IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Revue d'Intelligence Artificielle
Vol. 37, No. 5, October, 2023, pp. 1229-1236
Journal homepage: http://iieta.org/journals/ria

Using Natural Language Processing for Programming Language Code Classification with Multinomial Naive Bayes

Ayman Hussein Odeh[1*], Munther Odeh[2], Hussein Odeh[1], Nada Odeh[1]

[1] College of Engineering, Al Ain University, Al Ain 64141, United Arab Emirates
[2] Department of Mathematics and Science, Oldenburg University, Oldenburg D-26111, Germany

Corresponding Author Email: ayman.odeh@aau.ac.ae

https://doi.org/10.18280/ria.370515

Received: 31 May 2023
Revised: 25 August 2023
Accepted: 1 September 2023
Available online: 31 October 2023

Keywords:
artificial intelligent, classification, detection, classification, machine learning, programming language

ABSTRACT

Classifying Programming Languages scripts is very important task for several reasons such as: automated analysis, code maintenance, code search, quality assurance, and code understanding; this process is similar to processing natural languages, especially high-level languages like Python, Java, C#, C, C++, PHP, JavaScript, and others. Leveraging natural language processing concepts, this research explores the application of the Multinomial Naïve Bayes (MNB) algorithm to identify and classify programming languages used in source code files. MNB is a relatively simple and fast algorithm for text classification. The study utilizes a dataset comprising 12 programming languages and consists of 12,003 samples, totaling 396,090 lines of code. The MNB algorithm is trained on this diverse dataset, and its performance in classifying programming language source code is evaluated. The results of the study demonstrate an impressive accuracy rate of 95.09% in accurately identifying and classifying programming languages. This high accuracy highlights the effectiveness of the applied NLP techniques, specifically the MNB algorithm, in the classification task. The findings of this research have significant implications for multi-programming language editors such as Visual Studio Code and Notepad+ or any programming editor. With the automatic recognition of programming languages enabled by this approach, users can conveniently paste source code into these editors, and the system will automatically identify and classify the programming language being used. This functionality enhances the user experience and streamlines the coding process, particularly in multi-language development environments.

1. INTRODUCTION

With the rapid expansion of software development, the volume of programming language source code has increased exponentially. Classifying PL from source is very important task for several reasons such as: automated analysis, code maintenance, code search, quality assurance, and code understanding. Analyzing and understanding this vast codebase is crucial for software engineers, researchers, and stakeholders in the software industry. As a result, the development of efficient and automated methods for classifying source code has become a significant area of interest. Traditional approaches to analyzing and classifying programming language source code are based on each language's reserved words and the grammatical patterns and grammar of each language [1]. However, the appearance of new programming paradigms and languages has posed challenges to these rule-based methods [2, 3]. To overcome these limitations, researchers have turned to natural language processing (NLP) techniques [4], which offer a promising alternative for programming language analysis.

The classification of programming language source code is a challenging task that has garnered attention due to its potential to enhance software development processes. While existing approaches based on syntactic analysis have provided some success, they often struggle to accurately classify code written in domain-specific languages (DSL) [5] is a specialized PL used for solving problems in a particular domain or formal system designed to address a specific problem domain, task, or application area, scripting languages, or those that do not adhere strictly to traditional syntax rules [3]. This limitation has motivated researchers to explore the application of NLP techniques to source code classification. And also, since PLs, particularly high-level languages, contain English words and have well-defined structures; they can be processed similarly to natural languages. This similarity in structure and style motivated us to conduct this research and explore the application of artificial intelligence techniques to classify, detect, and identify texts written in programming languages. However, there is still a need for comprehensive studies that evaluate the effectiveness and performance of NLP-based approaches, particularly when combined with specific classification algorithms.

This research paper presents a novel approach to programming language source code classification by applying natural language processing techniques, specifically focusing on the Multinomial Naive Bayes (MNB) algorithm [6]. Our study aims to bridge the gap between NLP and programming language analysis by treating source code as natural language text and leveraging statistical algorithms to extract meaningful features and patterns. By adapting the MNB algorithm to source code classification, which is a simple and effective supervised machine learning technique based on probabilistic methods using Bayesian networks (BN) [7] (BN is a

probabilistic graphical model that represents a set of random variables and their probabilistic relationships using a directed acyclic graph); we expect to improve the accuracy and efficiency of the classification process. Although Naive Bayes (NB) is known for its simplicity, it can still achieve high prediction accuracy, even with relatively small datasets. The NB algorithm comprises three types: Gaussian, Multinomial, and Bernoulli. The choice of the NB classifier depends on the distribution of input features and the nature of the problem being addressed [8]. The Gaussian algorithm is suitable when the input features follow a normal distribution [9].

