Predicting Code Runtime Complexity Using ML Techniques

C. V. Deepa Shree, Jaaswin D. Kotian, Nidhi Gupta, Nikhil M. Adyapak, and U. Ananthanagu

Abstract There are several approaches to solving every coding algorithm in Computer Science. To achieve the same result, these methods may use various techniques and reasoning. The difficulty is that as the number of inputs increases, certain algorithms tend to perform poorly. Several metrics may be used to assess the quality of any code. The code runtime complexity is one of these measurements. To determine this runtime complexity, substantial study and a thorough understanding of algorithms are necessary, which is a challenging manual undertaking. In this study, the worst-case runtime complexity of codes in programming languages C, Java and Python are calculated as Big-O notations utilising code features like Abstract Syntax Trees, ML approaches and static code analysis. The novelty of the research is our labelled runtime complexity dataset which was constructed manually, implementing Deep Learning Algorithms like Bi-LSTM and calculating Code Runtime Complexity for the worst-case scenario as Big-O notations for codes in three languages, C, Java and Python. To predict the runtime complexity for a given code more accurately than the traditional legacy methods such as manually asserting code runtime complexity or running code for different amounts of input, we have presented a more effective manner for the same. The results portray that the XGBoost classifier outperforms the other models with an accuracy of 96%. The current study can also be extended to other high-level programming languages, including more training samples and making use of graph neural networks.

C. V. Deepa Shree · J. D. Kotian · N. Gupta (✉) · N. M. Adyapak · U. Ananthanagu
Department of Computer Science Engineering, PES University, Bengaluru, India
e-mail: nidhiguptx7@gmail.com

C. V. Deepa Shree
e-mail: chalapathicvenkata@gmail.com

J. D. Kotian
e-mail: kotianjashu09@gmail.com

N. M. Adyapak
e-mail: nikhiladyapak31@gmail.com

U. Ananthanagu
e-mail: ananthanagu@pes.edu

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

N. R. Shetty et al. (eds.), *Advances in Computing and Information*, Lecture Notes in Electrical Engineering 1104, https://doi.org/10.1007/978-981-99-7622-5_26

Keywords Runtime complexity · Static analysis · Abstract Syntax Tree · LSTM · AST · graph2vec · Bi-LSTM

1 Introduction

Efficient code is the need of the hour. A lot of factors determine the efficiency of a program. The execution time of the code is one of them. The execution time of a code is the measure of time taken for a program to complete running without any error. An important observation is that the actual time taken for program execution is machine-dependent. It depends on several factors like parallelism, processor utilisation, CPU overhead and so on. Instead of measuring the actual time taken for completion of program execution, time complexity labels are used. This is not exactly equal to the exact time taken for running a code snippet but is a quantification of the same as a function of the input length.

The process of expressing execution time based on the input size is also called asymptotic analysis. There are several asymptotic notations to express time complexity. These are Big-O, Omega and Theta. Big-O is also known as the upper bound of execution time. Omega represents the lower bound. Theta notation constitutes both the upper and lower boundaries. It is also known as the tight bound. In our work, we have used the Big-O notation to determine the time complexity.

Finding the time complexity of a code has a lot of applications. It can be used in online coding platforms to help users improve and eventually that the optimal solution. Having an idea of the algorithm's time complexity can help developers write better code. Most of the existing solutions online do not support static analysis. The user is required to design various test cases of hugely varying input sizes. The corresponding Big-O notation is then given as the output, based on the total execution time of the algorithm. Big-O Calculator is a library that supports the Big-O notation calculation on the Coderbyte platform. "Big-O" is a Python package restricted to Python programs. This process is time-consuming and tedious.

A few other solutions which involve static analysis have a lot of limitations. For instance, [1] uses text parsing and works only for programs with loops. A few other standard procedures such as the master's theorem and Recurrence Tree work only for recursion-based algorithms. These must be done manually and are time-consuming. Hence, an automated solution is required.

We aim to solve the problem at hand using machine learning (ML) and deep learning (DL) techniques. The dataset is collected from various sources available on the internet. As a part of the approach to solving the problem, the code is converted into an Abstract Syntax Tree (AST) to get graph embeddings rather than directly obtaining the word embeddings from the code.

The nodes from the AST are extracted, and a directed graph is built. This graph is then converted into a graph embedding using Graph2Vec. We have converted the problem at hand into a graph classification problem; hence, we use classification-based ML and DL algorithms.

2 Related Work

Sikka et al. [2] have presented an approach where ASTs are used to extract hand-engineered features and code embeddings from the code snippet. The authors have built a new dataset called CoRCoD, consisting of 933 Java codes. They have used various ML algorithms and SVM with 1024-dimensional code embeddings using graph2vec gave the highest accuracy amongst all the models. The authors have used only Java codes, and no other languages have been used. Agenis-Nevers et al. [3] have presented GuessCompx, an R package which does empirical estimation on both the space and time complexity of the given algorithm. The package takes inputs of multiple increasing-size test samples and tries to fit the best complexity label to it based on running time. This process is tedious as the user must design various samples of various sizes. Hutter et al. [4] have proposed a method of using regression machine learning techniques to predict the actual runtime of the program. All the runtimes have been recorded using the same environment as CPU configuration, etc. This setup is not efficient because the same factors must be used if the model results are to be reliable. Haridas et al. [5] worked on the representation of C/C++ programs as graphs. They formulated a unique method that utilised a neural tensor network to combine results from the GNN and time capsules. This fusion helped them to improve the accuracy of the model which predicted the similarity of a given software code to a set of codes. Although the accuracy of the similarity value was high, the trade-off was the runtime. Gao et al. [6] worked on the prediction of the runtime performance of DL models and neural architecture search algorithms that utilised GNNs and a novel approach DNNPerf (a tool). Their model accepted a DL model file, a model configuration specification and a runtime specification as input, using which it reported the runtime performance values as the output. For the models that implemented proprietary NVIDIA, CUDA, etc., their internal implementation details were hidden which made it difficult to arrive at the exact runtime performance metric. Chen et al. [7] worked on efficiently capturing code semantics with the efficient API-based AST. Zhang et al. [8] worked on a novel way to get the huge ASTs into memory by splitting them block-wise and processing them with the help of neural networks. Lin et al. [16] worked on the block-wise splitting of code into different sections and then processing them. A combination of these techniques [8, 9] can be implemented in our study if the size of ASTs generated becomes exceptionally large, and not sufficient to fit into memory. A tool called "J-CEL" [10] attempts to show the Big-O notations as graphs for a visual representation of Java codes. Code clone detection has been performed using RNNs that take in embeddings using Siamese networks and LSTMs [11] and flow-augmented ASTs [12] for comparing the similarity of codes. This research was useful in our research for generating ASTs in C and Java, respectively. Our study was supposed to be based on GNNs which we plan for future work and a study by Feng et al. [13] worked to predict vulnerabilities in functions of programs using GNNs. Reza et al. [14] worked on predicting the complexity of codes in the manner of lines of codes, depth of inheritance tree, object coupling, etc., using ML techniques. This research was studied in detail due to its similarity

with our study. Guzman et al. [15] worked on a test bench of 30 Python programs that were tested for time complexity correctness using Big Theta time complexity approximation.

