CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation

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

***Abstract*—Software clones are detrimental to software maintenance and evolution and as a result many clone detectors have been proposed. These tools target clone detection in software applications written in a single programming language. However, a software application may be written in different languages for different platforms to improve the application's platform compatibility and adoption by users of different platforms. Cross language clones (CLCs) introduce additional challenges when maintaining multi-platform applications and would likely go undetected using existing tools. In this paper, we propose CLCDSA, a cross language clone detector which can detect CLCs without extensive processing of the source code and without the need to generate an intermediate representation. The proposed CLCDSA model analyzes different syntactic features of source code across different programming languages to detect CLCs. To support large scale clone detection, the CLCDSA model uses an action filter based on cross language API call similarity to discard non-potential clones. The design methodology of CLCDSA is two-fold: (a) it detects CLCs on the fly by comparing the similarity of features, and (b) it uses a deep neural network based feature vector learning model to learn the features and detect CLCs. Early evaluation of the model observed an average precision, recall and F-measure score of 0.55, 0.86, and 0.64 respectively for the first phase and 0.61, 0.93, and 0.71 respectively for the second phase which indicates that CLCDSA outperforms all available models in detecting cross language clones.**

***Index Terms*—Code Clone, API documentation, Word2Vector, Source Code Syntax**

I. Introduction

With the availability of a number of different widely used programming languages and platforms, and to achieve compatibility and adoptability, developers may be required to develop the same functionality in different programming languages for different versions of a software system. A typical example of this is the development of mobile phone games where developers try to make a single game available on different platforms. For example, a very popular mobile phone game is *Temple Run* which is available on three popular mobile phone platforms: Windows[1], iOS[2] and Android[3]. As many differences exist among the different platforms, it is required to develop the games according to their designated platforms such as in C/C# for a Windows phone, Java for an Android phone and Objective-C for an iPhone [1]. Although there are tools available[4] that help to develop cross-platform mobile games, their execution still requires platform specific support. A similar scenario also exists for desktop and web based applications (such as microservices [3]), where the same functions need to be replicated across different programming languages such as Antlr [4], Lucene [5], and Factual [6]. Code fragments related to such functionality replication in different programming languages can referred to as *Cross Language Clones* or CLCs.

Source code clones can be considered both harmful [10] and useful [11] based on their characteristics. Previous research showed that 7% to 23% of the source code of a software system is cloned [7], [8]. Duplicating code blocks in source code can add complexity [2] and requires special attention to maintain and avoid unintended behaviour [12]. Code clones are also responsible for introducing bugs and functionality divergence among codebases which decreases overall software system quality [7], [8]. In addition to these, once a code block is changed in the source code of a software system, it is required to propagate that change to the other cloned code parts too, which usually requires additional effort by the developers or maintainers. However, for most cases, cross language clones are created intentionally by developers and are often unavoidable and cannot be removed [13]. So, to help software developers and curators to manage cross-language software systems in an easy, time-effective and cost-effective way, an automatic cross-language code clone detection technique is required. It is foremost in collaborative software system development where if a developer with expertise in one language changes or modifies their software version, developers with expertise in other programming languages need to perform the same functional change to their respective versions as well. This is more expensive and time consuming compared to changing a single language based software system as it requires prior knowledge regarding the system architecture, as well as understanding the modification of the code performed by the first developer.

Let us consider the example given in Figure 1. In Scenario A, we have code examples for three programming languages: Java, Python, and C#. Each code fragment performs one operation: adding two integers. Now, let us consider that the requirements change and it is needed to add two float-

[1] Windows Temple Run
[2]IOS Temple Run
[3]Android Temple Run

[4]Xamarin: https://visualstudio.microsoft.com/xamarin/
PhoneGap: https://phonegap.com/

ing point numbers instead of integers (Scenario B). At first Java code is modified and updated by a developer (marked with the **green** arrow labeled 'Java'). Now it is required to make the same change to the other versions of Scenario A. Sound knowledge of the application architecture and source code for the different platforms might guide developers to manually make the changes. However such a task would demand considerably more effort during real-life large-scale software development. To help developers in this situation, a cross language clone detection technique would be useful for automatically detecting functionally similar code blocks of the Java version in the C# and Python versions (marked with **blue** arrows). Once the clones are located, changes can easily be made to achieve Scenario B (marked with **green** arrows).

Fortunately, a good number of automated or semi-automated techniques and models have been proposed for detecting code clones in single language based software systems, such as CCFinder [14], Deckard [15], NiCad [16], and SourcerCC [17]. A very small number of models have also been proposed for Cross Language Clone detection, such as LICCA [12], and CLCMiner [13]. Among them, CLCMiner only works with code revision histories (diffs) and detects clones by measuring source code token similarity. A good number of software applications are available whose functionalities are supported by different platforms. For example, Apache Lucene can be applied for Java, C#, JavaScript, and Python. APIs of these tools are almost similar in name across different platforms. At the same time, their usage patterns are almost similar too in different programming languages. Authors of CLCMiner can track the changes or modifications of these API calls across different software applications without language dependencies. But this scenario is totally different from the real world Cross Language Clones given in figure 1. And this model fails to detect or track these clones. On the other hand, LICCA detects code clones based on intermediate states represented by Budimac et al.'s SSQSA [18] architecture. This architecture represents any source code in an enriched Concrete Syntax Tree (eCST). LICCA analyzes the eCST and generates a matrix from it, which was extended for cross language clone detection. But this model has a number of limitations. First, it requires the source code length to be equal. Second, code steps and flow of functionality of two code blocks needs to be the same. These limitations have made this model not applicable in real world scenarios. Very recently, Perez et al. [42] proposed a cross-language clone detection model where they try to learn an AST [22] representation of some given source code with the help of the skip-gram model [47]. Although their approach is sound, their model suffers from low average precision and F-measure scores which makes their model difficult to operate in real world cross language clone detection scenarios.

In this paper, we propose **CLCDSA**, a model for detecting cross language clones by analyzing source codes without the requirement of intermediate representations or restrictions. This model can detect cross language clones by analyzing the similarity of 9 syntactic source code features. From a previous

