Analysis of Code Submissions in Competitive Programming Contests

CS 229 Project, Autumn 2018
Wenli Looi (wlooi@stanford.edu)
Source Code: https://github.com/looi/CS229

Abstract—**Algorithmic programming contests provide an opportunity to gain insights into coding techiques. This paper analyzes contest submissions on Codeforces, a popular competitive programming website where participants solve about 5 to 10 algorithmic problems in a typical contest. We attempt to predict a user's rank and country based on a single C++ source code submission. Features were generated by running source code through the Clang C++ compiler and extracting bigrams from the tokens and traversal of the abstract syntax tree (AST). Out of several models, the neural network model achieved the highest accuracies of 77.2% in predicting rank (within one rank) and 72.5% in predicting country. Despite not achieving the highest accuracy, the GDA model was easier to interpret and allowed us to find specific differences in coding styles between programmers of different ranks and countries.**

I. Introduction

Codeforces [1] is one of many competitive programming platforms that provide an opportunity to gain insights into coding techniques. In a contest, participants solve about 5 to 10 well-defined algorithmic programming problems by writing short stand-alone solutions in a programming language such as C++, Java, or Python. Each solution is generally written by only one person (user). It is known whether the solution passed or failed. In addition, the user's skill level (rating/rank) and declared country are known. Most problems on Codeforces have hundreds or thousands of passing submissions. Unlike many other platforms, all submissions on Codeforces are publicly viewable, making it an ideal candidate for analysis.

On Codeforces, users are assigned a numerical rating based on their performance in past contests. Users are then assigned one of ten ranks based on their rating, ranging from Newbie to Legendary Grandmaster. These ranks are shown in the Data Set section in Table II.

The goal of this project is to predict a user's rank (within one rank) and country based solely on a single passing source code submission. As well, some interpretation of the learned models is done to find differences in coding styles between skill levels and countries. Since only passing submissions are considered, predictions are based only on coding style and not whether the code works or not (all code works).

Analysis from this project will not only highlight the coding techniques of competitive programmers, but may also be relevant for code written in industry or academia. While code written in programming contests differs from real-world code, some coding best practices may apply to both, and the analysis techniques used here may also be applicable to other code bases. Code from programming contests, however, is easier to analyze than most code for the reasons described above (e.g. written by exactly one person).

II. Related Work

Allamanis et al. (2018) [2] is a survey of machine learning techniques used to analyze source code. There has been significant past work on applying techniques from natural language processing (NLP) and analyzing the abstract syntax tree (AST), both of which are techniques used in this project. Machine learning has been applied to many problems, such as autocompletion, inferring coding conventions, finding defects, translating code, and program synthesis.

Burrows et al. (2007) [3] attempted to determine the author of C source code by finding N-grams in the tokens, similar to what is done here. They classified a code sample by finding the closest code sample in the corpus as measured by some similarity measure. On a collection of 1640 documents written by 100 authors, they were able to identify the author with 67% accuracy using 6-grams. A disadvantage of this approach is that searching through the entire training corpus may have scalability issues as it grows.

A study by Ugurel et al. (2002) [4] attempted to classify C/C++ source code archives into various categories, like Database, Network, and Word Processor. For features, they used single tokens from the source code as well as bigrams and lexical phrases from README files and comments. A support vector machine (SVM) was then trained to perform the classification. They achieved an accuracy of around 40-70% depending on the features and data set.

More recently, recurrent neural networks (RNNs), such as long short-term memory (LSTM) based networks, have been used to classify source code. Alsulami et al. (2017) [5] used an LSTM-based network to determine the author of Python and C++ source code. They fed a traversal of the AST into the RNN, which inspired the traversal-based method used in this paper. The RNN included an embedding layer to convert AST nodes into a fixed length vector. Their best-performing model, a bidirectional LSTM, achieved 85% accuracy on a C++ dataset with 10 authors and 88.86% accuracy on a Python dataset with 70 authors.

Techniques similar to doc2vec (Le et al., 2014) [6], where entire documents are converted to an embedding space, have also been used to classify programs. Piech et al. (2015) [7] encoded programs as a linear map between a precondition space and postcondition space. They used the linear maps with an RNN to predict feedback that an educator would provide for a piece of code, achieving 90% precision with recall of 13-48% depending on the coding problem.

