

Convolutional Neural Network applied to Code Assignment Grading

Fábio Rezende de Souza, Francisco de Assis Zampiroli^a, Guiou Kobayashi

Centro de Matemática, Computação e Cognição, Universidade Federal do ABC (UFABC)

9.210-580 – Santo André – SP – Brazil

{fabio.rezende, fzampiroli, guiou.kobayashi}@ufabc.edu.br

Keywords: Artificial Intelligence, Automatic Grading, Text Classification, Deep Learning.

Abstract: Thousands of students have their assignments evaluated by their teachers every day around the world while developing their studies in any branch of science. A fair evaluation of their schoolwork is a very challenging task. Here we present a method for validating the grades attributed by professors to students programming exercises in an undergraduate introductory course in computer programming. We collected 938 final exam exercises in Java Language developed during this course, evaluated by different professors, and trained a convolutional neural network over those assignments. First, we submit their codes to a cleaning process (by removing comments and anonymizing variables). Next, we generated an embedding representation of each source code produced by students. Finally, this representation is taken as the input of the neural network which classifies each label (corresponding to the possible grades A, B, C, D or F). An independent neural network is trained with source code solutions corresponding to each assignment. We obtained an average accuracy of 74.9% in a 10-fold cross validation for each grade. We believe that this method can be used to validate the grading process made by professors in order to detect errors that might happen during this process.


1 INTRODUCTION

Approximately 2000 freshmen students each year enrolls at our Federal University of ABC (UFABC) in Brazil, in one of the following interdisciplinary Bachelor Degrees: Bachelor in Science and Technology and Bachelor in Science and Humanities. Both Interdisciplinary Bachelor Degrees have a 3-year duration divided on three quarter periods per year (Q1, Q2 and Q3). Each student has the possibility of enrolling in a specific major such as Mathematics and Physics (which can be concluded with an additional one year coursework) or Engineering (which can be concluded with two additional years). UFABC offers more than 20 options of major programs for students to enroll after the Interdisciplinary Bachelor degree. All of these students are required to take a ILP course, ideally at the third quarter of their freshmen year. Considering that the Bachelor in Science and Technology has students with multiple academic interests, from Biology to Computer Science, their level of interest on the ILP course varies greatly. Because of this the ILP course has an average failure rate of 32%, see (Zampiroli et al., 2018).

On its blended-learning modality, the ILP Course

accepts enrollment of about 180 students at each quarter period each year. For each offering 4 to 6 professors are allocated to teach those classes, which can vary depending on their workload and the number of students currently enrolled on their classes. About 5 Teaching Assistants are also required to help students on their class assignments. The evaluation of student's performance in class is measured associating grades: A (Outstanding), B, C, D, or F (Fail).

This course has a 12-week length, with a class workload of 60 hours/week (for both its face-to-face and blended-learning modalities). However, at the blended learning modality, only four presential meetings are required: an introductory class, a midterm exam, a class project submission and a final exam. Every exam contains 3 questions, and different professors are required to grade the solutions provided by their students in their respective classes. The midterm exam is a handwritten assignment: each student is required to manually write 3 computer coding programs. The main goal of the midterm exam is to evaluate the student's programming logic skills: at this point, code syntax correctness on their answers are not required. The student has the possibility of write their programs using a pseudocode language for portuguese-speaking students, called *Portugol Studio* (univali.br/portugolstudio). The following

^a  <https://orcid.org/0000-0002-7707-1793>

assignments (class projects and final exams) are required to be in Java language (at this point, *Portugol*-based submissions are accepted with a penalty of maximum B grade).

The following concepts are covered at the ILP course: sequential instructions, conditional statements, loops, vectors, matrices, and modules. The first three concepts are covered at the midterm exam (at the 5th week). The class project (9th week) and final exam (10th week) covers all class concepts. The classes syllabus and materials, which are slides, videos, multiple-choice exercises (with automatic correction) and programming exercises (graded by teaching assistants), are made available to the students on a virtual online environment.

