ISTANBUL TECHNICAL UNIVERSITY★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

CODE CLONE DETECTION WITH CONVOLUTIONAL NEURAL NETWORK

M.Sc. THESIS

Harun DİŞLİ

Department of Computer Enginnering

Computer Enginnering Programme

JUNE 2019

İSTANBUL TECHNICAL UNIVERSITY★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

CODE CLONE DETECTION WITH CONVOLUTIONAL NEURAL NETWORK

M.Sc. THESIS

Harun DİŞLİ
(504151511)

Department of Computer Enginnering

Computer Enginnering Programme

Thesis Advisor: Dr. Ayşe TOSUN

JUNE 2019

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

KONVOLÜSYONEL SİNİR AĞI İLE KOD KLON TESPİTİ

YÜKSEK LİSANS TEZİ

Harun DİŞLİ
(504151511)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Dr. Ayşe TOSUN

HAZİRAN 2019

Harun Dişli, a M.Sc. student of ITU Graduate School of Science Engineering and Technology 504151511 successfully defended the thesis entitled "CODE CLONE DETECTION WITH CONVOLUTIONAL NEURAL NETWORK", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below

Thesis Advisor:	**Dr. Ayşe TOSUN** Istanbul Technical University	.....
Jury Members :	**Doç. Dr. Tolga OVATMAN** Istanbul Technical University	.....
	Doç. Dr. Gülfem IŞIKLAR ALPTEKİN Galatasaray University	.....

Date of Submission : 3 May 2019
Date of Deffense : 13 June 2019

To my family,

FOREWORD

In this master's thesis study carried out within the ITU Graduate School Of Science Engineering And Technology, a code clone detection approach has been proposed.

I would like to thank to my advisor, Dr. Ayşe TOSUN, for all guidance and motivations during my graduate studies.

Lastly, I would like to thank to my family for their patiance and support throughout all of my education life.

June 2019

Harun DİŞLİ
Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

A : Accuracy
AST : Abstract Syntax Tree
CF : Code Fragment
CNN : Convolutional Neural Network
DNN : Deep Neural Network
F : False Alarm Rate
kNN : k-Nearest Neighbor
LSTM : Long Short-Term Memory Network
MT3 : Moderately Type 3 Clone
P : Precision
PDG : Program Dependency Graph
R : Recall
ReLU : Rectifier Linear Unit
RNN : Recurrent Neural Network
ST3 : Strong Type 3 Clone
T1 : Type 1 Clone
T2 : Type 2 Clone
VST3 : Very Strong Type 3 Clone
WT3/4 : Weak Type 3 or 4 Clone

LIST OF TABLES

Page

LIST OF FIGURES

Page

CODE CLONE DETECTION WITH CONVULUTIONAL NEURAL NETWORK

SUMMARY

Code clones are similar or identical code portions produced by duplicating existing source code for resusing in another part of the software system. Code clones are one of the main reasons of the bug propogation, and thus high maintenance cost. When a code piece is duplicated, software defects and bugs are also duplicated. Therefore, a developer should check for bugs in all copies to avoid unexpected failures in the system. If the software consists of a large number of code clones, maintenance costs may be dramatically high. For that reason, code clone detection is crutial in order to reduce both maintenance costs and crashes in a software system.
In code clone detection, a group of code lines are named as code fragment (CF) and illustrated with file name, begin line and end line. A CF can include comments and white space. If two code fragments similar or identical, they are named as code clones. There are four type of code clones: Type-1, Type-2, Type-3, Type-4. Type-1 clones are identical code fragments except for white space and comments. In Type-2 clones, identifier names such as variables can be different. If a code fragment different sfrom another fragment by added, removed or modified line, they Type-3 clones. Type-4 clones are fragments performing same computation with different syntactic.

Code cloning detection includes five steps: pre-processing, transformation, match detection, formatting and post-processing/filtering In the pre-processing phase, the irrelevant sections are removed and the level of clone detection is determined. In the conversion phase, the source code is converted to tokens, tree structures or graphic models, and comments and gaps are removed. After the conversion phase is complete, the resulting output is given to any comparison algorithm. The matching algorithm is used to identify the matching parts. The output of the match detection step is the match list in the converted code. Clones detected in the match phase are returned to their original form in the formatting phase and clone detection is completed. In post-processing/filtering, the detected clones are filtered manually or using automated methods.

