CroLSSim: Cross-Language Software Similarity detector using Hybrid approach of LSA-based AST-MDrep Features and CNN-LSTM Model

Farhan Ullah[1], Rashid Naeem[2], Hamad Naeem[3], Xiaochun Cheng[4], Mamoun Alazab[5]

[1]School of Software, Northwestern Polytechnical University, Xi'an Shaanxi, 710072, P.R. China, farhankhan.cs@yahoo.com

[2]School of artificial intelligence, Leshan Normal University, 614000, PR China, rashidnaeem717@yahoo.com

[3]School of Computer Science and Technology, Zhoukou Normal University, Zhoukou 466001, Henan, China
hamadnaeemh@yahoo.com

[4]Department of Computer Science, Middlesex University, London, UK
x.cheng@mdx.ac.uk

[5]College of Engineering, IT and Environment at Charles Darwin University, Australia
alazab.m@ieee.org

Corresponding Author: Farhan Ullah

Abstract: Software similarity in different programming codes is a rapidly evolving field because of its numerous applications in software development, software cloning, software plagiarism, and software forensics. Currently, software researchers and developers search cross-language open source repositories for similar applications for a variety of reasons, such as reusing programming code, analyzing different implementations, and looking for a better application. However, it is a challenging task because each programming language has a unique syntax and semantic structure. In this paper, a novel tool called Cross-Language Software Similarity (CroLSSim) is designed to detect similar software applications written in different programming codes. First, the Abstract Syntax Tree (AST) features are collected from different programming codes. These are high-quality features that can show the abstract view of each program. Then, Methods Description (MDrep) in combination with AST is used to examine the relationship among different method calls. Second, the Term Frequency Inverse Document Frequency (TFIDF) approach is used to retrieve the local and global weights from AST-MDrep features. Third, the LSA-based features extraction and selection method is proposed to extract the semantic anchors in reduced dimensional space. Fourth, the CNN-based features extraction method is proposed to mine the deep features. Finally, a hybrid deep learning model of Convolution Neural Network- Long Short Term Memory (CNN-LSTM) is designed to detect semantically similar software applications from these latent variables. The dataset contains approximately 9.5K Java, 8.8K C#, and 7.4 K C++ software applications obtained from GitHub. The proposed approach outperforms as compared to the state-of-the-art methods.

Keywords: Abstract Syntax Tree, Latent Semantic Analysis, Software Similarity, Data Mining, Deep learning

1. Introduction

The software development process benefits from having access to similar Open Source

Software (OSS) projects. By studying a common OSS project, software developers can learn how appropriate classes are developed up to a certain level and reuse valuable code. This allows them to create mature, professional, and high-quality software. The emergence of several large-scale online coding databases, such as GitHub[1] or Stack Overflow[2], comprising possibly millions or billions of lines of code, has thrown numerous obstacles to code similarity assessment. The development of software is a complicated procedure that involves better skills to get the required results. It demands knowledge of many programming languages, methods, development environments, and upcoming technologies, as well as a vast array of third-party applications and tools. As a consequence, programmers devote considerable time and effort to learning new third-party libraries, written code, and how to properly incorporate new technologies. The amount of time needed to study and analyze data can have a considerable influence on the effectiveness of the application and the desired outcome. GitHub hosts over 29 million open source repositories from around 11 million application developers. A repository is a significant area in GitHub that typically stores software codes and reference materials. It also records the project's foundation and enhanced features, and the individuals who construct, contribute, deploy, and maintain it [1]. In recent years, a substantial amount of research has gone into establishing tools and resources to facilitate developers in examining and dealing with the large amounts of data available on OSS platforms such as GitHub. The major goal is to help programmers by making appropriate recommendations for building better applications. This is built together by combining current structures and background experience. For instance, when a software developer is designing a new application, it is possible to make suggestions about which frameworks he should employ based on a comparison with many other related OSS [2]. Furthermore, the Intergraded Development Environment (IDE) can be strengthened by the reasonable interpretation of tools that continuously observe the developer's coding style and contextual circumstances to trigger dedicated suggestion systems [3]. Several techniques are being developed that may be integrated with the described methods to improve the assessment of similar software, such as Closely reLated ApplicatioNs (CLAN) [4] and Repopal [5].

Code similarity is widely used in software engineering research. The skills and novel technologies have given for such OSS analysis are beneficial to improving software quality. It has a significant impact on both software research and industry. The advent of web-based open source hosting services such as GitHub has significantly changed the way code is accessed. Copying and pasting is a viable method in application development. In the context of OSS, two pieces of software are deemed semantically similar if they include characteristics specified in the same abstraction, even if they provide different capabilities for various disciplines. The analysis of the structural similarity between applications to evaluate their preciseness, style, and uniqueness is one of the most provoking activities to similar software. For optimum search, semantically similar OSS applications can be categorized in several repositories. In this method, developers may readily obtain up-to-date relevant OSS, permitting them to develop software innovations with better quality and at a faster pace. Another application of code similarity analysis is software plagiarism detection. It is caused by malicious software cloning to hide the authorship of the cloned code. It is a source of concern in both industry and academia, and it may have major legal consequences. For instance,

[1] https://github.com/about/press
[2] https://stackoverflow.com/

Oracle[3] filed a $9 billion claim in the United States Federal Circuit Court, alleging that Google had stolen its Java APIs in the Android version. The ability to reuse code is an intrinsic element of OSS that supports creating a new project using current open-source code. Identifying similarities between OSS projects allows for code reuse, prototyping, and the selection of suitable technologies. It could help with two parts of the software life cycle: implementation and maintenance [5]. Application developers usually develop similar applications to make them faster, more feature-rich, and to improve their design, analytics, efficiency, and effectiveness [6-8]. Identifying similar applications is also valuable for handling other computer science research problems, such as software re-usability [9], software plagiarism detection [10], code clones [11], and software digital forensics [12, 13], etc. Semantic similarity computation in cross-language OSS is a challenging task as each programming language has specific syntax and semantic structures. As a result, we require a novel approach that can predict semantic similarity among OSS written in different programming codes.

1.1 Challenges

- Identifying similar software applications based on code similarities in the same programming code is relatively simple, and some effective algorithms are already developed [14-16]. But, it is a challenging task to classify different programming codes because each language has different semantics and syntax structure.
- The tokenization approach is used to break source codes into tokens, which are then compared to determine the similarity between program pairs. Different methods are proposed, which are quite popular for the identification of similar tokens, such as JPlag [17], Measure of Software Similarity (MOSS) [18], Yet Another Plague (YAP) [19], etc. These online software technologies are unlikely to detect code reordering infringement attacks in huge source code corpora. Although the tokenization-based similarity detection technique is resistant to identifying renaming, spacing, and alignment, it is vulnerable to code sequence, structural, and semantic information. To recognize comparable patterns in a vast volume of data, we need a system that can go beyond the specific language and compare multiple types of programs using abstract syntactic structures.
- Typically, for code similarity detection, each element (e.g., API calls, class names, utilized library procedure names, syntax details, etc.) in the program is regarded as a semantic anchor that contributes to defining the semantic properties of that content [15, 20]. These anchors have different titles in different programming languages but with the same functionalities. For instance, the output methods used in C++, Java, and C# (“cout”, “print”, and “Write”) have the same functionalities but with different titles. These methods can be replaced with their descriptive text for improving the similarity score across different programming languages.

1.2 Contributions

In this research, we proposed CroLSSim, a novel approach for detecting similar software applications developed in different programming languages. The source code is converted to AST

[3] https://www.theverge.com/2018/3/27/17169064/federal-circuit-oracle-v-google-third-trial-java-android

features to explore the abstract view of different programming codes. Methods are replaced with MDrep within the code structures to enhance the power of semantic similarity. After that, the LSA method is used to explore the code semantics using the SVD algorithm. The main contributions of the paper are following:

1. A novel tool called CroLSSim is developed to detect semantic similar software using AST-MDrep features and LSA.
2. A large corpus of open source projects is gathered and pre-processed for noisy data. The AST features are extracted from the corpus using the ANTLR parser. The MDrep features are to strengthen the semantic similarity process.
3. The SVD algorithm is used to reduce the high dimensional features preserving the actual information of data. Then, the LSA method is used to extract the latent features in reduced dimensional space. These are high-quality features that can show the semantic relatedness across different programming codes
4. The combined CNN-LSTM deep learning model is designed to classify the semantically similar software across different programming codes. The proposed approach's efficiency is validated by comparison with other cutting-edge deep learning approaches.

The remaining sections of the paper are organized as follows: Section 2 contains the related work, section 3 explains the proposed methodology, the results, and the discussions part described in the 4th section, and section 5 includes the conclusion with future research directions.

