Automatic Assessment of the Design Quality of Python Programs with Personalized Feedback

J. Walker Orr
George Fox University
jorr@georgefox.edu

Nathaniel Russell
George Fox University
nrussell18@georgefox.edu

ABSTRACT

The assessment of program functionality can generally be accomplished with straight-forward unit tests. However, assessing the design quality of a program is a much more difficult and nuanced problem. Design quality is an important consideration since it affects the readability and maintainability of programs. Assessing design quality and giving personalized feedback is very time consuming task for instructors and teaching assistants. This limits the scale of giving personalized feedback to small class settings. Further, design quality is nuanced and is difficult to concisely express as a set of rules. For these reasons, we propose a neural network model to both automatically assess the design of a program and provide personalized feedback to guide students on how to make corrections. The model's effectiveness is evaluated on a corpus of student programs written in Python. The model has an accuracy rate from 83.67% to 94.27%, depending on the dataset, when predicting design scores as compared to historical instructor assessment. Finally, we present a study where students tried to improve the design of their programs based on the personalized feedback produced by the model. Students who participated in the study improved their program design scores by 19.58%.

Keywords

Assessment, neural networks, intelligent tutoring

1. INTRODUCTION

Recently there has a been a lot of work in the development of tools for education in programming and computer science. Specifically there are many systems for intelligent tutoring which are designed to help students learn how to solve a programming challenge. The tutoring involved is primarily focused in suggesting functional improvements, that is, how to finish the program so that it works correctly.

Intelligent tutors such as [4] uses reinforcement learning to predict a useful hint in the form of an edit to a student's program that will get them one step closer to the goal of a functioning program. It uses histories of edits made by students, starting with a blank slate and ultimately terminating with a functional program to train the model. The system is based on *Continuous Hint Factory* [9] which uses a regression function to predict a vector that represents the best hint then translates that vector into a human-readable edit. Similarly [11] used a neural network to embed programs and predict the program output. Using that model of the program output, an algorithm was developed to provide feedback to the student on how to correct their program. Also, [16] use a recurrent neural network to predict student success at a task given a history of student submissions of their program for evaluations.

All these systems model student programs from *Hour of Code* [2]. *Hour of Code* is a massively open online course platform that teaches people how to code with a visual programming language. The language is simple and does not contain control constructs such as *loops*.

Moreover, the combination of language and problem setting are simple enough that there is a single or very few functional solutions for each problem [4]. This level of simplicity precludes the consideration of program design. However for general purpose programming languages such as Python, there are many ways of creating functionally equivalent programs. It is important for the sake of maintainability, modularity, clarity, and re-usability that students learn how to design programs well.

When it comes to the quality of design, there are varying standards. Further, some standards are more objective or easier to precisely identify that others. For example, the use of global variables are both widely recognized as poor design and are easy to identify. For some programming languages, "linters" exist to apply rules to check for common design flaws. For Python, Pylint [12] is a code analysis tool to detect common violations of good software design. It detects design problems such as the use of global variables, functions that are too long or take too many arguments, and functions that use too many variables. Pylint is design to enforce the official standards of the Python programming community codified in PEP 8 [15].

There are aspects of good design that are difficult to identify. For example, simple logic is a good design idea, but is quite nebulous. The complexity of a program's logic is con-

Walker Orr and Nathaniel Russell "Automatic Assessment of the Design Quality of Python Programs with Personalized Feedback". 2021. In: Proceedings of The 14th International Conference on Educational Data Mining (EDM21). International Educational Data Mining Society, 495-501. https://educationaldatamining.org/edm2021/
EDM '21 June 29 - July 02 2021, Paris, France

textual, it entirely depends on the problem the program is solving. Also, modularity is universally judged as a quality of good design, however it is not always clear to what extent a program should be made modular. How many functions or classes are too many? Again, it depends on the context of the problem for which the program is designed.

In a professional setting, code reviews are often practiced to promote quality design that goes beyond the straightforward rules of "linters." Code reviews are a manual process which require a lot of human effort. A recently developed system call DeepCodeReviewer [5] automates the code review process with a deep learning model. By using proprietary data on historical code reviews taken from a Microsoft software version control system, DeepCodeReviewer was trained to successfully annotate segments of C# code with useful comments on the code's quality.