3 Data Collection and Labelling

The dataset contains a large collection of code samples taken from various sites such as geeks for geeks, tutorialspoint, etc. These websites contained sample codes for various data structures and algorithms where they were either labelled with the respective runtime complexity or they were not. If the runtime complexity was not labelled for a respective code sample, it was analysed by two people before assigning it a label.

"AProVE" [16] is a tool implemented to calculate runtime complexities of Java codes and Jar files. This tool was initially explored to check if codes could be labelled. Due to the limited functionality and prerequisite style of input to be given in the form of certain parameters, the codes scraped to be our dataset could not be labelled by this tool.

Further research was conducted into the Termination and Complexity Calculation competition to obtain tools to label our dataset. Since most of the tools were not open source and had a lot of dependencies, we chose to manually label our dataset.

The dataset contains a total of 10 runtime complexity labels, each code divided based on the language they belong to (C, Java, or Python). The dataset currently contains 769 codes with 41.22% Java, 42.78% Python and 15.99% C codes. The codes included belong to data structures such as Arrays, Stacks, Queues, Linked List, Trees and Graphs. The codes have the respective contributors' names in them wherever it was mentioned. Each code sample is a single file which includes the source website of the code, the code and the contributor's name. Using this method helped to prevent duplication of codes while assigning the codes to their respective time complexity label.

The 10 different complexity classes represented as Big-O notations used in this study are $O(1)$, $O(N)$, $O(N^2)$, $O(N^3)$, $O(\log(N))$, $O(N \log(N))$, $O(N * d)$, $O(2^n)$, $O(N!)$ (N factorial) and $O(\text{sqrt}(N)$ (square root of N) for C, Java and Python as shown in Fig. 1. The entire workflow of the study is shown in Fig. 2.

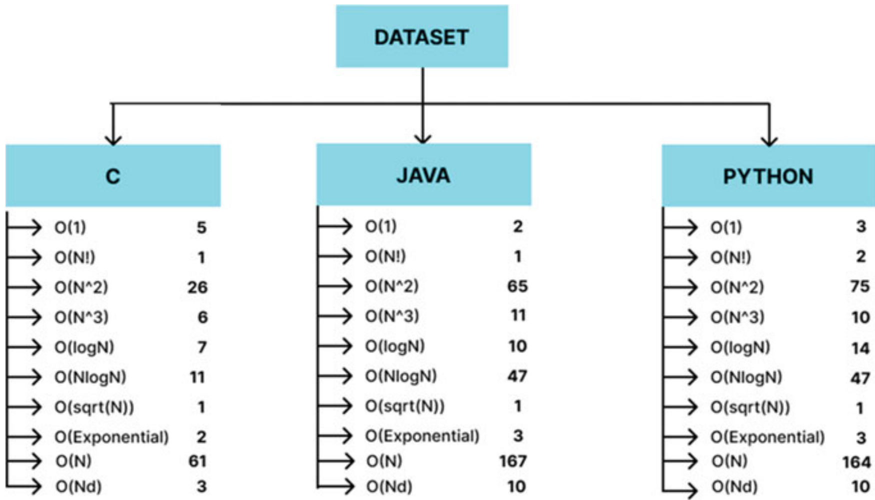


Fig. 1 Number of code snippets for each label, programming language-wise

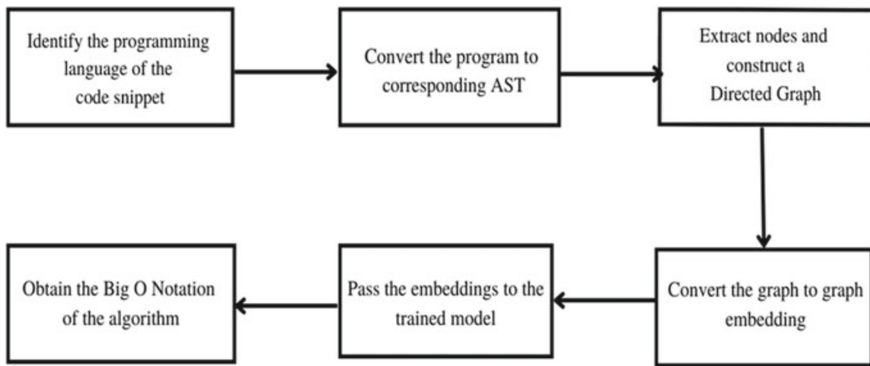


Fig. 2 Workflow of the proposed approach

4 Proposed Approach

4.1 Programming Language Identification

For the study, three high-level programming languages were used, C, Java and Python. These programs can be identified by either implementing Github's language identifier or a python package called "Guesslang." The language codes can also be identified by their extensions. It is important to classify the codes by the programming language because the packages for pre-processing and AST generation are different for each language.

4.2 AST Generation

An Abstract Syntax Tree is a representation of the program syntax in the form of a tree. This does not represent every detail of the code. It contains only content-related information. After the language was identified, the corresponding Python package was used to obtain the AST representation. ASTs for C codes were obtained using “pycparser.” This is a Python package which accepts C code in the form of a file or a string and generates the corresponding AST. The C code was first compiled by including the fake headers from the “pycparser” repository. This compiled code was then used to generate the AST. The typedef nodes and their children were removed from the tree as they do not contribute to determining the time complexity of the code snippet. Similarly, for Java codes, “javalang” was used to construct the AST and for Python, the AST module was used to get the tree. Each of these AST representations was traversed, and all the nodes were obtained. While obtaining the nodes, these were indexed accordingly considering the current parent of the node, along with the addition of a random number, so each node has a unique index. This is a requirement to construct graphs in the next step of the solution. Figures 3, 4 and 5 show the AST representations produced by the python libraries for all three programming languages for a simple program to display “Hello World.” Each node contains a unique number as the index and the value of the node.

```
// Simple C program to display "Hello World

// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins
int main()
{

        // prints hello world
        printf("Hello World");

        return 0;
```

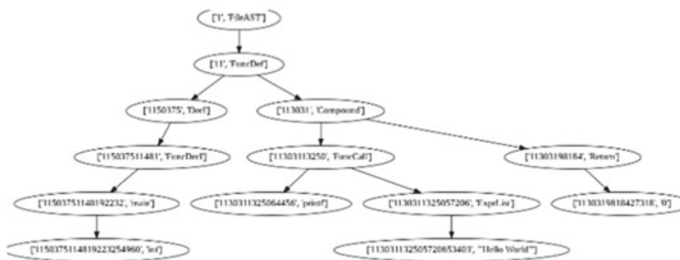


Fig. 3 A simple C program to print Hello World and its corresponding AST (without typedef nodes and their children