2. Related Work

Accessibility to similar OSS projects enhances the software development process. Software engineers can understand how suitable methods are developed up to a certain degree and reuse important code by examining a common OSS application. As a result, they can produce mature, professional, and high-quality applications. Nowadays, software application developers from numerous domains interact on open platforms that enable the convergence of a broad set of software approaches and systems via scientific procedures. GitHub [21] and Galaxy [22] are two examples of these web platforms. If a large number of software programs with similar specifications are available in a single repository, users can compare and choose the optimal one for their needs. Many experts are interested in determining cross-language similarities between software projects [5, 23-25]. For example, Kawser et al. [20] proposed Cross-Language Similarity (CroLSim) approach using API description. The transformed programming codes with API descriptions are tokenized to produce features for the LSA and deep learning model. The proposed technique achieves 82% classification accuracy but could not target structural similarities. Gharehyazie et al. [9] proposed research work on cross-application cloning in GitHub repositories to obtain a better understanding of code exchange and reuse in a community. They integrated the proposed method with the Deckard tool [26] to detect similar applications in 5,753 Java projects. The proposed method retrieves 69% to 91% of common clones within the same programming language. Quanlai et al. [5] proposed a method to detect similar applications on Github. They used three heuristics, i.e. Projects starred by the same individuals in a short span of time, projects starred by comparable individuals, and projects with identical readme files are probably similar. The proposed approach is tested with

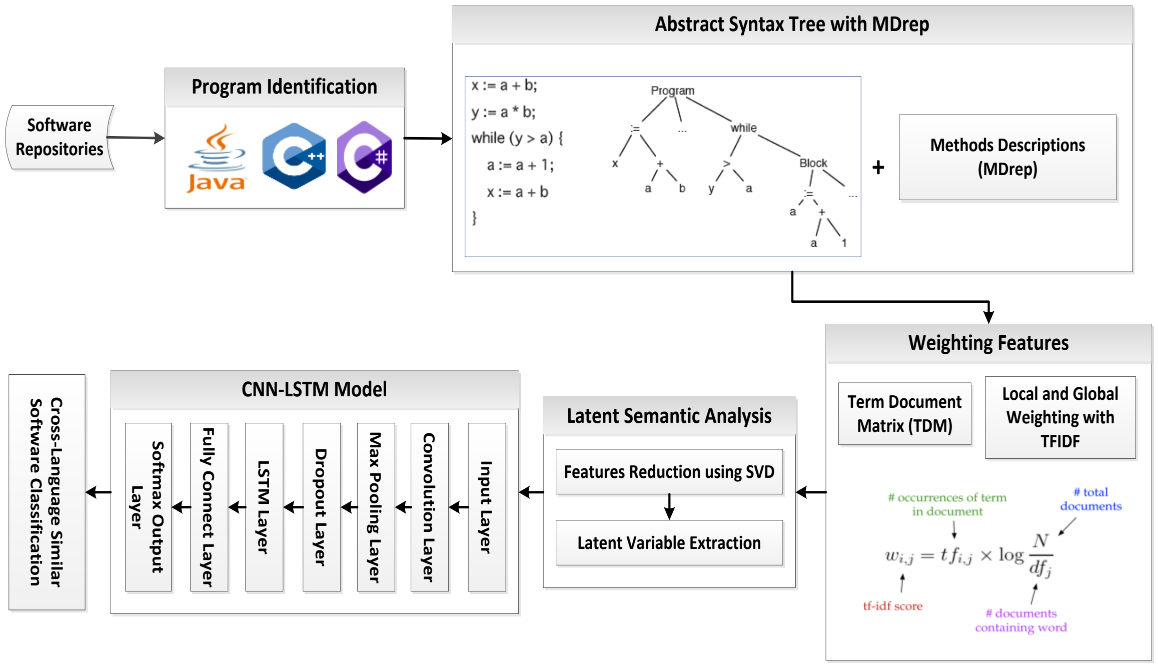
1,000 Java repositories and got an 88% of success rate for a single programming language. Miika et al. [27] presented a hybrid approach that takes into account the programming languages, concepts, and README files found in users' files. They implement a working prototype that provides recommendations for useful programming scripts. Phuong et al. [7] examine the problem of exploring OSS repositories for similar projects that can be reused by programmers. They present the CROSSSIM method for modeling the OSS ecosystem and computing software project correlations written in the same programming language. The approach has been tested on 580 software OSS projects with an accuracy rate of 80%. Gang et al. [2] introduce a deep learning-based approach called DeepSim for encoding code program flow and data flow into a conceptual matrix. Each member is a high dimensional dense binary feature vector, and then develop a new deep learning method based on this description to evaluate functions based on program similarity. The DeepSim approach is tested with 1,669 Google Code Jam (GCJ) Java projects and received precision, recall, and f1 scores of 82%, 71%, and 76%, respectively. Nichols et al. [28] adapted the optimal Smith-Waterman sequence alignment algorithm to measure similarity between source codes. This method is primarily applicable to any type of source code specified as grammar by ANother Tool for Language Recognition (ANTLR). This structural features are extracted from programing codes and then further preprocessed to apply the suggested approach. A dataset of 100 Java programs, which includes 50 original and 50 plagiarized files are generated to evaluate the approach. Results showed that the precision, recall, and f-measure are 67.3%, 66%, 66.7%, respectively. Kikuchi et al. [29] recommended an AST based sequence alignment strategy to recognize code similarity in C programs. The technique of sequence alignment is used to determine the similarity of source code sequences by inserting a space or changing the alphabet position. The syntactic structure generates a sequence of features to organize the tokens as specified by the AST. The dataset includes four separate benchmark programs: variable string lengths using a linked list, reverse polish notation using a stack, sorting numbers using the queue, and sorting integers using binary search. The similarity accuracy for these four programming exercises is predicted at 92.8%, 85.9%, 97.4%, and 81%, respectively. Despite recent research indicating that their approaches may be better for detecting cross-language similar software applications, we have yet to find a better approach that can target structural information with high classification results.

3. Proposed Method: Cross-Language Open Source Software (OSS) Classification

The proposed methodology for identifying code similarity in various programming languages consists of four major steps: AST features extraction with MDrep, Features weighting, Semantic Features extraction and reduction, and CNN-LSTM deep learning model. The complete architecture of the proposed framework for source code is shown in Figure 1. First, the AST features are extracted from OSS repositories collected from Github in three different programming codes, i.e. Java, C#, C++. Following that, the called methods are replaced by Method Description (MDreps) obtained from standard documentation such as Microsoft, for C++[4], C#[5], and Oracle[6] for Java. Second, the TFIDF tool is used to compute the local and global weights which zoom the importance of each feature. Third, the SVD features reduction algorithm is used to filter the most important

[4] https://docs.microsoft.com/en-us/cpp/cpp/
[5] https://docs.microsoft.com/en-us/dotnet/csharp/
[6] https://docs.oracle.com/javase/8/docs/api/

features, and after, the LSA data mining tool is applied to compute the latent variables in reduced dimensional space. These latent features contain the code semantics for each programming language. Fourth, the combined approach of the CNN-LSTM deep learning model is designed to classify cross-language OSS projects. The detailed steps of the proposed method are presented in the following sections.



$$w_{i,j} = tf_{i,j} \times \log \frac{N}{df_j}$$

Figure 1: Cross-Language Software Classification using Hybrid Approach of AST-MDrep Features and Latent Semantic Analysis

3.1 Abstract Syntax Tree (AST) Features Extraction

The AST [30] is a syntax tree description of the programming code's syntactic structure, with each node designating a construct in the series. It could include annotations of the content of each method and expression in the code. It creates a single node for such type of expression rather than creating a distinct node for the parenthesis used in the *if* statement. Then, for sub-statements, such as *if-else* statements, it constructs sub-nodes. The *if* condition is shown in the first node, and the contents of the parent node are represented by the sub-nodes. The sequence of each statement in the specified code can be viewed. As a result, the AST can quickly distinguish the abstract features in different programming languages. The AST uses ANother Tool for Language Recognition (ANTLR) parser to evaluate the abstract view of each document in a hierarchical fashion for different types of codes, such as Java, C#, and C++. The ANTLR [31] is an efficient parser producer to read, analyze, execute, and interpret structured source codes. It is often used to build parsers that can traverse syntax trees. It supports C++, Java, C #, and a variety of other programming languages. Figure 2 describes ANTLR architecture for extraction of AST features. It takes input in the form of characters and produces grammar for the walker. For instance, the lexer breaks programming codes into meaningful tokens, i.e. 25 and 45 are numbers, while + is a sum. The parser is being used to identify the statement structure by combining these tokens with metadata, such as supplementary data, acronyms, tabular data, data flow, and so on. The language recognition system determines the type of code and afterward activates the necessary parsing mechanisms. The ANTLR uses different methods to develop parser for different types of source codes, as shown below:

- ***ArrayInitParser()*:**It defines parser classes based on the syntax used.
- ***ArrayInitLexer ():*** It contains the definition of the lexer class.
- ***ArrayInit.tokens( ):*** ArrayInitLexer.*tokens( ):* These are the ANTLR core files. Such files provide token vocabulary support with appropriate IDs.
- ***ArrayInitListener( ): This interface is used for traversing AST for extensive analysis.***
- ***ArrayInitBaseLisener( ):*** It is the base listener class that is used to initialize the listener.
- ***ArrayInitVisitor( ):*** Using the visitor design pattern, this interface is used to navigate around AST.
- ***ArrayInitBaseVisitor( ):*** It is a visitor base class that is used to initialize visitors.

Furthermore, the AST walker has several methods, such as enter, listener, visitor, and exit, that are used to traverse the relevant trees. Walker calls the enter*( ) or exit*( ) function whenever a node is explored or finished. The walker then generates the code after exploring all nodes of the tree. The generated code is then used to produce the AST hierarchical features. As shown in Figure 3, we designed a technique to extract ASTs from various types of programming codes using ANTLR parsing methods. It shows the AST for the "MaxNumber" problem in three different programming languages such as **a)** C++, **b)** Java and **c)** C#. Despite the fact that the language type is different, the ASTs have the same depth level, i.e., 8. Furthermore, numerous nodes in all subtrees have almost the same node, i.e., FOR loop, IF condition, VAR, and METHOD. These hierarchical features represent the abstract view of software, including statements and expressions under procedures or logic. This allows us to readily evaluate structural information, which can help us investigate the sequence of different types of programming codes. Moreover, the method used in different types of source codes may have the same functionality but with different titles. For instance, "cout", "print", "Write" are the output functions in C++, Java, and C#, respectively. They perform the same functions but have distinct naming conventions. The MDrep is a natural language text that explains how each method of a programming language works. These are documented in a standard format so that we can easily understand the method's explanation. To obtain high-quality features for data mining tools, we employed MDrep documentation for each method within AST.