However, to our knowledge, there is no system to perform in-depth code analysis for the purposes of evaluating and assessing design for general purpose languages in an educational context. The process of assessing the design of a program is time consuming for instructors and teaching assistants and it is an important component of complete intelligent tutoring system. Such a system needs to be adjusted or calibrated for the context of particular problems or assignments since there are important aspects of software design are context dependent. Moreover the system needs to match the particular standards of an instructor. Hence we propose a system that models design quality with a neural network trained on previously assessed programs.

1.1 Our Approach and Contribution

We propose a design quality assessment system based on a feed-forward neural network that utilizes an abstract syntax tree (AST) to represent programs. The neural network is a regression model that is trained on assessed student programs to predict a score between zero and one. Each feature the model uses is designed to be meaningful to human interpretation and is based on statistics collected from the program's AST. We intentionally do not use deep learning as it would make the representation of the program difficult to understand. Personalized feedback is generated based on each feature of an individual program. By swapping a feature's value for an individual program with the average feature value of good programs, it is possible to determine which changes need to be made to the program to improve is design. The primary contributions of this work are the following:

- The first to explicitly predict the design quality of programs in an educational setting to the best of our knowledge.
- High efficacy with an accuracy from 83.67% to 94.27% with only small amounts of training data required.
- The first intelligent tutoring system for design quality for Python.
- Personalized feedback without the explicit training or annotation.

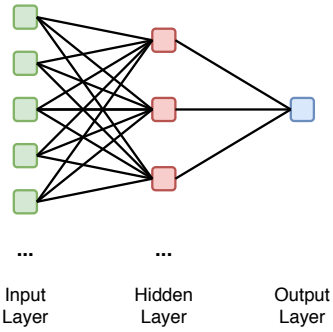

Figure 1: The model of program design quality, a feed-forward neural network. The "Input Layer" is the feature vector created from the AST. The "Hidden Layer" corresponds to calculation of x' specified in Equation 1. Finally, the "Output Layer" produces a single value, the design score as found in Equation 2.

2. METHOD

The task is to predict a design quality score for a student program written in Python. The score y is a real number between zero and one. The program is represented by a feature vector $\vec{x}$ produced by the output of a series of feature functions computed from the program's AST.

For an AST T, a series of feature functions $f_i((T)$ output is concatenated in to a feature vector $\vec{x}$ that represents key aspects of the program's design. The model $g(\vec{x}; \Theta)$ is a feed-forward neural network with a single hidden layer. It is a regression model that predicts the score y based on the feature vector $\vec{x}$ and parameters Θ.

2.1 Features

Despite recent advances in deep learning, we chose to represent the student program with feature functions computed on its AST. Deep learning is highly effective at learning useful feature representations of everything from images to time series to natural language texts. However, deep learning also requires large amounts of data and in this setting the quantity of manual annotated student programs is limited.

Additionally, AST are a natural and effective means of representing and understanding programs and can be created with free, available tools. An AST is an exact representation of the source code of program based on the programming language's grammatical structure. Producing an AST representation of a programming language is an essential first step in compilers and interpreters. The AST of a program contains all the content of its source but also is augmented with the syntactic relationships between every element. A parser and tokenizer to produce an AST for the Python programming language is provided by its own standard library. This makes the AST the natural representation to use, since it is free, convenient, exact, interpretable, and does not required any additional data. In contrast, deep learning would require a large amount of data to effectively reproduce the same representation.

Prior to representation as an AST, a program must first be broken into a series of tokens via the process of lexicaliza-