Several studies have been conducted using different approaches for the detection of cloned code fragments. However, very few of them take advantage of popular learning based approaches, such as deep learning. This thesis proposes a token-based technique that converts the source code into image data format and detects code clones using a convolutional neural network (CNN). We divided the candidate clone pairs into their tokens and calculated how many times each token passed in the code separately for both pairs. We recorded calculated values as frequency matrices, so we obtain separately calculated frequency matrices for both pairs. Then we combined these two matrices and save them as image data in jpeg format. Finally, we

trained CNN using these images representing clone pairs and classify candidate clone pairs as “clone” and “not clone”. Since the merged matrix is very similar to image data, we considered it appropriate to use this matrix as image data. We chose CNN because CNN assumes inputs as images and provides attributes encoding architecture. This feature helps reduce the number of parameters and more efficient calculations.

To implement this approach, we used candidate clone pairs extracted from the BigCloneBench data set. BigCloneBench is a set of six million validated clones in java projects. In this data set, Type-3 and Type-4 clone types were re-interpreted according to their similarity ratio and a total of six clone types were obtained.

We conducted experimental studies on our approach using three different training sets from the BigCloneBench data set. First of all, we took 50 thousand samples in total based on the rates in the dataset. As 94.63% of the original data set consisted of WT3/4 clones, the training set we obtained consisted of clones of WT3/4 type at the same rate. Second, we tried the microsampling methodology. Accordingly, we received 2000 samples for each type since there are 2083 VST3 which has minimum number of elements in BigCloneBench. The final training set was composed of random samples with a total of 50 thousand. We prepared five test sets for each training set and each test set consisted of 10 thousand samples. When preparing the test sets, we adhered to the proportions of the original data to reflect the actual scenario.

Our experimental studies with CNN using three different training-test sets show that our approach has a good performance in detecting code clones, especially complex clone types.

KONVOLÜSYONEL SİNİR AĞI İLE KOD KLON TESPİTİ

ÖZET

Kaynak kod içindeki kod parçalarının kopyalanması ve yeniden kullanılmasıyla oluşturulan benzer veya aynı kod parçaları, kod klonları olarak adlandırılır. Kod klonları, genellikle programcılar tarafından kopyala-yapıştır işlemlerinin bir sonucu olarak ortaya çıkar. Bir kod parçasında hatalar mevcutsa, aynı hatalar büyük olasılıkla diğer kopyalarda da mevcuttur. Bu sebeple, birden fazla kopyası bulunan bir kod parçasının güncellemesi gerektiğinde, geliştirici sistemdeki beklenmedik çökmeleri önlemek için diğer bütün kopyaları kontrol etmek zorundadır. Çok sayıda klon içeren ve çok sayıda programcı tarafından geliştirilen bir yazılım sisteminde, bu kopyalanan kod bölümleri nedeniyle bakım maliyeti ciddi şekilde yüksek olabilir. Bu nedenle bir yazılım sistemindeki kod klonlarını tespit etmek, hem önemli bakım maliyetlerini hem de klon işlemleriyle ilişkili olası hata risklerini azaltmak için çok önemlidir.

Herhangi bir kod satırı dizisi kod parçası (CF) olarak adlandırılır. Kod parçaları yorum satırı içerebilirler. Bir CF dosya adı, başlangıç satırı ve bitiş satırı oluşan üçlüler ile tanımlanır. Eğer iki kod parçası benzer yada birebir aynı ise bu kod parçaları kod klon olarak adlandırılır. Matematiksel olarak f(CF1) = f(CF2) şeklinde gösterilir (f benzerlik fonksiyonudur).Klon tipleri, program metinlerinin benzerliğine ve metinlerinden bağımsız olarak işlevselliklerinin benzerliğine dayanan iki ana tür olarak gruplandırılabilir. Metinsel klon tipleri Tip-1, Tip-2 ve Tip-3 olarak ayrılırken, işlevsel klon tipleri Tip-4 olarak bilinir. İki kod parçası, yorum veya boşluk dışında, birbirinin aynı ise Tip-1 klon olarak adlandırılırlar. İki kod parçası, sadece tanımlayıcılar, değişmezler, türler, boşluk ve yorumlar ile farklılaşmışsa, bunlar Tip-2 klonlardır. Eğer bir kod parçası, ifade ekleme-çıkarma ya da ifade değiştirme ile diğer bir kod parçasından elde edilmişse, bu iki kod parçası Tip-3 klonlardır. Eğer iki kod parçası tamamen farklı şekilde yazılmış olmasına rağmen aynı işi yapıyorsa, bu kod parçaları Tip-4 klon olarak bilinirler.

