Source Code Classification using a Neural Network

BRENT MARCUS ORLINA

ACM Reference Format:
Brent Marcus Orlina. 2023. Source Code Classification using a Neural Network. 1, 1 (April 2023), 6 pages. https://doi.org/1234567.7654321

1 INTRODUCTION

Software today are written in several different competing languages. The language chosen can affect several different factors in the implementation process of the software. However, larger codebases today are not created from a single language. Many typically use several different languages to handle different areas of the project [3]. Although most software engineers may be familiar with several different languages, most usually only know a few specific languages by heart. Companies must know the languages used in their codebase to be able to hire the correct software engineers to work for them.

A study from 2003 uses a frequency approach to correctly identify a language from written text. The study uses an algorithm that determines the probability of a language of each word, determined by the frequency of correctly matched letter in each language's alphabet. The probabilities were then used as the input for the neural network. The paper's approach managed to get an accuracy of 93.35% [7]. Although, this approach is used for non-programming languages, it can be potentially applied to programming languages as well, using keywords of each programming language to calculate its probabilities instead of letters.

Another recent study uses only the contents of source code to identify the language it was written in out of 133 languages. Data collected from popular Github repositories were tokenized, to build a vocabulary list. A token was defined as a punctuantion symbol or a sequence of characters between punctuation symbols. The tokens were only added to the vocabulary list if its frequency exceeded a certain threshold in that language. No prior vocabularies were added prior, letting the vocabulary be completely based on the files themselves. The vocabulary list was then used to build a bigram vocabulary, with each bigram similarly only being added to the vocabulary list if it exceeded a certain threshold. This vocabulary was used to tokenize the data and as input into a neural network with three hidden layers. The model managed to reach around 85.0% accuracy [1].

A different study compares text-based and image-based source code classification. Both approaches uses a convolution neural network (CNN) model, however the text-based approach tokenizes the data using word embedding which is then used as input into a 1D CNN model. The image-based approach uses a 2D CNN model. A 400x400 black and white image of the source code was used as the input. The text-based and image-based approach managed to achieve a 98.8% and 99.4% accuracy respectively [2].

Source code classification is a difficult task when reduced to purely the contents of the source code. Using a neural network allows the accuracy and reliability of classifying files and help with the analysis of large codebases. Many of the heuristics that other classifiers rely on, such as file extensions or shebang lines: `#!/usr/bin/bash`,

Author's address: Brent Marcus Orlina.

XXXX-XXXX/2023/4-ART $15.00
https://doi.org/1234567.7654321

may not be present and thus, can become unreliable. A better way is to use keywords, predefined and preserved words in programming languages, that are present in the file to estimate the language that the source code was written in. It is estimated by measuring each keyword's importance to the file.

To be able to measure the importance of the keywords in a file, a tokenizer can be created with a vocabulary of all the keywords of the chosen languages. Source code scraped from Github can then be processed by vectorizing each token in the file by using term frequency–inverse document frequency (tfidf)[6]. The vectorized array can then be used as the input for the neural network.

2 PROCEDURES

2.1 Data Collection, Tokenization, and Vectorization

The training data was collected from the top 30 most popular repositories containing the collected languages on Github. Languages collected were: Java, C++, Python, and Rust. All files of those languages from the repository were scraped using Github's REST API with the Octokit library. The data collected were decoded to plain ASCII as all scraped data using the REST API were encoded in Base64. A total of around 2800 files were collected.

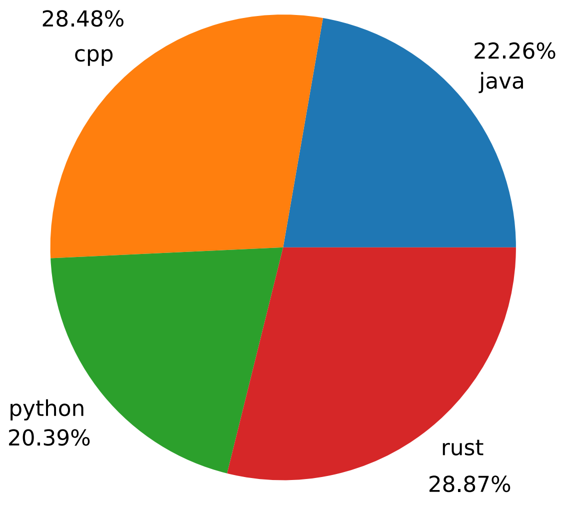

Fig. 1. Distribution of files scraped by langauge

A tokenizer is built using each language's keywords, e.g. `void` in Java, for its vocabulary. All of the data is then tokenized, breaking it down to only the keywords contained in the file. Then, the tokenized data is vectorized using term frequency–inverse document frequency (tf–idf). The tf–idf measures the importance of a term in the document, by accounting for the frequency of the term in the document, the "tf", and how common it is in all documents, the "idf". It takes the inverse of the document frequency as more common words, such as "the", "of", or "at", are likely to be less important than more uncommon words[4].