The main objective of this paper is to create an automatic grading approach of programming code assignments, in order to assist teachers on the grading process. This approach is different from automatic code correction programs that yield only two results: pass (correct code) or fail. We believe that adding an automatic evaluation to suggest a grade for each assignment, apart from manual correction, will help to minimize eventual misgrading problems (such as the impact of each personal characteristics for grading process). To achieve this objective, we will focus on the face-to-face exams - specifically, the programming code exercises at final exams.

2 RELATED WORKS

(Singh et al., 2013) proposes a method for generating automatic feedback for introductory programming assignments. This method is based on a simple language definition for describing different error messages, which consists of possible suggestions for correcting common mistakes the students might make.

(Gulwani et al., 2014) proposes an extension for programming languages on which professors are able to define an algorithm to look for certain patterns which might occur during a program development. It uses 2316 correct scripts for 3 proposed programming exercises, identifying 16 different strategies using the proposed language.

(Bhatia and Singh, 2016) presents a method for providing feedback on syntax programming errors on 14000 introductory program assignments submitted by students. This approach is based on Recurrent Neural Networks (RNNs) for modelling sequences of syntactically corrected tokens. The authors mention that previously proposed approaches generated Abstract Syntax Trees (AST) for each program, which is not possible for source codes containing syntactical

errors. Their approach achieved 31,69% accuracy on detecting syntax errors for 5 different programming assignments.

The approach proposed by (Gulwani et al., 2018) consists on a Matching Algorithm for fixing programming errors by comparing them to correct solutions provided by students. Their article also provides an extensive review of previously proposed methods on literature for fixing programming errors.

The method we are about to present does not consist on detecting syntax errors or suggesting alternatives for automatic code correction - instead, we propose a method to automatically evaluate the quality of programming assignments based on their underlying semantic structure, in order to help professors on the challenging task of grading programming assignments provided by students. Our work is based on the method proposed by (Kim, 2014), where a Convolutional Neural Network (CNN) with a convolutional layer built on word-embedding is applied on sentence classification tasks in natural language. The authors proposed many variations of their approach - some of them containing previously trained word-embedding following the method proposed by Mikolov (Mikolov et al., 2013) over 100 billion words of the Google News corpus, publicly available at code.google.com/p/word2vec.

3 METHOD

Here, we present how we collected our data and implemented our supervised machine learning method to grade student's assignments, using a dataset of exercises previously graded by different professors.

3.1 Data Collection

Our dataset consists of programming exercises presented in the final exams of the ILP courses, all of them with the same difficulty level. Each professor corrected and graded all students submissions for a given class and multiple classes were held at the same time at each quarter period term. The final exam have 3 questions with different difficulty levels - easy, medium and hard, respectively. Generally the easy-level question is a question regarding vector structures, medium-level questions covers matrices, and hard-level questions covers module structures. In order to solve a question regarding modules, a student must have been able to acquire sufficient programming abilities and expertise to solve the easy and medium-level questions (regarding more simple data structures).

A total of 938 questions were collected (corresponding to 2017.1, 2018.1, 2018.2 e 2018.3 quarter period terms). Figure 1 shows the number of graded submissions for each question (labeled as e2q1, e2q2 and e2q3, respectively) with grades consisting of A (outstanding), B (good), C (regular), D (minimum) or F (fail). We considered only code on Java programming language.

term	question	A	B	C	D	F	Grand Total
2017q1	e2q1	59	4	12	15	11	101
	e2q2	43	3	9	16	21	92
	e2q3	18	8	16	28	18	88
2017q1 Total		120	15	37	59	50	281
2018q1	e2q1	73	8	4	7	17	109
	e2q2	48	9	7	21	21	106
	e2q3	50	2	3	6	35	96
2018q1 Total		171	19	14	34	73	311
2018q2	e2q1	17	17	17	5	8	64
	e2q2	12	4	7	12	18	53
	e2q3	10	2	5	8	18	43
2018q2 Total		39	23	29	25	44	160
2018q3	e2q1	28	0	30	4	5	67
	e2q2	29	4	1	15	16	65
	e2q3	20	7	3	7	17	54
2018q3 Total		77	11	34	26	38	186
Grand Total		407	68	114	144	205	938