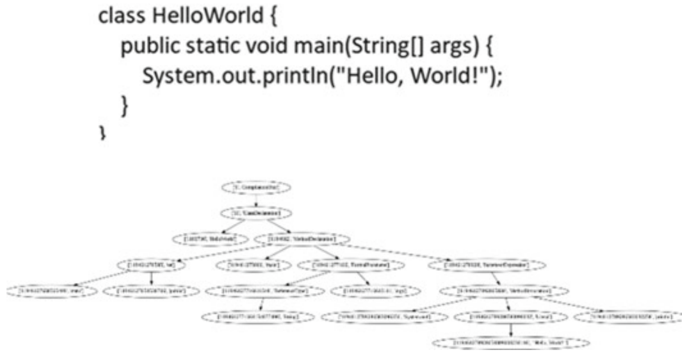
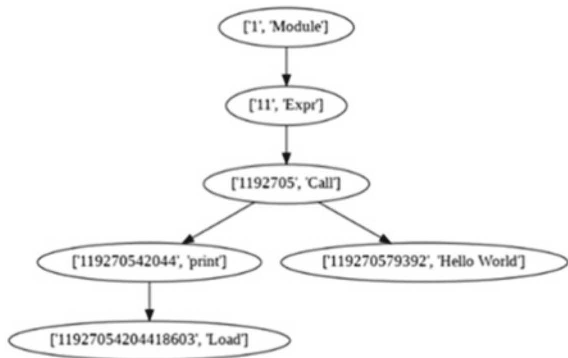


Fig. 4 A simple Java program to print Hello World and its corresponding AST

Fig. 5 A simple Python program to print Hello World and its corresponding AST

```
# python program to print "Hello World"
print("Hello World")
```



4.3 Construction of Directed Graph and Graph Embeddings

The extracted nodes and their features were then used to construct a directed graph using the "network" library. Each node has a unique index and value. The value is the class name for a few nodes and the value of the class along with the former for a few nodes. These were assigned as node attributes and the attribute was named "feature." These graphs were then passed as input to the graph2vec algorithm to obtain a graph embedding. These embeddings are NumPy arrays of shape (128,) for each graph. This algorithm was implemented using the assistance of the "karateclub" [17] library in Python. graph2vec library contains the Python implementation of [18]. Narayanan et al. [18] state that graph2vec produces embeddings that are task agnostic. This made the current algorithm more favourable than the others. The embeddings are numpy arrays that are further used to train and test the classification models.

4.4 Data Pre-processing

The outputs from the pre-processing steps, the AST to graph building to embeddings resulted in a set of vector embeddings stored and processed in the form of numpy arrays which is the independent variable, "*X*." The code names, complexity and the corresponding time complexity label were also stored. From this the time, complexity label was extracted and made the dependent variable, "*Y*." This data was then fed as the input to the classification models. To test out the data processed so far without any analysis, initially, these "*X*" and "*Y*" were passed to the Random Forest multi-class classifier ML model and the confusion matrix was plotted resulting in an accuracy of 60%. Analysis of the dataset was performed, visualising the skewness of the dataset, shown in Fig. 6, due to an imbalance in the number of code samples present in each code complexity. This imbalance was taken care of by two different techniques, resampling and SMOTE. Combining the "*X*" and "*Y*," we obtained a data frame that has 128 columns for embeddings and an extra column for the complexity class label. This Pandas data frame was used to train and test the model.

Resampling is the procedure to reproduce samples in minority classes to make up the majority class, known as upsampling or to cut down on the number of samples to make it into a common lower threshold known as downsampling. SMOTE is the technique used to intelligently find out which features contribute positively when the samples were duplicated and generate synthetic data to cover for the imbalance in data samples of classes. The dataset being imbalanced as shown in Fig. 6, the dataset had to be balanced before proceeding further. By experimenting with a combination of these techniques, the upsampling version of resampling is used in this study.

Getting the dataset to be balanced as shown in Fig. 7 was trained on different ML models such as Random Forest, AdaBoost, XGBoost, KNN, Logistic Regression

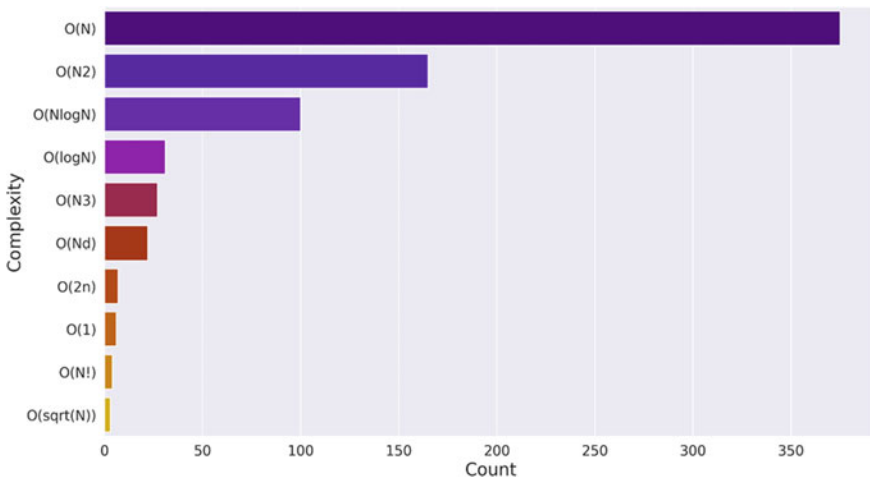


Fig. 6 Visual representation of skewness of the dataset

Fig. 7 Dataset distribution for Expt-1

Experiment	Target Class	Before Resampling	After Resampling
Exclusive of Language for 10 Classes (Expt-1)	O(1)	6	375
	O(N)	375	375
	O(LogN)	31	375
	O(NLogN)	100	375
	O(N!)	4	375
	O(N^2)	165	375
	O(N^3)	27	375
	O(Nd)	22	375
	O(sqrt(N))	3	375
	O(2^N)	7	375

and Naive Bayes. In this study, a permutation and combination of experiments were performed to analyse the performance of the models. A technique called feature importance was performed. This technique helps us get to know the most important features from "*X*" that are contributing positively towards the results. For each code in the dataset, 128 features are the vector values in each numpy array. All the features contribute differently towards the model prediction. By using "SHAP" [19], a library in Python and some packages in the Random Forest model to intelligently tell the feature importance, the top 20 features can be obtained as shown in Fig. 9. Using the top features, the models are retrained, and the results are recomputed.

As part of the pre-processing stage, another approach was built and used. Here, all the classes with less than 10 samples in each class were removed. This resulted in a total of 6 classes in each language, respectively. The dataset was initially divided into subsets of their respective languages. Resampling was performed on each of the subsets based on classes which had the maximum frequency. The resultant subsets gave 149 samples each for the python subsets, 165 samples each for the Java subsets and 61 samples each for the C subsets as shown in Fig. 8. These subsets were then combined to create the final dataset which was then used for model building and training.

In another approach, the programming language was combined with the time complexity as the target label. This approach was followed as the node labels for each language are different. Before proceeding with this approach, classes with less than 10 samples for each class were removed, as these labels were not available for all languages. This resulted in a total of 18 classes as shown in Fig. 10. SMOTE [20] was used to resample the dataset. This resulted in 165 samples for each class, which is a total of 2970 samples. This dataset was then used to train the chosen classification models as stated earlier.

