CroLSim: Cross Language Software Similarity Detector using API documentation

Kawser Wazed Nafi, Banani Roy, Chanchal K. Roy and Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada
Email: {kawser.nafi, banani.roy, chanchal.roy, kevin.schneider} @usask.ca

***Abstract*—In today's open source era, developers look for similar software applications in source code repositories for a number of reasons, including, exploring alternative implementations, reusing source code, or looking for a better application. However, while there are a great many studies for finding similar applications written in the same programming language, there is a marked lack of studies for finding similar software applications written in different languages. In this paper, we fill the gap by proposing a novel model *CroLSim* which is able to detect similar software applications across different programming languages. In our approach, we use the API documentation to find relationships among the API calls used by the different programming languages. We adopt a deep learning based word-vector learning method to identify semantic relationships among the API documentation which we then use to detect cross-language similar software applications. For evaluating *CroLSim*, we formed a repository consisting of 8,956 Java, 7,658 C#, and 10,232 Python applications collected from GitHub. We observed that *CroLSim* can successfully detect similar software applications across different programming languages with a mean average precision rate of 0.65, an average confidence rate of 3.6 (out of 5) with 75% high rated successful queries, which outperforms all related existing approaches with a significant performance improvement.**

***Index Terms*—API Calls, Paragraph2Vec, Cross-Language Software Similarity Detection, Singular Value Decomposition**

I. Introduction

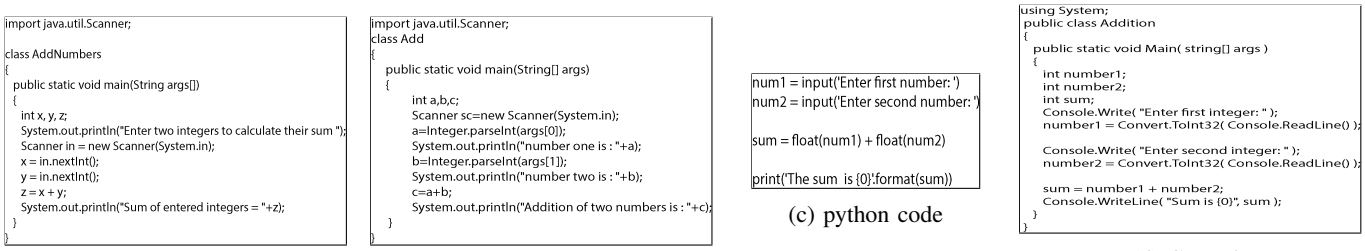
Research into source code mining and development of software applications to solve existing computational problems has played an important role for a long time with software researchers and developers. As a result, the number of software applications solving the same problem is increasing substantively in open-source software repositories. A report from October 2017 [15] shows that GitHub consists of 24M users with 67M repositories written in different programming languages. As a result it is helpful to group software applications performing the same tasks together so that users can more easily select a software application to solve their problem. In addition, it can help when open source repositories automatically recommend similar code examples or software applications within a short time with little overhead [8]. Managing and providing services in big software repositories like GitHub, SourceForge, BitBucket and so on are considered a big data management problem. Recently software researchers are developing common platforms for users and researchers working in different areas that can combine a variety of software applications and tools in workflows [1], [2] to satisfy their needs. By executing the workflows, researchers perform different analytic jobs and get useful research insights efficiently. Some examples of this type of platform are *Galaxy* [3], *iPlantCollaborative* [4], and *ensemble plants* [5]. If a good number of software applications performing similar kinds of functionality are present in a common repository, it can help users to check and select the best suited one as per their requirements (e.g., diversity in the functionality of a software application, a specific API, the number of times an application or API is used by other users in the repository, and so on). In addition, detecting similar software applications also is useful for addressing other computer science research problems, such as re-usability of software applications, rapid prototyping [6], detecting source code plagiarism [7], code clones detection [45], and so on.

TABLE I: Source Code Similarity Score for the applications in Figure 1

Programming Language	Java	C#	Python
Java	1	.062	0.021
C#	.0621	1	.075
Python	0.021	.075	1

Detecting similar software applications based on source code similarity within the same programming language is relatively straightforward and there are already promising approaches [8], [9], [45]. Usually for source code similarity detection each word (i.e., API names, Class names, used Library Method names, syntactic information, etc.) in the source code is considered a *semantic anchor* which helps to define the semantic characteristics of that document [9]. Let us consider the Java applications in Figure 1a and Figure 1b which add two integer numbers. Using the Vector Space Model (VSM)[1] [16] we found that, these two applications have a 76% match based on their source code semantic anchors. Unfortunately, for different programming languages the API names, identifiers, code structure, syntactic information and so on are often different. We applied VSM on the code examples in Figures 1a, 1c and 1d and the semantic anchor similarity results are given in Table I. From this table we see an almost zero semantic relationship among the API names and the identifiers used by Java, Python and C# programming languages. As a consequence, none of the source code similarity approaches focus on cross language software application similarity de-

[1]https://media.readthedocs.org/pdf/gensim/stable/gensim.pdf



```
import java.util.Scanner;

class AddNumbers
{
  public static void main(String args[])
  {
    int x, y, z;
    System.out.println("Enter two integers to calculate their sum ");
    Scanner in = new Scanner(System.in);
    x = in.nextInt();
    y = in.nextInt();
    z = x + y;
    System.out.println("Sum of entered integers = "+z);
  }
}
```

(a) Java code one

```
import java.util.Scanner;
class Add
{
    public static void main(String[] args)
    {
        int a,b,c;
        Scanner sc=new Scanner(System.in);
        a=Integer.parseInt(args[0]);
        System.out.println("number one is : "+a);
        b=Integer.parseInt(args[1]);
        System.out.println("number two is : "+b);
        c=a+b;
        System.out.println("Addition of two numbers is : "+c);
    }
}
```

(b) Java code two

```
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

sum = float(num1) + float(num2)

print('The sum  is {0}'.format(sum))
```

(c) python code

```
using System;
public class Addition
{
  public static void Main( string[] args )
  {
    int number1;
    int number2;
    int sum;
    Console.Write( "Enter first integer: " );
    number1 = Convert.ToInt32( Console.ReadLine() );

    Console.Write( "Enter second integer: " );
    number2 = Convert.ToInt32( Console.ReadLine() );

    sum = number1 + number2;
    Console.WriteLine( "Sum is {0}", sum );
  }
}
```

(d) C# code

Fig. 1: Code examples for adding two integers from the console written in four different programming languages

tection. There are some other challenges too. For example, the third party APIs used for developing software applications differ among programming languages. As well, the byte code generated by different programming language compilers is also different considering their semantic and syntactic aspects [13]. Considering all these challenges it can be said that software similarity detection across different programming languages is a challenging problem to solve. But with the increase of open source software applications, detecting cross-language similar software applications is indisputably an important need.