Figure 1: Data Collection: at four academic terms (2017q1, 2018q1, 2018q2 and 2018q3), we reunited 3 questions for each term - e2q1, e2q2 and e2q3. Every question has a grade A, B, C, D and F (fail). A total of 938 questions were collected.

Besides that, each question has in average 4 different versions with minor variations on given exercises values (with no impact on the solution logic or difficulty level). We used a web framework for generating exams called webMCTest (available at vision.ufabc.edu.br:8000) (Zampirolli et al., 2016; Zampirolli et al., 2019). This systems shuffles those possible question's variations, and selects one instance for each student. Considering that we have a 3-questions exam, each one having 4 possible variations, we have 64 different versions of the same exam. Although the webMCTest also supports automatic correction for multiple-choice questions, at this point we consider that written solutions allows us to detect multiple levels of code correctness between a hard *Pass* or *Fail* grading.

Every student submission is automatically compiled by the system (if no compilation error occurs) and renamed as `StudentName.StudentLastName.QuestionNumber.java` before becoming available for their professors. Those questions are manually corrected by the professors and after grading the student's exams they

submit their feedback to the webMCTest system, with automatically sends e-mails to the students reporting their results to respective exams. More details regarding this process are available on (Zampirolli et al., 2018).

Following, there is an example of a hard-level question covering programming modules:

Consider a matrix `matGRADE` of 150 rows and 4 columns, where each row represents a student and each column represents the concepts of the evaluations Exam1, Activities, Project, and Exam2. This matrix stores in its each element, grades A, B, C, D or F.

Create a `GenerateMat` function (to be available to call from the main program), which fills the `matGRADE` matrix with randomly generated grades.

For each of the following items, you must write a function and make their respective call in the main program.

- Write the `GenerateAverage` function to fill a vector with real numbers in which each element of the vector will be the average points of a student calculated from the grades in their respective row of the `matGRADE` matrix. To calculate the average points of each student, consider $A = 4.0$, $B = 3.0$, $C = 2.0$, $D = 1.0$ and $F = 0.0$. Consider also the following weights: Exam1 = 30%, Activities = 10%, Project = 15% and Exam2 = 45%. The average points of each student will be between 0.0 and 4.0. Example: If a row of the Matrix has A, A, B, D, the average points will be $(4 * 30) + (4 * 10) + (3 * 15) + (1 * 45) / 100 = 2.5$. In this example, `FINAL_GRADE` will be B, as follows.
 - Write the `FinalGrade` function that should receive as parameter the vector generated in item (1) and print on the screen the corresponding grade of each student considering the following rules:
 - if `VALUE < 0.8`, `FINAL_GRADE = F`,
 - otherwise,
 - if `VALUE < 1.5`, `FINAL_GRADE = D`,
 - otherwise,
 - if `VALUE < 2.5`, `FINAL_GRADE = C`,
 - otherwise,
 - if `VALUE < 3.5`, `FINAL_GRADE = B`,
 - otherwise,
 - `FINAL_GRADE = A`.
-

Finally, for evaluating a student’s submission, the professor has access to a footnote containing observations regarding possible source of errors based on a previous analysis of each question. This information is made available for the students alongside the professor’s feedback on their work on every specific question. More information on the evaluation and grading process can be found at (Zampirolli et al., 2018).

