A Universal Cross Language Software Similarity Detector for Open Source Software Categorization

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*—While there are novel approaches for detecting and categorizing similar software applications, previous research focused on detecting similarity in applications written in the same programming language and not on detecting similarity in applications written in different programming languages. Cross-language software similarity detection is inherently more challenging due to variations in language, application structures, support libraries used, and naming conventions. In this paper we propose a novel model, CroLSim, to detect similar software applications across different programming languages. We define a semantic relationship among cross-language libraries and API methods (both local and third party) using functional descriptions and a word-vector learning model. Our experiments show that CroLSim can successfully detect cross-language similar software applications, which outperforms all existing approaches (mean average precision rate of 0.65, confidence rate of 3.6, and 75% highly rated successful queries). Furthermore, we applied CroLSim to a source code repository to see whether our model can recommend cross-language source code fragments if queried directly with source code. From our experiments we found that CroLSim can recommend cross-language functional similar source code when source code is directly used as a query (average precision=0.28, recall=0.85, and F-Measure=0.40).**

***Index Terms*—API Calls, Doc2Vec, Cross-Language Software Similarity Detection, Singular Value Decomposition**

I. Introduction

Software researchers and developers are often developing similar applications, whether it is to make them faster, more reliable, more feature-rich, or to improve their design, algorithms, maintainability, sustainability, or user-friendliness. As a result of this activity, a great many software applications have been developed for solving the same problems. Consequently, similar applications are common in both open and closed source software repositories. With the growth of the open source community this has taken on a new momentum. It is often desirable to classify and group software applications that are similar so that users can more easily select the specific application with the functionality and qualities they need to best address their problem. Now a days, software researchers and users from different disciplines work on common platforms that support combining a variety of software applications and tools into scientific workflows. Some examples of this type of platform include *Galaxy* [1], *IPlantCollaborative* [2], and *ensemble plants* [3]. When a good number of similar software applications or tools are present in a common repository, it benefits users as they can choose the one best related to their need and a comparative result may also help ensure they select the most appropriate application or tool. Furthermore, identifying similarity among applications might help foster reusability, support source code understanding, speed up the development process [4], and perhaps help identify software plagiarism [5]. Unfortunately, detecting software similarity is a challenging task because of the diversity of software applications. This is even more challenging when the applications are developed in different programming languages since there are further variations in their structure, support libraries, function names, and so on.

A number of approaches have been proposed for categorizing software applications. Mudablue [6], an early approach, categorize software applications according to important identifiers used during development. This work requires manually selecting identifiers and its accuracy is very low. Collins et al. [7] developed a tool called 'CLAN' that mainly considered third party Application Programming Interface (API) classes and packages used in the software to determine related applications from an application corpus. Thung et al. [11] extracted collaborative tags for each software application and used these tags as a semantic feature for software application similarity detection. A combination of these three models works better than each individual one for predicting software similarity. Unfortunately, the approaches are limited to software similarity detection for a single programming language as these models were not designed to detect similar software applications across different programming languages. Although Thung et al. [11] claim that their model could be extended to cross-language similar software detection. However, their model is fully dependent on third party collaborative tagging, which is dependent on the user's experience and therefore limited to the user's knowledge of specific programming languages. Zhang et al. [8] proposed 'Repopal' and claim that their model can be extended to detecting cross-language similar software applications. They extracted three language independent features from Github repositories to determine similar software applications. They considered readme files with similar content, applications that are bookmarked (starred) by users with similar interests, and applications starred within a short period of time by the same user. As these features are solely GitHub features and not even available for all GitHub repositories, in practice their model is not able to detect cross-language similar software applications for any software repository.

Detecting similar software applications across different programming languages is challenging for several reasons. First

of all, different languages have different syntax and semantic features and structure. For example, the way Python programs are written is quite different compared to Java programs. Second, classes and methods used by different programming languages may look completely different even if the two methods from the two different languages achieve the same objectives. Third, API names and their naming conventions differ from one programming language to another. As a result, it is hard to know when two APIs from two different programming languages are similar to each other by just observing their names. Fourth, the binary code generated for different languages and compilers are also differ syntactically and semantically. Existing tools are thus not capable or have limited capability in detecting cross-language similar software applications. With the expansion of the open source community and the number of available software applications, a tool with the ability to detect and categorize similar software applications across different programming languages would be valuable.

In this paper, we propose a novel model, CroLSim, which is able to detect similar software applications across different programming languages. For establishing a connection between different programming languages, we used the descriptions of APIs and their methods[1,2,3] which briefly state the purpose of the specific package and method. We observed that although the name of the APIs and their methods are different across systems of different programming languages, APIs and methods performing the same operations have semantically similar descriptions in their documentation. First, we collected descriptions of APIs and methods for different languages and built a corpus with those descriptions. Second, we extracted all APIs and methods (including third party APIs) from the source code for each of the applications. For each application, we queried the corpus with the extracted APIs and methods names, retrieved the related descriptions, and combined the descriptions for the application into a single file, which we refer to as the descriptive representational document or *DReP* for short. From our observations we found that it is often hard to get the same structured descriptions for different APIs and methods across different languages. To address this challenge, we adapted Le and Mikolov's distributed memory model of Paragraph Vectors (PV_DM) [10], a deep learning based advanced Natural Language Processing (NLP) model to predict semantic information from the method descriptions. We applied NLP over DReP for each of the software applications to find semantic similarity between them. Finally, we clustered the similar software applications by using the semantic weights generated as an outcome of the deep learning model.

After showing that CroLSim successfully detected cross-language similar software applications we applied the model to see whether it can efficiently recommend cross-language source code if the query is performed using source code directly without having any human query or Natural Language Processing mechanisms. For this, we constructed a source code repository of more than 100K lines of code developed in Java, Python, and C# programming languages and already verified as functional similar source. From our experiments we found that without any change to the proposed CroLSim model, it can recommend cross-language functional similar source code with an average precision rate of 0.28 and an average recall rate of 0.23 for the top 10 recommendations and an average recall rate of 0.85 for the top 30 recommendations. We also observed an average F-Measure score of 0.40 for 300 successful queries. Our contributions are as follows:

- We developed a tool, CroLSim, which is able to detect similar software applications across different high-level programming languages from a large software repository. We used method descriptions (provided by programming language developers and maintainers) for each of the programming languages to perform this activity. The model is able to detect similar software applications from different software repositories. Currently, the model supports four programming languages: C, Java, Python and C#. We selected the C programming language as a representation of a legacy programming language and showed that CroLSim is successfully able to detect software similarity even between a legacy programming language and dynamic programming languages.
- We generated a *corpus* with the help of API and method descriptions for different programming languages. For each language, we maintained an individual XML file. Each XML file contains descriptions of APIs and methods used by the programming language.
- We introduced a means of establishing a semantic relationship between method descriptions of different programming languages with the help of the deep learning based PV-DM model. We calculated semantic relationships among the words used in the description of the software and predicted semantic words for the software from that description. Finally, we calculated the similarity of the predicted words of the different documents which leads us to create clusters of similar software applications for a software repository.
- To evaluate our work, we performed an extensive experiment which supports our claim that our model can detect similar software across different programming languages and outperforms the currently available models in terms of accuracy and precision.
- We extended the model to recommend functionally similar source code across different programming languages. Unlike other code search models, we did not use a natural language query, but used source code directly as the query to recommended functionally similar source code in another programming language.
- We configured a source code repository of more than 100K functionally similar source code developed in Java,

[1]Java Docs: https://docs.oracle.com/javase/8/docs/api/

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

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

```java
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 for adding two numbers

```csharp
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 );
  }
}
```

(b) C# code for adding two numbers

```python
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 for adding two numbers

```c
#include<stdio.h>

int main()
{
  int a, b, c;
  printf("Enter two numbers to add\n");
  scanf("%d%d", &a, &b);
  c = a + b;
  printf("Sum of the numbers = %d\n", c);
  return 0;
}
```

(d) C code for adding two numbers

Fig. 1: Code for adding two integer numbers and displaying results on console in four different programming languages

C#, and Python. To the best of our knowledge, this is the largest cross-language source code repository with functionally similar code developed in different programming languages.

This paper is an extended version of our SCAM 2018 paper [9]. We extended the paper in several ways, including considering additional systems written in different programming languages, examining the effectiveness of the model for source code search, expanding on the challenges involved in cross-language similarity detection, adding more detail when describing the model, extending the experimental section for detecting cross-language similar functional code blocks, and generally expanding related work, threats to validity, etc.

The rest of the paper is organized as follows. In Section II, we provide an example that motivated us to perform this research along with current challenges. In Section III we discuss the technologies and methods used in developing our model. In Section IV we briefly discuss evaluation and validation procedures of our proposed method. In Section VI we discuss our experiment with CroLSim for detecting functionally similar cross-language code blocks. In Section VII we discuss the probable threats our model may face and how we tried to reduce their effect. In Section VIII we discussed related work and, finally, in Section IX we conclude the paper.

II. Motivation and Challenges

A. Motivation and Problem Statement

At present, many open source code repositories are available where a user can upload their source code under development, share their code with collaborators, allow other developers to edit and update the code from time to time, and publish a stable version of a software application. One of the most popular open source code repositories is Github. According to statistics

```java
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);
    }
}
```

Fig. 2: Java source code for adding two numbers using command line arguments. Similar in functionality to Fig. 1a. These two examples can be considered similar software applications

published on October 2017, Github consists of 24M users, 67M repositories, almost 1M lines of Python code, 986K lines of Java code, and 326K lines of C# source code [14]. Github also stores software applications written in other programming languages. To manage these huge repositories and serve users better, it is important to categorize the different repositories based on their similarity in terms of source code, problems targeted, similar methods called, and so on. When a user searches source code repositories, categorizing repositories can also help to return better results.

Detecting similar software applications based on source code similarity for the same programming language is somewhat simpler than for different programming languages, and there are many publications available describing different approaches [68], [72], [76]. Usually, for source code similarity detection, each word (e.g., API names, library method names, syntactic information) in the source code is considered a *semantic anchor*, an element in a document which helps to define the semantic characteristics of that document [7]. In the case

TABLE I: Source Code Relevance Similarity Score for the source code shown in Figure 1

Language	JAVA	C#	Python	C
JAVA	1	0.062	0.021	0.014
C#	0.0621	1	0.075	0.02
Python	0.021	0.075	1	0.019
C	0.014	0.02	0.019	1

of the same programming language, since almost the same semantic anchors are used to develop software applications, it is comparatively easier to detect similarity among applications. Let us consider the two Java applications shown in Figure 1a and Figure 2. With the help of either of these applications, we can add two integer numbers and print the result on the console. To determine their semantic similarity, we use a well known Information Retrieval (IR) model, the Vector Space Model[4] [15], with which we analyse the source code of the two Java applications. We found that the two applications match 85% when considering source code semantic anchors. On the other hand, for different programming languages, API names, library method names, identifiers, variable declaration types, code structure, syntactic information, and so on are mostly different from each other. In Figure 1 we show the source code of a software application written in four different programming languages, namely Java (1a), C# (1b), Python (1c) and C (1d). All of these applications do the same thing; add two numeric values and display the results on the console. From the source code we can see that none of the source code terms, which are usually used as *semantic anchors* in source code, are similar in name or in code structure (i.e., code syntax). To confirm this, we input the source code of each of these four applications into the Vector Space Model. The resulting similarity scores are listed in Table I. From the similarity scores, we can see that considering semantic anchors using the Vector Space Model that for each of the different programming language examples it is not possible to consider them similar, even though they are almost identical in functionality. As most of the existing models are based on source code semantics or on used API patterns in source code, they also fail to detect similarity between applications across different programming languages, which we will show in our evaluation section.