The tf–idf of a term can be calculated as:

$$tf(t,d) * idf(t,D) \tag{1}$$

where t = term, d = document, and D = set of documents. $tf(t,d)$ can be calculated as:

$$tf(t,d) = \frac{\text{raw count of } t}{\text{total number of tokens in } d} \tag{2}$$

and $idf(t,D)$ can be calculated as:

$$idf(t,D) = \frac{\text{number of documents in } D}{\text{number of documents containing } t \text{ in } D} \tag{3}$$

The data is later automatically normalized using Scikit-learn's `TfidfVectorizer` [5]

2.2 Neural Network, Hyperparameter Optimization, and Testing

The neural network used is a basic multi-layer perceptron. The paramaters of the neural network can have significant impact on the accuracy. The activation and solver parameters were first tested with three hidden layers of one hundred neurons in each layer. The activations tested were: `Identity`, `Logistic`, `Tanh`, and `ReLU`. The solvers tested were: `L-BFGS`, `SGD` (Stochastic Gradient Descent), and `Adam`. After determining the best activation and solver functions, the number of hidden layers and number of neurons were tested: one, two, and three hidden layers, and two, ten, and one hundred neurons per layer.

As the distribution of the scraped data was unbalanced, not all of the data will be used. 400 files from each language, totaling to 1600 files, were selected as the training data and were shuffled. This ensured a balanced dataset, which was used in each of the previous tests. Another 100 files from each langauge, totaling to 400 files, were selected as the testing data. The training and testing data were kept constant throughout all tests to eliminate potential bias.

After constructing the model with the best parameters, the amount of training data needed to reach a high accuracy, around 98.5%, was tested. The amount ranged from 4 to 1600 files increasing in steps of 4 to keep the distribution of the languages in the dataset balanced. The starting state of the model was also kept constant to eliminate potential bias.

3 DATA & RESULTS

Table 1. Accuracy of Solver and Activation combinations

		Solvers		
		`L-BFGS`	`SGD`	`Adam`
Activations	`Identity`	0.988	0.990	0.989
	`Logistic`	0.986	0.259	0.989
	`Tanh`	0.987	0.989	0.991
	`ReLU`	0.988	0.989	0.990

Table 1 show the results of the hyperparameter optimization for the Solver and Activation parameters. The results show that the combination of `Adam+Tanh` had the highest accuracy. These parameters are used in the later tests. Although, all other combinations, with the exception of `SGD+Logistic`, all other parameters are sufficient to use and reach high accuracy with more data.

Table 2. Accuracy of number of layers and neurons combinations

		Layers		
		1	2	3
Neurons	2	0.986	0.985	0.988
	10	0.991	0.990	0.990
	100	0.990	0.990	0.989

Table 2 show the results of the hyperparameter optimization for the number of layers and neurons per layer. The results show that a single layer of ten neurons have the highest accuracy. In addition, ten neurons per layer have a greater or equal accuracy no matter the number of layers. However, all other combinations are still sufficient to use and reach higher accuracies with more data.

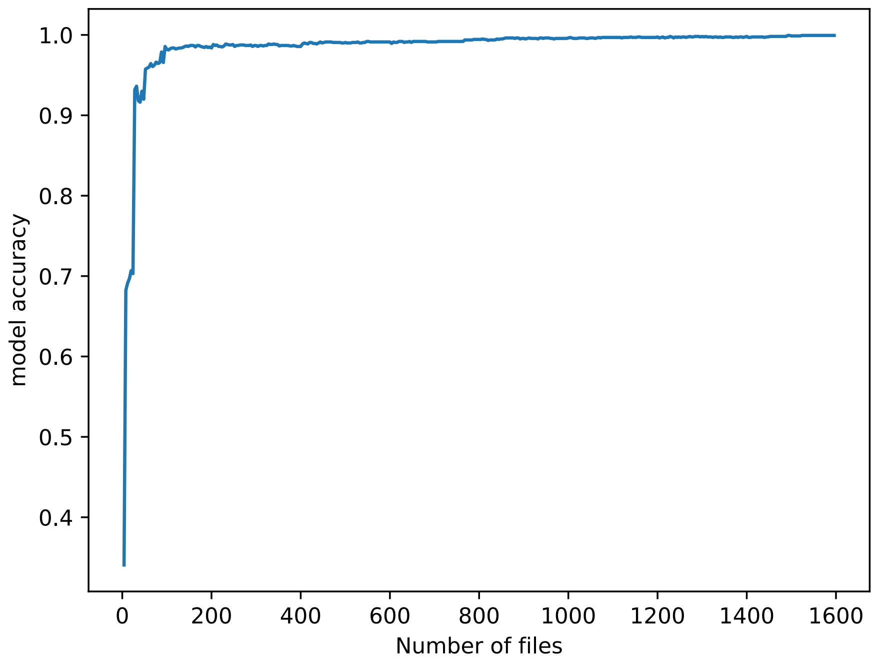

Fig. 2. How model accuracy grows as the number of files in training data increases