Recurrent neural networks may not be better than N-gram based methods, however. Hellendoorn et al. (2017) [8] found that carefully tuned N-gram based models are more effective than RNN and LSTM-based networks in modeling source code. They had higher modeling performance (entropy) and were able to provide more accurate code suggestions.

This paper uses an approach based on N-grams and AST traversal with machine learning methods taught in CS 229.

III. Data Set

The data set currently consists of 10 contests on Codeforces from Aug to Nov 2018. All of the contests are "combined division" contests, open to users of all ranks. Each contest has ~6k submissions for a total of ~60k, with the data format shown in Table I.

TABLE I
DATA FORMAT FOR EACH SUBMISSION

Country	Rating	Source Code
RU	2193	`#include <iostream>\n#include...`
US	1747	`#include <bits/stdc++.h>\nusing...`
...	...	...

Not all contest submissions are in the data set. Only contestants with a declared country and who participated in at least one previous contest are considered. For each problem, only the latest passing submission for each user is considered (if any). Only C++ solutions are considered. C++ is the most popular language, with about 90% of total submissions being C++ in the contests used here.

The number of submissions for each rank in the data set is shown in Table II. A user's rating may change based on their performance in the contest. We only consider the user's rating before the contest.

For the country analysis, we used a subset of the data consisting only of the users in the 10 most common countries. These countries cover about 70% of the full data set. A summary of this data set is shown in Table III.

Both data sets have a significant class imbalance. Various techniques were needed to handle this, as described later.

We implemented a custom scraper for Codeforces in Python using lxml [9] to parse the HTML.

TABLE II
DATA SET USED FOR RANK PREDICTION

Codeforces Rank	Rating Bounds	# in Data	% of Data
Legendary Grandmaster	3000+	378	0.63%
International Grandmaster	2600-2999	1319	2.20%
Grandmaster	2400-2599	1781	2.97%
International Master	2300-2399	1733	2.89%
Master	2100-2299	6375	10.64%
Candidate Master	1900-2099	8756	14.61%
Expert	1600-1899	17824	29.75%
Specialist	1400-1599	11445	19.10%
Pupil	1200-1399	7405	12.36%
Newbie	0-1199	2896	4.83%
Total		59912	100.00%

TABLE III
DATA SET USED FOR COUNTRY PREDICTION

Country	# in Data	% of Data
India	12331	29.31%
China	8818	20.96%
Russia	6761	16.07%
Bangladesh	4536	10.78%
Vietnam	1753	4.17%
Ukraine	1694	4.03%
Poland	1664	3.95%
Egypt	1662	3.95%
United States	1450	3.45%
Iran	1406	3.34%
Total	42075	100.00%

IV. Preprocessing

Before applying machine learning algorithms, the C++ source code is preprocessed as shown in Fig. 1.

Source code is first converted to a sequence of strings. It is run through the Clang C++ compiler to produce a list of tokens and an abstract syntax tree (AST). Comments are removed as our focus is coding style. To help the learning algorithms generalize better, all tokens equal to a variable or function name in the AST, or representing a string or character literal, are replaced with special tokens `!!VAR`, `!!FUN`, `!!STR`, and `!!CHR` respectively. The AST is converted to a list of strings as a pre-order traversal, where additionally a special token `endblock` is added when all of a node's children have been visited. To simplify later processing, the processed tokens are concatenated with the AST traversal to produce a single sequence of strings.

The sequence of strings is then processed further. Bigrams (and unigrams) with at least 1% frequency in the training set are counted to produce features. To help prevent the learning algorithms from favoring shorter or longer solutions, each count vector is normalized by the L2 norm. (TF-IDF [10] was briefly tested, but L2 normalization seemed to work better.) The features are then scaled to have zero mean and unit variance. Without the normalization and scaling, the GDA model had a much lower accuracy, and the logistic regression model failed to converge in training (probably because the gradients were ill-behaved).