B. Current Challenges

Similarity detection of word-documents is well-defined by Mizzaro in his conceptual framework for relevance [16]. According to this work, if two documents are sharing common concepts, then these two documents can be considered as relevant documents. This definition can be extended to cluster a group of documents based on their concepts' relevance scores. Similarly, for detecting similar software, if two applications share some similar features (e.g., text editing, editing images, executing calculations) then those two applications can be considered as similar applications. As we have discussed earlier, because of similarity in used libraries and API names, program structures, it is possible to define a relationship between two applications of the same language using available *Information Retrieval*(IR) methodologies. IR methodologies mainly focus on finding/establishing semantic relations among terms used in a document or in a source code of a software application. It works on two types of associations: *Paradigmatic Associations* and *Syntagmatic Associations* [17]. Syntagmatic associations take into account two documents as similar if the words in both of the documents appear similar where Paradigmatic associations calculates documents similarity based on highly semantically similar words present in those documents. Kawaguchi et al. leveraged the syntagmatic associations from software application source code in their work Mudablue [6] for calculating similarity score where McMillan et al. adapted Paradigmatic associations in source code by defining API calls as semantic anchors in their work CLAN [7]. However, their work may not be extended in detecting cross-language similar software applications since there remains a very low probability of any relationships between source code of two different programming languages (although some similarities in data types, e.g. Int, float, double, and identifiers are observed, the amount is too little to calculate application similarity). It is thus challenging to find a way to establish a common relationship between source code terminologies of different programming languages. In this paper we attempt to address this issue.

We observed that there are a lot of common API methods and code portions used in the source code of a software application which does not define the characteristics of the application and often produces false positive results in detecting software application similarity. Let us consider the method PRINT() in Java which is a method of the Java Class PRINTSTREAM or the class instance SYSTEM.OUT. Almost all the Java applications contain this method to print something in console. Thus, code blocks related to this method will detect all software applications as similar to each other. In 2010, Grechaink et al. [18] ran a study on Java applications of *SourceForge* open source software repository and found that almost 60% of Java applications had used STRING objects and 80% had collection objects. Not only in Java[5] but also software applications developed in other programming languages such as Python[6], C#[7], and so on contain some common APIs libraries and methods. Thus, for getting better performance in cross-language similar software application detection it is required to exclude effects of these common methods and libraries from source code of all the programming languages which is our another challenge to face during this work.

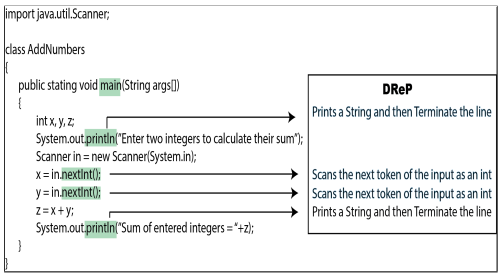
Moreover, the open source community is increasing everyday. At the same time, the size of each programming languages own repository and number of software applications in those repositories are rising too. As a result, the amount of source code in these open source software repositories are growing

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

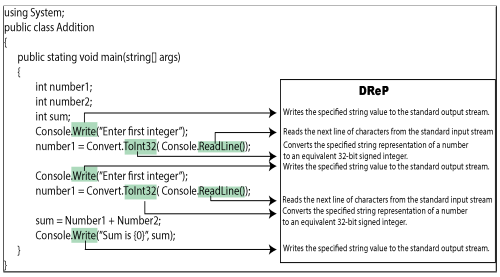
[5]https://www.programcreek.com/2011/08/the-most-widely-used-java-apis/

[6]https://docs.python.org/3.4/library/functions.html

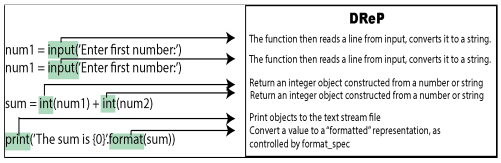
[7]https://msdn.microsoft.com/en-us/library/ms973806.aspx



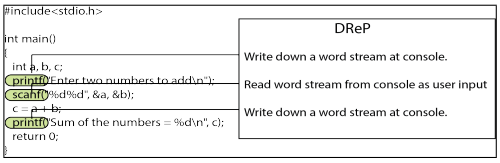
(a) Methods Extraction and DReP creation for Java Program



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



(c) Methods Extraction and DReP creation for Python Program



(d) Methods Extraction and DReP creation for C Program

Fig. 3: Procedures for creating DReP from source code analysis

everyday. In a report on October 2017, Github has claimed that right now they have 67M software repositories of different programming languages and 24M users and contributors in their system[8]. To detect cross-language software applications similarity of this huge number of open source software repositories we need to analyze source code of all the applications which is a Big Data handling and analysis problem. We thus need to find a light-weight model for analyzing source code of software applications within a reasonable time, which is highly challenging.

III. CroLSim - Model and Approach

In this section, we discuss our four phase approach for detecting cross-language similar software applications, the key methods we used in CroLSim (i.e., Singular Value Decomposition or SVD, Deep Learning & Natural Language Processing, and Paragraph Vector-Distributed Memory approach or PV-DM), and the CroLSim architecture.

A. Approach

CroLSim is divided into four phases. In Phase 1, we establish a common connection among the API and library methods used by each programming language. For this, we use the API and library descriptions available and maintained by developers and maintainers for the different programming languages. For example, Java and its packages, methods, and their user descriptions are maintained by Oracle[9], C# is developed and maintained by Microsoft[10], Python is maintained by Python Software Foundation[11], and so on. We assume that, if the combined descriptions of all of the methods used in a software application (i.e., the application's 'DreP' file) is similar to the combined descriptions of the methods used in another, then we can say these two applications are similar. We consider these DRePs as a Continuous Bag of Words (CBOW) [19] without considering any predefined semantics among the words. Let us consider the source code examples shown in Figure 3. Figure 3a, 3c, 3b and 3d provide examples of DReP files using our process. At first we select and extract methods used in the source code. After which, we query our corpus of method descriptions with each extracted package and method name, writing the results of the query to the application's DReP.

In Phase 2, we use Singular Value Decomposition (SVD) to filter out commonly used methods. We found that some APIs and their methods are common across applications and so provide little information for distinguishing an application, its use, or its category. Consider the four programs in Figure 3. We see that one of the methods used by the Java program is PRINTLN() from the JDK package SYSTEM.OUT, which prints something to the console with a newline. If we look at the C# program we see that the WRITE() method from the CONSOLE package does the same thing. The Python and C programs use PRINT() and PRINTF() respectively to accomplish something similar. If we give the same importance to these methods as to other more specialized and less frequently used methods it will lead us to incorrectly identify software applications as being similar merely if they use print functionality. Collins et al. [7] in their work also found this issue with methods used in Java source code. To deal with this, they used Singular Value Decomposition(SVD) along with Latent Semantic Indexing(LSI) to determine the less used but important methods in a Java program.

In Phase 3 we use an adapted Doc2Vec model [10] and cosine similarity [21] to determine the semantic similarities between cross-language software applications. Different developers and maintainers usually use different sentence structures and forms for describing different methods and library uses, and so working with a direct word to word match of the DRePs

[8]https://octoverse.github.com/
[9]https://www.oracle.com/java/index.html
[10]https://docs.microsoft.com/en-us/dotnet/csharp/
[11]https://www.python.org/psf/

for calculating software application similarity would not be useful. To predict words for representing the semantics of a DReP, we adapted the Doc2Vec model [10]. One of the main features of the Doc2Vec model is that it can predict semantic words from a paragraph or document on the fly without pre-training. We considered these predicted semantic words to be Semantic Anchors for each DReP and later we calculate the Cosine similarity [21] among the Semantic Anchors. In this way we calculated semantic similarities between cross-language software applications.

In Phase 4 we apply the K-Nearest Neighbors (KNN) algorithm to cluster software applications where similar software applications will appear in the same cluster.

B. Singular Value Decomposition

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

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

where A and B are orthogonal and D is diagonal where the values are positive real numbers. It is helpful to find a low rank matrix from a high dimensional matrix which can be used as an approximation of the regular one. SVD helps to select the peak variable in a dataset which in the other direction, helps to understand the variance existing in that dataset. This nature of SVD can easily be extended in selecting less frequently used API calls from the collection of all API calls used in a software system. SVD subordinate the frequently used API calls and highlights the less frequent ones from a collection.

To incorporate SVD in our work we developed a $m \times n$ dimension Term-Document Matrix (TDM) M with the help of extracted API calls from our collected software repository. In our matrix, row m is the name of the API call and column n is the software application. Each cell then represents the number of the specific API calls that 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 using the source code of the software applications from the software repository we collected from GitHub. Now, applying SVD on each of the M_l we derived the low rank matrix which consists of the least-used but effectively-used API calls in each of the software applications of the software repository.

To perform the entire process we adapted an Information Retrieval (IR) technique called Latent Semantic Indexing (LSI) [23] which embeds both TDM and SVD. According to the equation 1, SVD decomposes each of the matrices into three matrices of dimension d whose value is needed to be defined by the user. Usually the value of the dimension d is selected as 300 [23]. After performing LSI, based on equation 1, one of the matrices contains vectors for each of the software applications and using cosine similarity we can find similar software applications for a single language [7]. But it is quite impossible to detect software application similarity among different programming languages as the name of API calls are often different between programming languages.

C. Deep Learning & Natural Language Processing

Deep learning, also known as hierarchical learning, is a part of machine learning within Computer Science. It is a collection of learning algorithms which use neural networks and a learning structure inspired from human brain configuration and functionality as its architecture [24]. It uses multiple processing layers with multiple neurons, most often fully connected, to train the model with multiple levels of abstractions [25]. Deep learning can be performed in a supervised way and in an unsupervised way. In unsupervised deep learning, different features are learned during the training period in an unsupervised manner (train with unlabelled data). In deep learning, usually Rectified Linear Units (ReLU) are used as an activation function instead of Sigmoid or other activation functions in order to have a greater number of values between 0 and 1 for different points in the training range to solve Gradient Decent Vanishing or, in other words, to solve the lower Gradient value problem [26].

Recently, applications of Deep Learning have flourished in Natural Language processing (NLP) research. The main theme of NLP [28] is to automatically analyze and represent human language with the help of theory motivated computational algorithms and models [29]. The analysis phase in NLP is performed by representing a word in a vector of d dimensions (d is defined by an analyzer) and analyze the representational relation with surrounding word vectors which is a computationally expensive procedure [30]. In recent years NLP research has been successful by adapting deep learning which is able to analyse words from a sentence to a full paragraph for identifying the semantics of a word in a more accurate way [31].