Kod klon tespiti genel olarak beş adımdan oluşur: Ön işleme, dönüştürme, eşleşme tespiti, biçimlendirme ve işlem sonrası / filtreleme. Ön işleme aşamasında, kod ile alakasız kısımlar çıkarılıp, klon tespitinin hangi seviyede yapılacağı belirlenir. Dönüştürme aşamasında ise kaynak kod tokenlara, ağaç yapılarına veya grafik modellere dönüştürülüp yorum ve boşluklar kaldırılır. Dönüştürme aşaması tamamlandıktan sonra elde edilen çıktı herhangi bir karşılaştırma algoritmasına verilir. Karşılaştırma algoritması ile birbiriyle eşleşen bölümler tespit edilir. Eşleşme tespiti aşamasının çıktısı dönüştürülen koddaki eşleşme listesidir. Eşleşme aşamasında tespit edilen klonlar, biçimlendirme aşamasında orijinal koddaki hallerine döndürülerek klon tespiti tamamlanır. İşlem sonrası / filtreleme aşaması ise tespit edilen klonlar manuel olarak ya da otomatik yöntemler kullanarak filtrelenir.

Klonlanmış kod parçalarının tespiti için farklı yaklaşımlar kullanan çeşitli çalışmalar yapılmıştır. Çalışmalarda önerilen yaklaşımlardan bazıları metne dayalı teknikler gerçekleştirirken, bazıları token tabanlı teknikler, diğerleri ise anlamsal veya ağaç

bazlı teknikler kullanmaktadır. Metin tabanlı teknikler, kaynak kodunu kelimelerden oluşan dizeler olarak ele alır ve benzerliklerini tanımlamak için dizeleri birbirleriyle karşılaştırırlar. Diğer yaklaşımlar, klonlanmış parçaları tespit etmek için bir algoritma uygulamadan önce kaynak kodunu tokenlara veya grafik modellere dönüştürür. Kod klonu tespit modelini oluşturmak için, suffix tree, dotplot/scatter plot, hash-değeri karşılaştırması ve Öklid mesafesi gibi algoritmalar denenmiştir. Son zamanlarda, araştırmalar RNN (tekrarlayan sinir ağı) ve LSTM (uzun kısa süreli hafıza ağı) gibi derin öğrenme tekniklerin kaynak kodları modellemede kullanıldığını göstermektedir. Kaynak kodun modellenmesini gerektiren özel bir alan da kod klonu tespitidir. DNN (derin sinir ağı) gibi derin öğrenme teknikleri kullanılarak yapılan klon saptama çalışmaları, Tip-1 ve Tip-2 klonlarının en az% 98'inin tahmin edilebileceğini bildirmiştir.

Çeşitli kaynaklar tarafından Tip-3 klonun en çok bulunan klon tipi olduğu belirtilmiştir. Ancak, Tip-3 klonlar Tip-4 klonlar ile birlikte saptanması en zor klon tipleridir. Derin öğrenme kullanarak bile, CClearner zayıf Tip-3 ve Tip-4 klonlarında maksimum% 28'lik bir tespit oranına ulaşmıştır. Bu nedenle, bütün klon tiplerini yüksek doğruluk oranıyla saptamak için kod klon tespiti konusunda daha fazla araştırma yapılması gerekmektedir.

Bu çalışma, kaynak kodunu görüntü verisi biçimine dönüştürerek ve konvolüsyonel sinir ağı (CNN) kullanarak kod klonu tespiti yapan bir token-based teknik önermektedir. Bu çalışmada, aday klon çiftlerini tokenlarına ayırdık ve her bir tokenın kod içerisinde kaç defa geçtiğini çiftlerin her ikisi için de ayrı ayrı hesapladık. Hesaplanan değerleri frequency matrisleri olarak kaydettik. Böylece elimizde çiftlerin her ikisi için de ayrı ayrı hesaplanış matrisler oldu. Daha sonra bu iki matris birleştirilerek görüntü verisi olarak jpeg formatında kaydettik. Son olarak, CNN'i klon çiftlerini temsil eden bu görüntüler kullanılarak eğittik ve aday çiftleri "klon" ve "klon değil" olarak sınıflandırdık. Birleştirilen matris, görüntü verisine çok benzer olduğu için, bu matrisi görüntü verisiymişçesine kullanmayı uygun gördük. CNN'in girdileri görüntü olarak varsaydığı ve öznitelikleri mimariye kodlama özelliği sağladığı için CNN'i tercih ettik. Bu özellik, parametre sayısının azaltılmasına ve daha verimli uygulama yapılmasına yardımcı olmaktadır.