Following you can find an example of those footnote observations:

Dear Student,
 You can find on Table 1 the possible errors on this question:

Table 1: Table describing sources of possible student’s errors, and the penalties associated with each error on their final grade. Consider grade boundaries as A = 4, B = 3, C = 2, D = 1 and F = 0. Finally, the rightmost column shows a more detailed description for every error.

Error	Penalty	Error Description
1	-1	implemented GenerateMat method to create a matrix
2	-2	implemented GenerateAverage method to calculate average points
3	-1	implemented method FinalGrade
4	-2	code does not compile correctly
5	-1	developed on Portugol Studio (in portuguese or pseudocode)
6	-4	incomplete or unorganized code

As some examples, the highest grade associated with each error present on a student’s submission is defined below:

- A – No major error found;
- B – *Portugol* Implementation;
- C – The program did not compile;
- F – Incomplete Code.

3.2 Classification Approach

In order to classify the level of semantic correctness of programming language code, we use a language-independent approach based on Distributional Semantics (Lenci, 2018), in which we represent language semantics considering as the only information

available the latent distribution of elements in the language. In our work, to represent elements of a programming language, we create an vector embedding the representation of each element of this programming language based only on its vocabulary’s keywords and their distribution over each source code. We will use this representation as an input to a Convolutional Neural Network (CNN) in order to distinguish between different levels of skills of code structure development.

For doing this, we propose a three-step method. First, we developed a script to read the source code files as plain text, removing code comments and creating default names for class and variable definitions in order to reduce vocabulary variations and keep only the programming language keywords, variable values, digits and symbols (such as brackets and parentheses).

In the second step, we train a Skip-Gram, as proposed by Mikolov (Mikolov et al., 2013), to create ‘code-embedding’, where each keyword is represented by a vector embedding of variable size, containing latent probabilities of the possible contexts in which it appears on the source codes.

A Skip-Gram is a neural network containing only three layers: input, output and a hidden layer. The network input consists in a one-hot encoding vector of V dimensions of a w word, and its output consists on the prediction of C context words around it. Here, we consider as ‘words’ the keywords or elements (such as brackets or mathematical symbols) inside the vocabulary of the programming language.

The full input is the matrix $A(V \times N)$, where V is the dimension of the one-hot encoded representation of each word and N is the reduced embedded space of the hidden layer. The output is $B(N \times V)$ matrix containing embedding representations for each word based on the reconstructed probability.

For each w input word, we try to maximize the occurrence of other words occurring at the same sentence of w . See the Figure 2 a Skip-Gram illustration adapted to this paper. For details on this figure, see (Kim, 2014).

The network output consists of N -dimensional vectors for each vocabulary word.

The third step consists of an adaptation of Kim (Kim, 2014) approach for text classification in natural language, (see Figure 3). This approach consists of a CNN receiving the matrix $B(N \times V)$, where we apply a convolutional filter c applied for a h word-window to produce new features. For each word x , we have a feature map $c[]$, followed by a max-pooling operation, as proposed by (Collobert et al., 2011), and a dropout layer to apply constraints on the weight vec-