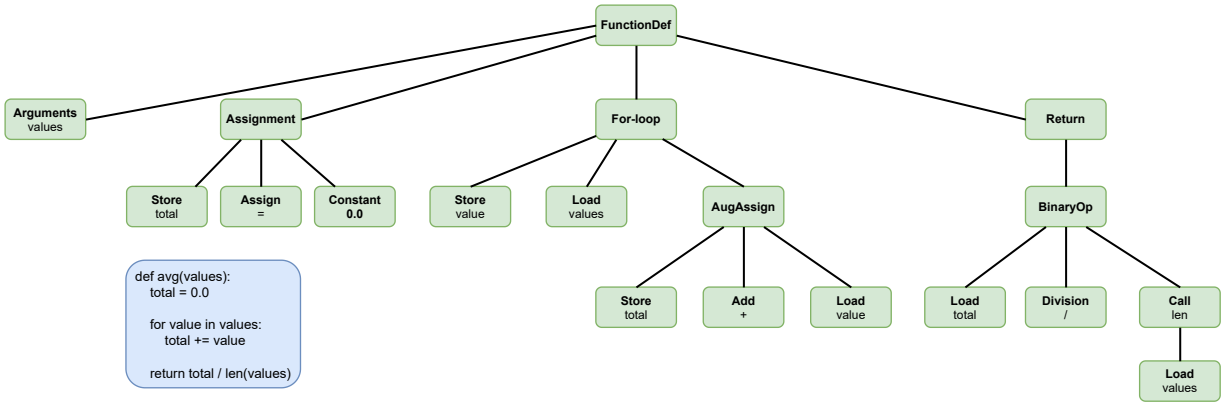

Figure 2: An abstract syntax tree for a segment of Python code that computes the average of the values in a list. The AST has been condensed for the sake of brevity and the restrictions of space. The related code segment is shown in blue.

tion. Lexicalization is the process of reading a program, character-by-character and dividing into work-like tokens. These tokens are also assigned a type such as a function call or variable reference. The grammatical rules of the language are applied to the lexicalized program to create the AST. In an AST, the leaf nodes of the tree are the program's tokens while the interior nodes correspond to syntactic elements and constructions. For example, an interior node could represent the body of a function or the assignment of a value to a variable. An example of an AST is found in Figure 2.

Given that an AST is a complete representation of a program, it is a natural basis for assessing the quality of a program's design. Deep learning may be able to automatically learning the same key syntactic relationships with enough data, however this information is simply available via AST. Further, features computed from the AST will be human interpretable unlike a representation produced by deep learning.

The features we created are all based on statistics collected from a program's AST. Some consist of simply counting the number of nodes of a given type, for example, the number of user defined functions. Other feature functions are based of subsections of the AST, such as the number of nodes per line or per function. Finally, some features are ratios or percentages such as the average percent of lines in a number in a function that are empty. All of the features are relatively simple and fast to compute, yet generally capture the design and quality of a program. Each of the feature functions $f_i(T)$ we defined can be found in Appendix A.

2.2 Model

The model is a feed-forward neural network [13] with a single hidden layer and single neuron in the output layer. The model's structure is illustrated in Figure 1. The values of the input layer are the feature vector $\vec{x}$. Each neuron in the hidden layer x'_j defined with the following equation:

$$x'_j = \text{ReLU}\left(\sum_{i=1}^{d} w_{i,j} x_i\right) \tag{1}$$

where d is the dimension of $\vec{x}$ and $w_{i,j} \in \Theta$ are the parameters, "weights" of the neuron. We use the ReLU [8] as the activation function for the hidden layer neurons. The final prediction of the design score is made by the output layer's single neuron:

$$y = \sigma\left(\sum_{j=1}^{d'} w_j x'_j\right) \tag{2}$$

where d' is the number of hidden layer neurons and $w_j \in \Theta$ are weights of this neuron. The function sigmoid is used because its domain spans from $(-\infty, \infty)$ but its range is $[0, 1]$ which ultimately guarantees the model always outputs a valid score. The model is trained with mean squared error as the loss function:

$$\text{MSE} = \frac{1}{n}\sum_{k=1}^{n}(y^*_k - y_k)^2 \tag{3}$$

where y^*_k is the ground-truth design score for the k^{th} instance i.e. program and n is the number of instances in the training data. The model is trained with the ADAM algorithm [7] and each parameter in the model was regularized according to their L2 norm [6]. For all our experiments, a hidden layer of size 32 was used. The model was trained for 250 epochs and the model from the best round according to a development set was selected for our experiments.

2.3 Ensemble

Due to the fact that fitting neural networks to data is a local optimization problem, the effect of initial values of the parameters Θ of the model remain after training. The process of training a neural network will produce a different model given the same data. This variation in the results of a trained model is particularly pronounced when the training data set is relatively small. To address this variation and mitigate its impact an ensemble of models can trained, each with different initial parameter values. Each model is independently trained and a single prediction is made by a simple of the average of individual predictions i.e.

$$y = \frac{1}{m}\sum_{l=1}^{m} y_l \tag{4}$$

where m is the number of models and y_l is the prediction of the l^{th} model. For our experiments, an ensemble of 10 models was used.