Experiment	Language	Target Class	Before Resampling	After Resampling
Inclusive of Language for 6 Classes (Expt-2)	Python	O(N)	6	149
		O(LogN)	375	149
		O(NLogN)	31	149
		O(N^2)	100	149
		O(N^3)	4	149
		O(Nd)	165	149
	Java	O(N)	6	165
		O(LogN)	375	165
		O(NLogN)	31	165
		O(N^2)	100	165
		O(N^3)	4	165
		O(Nd)	165	165
	C	O(N)	6	61
		O(LogN)	375	61
		O(NLogN)	31	61
		O(N^2)	100	61
		O(N^3)	4	61
		O(Nd)	165	61

Fig. 8 Dataset distribution for Expt-2

4.5 Model Analysis and Building

4.5.1 Bi-LSTM

Many times, a reference is required to certain data which was stored previously to predict the present output. RNNs are not capable of handling such long-term dependencies as there is no control over which parts of the data need to be remembered and which ones must be forgotten to make future predictions accurately. To overcome this problem, we chose to use a bi-directional LSTM. The input flows in two directions, making the Bi-LSTM different from the regular LSTM. With the regular LSTM, we can make input flow in one direction, either backwards or forwards. However, in bi-directional LSTM, we can make the input flow in both directions which helps consider both the future and the past information. Bi-LSTM is usually employed where sequence-to-sequence tasks are needed.

In this study, we re-sized the graph embeddings from 2 to 3D, as the Bi-LSTM expects 3D data. The input shape for the embeddings passed was 128×1. Hence, the first Bi-LSTM layer was allotted a total of 64 memory units. 64 as the input shape was 128 which suggested the sum of forwards (64) and backwards (64) should be equal to 128. We have a total of 10 classes to be predicted hence we used the SoftMax activation function. Finally, because this is a classification problem where the data is sparse, the sparse log loss (sparse_categorical_crossentropy in Keras) was used. The efficient ADAM optimisation algorithm was used to find the weights, and the accuracy metric was calculated and reported for each epoch. Figure 11 shows the different Bi-LSTM architectures for the models in different experiments.

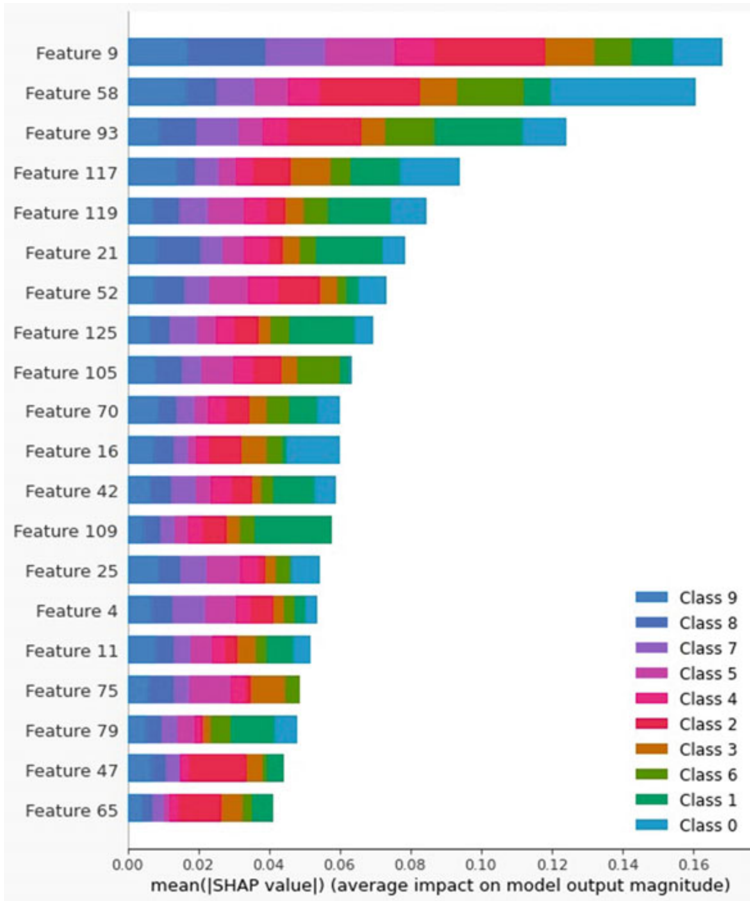


Fig. 9 Sorting the embedding columns (features) by importance

4.5.2 Random Forest Classifier

This is a supervised learning and ensemble model. It makes use of the bagging method to group the decision trees. It uses ensemble learning, which is a learning technique for enhancing the model's performance by combining numerous classifiers to solve a complicated issue. Random Forest is diverse since it does not account for all the attributes and features while building each tree. Each tree has its data and features and thus makes complete usage of the CPU while building Random Forests. Also, there is no need of splitting the data into training and testing sets, since there is always 25% of data that is unseen by the classifier.

Experiment	Target Class	Before Resampling	After Resampling
Inclusive of language for 18 classes(Expt-3)	Java_O(N)	165	165
	Python O(N)	149	165
	Python O(N^2)	74	165
	Java_O(N^2)	65	165
	C_O(N)	61	165
	Java_O(NlogN)	47	165
	Python_O(NlogN)	42	165
	C_O(N^2)	26	165
	Python_O(logN)	14	165
	Java_O(N^3)	11	165
	C_O(NlogN)	11	165
	Java_O(logN)	10	165
	Python_O(N^3)	10	165
	Python_O(N*d)	10	165
	Java_O(N*d)	9	165
	C_O(logN)	7	165
	C_O(N^3)	6	165
	C_O(N*d)	3	165

Fig. 10 Dataset distribution for Expt-3

4.5.3 K-Nearest Neighbour

K-nearest neighbour is a supervised machine learning model which is used widely. This method makes a key assumption that the unseen data and neighbour are related and places this new data in the class that is very alike amongst the existing classes. This means that any unknown data points can be easily classified into one of the existing categories, using some distance measure to estimate the similarity between the test sample and the target classes to make the classification decision. Euclidean distance is the most widely used distance measure to find the nearest neighbours of each query point and we have used it as a metric as part of this research work while training the KNN model for the classification task.

4.5.4 XGBoost

XGBoost stands for Extreme Gradient Boosting. It is an implementation of the gradient boosting-based decision tree, which is an ensemble learner. The key idea of this algorithm is that each predictor corrects its predecessor's error. A variety of hyperparameters are included in the XGBoost implementation. Tuning these hyperparameters can improve results depending on the job at hand. In our research work, we have used the default hyperparameters for XGBoost as provided by the Scikit-Learn module, since the model has performed well during both training and testing.