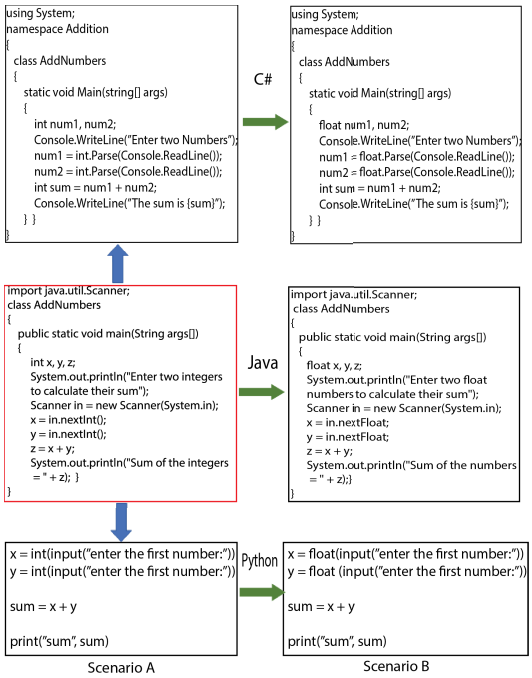

Fig. 1. Source code for adding two numbers in Java, C#, and Python. Scenario A illustrates adding integers & Scenario B illustrates adding floats

study, we found that these features have almost similar values if two source code fragments from two different programming languages have similar functionality. This model can detect cross-language clones in two different ways. First, it can detect clones on the fly without having any predefined knowledge. For this, CLCDSA measures Cosine Similarity between two matrices which are generated by extracting values of the 9 features from source code fragments. Metrics are extracted by traversing the AST of related code fragments. Second, CLCDSA supports a reconfigured Siamese architecture based Deep Neural Network [31], [39] which can automatically learn the values of 9 features from a good number of labeled data and can detect cross-language clones with greater accuracy. For providing support for large size datasets, this model uses an action filter. This action filter is designed based on API call similarity used in two different code fragments across different programming languages. Cross-language API call similarity is learned with the help of the API documentation and Mikolov's Word2Vec [19] model. This filter helps to discard non-probable clone pairs from consideration before advancing to the main model which reduces computational complexity, number of comparisons, and improves scalability. To the best of our knowledge, we are the first to use 9 effective source code syntactical features along with cross language API call similarity in detecting cross language clones (semantic features). For performing our evaluation and validating CLCDSA, we carefully collected a large size valid cross-language clone dataset. From our evaluation we observed that for different threshold values, CLCDSA can detect clones on

the fly with an average precision, recall, and F-measure score of 0.55, 0.86, and 0.6 respectively and for CLCDSA-DeepNN, the observed average precision, recall, and F-measure are 0.61, 0.93, and 0.71 respectively which outperforms all of the available models in terms of accuracy and scalability. The dataset we used for validating our proposed model along with our evaluation results are available for further use [63].

The remainder of the paper is organized as follows: Section II describes the design modules of the model, Section III discusses the architectural design of the model, Section IV briefly discusses the evaluation and validation steps and results of CLCDSA, Section V describes the probable threats to validity and how those were addressed, Section VI describes closely related work, and Section VII concludes the paper.

II. CLCDSA: Cross Language Clone Detector

In this section, we describe our cross language clone detector, CLCDSA. As with other clone detection models, we process source code to extract the metric information needed for our work. We use an **action filter** to filter out non-probable clones from consideration to make our model more scalable.

A. Feature Selection

Table I shows the features Saini et al. [24] proposed in their work as being useful for detecting code clones in Java. Most of the features are derived from the Software Quality Observatory for Open Source Software (SQO-OSS) quality model proposed by Samoladas et al. [25]. Although it has been shown that analyzing all these features is helpful in achieving great accuracy when detecting clones in a single programming language, not all of these features can be used for cross language code clone detection. In a previous study we determined which of the features in Table I were most effective for cross language clone detection. In that study we used a dataset that contained more than 300K functionally similar cross language clone code fragments written in three different languages: Java, C# and Python. We applied each of the features on functionally similar code clones to see the effect and performance of a feature in determining source code similarity. The primary objective of our study was to see how similar the values were for each of the features for three different but functionally similar code fragements. From our brief study and manual analysis, we found that out of 24 features, 9 features use almost the same metrics across the different programming languages (shown in bold in Table I). By observing this, we selected these 9 features for detecting cross language clones in the belief that two code fragments will be functionally similar if they share similar metrics. This study took more than 20 hours of manual labor to effectively find the suitable features. These features can also be useful in other cross language source code analysis based work, and other research work, such as code search [27], rapid prototyping [26], and code quality analysis [28].

TABLE I
Features to Analyse Source Code

Name of the Source Code Feature
Number of External Methods Called
Number of Variables Referenced
Number of Variables Declared
Number of Statements
Total Number of Operators
Number of Arguments
Number of Expressions
Total Number of Operands
Method, Maximum Depth of Nesting
Number of Loops (for,while)
Number of Local Methods Called
Halstead Vocabulary
Halstead Effort to Implement
Halstead Difficulty to Implement
Number of Exceptions Thrown
Number of Exceptions Referenced
Number of Classes Referenced
McCabes Cyclomatic Complexity
Number of Class Casts
Number of Boolean Literals
Number of Character Literals
Number of String Literals
Number of Numerical literals
Number of Null Literals

B. Preprocessing

Preprocessing of given code fragments for CLCDSA is comprised of four steps. First, we remove all the comments and string literals from the source code. The reason behind this is to keep track of only the selected features and their values rather than keeping buggy information to analyze. Second, we generate an Abstract Syntax Tree (AST) [22] for each of source code fragments in our repository. For generating AST, we used AntlrV4 [23] and its grammars. The reason behind selecting AntlrV4 is its acceptability and adaptability with various programming languages and platforms. AST is generated by traversing the Parse Tree produced by AntlrV4 related to the given code fragments. Third, we apply regular expressions on the generated ASTs to extract a metric for each of the features that we selected for detecting cross language clones. Once we represented each of the source code fragments with an AST, it became easier to track down all the features and their values with the help of regular expressions. Fourth, we extract the API calls by traversing the AST with the help of a NodeVisitor class. These API calls are then passed to the next step (Action Filter) to filter out probable non-clone pairs of source code to increase the scalability of CLCDSA.

C. Action Filter: API Call Similarity

Like other large-scale clone detection techniques [17], [24], CLCDSA also comprises a filter which helps to filter out the source code pairs which have almost zero probability of being a clone pair. Saini et al. [24] and Sajnani et al. [17] used a number of tokens from source code and their semantic similarity as parameters to filter out non-clone pairs before proceeding to the main clone detection model. Their approach is good enough to filter out non-clone pairs for a single programming language, but their approach will not work to filter out probable non-clone pairs across two different programming languages. In addition to these parameters, Saini et al. proposed an additional way to filter out non-clone pairs. They tried to see how many similar API calls occurred in a

probable clone pair. They believed that if API calls similarity is low between two code fragments, their probability of being a clone pair is also low. This hypothesis is also true for cross-language clone pairs. But since API calls of different programming languages are not semantically similar, Saini et al.'s proposed way of detecting *API calls similarity* based on their *semantics* will not be applicable here. A group of researchers recently found that API calls similarity across different programming languages could be learned and detected by their documentation similarity [29], [30]. Programming language developers and curators provide documentation of each of the APIs they added to their regular libraries[5] so that developers can understand the usage and scope of that API in the source code. The hypothesis of the two previous works is *if two APIs from two different programming languages perform the same task, their usage documentation will be semantically the same*. We also adapted this hypothesis in designing the action filter. For large scale code clone detection, this filter is very effective as regular code to code comparison for detecting clones is expensive and time consuming (increases exponentially as the number of code fragments increases).