2.4 Personalized Feedback

The goal of intelligent tutors is to provide personalized feedback and suggestions on how to improve a program. The most straight-forward means of providing feedback would be to simply predict which possible improvements apply to a given program. However, training a model to directly predict relevant feedback would require a dataset of program with corresponding feedback and such a dataset can be hard to find or is expense to construct.

In order to avoid the need for a dataset with explicit feedback annotation, we use our model, trained on predicting design score, to evaluate how changes in program features would lead to a higher assessed score. Using the training data, we compute an average feature vector $\underline{\vec{x}}$ of all the "good" programs i.e. those with a design score greater than 0.75. To generate feedback for a program, its feature vector $\vec{x}$ is compared to the average $\underline{\vec{x}}$. For each feature, a new vector $\vec{x}'$ is created by replacing the feature value x_i with value with the average's value $\underline{x}_i$. This process is setting up a hypothesis, what if the program was closer to the average "good" program with regards to a particular feature? To answer this, the trained model $g(\vec{x};\vec{\theta})$ is used to predict a design score for the new vector i.e. $y_i' = g(\vec{x}';\vec{\theta})$. By comparing the original score of the program y with the new score y_i', the hypothesis can be tested. If the new score y_i' is greater than the original predicted score y, then the alteration of x_i to be closer to $\underline{x}_i$ is an improvement. Feedback based on this alteration is recommended to the student as personalized feedback. Since each feature in $\vec{x}$ is understandable to a human, feedback is given in the form of the suggestion to increase or decrease particular features. The suggestion for alteration is based on the comparison of x_i versus $\underline{x}_i$, if $x_i > \underline{x}_i$, the feedback of decrease x_i is given. In the other case, where $x_i < \underline{x}_i$ the feedback is to increase x_i. Based on the feature and the feedback of increase or decrease, a user-friendly sentence is selected from a table of predefined responses. For example, if x_i is the number of user defined functions and $x_i = 3$, $\underline{x} = 5$, and $y_i' > y$ then the feedback of "increase the number of user defined functions" is created.

3. EXPERIMENTS

The system was evaluated in two different experimental settings. The first evaluation is direct test of the model's accuracy on known design scores. For this, several different datasets and settings were compared against several baselines. The second evaluation is a small study of how student's responded to the system's feedback. Students were given feedback on the quality of their programs based on the model. They were given the chance to correct their programs after receiving feedback and have it manually reassessed.

3.1 Dataset

The dataset was collected over three years from an introduction to computer science course which teaches Python 3. It consists of four separate programming assignments which involve a wide range of programming skills. The simplest is "Travel," an assignment that involves the distance a vehicle travelled after going a constant speed for a specified duration. There are 118 student programs for "Travel." The next assignment, "Budget" is a budgeting program that lets a user specify a budget and expenses and determines if they are over or under their budget. 168 student programs were collected for "Budget". The third assignment in the dataset consists of creating a program to play "Rock-Paper-Scissors" against the computer. For this assignment, there are 111 student programs. The last assignment is programming the classic casino game "Craps" which involves rolling multiple dice and placing different types of bets and wagers. This assignment has 120 collected student programs.

All the assignments require the student to write the program from scratch in Python 3. The programs are to have a command-line, text-based interface and user validation. Students are required to use if-statements, loops, user defined functions. The "Craps" program also requires the student to do exception handling, and file I/O. A requirement of the program was to maintain a record of their winnings across sessions of playing the game, hence the results were required to be stored to a file. Also, the standards of design quality go up as the course progresses and since "Craps" is the last assignment, it has the highest standards. Each student program has an associated design score that was normalized to value between zero and one.

3.2 Baseline Methods

The model is compared against a variety of baseline regression methods. The simplest is linear regression, which simply learns a weight per each feature. Next is a regression decision tree which is trained with the CART algorithm [1]. It has the advantage over linear regression in that it can learn non-linear relationships. Non-linearity means a model can learn "sweet-spots" rather than simply having a "more is better" understanding of some features. For example, having some modularity in the form of user defined functions is good, however, too many is cumbersome. The "correct" number of user defined functions likely should fall into a relatively small range. Model selection on the maximum depth of the tree with a development set was used to determine that 10 was the best setting.

However, both of these models have the issue that they are not constrained to produce a score between zero and one, their prediction can be any real number. Hence another baseline method was used, created to be an intermediary step between linear regression and the neural network model. It is a linear model with a sigmoid transformation which guarantees the output be between zero and one. This model is effectively the final layer of the neural network model, i.e. the neural network without the hidden layer. The model is specified by the equation:

$$y = \sigma\left(\sum_{i=1}^{d} w_{i,j} x_i\right) \tag{5}$$

This model is also trained with ADAM [7]. All the baseline models and the neural network are trained with MSE as the loss function.