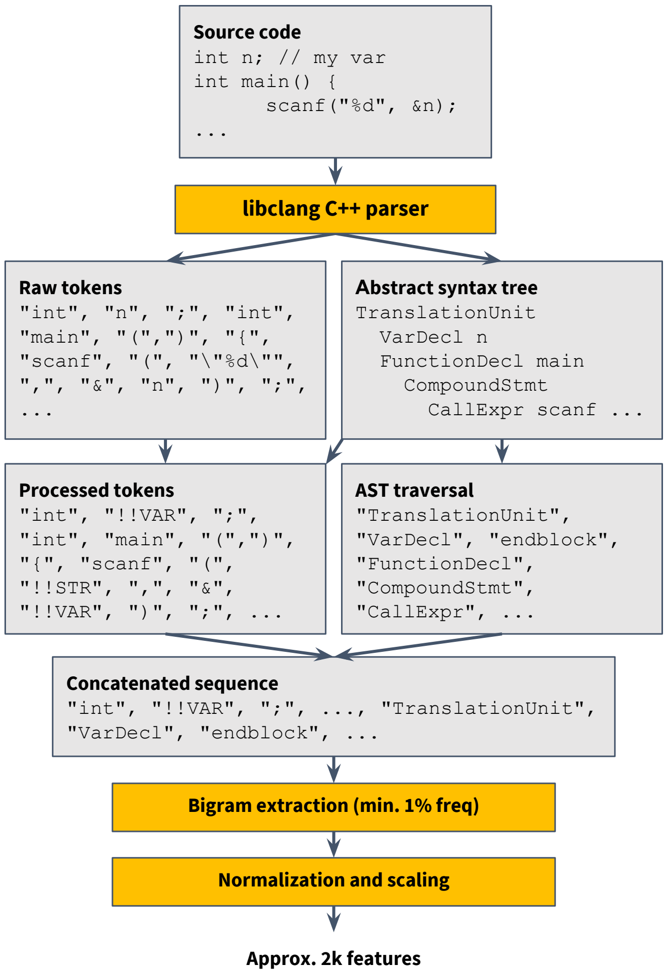
In the data set, the average number of tokens in a program is 428, the average length of the AST traversal is 627, and the average length of the concatenated sequence is 1055.

V. Methods

Several learning algorithms are used to predict a user's rating and country based on their code. Here, m denotes the number of training examples, $x^{(i)}$ denotes the feature vector for example i, and $y^{(i)}$ denotes the label for example i.

A. Linear regression

Linear regression is used to predict the user's rating and the rank is inferred from the rating using Table II. Due to class imbalance, we used a weighted least squares loss where



```
int n; // my var
int main() {
      scanf("%d", &n);
...
```

```
"int", "n", ";", "int",
"main", "(",")", "{",
"scanf", "(", "\"%d\"",
",", "&", "n", ")", ";",
...
```

```
TranslationUnit
  VarDecl n
  FunctionDecl main
    CompoundStmt
      CallExpr scanf ...
```

```
"int", "!!VAR", ";",
"int", "main", "(",")",
"{", "scanf", "(",
"!!STR", ",", "&",
"!!VAR", ")", ";", ...
```

```
"TranslationUnit",
"VarDecl", "endblock",
"FunctionDecl",
"CompoundStmt",
"CallExpr", ...
```

```
"int", "!!VAR", ";", ..., "TranslationUnit",
"VarDecl", "endblock", ...
```

Fig. 1. Data preprocessing.

the weight $w^{(i)}$ is the inverse of the number of users in the training set with the same rank:

$$\min_\theta \frac{1}{2}\sum_{i=1}^{m} w^{(i)}\left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

All of the other methods are classification algorithms, rather than regression, as they seemed to work better (see results).

B. Gaussian discriminant analysis (GDA)

The maximum likelihood estimators of each class mean μ_k and the covariance matrix Σ are computed, where again the weight $w^{(i)}$ is the inverse of the class size:

$$\mu_k = \frac{\sum_{i=1}^m 1\{y^{(i)} = k\}x^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = k\}}$$

$$\Sigma = \frac{\sum_{i=1}^m w^{(i)}\left(x^{(i)} - \mu_{y^{(i)}}\right)\left(x^{(i)} - \mu_{y^{(i)}}\right)^T}{\sum_{i=1}^m w^{(i)}}$$

Prediction assumes a uniform prior due to class imbalance:

$$p(y=k) = 1/(\#\text{ classes}) \qquad \text{(forced uniform)}$$
$$p(x|y=k) \sim \mathcal{N}(\mu_k, \Sigma)$$

C. Logistic regression

For country prediction, we use softmax regression with a weighted cross-entropy loss. Again, the weight $w^{(i)}$ is the inverse of the class size. $\hat{y}_k^{(i)}$ denotes the predicted probability that example i is in class k.