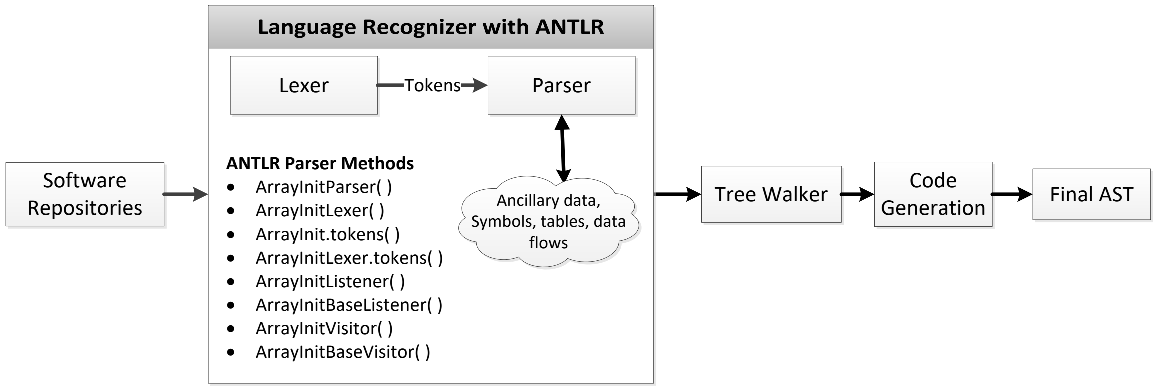
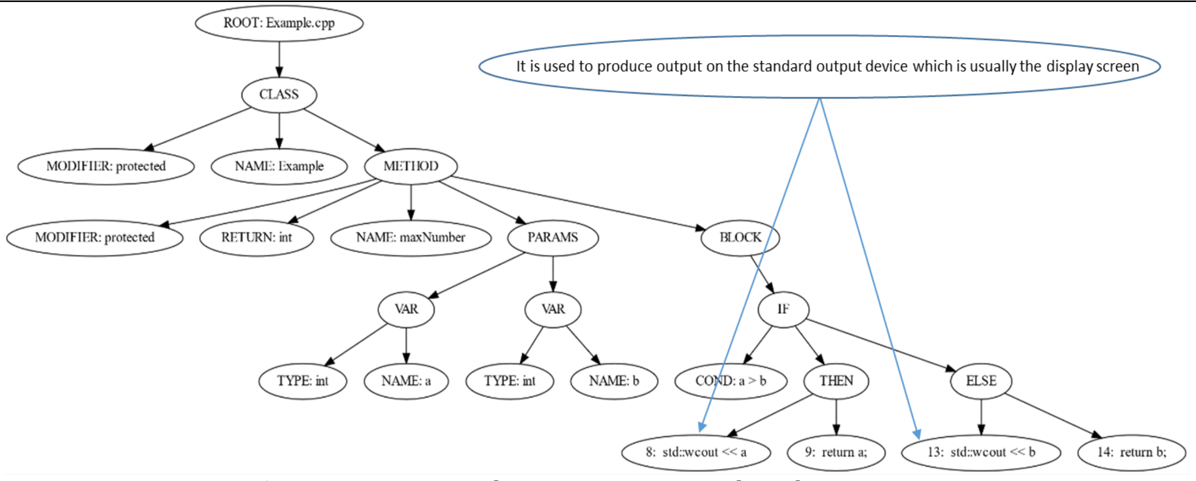
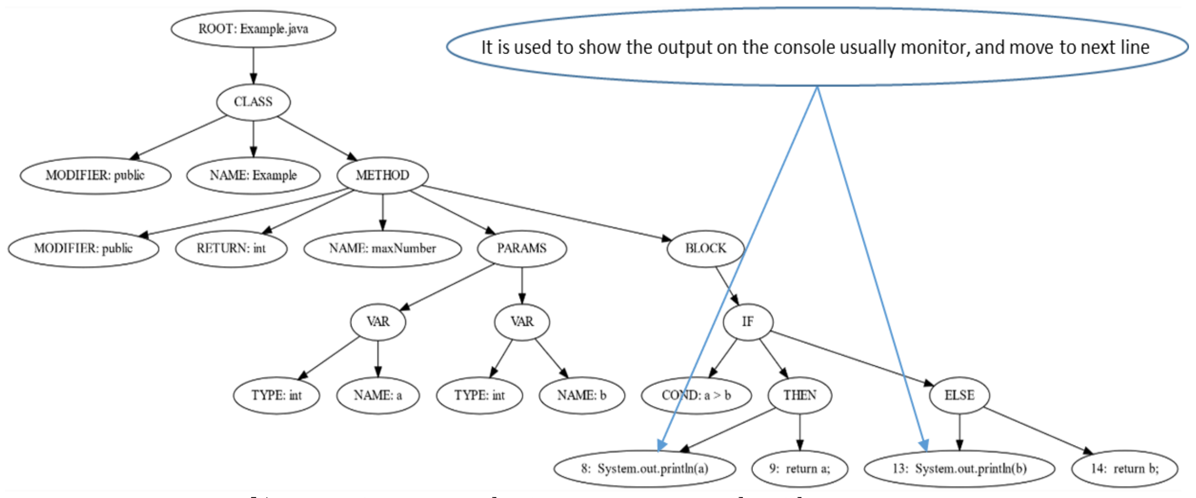


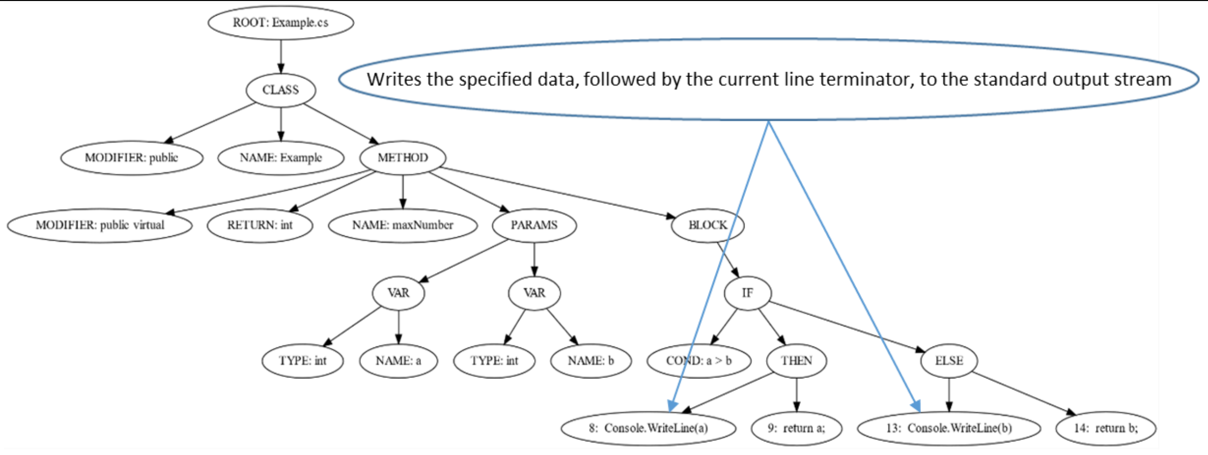
Figure 2: ANTLR Architecture for Extraction of Abstract Syntax Tree (AST) Features



a) An AST generated using **C++** source code with MDrep Features



b) An AST generated using **Java** source code with MDrep Features



c) An AST generated using **C#** source code with MDrep Features

Figure 3: An "Example" program with "MaxNumber" method is written in (a) C++, (b) Java, and (c) C# programming languages. The ASTs of the same program written in three different programming codes have similar behavior in edges and nodes, indicating a proportion of similarity in programming logic regardless of source code style. The called methods are replaced by the corresponding descriptions from standard documentation to analyze the semantic anchors

3.2 Features Weighting

We have converted ASTs into linear features for effective evaluation because examining ASTs in a hierarchical system takes a lot of time and resources. The basic goal is to create a set of indexes that may be used to get the best possible results from search and retrieval techniques. The ASTs are broken down into small features using preprocessing steps and the Bag of Words (BoW) model. Preprocessing steps [32] include stop word removal stemming, minimum and maximum frequency settings, etc. The stop words are noisy information that may not carry useful meanings which negatively affect the cross-language software similarity. For instance, Integer, char, string, class, and static, etc. These words have no relevance to the code notion. Natural language text is also used in MDrep documentation, which might add an unnecessary burden to the source code. For example, and, but, the, and so on. Furthermore, some non-textual features and special symbols are also removed for better similarity scores such as +, -, *, /, %, &, and so on. That is employed in a variety of expressions and is also unhelpful for the next step. To capture the most valuable AST node in linear form, we removed all of these forms of stop words. Now, we have a huge number of features with frequencies details.

It is unclear at this point which features are more significant for the source code similarity. We used the TFIDF method to extract the local and global importance for each feature. The TFIDF [33] is a statistical model designed to indicate the importance of a feature to a single document or a collection of documents. It is divided into two sections: TF for local weight and IDF for global weight. Let T designate the AST feature, and for each subtree s there is weight $w_{s,T}$. Equation 1 defines the overall predicted weight of a node as the multiplication of the TF and IDF techniques [30].

$$w_{s,T} = TF(s,T)\,.\,IDF(s,T) \qquad (1)$$

The TF(s,T) and IDF(s,T) can be defined mathematically as in Equations 2 and 3.

$$TF(s,T) = \frac{cnt(s,T)}{n(T)} \quad (2)$$

Where *cnt* is the number of occurrences of subtrees or nodes such that $s \in S_T$ and *n(T)* is the number of subtrees in the particular region shown in Equation 3.

$$IDF(s,T) = log\frac{N}{c(s)} \quad (3)$$

Where c(s) signifies the number of ASTs in s and N indicates the overall number of ASTs generated from a set of programs. Programming code similarity can be affected by nodes with equivalent TFIDF scores. But, if a node has a higher weight value in one document but zero or lower in another, the efficacy of the proposed system is unaffected. These unexpected numbers can have a severe impact on the overall classification accuracy. We are particularly interested in nodes that have TFIDF values that are equivalent or close to one another. As a result, the TFIDF approach can play a pivotal part in enhancing the impact of the AST nodes.