Fig. 2 shows the example of two code blocks, developed in Java and C# respectively, which take two integer numbers as input from users, adds them, and finally shows their sum on the console. In the Java code example, the API calls used are PRINTLN() and NEXTINT(). For the C# code, the API calls are WRITE(), PARSE() and READLINE(). None of these API calls are textually similar to each other, although they provide similar functionality. From Fig. 2 we can also see the related documentation for each of the API calls. By analyzing the API call documentation we see that the documentation for PRINTLN() and WRITE() have the same semantic meaning. The same is true for other API calls. By observing this, we decided to adapt this hypothesis in designing our Action Filter. With the help of the API call documentation we detect the level of similarity among API calls used in a probable cross language clone pair. If this similarity is less than the predefined threshold value then that clone pair is rejected and considered a non-clone pair. By doing this, we reduce the number of comparisons before advancing to the main clone detection model. This helps the proposed model to execute swiftly, as well as makes it scalable for large sized clone data.

D. Feature Metrics Extraction & Similarity Detection

Metric based clone detection approaches are common and have been popular for detecting clones in single programming language based software applications [32]–[34]. For detecting cross language clones, as they are functionally similar clones (e.g., Type-4 clones), it is hard to see that metrics of the features for two source code fragments of a cross language clone pair are directly similar to each other. For this, we selected the features very carefully and finely tuned the feature metrics. In Table I the name of the features we found feasible

[5]Java https://docs.oracle.com/javase/9/docs/api/overview-summary.html
Python: https://devdocs.io/python 2.7/
C# https://docs.microsoft.com/en-us/dotnet/api/?view=netframework-4.5.2

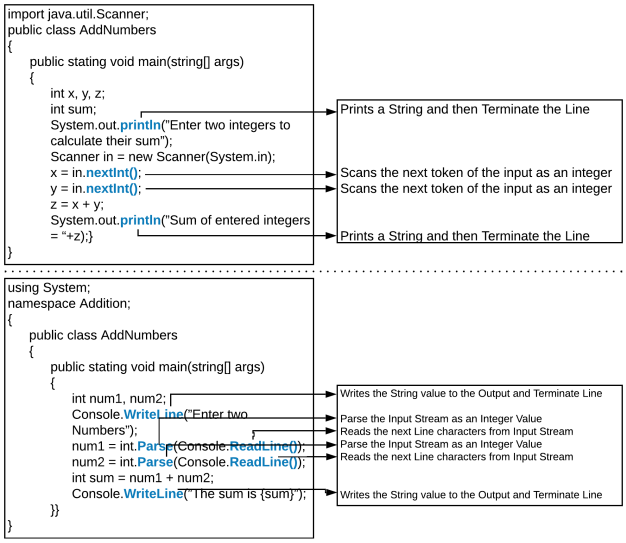

Fig. 2. Examples of API call documentation. API calls with the same functionality usually possess the same semantic description.

for detecting cross language clones have been identified in bold. By traversing the generated AST of each source code fragment in the repository, we extracted the value for each of the features listed in numeric form. We recorded those metrics in a CSV file. The reason behind saving them in a CSV file is to have a table-based record which is easy to handle.

For detecting cross language clones on the fly, this model takes a matrix consisting of 9 feature values as input. It used Cosine Similarity to detect a probable clone pair using the following equation where A and B are two matrices:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \, \|\mathbf{B}\|} \tag{1}$$

III. CLCDSA MODEL

In this section, we briefly discuss the design strategies of the model and the deep neural network it uses to detect cross language clones along with the workflow.

A. DataSet Collection

Because of the absence of an openly available cross-language clone dataset, for experimenting and evaluating our model, we created a dataset with more than 78K solutions from programming contests developed in developed in Java, C#, and Python [63]. We collected the source code for all of these solutions from three open source programming contest sites: AtCoder[6], Google CodeJam[7] and CoderByte[8]. The reason for selecting code from these websites is to ensure all the collected code, written in different programming languages, are functional clones of each other. According to the definition, cross

[6]https://atcoder.jp/
[7]https://www.go-hero.net/jam/10/languages/0
[8]https://www.coderbyte.com/challenges

TABLE II
DESCRIPTION OF DATABASE

Total number of code blocks	78,000
Total number of Questions	1,300
Total number of Java code blocks	26,000
Total number of Python code blocks	26,000
Total number of C# code blocks	26,000
Average Lines of Code Java	36∼47
Average Lines of Code Python	23∼28
Average Lines of Code C#	41∼52

language clones mean two or more code blocks developed in different programming languages that are functionally similar to each other. The open source contest sites accept solutions for a posted problem without any language restriction. At the same time, all the submitted code blocks are tested with the same input and expected output to validate them. So, we can easily say that accepted solutions for any programming language for a single problem statement are functional clones of each other, which we can claim as the best validated cross language clone database in every respect.

The present status of our dataset is given in Table II. At the time of collecting the dataset, we visited each of the archived problems and their solutions. For each of the problems, we collected at least 20 fully accepted solutions for each of the three programming languages under observation. We visited around 1,300 archived problem statements to construct our dataset. In this way, we built a complete cross-language clone dataset of 78K solutions from programming contests developed in Java, Python, and C# programming languages. In this dataset, accepted solutions for a single problem statement are validated cross-language clones of each other, whereas accepted solutions of two different problem statements are identified and validated as a cross-language non-clone pair. Among the entire dataset, we used AtCoder and Google Code Jam for training and testing the DeepNN model whereas source code collected from CoderByte was used for validating the model.

B. API Learning and Action Filter Model

For learning API call similarity across different programming languages, we first collected API documentation of the supported libraries of Java, C#, and Python. We developed a WEBCRAWLER tool to crawl through the API documentation provided by the language developers and curators. For each of the API calls, for every language, we collected only the first sentence in the belief that it carries the information regarding the API call's functionality and scope. We added the API calls documentation to an XML file. For each of the programming languages we maintained a separate API calls documentation XML file. Once the XML files were created, we traversed each of the XML files and read the collected API calls documentation manually. The purpose of this step was to verify the quality of the collected API calls documentation. If we found any non-useful API call documentation, we again searched for that API call description and manually replaced the previous documentation with a proper description. This process took us around 7 hours of manual labor on average for the three programming languages. Second, we adapted Mikolov et al.'s [19] proposed *Word2Vec* model to generate a vector representation for each API call description. We trained each API call documentation with the help of Google's word vector pre-trained model[9]. The reason behind this is to train each word of an API call documentation with greater accuracy with the help of a reasonable sized pre-trained word vector representation model. Third, with the help of Cosine Similarity stated in Equation 1 we calculated the similarity between the vector representation of two API calls' documentation selected from two different programming languages. We map each API call of one language to API calls of the other languages in a one to many manner. We also considered extracting information from the name of the API calls. But as various programming languages use different naming conventions, we found that this information is not feasible for our work. So we only considered API documentation to learn and detect similarity among two API calls of two different programming languages. We maintained a CSV file to record all of the API call similarity scores along with the name of the API calls themselves. We individually generated and maintained a CSV file for each programming language pair we considered. This is how we determined cross-language API call similarity.