Although recent studies [10], [12] indicated that their approaches could be extended to detect cross-language similar software applications, both works have a good number of restrictions. For example, Thung et al. [12] used collaborative tags for similarity detection which are hardly found associated with software applications in a repository. Very often collaborative tags give the wrong impression about software applications as these tags are provided by a user's experience which may vary from user to user and may cover a small part of the whole application [37]. Thus, figuring out a universal solution for detecting and searching similar software applications across different programming languages for any open source software repositories is still a challenge to solve.

In this paper, we propose a novel approach **CroLSim** which is able to detect similar software applications across different programming languages. For establishing a connection between different programming languages, we leveraged the descriptions of APIs and their methods[2][3][4] which briefly state the purpose of a specific package and methods for the APIs used in developing a software application. We observed that although APIs and their method names are semantically different across programming languages, the methods performing the same operations usually have semantically similar descriptions (as shown in Figure 2). Realizing this fact, we first collected descriptions of all the APIs available for different languages and built a **Corpus** with those descriptions. For each of the programming languages, we maintained an individual XML file in our *Corpus*. Second, we extracted all the API calls (including the third party ones too) from the source code of each of the software applications of our repository. Third, for each of the applications, we queried the Corpus with the names of the extracted API calls and retrieved the related descriptions. Then we combined all the descriptions in a single file for a software application and considered this as a descriptive representational document (in short *DReP*) for that subject software. From our observations we noticed that there are also not many similarities between the structures of the descriptions of similar APIs across different programming languages. In order to mitigate this challenge, we adapted the distributed memory model of Paragraph Vectors (**PV_DM**) model [11] which is a deep convolutional neural network based advanced Natural Language Processing (NLP) model for predicting semantic information from the API descriptions. We applied **PV_DM** over DReP for each of the software applications to find semantic similarity between them. To the best of our knowledge, we are the first to have attempted to design a generalized model (not dependent on any of the specific open source repositories such as GitHub, SourceForge, BitBucket and so on) for detecting cross-language similar software applications by leveraging the semantic meaning of API call descriptions. We also clustered similar software applications with the help of the semantic weights that are generated.

To evaluate our work, we performed an extensive number of experiments supporting our claim that our model can detect similar software applications across different programming languages resulting in a mean average precision of 0.65, an average confidence score of 3.6 (out of 5) and an average of more than 72% in highly successful queries of the top-5 recommendations of software applications.

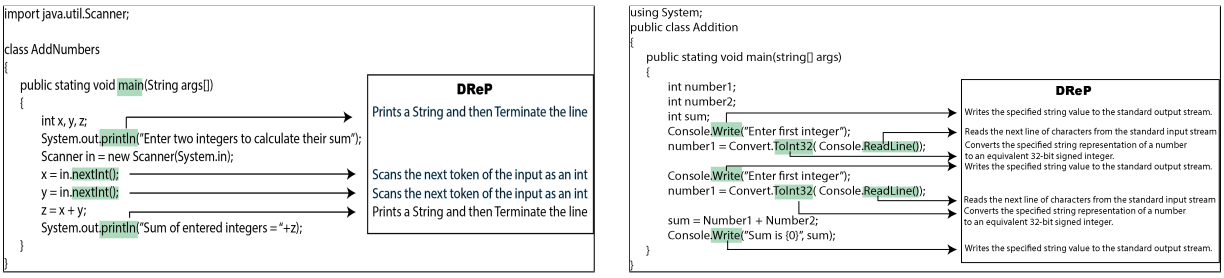
II. Motivation

Let us assume a software developer, who is good with C# but does not have good knowledge of Java and Python, needs to develop a feasible software solution for adding two different integer numbers in those programming languages. They are good in C# and can easily build the application given in Figure 1d. Now, to develop the same solution in Python and Java, they start looking for help in GitHub, one of the most popular and largest software repositories. They search for a sample code in GitHub with the following two queries, a. "Addition in Java" and b. "Addition in Python". They actually needs the code block given in Figure 1a for the Java application and Figure 1c for the Python application. Querying in GitHub with (a.) results in 168 Java applications where the very related code example to Figure 1c pops up

[2]Java Docs: https://docs.oracle.com/Javase/8/docs/api/

[3]Python 2.7 Docs: https://docs.python.org/2/library/index.html

[4]C# docs:https://docs.microsoft.com/en-us/dotnet/api/?view=netcore-2.0



(a) Methods Extraction and DReP creation for Java Program (b) Methods Extraction and DReP creation for C# Program

Fig. 2: DRePs created using source code analysis

after 21 software applications. For querying with (b.) 119 Python samples are found where the needed one appears after 35 recommendations. The reason behind this is that GitHub recommends only those software repositories whose name or short description matches most with the tokens of the query. The developer needs to visit each of the software applications, analyze each source code and needs to find out the required sample code block for their work. As a novice developer in Java and Python, it is really hard and time consuming to manually evaluate and find out the required code blocks given in Figure 1a and Figure 1c from these huge number of recommended software applications. They might use the same queries using Google but will still have the same challenges. The most feasible solution would be the availability of a search engine where they could search with the source code given in Figure 1d or the availability of a cross-language categorized software repository from which the developer can easily select a software application similar to the one in Figure 1d, which would be 1a in Java and 1c in Python. This scenario motivated us to develop CroLSim.

Different cloud based scientific workflow management systems are available for researchers (e.g. Galaxy [3], iPlant [4], and ClustEval [51]). In these systems, one can build workflows with different available software. When building a workflow, one may need to use different software applications which may also involve alternative ones of a certain software application. Our work will help such platforms effectively since CroLSim will categorize the similar ones based on the source code of the current one at hand.

III. CroLSim - Model and Approach

In this section, we discuss our approach to detect cross-platform similar software applications from software repositories, provide background for the methods and terminologies used in CroLSim, and present the model architecture and the way it works.