Among the different kinds of deep neural networks applied in NLP, one form is the Convolutional Deep Neural Network (CNN) which uses a grid-like topology for processing data [27]. This network model uses a linear mathematical operation layer, called 'Convolution', at the very beginning of data processing operations. It replaces general matrix multiplication for reducing the dimension of the data which helps to approximate a better training performance after completion of the training procedure [32]. It uses a lower dimensional filter to sub-sample the provided training data, analyse it with the help of ReLU and followed by Max Pooling (take the highest/average value from the filter space) and stride (value of moving the filter along the dimensions of the sample data) to traverse the entire sample data. By performing these steps iteratively the whole CNN procedure generates an approximation of the original sample data in a lower dimensional space.

In NLP, for word to vector representations, many of the recent models use a neural network [33], especially CNNs on word embeddings [31] to reduce vector dimensions as well as to learn every word vector deeply with respect to surrounding words in a sentence or context. Word embeddings, also known

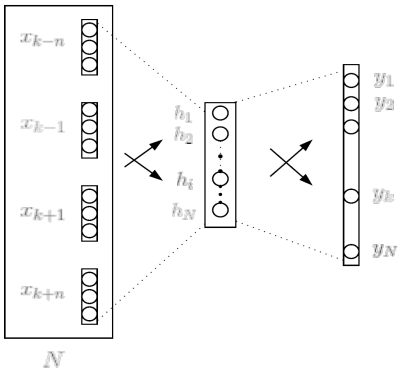


Fig. 4: Neural Network Architecture for Word2Vec

as distributional vectors, represents the characteristics of the surrounding words of a specific work which helps to learn the semantic meaning of a word [34], [35]. This procedure preserves similarity between words which tends to detect the context similarity of different word documents. Recently, word embeddings has become the heart of a lot of successful NLP tasks [36], [37]. One of the recent CNN based word vector representation models is Word2Vec [38] where the convolution layer operation (the first layer in the deep learning model) is applied on word embeddings to sub-sample the frequent words as a goal of training a large unlabelled corpus in a faster way.

In their Word2Vec research, Mikolov et al. revisited the word embeddings technique and proposed two word-vector learning techniques: Continuous Bag of Words (CBOW) and the Skip-Gram Model [31]. CBOW tries to learn the semantics of a word by learning the conditional probability of that word based on the context of the words surrounding it across a predefined window size. The Skip-Gram model tries to predict surrounding words by taking into account the current word. Both models use a fully connected neural network with one hidden layer which works as a projection layer. The input layer of the CBOW model considers word vectors for the two previous words and two post words (total 5 words including the target word itself) to predict the current word of the context when the window size s is 5. The Skip-gram model does the opposite. Each word vector for the selected words is learned from the column of the word embeddings generated from the given context. Figure 4 depicts the architecture of the CBOW learning model.

As shown in Figure 4, N is the number of words in the context and V is the number of neurons in the hidden layer h. We are trying to predict x_k in the output layer, denoted as y_k where $k \epsilon N$, with the help of n previous and post words from the context in such a way that $2n \leq s$. The output layer is calculated using a hierarchical softmax function of N words from the context vocabulary. The fully connected layer trains each of the words in the context with two weight matrices: the weight between the input layer and the hidden layer is $W_{N \times V}$ and the weight between the hidden layer and the output layer is $W_{V \times N}$ which generates two trained vector representations v_i and v_o according to the following equations:

$$W_{N \times V} = \{\{w_{si}\}_N\} \; and \; W_{V \times N} = w_{ik} v_i = W_{k,.} \quad and \quad v_o = W_{.,k} \tag{2}$$

Using the above learning equations, the softmax function for each of the words $word_{output}$ with respect to the given context word $word_{input}$ will be:

$$P\left(\frac{word_{output}}{word_{input}}\right) = \frac{\exp(v_o^T v_i)}{\sum_{i=1}^{N} \exp(v_o^T v_i)} \tag{3}$$

D. Paragraph Vector - Distributed Memory (PV-DM) Model

The design topology of Word2Vec was to have a learning model which can learn word vector representation in a faster way. With the help of Word2Vec it is possible to learn word vectors for a full phrase or a full sentence. In other words, Word2Vec can learn word vectors for a fixed length context. To extend this work for learning word vectors from a variable length context, Le and Mikolov [10] proposed a new model, called 'Paragraph Vector' and also frequently called Doc2Vec, which is able to learn continuous distributed vectors in an unsupervised way from a paragraph or full document. In this model, they tried to predict vectors for a paragraph by predicting multiple word vectors from the paragraph itself.

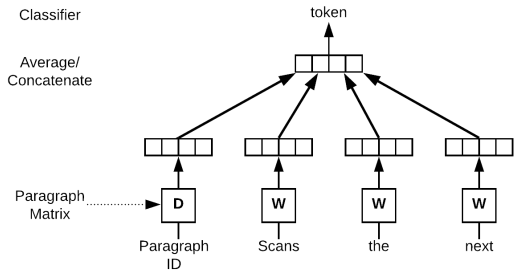


Fig. 5: Paragraph vector learning framework for predicting the next word ('token') from the paragraph ID and three input words ('Scans', 'the', 'next') (adapted from [10])

Figure 5 illustrates using the paragraph vector learning framework for our Java program DReP example given in Figure 3a. Three words from the Java DReP file ('Scans', 'the', and 'next') are trained for predicting the next word 'token'. All the words of a paragraph are embedded in a word matrix W. Each of the paragraphs in a document is identified with a unique paragraph id. Paragraph ids are embedded in a matrix D and provide further context. At the training phase, the paragraph id vector, represented by a column in matrix D and vectors for training words, represented by a column in matrix W are highlighted and concatenated for predicting the word 'token' for the DReP context. Paragraph vector is used to memorize the missing elements in the current context. Paragraph vectors are shared among all the paragraphs of a

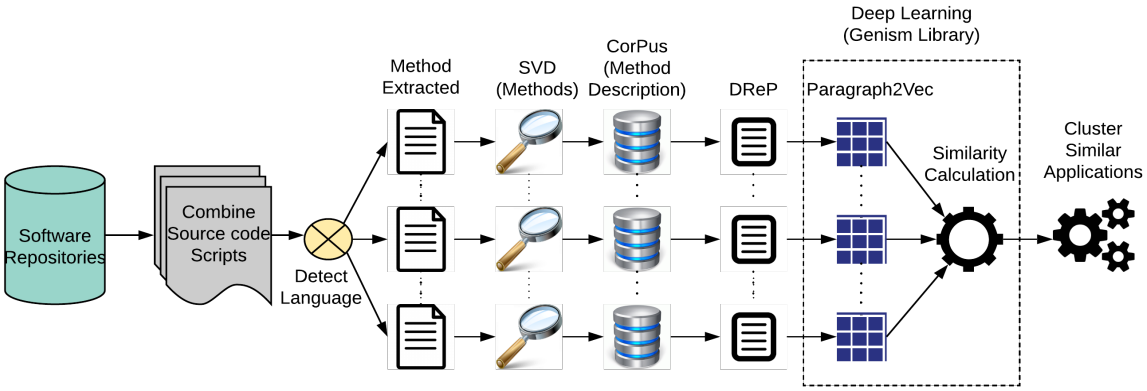


Fig. 6: Schematic diagram for CroLSim

document where word vectors of each paragraph are shared with other word vectors of that paragraph. The entire training process is accomplished with the help of stochastic gradient descent which is usually obtained by back propagation in a neural network. This model is also called the Paragraph Vector-Distributed Memory (PV-DM) model. A paragraph vector uses the same concept and structure as word2vec with the extension of a paragraph matrix along with a concatenation/average stage between word vectors and paragraph vectors to predict some word vectors or words as concepts for each of the paragraphs.

E. CroLSim's Architecture

A schematic diagram of the CroLSim architecture is shown in Figure 6. The main elements of our architecture are the software repositories that contain the applications to be considered, the scripts for combining the source code for each application, the application of the Singular Value Decomposition (SVD) model for identifying infrequently used API calls, a corpus of API method call descriptions, the constructed DReP for each application, the application of Doc2Vec and cosine similarity to determine semantically similar DRePs, and finally the use of KNN for clustering similar applications. All the software applications we collected for validating our model from Github are placed in Software Repositories. For each language we maintained a separate repository. The combined source code for each software application is also placed in Software Repositories.

At the beginning of the application of the CroLSim model, we selected one of the repositories and detected the programming language for that repository. We did this because the process of API call extraction from source code varies from language to language. At the same time, we also gathered the Readme files for each of the software applications in the selected software repository. After extracting API calls from all the software applications in the repository, we run SVD on them to identify infrequently used API calls from all the software applications in the repository. This procedure is already proven to be workable in previous work [7] in the belief that lower used API calls highly preserve the characteristics of the software application. We considered the top 400 API calls resulting from this approach. After obtaining filtered API calls, we store them in the software repository for each application. After performing SVD, once we obtained every filtered API call, for each API call we queried the corpus to get API documentation for the queried API call.

Before performing a query with API calls, we generated API documentation for each of the software applications. We generated an XML file by extracting all the third party API package and method details from each API documentation respectively. E.g. For java applications, we first considered JDK 8.0 and extracted all of its packages, classes and methods from its API documentation and recorded them to an XML file. This process took almost 3.5 hrs. It is also observed that developers of Java applications use some other external third party APIs, e.g. OpenCV, Apache Log4j, Jsoup, etc for developing their tool. To cover API documentation for all of these external third party APIs, once we selected one of the software applications from the software repository, we tried to scan for the third party APIs the respective software application contains. After identifying them, we checked whether we had already extracted API documentation for those APIs or not. If not, we again extracted packages, classes and methods for that specific API. We maintained a separate file to track which of the third party APIs we documented so far. Once we extracted API documentation from a third party API, we added the name of that API to our track file so that we can have a record and did not need to extract documentation for the same API for the second time. For getting API documentation, we searched for the documentation inside the jar file first (usually provided with the source code of the application). If we did not obtain documentation from inside the jar file, we downloaded the library and used javadoc API[12] [39] provided by Oracle with jdk 8.0. In this way, we developed a corpus for our

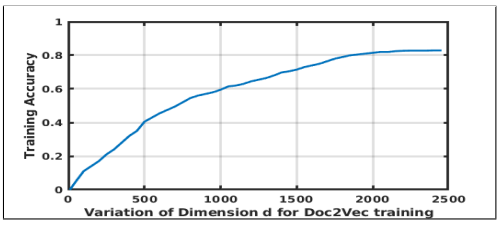
[12]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

work which is an XML file containing Packages, Classes and method names and descriptions for available third party APIs. We maintained a single XML file for each of the individual APIs. To develop a corpus only for the Java programming language, it took more that 15 hours. For other programming languages, we followed the same procedures for obtaining third party API documentation. E.g. for Python, we extracted API documentation for python2.7 and python3.6, for C# we considered API documentation for dotnet version 4.0. For additional undocumented API, we first collected that API and then for Python, we used pydoc[13] and for C#, we used DocFX [40]. It took almost the same amount of time for collecting API documentation for each of Python and C#.