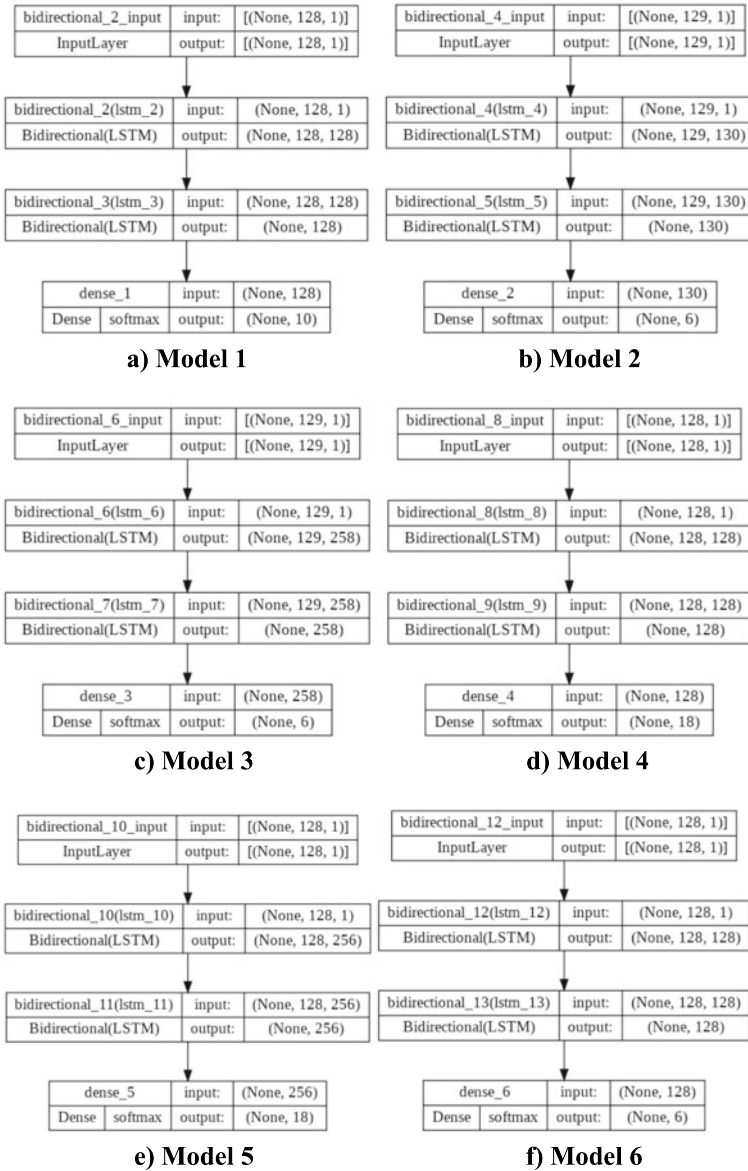


Fig. 11 Bi-LTSM architecture for Models 1–6

4.5.5 AdaBoost

AdaBoost is an abbreviation for adaptive boosting, which is one of the most popular ensemble learners and is mostly used with decision trees. This algorithm builds a learner and assigns equal weights to all the data points initially and eventually assigns higher weights to the samples after each iteration, such that it gives more importance to the higher weights in the next model. This process is continued until a lower error is received. In our research work, we have used the default hyperparameters for AdaBoost as provided by the Scikit-Learn module, since the model has performed well during both training and testing.

4.5.6 Logistic Regression

Multinomial logistic regression is a logistic regression extension that includes native support for multi-class classification issues. Logistic regression is restricted to two-class classification tasks. Some extensions, such as one-vs-rest, can be utilised for multi-class classification issues, but they require that the classification problem be first turned into several binary classification problems. To accommodate multi-class classification issues, the multinomial logistic regression technique is a modification to the logistic regression model that requires altering the loss function to cross-entropy loss and the predicted probability distribution to a multinomial probability distribution.

4.5.7 Naive Bayes Classifier

Naive Bayes is a probabilistic machine learning model that is used for classification tasks. The principle of this classifier is based on the Bayes theorem. It is a supervised learning algorithm which is used in text classification that includes a high-dimensional training dataset. This classification algorithm is used in building fast machine learning models that can make quick predictions. It is a probabilistic classifier, which means it predicts the probability of an object. Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods.

5 Metrics Used

1. *Accuracy*: Accuracy is one of the most important evaluation metrics when it comes to performance evaluation. It tells us how well the trained model has performed against the test data and is more useful when all the target classes have the same gravity. It is defined as the ratio of the count of true predictions to the count of all the predictions in the dataset.

2. *Precision*: Precision is an evaluation metric that computes a model's accuracy in classifying a test record as positive. It is computed as the ratio of the count of True positives in the dataset to the count of all the positive specimens in the dataset.
3. *Recall*: Recall quantifies the ability of the model to find a positive specimen. The higher the recall gets, the more positive the tests are being detected. This metric focuses only on how the positive records are being classified and are independent of the negative specimens in the dataset.
4. *F-measure*: Individually in some cases, neither precision nor recall gives the required insight into a model's performance and that is where F-measure comes in handy. F-measure gives us a single score that handles the problems of both precision and recall.
5. *Kappa Statistics*: Kappa score is used to evaluate the performance of model classification. It is used to measure the degree of agreement amongst two judges and is popularly referred to as inter-rater reliability.
6. *AUC Score*: AUC stands for area under the curve and is computed using Simpson's classifier. The higher the AUC score the better the classifier performs. The Y-axis refers to the True Positive Rate (TPR) and the X-axis refers to the False Positive Rate (FPR).

6 Experimental Results and Analysis

A permutation and combination of techniques were used to arrive at different results and evaluation metrics. Initially, using the imbalanced dataset in Fig. 6, and running the Random Forest model, an accuracy of 59.45% was achieved. For the same dataset, removing complexity class codes which had less than 10 samples, which were codes in $O(2^n)$, $O(1)$, $O(N!)$ (N factorial) and O(sqrt(N) (square root of N) were removed and retrained on the same model resulting in an accuracy of 58.89%. Since there was an imbalance in the dataset, the dataset had to be balanced. By using the resampling technique of upsampling, different experiments were performed. By taking only two majority classes, O(N) and O(N^2) as binary classification, an accuracy of 88.82% was achieved. Taking the four majority classes, $O(N)$, $O(N^2)$, $O(N \log(N))$ and $O(\log(N))$ and running them on the same configurations, an accuracy of 89.86% was achieved. Removing the class codes with less than 10 samples, an accuracy of 94.49% was achieved. Taking all 10 complexity classes and testing on the same configurations, an accuracy of 95.73% was achieved.Balancing the dataset by SMOTE, taking two classes, $O(N)$ and $O(N^2)$ as binary classification, an accuracy of 97.34% was achieved. Balancing the dataset by the resampling technique of upsampling, performing feature selection and retraining on the Random Forest model, an accuracy of 96.16% was achieved. To obtain the best results, a set of ML models used for training were Random Forest, AdaBoost, XGBoost, KNN, Logistic Regression and Naive Bayes. The metrics computed for these sets of models are Accuracy, Precision, Recall, F1 score, Cohen Kappa Score and ROC AUC Score as

shown in Experiment-1, Fig. 12, and the visual representation of results is obtained in Fig. 13a. The Random Forest classifier was initially used, and it gave us an accuracy of over 90% which was termed significantly good with the kind of data that was available. This is because Random Forest reduces the overfitting problem in decision trees which reduces the variance which in turn improves the accuracy. Hyperparameter tuning was performed on the Random Forest classifier to improve the evaluation metrics. This included the number of decision trees being used and the max features that will be used in the classifier. Grid Search, a hyperparameter tuning strategy, was used to fine-tune the model and provide the best parameters the model could run on. This strategy improved the accuracy to 91.83%. The assumption that Random Forest gave a good accuracy was because it adds additional randomness to the model while growing trees. It also searches for the best feature amongst a random subset of features instead of looking for the most important feature, when splitting a node. The pre-processed data were then tried on boosting algorithms like AdaBoost and XGBoost which gave an accuracy of 86% and 92% respectively. The algorithm helps in the conversion of weak learners into strong learners by combining n number of learners. Boosting also can improve model predictions for learning algorithms. We have used the default hyperparameters for AdaBoost as provided by the Scikit-Learn module since the model has performed well during both training and testing with different variations of data. The data for all the experiments are shown in Fig. 12.