Method	Travel		Budget		RPS		Craps		Combined	
	MSE	Accuracy	MSE	Accuracy	MSE	Accuracy	MSE	Accuracy	MSE	Accuracy
Linear Regression	0.009	93.09%	0.032	87.43%	0.038	83.2%	**0.043**	**84.06%**	**0.027**	**87.03%**
Decision Tree	0.018	90.10%	0.031	87.26%	0.078	77.33%	0.072	79.41%	0.076	81.42%
Sig. Linear Regression	0.022	89.60%	0.046	85.13%	0.086	77.91%	0.063	81.43%	0.070	80.64%
Neural Network	0.007	93.48%	0.024	88.48%	0.041	83.9%	0.08	79.57%	0.033	85.61%
Ensemble	**0.005**	**94.27%**	**0.022**	**90.14%**	**0.022**	**87.66%**	0.053	83.67%	0.031	86.99%

Table 1: Design Score Prediction Results

3.3 Results

The model was compared versus each baseline in five different settings: the "Travel", "Budget", "Rock-Paper-Scissors", and "Craps" programs, and a combined dataset which includes all the programs. The results of the experiments can be found in Table 1. Each model is evaluated according to two different metrics: MSE and average accuracy. Average accuracy is defined as $\frac{1}{n}\sum_{k=1}^{n}\left(1-|y_k-y_k^*|\right)$.

Overall, the decision tree and the sigmoid-transformed were clearly the two worst models. This was surprising since decision trees are generally thought to be strictly more powerful than linear models. However, decision trees look for highly discriminative features to partition the data into more consistent groups. The under-performance of the decision tree possibly indicates that none of the features were especially indicative of a good or bad design on their own. Instead, the quality of a program is better described by a collection of subtle features, which gives credence to the belief that design quality is nuanced.

The reason sigmoid-transformed linear model under-performed linear regression was likely due to it being trained with ADAM. ADAM does not guarantee convergence to a global optimum like the analytical solution to linear regression. Apparently the restriction on predictions to be within the specified range of zero to one was not important.

Linear regression did surprisingly well, beating both the neural network and network ensemble in the "Craps" and combined datasets, though barely. In those cases, the difference between the ensemble and linear regression was less than a percent. This is likely due to the stability of linear regression's predictions. Though linear regression does not have the power and flexibility of neural networks this can also be a benefit by limiting how wrong their predictions are. Neural networks and even ensembles can make overconfident predictions on outliers or other unusual cases. The "Craps" dataset contained the most complex programs and it is likely a handful of predictions significantly brought down the average.

The network ensemble outperformed the single neural network in every case, which is to be expected. The margin of improvement of the ensemble versus the single neural network in accuracy on the four individual program datasets ranged from 1% to 4%. The network ensemble did the best overall by being the best in most cases or coming in a close second in all the other cases. The importance of using an ensemble is evident on the "Craps" dataset where the individual neural network under-performed significantly. On the other datasets, the neural network outperformed linear regression by a small margin, but on "Craps" the neural network model under-performed the linear regression model by 5%. Again, this is most likely due the instability and variability of neural network predictions i.e. small differences in features can lead to a large difference in the prediction. In the "Craps" dataset, the improvement of the ensemble over the single neural network model illustrates the relative stability of the ensemble's predictions. In every case, the ensemble is superior to the single neural network and had the best overall performance by producing the most accurate results on three of the datasets and effectively tying for the best on the other two.

One noticeable pattern was that all the models performed better on the "Travel" and "Budget" datasets than on the "RPS", "Craps", and combined datasets. Universally, the most difficult dataset was "Craps" which likely lowers the accuracy on the combined dataset. Due to the shifting standards and expectations of student assignments, a model per assignment appears to worthwhile. This is a bit counter-intuitive since there are many common standards and expectations across assignments.

Overall, the network ensemble produced reliable, accurate results when trained per dataset. The accuracy of the ensemble is arguably close to being useful in practical application. Further, comparing the scores of an instructor versus another instructor or even against themselves, the rate of agreement must be less than 100% and with an accuracy of the network ensemble ranging from 83.67% to 94.27%, the model's accuracy is possibly close to a realistic ceiling.