3.3 Level-1 Features Extraction and Selection using LSA-based Semantic Anchors

For efficient classification using deep learning, we must provide enough meaningful data to train the model. Overfitting issues can be efficiently avoided with proper feature selection and reductions, and the model's robustness can be improved. The LSA-based semantic anchors play a key role in mining meaningful data in reduced dimensional space. LSA is a knowledge discovery technology that captures the semantics of textual data in sparse representation space. It is also called Latent Semantic Indexing (LSI), and the name LSI refers to an extracting strategy that uses latent variables to discover semantic anchors. LSA's hidden power is the Singular Value Decomposition (SVD). The SVD is a statistical approach for feature selection and reduction that is used to obtain relevant information from a dense matrix without losing the actual information. It deconstructs the TFIDF matrix into separate matrices that can be further used to detect similarities between weighted AST-MDrep features across multiple languages [34, 35]. As we have sparse weighted features that may contain unnecessary data. This data may not be useful for source code similarity detection and can affect negatively. We used the Singular Value Decomposition (SVD) algorithm to convert the sparse weighted AST-MDrep features into reduced patterns. As we are only interested in identifying similar patterns in different types of source codes, therefore we need semantic anchors that can reflect the actual meaning of programming codes. To address this problem, we used Latent Semantic Analysis (LSA) method to extract semantic anchors from cross-language codes. These semantic anchors are retrieved in the form of latent variables that can be used as input to the CNN model for deep features extraction. The weighted AST-MDrep features are given to the SVD algorithm to compute the reduced features preserving the actual meaning. The SVD algorithm transforms the weighted features into multiple of three matrices [36, 37], i.e., M, U, Σ, V^T as shown in equation 4.

$$M = U\Sigma V^T \quad (4)$$

Where M is $m * n$ matrix, U is a $m * n$ singular matrix, Σ is $n * n$ diagonal matrix, V^T is the transpose of V matrix. The diagonal matrix is one in which all cells except the diagonal have zero values, while the singular matrix is one in which the determinant is zero or a square matrix without a matrix inverse.

Figure 3.6 shows a full illustration of these matrices. M is the weighted AST-MDrep feature matrix, where the rows and columns represent the AST-MDrep features and programming codes, respectively. The U is a single matrix that represents the conceptual allocation of each AST-MDrep feature in its relevant programming code. The Σ is a diagonal matrix that displays the AST-MDrep information in both rows and columns. This matrix illustrates how different AST-MDrep properties are connected across programming languages. Consequently, the V^T matrix is the transpose of the V matrix, with rows and columns representing AST-MDrep characteristics and programming language type. This matrix illustrates the representation of each AST-MDrep in the corresponding codes. One AST-MDrep feature may be replicated in one or more language types, depending on the semantic anchors captured. The three matrices are combined to get semantic meaning about each feature, which may then be used to determine cross-language coding similarity. The basic working components of LSA are visualized in Fig 5. It uses the BoW model to compute the latent variables in the form of semantic anchors from the weighted AST-MDrep features in reduced dimensional space. The CNN-LSTM deep learning model is trained using the reduced semantic anchors. This allows the deep learning model to be better trained on the most relevant features for effective classification.

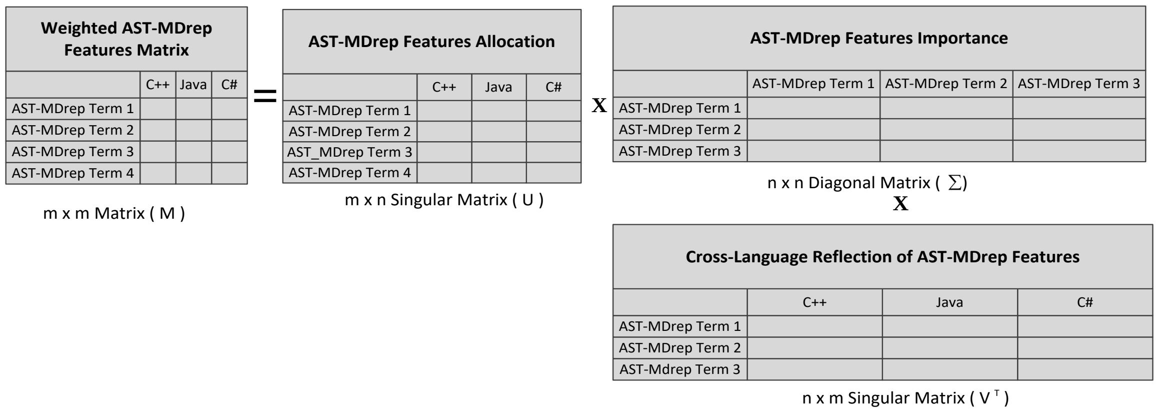


Figure 4: Weighted AST-MDrep Features Reduction using Singular Value Decomposition (SVD)

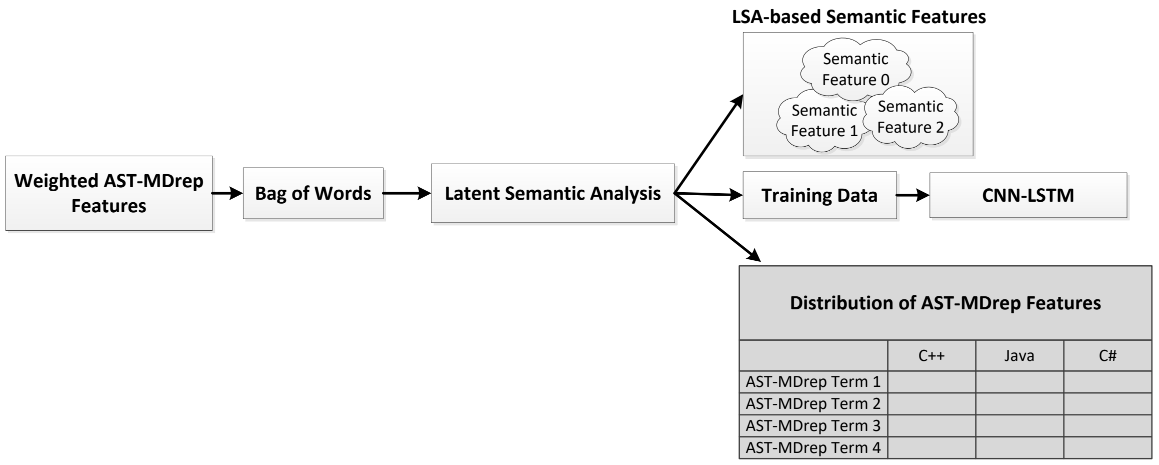


Figure 5: Latent Semantic Analysis using Weighted AST-MDrep Features

3.4 CNN-based Level-2 Features Extraction

For efficient classification using the deep learning model, the LSA-based feature vectors are further investigated using the CNN network. Several studies [38-40] have used the CNN-based neural network for programing codes classification. The CNN network is very good at processing high-dimensional data including textual information, pictures, and videos. However, instead of using direct AST-MDrep features as an input to the CNN network, we use the matrix formed by the LSA-based feature vectors in the proposed approach. As shown in Figure 6, the proposed method employs a one-dimensional CNN network with convolution layers, pooling layers, dropout layers, and a fully connected layer. The CNN network is built with Python's tensor flow library 1.9. The convolution layer acts as a filter to roll over the LSA-based feature vectors, and then extract optimum deep features. Each filter's obtained features are grouped into a new feature set called a feature map. Hyper-parameter tuning is used to determine the best length and number of filters. As a consequence, each element is activated using the non-linear activation function, i.e. Rectified Linear Unit (ReLU). Two convolution layers in the presented CNN network use 64 and 128 filters, a kernel of size four, and then the same padding value to produce the exact dimensions of the throughput as the input. The max-pooling layer reduces the spatial size, the number of features, and the computation cost. Two max-pooling layers in the CNN network use pooling of size two and strides of size one to create a feature map with the most significant features from the prior feature space. The softmax function and dropout layers are used to handle the overfitting problem in the proposed CNN network. Equation 5 is used to represent the outcome of the one-dimensional CNN network.

$$o_k^1 = f(c_k^1 + \sum_{i=1}^{N_{l-1}} Con1D(X_{ik}^{l-1}, t_i^{l-1})) \qquad (5)$$

Where c_k^1 is the parameter bias of the kth neuron in the first layer, t_i^{l-1} is the outcome of the ith neuron in layer l-1, X_{ik}^{l-1} is the kernel strength from the ith neuron in layer l-1 to the kth neurons in layer l, and "f()" is the activation function.

3.5 Features Detection and Classification using CNN-LSTM

In CNN networks, the fully connected layer receives the extracted features from max-pooling layers. But, in the proposed CNN network, deep features are routed to the LSTM layer instead of the fully connected layer. The CNN network captures and interprets LSA-based semantic vectors effectively, whereas the LSTM network identifies long-short-term dependencies [41]. The proposed approach provides a combined CNN-LSTM model for automatic detection and classification of programming codes in order to benefit from the two models' properties. As illustrated in Fig. 6, the proposed CNN-LSTM model is divided into two phases. While phase one includes convolution layers and max-pooling layers, phase two is comprised of the LSTM layer, which is the final step in the process. The convolution layers encode the LSA-based semantic features, and the LSTM layer decodes the encoded information contained in the feature set. The content is flattened even further and then sent into a fully connected layer for efficient classification performance.

The LSTM model [42] consists of one memory unit and three other interaction gates: input, forget, and output gates. The memory unit is the most important part of the model. The memory cell keeps track of the state from the prior state and stores it in its memory. The input gate specifies how much network data should be kept in the unit state at a time “t”. At time “t -1", the forget gate selects whether data can flow or not to the concerned input gate. The parameters for the output gate are specified by the output gate. The model's operation is defined by Equation 6.

$$\begin{aligned} i_t &= \sigma(V_{ixt} + W_i h_{t-1} + b_i) \\ f_t &= \sigma(V_{fxt} + W_f h_{t-1} + b_f) \\ c_t^{\sim} &= tanh(V_c X_t + W_c h_{t-1} + b_c) \\ c_t &= (f_t A C_t + W_c h_{t-1} + i_t A c_t^{\sim}) \\ o_t &= \sigma(V_o x_t + W_o h_{t-1} + b_o) \\ h_t &= o_t A tanh(c_t) \end{aligned} \tag{6}$$

The input at time "t" is denoted by the symbolx_t, the weight matrices are denoted by the symbols x_* and w_*, and the bias and hidden states are denoted by the symbols b and h. The activation functions are illustrated by the symbols σ and tanh. It is followed by the letters i_t, f_t, o_t, and c_t, which stand for input gate, forget gate, output gate, and memory cell, respectively.