A. Approach

CroLSim can be divided into four phases. In the first phase, we try to establish a common connection among the API calls performed in each of the programming languages. For this, we use the API documentation available and maintained by providers, e.g., programming language Java and its packages and method descriptions are maintained by Oracle[5], C# is developed and maintained by Microsoft[6], and Python is maintained by the Python Software Foundation[7]. We assume that, if a single document of combined descriptions of all of the API calls used in a software application is similar to the document of the combined descriptions of API calls of another software application, we can thereby say these two applications are similar. We named these combined descriptions DrePs which we considered as a continuous bag of words (CBOW) [20] without any predefined semantic relation. Let us consider the source code examples in Figure 2. Here, Figure 2a and 2b are examples of *DRePs* generated from analyzing the source code and getting the corresponding API descriptions from the available sources/documentation.

In our second phase, we worked on reducing the effects of "common and less effective" API calls [19]. Let us consider the source code examples given in Figure 2. One of the used methods in Java source code in Figure 2a is PRINTLN() from the JDK package SYSTEM.OUT which is used to print something to the console with a newline. If we look at the C# source code in Figure 2b we see that the WRITE() method along with the CONSOLE package used here does the same thing. If we put the same importance to these methods along with other dedicated and less frequently used methods in the source code it will mislead us in Cross-Language similar software detection. Collins et al. [9] in their work also found this issue with API calls in Java source code. To deal with this, they used Singular Value Decomposition (SVD) along with Latent Semantic Indexing (LSI) to find out less used but important methods in Java programs. In our work, we adapted SVD to filter out less effective API calls in source code from all of the software applications developed in the different programming languages.

As different developers and maintainers usually use different sentence structures and formations for describing the functionality of different methods and libraries, it is worthless to work with a direct word to word match of DRePs for calculating software application similarity. Thus, in order to define semantic similarity of the contents of the **DRePs**, we adapted Le and Mikolov's proposed model **Paragraph2Vec** [11] in the third

[5]https://www.oracle.com/Java/index.html
[6]https://docs.microsoft.com/en-us/dotnet/csharp/
[7]https://www.python.org/psf/

phase of our proposed model. One of the main features of the Paragraph2Vec model is that it can predict semantic words from a paragraph or document on the fly without any pre-training. We considered these predicted semantic words as the **Semantic Anchors** [17], [18] for the DRePs of the software applications and later on we calculated Cosine similarity [21] among the **Semantic Anchors** for calculating the semantic similarities among cross-platform software applications.

B. Singular Value Decomposition

To reduce the impact of less effective API calls, which are actually used in source code of almost all of the software applications, we adapted the Singular Value Decomposition (SVD) [22] model. SVD is normally applied on a matrix **M** to factor that matrix into multiplication of three matrices such that:

$$M = ADB^T \tag{1}$$

where **A** and **B** are orthogonal and **D** is diagonal where values are positive real numbers. It is helpful to use a low rank matrix as an approximation of the high dimensional matrix.

To incorporate SVD in our work we developed a $m * n$ dimension Term-Document Matrix (TDM) [14] **M** with the help of extracted API calls. In our matrix, each row m takes the name of the API call and each column n takes the software application as input. Each cell then represents the number of times the specific API calls appear in the source code of the related software application. For every programming language **l** used in our work we developed an individual TDM matrix M_l from the source code of the software applications. Now, applying SVD on each of the M_l we have derived the low rank matrix which consists of the less but effectively used API calls in each of the software applications of the software repository. To perform the whole process we used the Information Retrieval (IR) technique Latent Semantic Indexing (LSI) [23] which embeds both TDM and SVD.

C. ParaGraph to Vector

In recent years, Natural Language Processing (NLP) research has had considerable success by utilizing deep learning techniques, which provides the ability to analyze words from a sentence to a full paragraph for identifying the semantics of a word in a more accurate way [26]. Many recent NLP models use neural networks [27], and especially Convolutional Neural Network on Word embedding [26]. Word embedding, also known as distributional vectors, represents the characteristics of surrounding words of a specific work which helps to learn the semantic meaning of a word [28], [29]. One of the recent CNN based word vector representation models is Word2Vec [30] where the convolution layer operation (the first layer in the deep learning model) is applied on word embedding to sub-sample the frequent words as a goal of training a large unlabelled corpus in a faster way.

With the help of Word2Vec it is possible to learn word vectors for a full phrase or a full sentence. That is, Word2Vec can learn word vectors for a fixed length context. To extend this work for learning word vectors from variable length contexts, Le and Mikolov's [11] proposed a new model, called 'Paragraph Vector' which is able to learn continuous distributed vectors in an unsupervised learning way from a paragraph or full document. This model predicted the vectors for a paragraph by predicting multiple word vectors from the paragraph itself.

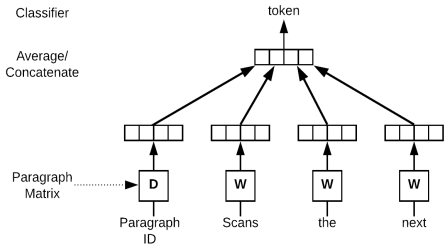


Fig. 3: Word Vector learning architecture for paragraph to vector. This Figure is adapted from [11]

Figure 3 represents the word vector learning mechanism for a paragraph. Here we are trying to learn a word vector from our Java program DReP example given in Figure 2a. Here, in Figure 3, three words from our Java DReP, 'scans', 'the' and 'next' words are trained for predicting the next **token**. All the words of a paragraph are embedded in a word matrix W. Each paragraph is uniquely identified with a Paragraph ID which is embedded in matrix D. At the training phase, the Paragraph ID vector, represented by a column in matrix D and vectors for training words, represented by the column in matrix W are popped up and are averaged or concatenated for predicting the **token** word of the context DReP. This **token** is used to memorize the missing elements in the current context. Paragraph vectors are shared among all the paragraphs of a document where word vectors of each paragraph are shared with other word vectors of that paragraph. The whole training process is done with the help of stochastic gradient decent which is usually obtained by back propagation in neural network. This model is also called as 'Paragraph Vector-Distributed Memory (PV-DM)'.