This research paper makes two significant contributions. Firstly, it provides a comprehensive analysis of current programming language source code classification methods, emphasizing their limitations and drawbacks. Secondly, it introduces a novel methodology that integrates NLP techniques with MNB algorithm to tackle the source code classification problem. Our approach encompasses customized preprocessing steps, feature extraction techniques, and tailored implementation details of the MNB algorithm specifically designed for programming language analysis. This study aims to:

•Apply NLP techniques to process PL code, which is similar to processing NL, to evaluate the performance of MNB algorithm in accurately.

•Identifying and classifying PL based on diverse dataset, to compare it with alternative approaches and demonstrate their effectiveness and advantages.

•As a result of this research, automated code analysis can be enhanced, software engineering tasks can be facilitated, and the field can be advanced.

In summary, this research aims to leverage NLP techniques and the MNB algorithm to improve programming language source code classification. By addressing the limitations of existing approaches, we strive to enhance the accuracy and efficiency of automated code analysis, providing valuable insights for software engineers and researchers in the field. This paper is an extending for a paper "Using Multinomial Naive Bayes Machine Learning Method to Classify, Detect, and Recognize Programming Language Source Code" [10], presented at the ACIT2022 Conference.

The paper is organized as follows: Section 2 illustrates the related works for PL and NLP classification. Section 3 describes the proposed work in detail. Section 4 presents the achieved results. Section 4 discusses the results of the proposed work and compares them with existing state-of-the-art methods. Section 6 presents the conclusion and provides context for future work.

2. RELATED WORKS

The classification, detection, and recognition of programming languages from source code are a highly interesting research topic. For example, DeepSCC research [11] uses Stack Overflow with 224,445 pairs of source code samples (snippets) written in 15 programming languages; it concludes to 87% of (Precision, Recall, and F1-score). The research [12] achieved an accuracy of 93.48% using Bayesian learning techniques to detect programming languages from a dataset consisting of more than 20,000 source code files. In the domain of design pattern recognition, another study [13] utilized a convolutional neural network (CNN) to classify 90 programming languages with an accuracy of 90%. The CNN model was trained on a dataset sourced published in GitHub repositories., Nagy and Kovari [14] presented a framework to validate and recognize design patterns in C# source code. They evaluated their model using various class libraries. Additionally, Van Dam and Zaytsev [15] employed NLP to identify the programming language of source code from GitHub, achieving a 97.5% accuracy for 19 programming languages. Rahman et al. [16] proposed a source code classification based on neural networks using long short-term memory (LSTM), the results of this research shows that the accuracy of classification is 94.8% based on training its model using dataset with more than 2000 problems. The Guesslang Algorithm [17] employed a deep learning model with neural networks and linear classifiers to classify a vast number of source code files from GitHub. It achieved a performance accuracy of 93.45% when tested against 230,000 distinct source code files. Several software tools were identified for programming language classification. SourceClassifier [18] utilized the Naive Bayes classifier to detect the programming language of a given source code, while SyntaxHighlighter [19], used in WordPress, leveraged predefined keywords for language detection and syntax highlighting. Other research explored alternative approaches. "Application of computational intelligence for Source Code classification" [20] employed Evolutionary Algorithms to generate classifiers based on source code content, yielding improved accuracy and flexibility. Kiyak et al. [21] combined text and image-based comparisons to identify programming languages with high accuracy (over 93.5%) across three datasets. For the Arabic language, research [22] achieved a classification accuracy of 100% using a deep learning model based on multi-layer perceptron neural networks. The concept of graph-based representations for programming language source code was introduced by Allamanis et al. [23], highlighting their ability to capture structural and semantic information for classification purposes. Another research [24] addressed the challenges of working with noisy and incomplete data, exploring machine learning and deep learning techniques for program synthesis and classification. Furthermore, Hellendoorn and Devanbu [25] investigated the effectiveness of deep neural networks in modeling programming language source code, comparing them with traditional machine learning algorithms. Nye et al. [26] focused on learning program sketches, using probabilistic models to infer missing code sections and their relevance to source code classification.

By building upon existing approaches in programming language source code classification, this research paper aims to contribute to the field by applying NLP techniques, specifically the Multinomial Naive Bayes (MNB) algorithm. The effectiveness of various techniques, including deep learning, NLP, and graph-based representations, has been demonstrated in previous studies (as shown in Table 1). A survey paper [27] provided an overview of machine learning techniques, such as Naive Bayes, decision trees, and support vector machines, applied to code classification tasks. These studies collectively enhance our understanding of programming language source code classification. The current research paper seeks to further advance the field by utilizing NLP techniques, particularly the MNB algorithm, to improve the accuracy and efficiency of programming language source code classification.