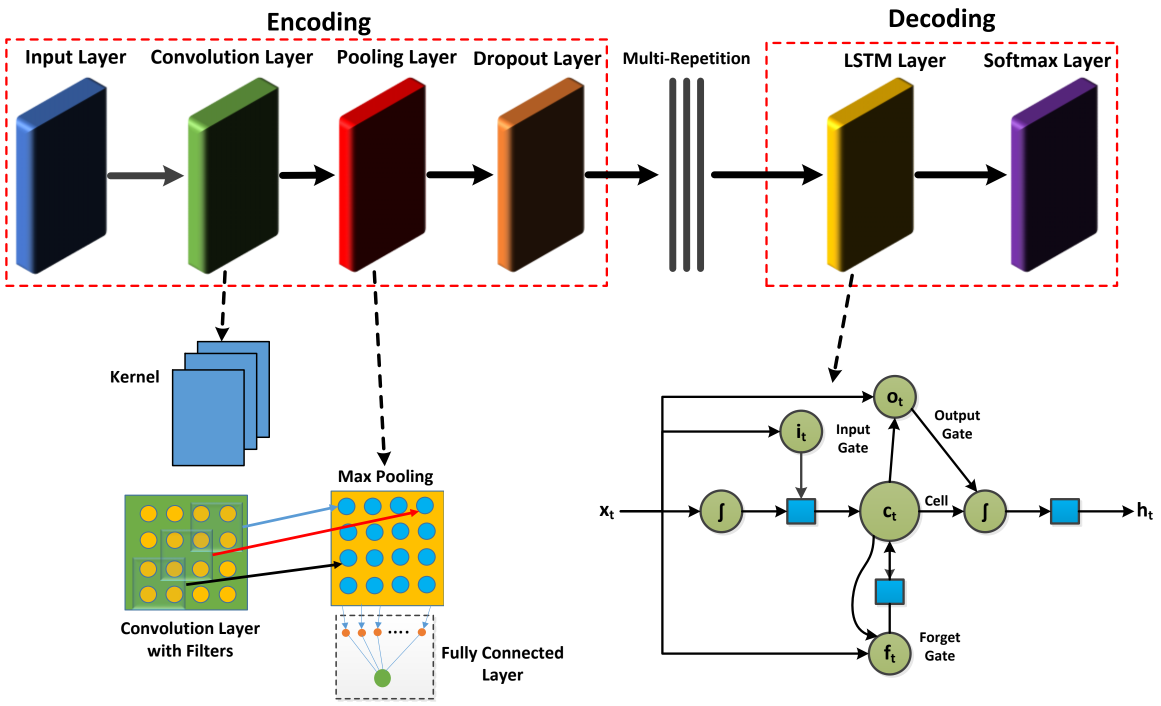


Figure 6: The CNN-LSTM Architecture for Cross-Language Software Classification

4. Results and Discussions

The proposed method is tested with comprehensive experiments which are described in the following sections.

4.1 Dataset

We utilized gist crawler to collect software applications from GitHub, which crawls through the GitHub repositories and downloads C++, Java, and C# source codes. We searched over 400K repositories and obtained over 10k software programs for each source code. Except for some descriptive files, many of these retrieved repositories did not contain any application software or code. These repositories were manually removed and then, the proposed approach is evaluated using the specified software applications. The detailed information about the dataset such as the number of selected applications in each programming code, edges, nodes, and TFIDF features are shown in Table 1.

Table 1: Datasets Details in three different programming languages

Programming Languages	Applications	Edges in AST-MDrep Features	Nodes in AST-MDrep Features	TFIDF AST-MDrep Features
C++	7,432	1,153,539	1,115,376	2,018,346
Java	9,576	990,321	937,046	1,636,563
C#	8,828	923,755	903,497	1,526,478

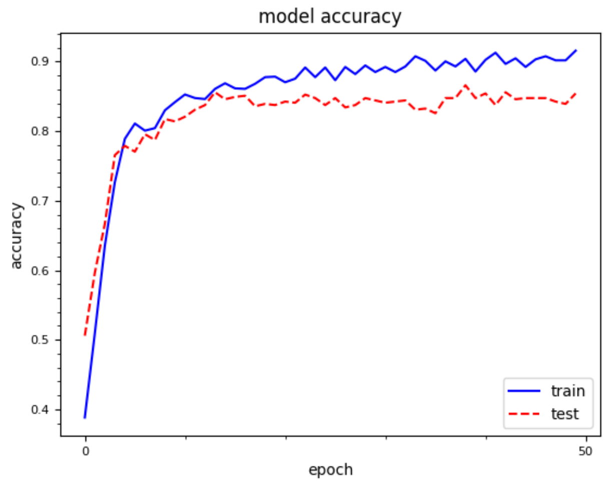
4.2 Performance Analysis and Evaluation Matrices

Because we have high-dimensional data with thousands of features, each programming code contains them. We established standard training and testing ratios, such as 80%, 20%, as indicated by several studies. We used five types of evaluation metrics for quantitative analysis: precision, recall, f-measure, accuracy, and confusion matrix. The number of True Positives (TPs) and False Positives (FPs) represents the number of AST-MDrep classified as true or false. Similarly, the number of True Negatives (TNs) and False Negatives (FNs) specify the number of categorized AST-MDrep features as true or false for the relevant programming code. An accuracy matrix is used to measure the overall classification performance. This is equal to the total number of instances divided by the number of successfully classified instances. Equations 7 and 8 show the evaluation matrices.

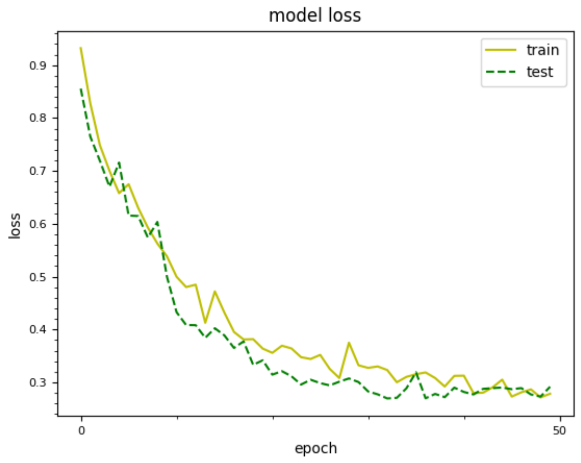
$$Recall = \frac{FP}{FP+TN},\ Precision = \frac{TP}{TP+FP} \qquad (7)$$

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN},\ F - measure\ \frac{2*TP}{2TP+FP+FN} \qquad (8)$$

The weighted AST-MDrep features are sparse and noisy, which may have an impact on deep learning classification accuracy. The TFIDF approach is used to compute the weighted AST-MDrep features, which are subsequently fed into the CNN-LSTM deep learning model. Figure 7 shows the accuracy and loss epoch curves, which were used to assess the importance of the weighted AST-MDrep features. The blue and red curves in portion (a) represent the train and test data. The training curve begins at 20% at epoch 0 and steadily grows to 82%. After that, it is more or less steady, reaching a maximum of 90%. While the testing curve begins at 50% and progresses to 77%. According to the training data, the testing curve abruptly declines after hitting 85%. The testing curve acts slightly differently, indicating that there may be an overfitting issue. As can be observed, the training data has a higher accuracy rate than the testing data. The yellow and green curves in portion (b) illustrate the loss of training and testing data, respectively. The training loss begins at 95% and drops to 30% on the 35th epoch. The testing loss curve begins at 86% and decreases concurrently with the training curve to a minimum of 25%. On the 40th epoch, both the training and testing curves are more or less constant. Figure 8 shows the confusion matrix to analyze the accurate classification and miss classification for each programing language. It can be seen that C# and Java have more closed classification rates with 88%, and 90%, respectively. On the other hand, C++ has 78% classification and 22% miss classification. The C#, and Java have more classification rates according to C++ because these two programming languages have more similar syntax. Moreover, the precision, recall, and f-measure metrics are calculated for weighted AST-MDrep features to analyze these space features more comprehensively. Table 1 shows the precision, recall, f-measure, and overall classification accuracy for weighted AST-MDrep features using CNN-LSTM deep learning model. The weighted average precision, recall, f-measure, and classification accuracy are 84%, 90%, 85%, and 85%, respectively. The reason for the low classification accuracy is because TFIDF features are directly fed into the deep learning model. These sparse and noisy features can be further filtered in order to detect the most essential features in a reduced dimensional space.



a) Accuracy curve using TFIDF and CNN-LSTM



b) Loss curve using TFIDF and CNN-LSTM

Figure 7: Epoch curves of Accuracy and Loss on Train and Test data using a combined approach of TFIDF and CNN-LSTM

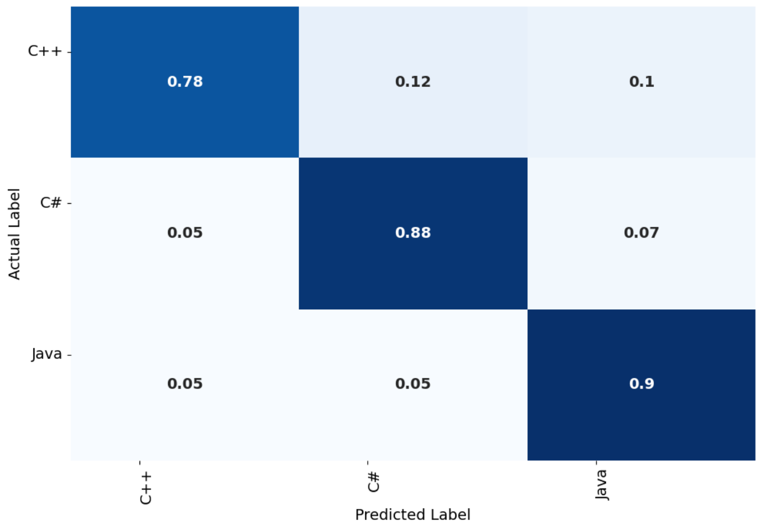
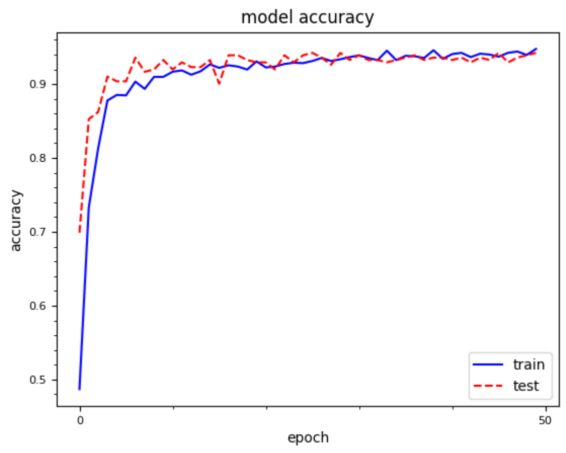