Once the cross-language API call similarity is learned, the action filter is invoked, which is composed of the following three steps. First, we selected a probable clone pair from two different programming languages, retrieved their ASTs and extracted API calls. Second, we selected two API calls from two source code fragments and queried them against the language pair specific CSV file that was generated in the API learning step, to obtain the similarity score. Each API call in a single source code fragment is paired with all the API calls in the other source code fragments and queried against the appropriate CSV file to obtain the similarity score for that API pair. The API pair with the highest similarity score is kept and the others are omitted. At the same time, that API pair was removed from consideration. The same process was followed for the rest of the API calls until all the API calls of a source code fragment were paired up with those of the other source code fragments having the highest similarity score. Third, we normalized the API call similarity score by taking the average of the remaining pairs similarity score. Let us consider A and B are representing lists of API calls used in two given source code fragments. If $(A_1, B_1), (A_2, B_2), ..., (A_k, B_k)$ are API call pairs depicting the highest similarity scores then the API call similarity for a probable clone pair is calculated using the following equation to normalize the API calls similarity score:

$$\textit{APICallsSimilarity} = \frac{\sum_{i=1}^{k} (A_k, B_k)}{k} \tag{2}$$

Let us consider the clone pair example given in Fig. 2.

[9]https://github.com/mmihaltz/word2vec-GoogleNews-vectors

The Java code API calls are PRINTLN(), NEXTINT() and C# API calls are WRITE(), READLINE(), PARSE(). According to our action filter algorithm PRINTLN() was paired with each of WRITE(), READLINE(), PARSE() and the (PRINTLN(), WRITE()) pair survived with a 0.82 similarity score. After that, NEXTINT() was paired with READLINE(), PARSE() and the (NEXTINT(), READLINE()) pair remained with a 0.56 similarity score. So the API call similarity score was $(0.82 + 0.56 + 0.82 + 0.56)/4 = 0.69$. The Action Filter will not consider a source code pair as a probable clone if their API call similarity score is less than 0.5.

C. Feature Vector Formulation

Section III-A describes the dataset we considered for evaluating our model. For the CLCDSA Deep Learning model, we randomly divided our collected database for training, testing and validation purposes. First, we started working with the AtCoder dataset. For each of the questions, we selected 10 accepted solutions for each of the three programming languages: Java, C#, and Python for training purpose, 5 accepted solutions for each language for testing the model and 5 for validating the model. In this way, 20 accepted solutions for a single problem statement for each of the programming languages were divided into training, testing, and validation sets. Next, following the same process we divided the Google Code Jam dataset. We configured the feature vector with 9 metrics for each of the source code fragments in a pair with an additional boolean metric, which indicated whether this code pair was a clone or not (0 for non-clone and 1 for clone). So, the feature vector dimension was 19. In this way we generated a sample labeled dataset of 3B feature vectors, where 1.5B feature vectors represent clone pairs and the remaining 1.5B feature vectors represent non-clone pairs. The whole sample was selected randomly without any preconditions or constraints. Thus, we can reasonably claim that the model that we generated for CLCDSA represents the regular structure of source code fragments and can show the same performance with source code fragments collected from other repositories for detecting cross language clones.

D. CLCDSA Deep-Learning Model

Although various machine learning techniques exist (for example, Oreo [24]), we exploited the deep learning methodology to detect cross language clones. Deep Neural Networks or plain Neural Networks usually use a good number of intermediate layers of neurons, to automatically learn the features of a model in a non-linear transformation manner. These models are popular because of their ability of learning features and properties of universal approximation. These models can easily be adapted for large datasets. Some other areas successfully addressed by Deep Neural Networks [36] include computer vision [35] and natural language processing [19], [37].

CLCDSA's Deep-Learning Model was developed on the basis of the Siamese Architecture Neural Network [39] which works efficiently when it is required to compare two objects side by side to detect their similarity. This model can also handle the symmetry of input vectors with great care [38]. This gives the advantage of the non-requirement of maintaining a predefined order of input vectors. The source code input pairs (c_1, c_2) and (c_2, c_1) will be treated in the same manner. Siamese architecture also requires a reduced number of parameters as the weight parameters are shared between the two identical sub-neural networks.

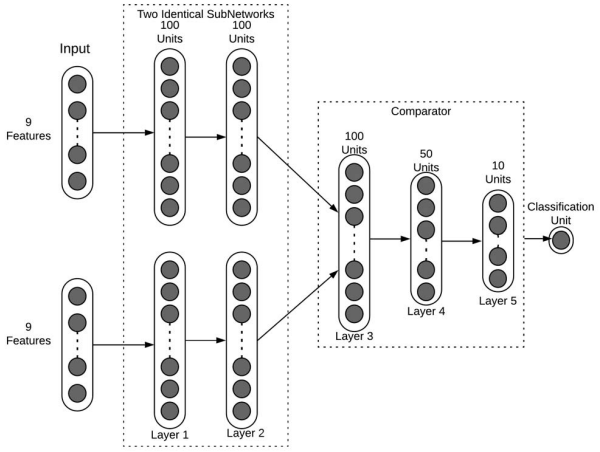

Fig. 3. Siamese Architect for CLCDSA. The model comprises of a) two identical subnetworks, b) comparator and c) classification unit

Fig. 3 shows the CLCDSA deep learning model architecture. We considered 9 dimensional feature vector for each source code as input to the model (for a pair, it is 18). These two feature vectors were then fed to two identical subnetworks, which performed the same transformation on both of the feature vectors. Both of the subnetworks had the same configuration, and had 2 fully connected hidden layers of 100 units each. Once the output of this subnetwork model was generated, two output vectors were concatenated and fed to the next comparator model. The comparator model was comprised of 3 hidden layers of sizes 100-50-10, with full connectivity among the neurons between two layers. Finally, the output of the comparator was fed to the classification unit. The classification unit was comprised of a single neuron which consists of a logistic unit having the following calculation mechanism [40]:

$$f\big(\sum_{i=1}^{10} w_i x_i\big) = \frac{1}{1 + \epsilon^{-\sum_{i=1}^{10} w_i x_i}} \tag{3}$$

Here i represents the number of units at the last layer of the comparator unit which is 10 for our experiments, x_i is the input value of that layer, and w_i is the estimated weight for that corresponding parameter. Although it is usually considered that the classification unit value greater than 0.5 is a probable clone for any learning based clone detection model [24], [42], for our evaluation, we also considered classification unit values of 0.65 and 0.8 as thresholds for accepting a code pair as clone.