D. CroLSim's Architecture

The CroLSim architecture is shown in Figure 4. The main elements of our architecture are software repositories, source code scripts, Singular Value Decomposition (SVD), Corpus of API calls description, configuring DReP, application of Paragraph2Vec and finally an applied K-nearest neighbours (KNN) [35] algorithm for clustering similar software applications. For each of the language we maintained an individual repository. We combined all the source code individually for each of the software applications and kept them along with the software application in our software repository.

As can be seen from the figure, first we detect the programming language for each of the software applications from the repository because of variations in API calls extraction process for different languages. After extracting API calls from all the software applications we run SVD on them to find out

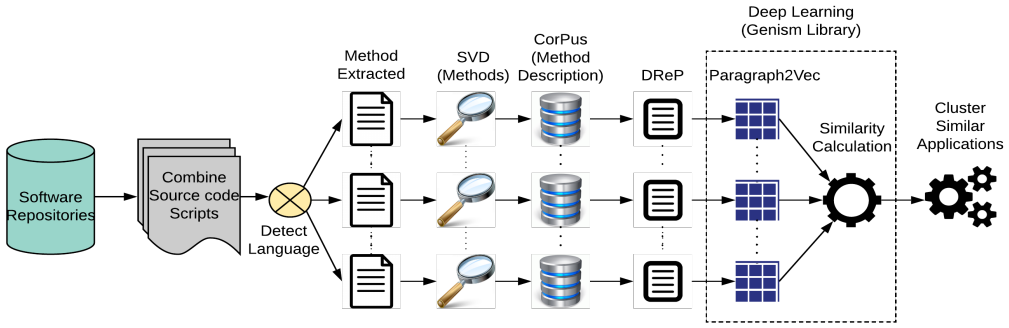


Fig. 4: Schematic diagram for CroLSim

less frequently used API calls. We considered the top 400 less frequently used API calls. After performing SVD, for each of the API calls, we queried the **Corpus** to get the API documentation for the queried API call.

Once we queried the **Corpus** with the extracted API calls, we retrieved the API documentation for each of the API calls and added them to a single text file *DReP*. For a single software application we observed that we extracted a lot of API calls which had led us to a large size DReP. Processing larger size word documents might increase computational complexity as well as require extensive hardware support. In addition, we observed that each DReP may contain repeated entries for a single API call. To overcome these issues, for each API call we kept only one entry even if it was called multiple times. In our work this would not generate any problem as we only need an informative description about the functionality of a software application. Using multiple entries of a single API description is redundant since the information provided by them altogether are equivalent to that of a single entry. For each of the software applications we generated a single DReP. There are some other popular 3rd party APIs and Libraries which developers often use inside the program for ease of software development, such as OpenCV, Apache Log4j, and Jsoup for Java, and Numpy and PyQt, for Python. To obtain documentation for these APIs we first tried to collect descriptions from verified sources. If we failed to find them, then we collected the source code and generated their documentation with the help of document generation APIs provided by different programming languages, such as Javadoc [31] for Java, Pydoc[8] for Python and DocFX [32] for C#.

Once we generated *DReP* for all the software applications in the software repository, we used all these DRePs as input to 'Paragraph2Vec'. We adapted the library and structure of 'Paragraph2Vec' from Genism [33] with some small modification to the parameters. We set the parameters of 'Paragraph2Vec' in the following way for our study: 50 epochs, 10 window size, 2000 dimensions, 0.25 learning rate and a sample size equal to the number of software applications we are considering. The reason behind considering high dimension in word embedding is that our DRePs are a collection of retrieved

[8]https://docs.python.org/2/library/pydoc.html

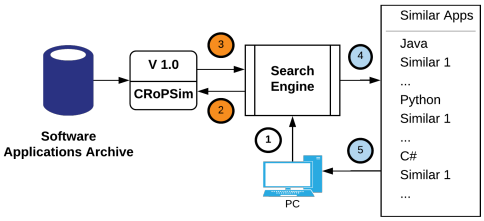


Fig. 5: Workflow for detecting Similar Software Applications

API documentations instead of a regular paragraph. As a result, most of the sentences are not related to each other. On the other hand, DReP also depends on the appearance of API calls which is also not fixed for all the software applications. So, to learn the distributed representations of a word inside a paragraph more accurately we used a high dimension vector representation for each of our words. We executed our model on the **Tensorflow** [34] deep learning platform. Finally, for calculating similarity between two projects, we calculated the *Cosine* similarity of vector representation of two DRePs according to the following equation:

$$CosineSimilarity = cos(\theta) = \frac{A.B}{\|A\| \, . \, \|B\|} \qquad (2)$$

where, A and B are word vectors of dimension D.

Using the cosine similarity, we generated a matrix where both the row and column of the matrix represent names of the software applications. Cells of the matrix represent the cosine similarity scores between the indexed projects. After performing the whole matrix, we applied the KNN algorithm to cluster similar software applications

E. CroLSim's Workflow

The working steps for CroLSim are listed in Figure 5. In step (1) the user queries with a software application they have along with the source code. Once the query is made, in step (2), the search engine sends the query to CroLSim. As shown in Figure 4 CroLSim first generates **DReP** for the queried software application. After that, it compares the newly generated *DReP* with the already generated *DReP*s of the software applications in the repository. After completing all

the phases of the CroLSim architecture, in step (3) the recommended similar software applications along with the *Cosine* similarity scores are returned to the search engine. The search engine categorizes all the resulting software applications in step (4) based on their *Programming Language* and ranks them according to their similarity scores in an ascending way. Finally, in step (5) this page is returned to the user who sent the query.

F. DataSet Collection

For collecting software applications from GitHub, we developed a crawler, named **GitCrawl** which crawls throughout the GitHub repositories and downloads Java, Python and C# software applications. With the help of the **GitCrawl** we traversed more than 300K repositories and downloaded more than 10k software applications for each of the programming languages. Many of these downloaded repositories did not include any software application or soure code except some descriptions or PDF files or other documents. We manually removed those repositories and finally selected 8,956 Java, 7,658 C#, 10,232 Python applications to proceed with the CroLSim evaluation process.

IV. Experimental Setup

We used an Intel(R) Core(TM) i7-2600 3.4 GHz CPU with 16GB RAM to run the experiments with CroLSim. As we have seen that search and retrieval engine related tools and models are usually evaluated by experts with manual relevance judgments [9], [10], [36] we validated the results of CroLSim with 9 participants who have all been engaged with software development for at least 5 years. To evaluate our work, we compared our model results with two of the baseline methods, CLAN [9] and Repopal [10] as these two models are very related to our work and state of the art solutions in detecting software similarity.