Figure 8: Confusion Matrix on Train and Test data using a combined approach of TFIDF and CNN-LSTM

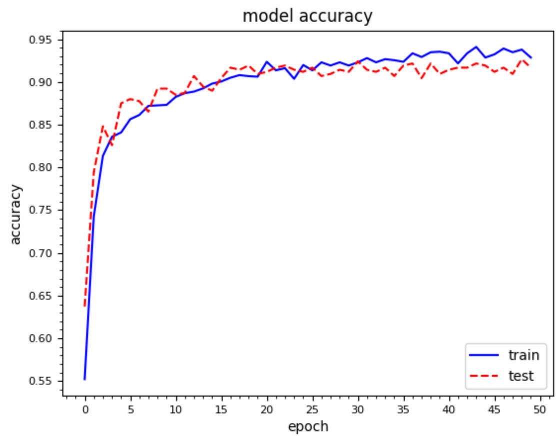
Table 2: Evaluation measures using combined approach of TFIDF and CNN-LSTM

Program Language	**Precision (%)**	**Recall (%)**	**F-measure (%)**	**Classification Accuracy (%)**
C++	89	78	83	85
C#	84	88	86	
Java	84	90	87	
Weighted Average	**86**	**85**	**85**	

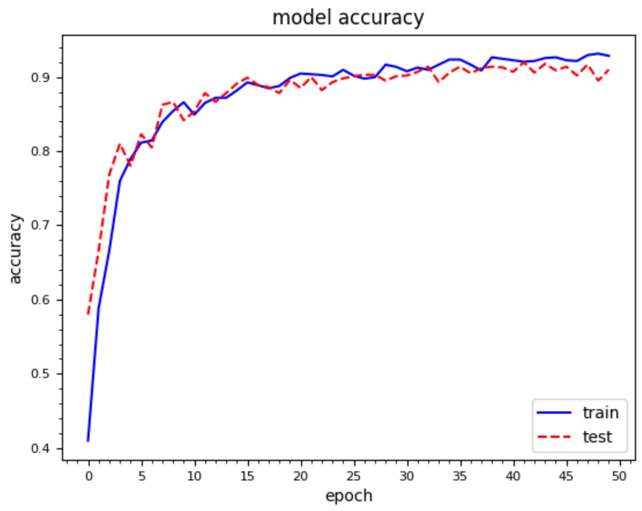
The semantic features are obtained by filtering the weighted AST-MDrep features. The LSA method is used to extract the most relevant features, also known as semantic anchors, for the CNN-LSTM deep learning classification approach. The SVD algorithm is LSA's hidden power, used to filter noisy weighted features and choose the most significant features in reduced dimensional space. Following that, CNN-based level 2 feature extraction is utilized to mine the features deeper before incorporating them into the combined CNN-LSTM deep learning model. The CNN-based deep features are then put into the designed deep learning approach for efficient classification performance. As shown in Figure 9, we developed three different deep learning approaches such as CNN-LSTM, CNN-RNN, and CNN-GRU, to compare the effectiveness of the proposed approach. It shows the training and testing curves for each deep learning approach using LSA-based semantic anchors. The parts (a, b, and c) show the training and testing accuracy curves for CNN-LSTM, CNN-RNN, and CNN-GRU, respectively. Part (a) shows that the testing curve behaves slightly different than the training curve from epoch 0 to the 20th, and then becomes more or less constant. In comparison to CNN-RNN and CNNN-GRU, the training and test curves in CNN-LSTM behave better after the 25th epoch. In CNN-LSTM, the epoch curves go up to 95% while CNN-RNN is 93% and CNN-GRU 92%. The reason behind this, training and testing curves behave abnormally after the 30th epoch as the testing curve drop down according to the testing curve. Similarly, in CNN-GRU, the testing curves drop down according to the training curve after the 35th curve. Thus, the CNN-LSTM deep learning approach provides better performance as compared to CNN-RNN and CNN-GRU.



a) CNN-LSTM



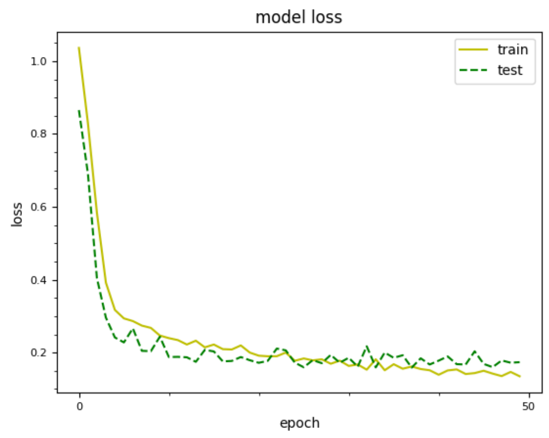
b) CNN-RNN



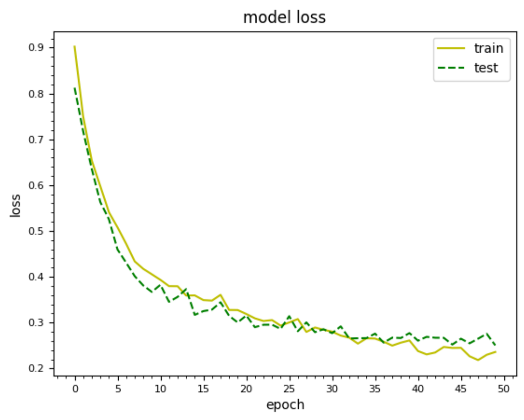
c) CNN-GRU

Figure 9: Comparisons of accuracy Curves on train and test data for CNN-LSTM, CNN-RNN, and CNN-GRU using a combined approach of Weighted AST-MDrep Features, and LSA

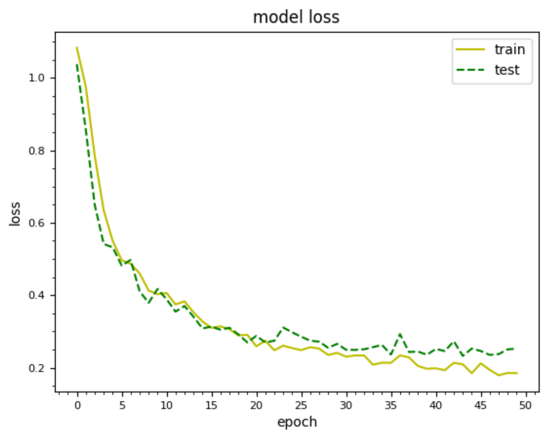
In portions (a, b, and c), figure 10 shows the training and testing loss curves for CNN-LSTM, CNN-RNN, and CNN-GRU, respectively. The training loss curve is smooth, whereas the testing loss curve has ups and downs. It can be noticed that the training and testing curves in CNN-LSTM are more stable when compared to the other two algorithms tested. The loss curves for training and testing are approximately 40% on the 10th epoch and remain relatively stable from the 20th to the 38th epochs. After the 40th epoch, the curves begin to behave irregularly, affecting the overall loss values. When compared to the other models, the CNN-GRU technique provides the lowest loss values. The overall loss values for CNN-LSTM, CNN-RNN, and CNN-GRU are approximately 10%, 20%, and 25%, respectively. Figure 11 shows the AUC curves for the three state-of-the-art deep learning approaches using LSA-based AST-MDrep features.`



a) CNN-LSTM



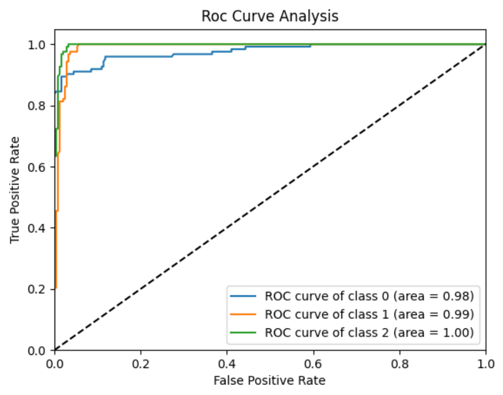
b) CNN-RNN



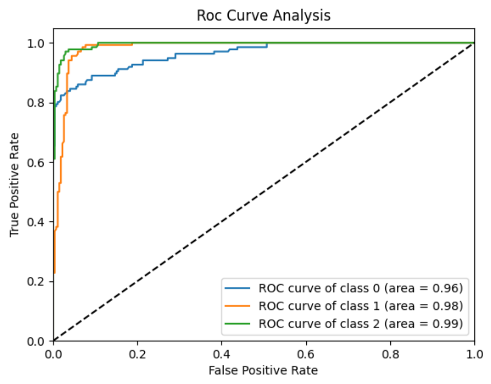
c) CNN-GRU

Figure 10: Comparisons of loss curves on train and test data for CNN-LSTM, CNN-RNN, and CNN-GRU using a combined approach of Weighted AST-MDrep Features, and LSA