$$\hat{y}_k^{(i)} = p\left(y^{(i)} = k | x^{(i)}\right) = \frac{\exp\left(\theta_k^T x^{(i)}\right)}{\sum_j \exp\left(\theta_j^T x^{(i)}\right)}$$

$$\max_\theta \sum_{i=1}^m w^{(i)} \log \hat{y}_{y^{(i)}}$$

For rank prediction, since the goal is to predict within one rank of the actual rank, we trained a separate logistic regression model for each rank. Each training example of rank r is considered to be a positive example in the models for ranks within 1 rank of r.

$$\hat{y}_k^{(i)} = \frac{1}{1 + \exp\left(-\theta_k^T x^{(i)}\right)}$$

$$\max_\theta \sum_{i=1}^m \sum_{j=1}^{\#\text{ ranks}} w^{(i)}\Big(1\{|j - y^{(i)}| \le 1\}\log \hat{y}_j$$
$$+1\{|j - y^{(i)}| > 1\}\log(1 - \hat{y}_j)\Big)$$

All examples are considered positive examples in three ranks, except for examples of the lowest and highest ranks, which are only considered positive in two ranks. Therefore, their weight is multiplied by $3/2$ here. Empirically, this results in a model where the classification accuracy for each rank is more even.

D. Neural network

The neural network model is similar to logistic regression and uses the same loss functions. A single fully-connected rectified linear layer with 100 units is inserted between the input and output layers as shown in Fig. 2. Adding even more hidden units seemed to increase the accuracy but this was not fully tested due to time constraints.

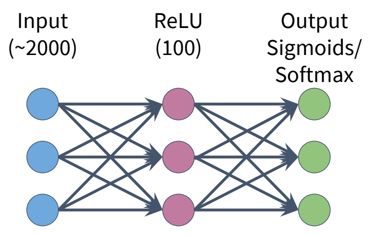


Fig. 2. Neural network architecture.

VI. Experiments

All experiments were conducted using 10-fold cross validation. For each type of model, we trained 10 models where each model is trained on 9 contests (~54k examples) and tested on 1 contest (~6k examples). The values reported here are averages over the 10 models. With this methodology, the models are tested on problems never seen in training. This ensures that the models are not learning specific features about the problems in the training set.

Due to the class imbalance described before, accuracy is defined as the weighted accuracy where the weight $w^{(i)}$ of each example is the inverse of the class size in the test set. For rank, we allow the predicted rank to be within one rank of the actual rank. If $y^{(i)}$ is the actual label and $\hat{y}^{(i)}$ is the predicted label for example i:

$$\text{Accuracy (Country)} = \frac{\sum_{i=1}^{m} w^{(i)} 1\{y^{(i)} = \hat{y}^{(i)}\}}{\sum_{i=1}^{m} w^{(i)}}$$

$$\text{Accuracy (Rank}\pm 1) = \frac{\sum_{i=1}^{m} w^{(i)} 1\{|y^{(i)} - \hat{y}^{(i)}| \leq 1\}}{\sum_{i=1}^{m} w^{(i)}}$$

The weighted accuracy shows how well the model can predict all classes and not just the majority. A model that strongly favors larger classes would achieve a high unweighted accuracy but low weighted accuracy.

For the linear regression model, we also report the weighted root mean-squared error (RMSE) for the predicted rating:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{m} w^{(i)} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{m} w^{(i)}}}$$

Scikit-learn [11] is used to train the linear regression and GDA models, while TensorFlow [12] is used to train the logistic regression and neural network models. Models were trained with the entire training set as a single batch. For logistic regression, we used gradient descent with a 0.1 learning rate, while for the neural network, we used the Adam algorithm [13] with a 0.0001 learning rate. These learning rates were experimentally found to converge. 50% dropout is used for the hidden layer, meaning that on every iteration, 50% of the hidden nodes are inactive. This helps prevent the network from overfitting and was found to increase the test accuracy.

VII. Results and Discussion

The accuracies obtained for each model are shown in Table IV. For reference, the accuracy of a model that outputs a random or constant output is shown in the first row. A model that outputs a constant or random rank, except for the highest and lowest rank, would achieve 30% accuracy because there are 3 ranks within 1 rank of the chosen rank. For country, however, we require that the model classifies the exact country, and there are 10 countries in the data set.

