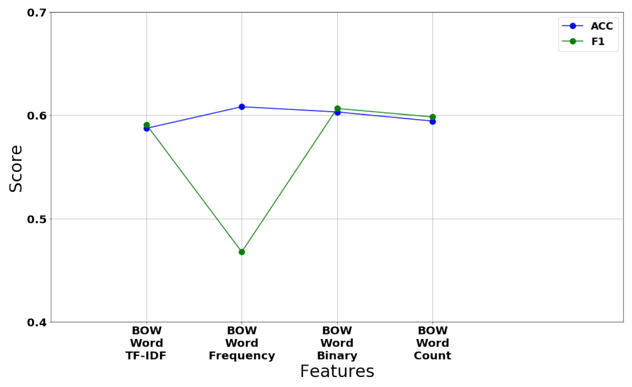
user2code2vec has been performed on students of two of the computer programming courses at Dublin City University for a full academic year. Course details can be found in Table 5. User Representation Matrices were constructed using the code submissions for each student. Then, we flattened them out to input them to a Deep Learning Network similar to code2vec with an Embeddings layer that learns representations of users in a continuous space with a reduced dimensionality. User Embeddings are given 100 dimensions. The input data are very large and sparse vectors with the indexes of the vocabularies for the code submissions.

In CS1 during 2016/17, these student vectors have 13,800 dimensions as there are 276 tasks to be completed in the course and the limit of the submission sequences is 50. Figure 7a shows the input to the Neural Network and Figure 7b shows how difficult it is to distinguish among a few hundred students with such a large sparse matrix of code submissions. Unfortunately, we cannot add more data as there were no more students on that cohort, unlike other domains which allow downloading of more tweets or crawling more websites when this situation happens. The vectors are transformed to 2 dimensions using PCA. The variance retained is very low (between 2% and 6%). Each dot on the graphs represents a student based on the projection of their student vector. The colour used represent the average grade of the exams that student took that year on that course.

Deep Learning is known to work well when more data is provided and it is expected that more data results in improved performance across most domains. Due to the curse of dimensionality, in



(a) Words vs. Token Categories vs. Token Words vs. AST Nodes Using Count



(b) Words Using Count, Binary, Frequency & TF-IDF

Figure 4: Performance of code2vec using BOW (bag-of-words).

Table 3: Performance of the Models Using BOW and Embeddings

Model	Accuracy	F1 Score
Naive Bayes using BOW & Words	59.44%	59.84%
Naive Bayes using BOW & Category Tokens	58.29%	46.95%
Naive Bayes using BOW & Word Tokens	57.63%	57.93%
Naive Bayes using BOW & AST Nodes	56.91%	56.18%
Neural Network using Embeddings & Words	73.25%	64.11%
Neural Network using Embeddings & Category Tokens	74.93%	19.64%
Neural Network using Embeddings & Word Tokens	74.93%	67.18%