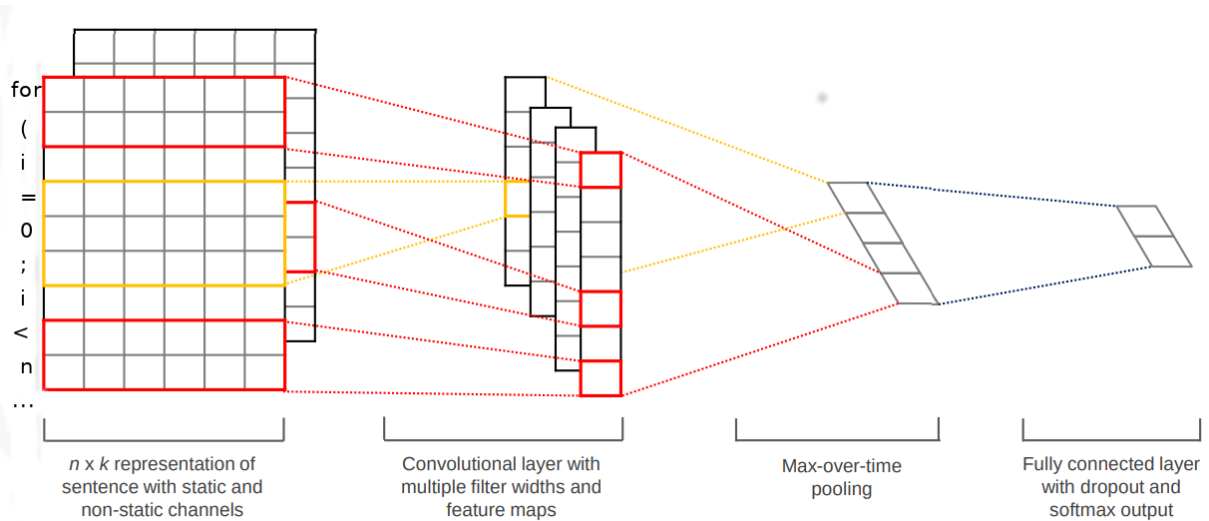


Figure 2: An overview of our proposed adaptation of the method by (Kim, 2014), using vector-embedding representations of keywords in a programming language, instead of natural language vocabulary. Our output consists of a 5–dimension vector where the probabilities for each class is disposed.

tors (Hinton et al., 2012). The output consists of a softmax layer displaying the probability distribution over each label - which corresponds to a grade associated to the code, varying from *A* (outstanding) to *F* (fail). The grade having the highest associated probability, between the 5 possible grades, is taken as the chosen grade by the neural network for each assignment.

4 RESULTS AND DISCUSSION

We performed a total of 12 different experiments (see Table 2), each one consisting of a independent trained model over student’s solutions to the final exam programming questions on 4 different academic quarter period terms.

The displayed results were achieved using the following model configurations: input vector embeddings of 50 dimensions, convolutional word windows $h = [2, 3, 8]$, dropout rate of 0.5, batch–size of 64 and 25 training epochs.

To evaluate our method, we performed a 10–fold cross validation for each experiment. For every iteration of the 10–fold cross validation, we calculated our method’s accuracy by comparing the trained model predictions for their test set data (10–fold splits \times 12 training models, in a total of 120 independent test sets) with the test sets’ expected results (hidden from the network model in which they were used as test sets). For a total of 1020 test samples, we achieved the results displayed at Table 3 and Figure 4.

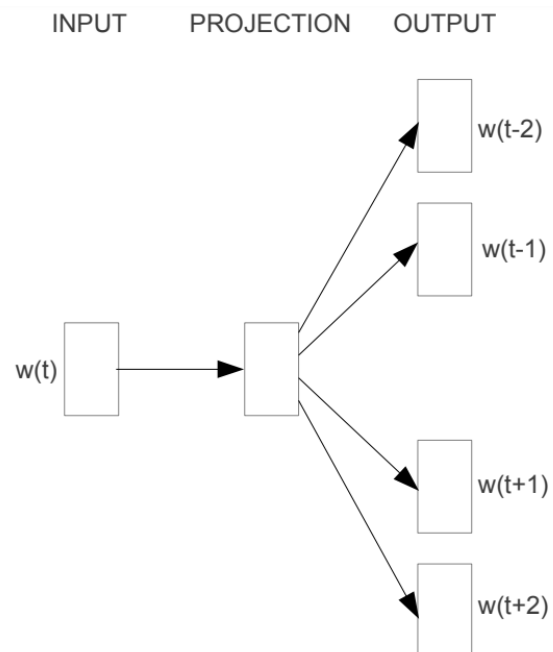


Figure 3: Skip-Gram method, as proposed by (Mikolov et al., 2013), where we take a word from a vocabulary to maximize probability of occurring words surrounding it on a same sentence.