To address the research gaps and challenges in programming language source code classification, this proposed study focuses on utilizing NLP techniques,

specifically the MNB Algorithm. By reviewing the literature related to this research, we found several gaps, including: Challenges in classifying programming languages and keeping abreast of their developments; the need to improve the accuracy of recognition and classification. And take into account the variation in code quality. To avoid these gaps MNB algorithm was used, which creates probability distributions based on word (bigram) frequencies in the training data (source code scripts). This enables the algorithm to classify new source code data by calculating the probability of belonging to each class. By applying this algorithm, a diverse dataset consisting of source code files from 12 different programming languages, the research enhances the classification of PL source code achieving an average accuracy (95.09%). Furthermore, the proposed study allows the algorithm to adapt to evolving programming languages by retraining on relevant datasets. Its adaptability makes it useful for software developers, as it can automatically classify source code even for new programming languages.

Table 1. Summary of reviewed works

Reference	Methodology	Dataset	Accuracy
[13]	Convolutional Neural Network (CNN)	Dataset from GitHub repositories	90%
[12]	Bayesian Learning Techniques	More than 20,000 source code files from multiple GitHub repositories	93.48%
[14]	Framework for PL Design Pattern Recognition	C# source code with class libraries	great accuracy
[15]	Natural Language Processing (NLP)	Source code from GitHub	97.5%
[16]	long short-term memory (LSTM)	Dataset with more than 2000 problems.	94.8%
[17]	Guesslang Algorithm using neural networks	1,900,000 source code files from 170,000 public GitHub projects	93.45%
[21]	Deep Learning Model comparing text and image	Three datasets of source code for eight different programming languages	>93.5%
[23]	Graph-based Representations	Programming language source code GitHub 29 2.9 million LOC	85.5%
[24]	Machine Learning for noisy and incomplete data	1100, 000 JavaScript code samples	up to 77%
[25]	Deep Neural Networks	Programming language source code: 14,317 projects consisting of 2230075 files	94%
[26]	Machine Learning for program sketches	Incomplete code snippets (program sketches) 8000 samples	95.8%
[11]	fine-tuned RoBERTa	Stack Overflow, with 224,445 pairs of code snippets	87%

3. METHOD

All high-level programming languages utilize a language that closely resembles English, and occasionally, we can encounter explicit usage of English words within these languages. Consequently, we can consider all programming languages as a unified form of English, although with different structures and styles. The following is the algorithm for using MNB Machine Learning Method to classify, detect, and recognize programming language source code. The MNB method is a relatively simple and fast algorithm for text classification comparing to other NLP methods. It is a one of the simplest classification techniques based on supervised machine learning probabilistic method, but this simplicity does not make it a pitiable choice, it capable to produce a very high prediction accuracy even if there are not very large dataset of samples.

The proposed method was implemented through the following steps: Collect a dataset of source code files in various programming languages. Preprocess the data by extracting relevant features from the source code files, such as syntax elements and statistical information. Split dataset into two parts (training 90%, and testing part 10%), then train the proposed model on the extracted victor of features. Evaluate the performance of the model on the testing data by calculating the accuracy, precision, recall, and F1 score. Use the trained model to classify new, unseen source code files by extracting features and predicting the programming language of the file.

The general structure of this model shown in in Figure 1, and its components will be discussed.