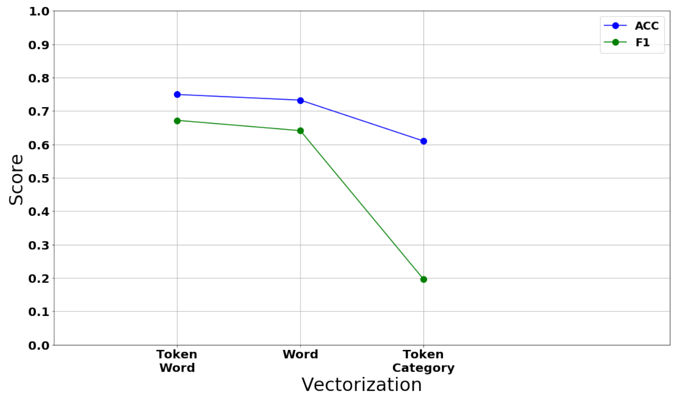


Figure 5: Performance of code2vec using Embeddings

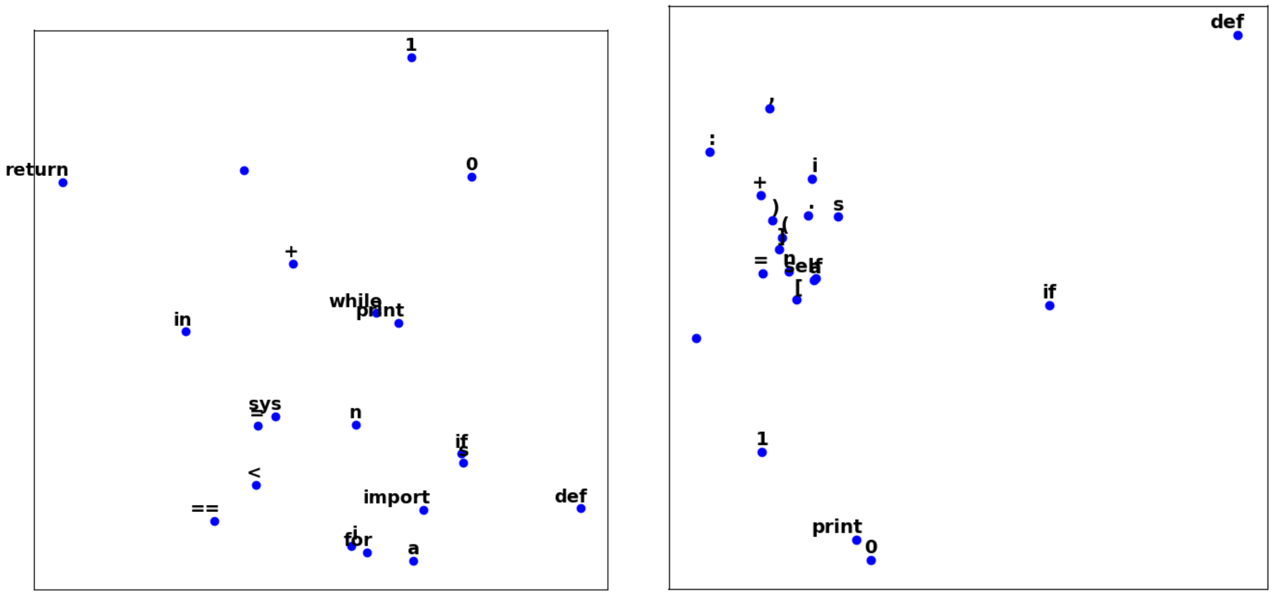
a high-dimensional feature space with each feature having a range of possible values, typically an enormous amount of training data is required to ensure that there are several samples with each combination of values. A typical rule of thumb is that there should be at least 5 training examples for each dimension in the representation [27]. However, the constraint for Learning Analytics in Education in VLEs, but not in MOOCs, is the number of students enrolled in a course. Thus instead of representing each student using the concatenation of all their submission made in a course, it would be better to identify important features from each submission and concatenate key features across the code submission. In short, keep the number of features low to effectively learn from constrained data. These user2code2vec representations can then be used to identify student neighbours for programming recommendations.

Table 4: Cosine Distance Between Word Vectors

$Token_i$	$Token_j$	Cosine Distance
'('	')'	0.9136
'<'	'>'	0.9241
'['	']'	0.9792
'if'	'elif'	0.9732
'}'	']'	0.8857
'+'	'-'	0.8846