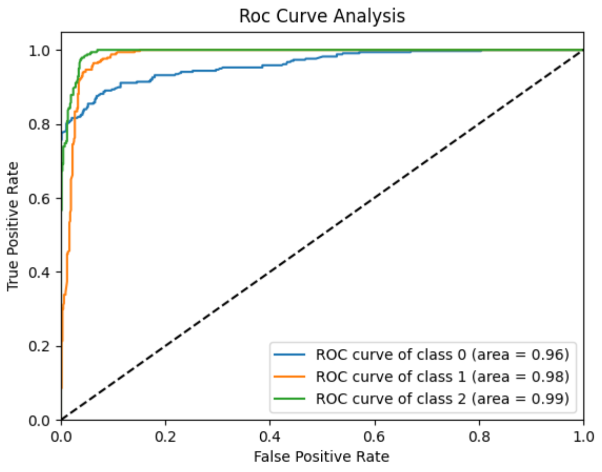
The roc curve is obtained by plotting the True Positive Rate (TPR) with the False Positive Rate (FPR). Equation 8 is used to compute the true and false-positive cases. Figure 11 shows the roc curves for the three state-of-the-art deep learning models. In part a, the roc curves are appearing closer to the top left corner of the graph, indicating that the proposed CNN-LSTM model has achieved better classification accuracy. While in parts (a, and b), the roc curves are not as close to the left corner of the graphs, indicating that these two models perform worse than part a. Figure 12 shows the confusion matrices for the three deep learning algorithms, which can be used to evaluate the miss classification rate. Parts (a, b, and c) illustrate the confusion matrices for CNN-LSTM, CNN-RNN, and CNN-GRU, respectively. It can be seen that the classification and miss classification rates for C# and Java are closed, which is due to the fact that their coding styles are similar. For instance, using the CNN-LSTM approach, the C++, C#, Java have 87%, 97, and 99% classification rates, respectively. It means that C++, C#, and Java have miss classification rates of 13%, 3%, and 1%, respectively, indicating that C++ has the maximum miss classification rate when compared to the other two programming languages, C#, and Java. In CNN-RNN, the C++, C#, Java have 85%, 93%, and 98% classification rates, respectively. It means that C++, C#, and Java have 15%, 7%, and 2% miss classification rates respectively. Similarly, in CNN-GRU, the C++, C#, Java have 81%, 96%, and 96% classification rates, respectively.



a) CNN-LSTM

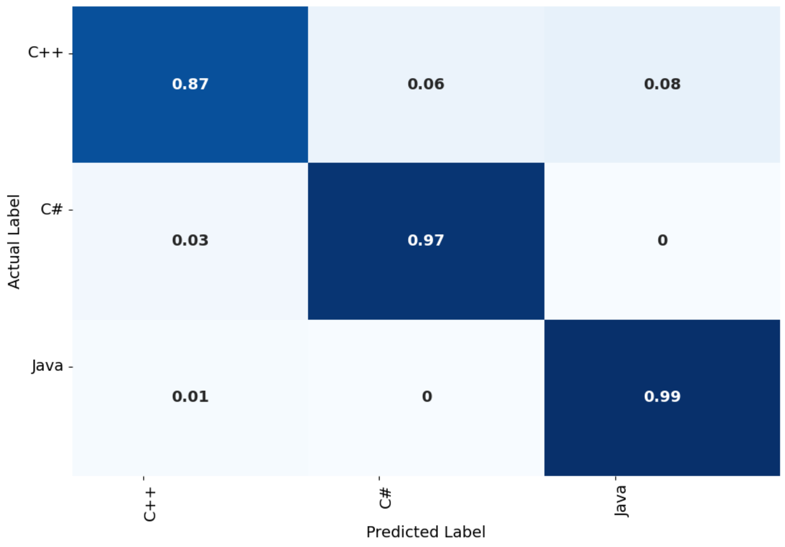


b) CNN-RNN

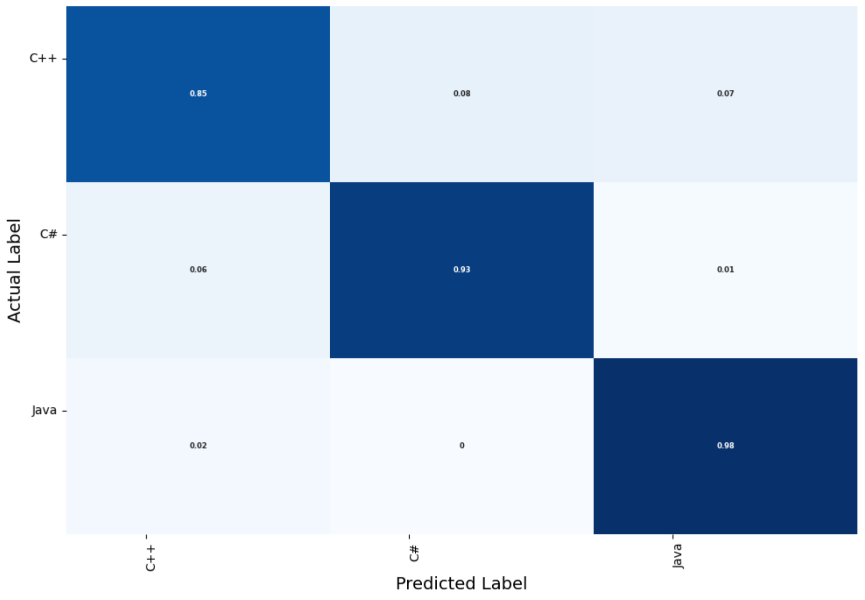


c) CNN-GRU

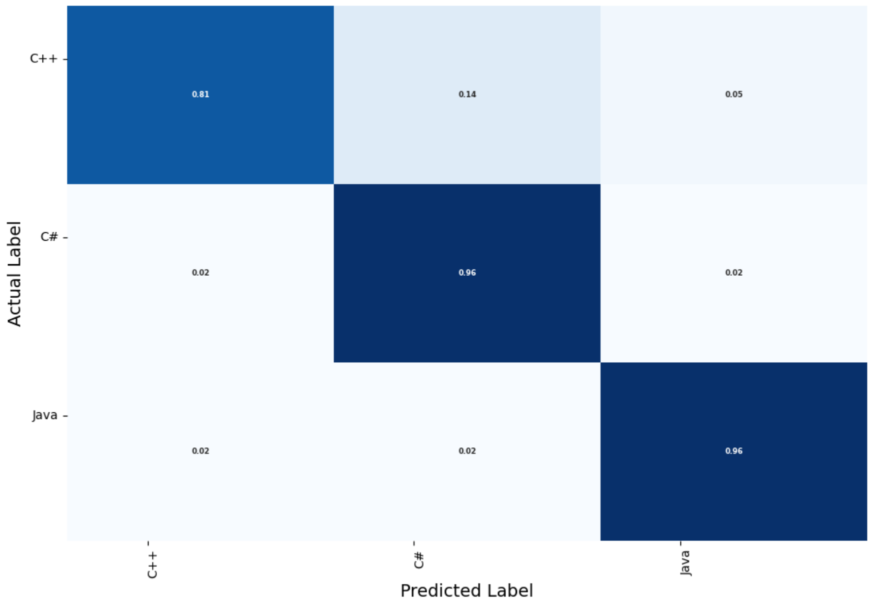
Figure 11: Comparisons of roc curves for CNN-LSTM, CNN-RNN, and CNN-GRU using combined approach of Weighted AST-MDrep Features, and LSA



a) CNN-LSTM



b) CNN-RNN



c) CNN-GRU

Figure 12: Comparisons of confusion Matrices for CNN-LSTM, CNN-RNN, and CNN-GRU using a combined approach of Weighted AST-MDrep Features and LSA

Table 3 compares the performance of CNN-LSTM, CNN-RNN, and CNN-GRU in each programming language in terms of precision, recall, and f-measure. When compared to the other approaches, the CNN-LSTM has higher precision, recall, and f-measure values. On the other hand, the CNN-GRU has the lowest performance. Table 4 shows the weighted average of precision, recall, and f-measure values for each programming language. The precision, recall, f-measure for CNN-LSTM, CNN-RNN, and CNN-GRU have (94%, 94%, 94%), (92%, 93%, 91%), and (91%, 91%, 91%), respectively. It can be seen that the proposed approach, i.e. CNN-LSTM has the highest weighted averages as compared to other deep learning models. Table 5 shows the overall classification accuracy for the three deep learning models. The classification accuracy of CNN-LSTM, CNN-RNN, and CNN-GRU have 94%, 92%, and 91%, respectively. In addition, as shown in Table 6, the proposed study is compared to already published works. Gharehyazie et al. [9], proposed the cross-application clone detection in GitHub repositories to learn more about code sharing and reuse. They used the proposed method with the Deckard tool to classify 5,753 Java projects with an accuracy of 80%. Gang et al. [2], used the logical features using Control Flow Graph (CFG) with Deep Neural Network (DNN) and achieved an accuracy of 82%. Kawser et al. [20], proposed CroLSim, which uses API description to evaluate code similarities. The source code is tokenized and then given these features to LSA to extract semantics. Later, the LSA-based code semantics are then input to the deep learning model for classification. Yi et al. [43], extracted the AST features from programming codes to study the syntactic information. The term embedding method, in combination with the Euclidean distance method, is then utilized to compute code similarity. This approach has a classification accuracy of 88%, which is quite impressive. Nguyen et al. [7], proposed a semantics-based method that is used to extract program flow features. The code is compiled into graph structures, which are then mined for code similarities across programming languages. This method achieves a classification accuracy of 80%. Yuede et al. [44], proposed a hybrid approach of Attributed Control Flow Graph (ACFG), and Graph Triplet Loss Network (GTN) to extract code similarity. The program flow is collected using ACFG, and a Tensorflow-based GTN model is used to rank related functions across many software projects. The proposed method obtains a classification accuracy of 90%, which is quite good.