The KNN algorithm gave an accuracy of 78% on the same configured data. We use the built-in library from Scikit-Learn to train our KNN model. We split the input and output data into train and test sets to train the model and test the model's accuracy on testing data. The Euclidean distance was used as the distance metric here. KNN assumes that if a datapoint is close to another datapoint, then they belong to similar classes. One of the reasons why KNN had lower accuracy in contrast to other algorithms is due to its inability to work with high-dimensionality data as it complicates the distance calculating process. Another reason could be feature scaling where the data in all dimensions need to be scaled properly.Logistic Regression and Naive Bayes were the next set of models that were used to analyse the behaviour of the data. The classifiers gave an accuracy of 83% and 35% respectively. In logistic regression, the parameter's random state was set to 0 and multiclass was set to multinomial to perform better. One of the reasons for Naive Bayes performing badly might be due to the bad binning of continuous variables with multinomial Naive Bayes. Figures 14, 15 and 16 are the confusion matrices for the different experiments. The same models were tried for the approach where the programming language was combined with the time complexity as the output label. Experiment 3 in Fig. 12 and the visual representation of results obtained in Fig. 13c show the various performance metrics obtained by the mentioned ML algorithms for this approach. The accuracy achieved by this model was around 93% using Random Forest and around 94% for XGBoost. Figure 18 shows the confusion matrix obtained for XGBoost. The study was carried out on six different versions of the Bi-LSTM models. In the first model version, resampling was done on the dataset which made the runtime complexity labels equal in number. A total of 10 complexity labels were obtained which were

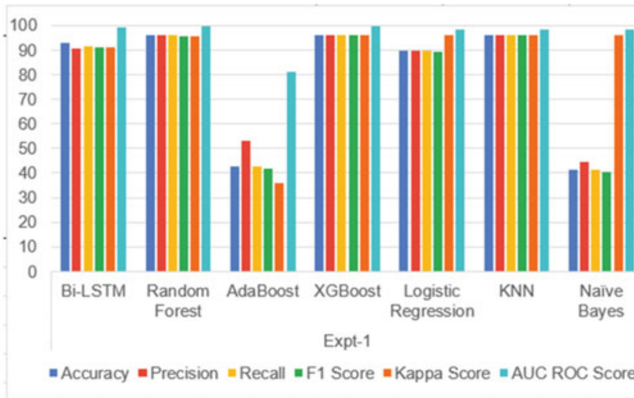
Experiment	Classifiers	Accuracy	Precision	Recall	F1 Score	Kappa Score	AUC ROC Score
Exclusive of Language for 10 Classes (Expt-1)	Bi-LSTM	92.71	90.80	91.36	90.93	90.93	99.02
	Random Forest	95.96	96.10	95.96	95.73	95.51	99.74
	AdaBoost	42.49	52.89	42.49	41.78	35.88	80.93
	XGBoost	**96.26**	**96.25**	**96.26**	**96.15**	**95.85**	**99.76**
	Logistic Regression	89.71	89.65	89.71	89.48	95.85	98.19
	KNN	96.26	96.25	96.26	96.15	95.85	98.19
	Naïve Bayes	41.20	44.48	41.20	40.40	95.85	98.19
Inclusive of Language for 6 Classes (Expt-2)	Bi-LSTM	88.50	80.42	81.11	80.56	7739	96.16
	Random Forest	91.83	91.85	91.82	91.69	90.16	98.69
	AdaBoost	86.67	86.73	86.67	86.66	45.49	84.71
	XGBoost	**92.90**	**92.86**	**92.89**	**92.81**	**91.45**	**98.74**
	Logistic Regression	83.12	82.98	93.12	82.64	79.76	94.70
	KNN	78.68	77.41	78.68	77.36	74.45	95.50
	Naïve Bayes	35.16	45.75	35.16	31.72	22.97	91.34
Inclusive of Language-II for 6 Classes (Expt-3)	Bi-LSTM	91.96	84.74	84.65	84.35	83.37	99.13
	Random Forest	93.27	93.33	93.27	93.14	92.86	99.45
	AdaBoost	48.85	64.22	48.85	47.83	45.80	95.11
	XGBoost	**94.48**	**94.55**	**94.48**	**94.44**	**94.15**	**99.81**
	Logistic Regression	91.25	91.32	91.25	90.86	90.72	99.45
	KNN	88.86	89.14	88.96	88.29	88.30	98.60
	Naïve Bayes	55.98	59.39	55.98	52.83	53.47	95.45

Fig. 12 ML model metrics and results

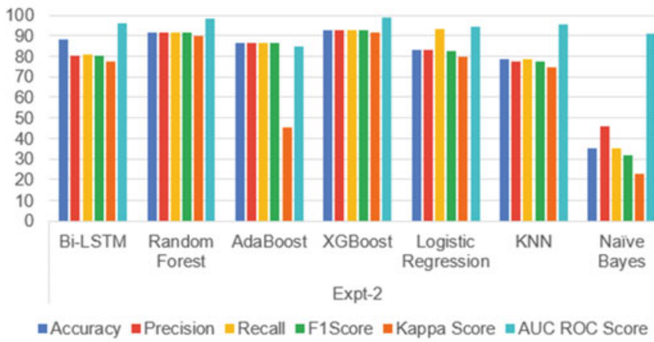
then trained on a Bi-LSTM model with 64 memory units for a total of 25 epochs. The loss and accuracy found are shown in Figs. 17 and 18a.

In the second model version, the embeddings were combined with the respective language used and the runtime labels and embeddings were removed for the labels which consisted of fewer than three codes. We then performed resampling on the dataset, and a total of six complexity labels were obtained. This was trained on a Bi-LSTM model with 65 memory units for a total of 50 epochs. The results are shown in Fig. 18b.

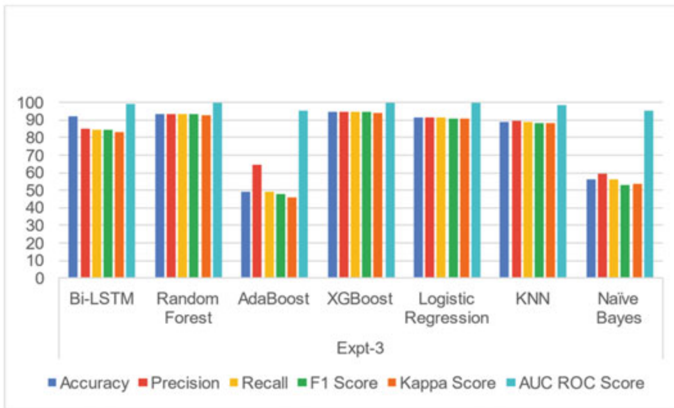
In the third model version, similar steps were repeated as in the second model. The only difference was that the Bi-LSTM was trained on 129 memory units. It was



(a)



(b)



(c)

Fig. 13 **a** Bar chart of performance metrics of Experiment-1, **b** Bar chart of performance metrics of Experiment-2, **c** Bar chart of performance metrics of Experiment-3