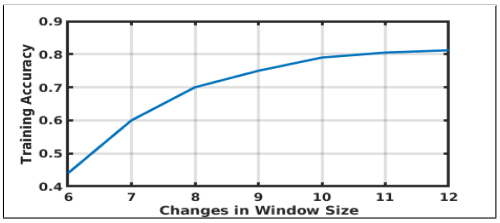
For our model, another potential source of information could be the comments and descriptions provided by the developer or any descriptions of the developed APIs provided by the developers of the software application. Like Readme files in GitHub, source code comments and descriptions of used APIs could be an important source of information to understand the functionality of the software application. But, from a manual analysis of the source code of the software repositories collected from Github or other open source software we found that most of the cases source code comments or description on the used APIs are not available. From our analysis we also observed that comments and source code descriptions are mostly available for software repositories managed by a group. But software applications developed by a single developer or small size software applications rarely contain any informative comments and descriptions. As our proposed model CroLSim's main target is to detect similarity among any kind of open source software repositories, to handle all types of software repositories (without considering size, code quality, number of developers, and so on) we excluded comments available in the source code or descriptions of API calls or developed APIs for this study.

Once we queried the corpus with the extracted API call, we retrieved the API documentation for each of the API calls and added them to a single text file. We named this text file 'DReP'. For a single software application we usually extracted a lot of API calls which might lead to a large DReP as we added retrieved API documentation for each of the API calls. To reduce the size of the DReP, if an API is called multiple times, we added API documentation to DReP only once as our main target was to find out the characteristic and feature of the software application. For each of the software applications in the repository, we generated a single DReP and saved it inside the repository along with the software application. With this approach, we did not consider user-defined methods and their descriptions, but rather we fully depended on API calls developers used in their work. The reason behind this is due to the quality of the documentation used by developers during development of the user defined functions; very often it is found that developers did not use any documentation when developing their method's operation inside the source code. In our evaluation section we show that our method outperforms all the existing methodologies in this way.

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



(a) Vector Dimension vs Training Accuracy



(b) Window Size vs Training Accuracy

Fig. 7: Doc2Vec training accuracy with change in parameters

Once creating a DReP for each software application in the software repository for all the programming languages, we used these DRePs as input to the word vector learning model Doc2Vec. We adapted the library and structure of Doc2Vec from Genism [41] with some small modification in the parameters. We set the parameters of Doc2Vec according to the following way for our study: 50 epochs, 10 window size, 2000 dimensions, 0.25 learning rate and number of projects considering as sample size. Experimental results shown in Figure 7a and Figure 7b helped us to finalize the values of the parameters to be used in our experiment. For performing these experiments, we used the baseline values of the parameters proposed by Le and Mikolov in their work Doc2Vec [10]. For epoch and learning rate, we used the same values as the authors suggested. For finding the most accurate and cost effective values for window size and vector dimension, we used the proposed value for one parameter and tried to see the training accuracy by varying the other parameter. By this we found the right values of window size and vector dimension for our model. The reason behind this experiment is to configure high dimensional word vector for training CroLSim as our configured DRePs are a collection of retrieved API documentations instead of a regular paragraph. As a result, most of the sentences are not related to each other. On other hand, DReP also depends on 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 high dimension vector representation for each of our word. We executed our model using the Tensorflow [42] deep learning platform. Finally, for calculating similarity between two projects, we determine the cosine similarity between Doc2Vec scores for two DRePs (the DRePs represent the documentation of the API calls

that occurred in the two projects) according to the following equation:

$$CosineSimilarity = \cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} \quad (4)$$

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

Using the cosine similarity, we generated a matrix where both row and column of the matrix represent names of the project samples of the software repository. 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 software applications where similar software applications will appear in the same cluster. We used the value of K as $K = 2, 3, 4...n$. Among them, we found that for our dataset $K = 5$ works better. Actually, for having an optimal value for K, we needed to have previous knowledge about the variations of software applications present in our software repository. In a real scenario, it is impossible to know about the varieties of stored software applications. So, we did not fix any value for K and kept it as one of our future research questions.

F. CroLSim's Workflow

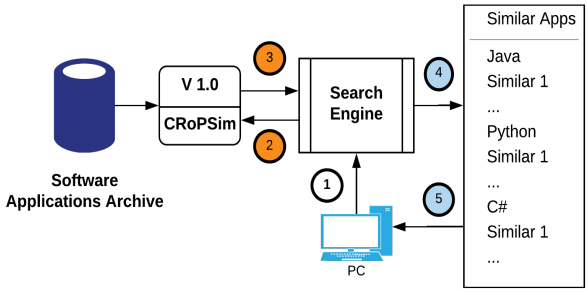


Fig. 8: Workflow for detecting Similar Software Applications

Working steps for CroLSim is stated in Figure 8. In step (1) a user queries using the name of the software application they are looking for. This step can be performed in two different ways. In the first way, a user can just enter the name of the software application as a query. In the second way, a user can upload the entire software application as a query to find similar software applications across different platforms (Java, Python or C#) from the software applications archive. Once the query is entered, in step (2), the search engine sends the query to the CroLSim execution panel. CroLSim is directly linked with the software applications archive where each of the software applications in the repository is already trained according to the architecture stated in Figure 6. For querying with the complete software application CroLSim trains it with pre-trained data available from the CroLSim execution and responds with similar applications from the software repository. If the query is with the name of the software application and it is in the repository, then CroLSim responds with the related software applications of the queried one. If the query is with the name of the software application but it is not in the repository, it will then try to match with the name of the software applications in the software repository. The main focus of this work is to detect similar software applications based on the software application source code. Designing an efficient search engine is out the the scope of this work. Consequently, in step (3) the results of the query executed by CroLSim are returned to the search engine. The search engine ranks all the resulting software applications based on their cosine similarity score in step (4) and adds them to a single page with link to each of the software applications. Finally, in step (5) this page is returned to the user.

IV. Experimental Setup

In this section we are going to show how we designed our CroLSim experimental setup. We used an Intel(R) Core(TM) i7-2600 3.4 GHz CPU with 16GB RAM to run the experiments with CroLSim. Search and retrieval engine related tools and models are usually evaluated by experts with manual relevance judgments [7], [8], [43]. We validated the results of CroLSim with 12 participants who have all been involved in software application development for at least 5 years. To evaluate our work, we compared our model results with two of the base methods, CLAN [7] and Repopal [8] as these two models are closely related to our work and are state of the art solutions in detecting software similarity. In the following sections we discuss the baseline methods and our experimental setup.

A. Baseline Methods

Our goal of developing CroLSim was for detecting software similarities across software applications written in different programming languages, in particular because there is a marked lack of tools and research for detecting software application similarity across different programming languages. Some of the work, Thung et al. [11] and Yang et al. [8] claimed that their work can be extended to software similarity detection across different programming languages although they have not provided any evaluation regarding their claims. Among them, Collaborative Tagging based software similarity detection [11] is hard to work with in real scenarios as their work requires manual tagging with tag words or topics [44] which are usually available in different software repositories. Unfortunately, most of the time, developers or users of a software application do not associate it with tags available in the repository. Moreover, collaborative tagging mechanisms face a number of challenges [45], which further hinders the acceptance of this model. Further, this work does not fit with our work as it works with external information rather than focusing on the information available within the software application itself. Thus we did not evaluate our work with respect to [11].

Repopal [8] was mainly developed for organizing GitHub repositories. The authors of this work used three features to detect similar software repositories in Github; they are Readme files relevance, Stargazer relevance and Time Relevance. Readme files are usually provided by the developers or

owners of the Github repositories where developers provide descriptions regarding the repository or tool they are going to share with open source community. This readme files sometime contain instructions on how to use the shared tool to make help to the open source community. This document can be called as developer document which is usually provided with almost all of the software repositories in Github. It is expected that, the tools which are solving the same problem should contain the same information in their readme files. To evaluate CroLSim, we forked the relevance calculation mechanism of readme files stated in Repopal and showed performance comparison between them. The second feature is Github Stargazer which refers to when the user puts a star on a repository if they are interested in that repository or tool or want to approve this tool or repository for future use. In some other cases a user puts a star to keep track of that repository, its updates, or any change or addition of features to that repository. In Repopal authors determined the Stargazers of a repository first. After that, they tried to define a similarity score between two Stargazers keeping in mind how many repositories are commonly starred by them and how many repositories are in total starred by those two Stargazers. Once they found similarity scores for all of the Stargazers for two repositories, they calculated the relevance score between two repositories by averaging the similarity scores of the Stargazers. The Stargazer feature, or the power of putting a star on a repository is solely a feature of Github. Moreover, in Github Stargazer information is available only for a few repositories as only some of the popular repositories are starred by users. Even some of the repositories are starred by the developers only so that they can keep track of their own repositories. As a result, the authors of Repopal only evaluated their work with 1,000 software applications or repositories which is a very small number compared to the number of repositories in Github. As our CroLSim does not depend on any of the specific features of any open source software application repositories we are not going to evaluate our CroLSim with the Stargazer feature of Repopal. Finally, the last feature of Repopal was time relevance between two repositories by a single user. The authors of Repopal assumed that two repositories created by a single user within short time difference will be similar to each other. This feature is also a Github feature, usually used to track a user's everyday activity, which is hard to extend to other open source repositories. So we also do not consider the time relevance feature of Repopal and only considered the readme files relevance mechanism of Repopal to evaluate CroLSim.

Another recent and successful work on software similarity detection is JavaCLAN [7] where Collins et al. measured API call similarity among different software applications. This work performs well for a single programming language like Java but is not intended for detecting software similarity across different programming languages. Although CroLSim was developed for detecting software application similarities across different programming languages it can detect software similarities for a single programming language as well. To evaluate this feature of CroLSim we compared its performance with that of JavaCLAN.

B. DataSet Collection

For collecting software applications from GitHub, we developed a crawler, named GitCrawl which crawls throughout the GitHub repositories and downloads C, Java, Python, and C# software applications. With the help of GitCrawl we traversed more than 350K 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 source code except some descriptions or PDF files or other documents. We manually removed those repositories and finally selected 4,139 C, 8,956 Java, 7,658 C#, 10,232 Python applications for the CroLSim evaluation.

C. Evaluation Methodologies

In this section, we discuss how we evaluated CroLSim's performance with the help of 9 participants and performance measurement matrices.

1) Confidence: Confidence level means the degrees of relevance one participant can provide at time of validating the results of detected similar software applications of our proposed model. We asked our participants to evaluate the detected similar software applications related to their query and after completing manual evaluation, by giving a confidence score ranges from 1 to 5 according to Table II. We built an application with appropriate user interface 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 $20*2 = 40$ recommendations for four programming languages in total. As a result, we got 480 recommendations and related confidence scores to evaluate our work. We could do more, but as each of our participants needed to validate 20 software applications for each of their queries, which was quite time consuming ($>$9hours/query) and required considerable manual labour, we limited our validation process to 24 queries in total. However, since 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.