3.4 Feedback Study

In order the evaluate the effectiveness of the personalized feedback, we conducted a small study on the effect of the feedback on the design score of student programs. For the "Rock-Paper-Scissors" program the network ensemble was used to generate personalized feedback for the student programs instead of the usual instructor feedback. The network ensemble was the same as used in the design score experiments, it was trained with prior years worth of student programs. Having received the personalized feedback, students opted into correcting their program for extra credit on their assignment. The feedback was in form of a series of comments, where each comment was "increase" or "decrease" the name of a feature as described in Section 2.4.

The class is an introduction to computer science course with multiple sections and two different instructors. Students from both instructors participated in the study. Out of 73 students enrolled across the sections of the course, 15 stu-

dents chose to opt-in.

The revised programs were assessed again manually for design quality and the scores were compared against the originals. The design score of the programs started at an average of 68.33% and after the feedback and correction the average rose to 87.92%, a 19.58% absolute improvement. Using a paired t-test, the improvement was judged to be significant with a p-value of 0.001.

The results of the study suggest the feedback was generally useful in guiding students to improve the design quality of their programs. The improvement was noticeable to the instructors anecdotally as well. For example, the usage of global variables and "magic numbers" decreased significantly. Though the study does have some caveats including its small sample size and opt-in participation. It could be that those students willing to opt-in are those most willing or able improve with a second chance.

4. CONCLUSIONS & FUTURE WORK

Overall, we proposed a neural network model ensemble for predict the design quality score of a student program and experimentally demonstrated its effectiveness. Further, our system provided personalized feedback based on the difference between a program's feature values and the average features' value of "good" programs. A small study provides evidence that the feedback was of practical use to students. Students were able to improve their programs significantly based on the feedback they received.

There is also evidence that training models per assignment is most effective. However, the model needs to be evaluated on more programming assignments. Further, there is a possibility of utilizing transfer learning [10] to help the model learn what is in common across the assignments.

The feedback given was shown to be effective, but more nuanced feedback could be useful. Specifically, feedback targeted to individual lines or segments of code would possibly help students improve their program's more effectively. However, this may require additional supervision i.e. annotation for explicit training. Active learning [14] or multi-instance learning [3] may be alternatives to gathering additional annotation.

5. REFERENCES

[1] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[2] Code.org. Code.org: Learn computer science. https://code.org/research.

[3] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial intelligence*, 89(1-2):31–71, 1997.

[4] A. Efremov, A. Ghosh, and A. Singla. Zero-shot learning of hint policy via reinforcement learning and program synthesis. *International Educational Data Mining Society*, 2020.

[5] A. Gupta and N. Sundaresan. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[6] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[7] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *the 3rd International Conference on Learning Representations (ICLR)*, 2014.

[8] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[9] B. Paassen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart. The continuous hint factory-providing hints in vast and sparsely populated edit distance spaces. *Journal of Educational Data Mining*, 2018.

[10] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[11] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*, pages 1093–1102. PMLR, 2015.

[12] Pylint.org. Pylint - code analysis for python. https://pylint.org.

[13] F. Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document, 1961.

[14] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[15] G. Van Rossum, B. Warsaw, and N. Coghlan. Pep 8. https://www.python.org/dev/peps/pep-0008/.

[16] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to represent student knowledge on programming exercises using deep learning. *International Educational Data Mining Society*, 2017.

APPENDIX

A. FEATURE FUNCTIONS

- The number of functions
- The number of assignments
- AST nodes per function
- Lines of code per function
- Total lines of code
- Number of literals
- The proportion of white-space characters to the total number of characters
- Number of empty lines
- Deepest level of indentation
- Number of "if" statements
- Number of comments
- Number of AST nodes per lines of code
- Number of try-except statements

- AST nodes per try-except statement
- AST nodes per “if” statement
- Number of lists
- Number of tuples
- Average line number of literals
- Average line number of function definition
- Average line number of “if” statement
- Ratio of AST nodes inside functions versus total number of AST nodes
- Number of function calls
- Number of “pass” statements
- Number of “break” statements
- Number of “continue” statements
- Number of global variables
- Number of zero and one integer literals
- Average line number of “import” statement
- Number of numeric literals
- Number of comparisons
- Number of “return” statements
- Maximum number of “return” statements per function
- Maximum number of literals per “if” statement