Bu yaklaşımı uygulamak için, BigCloneBench veri kümesinden çıkarılan aday klon çiftlerini kullandık. BigCloneBench büyük java projelerindeki altı milyon doğrulanmış klondan oluşan bir veri kümesidir. Bu veri kümesinde, Tip-3 ve Tip-4 klon tipleri aralarındaki benzerlik oranına göre yeniden yorumlanmış ve toplamda altı adet klon tipi elde edilmiştir: T1 (Tip-1), T2 (Tip-2), VST3 (Çok Güçlü Tip-3), ST3 (Güçlü Tip-3), MT3 (Orta Tip-3) ve WT3/4 (Zayıf Tip-3 veya Tip-4). İki klon adayı arasındaki benzerlik oranı 0.9'dan daha büyükse, VST3 olarak etiketlenirler. Benzerlik oranı 0,7 ile 0,9 arasındaysa, bunlar ST3 klonlardır. Oran 0,7'den düşük ve 0,5'ten yüksek olduğunda, klon tipi MT3 olur. Benzerlik 0,5'in altında ise, klon tipi WT3/4 olarak adlandırılır.

Yaklaşımımız üzerinde, BigCloneBench veri setinden elde edilen üç farklı eğitim seti kullanarak deneysel çalışmalar yaptık. İlk olarak veri setindeki oranları baz alarak toplamda 50 bin adet örnek aldık. Orijinal veri setininin %94.63'ü WT3/4 tipindeki klonlardan oluştuğundan, elde ettiğimiz eğitim seti de aynı oranda WT3/4 tipindeki klonlardan oluştu. İkinci olarak microsampling yöntemini denedik. Buna göre veri setindeki en az sayıda bulunan sınıfa eşit sayıda her sınıftan örnekler aldık. Verisetinde en az sayıda örneği bulunan sınıf VST3 klon tipidir ve 2083 adettir. Bu yüzden her bir klon tipinden 2000 adet örnekler aldık. Son eğitim setini ise toplamda 50 bin tane olacak şekilde rastgele örnekler alarak oluşturduk. Her bir eğitim seti için

beş farklı test seti hazırladık ve her bir test setimiz 10 bin örnekten oluştu. Test setlerini hazırken, gerçek senaryoyu yansıtması için orijinal verisetindeki oranlara bağlı kaldık.
Üç farklı eğitim-test seti kullanılarak CNN ile yapılan deneysel çalışmalarımız, yaklaşımımızın kod klonlarını, özellikle de karmaşık klon türlerini tespit etmede iyi bir performansa sahip olduğunu göstermektedir.

1. INTRODUCTION

In a software system, there may be some code portions, which are identical or similar to one another. These code portions are named as code clones and most of the time; they are the results of copy-paste activities of software engineers. When an engineer duplicates a code piece, she also duplicates residual bugs of that code piece. For that reason, both original and duplicated code pieces have the same problems needed to be fixed [26]. Accordingly, the point at which an update is required in a code piece that has multiple duplicates in the source code, the engineer should check all of the duplicates to keep the system away from unexpected accidents. In a software system in which development is made by large software group, it is highly possible that the software contains high number of clones. Support and maintenance costs might be truly high in such a software system [1]. Thus, because of diminishing in both the critical support costs and the dangers of potential crashes related with clone operations, code clone detection is crucial.

There have been several research studies on code clone detection employing different approaches to identify cloned code pieces in large scale software applications. Some of the approaches proposed in the studies perform text-based techniques [2,3,4], while others use token-based techniques [5,6,7,8], semantic or tree-based techniques [9,10]. Text-based techniques treat source code as strings consisting of words and compare strings among each other to identify their similarities [17]. The other approaches transform the source code into tokens, or graphical models before applying an algorithm to detect cloned pieces. To build the code clone detection model, algorithms such as suffix tree [6,27], dotplot/scatter plot [28], hash-value comparison [9,10] and Euclidean distance [29] have been tried. Recently, studies show that deep learning techniques such as RNN (recurrent neural network) [30] and LSTM (long short-term memory network) [31] are used for modeling source codes. One particular field that requires modeling the source code is code clone detection. Studies on clone detection using deep learning techniques, such as DNN (deep neural network) [16] report that at least 98% of Type-1 and Type-2 clones can be predicted.