A. Baseline Methods

From Yang et al.'s Repopal [10] we considered the Readme files relevance calculation mechanism as the other two parameters, e.g: **Stargazer similarity** and **time relevance of giving a star by a single user** are only workable with GitHub which hinders universal acceptability of this work and thus, we did not considered this work in our comparisions. We also considered McMillan et al.'s CLAN [9] as this model is closely related to our approach. Our evaluation with Repopal(readme) and CLAN on our repository experienced an average 2-3% error from the evaluation results shown in the original publication which is acceptable for reproducing any previous model. We did not compare with collaborative tagging [12] because it requires manual tagging and at the same time has a performance challenge [37]. Thus, finally we ended up comparing with the two closely related state of the art approaches with ours.

TABLE II: Degree of relevance assigned by participants

Score	Relevance	Description
1	Highly Dissimilar	Two software applications are entirely different
2	Dissimilar	Two software applications are mostly different
3	Neural	It is hard to say two software applications are similar or not
4	Similar	Two software application are somewhat similar
5	Highly Similar	Two tools are completely similar to each other

B. Evaluation Methodologies

In this section, we we discuss how we evaluated CroLSim with the help of nine participants and widely used performance measurement metrics.

1) Confidence: Confidence level means the degrees of relevance one participant can provide at time of validating the results of already detected similar software applications according to their own manual evaluation of the resulted repositories. We asked our participants to evaluate our detected similar software applications based on their query by giving a confidence score ranges from 1 to 5 according to table II. We built appropriate user interfaces to make it easier for our participants and for collecting confidence scores for each of the software applications ranked in the top-5 for a query performed by each participant of our system. Once we collected the scores, we calculated the mean and median confidence levels for the recommended top-5 results for each query. We allowed each of our participants to query twice with two different software applications which generates $15*2 = 30$ recommendations for three programming languages in total. As a result, we got 270 recommendations and related confidence scores to evaluate our work. We could do more, but as each of our participants needed to validate 15 software applications for each of their queries, which was really time consuming (>6hours/query) and required considerable manual labour, we limited our validation process to 18 queries in total. But as those queries were randomly selected and performed on a collection of real software applications, the evaluation process would not differ much with a higher number of queries.

2) SuccessRate@T: We adapted this performance analysis parameter from Zhang et al.'s [10] work. According to their definition, SuccessRate@T defines the proportion of successful top-5 recommendations we found after performing a query in our system. Here the value of T can be any value of our defined confidence level. For example, if the top-5 recommended software applications for a given query are evaluated with confidence level 4, 3, 2, 1, 2 at time of SuccessRate@5, this query will not be considered as a successful one. In our work, we considered up to SuccessRate@4 as successful recommendation for a query.

3) User Study: To cross validate our work, we divided our participants randomly in two groups. Participants from each

group were asked to run CroLSim in their own way. For each of the participants we defined a list of tasks needed to be performed. Each of the participants were asked to select one software application and perform a query with that. To ensure the accuracy and adaptability of our cross validation procedures, we did not provide any restriction on the participants at the time of selecting software applications. Once a user queried with a software application of one language, related software applications for the other two programming languages are listed on the CroLSim's user interface. We also listed recommended similar software applications from the same programming languages. We limited the number of software applications on the interface to 5 for each of the programming languages. Once the queried results popped up, we asked our participants to assign a confidence level, C, to each of the resulting software applications according to the Table II.

Finally, when both of the groups were done with their queries and evaluation, we asked one group to validate the evaluation results of the other group. At this time, we asked the participants to check and comment regarding the already evaluated queries and their results and finalize their decision of accepting or rejecting the previous participants evaluation. We collected all the accepted queries and results of confidence level. For the rejected results, we performed those queries again with other participants and collected the final results once they were finally accepted by the validating participant.

4) Precision: For our evaluation we defined **Precision** as the portion of similar and highly similar software applications recommended by our top-5 position for a given software application query. For the set of queries performed by our participants we calculated mean and median precision for our proposed model. As we are going to evaluate whether our model can detect software similarity across different programming languages, we calculated mean and median precision for recommended similar software applications individually across different programming languages. For example, we calculated mean and median precision between Java and Python, Java and C# and C# and Python. At time of calculating precision for a recommendation system we also need to calculate Recall of the system. But for our case, we could not calculate it as we did not define and did not have any previous knowledge about our repository regarding how many software applications are similar in our experimental software repository. The main reason to work with random and unlabeled software applications is to evaluate the performance of CroLSim in a real life scenario and with bias free experimental analysis.

5) Research Questions: For performing the evaluation work, we designed our experiments to answer the following three research questions:

RQ1 What are the SuccessRate@T scores of CroLSim, CLAN and Repopal(Readme) for detecting software similarity?

RQ2 What are the Confidence Scores of CroLSim, CLAN and Repopal(Readme) for detecting similar software applications?

TABLE III: SUCCESSRATE COMPARISION FOR CLAN, REPOPAL(README) AND CROLSIM

Programming Language	SuccessRate@4			SuccessRate@5		
	CLAN	Repopal (Readme)	CroLSim	CLAN	Repopal (Readme)	CroLSim
Java	65%	25%	**72%**	32%	21%	**68%**
Python	62%	23%	**78%**	28%	25%	**64%**
C#	73%	21%	**81%**	44%	26%	**75%**
Java & Python	N/A	19%	**71%**	N/A	24	**62%**
Java & C#	N/A	22%	**82%**	N/A	29%	**70%**
Python & C#	N/A	18%	**76%**	N/A	28	**65%**

[9]N/A: Not Applicable

RQ3 What are the Precision Scores of CroLSim, CLAN and Repopal(Readme) for detecting similar software applications?

V. EVALUATION

In this section, we discuss the answers to the research questions defined in previous section for evaluating the performance of our proposed model.