We selected *ReLU* as the activation function and

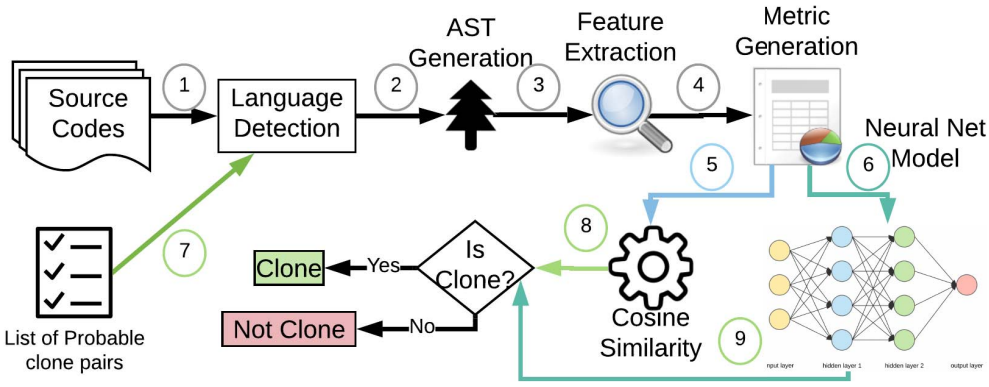

Fig. 4. CLCDSA workflow for detecting cross language clones

dropout [43] for keeping the model free from over-fit. We also selected relative entropy [44] as a loss function among the units of each layer to calculate the distance between the predicted label and the expected label. We selected the stochastic gradient descent of learning rate 0.0001 with a 3% reduction after every epoch. We considered a batch size of 3000 for each epoch, randomly initialized the values with the help of 'He normal' [45], and set the iteration value to 55 to get the final trained model.

E. CLCDSA Workflow

From Fig. 4 we can observe the working strategy of our proposed cross language clone detection model *CLCDSA*. Steps 1, 2, 3, 4 and 6 are related to the generation of the training model for CLCDSA and steps 1, 2, 3, 4 and 5 are related to the on-the-fly clone detection. In Step 1, source code from the repository is fetched into a language detection chamber. It is required as the grammar for AST generation varies from one programming language to another. Once the language is detected, in Step 3, the AST is generated and recorded. This step is repeated for each source code fragment in our repository. Once the AST generation step is completed, all the generated ASTs are fed to the feature extraction module. In this step, all the ASTs are traversed and numeric metrics for our selected features are calculated and recorded. And these metrics are preserved as the name of the source code file.

Once the metrics are generated, in Step 5, with the help of Cosine Similarity, CLCDSA detects whether the selected source code pair is a clone or not. Before Step 5 or Step 6, all the source code pairs, which are considered as probable clone pairs, are required to pass through the action filter described in Section II-C. During the training phase, since we are already providing labelled data, those feature vectors are not required to pass through the action filter. In Step 6, with the help of the labelled data, the Neural Net model has already been trained, before deploying it to detect clones.

In Step 7, CLCDSA starts clone detection phase. Once the list of the probable clone pairs are given to the model as input, with source code fragments developed in any programming language, steps 2, 3, and 4 are executed along with the action filter. After that, if on-the-fly clone detection is selected, Step 8 will occur. It will take the feature values of both the source code fragments as a matrix, and with the help of Cosine Similarity will determine whether this pair is a clone or not based on the predefined threshold value to define a clone. If the clone detection Neural Net model is preferred, the feature vector for the source code pair will then be fed to the model and Step 9 will come in effect. The output of the neural net will define whether the given source code pair of two different programming languages are clones or not based on the predefined threshold value.

IV. Evaluation

In this section, we explain the analyses on the results that we obtained from **CLCDSA**'s evaluation and validation processes. For analyzing the performance of the model we used three metrics which are popular and widely used in defining the performance of the clone detection models [17], [41], [46]. These are Precision, Recall and F-Measure.

A. Evaluation Metrics

Precision stands for the relation of correctly predicted samples to the total number of positively evaluated samples. In other words, in clone related research, precision refers to the percentage of accurately detected clones from a sample where both clones and non-clones are placed together. If N_p is the number of detected true positive clones and n_p is the number of detected false positive clones, then the precision is:

$$Precision, P = \frac{N_p}{N_p + n_p} \tag{4}$$

Recall shows to the comparative relation of correctly predicted samples to the total number of positive samples. In

clone research, the percentage of accurately detected clones from the total number of positive clones present at sample is called Recall. If N_p is the number of detected true positive clones and N_n is the number of detected false negative clones, then the recall is:

$$\textit{Recall}, R = \frac{N_p}{N_p + N_n} \tag{5}$$

F-Measure or F_1 shows the testing accuracy. It is the harmonic average of the precision and recall scores of a model validation process. If P is the calculated precision of a test system and R is the recall score of that system, then, F-measure is:

$$\textit{F-measure}, F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{6}$$

B. Research Questions

To evaluate, validate, and show the acceptability of our model we address the following 4 research questions:

- RQ1 Can CLCDSA detect cross-language clones on the fly without having any predefined knowledge and only by analyzing the given source code pair's syntactic features?
- RQ2 Can CLCDSA's DeepNN model detect cross-language clones with high accuracy?
- RQ3 Can CLCDSA outperform all the available cross-language clone detection models and methodologies in terms of Precision, Recall, and F-Measure Score?
- RQ4 Does CLCDSA support scalability in order to responsively process a large cross-language clone dataset?

C. Baseline Methods

To the best of our knowledge, there exist only three tools and models which can detect cross-language clones. Among them, Vislavski et al. [12] proposed LICCA which can detect similar source codes without being language specific. Their model fully depends on the SSQSA [18] platform, which generates a common intermediate representation of source code entities called eCST (enriched Concrete Syntax Tree). Next, Cheng et al. [1], [13] proposed CLCMiner where they tried to find out source code tokens similarity to detect cross-language clones. As their tool was not available, we replicated their model to show comparison with our work. Our reproduction results show almost the same accuracy that the authors stated in their paper. Finally, Perez in his Master thesis dissertation [41], [42] showed that generating Token Vectors from traversing AST and learning vectors using the SkipGram algorithm [47] is helpful in detecting cross language clones. Their model considered source code pairs a clone if the estimated score is greater than 0.5.

D. RQ1: CLCDSA On The Fly Performance

From Table I we see the features for which we extracted metrics from each of the source code fragments in our repository. We used Cosine Similarity as shown in Eq. 1 for detecting similarity among the feature matrices, related to the source code of probable clone pairs. From Table III we see the performance of CLCDSA in detecting cross-language clones on the fly. With considering 20% difference in feature values of two different source code fragments, we see that CLCDSA can detect clones among Java, Python, and C# programming languages with an average precision of more than 0.68, an average recall of around 0.5 along with an average F-Measure score F1 of more than 0.57. For this test case, we actually restricted the feature metrics of the two source code fragments belonging to a probable clone pair to be very similar. Even in that scenario, we saw that CLCDSA could detect cross language clones with around 60% accuracy. When we allowed more difference, around 35% in matrix values of two source code fragments, we observed that CLCDSA can successfully detect cross language clones among our selected programming languages with average precision, recall, and F1 score of 0.6, 0.65, and 0.6 respectively. From this scenario, we could see that with a small compromise with precision rate, CLCDSA would experience a high recall rate while maintaining the same accuracy in terms of an F-Measure score. In cross-language clones, based on developers choice, and differences in programming language structures and features, it is hard to get an exact match between two source code fragments from two different programming languages. So, accepting a good difference in matrix values of two probable clone candidates in a cross language environment observes a higher recall rate. With the similarity Threshold@50% we observed an average precision, recall, and F-measure score of 0.5, 0.86, and 0.62 respectively which is still acceptable.