TABLE II: Degree of relevance assigned by participants

Score	Relevance	Description
1	Highly Dissimilar	The queried tool and the retrieved tool are fully different from each other
2	Dissimilar	The queried tool and the retrieved tool are mostly different from each other
3	Neutral	It is hard to say whether the queried tool and retrieved tools are similar or not
4	Similar	The queried tool and the retrieved tool are similar to each other at various points
5	Highly Similar	The queried tool and the retrieved tool are fully similar to each other

2) SuccessRate@T: We adapted this performance analysis parameter from Zhang et al. Repopal [8] 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 value of T can be any value of our defined confidence level. E.g: If we assign confidence level 5 as value of T (SuccessRate@5) and top-5 recommendations by our system for a given query are evaluated by our participant with the confidence level 4, 5, 4, 3, 1, we will consider this as a successful similar software detection by our **CroLSim**. On the other hand, if top-5 recommended tools 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: The main goal of the user study is to validate the performance of our proposed model. In other words, to see whether our model is successful enough to detect cross-language similar software applications without any restriction to development platform. As we didn't have any previous knowledge regarding the collected software applications of our testing repository, it is required to evaluate the results manually. And to keep it bias free, we sought help from the people who are experienced with programming languages and software applications development and are not related at any respect to the development of our model. And at the same time, we ensure that the participants of this user study are not known about which tool they are using during their study. To do this, we covered the tools' name by a tool id without the tool name by themselves.

To cross validate our work, we divided our participants randomly in two groups. Each of the group participants were asked to run CroLSim in their own way. We divided our whole experiments in two different sets. For the first set, we defined a list of tasks needed to perform by the participants and gave it to the participants along with a list of software applications we collected from Github. Each of the participants was asked to perform query with the name of one of the software applications for each of the programming languages from the repository. To ensure the accuracy and and adaptability of cross validation, we did not provide any restrictions to the participants at the time of selecting software applications for performing queries and have given freedom in choosing software applications from any language. Once a user queried with the name of the software application for one language, related software applications for other three programming languages are resulted and listed on the interface of the CroLSim tool. We limited the number of software applications on the interface to 5 so that the redundant queried results do remain from out of the focus. Once the queried results are shown, we asked our participants to assign a confidence level, C, to each of the resulting software applications according to Table II. As we used software applications from 4 programming languages for evaluation, namely C, Java, C# and Python, we ensured that all the participants of our study have experience with working with these programming languages. To ensure this, we asked our participants to self declare their experience with the programming languages we used in our model. And we invited only those who had experience on using these programming languages for more than 3 years. For the second set of study, we asked our participants to upload and query with a single software application along with source code and perform the same steps up to assigning a confidence level to each of the resulted related applications for each of the programming languages.

Finally, when both of the groups are 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 have collected the final results once they are finally accepted by the validating participant.

4) Precision: For our evaluation we defined precision as the portion of similar and highly similar applications recommended by our model in the top-5 positions for a given application repository query, which is the same definition that is used by the authors of Repopal [8]. 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 tools individually across different programming languages, e.g., we calculated mean and median precision between C and Java, C and C# and C and Python, Java and Python, Java and C# and C# and Python respectively. At the 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 behind this is to evaluate the performance of **CroLSim** in a real life scenario and bias free experimental analysis.

5) Base Work Development: We developed CLAN, one of our baseline methods described in section IV-A, following the description and experimental setup discussed in CLAN's own publication [7]. We collected API classes and package names for the JDK1.8[14] to build API archive. As it is not defined in the publication on which software applications McMillan et al. performed their evaluation for CLAN we applied and evaluated our developed CLAN on our collected software applications from Github. We experienced an average of 2-3% error with the evaluation results given in the original publication but this is accepted as our software repository is different from them. At the same time, Zhang et al. also evaluated CLAN for their work [8]. We didn't find much different with their evaluation of CLAN with our developed one. So we can say, the CLAN

[14]https://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html

we developed is a good fit for evaluating our proposed model **CroLSim**.

Another baseline method is detecting similar software applications based on the similarity of Readme files adapted from Zhang et al. [8]. For this, we collected the Readme files from each of the software applications in the repository and applied Vector Space Model (VSM) on them to calculate relevance scores according to the description given in the publication. We also forked the Readme files relevance calculation mechanism from Repopal Github repository and have tried to evaluate our developed model with their one. Our experiments show similar performance for both of the models.

6) Research Questions: For performing the evaluation work, we designed our experiments to find the answer of the following research questions:

RQ1 What are the SuccessRate@T Scores of CroLSim, CLAN and Repopal(Readme) for a single and cross programming languages?

RQ2 What are the Confidence Scores of CroLSim, CLAN and Repopal(Readme) for a single and cross programming languages?

RQ3 What are the Precision Scores of CroLSim, CLAN and Repopal(Readme) for a single and cross programming languages?

V. Evaluation

In this section, we are going to answer the research questions we have defined in earlier section for evaluating the performance of our proposed model. As a support for answering the questions we will also show our experimental results and have a discussion on the results.

A. Answer to RQ1

Our experimental study on success rates for three tools, CLAN, Repopal(Readme) and CroLSim are shown in table III. From the table we can see that CroLSim has higher success rate than all other models for both single programming language and cross-language programming language. For single programming language e.g. C, Java, Python and C#, CroLSim can recommend software applications for a given query with SuccessRate@4 (contain at least 1 similar recommendation for the given query) for a rate of 71%, 72%, 78% and 81% respectively where the very related work CLAN has been observed 68%, 65%, 62% and 73% respectively at a rate of SuccessRate@4. For this scenario, Repopal (reademe) application has been observed with very poor performance, 28% for C, 32% for Java, 38% for Python and 31% 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; 65%, 68%, 64% and 75% for C, Java, Python and C# programming languages respectively. For this scenario, other models, CLAN and Repopal (Readme) have observed a highly lower success rate (lower than 45%) comparative to CroLSim.

TABLE III: SuccessRate comparision for CLAN, Repopal(Readme) and CroLSim (N/A=Not Applicable)

Programming Language	SuccessRate@4			SuccessRate@5		
	CLAN	Repopal (Readme)	CroLSim	CLAN	Repopal (Readme)	CroLSim
C	68%	28%	**71%**	37%	22%	**65%**
Java	65%	32%	**72%**	32%	21%	**68%**
Python	62%	38%	**78%**	28%	25%	**64%**
C#	73%	31%	**81%**	44%	26%	**75%**
C & C#	N/A	27%	**82%**	N/A	21%	**74%**
C & Java	N/A	24%	**76%**	N/A	23%	**68%**
C & Python	N/A	24%	**68%**	N/A	22%	**63%**
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%**

We tried to find out the cause of CroLSim's high success rate than others. In CLAN, it fully depends on JDK API calls (For C, available API documentation in web, For C# we considered .netframework 4.6.1 and for Python, we considered python library 2.7.14 and 3.5.4). But at time of developing a project, developers usually use a lot of external third party libraries and APIs. CLAN didn't consider those external API calls which actually made its work hard to detect a concrete recommended tool for a given query. CroLSim outperforms CLAN on this point as it considered not only the development platform provided APIs but also external API calls used in a software application. On other hand, Repopal (Readme) only depends on the quality and information received from analyzing Readme files usually written by developers. Most of the Readme files consists of a very small information (very often with less informative about the tool) regarding the tools they have come with. It is also found that, some readme files contain description on how to use the tool rather than a short description regarding the tool. So, it is really hard to detect similarity between two readme files only based on semantic analysis. As a result, we observed a very lower success rates for both SuccessRate@4 and SuccessRate@5 respectively in case of Repopal (readme).

For cross-language similar software application detection, at SuccessRate@4, we experimented that CroLSim can recommend software application tools with a success rate of 82% between C and C#, 76% between C and Java, 68% between C and Python, 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 has observed a lower success rate than 30%. The performance of CroLSim here depends on the PV_DM model described in section III-D. And for generating DReP, we used the API descriptions provided by the developers and maintainers of different programming languages which remain unchanged for a long period of time. So, the performance we examined for CroLSim will remain same even if we apply them with other software repositories except the one we used.

B. Answer to RQ2

The mean and median confidence scores of all three tools, e.g., CLAN, Repopal (Readme) and CroLSim are stated in Table IV. From the table we can see that, for single programming

TABLE IV: Confidence Score comparison for CLAN, Repopal(Readme) and CroLSim (N/A=Not Applicable)

Programming Language	Mean Confidence Score			Median Confidence Score		
	CLAN	Repopal (Readme)	CroLSim	CLAN	Repopal (Readme)	CroLSim
C	2.78	1.9	**3.75**	2.6	1.8	**3.8**
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**
C & C#	N/A	2.1	**3.64**	N/A	1.71	**4.0**
C & Java	N/A	1.74	**3.26**	N/A	1.65	**3.9**
C & Python	N/A	1.68	**3.31**	N/A	1.4	**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**

TABLE V: Precision score comparison for CLAN, Repopal(Readme) and CroLSim (N/A=Not Applicable)

Programming Languages	Mean Precision			Median Precision		
	CLAN	Repopal (Readme)	CroLSim	CLAN	Repopal (Readme)	CroLSim
C	0.256	0.154	**0.784**	0.4	0.19	**0.7**
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	**0.755**	0.46	0.2	**0.64**
C & C#	N/A	0.162	**0.712**	N/A	0.22	**0.62**
C & Java	N/A	0.17	**0.722**	N/A	0.19	**0.58**
C & Python	N/A	0.146	**0.684**	N/A	0.2	**0.57**
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**

language, CroLSim has showed mean and median confidence score 3.75 and 3.8 respectively for C, 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#. For this scenario we examined that CLAN has observed a mean and median confidence score of 2.78 and 1.9 for C, 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#. 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. Thus, it actually detects a lot of similar tools which are not similar at all. Eventually, the confidence scores deteriorates a lot. C# development is mostly dependent of framework APIs, thus CLAN performs good, almost near to the CroLSim. For Repopal (readme) we noticed that its recommendation based on readme files similarity gets very low mean and median confidence scores (less than 1.9 most cases) which is possibly not even acceptable as a similar software recommendation method itself.

For detecting cross-language similar software applications from a repository, we examined a good and acceptable performance for CroLSim. For recovering similar software applications between C and C#, the mean and median confidence scores observed are 3.64 and 4.0 respectively, for C and Java, they are 3.26 and 3.9 respectively and for C and Python they are 3.31 and 3.6 respectively. 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 a great accuracy.

C. Answer to RQ3

At this point we are going to discuss about the mean and median precision we examined for our method and base line methods for the submitted and recommended queries. 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 C, Java, Python and C# we observed mean average precision values 0.784, 0.764, 0.798 and 0.755 respectively and median precision values 0.7, 0.68, 0.7 and 0.64 respectively. For C and Java, CLAN achieves a mean average precision of 0.256 and 0.284 and median precision of 0.4 for both 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 defines 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 C and C#, the mean average precision for CroLSim is 0.712 and median precision is 0.62. For C and Java, the mean average precision we observed is 0.722 and median precision is 0.58. And For C and Python, the mean average precision is 0.684 and median precision is 0.57. For Java and Python, the mean average precision for CroLSim is 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 could 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 repositories across different programming languages, a mean average precision of more than 0.65 is an acceptable one for all respects.