A. Answer to RQ1: SuccessRate@T

Our experimental study on success rates for three tools, baseline methods: CLAN, Repopal(Readme) and our proposed CroLSim are shown in Table III. From the table we can see that CroLSim has a higher success rate than all other models for both single programming language and cross-language programming languages. For single programming language such as Java, Python and C#, CroLSim can recommend software applications for a given query with a SuccessRate@4 (contain at least 1 similar recommendation for the given query) for a rate of 72%, 78% and 81% respectively where the very related work CLAN has observed 65%, 62% and 73% respectively at a rate of SuccessRate@4. For this scenario, Repopal (Readme) application has been observed with very poor performance, 25% for Java, 23% for Python and 21% for C# programming language based software applications. For SuccessRate@5 (contain at least 1 highly or fully similar recommendation of software application for the given query) CroLSim outperforms all other models with its success rate; 68%, 64% and 75% for Java, Python and C# programming languages respectively. For this scenario, other models, CLAN and Repopal (Readme) observed a considerably lower success rates (lower than 45%) comparative to CroLSim.

For Cross language similar software application detection, at SuccessRate@4, our experiments showed that CroLSim can recommend software application tools with a success rate of 62% between Java and Python, 70% between Java and C# and 65% between Python and C# programming language. This result is much higher comparative to Repopal (Readme) which showed a success rate lower than 30%. Thus, to answer RQ1, we can confidently say that CroLSim outperforms the other two state of the art tools in detecting similar software applications of different programming languages in term of SuccessRate@T.

TABLE IV: CONFIDENCE SCORE COMPARISON FOR CLAN, REPOPAL(README) AND CROLSIM

Programming Language	Mean Confidence Score			Median Confidence Score		
	CLAN	Repopal (Readme)	CroLSim	CLAN	Repopal (Readme)	CroLSim
Java	2.24	1.94	**3.29**	2.0	1.74	**3.5**
Python	1.72	1.75	**3.62**	1.8	1.7	**3.7**
C#	3.32	1.82	**3.48**	3.2	1.63	**3.6**
Java & Python	N/A	1.78	**3.24**	N/A	1.6	**3.6**
Java & C#	N/A	1.89	**3.52**	N/A	1.6	**4.2**
Python & C#	N/A	1.74	**3.35**	N/A	1.4	**4.0**

[10]N/A: Not Applicable

TABLE V: PRECISION SCORE COMPARISON FOR CLAN, REPOPAL(README) AND CROLSIM

Programming Languages	Mean Precision			Median Precision		
	CLAN	Repopal (Readme)	CroLSim	CLAN	Repopal (Readme)	CroLSim
Java	0.284	0.147	**0.764**	0.4	0.2	**0.68**
Python	0.178	0.158	**0.798**	0.24	0.19	**0.7**
C#	0.304	0.151	**0.725**	0.46	0.2	**0.64**
Java & Python	N/A	0.168	**0.652**	N/A	0.22	**0.57**
Java & C#	N/A	0.162	**0.645**	N/A	0.22	**0.54**
Python & C#	N/A	0.158	**0.671**	N/A	0.2	**0.58**

[11]N/A: Not Applicable

B. Answer to RQ2: Confidence Score

The mean and median confidence scores we observed at the time of our experiments for all three tools, e.g., CLAN, Repopal (Readme) and CroLSim are stated in Table IV. From the table we can see that, for a single programming language, CroLSim has a mean and median confidence score of 3.29 and 3.5 respectively for Java, 3.62 and 3.7 respectively for Python and 3.48 and 3.6 respectively for C#. On other hand, CLAN has an observed mean and median confidence score of 2.24 and 2.0 respectively for Java, 1.72 and 1.8 respectively for Python and 3.32 and 3.2 respectively for C#. For Repopal (readme) we see that its recommendation based on readme files similarity gets very low mean and median confidence scores (less than 1.9 in most cases) which is not even acceptable as a similar software recommendation method itself.

The reason for CLAN's showing poor performance with Python is that Python supports a lot of third party libraries rather than depending only on the Python API itself. So, it false positively detects a lot of similar software applications which are not similar at all. So, the confidence scores deteriorates considerably. C# development is mostly dependent on the .Net framework provided APIs so CLAN performs good, almost near to CroLSim.

For detecting cross language similar software applications from a repository, we examined a good and acceptable performance for CroLSim. For detecting similar software applications between Java and Python, it is found mean and median confidence scores are 3.24 and 3.6 respectively. For similar software detection between Python and C#, mean and median confidence scores are observed 3.35 and 4.0 respectively and for the case of Java and C#, these values are 3.52 and 4.2. From this analysis we can say, CroLSim can detect similar software applications across different programming language with great accuracy.

C. Answer to RQ3: Precision Score

From Table V we can see that CroLSim has a higher mean and median precision values than all other existing methods. For single programming language software repositories like Java, Python and C# we observed mean average precision values 0.764, 0.798 and 0.725 respectively and median precision values 0.68, 0.7 and 0.64 respectively. For Java, CLAN achieves a mean average precision of 0.284 and median precision of 0.4 which is lower than CroLSim. For Python and C#, CLAN's mean and median precision achievements are 0.178, 0.304 and 0.24, 0.46 respectively. Compared to CroLSim, mean and median precision values achieved by CLAN for all the programming languages are lower. For another baseline method, Repopal(Readme) achieves very low mean and median precision values (most of the cases lower that 0.2) which actually confirms that CroLSim outperforms all the existing methodologies and models in terms of detecting similar software applications from software repository in terms of accuracy.

For Cross-Language software similarity detection, we examined that CroLSim can detect similar software applications with higher accuracy and precision value across different programming languages. For Java and Python, the mean average precision for CroLSim in 0.652 and median precision is 0.57. For Java and C# the mean average precision is 0.645 and median precision is 0.54 and for Python and C# we observed mean and median precision values 0.671 and 0.58 respectively. Compared to Repopal (Readme) CroLSim achieved much higher precision as Repopal (readme) achieved a mean and median precision rate of less than 0.2.

From the above analysis we come to a conclusion that, CroLSim outperforms the existing methodologies of similar software detection for repositories in terms of success rate, confidence level and precision score achieved for single language. For Cross-Platform software similarity detection, to the best of our knowledge, as it is the first approach to detect similar software applications across different programming languages by performing semantic relationships among API calls, a mean average precision of more than 0.65 is an acceptable one for all respects.

VI. THREATS TO VALIDITY

Participants. Any study involving users may have potential bias and we are not an exception. However, in this study we were careful in selecting the users and only selected those who have at least five years of experience with the programming languages. Furthermore, we conducted cross-validations with different groups of users until a consensus was reached.