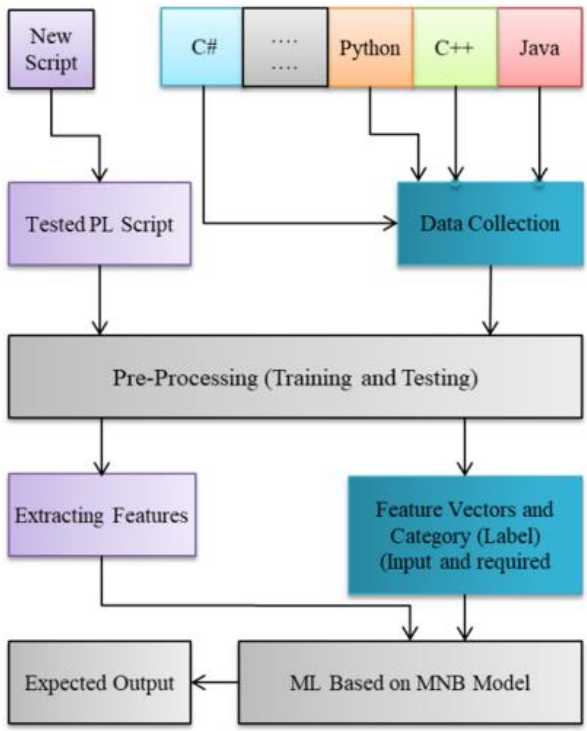

Figure 1. The general structure of the proposed model

3.1 Data collection

This module uses file samples written in 12 different PL. The first step involved cleaning each sample by removing unnecessary spaces and certain unwanted characters. Subsequently, the cleaned source code files were combined into a single array using a Python library known as "numpy," which is a highly significant multidimensional array object [28]. Furthermore, the data corpus comprising all text files was prepared, with the source of programming language text files primarily obtained from GitHub, along with additional files randomly collected from our own projects and those of students. The dataset utilized to train and test the proposed model was compiled from a collection of source code samples written in 12 programming languages, including Java, C++, C#, Python, HTML, React, XML, PHP, JavaScript, Perl, SQL, and CSS. The main components of used data set are shown in Table 2.

Table 2. Dataset components

Dataset Components	Value
No Samples	12,003
LOC	396,090
No of PL	12

3.2 Pre-processing

In this step, we will create char bigrams [29] as a vector of features for the proposed languages. Each language will be represented using a token consisting of two letters, as presented in Table 3. Two-letter character bigrams (also known as 2-grams) play a significant role in NLP tasks due to their ability to capture certain linguistic, typographical patterns, morphological analysis, named entity recognition, and contextual patterns.

Table 3. PL token

No	PL Name	Token
1	Java	ja
2	Python	py
3	C#	cs
4	C++	cp
5	HTML	ht
6	Php	ph
7	JavaScript	js
8	React	re
9	XML	xm
10	CSS	ss
11	Perl	pl
12	SQL	sq

3.3 Feature vectors and category (label) input and required output

The categorization of the script involves assigning a label for each programming language (PL) using a token consisting of two letters. This label will be used to represent the programming language. Additionally, the bigram preparation process will be carried out. Figure 2 provides an example illustrating the conversion of Java source code to a bigram representation through two steps. In preparation for creating the Bigram chart, it is necessary to define various options for the API used to generate the Bigram. These options include the Bigram unit, which can be set as Words or Letters, as well as options for sentence handling and other cleaning choices, as illustrated in Table 4. Notably, stop words are not utilized in this process. The proposed model employs bigrams through the utilization of a Python library function called "CountVectorizer." This function facilitates the conversion of text, specifically source codes, into a matrix or array of token counts, allowing for further analysis and processing.

In two steps, the sample from the first line in the mentioned code example (Figure 2) will be processed to produce the Bigram output. This process is illustrated in Table 5.

Table 4. Bigram options

Step	Bigram Unit	Sentence Option	Other Options
1	Word	Corpus Mode	Output in lower case, remove space, ., ?, and ()
2	Letters	Corpus Mode	

Java Example	Bigram (Step1)

import java.util.Scanner;
public class e1 {
public static void main(String[] args){
Scanner input= new Scanner(System.in);
int x;
x=input.nextInt();
if(x>0)
System.out.println("Positive");
else
System.out.println("Negative");
}
}
	importjavautilscanner;publicclass{
	publicclass{publicstaticvoidmainstring[]args{
	publicstaticvoidmainstring[]args{scanner
	scannernewscannersystemin;
	newscannersystemin;intx;
	intx;input
	inputnextint;
	nextint;if
	ifsystemoutprintln;
	systemoutprintln;else
	elsesystemoutprintln;

Figure 2. Java example with its bigram representation

Table 5. Step 2: Two char bigram

Original Text	import java.util.Scanner;
Bigram (step 1)	importjavautilscanner;
Bigram (step 2)	im, mp, po, or, rt, tj, ja, av, va, au, ut, ti, il, ls, sc, ca, an, nn, ne, er, r;

The number of letters in the English language, which is used for writing source code, amounts to 26. Additionally, there are 10 digits and 10 programming language symbols, such as (;, #, :, _, and $). Therefore, the total number of characters considered is 46. In this case, the maximum number of character bigrams can be calculated as 46 multiplied by 46, resulting in 2116. This representation or corpus can serve as a general feature for all programming languages. Furthermore, this model is not restricted by any limitations regarding the number of words it can handle. It is capable of processing new words, even if they have never been included in its training, as long as they consist of the same character bigrams.