VI. Code Search using API documentation

In the above, we showed that our proposed tool performs considerably better than other alternatives in detecting cross-language similar software applications. In this section, we

further wanted to explore the idea of detecting cross-language code fragments, at a very finer granularity, from an entire software application to a code fragment. Finding similar code fragments of other languages for a code fragment of a specific programming language may foster program development in different languages for a popular software application. This may also help in software maintenance and program understanding in case someone is expert in one programming language but need to work with the applications of another.

Similar to detecting cross-language similar software applications, recommending similar functional source code fragments in cross-language environment is also challenging to accomplish. The main challenge is the diversity sustains across different programming languages in terms of code structure, semantic and syntactic difference, difference in generated intermediate states and so on. Some group of researchers formulated this problem as detecting cross-language code clone detection and tried to solve this problem in different ways such as CLCMiner [74] which extended source code semantic similarity, LICCA [75]- a common intermediate state for different programming languages, learning AST of code blocks in different programming languages [80] and so on. Although all of these models tried to detect cross-language clones they possess a lot of restrictions and limitations noted above which actually made these tools impractical in real world scenarios.

Thus, in this paper, we extend **CroLSim's** architecture to recommend source code fragments of different languages from repositories for a code fragment of a different language. Note our objective is not to make a better cross-language clone detection tool. Rather, we simply wanted to explore whether the proposed model of **CroLSim's** could also work at a finer granularity. An in-depth study and evaluation of such an approach is out of the scope of this work. In our extended architecture, we assumed that each of the user will have a piece of already developed source code and will search for related source code developed in another programming language from a source code repository. To perform this experiment and evaluation, we collected and created a dataset of having 100K source code fragments developed in Java, C# and Python. The whole process of collecting data is described in Section VI-A. The main goal of our extended work is to see whether API documentation of each programming language can help us recommend source code fragments across different programming languages based on the search source code snippet. From our evaluation we found that, with only help of API calls and their documentation similarity (the main theme of **CroLSim**) used in source code development, extended module of **CroLSim** can recommend source code snippet of a different programming language than the original one with an average precision, P of 0.36 at top-10 position with an average recall rate, R of 0.75 and average F-Measure F_1 of 0.5. These results show promise of the extended tools in finding cross-language code fragments and possibly also for cross clone detection. Again, a detailed evaluation and comparisons of these aspects are our future work.

TABLE VI: Description of Dataset

Total number of code blocks	101,440
Total number of clones	816
Total number of Java codes blocks	34,146
Total number of Python codes blocks	33,025
Total number of C# code blocks	34,269
Average Line of Codes Java	41-48
Average Line of Codes Python	23-27
Average Line of Codes C#	46-53

A. Dataset Creation

For performing our empirical study on detecting cross-language functional clones we have created a database of having more than 100K code fragments developed using Java, C# and Python programming languages. All these code fragments are basically collected from three open source programming contest sites; they are AtCoder[15], Google Code Jam[16] and CoderByte[17]. The reason of collecting code fragments from here was to technically assure all these fragments are functional clones to each other. Before this, to the best of our knowledge, no such dataset of cross-language clones was available. Usually, cross-language clones mean two or more code blocks developed in different programming languages will be functionally similar to each other. The open source contest sites usually accept solutions for a posted problem without any language restriction. At the same time, all the submitted code blocks are tested with specific input and output data to validate. Thus we can say, for a single problem posted in the site, all the accepted answers are functionally similar to each other.

We manually traversed all the contest sites mentioned above. For each of the posted contest problems, we collected the top 20 accepted solutions for each of the three subject system programming languages. So for each of the posted problem we actually collected 60 functionally similar source codes. By doing this, from our traversal of more than 1300 problems through three programming contest websites we collected over 100K source codes for our experiment.

The present status of our dataset is given in Table VI. For each of the posted problems, we collected at least 20 fully accepted solutions for Java, C# and Python individually. We only considered accepted solutions in a believe that they are already validated and can be considered as a cross-language code clone base. From three different contest sites we visited more than 10K problems and their solutions, collected 60 accepted solutions for three programming languages and finally come up with more than 100K functional code clone blocks which to the best of our knowledge the largest cross-language clones database have ever been created.

B. Preprocessing and Experimental Setup

Like other clone detection tools, we preprocessed the source code fragments of the dataset we had prepared for our work.

[15]https://atcoder.jp/

[16]https://www.go-hero.net/jam/10/languages/0

[17]https://www.coderbyte.com/challenges

We dropped all the comments and other string literals from each of the source code. After that, we extracted API calls from each of the source code fragments. For this case, as of CroLSim for detecting similar software applications, we dropped SVD part, meaning we kept all the API calls even if they were called multiple times inside a source code. The reason for this is in detecting similar software applications, we only looked for the operations what each of the software applications are doing rather than the direct similarity of a piece of source code fragment to the other. Thus, keeping information of only unique API calls helped CroLSim to detect similar software applications. However, at code fragment granularity, it is required to track all the API calls because in source code two similar API calls can be used for two different purposes. Rather than SVD, we used rest of the methodologies of CroLSim without any modification to examine its performance in detecting cross-language similar fragments. And we kept the same experimental setup as we did for detecting cross-language similar software applications.

For performing the validation of our extended model, we evaluated the model in a k-fold cross validation way. Unlike regular k-fold cross validation, we randomly selected source code as search query and recorded the recommended source code fragments based on the search query. From the dataset we can see that our dataset contains over $100K$ unique functional code samples with 60 variations each (20 functionally similar source codes for each of the three programming languages). We performed 300 query over our dataset and observed the performance of the model. These 300 queries are uniquely selected from $100K$ source code samples. And once a source code is selected as query set, its functionally similar code blocks are automatically considered as gold dataset for evaluating the performance of the query (that means each query has 20 source code fragments per language, all of which should be recommended by the system for 100% precision and recall rate). To make sure that all of the programming languages are covered perfectly during our experimental analysis, at time of randomly selecting queries we ensure at least 50 unique queries are selected from each of the programming languages. In other words, our randomly selected 300 queries have at least 50 queries for each of the Java, Python and C# programming languages.

C. CroLSim's Code Search Architecture

From Figure 9 we can see the schematic diagram of CroLSim when it is extended to recommend cross-language source codes fragments and the step by step descriptions on this works. Step 1 to 5 are designed to train the source code fragments with the help of Le and Mikolov's proposed Doc2Vec [10] model. To perform this Paragraph Vector learning technique, in step 1 we extracted all the API calls from each of the source code fragments in the source code repository. We maintained individual document for keeping track of extracted API calls for each of the source code in the repository. We used AntlrV4 [81] to generate Abstract Syntax Trees (AST) and extract API calls from that tree. Once

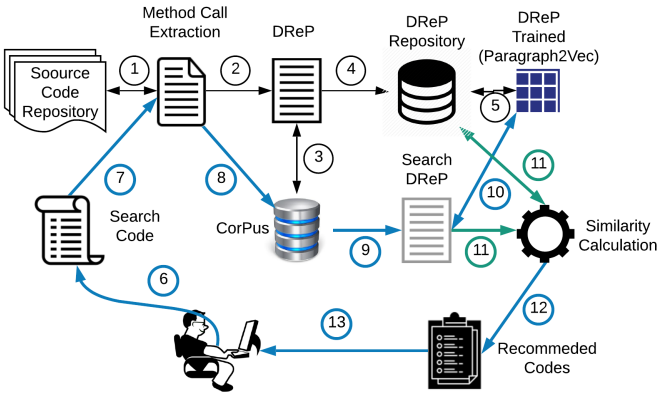


Fig. 9: Schematic Diagram for Cross Language Code Search

API calls are extracted from all the source code fragments, we created a DReP for each of the source code fragments. Each DReP consists of the API descriptions provided by the developers of the programming language. So, in step 3, each of the API call descriptions are pulled from the corpus and listed in the DReP. In this way, a DReP for each of the source code fragments in the repository is created and in step 4 we store them in the DReP repository. For each programming language, we maintained an individual source code and DReP repository. In step 5, using Doc2Vec we trained each of the DRePs stored in the DReP repository and stored those trained vectors in the DReP repository. This training is performed for each of the programming languages inside repository and cross repository, means each repository for a programming language is trained with another programming language's repository. For Doc2Vec, we used the same configurations for parameters we had used for CroLSim at the time of detecting cross-language similar software applications. Performing the whole training process took us a total of 15 hours to accomplish. In step 6, we assume that a developer has already developed or has a piece of source code in one programming language and is looking for functionally similar source code fragment in another programming language. They enter that code snippet as a search query. Once they query with that code snippet, API calls are extracted from it in step 7 and the corpus is used for extracting related API documentation. Once API documentation is extracted, in step 9, all of the API documentation is listed in a new DReP named SearchDReP. In step 10, this newly created SearchDReP is trained with the already trained model created in step 5.

Once the training is finished, in step 11, with the help of the Cosine similarity mechanism, similarity between SearchDReP and DRePs from the repository are calculated and the name of the DRePs listed in a page in ascending order of similarity scores to the SearchDReP. This step is performed for all the repositories of different programming languages individually and results are listed according to their language. Once this

step is performed, in step 12, source code fragments of the listed DRePs are recovered from the source code repositories and are returned to the user or developer in step 13 as a single web page. All the retrieved results are categorized based on the developed language. From this page, a user can select and get their required source code in another programming language which they can easily use for development and rapid prototyping.

D. Experimental Results on Code Search

In this section we discuss the experimental results and evaluation we performed to evaluate our model in performing cross-language code search. We considered three matrices for the full evaluation, they are Precision P, Recall R and F-Measure F_1. These three matrices are widely used and adapted by most of the researchers in code search and source code similarity detection studies [77], [82]. The research questions (RQ) which we tried to answer during our evaluation process are:

- RQ1. What is the observed precision rate P of CroLSim in searching similar code examples among the programming languages Java, Python and C#?
- RQ2. What is the observed recall rate R of CroLSim in searching similar code examples among the programming languages Java, Python and C#?
- RQ3. What is the observed F_1 of CroLSim in searching similar code examples among the programming languages Java, Python and C#?

In the following section we are going to answer all these research questions.

1) RQ1: Precision Rate: From Table VII we can see the precision rate we observed. We can see that, for cross-language source code recommendation on Top 10 position, average precision rate is 0.20 plus. While the precision values are low, they are kind of promising given that we only consider API calls and their documentation used in a programming language and using that information we tried to detect source code similarity. But our manual observation with some of the code fragments says that for different cases, with some modifications in code structure, addition of looping or additional features like try/catch block, change in operations, statements and expressions, the same API calls can be used to perform different functionality based on the requirement. As a reason, some of the code blocks from repository with the same API calls are recommended by the system even if they are not actually similar to the queried source code. On Top 20 recommended code blocks, we observed an average precision of 0.27 for all of the subject systems and for Top 30 recommendation, an average precision for all 300 random queries is 0.27. So we can say, using only API calls' documentation similarity of source code fragments across 3 subject system programming languages we achieved around 30% precision in recommending cross-language source fragments.