Repositories. The sample size of our experiments may not be enough to generalized the whole ecosystem of a software repository. However, we used about 9K Java, 8K C#, and 10K Python software applications from GitHub which is much large in size compared to any of the subject systems. Since we were able to directly use GitHub systems without any change or

fabrication, we can confidently say that CroLSim would be able to work with other systems and repositories too.

Third party API documentation. Unavailability or poor quality of API documentation is a potential threat for CroLSim. However, we were able to limit this threat by using the documentation which is made available by the providers of the APIs. Of course, for third party ones, we relied on developers' documentations. We also made sure those are in an understandable format.

Parameters for Evaluation. One might argue about our selection of the evaluation metrics as well. In order to mitigate those threats we adapted the widely used metrics for such studies from previous work such as CLAN [9], Collaborative Tagging [12] and Repopal [10]. Such evaluations were also used in various other related studies too [8], [38], [39].

VII. Related Work

Finding similar software applications is not a new research topic. However, there is a marked lack in detecting similar applications from systems of different languages. In the following we position our work with the state of the art:

A. Detecting Similar Software Applications

The work of Kawaguchi et al. [8], McMillan et al. [9], Thung et al. [12] and Zhang et al. [10] are most closely related to ours. In MudaBlue [8], the authors only considered the identifiers of source code which requires manual investigation to pick the informative one. McMillan et al. in their CLAN [9] used JDK API calls similarity to detect similar Java based software applications from the repository and evaluated with more than 8K Java applications. Unfortunately, since API calls across systems of different programming languages are significantly different, their approach cannot be used for cross language similarity detection. In another work, Thung et al. [12] detected similar applications using manual collaborative tagging of the applications. While this holds promise for cross language similarity detection, applicability of this approach is limited since it is solely dependent on manual evaluation and user experience in tagging the applications. In a recent work, Zhang et al. [10] extracted three pieces of information, relevance of Readme files, Stargazer relevance and Time relevance for each of the software applications from GitHub and used those to detect similar software applications. While this approach also has promise in cross language similarity detection, its applicability is limited to GitHub repositories. Other repositories may not have these pieces of information. Even stargazer information and time relevance of putting the stars by a stargazer are not available for many of the GitHub repositories. This also possibly the primary reason why the authors were restricted to use only 1K software applications.

In our proposed work, we attempted to overcome the limitations of the existing state of the art approaches discussed above. Our work is based on documentation of all API calls and on Deep Convolutional Neural Network based Paragraph Vector Model which were not used in such contexts in the past. We also compared our approach with those of McMillan et al. and Zhang et al. and demonstrated with rigorous experiments that CroLSim outforms those approaches in detecting similar software applications for both single and cross programming languages.

B. Code Search and Clone Detection

Several studies exist to search code fragments from source code, e.g. Exemplar [39], SNIFF [40], Portfolio [38], Parseweb [42], Spotweb [41] and so on. These studies perform code searching with the help of Natural Language Processing and matching a certain amount of word queries. These are not directly related to our work as we are recommending fully functional software applications related to another fully functional software application but our method can be applied in these circumstances which is another future goal of our ongoing study.

At present a lot of clone detection tools in source code are available [43], [44], [45] which use various NLP and Abstract Syntax Tree (AST) [52] based approaches to detect similar behaving code fragments across different software applications. Cross-language code clone detection has also been studied using intermediate languages [46] and version histories [47] with limited success. As with these models, CroLSim also possibly has potential in cross language clone detection, which we plan to explore in the future.

C. Software Categorization

Another related research area is categorizing software applications in a software repository. A good number of studies have already been performed by Kawaguchi et al. (using their tool MUDAblue) [8], Wang et al. [48], Xu. et al. [49], Zhang et al. [50] and so on. The basics of categorizing software applications is part of detecting similar software applications. Once one can detect similar software applications, with the help of different clustering approaches, e.g. KNN, Random Forest, SVM and so on. It is possible to cluster software applications based on the similarity score. So we can say, CroLSim can be easily extended to categorize software applications which is discussed in our Architecture Section III-D. However, we did not evaluate the performance of clustering the software applications since our objective was to detect similar software applications of different languages.

VIII. Conclusion

In this paper, we introduced CroLSim, which is able to detect cross language similar software applications with an average precision of more than 65%. To the best of our knowledge, we are the first to explore a deep learning based approach to detect semantic relationships among API calls in order to identify similar applications written in different programming languages. Our approach provides a general solution for detecting software similarity and can detect similar software applications written in the same programming language as well as similar applications written in different programming languages for any type of software repository. We use the functional description of the APIs to identify

the semantic relationship among the APIs, which we use to detect similar software applications in a source code repository. Our experimental evaluations show that CroLSim outperforms available mechanisms for detecting similar software applications with a significant performance improvement. In the future, our plan is to extend this work to detect code clones in software applications, searching for similar code blocks from different software applications and automatically categorizing software applications with meaningful names in a cross-language environment. In addition, we plan to study the benefits of using AST-based features along with CroLSim for detecting similar software applications.

REFERENCES

[1] The Workflow Management, https://en.wikipedia.org/wiki/Workflow_management_system

[2] H. A. Reijers, I. Vanderfeesten, and W. M. P. van der Aalst, *The effectiveness of workflow management systems*, Intl. J. of Info. Management 36:1 (February 2016), 126-141

[3] E. Afgan, D. Baker et al.,*The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update*, Nucleic Acids Research (2016) 44(W1):W3-W10 doi:10.1093/nar/gkw343

[4] S. A. Goff et al.,*The iPlant Collaborative: Cyberinfrastructure for Plant Biology*, Frontiers in plant science 2 (2011): 34. PMC. Web. 31 Oct. 2017.

[5] D. Bolser, D. M. Staines, E. Pritchard and P. Kersey ,*Ensembl Plants: Integrating Tools for Visualizing, Mining, and Analyzing Plant Genomics Data*, Methods Mol Biol. 2016;1374:115-40.

[6] A. Michail and D. Notkin, *Assessing software libraries by browsing similar classes, functions and relationships*, In Proc. ICSE, 1999, pp. 463-472.

[7] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, *Detecting similar Java classes using tree algorithms*, In Proc. MSR, 2006, pp. 65-71.

[8] S. Kawaguchi, P. K. Garg, M. Matsushita and K. Inoue, *MUDABlue: an automatic categorization system for open source repositories*, In Proc. APSEC 2011, pp. 184-193.