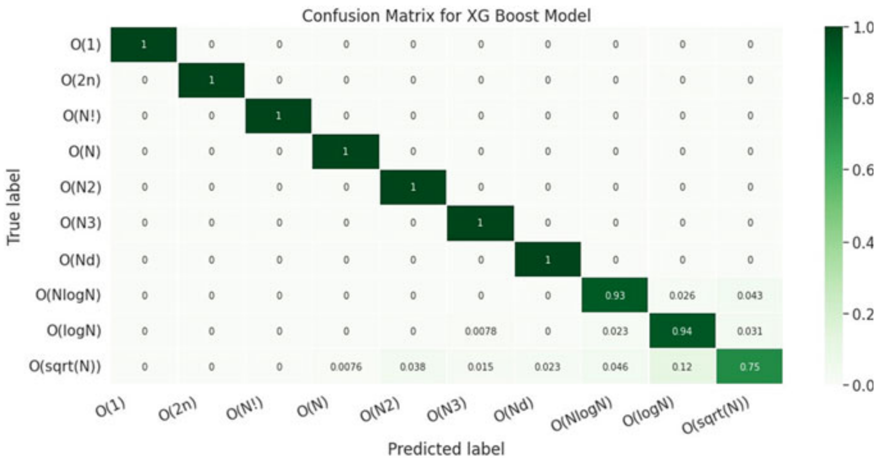


Fig. 14 Confusion matrix for XGBoost exclusive of programming language in Experiment-1

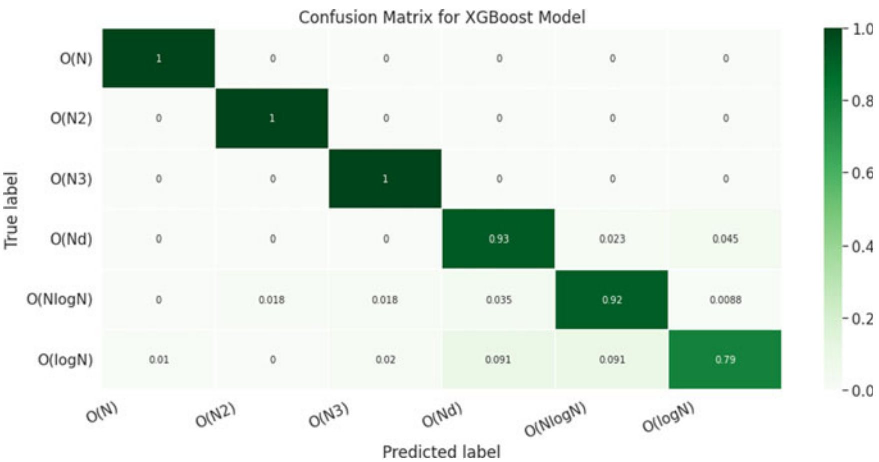


Fig. 15 Confusion matrix for XGBoost inclusive of programming language and time complexity

more time-consuming as compared to the previous model due to the high number of memory units. The results obtained are shown in Fig. 19a.

In the fourth model version, the runtime labels were combined with the language used which yielded a total of 18 runtime complexity labels. This was trained on a Bi-LSTM which consisted of 64 memory units for a total of 50 epochs. The results are shown in Fig. 19b, and the table in Fig. 17. The fifth model consisted of similar steps as in model four with the difference being in the number of memory units utilised in the Bi-LSTM which was 128 in number. The results obtained are shown in Fig. 20a.

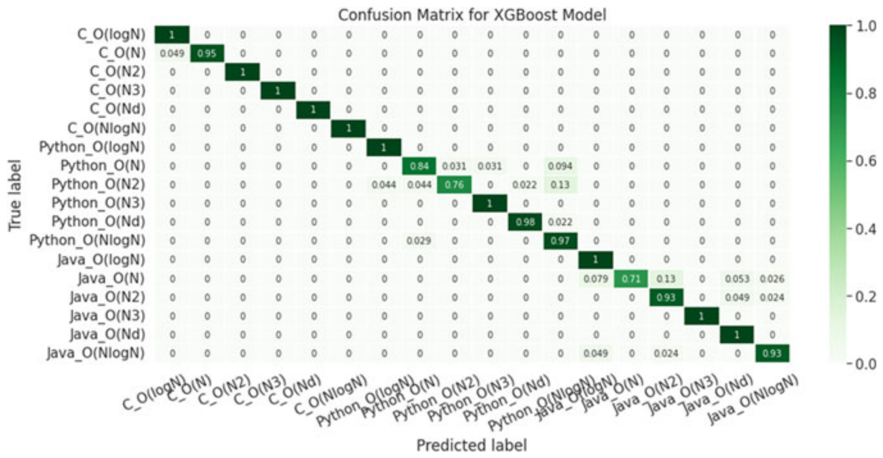


Fig. 16 Confusion matrix for XGBoost when both the programming language and the time complexity are combined into a single label

Model	loss	accuracy
1	0.309871435	0.906183362
2	0.394044429	0.888099492
3	0.381253242	0.911190033
4	0.30856508	0.909825027
5	0.268455386	0.928667545
6	0.337583184	0.893428087

Fig. 17 Accuracy and loss table for six Bi-LSTM models

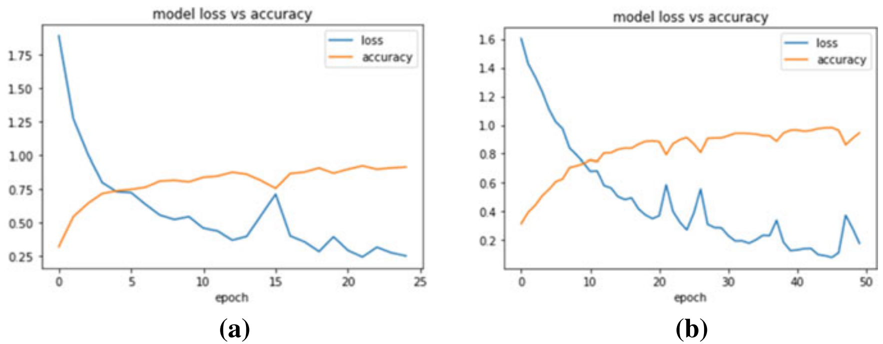


Fig. 18 **a** Loss versus accuracy plot for Bi-LSTM model 1, **b** loss versus accuracy plot for Bi-LSTM model 2

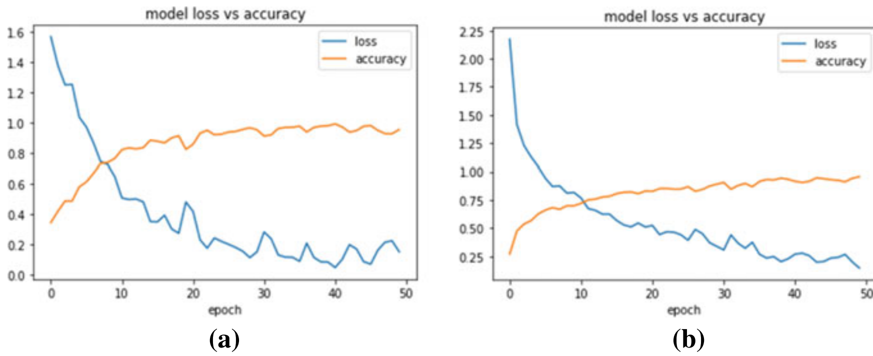


Fig. 19 **a** Loss versus accuracy plot for Bi-LSTM model 3, **b** loss versus accuracy plot for Bi-LSTM model 4