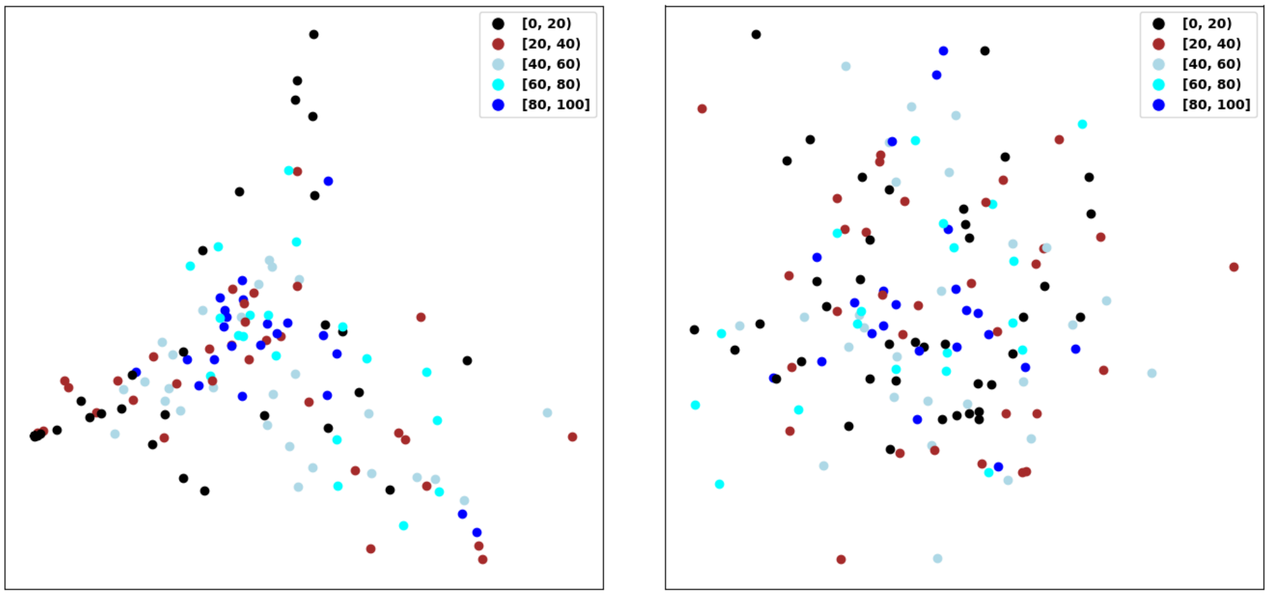
Table 5: Courses Analysed on user2code2vec

Course	Year	Code Submissions	Tasks	Students
CS1	2016/17	68,313	276	126
CS2	2016/17	74,065	132	140



(a) Embeddings for the Top 20 Most Common Words (b) Embeddings for the Top 20 Most Common Token Words

Figure 6: Embeddings for the Top Words & Token Words Projected from 100D to 2D Using PCA



(a) User Raw Representations Using Word Tokens (b) User Learned Embeddings Using Word Tokens

Figure 7: user2code2vec applied to CS1 course during 2016/17 academic year.

9 CONCLUSIONS AND FUTURE WORK

Our code2vec implementation and results confirm the power of code embeddings and the latent learning analytics properties in student code submissions, as compared to using bag-of-words based representations of source code. We are now working on combining Tokens and ASTs for creating a richer model that has code details including structure and context. In addition, we are exploring Concrete Syntax Trees which are parse trees, typically built by a parser during the source code translation and compiling process, adding subsequent processing to ASTs such as contextual information.

User2Code2vec is a novel technique to represent students in a high-dimensional space using distributional representations of student profiles and their programming code. Other techniques such as Matrix Factorization can be used to find students with similar coding patterns. In addition, the User Representation Matrix can be built as as a Tensor with a new dimension using all the submissions instead of the last one or one at random. That might give us a better representation of their learning and progression that we could use.

Embeddings have proven to successfully identify hidden or latent patterns for code submissions and user representations. Measuring the quality of these vectors is not straightforward. Several factors influence the quality of the vectors such as the amount and quality of the training data, the size of the vectors and the learning algorithm used. The quality of these vectors is crucial for the representations but trying out different hyper-parameters takes a lot of computation and time. Pre-trained vectors with a large corpus is the standard in other domains like word vectors using Google's News dataset and this community should make the effort to develop good Code Embeddings that can be used to learn higher level abstracts like User Embeddings.