From our results we can say that CLCDSA is capable of detecting cross language clones on the fly with great accuracy in terms of high Precision, Recall, and F-Measure scores.

TABLE III
CLCDSA'S PERFORMANCE DETECTING CROSS-LANGUAGE CLONES WITHOUT PRE-TRAINING AND ON THE FLY

	Threshold@80%			Threshold@65%			Threshold@50%		
Language Combination	Precision P	Recall R	F-Measure F1	Precision P	Recall R	F-Measure F1	Precision P	Recall R	F-Measure F1
Java & Python	0.64	0.48	0.55	0.56	0.63	0.59	0.45	0.87	0.59
Java & C#	0.78	0.56	0.65	0.67	0.72	0.69	0.57	0.91	0.7
C# & Python	0.67	0.5	0.57	0.6	0.65	0.624	0.47	0.83	0.6

TABLE IV
TESTING ACCURACY FOR CLCDSA'S DEEPNN

	Threshold@80%			Threshold@65%			Threshold@50%		
Language Combination	Precision P	Recall R	F-Measure F1	Precision P	Recall R	F-Measure F1	Precision P	Recall R	F-Measure F1
Java & Python	0.84	0.43	0.57	0.72	0.7	0.71	0.64	0.93	0.76
Java & C#	0.93	0.54	0.68	0.78	0.76	0.77	0.69	0.97	0.81
C# & Python	0.86	0.45	0.60	0.73	0.73	0.73	0.66	0.92	0.77

E. RQ2: CLCDSA's DeepNN performance

We evaluated the DeepNN model of CLCDSA, described in Section III-D, with the help of a testing and a validating dataset and performance metrics. Table IV shows the testing performance of the CLCDSA model. With the model

TABLE V
CLCDSA'S DEEPNN PERFORMANCE DETECTING CROSS-LANGUAGE CLONES

	Threshold@80%			Threshold@65%			Threshold@50%		
Language Combination	Precision P	Recall R	F-Measure F1	Precision P	Recall R	F-Measure F1	Precision P	Recall R	F-Measure F1
Java & Python	0.76	0.46	0.57	0.67	0.65	0.66	0.58	0.9	0.705
Java & C#	0.86	0.57	0.685	0.75	0.68	0.713	0.67	0.96	0.79
C# & Python	0.73	0.46	0.564	0.65	0.65	0.65	0.62	0.89	0.731

TABLE VI
PERFORMANCE OF AVAILABLE TOOLS IN DETECTING CLONES AMONG JAVA & PYTHON PROGRAMMING LANGUAGES

Tools	Precision	Recall	F1-Measure
LICCA	0.14	0.32	0.194
CLCMiner	0.09	0.13	0.11
AST Learner	0.184	0.81	0.30

[11]Since AST Learner currently only supports Java and Python, we only performed our clone detection experiment with Java and Python

configuration described in Section III-D we see that with the threshold value of a classification unit of 0.8 or greater, the model can detect cross language clones among the three programming languages under observation i.e., Java, C#, and Python for the testing dataset with an average precision of above 0.85, along with an average recall rate of 0.46, and an average F-measure score of above 0.6. From this test case, we see that with the training model that we configured we can detect cross language clones with a good accuracy even under a tight acceptance rate. Although the training model should perform more effectively in terms of recall rate as regular models do, for cross language clones, since the metrics for the features highly vary from language to language, by accepting a low error rate in model prediction, we observed a low recall rate. With similarity threshold@65%, which means accepting a classification unit value of 0.65 and above, CLCDSA could detect cross language clones with an average precision, recall, and F-measure score of 0.74, 0.72, and 0.74 respectively. From this, we can observe that despite compromising the precision a little, the recall rate increased around 23%. With the threshold value of 0.5 and above, we observed an average precision, recall, and F-measure score of 0.66, 0.95, and 0.79 respectively. Considering the observed values of the performance metrics for the test set data we can say that the CLCDSA DeepNN model is well designed. As well, training the model with our cross language clone training dataset helped the model to detect clones from the test data set with an average of 75% accuracy, for the selected programming languages.

For validating the DeepNN model, we generated a validation dataset. We took 5 accepted solutions for each of the programming languages for a single problem statement collected from two open source programming contest sites: AtCoder and Google Code Jam. Along with this dataset, we collected accepted solutions from another open source contest site: Coderbyte[10]. For a single posted problem, we collected at least 5 accepted solutions for each of the programming languages from this site in the same manner as we did for AtCoder and Google Code Jam. In this way, we built our dataset for validating the DeepNN model. From this dataset, we randomly selected 200 questions and their solutions for validation purpose. We also randomly selected the probable clone pairs from this dataset. From Table V we can see CLCDSA's DeepNN model's performance in detecting cross language clones for our validating dataset. Comparing it with that of Table IV we see that we did not loss much accuracy in detecting cross language clones for the testing and validating dataset. With threshold@80%, which means accepting a classification unit value of 0.8 or more as clones, for the validation dataset, DeepNN of CLCDSA detected clones with an average precision of 0.80, with an average recall rate of 0.48, along with an average F-measure score of 0.59. From this we can say that we observed a little bit of a reduced precision rate for validating data compared to test data. However, it was acceptable because reducing the threshold value to 0.65, we observed around 8% or less reduction in average precision rate, but got an increment of around 20% in average recall rate. These results are still less compared to the test data, but their differences were very little and we could easily accept this, since the validated data was completely new to the model. Finally, by accepting a classification unit value of 0.5 or more as clones, we experienced that CLCDSA could detect cross-language clones with an average precision of 0.62, an average recall 0.9, and an average F-Measure 0.71. This indicates that CLCDSA could detect clones with around 70% accuracy.

From the above discussion we can say that CLCDSA DeepNN detects cross language clones with greater accuracy than CLCDSA On-the-Fly. This model and the experimental data along with the experimental results available for others [63].