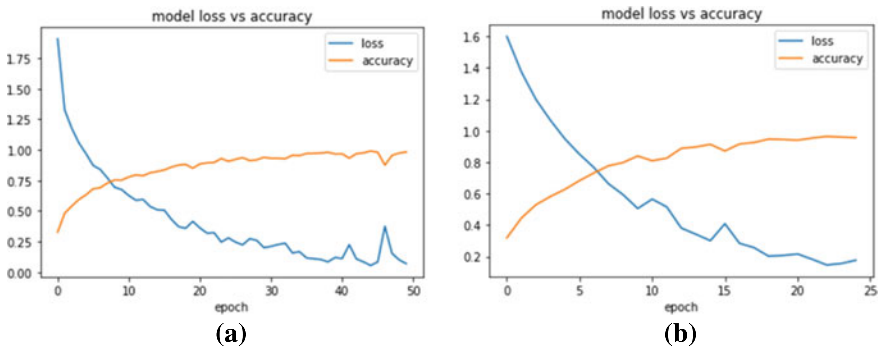


Fig. 20 **a** Loss versus accuracy plot for Bi-LSTM model 5, **b** loss versus accuracy plot for Bi-LSTM model 6

In the sixth model version, the runtime complexities which consisted of fewer than 3 codes were removed and then steps similar to model one were followed. The results obtained are shown in Fig. 20b. In conclusion, training the model with a higher number of memory units reduced the loss and increased the accuracy. From the results, model 5 performed the best with an accuracy of 92.86%.

6.1 Assumptions

The current approach does not probe into the syntactical correctness of the program. It is assumed that all the programs are error-free. The solution presented by us does not involve built-in python packages like sklearn, pandas, etc. This solution supports the prediction of algorithmic time complexity. Another assumption is that the program completes running in a finite time and has utilised such codes in the study.

7 Limitations and Future Work

The current project is restricted to three languages, C, Python and Java. We would like to extend it to more languages like C++ and other commonly used languages for writing algorithms. This study can be extended to include some less commonly used packages in these languages to identify their runtime complexity. As a part of future work, we intend to extend the dataset by adding more data samples and trying Graph Neural Networks, for classification, since the problem also falls under graph classification. This can also be implemented as a tool or a web browser extension to calculate code runtime complexity. Currently, a frontend is built using the Streamlit Python package where the user can drop in a zip file with codes and the backend will compute the code's runtime complexity. This study can also be implemented as a web browser extension for easy computation of runtime code complexity of codes in various other high-level languages.

8 Conclusion

Predicting the code complexity can help aid in improving code quality. This can be tedious if done manually, hence using static analysis and machine learning to make things easier. The current approach aims at solving the current problem at hand for three languages, C, Java and Python. With the current approach, we also use ASTs and graph embeddings, rather than just word embeddings from programs. We find that Random Forest, accompanied by GridSearchCV for hyperparameter tuning, outperforms all the other models. We would also like to try various other algorithms to achieve accurate and better results. We hope that our research helps developers and learners who always strive to write better code.

References

1. Shunnarski A (2022) Welcome to the Big O Notation calculator! https://shunnarski.github.io/BigO.html. Accessed 05 May 2022
2. Sikka J, Satya K, Kumar Y, Uppal S, Shah RR, Zimmermann R (2020) Learning based methods for code runtime complexity prediction. Lect Notes Comput Sci 12035:313–325
3. Agenis-Nevers M, Bokde ND, Yaseen ZM, Shende MK (2020) An empirical estimation for time and memory algorithm complexities: newly developed R package. Multimedia Tools Appl 80(2):2997–3015
4. Hutter F, Xu L, Hoos HH, Leyton-Brown K (2014) Algorithm runtime prediction: methods & evaluation. Artif Intell 206:79–111
5. Haridas P, Chennupati G, Santhi N, Romero P, Eidenbenz S (2020) Code characterization with graph convolutions and capsule networks. IEEE Access 8:136307–136315. https://doi.org/10.1109/ACCESS.2020.3011909
6. Gao Y, Gu X, Zhang H, Lin H, Yang M (2021) Runtime performance prediction for deep learning models with graph neural network. MSR-TR-2021-3/Microsoft

7. Chen L, Ye W, Zhang S (2019) Capturing source code semantics via tree-based convolution over API-enhanced AST. In: Proceedings of the 16th ACM international conference on computing frontiers (n. Pag.)
8. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), pp 783–794. https://doi.org/10.1109/ICSE.2019.00086
9. Lin C, Ouyang Z, Zhuang J, Chen J, Li H, Wu R (2021) Improving code summarization with block-wise abstract syntax tree splitting. In: 2021 IEEE/ACM 29th international conference on program comprehension (ICPC), pp 184–195. https://doi.org/10.1109/ICPC52881.2021.00026
10. Kurniawati G, Karnalim O (2018) Introducing a practical educational tool for correlating algorithm time complexity with real program execution. J Inf Technol Comput Sci 3(1):1–15
11. Büch L, Andrzejak A (2019) Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER), pp 95–104. https://doi.org/10.1109/SANER.2019.8668039
12. Wang W, Li, Bo Ma, Xin Xia, Zhi Jin. "Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree." SANER 2020, London, ON, Canada 978–1–7281–5143–4/20/© 2020 IEEE.
13. Feng Q, Feng C, Hong W (2020) Graph neural network-based vulnerability predication. In: 2020 IEEE international conference on software maintenance and evolution (ICSME), pp 800–801. https://doi.org/10.1109/ICSME46990.2020.00096
14. Reza SM, Rahman Md, Parvez Md, Badreddin O, Al Mamun S (2020) Performance analysis of machine learning approaches in software complexity prediction. https://doi.org/10.1007/978-981-33-4673-4_3
15. Guzman J, Limoanco T (2017) An empirical approach to algorithm analysis resulting in approximations to big theta time complexity. J Softw 12:964–976. https://doi.org/10.17706/jsw.12.12.964-976
16. Ströder T, Aschermann C, Frohn F, Hensel J, Giesl J (2015) Aprove: termination and memory safety of C programs. In: Tools and algorithms for the construction and analysis of systems, pp 417–419
17. Rozemberczki B, Kiss O, Sarkar R (2020) Karate Club: An API oriented open-source python framework for unsupervised learning on graphs. In: Presented at proceedings of the 29th ACM international conference on information & knowledge management, Ireland
18. Narayanan A, Chandramohan M, Venkatesan R, Chen L, Liu Y, Jaiswal S (2017) graph2vec: learning distributed representations of graphs
19. Lundberg SM, Lee S-I (2017) A unified approach to interpreting model predictions. In: Proceedings of the 31st international conference on neural information processing systems (NIPS'17), Red Hook, NY, USA, pp 4768–4777
20. Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority oversampling technique. J Artif Intell Res 16:321–357