Embeddings for source code and student code representations is still at an early stage of development but has potential to change how we understand learning to program, recommend code and peer learning of programming using higher level abstractions.

In our future work we plan to focus on two main aspects, to learn better distributional semantics using abstract trees to capture syntactic structure effectively following recent work proposed in [3, 20] and to use the recommendation learned using the user2code2vec representation proposed in this work and to evaluate how this representation helps students to improve their learning. [3]

ACKNOWLEDGEMENTS

This research was supported by the Irish Research Council in association with the National Forum for the Enhancement of Teaching and Learning in Ireland under project number GOIPG/2015/3497, by Science Foundation Ireland under grant numbers 12/RC/2289 and 13/RC/2106, and by Fulbright Ireland. The authors are indebted to Dr. Stephen Blott who developed the programming grading platform.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. *arXiv preprint arXiv:1803.09473* (2018).

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *arXiv preprint arXiv:1803.09544* (2018).

[5] David Azcona, Owen Corrigan, Philip Scanlon, and Alan Smeaton. 2017. Innovative Learning Analytics Research at a data-driven HEI. In *Proceedings of the 3rd International Conference on Higher Education Advances*. Editorial Universitat Politècnica de València, Valencia, Spain, 435–443.

[6] David Azcona, I-Han Hsiao, and Alan F Smeaton. 2018. Detecting Students-In-Need in Programming Classes with Multimodal Learning Analytics. *International Journal of Artificial Intelligence in Education (ijAIED)* (2018).

[7] David Azcona, I-Han Hsiao, and Alan F Smeaton. 2018. An Exploratory Study on Student Engagement with Adaptive Notifications in Programming Courses. In *European Conference on Technology Enhanced Learning*. Springer, NY, USA, 644–647.

[8] David Azcona, I-Han Hsiao, and Alan F Smeaton. 2018. PredictCS: Personalizing Programming Learning by Leveraging Learning Analytics. *Companion Proceedings 8th International Conference on Learning Analytics & Knowledge (LAK18)* (2018).

[9] David Azcona and Alan F Smeaton. 2017. Targeting At-risk Students Using Engagement and Effort Predictors in an Introductory Computer Programming Course. In *European Conference on Technology Enhanced Learning (EC-TEL'17)*. Springer, NY, USA, 361–366.

[10] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. 2014. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 238–247.

[11] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research* 3, Feb (2003), 1137–1155.

[12] Sebastian Gross, Bassam Mokbel, Benjamin Paaßen, Barbara Hammer, and Niels Pinkwart. 2014. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10* 9, 3 (2014), 248–280.

[13] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. Text classification and naive bayes. *Introduction to information retrieval* 1, 6 (2008).

[14] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[15] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[16] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. 2014. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358* (2014).

[17] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.. In *AAAI*, Vol. 2. 4.

[18] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[19] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2017. The Continuous Hint Factory-Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *arXiv preprint arXiv:1708.06564* (2017).

[20] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*. JMLR. org, 1093–1102.

[21] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched event streams: a general dataset for empirical studies on in-IDE activities of software developers. In *Proceedings of the International Conference on Mining Software Repositories*.

[22] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2016. A dataset of simplified syntax trees for C. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 476–479.

[23] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[24] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[25] Shaghayegh Sahebi, Yu-Ru Lin, and Peter Brusilovsky. 2016. Tensor factorization for student modeling and performance prediction in unstructured domain. In *Proceedings of the 9th International Conference on Educational Data Mining*. IEDMS, 502–506.

[26] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[27] Sergios Theodoridis, Konstantinos Koutroumbas, et al. 2008. Pattern recognition. *IEEE Transactions on Neural Networks* 19, 2 (2008), 376.

[28] Michael E Tipping and Christopher M Bishop. 1999. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 61, 3 (1999), 611–622.

[3] Code developed in this work has been made publicly available as a repository on Github at https://github.com/dazcona/user2code2vec where further details such as PCA graphs for the learned embeddings can be found.