3.4 Model training and evaluation

In preparation for training this model, the software program utilized the (sklearn) API, which was imported for that purpose. To ensure a more consistent corpus and achieve a good distribution of programming languages (PL), it is advisable to employ an extensive dataset. The distribution of PL in the prepared corpus is illustrated in Figure 3. The training dataset was compiled from various projects implemented in languages such as C++, C#, Java, JavaScript, HTML, Python, PHP, React, Perl, CSS, XML, and SQL. It consisted of 12,003 files, encompassing over 396,090 lines of code (LOC). After converting these LOC to Bigram Char, the dataset contained more than 7,838,301 two-character bigrams, as depicted in Figure 4, and Table 6 display the detailed data

used as 90% for training, and 10% for testing. Upon completing the training process, the model will yield the expected output, which can be assessed through different forms of accuracy. Notably, an overall accuracy rate of 95.09% was achieved during training, demonstrating the effectiveness of the model. Consequently, the model is now prepared for testing and ready to be utilized.

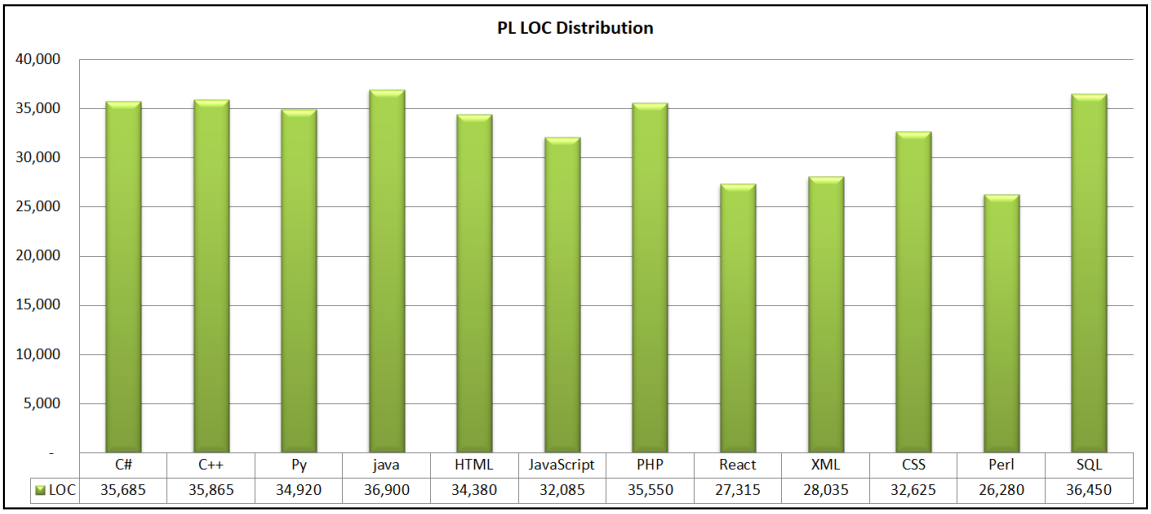

Figure 3. Distribution PL LOC in used corpus

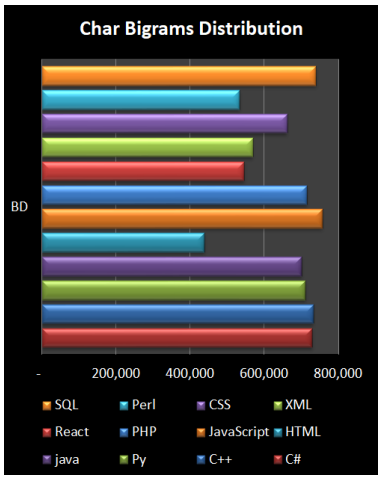

Figure 4. The char bigrams distribution

Table 6. Detailed dataset components

N	PL	LOC	BD	Samples
1	C#	35,685	729,520	1,081
2	C++	35,865	733,159	1,087
3	Py	34,920	711,090	1,058
4	java	36,900	700,972	1,118
5	HTML	34,380	439,300	1,042
6	JavaScript	32,085	757,206	972
7	PHP	35,550	715,938	1,077
8	React	27,315	546,300	828
9	XML	28,035	569,111	850
10	CSS	32,625	662,288	989
11	Perl	26,280	533,484	796
12	SQL	36,450	739,935	1,105
Total		396,090	7,838,301	12,003

3.5 Testing

The testing process comprises several steps, including reading unknown source code (tested PL Script), pre-processing, extracting features, employing Machine Learning (ML) based on MNB, obtaining the expected output, and verifying the results. To evaluate the model's performance, a set of new source codes that were not part of the training dataset was used. Totally, 1458 source code file samples were selected for testing the MNB model.