A detailed information of the performance of each corresponding grade (across every training iteration) can be found at the Table 3.

As it can be seen at the Confusion Matrix at Figure 4, each class had a accuracy (across different trained

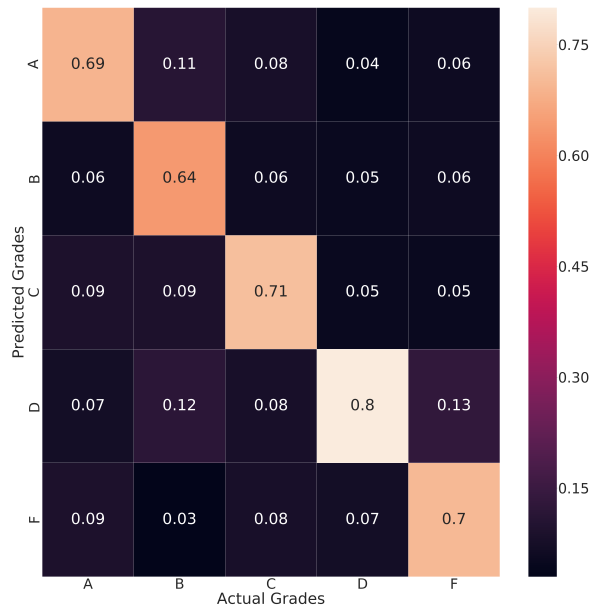


Figure 4: Confusion Matrix of the test sets for every training iteration of the 10-fold validation (a model for each one of the 3 exam questions, in the 4 different academic terms). The colors in black represent values close to 0%. It should be noted that the classification with the best result was with the D grade, with 80% (closer to white color).

Table 2: Accuracy of our models for each question, evaluated using a 10-fold cross validation method.

Period	Question	Validation Accuracy (%)
2017.1	1	82.2
	2	68.3
	3	64.1
2018.1	1	81.1
	2	76.2
	3	83.8
2018.2	1	72.6
	2	73.6
	3	72.5
2018.3	1	69.5
	2	73.2
	3	81.9
Avg. (%)		74.9

networks, each one of them independently trained for a single question) varying from 69% (A grade) to 80% (D grade), with the largest degree of confusion being 13% (between D and F grades), which is expected considering the subjective nature of code correction and also the fact that both are the lowest possible grades. More than 10% of confusion is also found be-

tween A and B grades (11%), which is also expected for the similar reasons (the best possible evaluations). A unexpected confusion happened between D and B grades (13%). Confusion between other classes did not went higher than 10%, which can be seen as a proof that this method can perform a good analysis of code quality in an academic environment.

Discussions

The proposed method does not intend to replace the evaluation process performed by the professors which is a very important step of the teaching process. Our method intends to mitigate possible inconsistencies that might happen during this process, taken previously made corrections as a reference (which were also performed by the professors).

Before making each assignment grade available for students, this method could be used by professors to validate each grade given, in order to find clues of inconsistent corrections. We believe that this is a real possibility since each question on the final exam covers one specific taught concept, and we traditionally repeat the same topics for the questions having similar difficulty levels across different academic terms.

Considering that there are 5 possible classes (vary-

Table 3: Performance of the method.

Grade	Precision	Recall	F1-score	Support
A	0.69	0.68	0.68	182
B	0.64	0.74	0.69	170
C	0.71	0.74	0.73	204
D	0.80	0.71	0.75	273
F	0.70	0.72	0.71	191
avg / total	0.72	0.72	0.72	1020

ing between A, B, C, D, and F grades), the results presented in this paper can be considered excellent. Usually, when we, as professors, are in duty of evaluating a student submission to a question, we tend to divide opinions when the student’s work did not achieve polarized results (being perfectly correct or extremely wrong), leading to a subjectivity in the evaluation process. For example: while some professors consider that they should assign a D for a poor solution arguing that a minimum skill was demonstrated by the student, other professors would consider it as a complete failure assigning a F grade for the same proposed solution. In our dataset, consisting of blended-learning modality with unified course and exams, this subjectivity between D and F grades resulted in a variation of 20%. In (Zampirolli et al., 2018), a variation of up to 40% was presented in the evaluation of several classes in the face-to-face modality when there was no unified process. Analysing the Confusion Matrix in Figure 4, the difference between these two grades in our method (where the method had classified as F but the teacher attributed the D grade) was 13% .