Although Type-3 clones are the most common clone types in source code [32,33], previous works could not detect these clone type satisfyingly and Type-3 clones have the lowest detection rate with Type-4 clones: Even with deep learning, CClearner[16] has achieved only 28% detection rate in weakly Type-3 and Type-4. Therefore, for detecting all types of code clones with high accuracy, code clone detection needs further invastigating.

This thesis proposes a token-based technique for detection code clones. It converts code pieces to image data and train convolutional neural network (CNN) in order to predict whether given two code pieces are clone or not. We used BigCloneBench dataset [13] to extract candidate clone pair methods. Firstly, we saperated each method to its token and calculated number of occurences of each token. Number of occurences for each method were strored as frequecy matrices. Later, these two matrices are merged into a single matrix and saved as jpeg image format. Because the merged matrix looks like an image data and CNN accepts inputs in the form of image data, we used merged feature matrix as an image. We prefer CNN because it helps reduce the number of parameters and makes more efficient implementation by encoding features to the architecture [12]. We made empirical analysis on three different train-test splits to see our clone detection performance with CNN. Our experimantal results show that code clone detection with CNN has good performance even on sophisticated clone types.

1.1 Purpose of Thesis

The purpose of this thesis is to develop a novel code clone detection approach taking advantage of a deep learning technique in order to detect both simple and complicated clone types. Type-1 and Type-2 clones can be easily detected by clone detection tools. However, although there are mostly Type-3 clones in source code, Type-3 and Type-4 clones are the lowest detection rate. Thus, detecting these complicated clone types is still an open research area. Recent studies show that source code can be modeled thanks to deep learning techniques. Regarding the success of deep learning on source code modeling, we used deep learning to detect code clones. In additon, because of similarity between our feature matrix and image data, we think CNN will be the best fit.

1.2 Literature Review

Code clone detection and plagiarism detection are two different research areas that apply similar approaches. Code clone detection performs operations on the source code in the same software system, while plagiarism detection compares source codes from two or more different systems. In this section, some plagiarism detection and code clone detection tools are introduced. Table 1.1 summarizes the studies described in this section.

1.2.1 Source Code Plagiarism Detection Tools

JPlag [23] finds similar programs within a given set of programs. It works as a webservice and supports Java, Scheme, C, or C++ languages. In JPlag comparison algorithm, all input programs are parsed and transformed into token strings. Gredy String Tiling algorithm is used to decide similarity between obtained token strings. Plaggie [24] is another tool that applies similar functionality. Unlike JPlag, it is open source and desktop application.

Bandara et al. [34] utilizes machine learning algorithms for plagiarism detection tool. Bandara et al. tested three different algorithms, namely Naïve Bayes Classifier, k-Nearest Neighbor (kNN) and AdaBoost Meta-Learning. Nine different metrics such as LineLengthCalculator (number of charectes in a line) and UnderscoresCalculator (number of underscore in identifiers) were used to classify source codes. If the algorithm makes the wrong classification, this indicates the plagiarism. Bandara et al. also state that any algorithm for pattern recognition may be applied to their work.

Burrows et al. [35] present plagiarism detection approach for large code repositories. The study combines the local alignment methods in bioinformatics with indexing in search engines. It transforms source files into tokens and constructs an index of all files. Then, it queries the index using each program file in order to determine candidate matches. Lastly, local alignment is used to examine highly ranked files for refining the results.

Narayanan et al. [36] developed a plagiarism tool based on finger printing approach. The tool supports C, C++, Java, and Matlab and C# languages and detect the similarity between same or different languages.

Table 1.1 : Related works.

Reference	Tool / 1st author	Tool Type	Approach	Description
[23]	JPlag	Code Plagiarism	-	A web service that finds pairs of similar programs among a given set of programs.
[24]	Plaggie	Code Plagiarism	-	Gnu-licensed source code plagiarism detection engine for Java exercises.
[34]	Bandara	Code Plagiarism	-	A machine learning based tool for source code plagiarism detection.
[35]	Burrows	Code Plagiarism	-	Efficient plagiarism detection for large code repositories.
[36]	Narayanan	Code Plagiarism	-	Source code plagiarism detection and performance analysis using fingerprint based distance measure method.
[17]	Ducasse	Code Clone	Textual	A language independent approach for detecting duplicated code
[6]	CCFinder	Code Clone	Lexical	A multilinguistic token-based code clone detection system for large scale source code.
[5]	Dup.	Code Clone	Lexical	A program for identifying duplicated code.
[37]	FRICS	Code Clone	Lexical	Folding repeated instructions for improving token-based code clone detection.
[38]	Sajnani	Code Clone	Lexical	Parallel code clone detection using MapReduce.
[9]	CloneDr	Code Clone	Tree-matching	Clone detection using abstract syntax trees.
[18]	Davey	Code Clone	Metric-based	The development of a software clone detector.
[19]	Komondoor	Code Clone	Semantic	Using slicing to identify duplication in source code.
[20]	White	Code Clone	Hybrid	Deep learning code fragments for code clone detection.
[16]	CCLearner	Code Clone	Lexical	A Deep Learning-Based Clone Detection Approach.