Table 3: Performance comparisons for each class of programming language using AST-MDrep features, and LSA

CNN-LSTM			
Program Language	**Precision (%)**	**Recall (%)**	**F-measure (%)**
C++	100	83	91
C#	89	100	94
Java	94	98	96
CNN-RNN			
C++	91	85	88
C#	92	93	92
Java	92	98	95
CNN-GRU			
C++	95	81	88
C#	86	96	91
Java	93	96	94

Table 4: Performance comparisons of weighted evaluation measures using AST-MDrep features, and LSA

Models	**Precision (%)**	**Recall (%)**	**F-measure (%)**
CNN-LSTM	94	94	94
CNN-RNN	92	93	91
CNN-GRU	91	91	91

Table 5: Performance comparisons of classification accuracy using AST-MDrep features, and LSA

Models	Classification Accuracy (%)
CNN-LSTM	94
CNN-RNN	92
CNN-GRU	91

Table 6: Performance comparisons of classification accuracy with published works

Related Work	Year	Method	Classification Accuracy (%)
Gharehyazie et al. [9]	2017	Deckard	80
Gang et al. [2]	2018	CFG with DNN	82
Kawser et al. [20]	2018	LSA with CNN	82
Yi et al. [43]	2019	Term embedding with AST	88
Nguyen et al. [7]	2020	Graph-based Semantic Features	80
Yuede et al. [44]	2021	ACFG with GTN	90
Proposed approach	….	LSA-based MDrep with CNN-LSTM	94

4.3 Threat to Validity

Sample Size: The sample size of our analysis may not be enough to represent the complete ecology of a software repository. However, we selected approximately 9.5K Java, 8.8K C#, and 7.5 K C++ software applications from GitHub that is much huge in size compared to several of the relevant systems. Because we were able to immediately use GitHub resources without any modification or fiction, we can certainly say that our proposed approach can be capable of operating with other platforms too.

Methods documentation: A possible challenge to the proposed methodology is the lack of or low quality of methods description. However, we are able to mitigate this risk by referring to the documentation provided by the standard organization such as Microsoft, Oracle, etc. we can surely rely on the methods descriptions provided by these standard organizations. We also made ensured that these descriptions are presented in an understandable manner.

Structural Information: The structural information may not be enough to analyze the semantic information of different programming codes. To overcome this threat, we used the combination of AST and methods description to address the syntactic structure as well as contents details of different programming codes. Moreover, we designed level-1 and level-2 features extraction strategies to mine the most significant code semantics for the CNN-LSTM model.

Evaluation measures: One may even argue that our selection of evaluation measures is inappropriate. To avoid these risks, we adapted widely used measures for such research from previous work, including precision, recall, f-measure, and classification accuracy. To demonstrate the effectiveness of the proposed approach, we developed a complete evaluation comparison of CNN-LSTM with CNN-RNN and CNN-GRU. Additionally, it is shown that Level-1 and Level-2 feature extraction schemes are significantly more effective than simple TFIDF features.

5. Conclusion

Finding the relevant programming codes and repurposing them for their own use is a common activity for software developers. Due to this, a large number of software applications have been developed to solve the same problems. The development of similar software applications has become more popular as the open-source community has grown. However, the OSS repositories are not semantically ranked according to the functionalities of the given codes. Furthermore, these codes may contain buggy and outdated information which leads to the compilation problem for the new software. This study introduces a new tool called CroLSSim for detecting software similarities across different programming languages. First, the AST features are gathered from various source codes. These are high-quality features that show each program's abstract view. Then, the combination of AST and MDrep is used to explore the relationships between different method calls. Second, TFIDF is utilized to get local and global weights. Third, the Singular Value Decomposition (SVD) method is used to obtain meaningful features in reduced dimensional space. Then, level-1 and level-2 features extraction strategies are designed to mine the code semantics. Finally, a CNN-LSTM hybrid model is proposed to recognize semantically similar software applications from code semantics.

The proposed method can be extended to include three heuristics. Software applications starred by the same users within a short amount of time are probably similar, applications starred by similar users can be similar, and applications with similar contents in readme files can be similar. In addition, we intend to investigate the advantages of employing CFG-based features in combination with method descriptions for finding similar software applications.

References:

1. Prana, G.A.A., et al., *Categorizing the content of GitHub README files.* Empirical Software Engineering, 2019. **24**(3): p. 1296-1327.
2. Zhao, G. and J. Huang. *Deepsim: deep learning code functional similarity*. in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018.
3. Nguyen, P.T., et al. *Focus: A recommender system for mining api function calls and usage patterns*. in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019. IEEE.
4. Nafi, K.W., et al., *A universal cross language software similarity detector for open source software categorization.* Journal of Systems and Software, 2020. **162**: p. 110491.
5. Zhang, Y., et al. *Detecting similar repositories on GitHub*. in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017. IEEE.
6. Hussain, S., J. Keung, and A.A. Khan, *Software design patterns classification and selection using text categorization approach.* Applied soft computing, 2017. **58**: p. 225-244.
7. Nguyen, P.T., et al., *An automated approach to assess the similarity of GitHub repositories.* Software Quality Journal, 2020: p. 1-37.

8. Fang, C., et al. *Functional code clone detection with syntax and semantics fusion learning*. in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020.
9. Gharehyazie, M., B. Ray, and V. Filkov. *Some from here, some from there: Cross-project code reuse in github*. in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017. IEEE.
10. Tian, Z., et al., *Reviving sequential program birthmarking for multithreaded software plagiarism detection.* IEEE Transactions on Software Engineering, 2017. **44**(5): p. 491-511.
11. Liu, S. *A Unified Framework to Learn Program Semantics with Graph Neural Networks*. in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020. IEEE.
12. Vincze, E.A., *Challenges in digital forensics.* Police Practice and Research, 2016. **17**(2): p. 183-194.
13. Ullah, F., et al., *Cyber security threats detection in internet of things using deep learning approach.* IEEE Access, 2019. **7**: p. 124379-124389.
14. Kawaguchi, S., et al., *Mudablue: An automatic categorization system for open source repositories.* Journal of Systems and Software, 2006. **79**(7): p. 939-953.
15. McMillan, C., M. Grechanik, and D. Poshyvanyk. *Detecting similar software applications*. in *2012 34th International Conference on Software Engineering (ICSE)*. 2012. IEEE.
16. Roy, C.K., J.R. Cordy, and R. Koschke, *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach.* Science of computer programming, 2009. **74**(7): p. 470-495.
17. Prechelt, L., G. Malpohl, and M. Philippsen, *Finding plagiarisms among a set of programs with JPlag.* J. UCS, 2002. **8**(11): p. 1016-.
18. Bowyer, K.W. and L.O. Hall. *Experience using" MOSS" to detect cheating on programming assignments*. in *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011*. 1999. IEEE.
19. Wise, M.J., *String similarity via greedy string tiling and running Karp-Rabin matching.* Online Preprint, Dec, 1993. **119**(1): p. 1-17.
20. Nafi, K.W., et al. *Crolsim: Cross language software similarity detector using api documentation*. in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2018. IEEE.
21. Kalliamvakou, E., et al. *The promises and perils of mining github*. in *Proceedings of the 11th working conference on mining software repositories*. 2014.
22. Jalili, V., et al., *The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update.* Nucleic acids research, 2020. **48**(W1): p. W395-W402.
23. Bogomolov, E., et al. *Sosed: a tool for finding similar software projects*. in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020. IEEE.
24. Teixeira, G., J. Bispo, and F.F. Correia. *Multi-language static code analysis on the LARA framework*. in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 2021.
25. Naeem, H., et al., *Malware detection in industrial internet of things based on hybrid image visualization and deep learning model.* Ad Hoc Networks, 2020. **105**: p. 102154.

26. Jiang, L., et al. *Deckard: Scalable and accurate tree-based detection of code clones*. in *29th International Conference on Software Engineering (ICSE'07)*. 2007. IEEE.
27. Koskela, M., I. Simola, and K. Stefanidis. *Open source software recommendations using github*. in *International Conference on Theory and Practice of Digital Libraries*. 2018. Springer.
28. Nichols, L., et al. *Syntax-based Improvements to Plagiarism Detectors and their Evaluations*. in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 2019. ACM.
29. Kikuchi, H., et al. *A source code plagiarism detecting method using alignment with abstract syntax tree elements*. in *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 2014. IEEE.
30. Fu, D., et al., *WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection.* Scientific Programming, 2017. **2017**.
31. Ankali, S.B. and L. Parthiban, *Detection and Classification of Cross-language Code Clone Types by Filtering the Nodes of ANTLR-generated Parse Tree.* International Journal of Intelligent Systems & Applications, 2021. **13**(3).
32. Karnalim, O. and W. Chivers. *Preprocessing for source code similarity detection in introductory programming*. in *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 2020.
33. Paik, J.H. *A novel TF-IDF weighting scheme for effective ranking*. in *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. 2013.
34. Bauer, V., T. Völke, and S. Eder. *Combining clone detection and latent semantic indexing to detect re-implementations*. in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2016. IEEE.
35. Landauer, T.K., et al., *Handbook of latent semantic analysis*. 2013: Psychology Press.
36. Cosma, G. and M. Joy, *An approach to source-code plagiarism detection and investigation using latent semantic analysis.* IEEE transactions on computers, 2011. **61**(3): p. 379-394.
37. Flores, E., et al., *Cross-Language Source Code Re-Use Detection Using Latent Semantic Analysis.* J. UCS, 2015. **21**(13): p. 1708-1725.
38. Wei, H. and M. Li. *Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code*. in *IJCAI*. 2017.
39. Hua, W., et al., *FCCA: Hybrid code representation for functional clone detection using attention networks.* IEEE Transactions on Reliability, 2020. **70**(1): p. 304-318.
40. Zhang, J., et al. *A novel neural source code representation based on abstract syntax tree*. in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019. IEEE.
41. Hamdy, A. and G. Ezzat, *Deep mining of open source software bug repositories.* International Journal of Computers and Applications, 2020: p. 1-9.
42. Perez, D. and S. Chiba. *Cross-language clone detection by learning over abstract syntax trees*. in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019. IEEE.
43. Gao, Y., et al. *Teccd: A tree embedding approach for code clone detection*. in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019. IEEE.

44. Ji, Y., L. Cui, and H.H. Huang. *Buggraph: Differentiating source-binary code similarity with graph triplet-loss network*. in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021.