[9] C. McMillan, M. Grechanik and D. Poshyvanyk, *Detecting Similar Software Applications*, In Proc. ICSE, 2012, pp. 364–374

[10] Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li and J. Sun,*Detecting similar repositories on GitHub*, In Proc. SANER, 2017, pp. 13-23.

[11] Q. Le and T. Mikolov, *Distributed representations of sentences and documents*, In Proc. ICML, 2014,pp. II-1188-II-1196

[12] F. Thung, D. Lo and L. Jiang, *Detecting similar applications with collaborative tagging*, In Proc. ICSME, 2012, pp. 600-603.

[13] I. Keivanloo, C. K. Roy and J. Rilling, *SeByte: A semantic clone detection tool for intermediate languages*, In Proc. ICPC, 2012, pp. 247-249.

[14] https://en.wikipedia.org/wiki/Document-term_matrix

[15] https://octoverse.GitHub.com/

[16] S. K. M. Wong and V. V. Raghavan, *Vector space model of information retrieval: a reevaluation.*, In Proc. SIGIR, 1984, pp. 167-185.

[17] S. Mizzaro,*How many relevances in information retrieval?*, Interacting with Computers, Volume 10, Issue 3, 1 June 1998, Pages 303–320

[18] Rapp R., *Syntagmatic and Paradigmatic Associations in Information Retrieval*, In: Schader M., Gaul W., Vichi M. (eds) Between Data Science and Applied Data Analysis, 2003, pp. 473-482.

[19] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, *An empirical investigation into a large-scale Java open source code repository*, In Proc. ESEM, 2010, pp. 11:1–11:10

[20] Z. S. Harris, *Distributional Structure*, In Proc. WORD, 10:2-3, pp. 146-162, 1954

[21] A. Singhal, *Modern Information Retrieval: A Brief Overview*, IEEE Data Eng. Bull., Vol: 24, pp. 35-43, 2001.

[22] J. Hopcroft and R. Kannan, *Foundations of Data Science*, chapter 4, pp: 115-146, 2013.

[23] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer and R. Harshman, *Indexing by latent semantic analysis*, Journal Of the American Society for Info. Sci., 4(6), 1998, p: 391 - 407.

[24] Y. LeCun, *Generalization and network design strategies*, Technical Report CRG-TR-89-4, U of T.

[25] M. Bates, *Models of natural language understanding*, Nat. Academy of Sci., vol 95, pp 9977-9982,1995.

[26] T. Mikolov, K. Chen, G. Corrado and J. Dean, *Efficient Estimation of Word Representations in Vector Space*, arXiv 1301–3781, 2013.

[27] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, *A neural probabilistic language model*, Journal of machine learning research, vol. 3, no. Feb, pp. 1137–1155, 2003.

[28] X. Glorot, A. Bordes, and Y. Bengio, *Domain adaptation for large-scale sentiment classification: A deep learning approach*, in Proc. ICML, 2011, pp. 513–520.

[29] R. Collobert and J. Weston, *A unified architecture for natural language processing: Deep neural networks with multitask learning*, in Proc. ICML, 2008, pp. 160–167.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, in Proc. NIPS, 2013, pp. 3111–3119.

[31] Javadoc by Oracle: \http://www.oracle.com/technetwork/articles/Java/index-jsp-135444.html

[32] DocFx: https://dotnet.GitHub.io/docfx/tutorial/docfx_getting_started.html

[33] R. Rehurek and P. Sojka, *Software framework for topic modelling with large corpora*, In LREC Workshop, 2010.

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard and M. Kudlur, *TensorFlow: A System for Large-Scale Machine Learning*, In Proc. OSDI, Vol. 16, pp. 265-283, 2016.

[35] N. S. Altman, *An introduction to kernel and nearest-neighbor nonparametric regression*, The American Statistician 46.3, pp. 175-185, 1992.

[36] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*, In Cambridge University Press, New York, NY, USA, 2008

[37] G. Macgregor, E. McCulloch, *Collaborative tagging as a knowledge organisation and resource discovery tool*, In Library Review, Vol. 55 Issue: 5, pp.291-300, https://doi.org/10.1108/00242530610667558

[38] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, *Portfolio: finding relevant functions and their usage* In Proc. ICSE, 2011, pp. 111-120.

[39] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu and Q. Xie, *Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications*, In TSE, vol. 38, no. 5, pp. 1069-1087, 2012.

[40] S. Chatterjee, S. Juvekar, and K. Sen, *Sniff: A search engine for Java using free-form queries*, in Proc. FASE, pp. 385–400, 2009.

[41] S. Thummalapenta and T. Xie, *Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web*, in Proc. ASE, pp. 327–336, 2008.

[42] S. Thummalapenta and T. Xie, *Parseweb: A programmer assistant for reusing open source code on the web*, in Proc. ASE, pp. 204–213, 2007.

[43] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, *SourcererCC: scaling code clone detection to big-code*, In Proc. ICSE, 2016, pp. 1157-1168.

[44] C. K. Roy and J. R. Cordy, *NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization*, In Proc. ICPC, 2008, pp. 172-181.

[45] C. K. Roy, J. R. Cordy, and R. Koschke, *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach*,In Sci. Comput. Program. 74, 7, pp. 470-495.

[46] N. A. Kraft, B. W. Bonds and R. K. Smith, *Cross-language clone detection*, In SEKE 2008, pp. 54–59.

[47] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu and J. Zhao, *Mining revision histories to detect cross-language clones without intermediates*, In Proc. ASE, 2016, pp. 696-701.

[48] T. Wang, H. Wang, G. Yin, C. Ling, X. Li, and P. Zou, *Mining software profile across multiple repositories for hierarchical categorization*, in Proc. ICSM, pp. 240–249, 2013.

[49] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, *Predicting semantically linkable knowledge in developer online forums via convolutional neural network*, in Proc. ASE, 2016, pp. 51–62.

[50] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, *Combining software metrics and text features for vulnerable file prediction*, in Proc. ICECCS, 2015, pp. 40–49.

[51] C. Wiwie, R. Rottger and J. Baumbach, *Comparing the performance of biomedical clustering methods*, In Nature Methods, 2015.

[52] https://en.wikipedia.org/wiki/Abstract_syntax_tree