Classification was found to work better than regression when predicting the rank. This may be because classification optimizes what we actually care about, which is predicting the correct rank, rather than the rating. The linear regression model had a weighted RMSE (as previously defined) of 545 when predicting a user's rating in the test set. Given that ranks have a rating range of ~200, this is a fairly large error.

TABLE IV
Accuracy for each model (10-fold cross validation)

Model	Accuracy (Rank$\pm$1)		Accuracy (Country)	
	Train	Test	Train	Test
Random/constant	30.0%	30.0%	10.0%	10.0%
Linear regresion	69.6%	60.1%	N/A	N/A
GDA	75.7%	67.2%	75.0%	65.0%
Logistic regression	86.1%	71.6%	92.2%	68.4%
Neural network	94.4%	77.2%	97.0%	72.5%

GDA worked surprisingly well, achieving accuracies that are almost as high as logistic regression. While GDA assumes that $p(x|y)$ is multivariate Gaussian, logistic regression does not make that assumption and is capable of modeling a large variety of other distributions. Since the accuracies are similar, this indicates that $p(x|y)$ is Gaussian to some degree.

Out of all the algorithms, the neural network had the highest accuracies. The neural network was probably able to learn more complex relationships between the features compared to the other algorithms. Perhaps some combination of several bigrams is highly indicative of rank or country. Interpretation of the neural network is out of scope of this project, however.

The high training accuracies, compared to test accuracies, may indicate overfitting. In the neural network, dropout helped reduce overfitting (as described before), but no other regularization techniques were used. We briefly tried using principal component analysis (PCA) to reduce the number of features, and L2 regularization on the parameters, but these techniques decreased the test accuracy. More data helped reduce overfitting, as the accuracy values are about 5% higher than initial tests performed with 5 contests instead of 10.

For each actual rank and country, the neural network test accuracies are shown in Fig. 3 and 4. The model seems to be able to predict all ranks with similar accuracy. For country, the model is able to predict the more common countries with higher accuracy despite the weighted loss function used. This may be because there is significantly more training data for the more common countries.

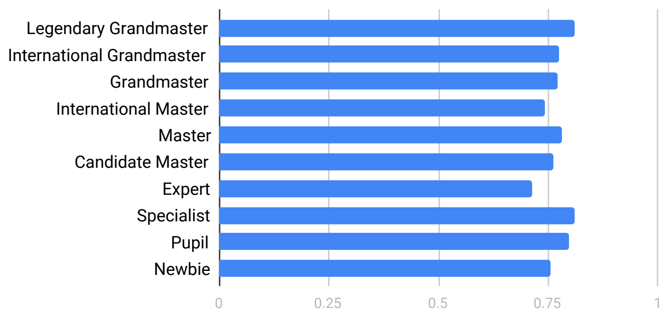


Fig. 3. Neural network test accuracy for rank ($\pm$1) by actual rank

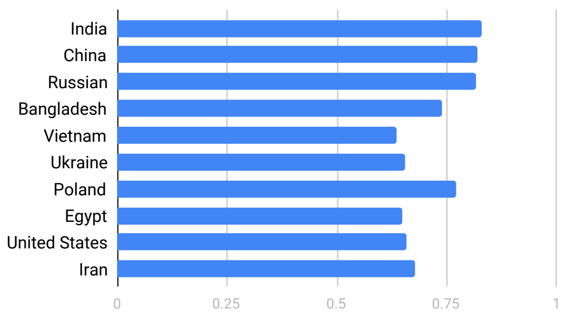


Fig. 4. Neural network test accuracy for country by actual country

VIII. Interpretation of the GDA model

While the GDA model did not achieve the highest accuracy, its simplicity makes it possible to interpret the learned model more easily. For this analysis, we randomly chose one of the models from the 10-fold cross validation. To determine the unigrams and bigrams that were the strongest indicators of high and low skill level, we compared the class means μ_k for the International Grandmaster and Pupil ranks and found the features where the class means had the largest (positive) and smallest (negative) absolute difference. These features are shown in Tables V and VI. The features are ordered in decreasing strength from left to right and top to bottom.