5 CONCLUSIONS

In this article, we present a method for helping professors evaluating student code submissions in a undergraduate introductory programming language course (ILP). We believe that our approach could be incorporated on Massive Open Online Courses (MOOCs) since it offers a deeper evaluation of source code instead of a binary *pass* or *fail* feedback as it happens on traditional online programming judges such as URI (urionlinejudge.com.br), repl.it, VPL (Virtual Programming Lab for Moodle - vpl.dis.ulpgc.es), among others.

We validated our method over a corpus of 938 programming exercises developed by undergraduate students during the final exam of a introductory level programming course, which was held on a blended-learning modality (combining face-to-face and online classes). As explained on Section 1, those stu-

dents share different levels of interest and/or skills in computer programming – therefore, we trained our models over a corpus reflecting many different types of students believing that it would reflect a real-world scenario.

Our method consisted in cleaning the text in source codes (removing comments and providing patterns for variable names), representing those source codes on *code-embedding* based on Skip-Gram method, and training them over a Convolutional Neural Network (CNN).

We achieved an average accuracy of 74.9% for each question (all of them representing hard-level exercises). Considering the subjectivity of the process of attributing grades to code assignments, which is a very challenging task by itself, those results reflects many possibilities of using this method to help professors on grading actual code assignments.

Future work

We will perform further experiments on other programming languages different from Java (such as Python, C++ and Javascript) to validate the possibility scaling this approach for those languages.

We will also reproduce those experiments with other model configurations, expanded and improved datasets, different neural network architectures, such as Recurrent Neural Networks (RNNs) and Hierarchical Attention Network (HAN), and other embedded representations (different from Skip-Gram) to compare with our current results.

In further experiments, we will also try to analyze other levels of code quality: while in this work we focused on code semantics, we will continue our research adding more relevant information for checking code quality, such as considering code syntax tree representation, improving error detection and/or suggesting possible corrections for wrong exercise solutions in general.

ACKNOWLEDGEMENTS

We thank the Preparatory School of UFABC for the extensive use of the webMCTest tool in order to perform three exams per year, with 600 students each. These students were selected through an exam with approximately 2,600 candidates.

REFERENCES

- Bhatia, S. and Singh, R. (2016). Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*.
- Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.
- Gulwani, S., Radiček, I., and Zuleger, F. (2018). Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–480. ACM.
- Gulwani, S., Radiček, I., and Zuleger, F. (2014). Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 41–51.
- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.
- Kim, Y. (2014). Convolutional neural networks for sentence classification. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 17461751.
- Lenci, A. (2018). Distributional models of word meaning. *Annual review of Linguistics*, 4:151–171.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26.
- Zampirolli, F. A., Batista, V. R., and Quilici-Gonzalez, J. A. (2016). An automatic generator and corrector of multiple choice tests with random answer keys. In *Frontiers in Education Conference (FIE), 2016 IEEE*, pages 1–8. IEEE.
- Zampirolli, F. A., Goya, D., Pimentel, E. P., and Kobayashi, G. (2018). Evaluation process for an introductory programming course using blended learning in engineering education. *Computer Applications in Engineering Education*.
- Zampirolli, F. A., Teubl, F., and Batista, V. R. (2019). Online generator and corrector of parametric questions in hard copy useful for the elaboration of thousands of individualized exams. In *International Conference on Computer Supported Education*.