F. RQ3: Performance of Available Models

As discussed in Section IV-C we found that at present three tools are available which could detect cross language clones. Among them, CLCMiner was designed for a different purpose and was not designed to detect plain cross language functional clones (two source codes written in different programming languages performing same operation). From the Table VI we can also see the poor performance of this model. Each of the average precision and recall rates we observed for CLCMiner was on average below 0.15. In addition to that, LICCA also performed poorly for our clone dataset. The reason behind LICCA's poor performance is the set of the restrictions it possesses. For detecting clones, LICCA needs the code blocks size to be the same and requires the same control flow and sequences along with the same flow of statements. In regular cross language clone data, these preconditions are hard to meet because of the nature and structure of different programming languages. From our observation and the description provided

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

TABLE VII
PERFORMANCE OF ACTION FILTER IN REDUCING TOTAL COMPARISONS

Comparison Style	Action Filter Status	Number of Comparisons	Action Filter Status	Number of Comparisons
Total Dataset	No Filter	5.8077e+838	Filter Active	2.84253e+30
Java & Python	No Filter	3.87259e+279	Filter Active	2.84217e+30
Java & C#	No Filter	3.87259e+279	Filter Active	3.7252e+19
C# & Python	No Filter	3.87259e+279	Filter Active	3.63798e+26

in their paper, we found that LICCA could detect cross-language clones with an average precision, recall, and F-measure score of 0.14, 0.32, and 0.20 respectively. Finally, we experimented with the state of the art Cross Language clone detection model *AST Learner*. With the acceptance of prediction value score of 0.5 and up, *AST Learner* could detect cross language clones with around 0.18 average precision and 0.81 average recall. This helped them achieve an F-Measure score of 0.30. Comparing the experimental results described in Table VI with Table III and Table V, CLCDSA outperforms all available tools and methodologies for detecting cross language clones by a high margin in terms of precision, recall, and F-Measure.

G. RQ4: Scalability

To evaluate the performance of the action filter that we proposed for CLCDSA we compared the total number of estimated comparisons among the given source code and the actual comparison performed at the time of validating the model. For this experiment, we selected the dataset that we configured for validating the CLCDSA DeepNN model. The dataset consists of 200 problem statements where for each problem statement we selected 5 accepted solutions for each of the three programming languages. From Table VII we see the comparative performance of a number of source code comparisons with the absence and presence of the action filter. From the data in Table VII, we see that our proposed action filter successfully reduced the number of source code comparisons by a significant amount, which illustrates the scalability and capability of our model in handling large cross language clone datasets.

V. THREATS TO VALIDITY

We consider a number of threats to the validity of our study. One threat concerns our collected dataset and its validation. We performed our evaluation and validation process with a collected dataset of 78K source code written in Java, C#, and Python to solve programming contest problems. We collected the dataset from open source contest sites where submitted solutions for a single problem are tested with the same input and the expected output. We only collected the source code which was accepted with full scores. Solutions of one single problem statement are considered clones to each other and solutions of different problem statements are considered non-clones. Based on these considerations, we claim that the dataset is a validated source of cross language code clones and that CLCDSA will be able to show similar performance in general, and with other cross-language clone datasets.

Another probable threat could be the quality of the API call documentation used by our Action Filter. We collected documentation from the programming language curators' websites. The documentation has been used and considered of good quality in previous studies. Our consideration of using a similar approach as of previousl studies helps mitigate the threat to validity of our study.

Finally, the selection of DeepNN for our feature vector learning model could be questionable too. We selected this model based on the model selection process described in Oreo [24]. DeepNN has already proved to be applicable to work for similar kinds of problems and usage scenarios. So, we claim that we have selected a reasonable model to learn our feature vector. In the future, we would like to perform a further study to verify whether DeepNN is the best choice for our proposed model or not.

VI. RELATED WORK

Detecting clones across projects is not a new concept. Considerable research and tools are available to detect clones between and within projects. Single language clones can be detected using source code text similarity (e.g., Duploc [48], Marcus et al. [49], and NiCad [16]), code-token based similarity (e.g., CCFinderX [14], Clone Detective [50], iClones [51], and SourcererCC [17]), AST-based similarity detection techniques (CloneDr [52], Deckard [15], CloneDigger [53], and SimScan [54]), metrics-based techniques [34], [55], clones in binary executables [60], and Program Dependency Graph (PDG) based clone detection techniques [56], [57]. Although progress has been made in detecting clones for a single language, a significant gap still exists in detecting clones across different programming languages.

One of the very first works in cross language clone detection was proposed by Kraft et al. [58] where they leveraged CodeDOM as an intermediate representation. Their model could detect clones across programming languages developed by Microsoft .Net framework that usually used the same intermediate representations, e.g. C#, ASP.Net, Visual Basic, and so on. Their model is not able to directly detect clones across different programming languages such as clones between Java and C# or Java and Python (where intermediate states are totally different, at the same time supported platforms and programming language maintainers are different too). Another very closely related work was by Al-Omari et al. [59]. They detected clones in .Net programming languages by analyzing the generated Common Intermediate Language (CIL).

Vislavski et al. [12] proposed LICCA to detect cross language clones. They leveraged the SSQSA [18] platform's capability of generating common intermediate representations for various programming languages. Although it is a sound approach for detecting cross language clones, their model suffers from a number of limitations such as failing to detect cross language clones for non-equal length source codes, non-similar control flow of data and non-similar usage pattern of APIs' in different source codes and so on (discussed in Section I). These limitations restrict their model from being

usable for real-world cross language clone detection. Cheng et al. [1] proposed CLCMiner to detect code changes for a 3rd party tool's API calls across different programming languages. They use a word-token matching strategy to track code changes across different revision histories. Their approach fails to detect similar functional code blocks across different programming languages.

Very recently an AST learning based approach was proposed by Perez et al. [41], [42] which can learn AST representations across different programming languages. Their model is quite capable of detecting code clones between Java and Python. However, their model has a very low precision rate even though their model accepts 50% and up similar AST blocks as code clones. Supervised [24], [62] and unsupervised [61] deep learning based clone detection models are have been proposed for detecting clones for a single programming language. However, the approaches used are not suitable for detecting cross language clones.

Learning similar APIs across different programming languages has already been shown to be capable of detecting similar software applications [29] and finding similar cross language libraries [30]. Currently the performance of these models is acceptable, however, more research is required to develop a highly accurate similar API calls mapping technique that could easily be adaptable in a cross language software development environment.

VII. Conclusion

While there are many clone detection tools available for detecting clones in software systems written in a single programming language, there are only a few approaches for detecting clones in software systems implemented in different programming languages. Given that cross language clones are likely to have more varieties than single language clones, and that they may pose additional challenges in software maintenance and evolution, it is important to detect cross language clones. In this paper, we proposed *CLCDSA* which can detect cross language clones with greater accuracy compared to all other available models in terms of Precision, Recall, and F-Measure. CLCDSA can detect clones both on the fly, and with the help of a deepNN network. In the future we plan to study its suitability for other cross language based source code analysis.

Acknowledgment

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

References

[1] 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.

[2] C. Wagner, *ModelDriven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*, Springer Science & Business Media, 2014.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: yesterday, today, and tomorrow*, In Present and ulterior software engineering, pp. 195-216. Springer, Cham, 2017.

[4] Antlr3 http://www.antlr3.org/

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

[6] Factual Developers http://developer.factual.com/

[7] 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 74.7 (2009): 470-495.

[8] C. Kapser and M. Godfrey, *Supporting the Analysis of Clones in Software Systems: A Case Study*, JSME: Research and Practice, 18(2):61-82, 2006.

[9] A. Sheneamer and J. Kalita, *Article: A Survey of Software Clone Detection Techniques*, International Journal of Computer Applications 137.10 (2016).

