

Programming Language Prediction using Machine Learning

Nidhun M

Department of Computer Applications
Amal Jyothi College of Engineering
Kanjirappally, India
nidhunofficial@gmail.com

Sona Maria Sebastian

Department of Computer Applications
Amal Jyothi College of Engineering
Kanjirappally, India
sonasebastian@amaljyothi.ac.in

Abstract : The primary tool used in the software development industry is programming languages. Since the 1940s, hundreds of them have been developed, and every day, a sizable number of new lines of code are written in a variety of programming languages and pushed to active repositories. We consider a source code classifier to be a highly valuable tool for automatic syntax highlighting and label suggestion on systems, such as code editors, that can identify the programming language used to write a certain piece of code. This motivated us to use cutting-edge AI methods for text classification to build a model for categorizing code snippets according to their language. We developed a new dataset for our empirical investigation using the GitHub Repos Dataset, which includes 131450 code snippets dispersed over 34 programming languages.

Keywords — *Classification, Machine learning, Random Forest, NLP, Source code Detection*

I. INTRODUCTION

Since distinct development jobs sometimes need different programming languages, these days, the large-scale software creation process commonly uses multiple programming languages like Python, JavaScript, C, and Ruby. Users can organise and exchange source code with other users using platforms for code snippets like Gist and pastebin. These technologies assume that the user has already assigned the appropriate programming language to the source code and are unable to predict the coding languages of those source code.

The current answers to this prediction issue are insufficient. Instead of relying on the source code itself, integrated development environments (IDE) like Atom, Pycharm, and text editors like VSCode, Vim, and Bluefish guess the coding language based on extension of file. Users may experience issues since they must manually generate a code snippet with the appropriate extension to the syntax highlighting in various editors.

II. LITERATURE REVIEW

Van Dam et al. [2] suggested a software language model in earlier studies on the classification of source code in order to identify the whole code snippet from GitHub. Five natural language statistical models serve as the foundation for their classifier. The classification looked at 19 programming languages. Over 20,000 source code files were gathered by Khasnabish [3] and from several GitHub repositories, these code snippets can be obtained. In order to forecast ten programming languages, the model makes use of a Bayesian classifier. Klein [5] collected 25

randomly chosen source code files for the test set and 45,612 code snippets for the training set using GitHub. However, their approaches, that are based on feature selection and supervised learning approach, can only manage a maximum accuracy of 48%. Alrashody[6] proposed the Source Code Classifier technique for categorizing source codes using a Naive Bayes classifier, which had an accuracy 75% and this technique is also 80% accurate at differentiating across families of programming languages (like C, C#, and C++) and 61% accurate at differentiating between versions of Csharp programming language.

III. PROPOSED SYSTEM

Applications of machine learning (ML) and NLP-based techniques to source code analysis have been widely used. These methods have been shown to be effective for a variety of essential programming language-related tasks, including as detecting programming languages, reading summaries of source code, and offering code completion and suggestions. The use of ML and NLP approaches to categorise programming languages in source code files has drawn considerable attention from the scientific community. It has been demonstrated that a source code file's programming language may be determined with great accuracy.

Prior studies that examine the categorization of programming languages have relied heavily on the GitHub dataset, which usually has a large number of source code files. Since the large sample provides a variety of attributes that help the machine learning model improve its learning capabilities, a huge source code file can be categorised using ML and NLP techniques with a very high degree of accuracy. We focus on a tool that can categorise code snippets in this paper, which are condensed, reusable bits of code that include at least two lines. – This task is significantly more difficult. Finding the computer language used in a code snippet might still be challenging. In order to identify the programming language used in a few lines of source code, we are therefore proposing a classification model.

The following is a summary of our study's significant contributions:

- (1) Contributing a classifier that determines the coding language of GitHub source codes using textual information.

- (2) A prediction model that uses code snippets to forecast the programming language and is based on Random Forest (Breiman, 2001). This model is demonstrated to have a 92.8% accuracy rate, a 0.94 precision rate, and a 0.91 recall rate.

IV. TOOL DESCRIPTION

The coding language of code snippets can be determined using this tool for source code classification. At the moment, it can recognise 34 distinct programming languages.