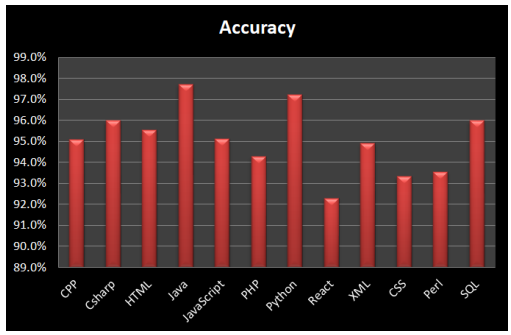

Figure 5. Percentage of correctly detected source codes in different PL

Table 7. Test results

No	PL	Accuracy	Detected	Total Tested
1	CPP	95.1%	116	122
2	Csharp	96.0%	96	100
3	HTML	95.6%	172	180
4	Java	97.7%	260	266
5	JavaScript	95.1%	78	82
6	PHP	94.3%	66	70
7	Python	97.2%	210	216
8	React	92.3%	48	52
9	XML	94.9%	112	118
10	CSS	93.3%	84	90
11	Perl	93.5%	58	62
12	SQL	96.0%	96	100
Average Accuracy		95.09%	1396	1458
Total				

These testing samples encompassed a range of 12 programming languages. The outcomes of detecting and recognizing the programming languages are presented in

Figure 5 and Table 7.

4. RESULTS

The data used to test the proposed model is consist of total number of source code using 12 Programming Languages (PL) is 1458 samples, the total number correctly detected is 1396 and the average accuracy is 95.09%, this result as test results, classification report, and confusion matrix are shown in Tables 7-9 respectively. As a summary of the results of this study we can conclude that the MNB algorithm was able to correctly classify, detect, and recognize programming language source code with an accuracy of 95.09%. It achieved high precision, recall, and F1-score for all of the programming languages, and it was able to correctly classify most of the programming language source code. The only languages that it had some difficulty with were React and Perl. This is likely because these languages are not as common as the others; samples were less than other languages, and due the JSX syntax used by React, which combines JavaScript and HTML codes within the same code sample leads to complication in tokenization and vector extraction of code features for the used MNB algorithm. Also, Perl known as highly flexible language, so the code can be written in different ways, which leads to high diversity of coding and make hard to detect by the MNB algorithm.

Table 8. Classification report

PL	Precision	Recall	F1-Score	Support	Total Tested
CPP	0.95	0.97	0.96	116	122
Csharp	0.96	0.95	0.96	96	100
HTML	0.96	0.96	0.95	172	180
Java	0.98	0.98	0.98	260	266
JavaScript	0.95	0.98	0.97	78	82
PHP	0.94	0.95	0.95	66	70
Python	0.97	0.96	0.98	210	216
React	0.92	0.93	0.93	48	52
XML	0.95	0.945	0.935	112	118
CSS	0.93	0.94	0.94	84	90
Perl	0.94	0.95	0.935	58	62
SQL	0.96	0.95	0.94	96	100
Accuracy	95.09%	95.54%	95.25%	1396	
			Total		1458

Table 9. Confusion matrix

	CPP	C#	HTML	Java	JS	PHP	Python	React	XML	CSS	Perl	SQL
CPP	113	0	0	0	0	0	0	0	0	0	0	3
C#	0	91	0	0	0	0	0	0	0	0	1	4
HTML	1	1	165	0	0	0	0	1	0	0	2	2
Java	0	0	0	255	0	0	0	0	0	0	0	5
JS	0	0	0	0	76	0	0	0	0	0	0	2
PHP	0	0	0	0	0	63	0	0	0	0	2	1
Python	0	0	0	0	1	0	202	0	0	0	0	7
React	0	0	1	0	0	0	0	44	0	1	1	1
XML	0	0	0	0	0	0	0	0	106	1	0	5
CSS	0	0	0	0	0	0	0	0	0	79	0	5
Perl	0	0	1	0	0	0	0	0	0	0	55	2
SQL	0	0	0	1	0	0	0	0	0	0	3	92

The MNB algorithm emerges as a compelling choice for classifying, detecting, and recognizing programming language source code, as indicated by the findings of this study. This simple and efficient algorithm demonstrates its ability to attain remarkable accuracy rates, even when confronted with limited training data. Nonetheless, the study also uncovers instances of misclassifications, primarily occurring between languages exhibiting similar syntax or structures.

5. DISCUSSION

The proposed methodology, utilizing the MNB algorithm and character bigrams, demonstrates its effectiveness in classifying, detecting, and recognizing programming language source code, as supported by the study's findings. With an impressive overall accuracy rate of 95.09% during training, the model proves its capability to accurately identify and classify programming languages based on source code. This achievement has significant implications for language identification, code analysis, and a language-agnostic approach in computer science and software engineering. Notably, the strengths of the proposed methodology lie in its high accuracy and the use of a diverse dataset encompassing multiple programming languages. These strengths highlight the effectiveness of the MNB algorithm and underscore the relevance of character bigrams as valuable features for language identification. However, the study acknowledges specific limitations and suggests potential areas for improvement. These limitations include a limited range of considered languages, the challenge of handling new languages and variants, reliance on character-level features without capturing higher-level semantics, and the need for additional evaluation metrics such as recall and F1 score.