[10] R. Fanta and V. Rajlich, *Removing clones from the code*, Journal of Software Maintenance 11, 4 (July 1999), 223-243.

[11] C. Kapser and M. W. Godfrey, *"Cloning Considered Harmful" Considered Harmful*, 13th Working Conference on Reverse Engineering (WCRE'16), Benevento, 2006, pp. 19-28.

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

[13] X. CHENG, Z. PENG, L. JIANG, Z. Lingxiao, H. YU, and J. ZHAO, *CLCMiner: Detecting cross language clones without intermediates*, IEICE Transactions on Information and Systems. E100-D, (2), 273-284.

[14] T. Kamiya, S. Kusumoto and K. Inoue, *CCFinder: a multilinguistic token-based code clone detection system for large scale source code*, in IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, July 2002.

[15] L. Jiang, G. Misherghi, Z. Su and S. Glondu, *DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones,* 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, 2007, pp. 96-105.

[16] J. R. Cordy and C. K. Roy, *The NiCad Clone Detector,* 2011 IEEE 19th International Conference on Program Comprehension, Kingston, ON, 2011, pp. 219-220.

[17] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, *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.

[18] Z. Budimac, G. Raki, and M. Savi, *SSQSA architecture*, In Proceedings of the Fifth Balkan Conference in Informatics (BCI '12). ACM, New York, NY, USA, 287-290.

[19] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, In Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13), 3111-3119.

[20] H. Anton, *Elementary Linear Algebra*, 7th edition, 1994, John Wiley & Sons, pp. 170171,

[21] M. Young, *The Technical Writer's Handbook*, Mill Valley, CA: University Science, 1989.

[22] Abstract Syntax Tree (AST) https://en.wikipedia.org/wiki/Abstract_syntax_tree

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

[24] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, *Oreo: detection of clones in the twilight zone.*, In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). ACM, New York, NY, USA, 354-365.

[25] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, *The SQO-OSS quality model: measurement based open source software evaluation*, In Proceedings of the International Conference on Open Source Systems. 237248, 2008.

[26] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, *Recommending source code for use in rapid software prototypes*, In Proceedings of the 34th International Conference on Software Engineering (pp. 848-858). IEEE Press.

[27] C. Sadowski, K. T. Stolee, and S. Elbaum.*How developers search for code: a case study.* In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 191-201.

[28] A. Monden, D. Nakae, T. Kamiya, S. Sato and K. Matsumoto, *Software quality analysis by code clones in industrial legacy software,* Proceedings Eighth IEEE Symposium on Software Metrics, Ottawa, Ontario, Canada, 2002, pp. 87-94.

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

[30] C. Chen, S. Gao and Z. Xing, *Mining Analogical Libraries in Q&A Discussions – Incorporating Relational and Categorical Knowledge into Word Embedding*, 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, 2016, pp. 338-348.

[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, MIT press, 2016.

[32] K. Kontogiannis, *Evaluation experiments on the detection of programming patterns using software metrics*, In Proceedings of the Fourth Working Conference on Reverse Engineering. IEEE, 4454.

[33] J. Mayrand, C. Leblanc, and E. Merlo, *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics*, In Proceedings of International Conference on Software Maintenance. 244

[34] J. F. Patenaude, E. Merlo, M. Dagenais, and B. Lagu, *Extending software quality assessment techniques to java systems*, In Proceedings of Seventh International Workshop on Program Comprehension. IEEE, 4956.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, In Advances in Neural Information Processing Systems 25. 10971105.

[36] J. Schmidhuber, *Deep learning in neural networks: An overview*, Neural Networks 61 (2015), 85117.

[37] R. Socher, Y. Bengio, and C. D. Manning, *Deep learning for NLP (without magic)*, In Tutorial Abstracts of ACL 2012. Association for Computational Linguistics, 55

[38] G. Montavon and K.-R. Mller, *Better representations: Invariant, disentangled and reusable.* In Neural Networks: Tricks of the Trade. Springer, 559560.

[39] P. Baldi and Y. Chauvin, *Neural networks for fingerprint recognition*, Neural Computation 5, 3 (1993), 402418.

[40] X. Glorot, A. Bordes, and Y. Bengio, *Deep sparse rectifier neural networks*, In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. 315323.

[41] Daniel Perez, *A study of machine learning approaches to cross-language code clone detection*, Master Thesis dissertation

[42] D. Perez and S. Chiba, *Cross-language clone detection by learning over abstract syntax trees*, in proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (to Appear)

[43] P. Baldi and P. Sadowski, *The dropout learning algorithm*, Artificial intelligence 210 (2014), 78122.

[44] S. Kullback and R. A. Leibler, *On information and sufficiency*, The annals of mathematical statistics 22, 1 (1951), 7986.

[45] 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, 10261034.

[46] 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

[47] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, In Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13), pp 3111-3119.

[48] S. Ducasse, M. Rieger and S. Demeyer *A Language Independent Approach for Detecting Duplicated Code,* in Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, pp. 109-118 (1999).

[49] A. Marcus and J. Maletic, *Identification of High-level Concept Clones in Source Code,* in Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, pp. 107-114 (2001).

[50] Tool Clone Detective (part of ConQAT). URL http://conqat.in.tum.de/index.php/Main Page Last accessed November 2008.

[51] N. Gode, *Incremental Clone Detection,* Diploma Thesis, Department of Mathematics and Computer Science, University of Bremen, Germany, 2008.

[52] I. Baxter, A. Yahin, L. Moura and M. Anna, *Clone Detection Using Abstract Syntax Trees,* in Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, pp. 368-377 (1998).

[53] P. Bulychev and M. Minea, *Duplicate Code Detection Using Anti-Unification,* in Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008, 4 pp. (2008).

[54] Tool SimScan, URL http://www.blue-edge.bg/simscan/ Last accessed November 2008.

[55] G. Di Lucca, M. Penta and A. Fasolino, *An Approach to Identify Duplicated Web Pages,* in Proceedings of the 26th International Computer Software and Applications Conference, COMPSAC 2002, pp. 481-486 (2002).

[56] J. Krinke,*Identifying Similar Code with Program Dependence Graphs,* in: Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, pp. 301-309 (2001).

[57] C. Liu, C. Chen, J. Han and P. Yu, *GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis,* in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006, pp. 872-881 (2006).

[58] N. A. Kraft, B. W. Bonds, and R. K. Smith, *Cross-language Clone Detection*, In Proceedings of SEKE, pp. 54-59. 2008.

[59] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, *Detecting clones across microsoft. net programming languages*, In proceedings of 19th Working Conference on Reverse Engineering, pp. 405-414. IEEE, 2012.

[60] A. Sbjrnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, *Detecting code clones in binary executables*, In Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09). ACM, New York, NY, USA, 117-128. 2009 DOI: https://doi.org/10.1145/1572272.1572287

[61] H. -HuiWei and M. Li, *Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code*, In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17). 30343040.

[62] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, *Deep learning code fragments for code clone detection*, In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 8798.

[63] CLCDSA working version: https://github.com/Kawser-nerd/CLCDSA