A. Dataset

The 2.8 million GitHub repositories that make up the dataset, which goes by the name of "github dataset" comprise code and comments. Big Query Helper class helps to simplify common BigQuery tasks like executing queries, showing table schemas, etc without worrying about table or dataset pointers. BigQuery module helps to improve the flexibility by separating the system that verify the data from the corresponding storage options. Utilizing the BigQuery helper module, a custom dataset is created. The following requirements needed to be in the search query:

- Select code snippets and scripts file names only
- Determine how many rows should be chosen for each language.
- Recognize a language by the various file extensions that it may use.

Our resulting dataset contains 131455 code snippets distributed over 34 programming languages.

The resulting Query looks like this :

```
(SELECT sample_path, content
FROM `bigquery-public-data.github_repos.sample_contents`
WHERE (binary = False AND (sample_path LIKE "%.bat" OR sample_path
LIKE "%.cmd" OR sample_path LIKE "%.btm")))
LIMIT 5000)

UNION ALL

(SELECT sample_path, content
FROM `bigquery-public-data.github_repos.sample_contents`
WHERE (binary = False AND (sample_path LIKE "%.c")))
LIMIT 5000)

UNION ALL

...
```

Figure 1

B. Random Forest Classifier

Random forests, also known as random decision forests, are an ensemble learning method for classification, regression, and other tasks that works by constructing a large number of decision trees during training. For classification tasks, the random forest output is the class chosen by the majority of trees. The mean or average

prediction of the individual trees is returned for regression tasks. trees gathered from a randomly chosen subset of the training set, with those decision trees being used to make the final prediction. Notable benefit of the proposed model is that even if few trees decide incorrectly, the accuracy of the outcome won't be severely impacted. Additionally, it does not have the overfitting issue that the Decision Tree model does. Because a notable amount of trees in the forest provide excellent accuracy, the total amount of trees in the model play a crucial role.

V. METHOD

A. Data exploration

The figure 2 shows resulting Dataset includes 131455 code fragments spread over 34 different programming languages.

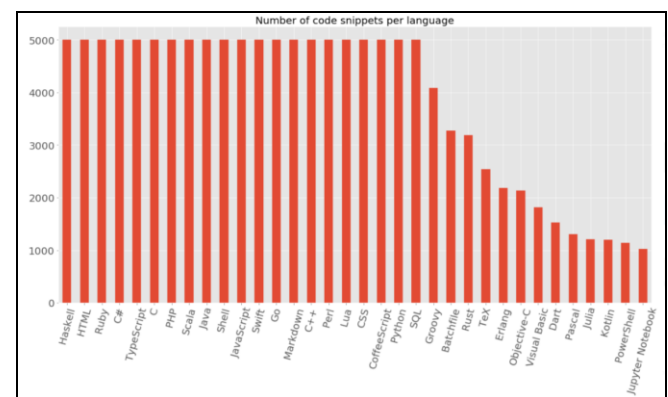


Figure 2

B. Data Preprocessing

This stage involves cleaning and filtering the data. But we discover the earlier snippet identification techniques view the snippets as the basic component. The main drawback is the problem of out-of-vocabulary (OOV) cannot be properly resolved. That implies that there are certain words that are present in the testing set but not in the training set.

Instead of working with plain English text, we are using source code. To tokenize the data, we will alter Sklearn's tokenizer.

This is the outcome that we should anticipate for Python code:

```
import pandas as pd

data = pd.read_csv("file.csv")
```

To describe the pattern of a token for this use, regular expressions will be used.

For this classifier, initially had 3 types of tokens are employed:

- Identifiers
- Operators

- Brackets

When evaluating the various tokens obtained, it is observed that many of them are single-character variable names or ones made up of a string of the same character, which do not provide the classifier with much additional information. A class called FunctionTransformer from scikit-learn eliminates stop words.

C. Vectorization

The dataset is made up of a list of tokens rather than raw text in order to identify discriminating words for each language with a vocabulary of size M (M being the number of unique tokens across all the code snippets). The next step is to turn each code fragment into an array of size M after assigning each token to its corresponding index i in the vocabulary.

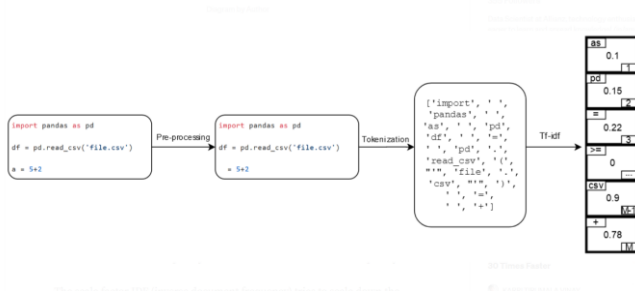


Figure 3

The term frequency-inverse document frequency (TF-IDF) vector representation links each source code document to an array of size M, where the i-th element of the array represents the scaled frequency of the token in the document.