1.2.2 Code Clone Detection Tools

There are five main categories regarding code clone detection approaches based on their analyses on source code and matching algorithms [11]: Textual, lexical, syntactic, semantic and hybrid approaches. Syntactic approaches are include two

sub-categories which are tree-matching and metric-based. Figure 1.1 illustrates all of these approaches graphically.

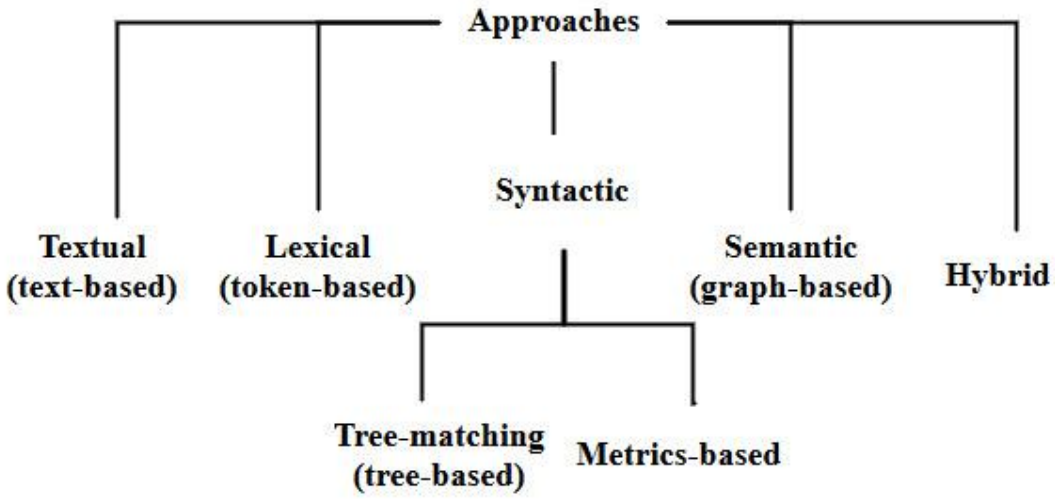


Figure 1.1: Code clone detection approaches.

1.2.2.1 Textual Approaches

Textual approaches use raw source or make little transformations on source code. They utilize string matching algorithms to detect clone pieces in the source. These approaches effectively detect Type-1 and Type-2 clones, but they could not manage to identify Type-3 and Type-4 clones successfully [11,16]. Ducasse et al. [17] is one of the most-known approaches that make profit from text based technique. Ducasse et al. construct a dot plot whose axes are source code entities and then compare hash values of these entities. If hash values are equal on two axes, then there is dot showing two source code entities are duplicate.

1.2.2.2 Lexical Approaches

Lexical approaches separate the source code into tokens using lexical analysis methods after removing white comments and white spaces. Each line is often transformed to identifiers that represent tokens. These identifiers may be numbers or any symbols, in fact one identifier might represent multiple tokens. For instance; a statement in the form of "a = b + 1;" may be transformed to "$ = $ + $;". After tokenization and transformation, the search for locating duplicated subsequences is started. After localization, the original source code corresponding to the duplicated

subsequences is transformed into code clones. Because of tokenization operation, lexical approaches are also known as token-based approaches.

CCFinder [6] uses token-based clone detection approach. It removes white spaces and transforms the source code based on transformation rules defined with respect to the programing language. For instance, accessibility keywords are removed in Java, so “protected: void foo();” becomes “void foo();”. After transformation, each parameter is replaced with a special token. Then, it searches for a duplicated token sequences. When it detects identical token sequences, they are marked as duplicate. CCFinder also uses metrics for quantitative evaluation, selection, and filtering on marked clone pairs in order to analyze the system deeply such as detecting frequently used code portions. These metrics are length, population of a clone class, deflation by a clone class, coverage of clone code and radius [6]. Length can be the length of code portion or the length of clone class. It can be measured with the number of tokens or the number of lines. Population of a clone class is the number of elements in clone class. The fact that this number is large means that similar code portions appear in many places. Deflation by a clone class measures an impact in size of reduction of each clone class. Coverage of clone code is defined for all clones in a whole source code. It measures the percentage of lines (% LOC) that includes any portion of clone, and the percentage of files (% FILE) that includes any clones. Radius measures of the level of influence of a clone class. If aclone class has a large RAD, the code portions are widely spread over a software system.