In conclusion, the study presents a promising methodology for programming language identification, but further research

and enhancements are necessary to address the mentioned limitations and improve the model's applicability in diverse and evolving code environments. By demonstrating the effectiveness of the MNB algorithm and character bigrams in accurately classifying, detecting, and recognizing programming languages, the study's results contribute to the field of programming language source code classification. The achieved overall accuracy rate of 95.09% during training establishes the strength of the proposed methodology.

Comparison of this research with other research related to the same topic showed its superiority. For example, research [9] achieved an accuracy of 93.48% by using the Bayesian learning algorithms and the GitHub dataset, while research [16] reached an accuracy of 93.5% by applying deep learning on three databases that are covering eight programming languages. The results of this study, with an accuracy of 95.09%, demonstrate competitive performance and suggest that the MNB algorithm is a good choice for PL source code classification. Moreover, it acknowledges the limitations and challenges found in existing literature, such as the difficulty of classifying numerous programming languages, adapting to evolving languages, and addressing variations in code quality. These challenges emphasize the need for further research and improvements in programming language source code classification.

The achievement of this research in detecting and classifying PL addresses some of the limitations and challenges identified in existing literature. Also using the MNB algorithm and character bigrams offers a simple and effective approach to language identification, with practical implications for various applications in computer science.

6. CONCLUSION

The aim of this research was to investigate the effectiveness of MNB algorithm as important NLP techniques in classification and detection programming language from source code. It evaluated the performance of this algorithm in accurately categorizing source code and examined the impact of different feature selection methods on classification accuracy. In this research, we followed a comprehensive methodology involving data collection, pre-processing, feature extraction, model training and evaluation, and testing. During training, the MNB algorithm demonstrated an impressive accuracy rate of 98% (for Java) and maintained an average accuracy of 95.09% during the testing phase. It exhibited the ability to correctly classify programming language source code with high precision, recall, and F1-score for most languages. However, it encountered challenges with less common languages such as React and Perl. Nevertheless, it is important to acknowledge the limitations of the study. The findings may not be applicable to other programming languages and datasets due to specific selections made. The quality and size of the training dataset played a significant role in influencing the performance of the MNB algorithm. Additionally, the study solely focused on classification and did not compare the MNB algorithm with other machine learning or deep learning approaches. The effectiveness of NLP techniques and the MNB algorithm relies on the quality and consistency of the source code dataset, and the presence of poorly structured or unstructured code may impact classification accuracy. Furthermore, the study did not consider the dynamic or runtime aspects of programming languages. In summary, the study suggests that the MNB algorithm shows promise for programming language source code classification. However, further research and consideration of the identified limitations are necessary to validate its effectiveness across various programming languages and datasets.

REFERENCES

[1] Phan, A.V., Chau, P.N., Le Nguyen, M., Bui, L.T. (2018). Automatically classifying source code using tree-based approaches. Data & Knowledge Engineering, 114: 12-25. https://doi.org/10.1016/j.datak.2017.07.003

[2] Kahanwal, B. (2013). Abstraction level taxonomy of programming language frameworks. International Journal of Programming Languages and Applications, 3(4): 1-12. https://doi.org/10.5121/ijpla.2013.3401

[3] Hussain, Z., Nurminen, J.K., Mikkonen, T., Kowiel, M. (2022). Combining rule-based system and machine learning to classify semi-natural language data. In Proceedings of SAI Intelligent Systems Conference, Amsterdam, The Netherlands, pp. 424-441. https://doi.org/10.1007/978-3-031-16072-1_32

[4] Treviso, M., Lee, J.U., Ji, T., et al. (2023). Efficient methods for natural language processing: A survey. Transactions of the Association for Computational Linguistics, 11: 826-860. https://doi.org/10.1162/tacl_a_00577

[5] Heering, J., Mernik, M. (2002). Domain-specific languages for software engineering. In Proceedings of the 35th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, pp. 3649-3650. https://doi.org/10.1109/HICSS.2002.994484

[6] Steele, B., Chandler, J., Reddy, S., Steele, B., Chandler, J., Reddy, S. (2016). The multinomial naïve bayes prediction function. Algorithms for Data Science, pp. 313-342. https://doi.org/10.1007/978-3-319-45797-0_10

[7] Sharmila, A., Geethanjali, P.J.I.A. (2016). DWT based detection of epileptic seizure from EEG signals using naive Bayes and k-NN classifiers. IEEE Access, 4: 7716-7727. https://doi.org/10.1109/ACCESS.2016.2585661