The following equation can be used to determine the TF-IDF factor for a given token i in a document d:

$$tf-idf(i, d) = tf(i, d) * idf(i)$$

TF - Term Frequency

IDF - Inverse Document Frequency.

VI. IMPLEMENTATION

A. Experimental Subject

We choose to carry out our experiment using the dataset produced through custom training. This includes snippets of code written in 34 different programming languages, like Batchfile, C, C#, C++, CSS, CoffeeScript, Dart, Erlang, Go, Groovy, HTML, Haskell, Java, JavaScript, Julia, Jupyter Notebook, Kotlin, Lua, Markdown, Objective-C, PHP, Pascal, Perl, PowerShell, Python, Ruby, Rust, SQL, Scala, Shell, Swift, TeX, TypeScript, Using stratified sampling, we use 1,05,164 snippets for the training of the model and 26,291 snippets for testing of the model

B. Implementation Details

Based on the GitHub dataset and Random Forest classifier, this straightforward model was created. The dataset must be loaded, and the feature set must be chosen, before the classifier can be run. The dataset is

then divided into test and training sets with a 0.2 test size, and the data is then preprocessed and vectorized.

After obtaining our data into the proper shape, the next step is to select a model to use it with. Since no algorithm is effective for all data science issues, it is up to us to consider all of our possibilities and choose the classifier that performs the best. GridSearchCV and Pipeline from Sklearn will be used to fully automate the deployment and model search processes.

1) Pipeline

Multiple estimators can be chained into one using pipeline. The pipeline's goal is to put together a number of phases that can be cross-validated while using various parameters.

2) GridSearchCV

For an estimator, GridSearchCV can be used to thoroughly search across the provided parameter values.

We used 30% of the Dataset due to computational costs to run the GridSearch over the many models we've chosen to compare. Our GridSearch results show that the Random Forest classifier will be the best choice for the ideal model.

The model must then be trained using the chosen features. The command line will then prompt users to enter their code snippet. The model will then produce the snippet's estimated programming language. On a machine with an Intel Core i5-8550K 3245 processor and GeForce RTX390 GPU with 16 GB of memory, all of the experiments are run. Windows 10 is the OS platform currently in use.

C. Performance Measures

The important performance evaluations are Accuracy, Precision, Recall, and F1. Used to evaluate the performance of our proposed technique to the baselines. In order to introduce these measurements, we first provide illustrations of the following ideas:

- True Positive (TP): The result of the prediction is accurate for the positive sample.
- True Negative (TN): It is correctly predicted that the negative sample will be negative.
- False Positive (FP): A false positive prediction is made for the negative sample.
- False Negative (FN): A false negative prediction is made for the positive sample.

Consequently, the following are the four performance measures that can be identified:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

The best performance metric is known as accuracy, this metric is calculated as the proportion of correctly predicted observations to all observations. Precision is defined as the ratio of all successfully predicted positive observations to correctly expected positive observations. Recall shows how many of the sample's positive examples were properly predicted. The F1 factor combines Precision and Recall.

VII. RESULT ANALYSYS

The average precision, recall, and F1-score scores for our suggested model were 0.99, 0.99, and 0.99, respectively, and it had a 96.8% accuracy rate.

In Table I, the performance of each programming language's identifier is shown. The model in the confusion matrix (figure 1) above has a tendency to mix up some languages with particular others. JavaScript and TypeScript are two examples of this.

We can observe, for instance, that the model tends to mix up TypeScript and JavaScript samples about 4% of the time in this zoomed-out view of the confusion matrix. The fact that Typescript is a superset of JavaScript and every JavaScript code is fully valid TypeScript means that these outcomes are not unexpected.