TABLE V
Strongest indicators of high skill level

`ifdef`	`# ifdef`	`assert`	`endif`
`# endif`	`assert (`	`( ...`	
`FunctionTemplate TemplateTypeParameter`			
`__VA_ARGS__`	`FunctionTemplate`		
`ifdef LOCAL`	`LOCAL`		

TABLE VI
Strongest indicators of low skill level

`cin >>`	`cin`	`>> !!VAR`	`>>`
`cout <<`	`cout`		
`TranslationUnit InclusionDirective`			
`TranslationUnit`		`std ;`	
`IfStmt BinaryOperator`		`main`	`main (`

From this analysis, we can see that both tokens, like `cin >>`, and AST nodes, like `FunctionTemplate`, are important to the model. As well, both unigrams and bigrams are important, although they are often related.

High skilled competitors appear to use `#ifdef` significantly, perhaps to change the code's behavior at compile time by defining macros in the compiler flags. Also, they appear to use assertions and C++ function templates.

Low skilled programmers appear to use `cin` and `cout` for input. This makes sense since `scanf` and `printf` are faster input methods and often preferred by experienced competitors.

It is interesting to see `TranslationUnit` as a strong indicator of low skill level. `TranslationUnit` is the root of the AST and appears exactly once per per program, but since the count is normalized by the L2 norm of the count vector, its value will be higher in shorter programs. Thus, it appears that GDA has learned to associate smaller programs with lower skill levels, despite having the L2 normalization to try to prevent this. It makes sense that a long program would likely indicate a hard problem and a high skilled competitor.

Tables VII and VIII show the features with the highest class means for Chinese and American competitors respectively. It seems that Chinese competitors often use `getchar` to read single characters from standard input, and import C input libraries like `cstdio`. American competitors seem to often spell out `std` in their code (like `std::cout << std::endl`) instead of importing the entire namespace with `using namespace std`, and use `ld` which is a commonly used alias for `long double`.

TABLE VII
Strongest indicators of a Chinese competitor

`= getchar`	`getchar`	`getchar (`	`char !!VAR`
`; char`	`cstdio`	`cstdio >`	`< cstdio`
`cstring`	`cstring >`	`< cstring`	`> !!CHR`
`< !!CHR`	`{ scanf`		
`UnexposedExpr CharacterLiteral`			`>= !!CHR`

TABLE VIII
Strongest indicators of an American competitor

`( std`	`<< std`	`ld >`	`> struct`
`| (`	`::`	`< ld`	`std ::`
`|`	`, std`	`struct`	`; template`
`ClassTemplate`			
`endblock ClassTemplate`			`os <<`

IX. Conclusion and Future Work

In this paper, we studied the application of machine learning techniques in predicting the rank and country of a Codeforces competitor based on a single source code submission. The neural network model achieved the highest accuracy of 77.2% accuracy in predicting rank (within one rank) and 72.5% in predicting country. Despite not achieving the highest accuracy, the GDA model was easier to interpret and we were able to find unigrams and bigrams that were the strongest indicators of certain skill levels and countries.

Future work may include testing RNN or LSTM based models, as discussed in Related Work. Acquiring more data may help reduce overfitting. Token processing could be improved, for example by replacing class and macro names with special tokens in addition to variable and function names. N-grams with $N > 2$ could be tested as only unigrams and bigrams were considered here. More hidden units or layers could be added to the neural network. Interpretation of the logistic regression or neural network model could be attempted.

References

[1] M. Mirzayanov. Codeforces. [Online]. Available: http://codeforces.com/

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Comput. Surv.*, vol. 51, no. 4, pp. 81:1–81:37, Jul. 2018. [Online]. Available: http://doi.acm.org/10.1145/3212695

[3] S. Burrows and S. M. Tahaghoghi, “Source code authorship attribution using n-grams,” in *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*. Citeseer, 2007, pp. 32–39.

[4] S. Ugurel, R. Krovetz, and C. L. Giles, “What’s the code?: automatic classification of source code archives,” in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 632–638.

[5] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, “Source code authorship attribution using long short-term memory based networks,” in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 65–82.

[6] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *International Conference on Machine Learning*, 2014, pp. 1188–1196.

[7] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, “Learning program embeddings to propagate feedback on student code,” in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. JMLR.org, 2015, pp. 1093–1102. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045235

[8] V. J. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 763–773. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106290

[9] S. Behnel, M. Faassen, and I. Bicking, “lxml: Xml and html with python,” 2005.

[10] G. Salton and M. J. McGill, “Introduction to modern information retrieval,” 1986.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[13] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980