2) RQ2: Recall Rate: : As discussed earlier, for each of our successful queries, there is 10 functionally similar source code fragments for each of the subject system programming languages. From Table VII we can see that, for Top 10 recommendation, the Recall rate R for searching out cross-language similar source code fragments using direct source code as query is on average of 0.21 among the three subject system programming languages. At top 20 recommended source codes, the observed recall rate is on average of 0.60 and finally for top 30 recommendation, the recall rate we experimented is on average of 0.82 for all the three subject system programming languages among them. From this observations we can say that, with API calls similarity in source codes across different programming languages, with its present structure, **CroLSim** is successful enough to find out similar source codes across different programming languages. We have already discussed the reason behind our claims in previous section. In addition to that, to the best of our knowledge, we are the first to use API calls' documentation similarity in detecting cross-language similar code fragments, and search code blocks in repository by directly querying with source code from any language.

3) RQ3: F-Measure: : From Table VII we can see that, for all of our successful 300 queries across three subject system programming languages, for top 10 recommendations, **CroLSim** has observed an average F-measure score of 0.21. For top 20 and top 30 cross-language source code recommendations, the average F-Measure score we observed is around 0.40. From these experimental results we can say, API calls' documentation similarity can be a promising way to search similar code blocks from a source code repository as well as detecting cross-language code clones in different versions of a software applications or across different software applications.

VII. Threats to Validity

In this section we discuss the threats to validity of our proposed system and our evaluation. As well, we discuss the mechanisms and precautions we took to minimize the threats.

Participants. The evaluation of our proposed model is primarily based on the similarity scores given by nine participants. We considered some factors which may create threats to the validity of the evaluation results. They are: (a) Interest of the participants to carefully evaluate the results, (b) The familiarity of the participants with Java, Python, and C# programming languages, and (c) consistency of participants in evaluating the responses the system produces.

For mitigating the effects (a) we asked our participants at the beginning whether they are interested in evaluating the results of the three systems (CLAN, Repopal (Readme) and CroLSim) we considered for our work. We already acknowledged that the process of manual validation is time consuming and requires a lot of patience. At the time of selecting participants we ensured that all our participants were really interested in evaluating our work and had enough time to evaluate the observed results (according to our statistics, each of our participants required

TABLE VII: CroLSim's Performance in searching similar cross-language source codes from repositories

Programming Languages	Searching Source Code								
	Top 10			Top 20			Top 30		
	Precision P	Recall R	F-Measure F_1	Precision P	Recall R	F-Measure F_1	Precision P	Recall R	F-Measure F_1
Java & C#	0.25	0.25	0.25	0.31	0.62	0.41	0.28	0.87	0.42
Java & Python	0.18	0.18	0.18	0.26	0.53	0.35	0.27	0.79	0.40
C# & Python	0.21	0.21	0.21	0.28	0.56	0.37	0.27	0.81	0.405

an average of two hours to perform the evaluation process with the three subject systems).

It is hard to evaluate the expertise of the participants with C, Java, Python and C# programming languages. We do consider that a lack of knowledge of a programming languages will affect the validation process of our work. To reduce this threat, we only invited participants with more than 5 years development experience with the subject system programming languages individually. Most of the participants are from inside our organization which actually helps us to verify their proficiency with the programming languages.

For maintaining the consistency of the participants and the standard of the evaluation we tried to evaluate one query for three of our subject systems with a single participant. Once a query is evaluated, it is verified by another participant. In this way we tried to maintain the standard of evaluation and the decision of degree of similarity equality distributed for each query.

Repositories. In this study, we evaluated 18 queries in total with the help of external participants, which is low in number. We could do some more queries but the time and cost require for evaluation have limited our number of queries. In future we will increase the number of queries to further evaluate our work. Our baseline methods also performed around the same number of queries when evaluating their work.

Another important threat could be the quantity and quality of the repositories. We collected 4,139 C-based software applications, 8,956 Java software applications, 7,658 C# software applications, and 10,232 Python software applications from GitHub. For ensuring the quality of the applications we selected only those software applications which are executable. And as our goal is to work with the real time scenario, we not only tried to be selective with the software applications for repositories and have tried to work with a portion of regular repositories. In this way hope to limit the threats to validity with the selection of repositories.

For the code search problem, the repositories we collected and created for the experiment are solutions to different programming contest problems which were made available by programming contest sites. We collected only the accepted solutions in three subject systems for each of the posted problems from those contest sites. As the solutions were already accepted by a group of judges of each contest sites, so we can say, accepted solutions in different languages for each of the problems are functionally similar to each other. And we collected more that $100K$ source code examples in three different subject systems which we think are enough to perform our experiment and report the results at this stage. We have a plan to increase the source code examples in the near future.

Programming Languages. Open source software application repositories like Github, SourceForge, BitBucket, and so on contain software applications built with different programming languages. For evaluating CroLSim we randomly selected software applications written in three different programming languages (Java, Python and C#) to show our approach is workable in detecting cross-language software similarity. This model can work with other programming languages as well.

Third party API documentation. Unavailability or poor quality third party API documentation can also be a threat for the validity of CroLSim. As our work is dependent on the user documentation of the API calls, this information is very important. To limit this threat we tried to use the API documentation which is made available by the developers of the programming language's own platform. For external third party libraries, we collected the available API documentation first. If the documentation is not available, we collected the source code and generated the documentation for that API. By this we maintained the quality too because usually the available documentation is already verified and understandable by the users.

Evaluation Metrics. The evaluation metrics we used in this work was adapted from the closely related models, CLAN [7], [11], and Repopal [8]. These metrics are also used in various previous studies [6], [46], [47]. In this way we hope to lessen the probability of a threat by using the same evaluation metrics.

VIII. Related Work

Detecting similar applications in a software repository is not a new research topic. There are a number of approaches related to our work, or related to the model we used. Below we provide a summary of those approaches.

A. Detecting Similar Repositories

Studies closely related to our proposed CroLSim model include: Kawaguchi et al. [6], McMillan et al. [7], Thung et al. [11] and Zhang et al. [8]. Kawaguchi et al. only considered

the semantics of source code with their MudaBlue tool [6]. Their work also requires manual investigation of informative semantics of source code. Collins et al. in their tool CLAN [7] used the JDK API calls similarity to detect similar Java based software applications from the repository. They had worked with more that 8k applications to evaluate their work. This work is not applicable to detect similar software applications across different programming languages. In addition to this, depending only on JDK API calls may only partly represent the source code of a software application which may lead to identifying the wrong software applications as being similar or not. In another work, Thung et al. [11] proposed to work with manual Collaborative Tagging of applications for detecting similar applications. This work is challenging to implement in detecting software similarity in real world software application repositories because of its sole dependency on manual evaluation and user experience. Recently Zhang et al. [8] evaluated three pieces of information for each software application extracted from Github to detect similar software, Readme files relevance, Stargazer relevance and Time Relevance. Their approach works for Github alone. Even Stargazer information is not available in a lot of Github repositories. Zhang et al. evaluated their result only on 1000 repositories which is possibly not enough for such studies.

In our work, CroLSim, we tried to address the limitations of the existing solutions discussed above. Our work is based on the documentation of API calls and the Deep Convolutional Neural Network based Paragraph Vector model, which has not been studied before in this context. To evaluate our work we compared our model to the work of McMillan et al. and Zhang et al. and demonstrated with rigorous experiments that our model outperforms those approaches in detecting similar software applications for both single and cross-language software applications.

B. Recommendation Systems

There have been a great many recommendation systems, such as the work of Bajracharya et al. [48], Thung et al. [49], Bauer et al. [50], and Cubranic et al. [51]. Structural Semantic Indexing (SSI) as proposed by Bajracharya et al. [48] associates direct words with source code blocks based on API call similarity. Thung et al. [49] recommended libraries to developers by analyzing association rules among the libraries. They considered the present library call and the use of that library in other software applications to reveal a usage pattern for that library. Cubranic et al. [51] in their tool Hipikat tried to recommend artifacts by analyzing archives which might be helpful for understanding that project. All these studies are related to our work as we leveraged the similarity of API usage patterns in the source code to detect and recommend similar software applications. But we differ in scope since we focused on detecting and recommending similar software applications across different programming languages from source code repositories, which is not studied prior to our work. The recommendation system research focused on systems written in the same programming language.

In addition to this work, quite a few context-based approaches are used for bug detection and localization in source code [52], [53], [54]. Most of these bug localization and fixing techniques are for systems written in a single programming language. In the future we plan to extend our work to detecting bugs in cross-language software applications.

C. Code Search and Clone Detection

Several studies search for code fragments from source code, including, Exemplar [47], SNIFF [82], Portfolio [46], Parseweb [56], Spotweb [55], and CCFinder [62]. 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 CroLSim as we performed recommending similar software applications based on search terms or using software applications directly. But we also extended CroLSim for recommending cross-language source code fragments where we used source code directly as a query. This feature made this model unique and is different from existing models as none of them support cross-language source code recommendation directly.

At present many source code clone detection tools are available, such as SourcererCC [57], NiCad [58], and many others [59] which use different text, token, AST, graph and other approaches to detect similar code fragments across different software applications for single languages [68], [72], [73]. Cross-language code clone detection in software applications is also studied in different research work, e.g. Craft et al. [60], Zheng et al. [61], and so on. However, like these approaches, an extended version of CroLSim can be used to detect cross-language clones too. We have a plan to work on this too in the near future in order to increase the accuracy of CroLSim in detecting cross-language functionally similar code blocks.

D. Software Categorization

Another research area related to our study is categorizing software applications of a software repository. A good number of studies have already been conducted, including by Kawaguchi et al. [6], Wang et al. [63], Xu [64], and Zhang [65]. Wang et al. proposed a SVM-based hierarchical software categorizing approach by analyzing and aggregating different online profiles across different software repositories with a good accuracy in terms of precision, recall and F-measure. However, the primary basics of categorizing software applications is by detecting similar software applications. Once one can detect similar software applications, with the help of different clustering approaches, such as KNN, Random Forest, and SVM, it is possible to cluster software applications based on their similarity score. Therefore, CroLSim can be easily extended to categorize software applications which is discussed in our architecture section and could be done so in a cross-language context, which other categorization approaches cannot address. In the future, we plan to explore this feature in depth.

IX. Conclusion

We proposed CroLSim, a tool and model for detecting cross-language similar software applications from open source software repositories with an average precision of more than 65%. To the best of our knowledge, we are the first who have studied a deep neural network based natural language processing technique for finding a universal solution for software similarity detection which can perform equally for both single and cross-programming language applications. We extracted API calls from applications and used API documentation related to the API calls for detecting similar software applications. Our experimental evaluations show that CroLSim outperforms the state of the art techniques in detecting similar software applications in every aspect with a significant performance gain. In addition, we extended the work for searching functionally similar code blocks from a source code repository by querying with the source code directly and without the help of any natural language query processing. Early results show promise and we plan to explore this further in the future. Our plan is to extend this work to detecting code clones in software applications, searching similar code blocks from different software applications with better accuracy and precision, and automatically categorizing software applications with meaningful names in a cross-language environment. We also plan to improve the performance of CroLSim and study AST-based approaches for detecting similar software applications.

X. Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by a Canada First Research Excellence Fund (CFREF) grant coordinated by the Global Institute for Food Security (GIFS).

References

[1] Afgan E, Baker D, van den Beek M, Blankenberg D, Bouvier D, ech M, Chilton J, Clements D, Coraor N, Eberhard C, Grning B, Guerler A, Hillman-Jackson J, Von Kuster G, Rasche E, Soranzo N, Turaga N, Taylor J, Nekrutenko A, Goecks J. *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

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

[3] Bolser D, Staines DM, Pritchard E and Kersey P *Ensembl Plants: Integrating Tools for Visualizing, Mining, and Analyzing Plant Genomics Data* Netmad

[4] Amir Michail and David Notkin. *Assessing software libraries by browsing similar classes, functions and relationships. In Proceedings of the 21st international conference on Software engineering* . ACM, New York, NY, USA, p:463–472.

[5] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. *Detecting similar Java classes using tree algorithms.* In Proceedings of the 2006 international workshop on Mining software repositories. ACM, New York, NY, USA, 65–71.

[6] S. Kawaguchi, P. K. Garg, M. Matsushita and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," 11th Asia-Pacific Software Engineering Conference, 2004, pp. 184–193.

[7] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, "Detecting Similar Software Applications", ICSE 2012,

[8] Zhang, Y., Lo, D., Kochhar, P.S., Xia, X., Li, Q. and Sun, J., *February. Detecting similar repositories on GitHub.* IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), (pp. 13–23). IEEE. 2017

[9] K. W. Nafi, B. Roy, C. K. Roy, and K. A. Schneider. *CroLSim: Cross Language Software Similarity Detector Using API Documentation.* In 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 139–148. IEEE, 2018.

[10] Quoc Le and Tomas Mikolov. *Distributed representations of sentences and documents.* In Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14), Eric P. Xing and Tony Jebara (Eds.), JMLR.org II-1188-II-1196

[11] F. Thung, D. Lo and L. Jiang *Detecting similar applications with collaborative tagging* 28th IEEE International Conference on Software Maintenance (ICSM), pp. 600–603, 2012.

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

[13] *https://en.wikipedia.org/wiki/Latent_semantic_analysis*

[14] *https://octoverse.github.com/*

[15] S. K. M. Wong and Vijay V. Raghavan. *Vector space model of information retrieval: a reevaluation.* In Proceedings of the 7th annual international ACM SIGIR conference on Research and development in information retrieval, British Computer Society, Swinton, UK, 167–185.

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

[17] Rapp R. *Syntagmatic and Paradigmatic Associations in Information Retrieval.* In: Schader M., Gaul W., Vichi M. (eds) Between Data Science and Applied Data Analysis. Studies in Classification, Data Analysis, and Knowledge Organization, 2003 Springer, Berlin, Heidelberg

[18] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. *An empirical investigation into a large-scale Java open source code repository.* In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA, , Article 11 , 10 pages.

[19] Zellig S. Harris, *Distributional Structure*, WORD, 10:2-3, 146–162, 1954

[20] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[21] Singhal, A. *Modern Information Retrieval: A Brief Overview.* IEEE Data Eng. Bull., 24, 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 Information Science, Volume: 41, no: 6, 1998, p: 391–407.

[24] Q.V. Le *A Tutorial on Deep Learning* Lecture 2015, Stanford University, USA.

[25] Y. LeCun, Y. Bengio and Geoffrey Hinton *Deep Learning*, Nature Review, Vol 521, 25th May 2015.

[26] K. He, X. Zhang, S. Ren, and J. Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* In Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV '15). IEEE Computer Society, Washington, DC, USA, 1026–1034.

[27] LeCun, Y. *Generalization and network design strategies.* Technical Report CRG-TR-89-4, University of Toronto

[28] Bates. M. *Models of natural language understanding*, Proceedings of National Academy of Sciences, vol 95, pp 9977–9982, October 1995.

[29] T. Young, D. Hazarika, S. Poria, and E. Cambria, *Recent trends in deep learning based natural language processing,* in arXiv preprint arXiv:1708.02709, 2017.

[30] D. Jurafsky and J. H. Martin, *Speech and Language Processing.*, Book Draft on August 7, 2017

[31] T. Mikolov, K. Chen, G. Corrado and J. Dean , *Efficient Estimation of Word Representations in Vector Space*, arXiv preprint arXiv:1301.3781, September 2013.

[32] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, link: www.deeplearningbook.org.

[33] 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.

[34] X. Glorot, A. Bordes, and Y. Bengio, *Domain adaptation for large-scale sentiment classification: A deep learning approach* in Proceedings of the 28th international conference on machine learning (ICML-11), 2011, pp. 513–520.

[35] R. Collobert and J. Weston *A unified architecture for natural language processing: Deep neural networks with multitask learning* in Proceedings of the 25th international conference on Machine learning. ACM, 2008, pp. 160–167.

[36] J. Weston, S. Bengio, and N. Usunier *Wsabie: Scaling up to large vocabulary image annotation,* in IJCAI, vol. 11, 2011, pp. 2764–2770.

[37] P. D. Turney and P. Pantel,*From frequency to meaning: Vector space models of semantics* Journal of artificial intelligence research, vol. 37, pp. 141–188, 2010.

[38] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality* in Advances in neural information processing systems, 2013, pp. 3111–3119.

[39] javadoc by Oracle: \http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html

[40] DocFx: https://dotnet.github.io/docfx/tutorial/docfx_getting_started.html

[41] Rehurek, R. and Sojka, P.,*Software framework for topic modelling with large corpora.* In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.

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

[43] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze.*Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008

[44] David M. Blei. *Probabilistic topic models.* Communication ACM 55, 4 (April 2012), 77–84. DOI: https://doi.org/10.1145/2133806.2133826

[45] George Macgregor, Emma McCulloch *Collaborative tagging as a knowledge organisation and resource discovery tool*, Library Review, Vol. 55 Issue: 5, pp.291–300, https://doi.org/10.1108/00242530610667558

[46] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. *Portfolio: finding relevant functions and their usage.* In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 111–120.

[47] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu and Q. Xie, *Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications,* In IEEE Transactions on Software Engineering, vol. 38, no. 5, pp. 1069–1087, Sept.-Oct. 2012.

[48] S. K. Bajracharya, J. Ossher, and C. V. Lopes *Leveraging usage similarity for effective retrieval of examples in code repositories* In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10). ACM, New York, NY, USA, 157–166

[49] F. Thung, D. Lo and J. Lawall, *Automated library recommendation,* 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, 2013, pp. 182–191.

[50] V. Bauer, T. Vlke and E. Jrgens, "A Novel Approach to Detect Unintentional Re-implementations," 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 491–495.

[51] D. Cubranic and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," 25th International Conference on Software Engineering, 2003. Proceedings., 2003, pp. 408–418.

[52] Xin Xia and David Lo. 2017. An effective change recommendation approach for supplementary bug fixes. Automated Software Engg. 24, 2 (June 2017), 455–498.

[53] X. Xia, D. Lo, Y. Ding, and J. M. Al-Kofahi, Improving automated bug triaging with specialized topic model, IEEE Transactions on Software Engineering, pp. 1–1, 2016

[54] X. Xia, D. Lo, X. Wang, and B. Zhou, Accurate developer recommendation for bug resolution, vol. 8144, pp. 72–81, 2013

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

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

[57] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). ACM, New York, NY, USA, 1157–1168. DOI:

[58] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," 2008 16th IEEE International Conference on Program Comprehension, Amsterdam, 2008, pp. 172–181.

[59] C. K. Roy, James R. Cordy, and Rainer Koschke. *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach.* Sci. Comput. Program. 74, 7 (May 2009), 470–495.

[60] N. A. Kraft, B. W. Bonds and R. K. Smith,*Cross-language clone detection* 20TH International Conference On Software Engineering And Knowledge Engineering, 2008

[61] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu and J. Zhao, *Mining revision histories to detect cross-language clones without intermediates* 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 2016, pp. 696–701.

[62] T. Kamiya, S. Kusumoto, and K. Inoue. *CCFinder: a multilinguistic token-based code clone detection system for large scale source code.* IEEE Trans. Softw. Eng. 28, 7 (July 2002), 654–670.

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

[64] 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 The Ieee/acm International Conference, pp. 51–62, 2016.

[65] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, Combining software metrics and text features for vulnerable file prediction, in International Conference on Engineering of Complex Computer Systems, pp. 40–49, 2015.

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

[67] *https://en.wikipedia.org/wiki/Abstract_syntax_tree*

[68] C. K. Roy and J. R. Cordy, *An Empirical Study of Function Clones in Open Source Software,* 2008 15th Working Conference on Reverse Engineering, Antwerp, 2008, pp. 81–90. doi: 10.1109/WCRE.2008.54

[69] Antlr https://www.antlr.org/

[70] Spark https://spark.apache.org/

[71] Apache Lucene http://lucene.apache.org/

[72] C.K. Roy, J.R. Cordy and R. Koschke, *Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,* Science of Computer Programming, 74 (2009) 470–495, 2009.

[73] M. Asaduzzaman, C. K. Roy, K. A. Schneider and M. D. Penta, *LHDiff: Tracking Source Code Lines to Support Software Maintenance Activities,* 2013 IEEE International Conference on Software Maintenance, Eindhoven, 2013, pp. 484–487. doi: 10.1109/ICSM.2013.78

[74] X. Cheng, Z. Peng, L. Jiang , H. Zhong, H. Yu and J. Zhao, *CLCMiner: Detecting cross-language clones without intermediates,* IEICE TRANSACTIONS on Information and Systems, 100(2), pp.273–284, 2017.

[75] T. Vislavski, G. Rakic, N. Cardozo and Z. Budimac *LICCA: A tool for cross-language clone detection,* In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 512–516, March 2018.

[76] D. Rattan, R. Bhatia, and M. Singh *Software clone detection: A systematic review, Information and Software Technology,* Volume 55, Issue 7, 2013, Pages 1165–1199, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2013.01.008.

[77] D. A. Smith, *Rapid Software Prototyping,* Ph.D. Dissertation. University of California, Irvine. AAI8303547.

[78] L. Luqi, and R. Steigerwald, *Rapid software prototyping,* In Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, vol. 2, pp. 470–479. IEEE, 1992.

[79] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang and B. Mobasher, *Recommending source code for use in rapid software prototypes,* 34th International Conference on Software Engineering (ICSE), Zurich, 2012, pp. 848–858. doi: 10.1109/ICSE.2012.6227134

[80] Daniel Perez M.Sc Thesis dissertation: https://daniel.perez.sh/research/master-thesis/thesis.pdf

[81] Antlr version 4: https://www.antlr.org/

[82] S. Chatterjee, S. Juvekar, K. Sen, *SNIFF: A Search Engine for Java Using Free-Form Queries.* In: Chechik M., Wirsing M. (eds) Fundamental Approaches to Software Engineering. FASE 2009. Lecture Notes in Computer Science, vol 5503. Springer, Berlin, Heidelberg

[83] https://github.com/Kawser-nerd/CroLSim