Figure 2 shows how the accuracy of the model, with the previous parameters established, grows as the number of data increases. It shows a sharp growth from 4 to 160 files. The accuracy grows to around 98%, and plateaus onwards. However, it still slowly grows to around 99.94% at 1600 files. For sufficiently high accuracies (98.5%), only around 160 files are needed. To achieve higher accuracies, around a thousand files are needed.

4 CONCLUSIONS AND FUTURE RESEARCH

There are several approaches to the problem of source code classification. Tian's neural network langauge classifier uses the approach of using frequencies of letters having a match in the alphabet of each language tested, reaching an accuracy of 93.35% [7]. Bonifro's approach only uses the content of the source code files to build the vocabulary of both the tokenizer and n-grams. In essence, the model, tokenizer, and n-grams have no prior knowledge of anything about the programming languages, such as its keywords. The approach managed to reach an average accuracy of 85.00% [1]. Kiyak uses a simpler approach in both image-based and text-based classification. For image-based classification, the source code is converted into a 400x400 black and white image and is used as input for a two dimensional convolutional neural network. For text-based classification, the file is vectorized using word embedding and is used as input to a one dimensional convolutional neural network. The approach reached an accuracy of 98.8% and 99.4% for both text-based and image-based approaches respectively [2].

This paper uses the approach of using keywords of programming languages to tokenize data. The data was collected from each langauge's top 30 most popular repositories, scraping all of the files of the given language. The files are decoded from base64 and tokenized using the keywords of the languages Java, C++, Python, and Rust. The data is then vectorized using term frequency–inverse document frequency to measure each keyword's importance in a given file. The vectorized data is then used as input for a basic multi-layer perceptron neural network. The parameters of the neural network were then optimized, testing for the solvers, activation functions, number of layers, and the number of neurons per layer. The amount of data used to reach a high accuracy was also tested.

The results from Table 1 show that using the combination of `Adam` and `Tanh` for the solver and activation function, it reached the highest accuracy of 99.1% The results from 2 show the a single layer of ten neurons reached the highest accuracy of 99.1%, using the parameters of `Adam` and `Tanh` for its solver and activation funtion. Only around 160 files, assuming a balanced dataset, is needed to reach a high accuracy (98.5%).

The data supports that the paper's approach is a reliable way of classifying source code. By optimizing the parameters of the neural network, using the solver `Adam` and the activation function `Tanh` with a single layer of ten neurons, it can reliably reach an accuracy of around 98.5% with only around 160 files of training data. Increasing the data to around 1600 files pushes the accuracy to around 99.94%.

There are still some areas and potential flaws needed to be addressed in the future. There are a great amount of programming languages today and there are still new langauges being created and developed today. A basic first step in future research is increasing the number of languages predicted. Inflating the number of programming languages predicted from five to around one hundred can lead to pitfalls, such as the languages using similar keywords. Another downside of this approach is that languages are being updated and developed, so keywords used today may be changed or removed in the future, making the model unreliable. Bonifro addresses this issue by not allowing any prior knowledge of programming languages to be used in the model [1].

REFERENCES

[1] Francesca Del Bonifro, Maurizio Gabbrielli, and Stefano Zacchiroli. 2021. Content-Based Textual File Type Detection at Scale. In *2021 13th International Conference on Machine Learning and Computing* (Shenzhen, China) *(ICMLC 2021)*. Association for Computing Machinery, New York, NY, USA, 485–492. https://doi.org/10.1145/3457682.3457756

[2] Elife Ozturk Kiyak, Ayse Betul Cengiz, Kokten Ulas Birant, and Derya Birant. 2020. Comparison of image-based and text-based source code classification using Deep Learning. *SN Computer Science* 1, 5 (2020). https://doi.org/10.1007/s42979-020-00281-1

[3] Damian M Lyons, Anne Marie Bogar, and David Baird. 2018. Lightweight multilingual software analysis. *arXiv preprint arXiv:1808.01210* (2018).

[4] Darla M. 2021. How TF-IDF Works. https://towardsdatascience.com/how-tf-idf-works-3dbf35e568f0
[5] Scikit-Learn. 2023. TfidfVectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
[6] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.
[7] J. Tian and J. Suontausta. 2003. Scalable neural network based language identification from written text. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, Vol. 1. I–48. https://doi.org/10.1109/ICASSP.2003.1198713