Dup. by Baker [5] is another tool that utilizes lexical approach. It takes advantage of p-string and p-matching. It first removes white spaces and comments and renames identifiers such as variable, method and class. Later, each line of code is hashed for the comparison. Duplicated code portions are located using suffix-tree algorithm. Compared to other approaches, token-based approaches are more efficient on detection of code clones [11].

FRICS by Murakami [37] normalizes the source code and replaces each identifier with a token. Then, it calculates the hash value of every expression between “;”, “{”, ve “}”. By subtracting the repeated expressions, the values are added to the first expression as weight. This method is to decrease the false-positive number to increase the detection rate. Expressions that have the same hash value are marked as clone.

Sajnani et al. [38] parallelizes the code clone detection process using MapReduce paradigm. Tokens are sorted by number of repetitions to obtain the list. By using the list in the first step and using the prefix filtering technique, similar code blocks are detected.

1.2.2.3 Syntactic approaches

Tree-matching and metric-based approaches are two subcategories of syntactic approaches. Tree-matching approaches also known as tree-based techniques. These techniques are based on creating AST tree for each fragments and searching for similar subtrees. Abstacting literal values, variable names and other tokens may help to detect more complicated clone types. Since tree-matching approaches focus on the syntactic structure of the source code rather than their statements, they can detect near-miss clones [11,16]. CloneDr by Baxter et al. [9] uses a tree-matching technique. First, it transforms the source code into a parse tree. Later, subtrees are hashed into buckets. These hashed subtrees are compared in order to identify clone portions.

Metrics can be anything related with source code such as number of input variable, number of function calls, and number of statements. Metric based approaches collect these metrics and compare vectors that contain collected metrics instead of comparing raw source code or transformed code [11,16]. Davey et al. [18] is an example work utilizing the metric based techinque. It uses metrics to produce features for code portions and trains a neural network to detect code clones.

1.2.2.4 Semantic approaches

In semantic based approaches, source code is represented as Program Dependency Graph or Control Flow Graph. They use semantic program anaylysis technique in order to transform statements and expressions as nodes, and to transform data and control dependencies as edges in the graph. Detecting isomorphic subgraphs corresponds to locating code clones in the original source code [11]. Komondoor and Horwitz's tool [19] generates Program Dependency Graph for detection of code clones.

1.2.2.5 Hybrid approaches

Hybrid approaches to identify duplicated code blocks are combinations of other two or more techniques.

There also exist code clone detection works utilizing deep learning, similar to our work. An example of these kind of approaches is the work of White et al. [20] which presents a learning-based detection technique. This technique relies on a language model which is a probability distribution over sentences in a language. Recurrent neural network and recursive neural network are used to map each term in a fragment to an embedding and to encode this embedding to characterize fragments.

CCLearner [16] is another example work that benefits from deep learning. CCLearner uses token-based techniques that combine tokens into eight category. These categories are reserved words, operators, markers, litarels, type identifiers, method identifiers and qualified names. After tokenization of the source code and separating these tokens into categories, a similarity score is calculated for each category in order to produce frequency vector. In this way, acquired vector with length of eight is used as an input to train deep neural network. CCLearner experiments with 1,626,544 samples taken from the dataset BigCloneBench. While it achieves 100%, 98%, 98% and 89% recall for T1, T2, VST3 and ST3 respectively, it has only 28% and 1% for MT3 and WST3/4 clone types. The reason is that training data does not include MT3 and WT3/4 types, so the deep neural network could not produce satifying results on these complicated clone types. Moreover, CCLearner's calculation for precision is based on 385 samples, and it is generalized to the all dataset. In addition, false positive rates of the experiment weren't reported in CCLearner. Regarding Type-3 clones are the most seen clone type, we would like to develop a technique that detects all clone types to see whether it can identify Type-3 and Type-4 clone types. We applied deep learning like CCLearner, but rather than applying deep neural network, we specifically preferred CNN that we thought it would be the best fit because our feature matrix looks like an image data. We also reported false positive rate in addition to recall and precision. Additionly, we didn't need to categorize tokens because of avoiding loss of information and relying on the power of CNN. We made empirical analysis on BigCloneBench like CCLearner, but with different number of samples.