Based on the three performance criteria F1, Precision, and Recall and support, Table 1 displays the specific results for each language

Table 1: The detailed performance for each programming language

	precision	recall	f1-score	support
Batchfile	0.98	1.00	0.99	675
C	0.99	1.00	0.99	987
C#	1.00	1.00	1.00	1044
C++	0.99	0.99	0.99	999
CSS	1.00	1.00	1.00	995
CoffeeScript	0.99	0.99	0.99	984
Dart	1.00	0.98	0.99	328
Erlang	1.00	1.00	1.00	453
Go	1.00	1.00	1.00	1009
Groovy	1.00	0.99	0.99	827
HTML	0.98	0.99	0.99	973
Haskell	1.00	1.00	1.00	950
Java	1.00	1.00	1.00	1035
JavaScript	0.98	0.98	0.98	956
Julia	1.00	0.98	0.99	194
Jupyter Notebook	1.00	1.00	1.00	204
Kotlin	1.00	0.99	0.99	222
Lua	1.00	1.00	1.00	1064
Markdown	0.98	0.98	0.98	1022
Objective-C	1.00	0.97	0.99	479
PHP	1.00	0.99	1.00	989
Pascal	1.00	1.00	1.00	257
Perl	1.00	1.00	1.00	1000
PowerShell	1.00	0.99	0.99	226
Python	0.99	0.99	0.99	979
Ruby	0.99	1.00	1.00	1035
Rust	1.00	1.00	1.00	635
SQL	1.00	1.00	1.00	993
Scala	1.00	1.00	1.00	957
Shell	0.99	0.99	0.99	1001
Swift	1.00	1.00	1.00	979
TeX	1.00	1.00	1.00	494
TypeScript	0.98	0.99	0.99	969
Visual Basic	1.00	1.00	1.00	375

Table1

Table 1 shows that a programming language identifier performs well in the majority of programming languages. We used a grid search to adjust the model's hyperparameter in order to improve the classification model's performance.

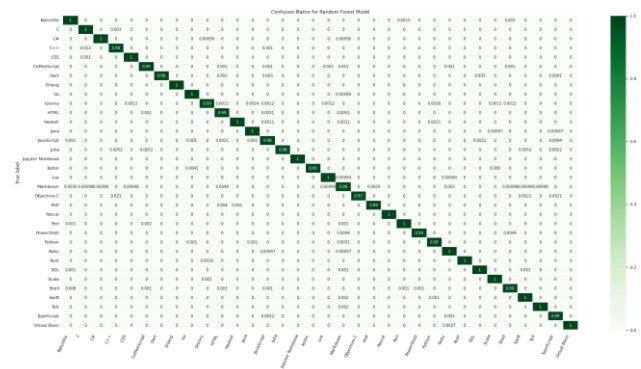


Figure 4 Confusion Matrix of model

VIII. CONCLUSION

The significance of identifying languages from code snippets was covered in this paper. We claimed that, given the complexity of today's programming languages, it is far more challenging to determine the coding language from source code than it to do so from source code files. With the help of a Random Forest classifier that was trained using data from GitHub, and we introduced Programming language Identification tool. The suggested tool's average scores for precision, recall, and the F1 score were 0.92, 0.96, and 0.97, respectively, while the programming language identifier had a success rate of 96.8%.

Finally, we can say that while our model typically produces outstanding results, it can struggle (understandably) with very short code snippets that have few or no distinguishing syntax traits.

IX. REFERENCES

- [1] Guang Yang ,Yanlin Zhou , Chi Yu and Xiang Chen, "DeepSCC: Source Code Classification Based on Fine-Tuned RoBERTa", 2021
- [2] J. K. Vans Dam and V. Zaytsev, "Software language identification with natural language classifiers,"
- [3] J-N-Khasnbish, M. Sudhi, J. Deshmukh, and G. Srinivasaraghavan, "Detecting programming language from source code using bayesian learning techniques," in International Session on Machine Learning and Data Mining, 2014
- [4] X. Chen, C. Chen, D. Zhang, and Z. Xing, "Sethesaurus: Wordnet in software engineering," IEEE Transactions on Software Engineering, 2019.
- [5] D. Klein, K. Murray, and S. Weber, "Algorithmic programming language identification," arXiv preprint arXiv:1106.4064, 2011.
- [6] K. Alreshedy, D. Dharmaretnam, D. M. German, V. Srinivasan, and T. A. Gulliver, "Scc: automatic classification of code snippets," arXiv preprint arXiv:1809.07945, 2018.
- [7] K. Cao, C. Chen, S. Baltes, C. Treude, and X. Chen, "Automated query reformulation for efficient search based on query logs from stack overflow," in Proceedings of the International Conference on Software Engineering, 2021
- [8] Christopher D. Manning, Prabhakar Raghavan, H. S. (2008). Introduction to information retrieval. <https://nlp.stanford.edu/IR-book/>