[8] Shrivas, A.K., Dewangan, A.K., Ghosh, S. M. (2021). Robust text classifier for classification of spam e-mail documents with feature selection technique. Ingénierie des Systèmes d'Information, 26(5): 437-444. https://doi.org/10.18280/isi.260502

[9] Ontivero-Ortega, M., Lage-Castellanos, A., Valente, G., Goebel, R., Valdes-Sosa, M. (2017). Fast Gaussian Naïve Bayes for searchlight classification analysis. Neuroimage, 163: 471-479. https://doi.org/10.1016/j.neuroimage.2017.09.001

[10] Odeh, A.H., Odeh, M., Odeh, N. (2022). Using multinomial naive bayes machine learning method to classify, detect, and recognize programming language source code. In 2022 International Arab Conference on Information Technology (ACIT), Abu Dhabi, United Arab Emirates, pp. 1-5. https://doi.org/10.1109/ACIT57182.2022.9994117

[11] Yang, G., Zhou, Y., Yu, C., Chen, X. (2021). DeepSCC: source code classification based on fine-tuned RoBERTa. arXiv preprint arXiv:2110.00914. https://doi.org/10.48550/arXiv.2110.00914

[12] Khasnabish, J.N., Sodhi, M., Deshmukh, J.,

Srinivasaraghavan, G. (2014). Detecting programming language from source code using Bayesian learning techniques. In 10th International Conference, MLDM 2014, St. Petersburg, Russia, pp. 513-522. https://doi.org/10.1007/978-3-319-08979-9_39

[13] Gilda, S. (2017, July). Source code classification using neural networks. In 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), NakhonSiThammarat, Thailand, pp. 1-6. https://doi.org/10.1109/JCSSE.2017.8025917

[14] Nagy, A., Kovari, B. (2015). Programming language neutral design pattern detection. In 2015 16th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, pp. 215-219. https://doi.org/10.1109/CINTI.2015.7382925

[15] Van Dam, J.K., Zaytsev, V. (2016). Software language identification with natural language classifiers. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, pp. 624-628. https://doi.org/10.1109/SANER.2016.92

[16] Rahman, M.M., Watanobe, Y., Nakamura, K. (2020). Source code assessment and classification based on estimated error probability using attentive LSTM language model and its application in programming education. Applied Sciences, 10(8): 2973. https://doi.org/10.3390/app10082973

[17] Guesslang documentation. https://guesslang.readthedocs.io/en/latest/contents.html#deep-learning-model.

[18] Chrislo/Sourceclassifier: Use a Bayesian classifier to determine source code language. https://github.com/chrislo/sourceclassifier.

[19] "SyntaxHighlighter Evolved – WordPress plugin | Wordpress.org." https://wordpress.org/plugins/syntaxhighlighter/.

[20] Alvares, M., Marwala, T., de Lima Neto, F.B. (2014). Application of computational intelligence for source code classification. In 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing, China, pp. 895-902. https://doi.org/10.1109/CEC.2014.6900300

[21] Kiyak, E.O., Cengiz, A.B., Birant, K.U., Birant, D. (2020). Comparison of image-based and text-based source code classification using deep learning. SN Computer Science, 1(5): 266. https://doi.org/10.1007/s42979-020-00281-1

[22] El Atillah, M., El Fazazy, K. (2020). Recognition of intrusive alphabets to the Arabic language using a deep morphological gradient. Revue d'Intelligence Artificielle, 34(3): 277-284. https://doi.org/10.18280/ria.340305

[23] Allamanis, M., Brockschmidt, M., Khademi, M. (2017). Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740. https://doi.org/10.48550/arXiv.1711.00740

[24] Raychev, V., Bielik, P., Vechev, M., Krause, A. (2016). Learning programs from noisy data. ACM Sigplan Notices, 51(1): 761-774. https://doi.org/10.1145/2837614.2837671

[25] Hellendoorn, V.J., Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, pp. 763-773. https://doi.org/10.1145/3106237.3106290

[26] Nye, M., Hewitt, L., Tenenbaum, J., Solar-Lezama, A. (2019). Learning to infer program sketches. PMLR, 97: 4861-4870.

[27] Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., Sarro, F. (2021). A survey on machine learning techniques for source code analysis. arXiv preprint arXiv:2110.09610. https://doi.org/10.48550/arXiv.2110.09610

[28] What is NumPy. https://numpy.org/doc/stable/user/whatisnumpy.html.

[29] Johnson, D., Malhotra, V., Vamplew, P. (2006). More effective web search using bigrams and trigrams. Webology, 3(4): 34.