2. CODE CLONE DETECTION

2.1 Definitions

In this section, we described definitions in the code clone detection literature.

2.1.1 Code Fragment

Any grouping of code lines in any granularity such as function definition, begin_end block, or sequence of statements is named as code fragment(CF). A CF is recognized by its file name and begin_end line numbers in the original code base. It is signified as a triple (CF.FileName, CF.BeginLine, CF.EndLine) [11]. In this work, the fragments are equal to methods (functions).

2.1.2 Code Clone

If two fragments are similar or identical, they are called code clone. In another words, if two fragments are code clone, it should be that f(CF1) = f(CF2) where f is the similarity function. Code clone may be among two or more code fragments. While a clone pair is formed by two similar code fragments (CF1, CF2), clone group or clone class is formed by many similar fragments [11].

2.1.3 Clone Types

Clone types between fragments can be grouped as two main kinds, which are based on the similarity of their program text and similarity of their functionality (independent of their text). [11]. The textual (Types 1 to 3) and functional (Type 4) clone types are provided as follows:

2.1.3.1 Type-1

Two fragments are Type 1 clones if they are identical except for variations in whitespace, layout and comments [11]. In Figure 2.1, whitespace is the only difference between two methods.

```
public String stringOfUrl(String addr) throws IOException {
      ByteArrayOutputStream output = new ByteArrayOutputStream();
      URL url = new URL(addr);
      IOUtils.copy(url.openStream(), output);
      return output.toString();
}

public String stringOfUrl(String addr) throws IOException {
      ByteArrayOutputStream output = new ByteArrayOutputStream();
      URL url = new URL(addr);
      IOUtils.copy(url.openStream(), output);

      return output.toString();
}
```

Figure 2.1 : An example of Type-1 clone.

2.1.3.2 Type-2

If two fragments vary with only identifiers, literals, types, whitespace, layout and comments, they are Type 2 clones [11]. Figure 2.2 is an example of Type-2 clone. Both methods copy given files in same way, but variable and parameter names are different. Regarding our results and previous research [11,25], Type 1 and Type 2 clones are the simplest types to detect.

```
public static void copyFile(String src, String target) throws IOException {
     FileChannel ic = new FileInputStream(src).getChannel();
     FileChannel oc = new FileOutputStream(target).getChannel();
     ic.transferTo(0, ic.size(), oc);
     ic.close();
     oc.close();
}

public void copyFile(File in, File out) throws Exception {
      FileChannel sourceChannel = new FileInputStream(in).getChannel();
      FileChannel destinationChannel = new ileOutputStream(out).getChannel();
      sourceChannel.transferTo(0, sourceChannel.size(), destinationChannel);
      sourceChannel.close();
      destinationChannel.close();
}
```

Figure 2.2 : An example of Type-2 clone.

2.1.3.3 Type-3

If a fragment can be obtained from another with little modifications such as changed, added or removed statements, these two fragments are Type 3 clone [11]. Figure 2.3 shows an example of Type-3 clone. Added line 3 and modified line 4 in second method are variations between two methods.

```
public String streamToString(InputStream stream) throws IOException {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    IOUtils.copy(stream, output);
    return output.toString();
}

public String stringOfUrl(String addr) throws IOException {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    URL url = new URL(addr);
    IOUtils.copy(url.openStream(), output);
    return output.toString();
}
```

Figure 2.3 : An example of Type-3 clone.

2.1.3.4 Type-4

Two or more code fragments may perform the same computation by implementing different syntactic variants. These types of fragments are known as Type-4 clone [11]. In Figure 2.4, both methods perform the same operation in different ways.

```
public static final String getMD5Hash(byte[] data, int offset, int length) throws
NoSuchAlgorithmException {
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(data);
    return generateHash(md5.digest());
}

public String getHash(String key, boolean base64) throws Exception {
    MessageDigest md = MessageDigest.getInstance("SHA");
    md.update(key.getBytes());
    if (base64) return new String(new Base64().encode(md.digest()), "UTF8");
    else return new String(md.digest(), "UTF8");
}
```

Figure 2.4 : An example of Type-4 clone.

2.2 Clone detection process

Clone detection process consists of several steps: pre-processing, transformation, match detection, formatting and post-processing/filtering. Tools may implement any of these steps. It is not mandatory to apply all of these steps at the same time.

2.2.1 Pre-processing

Pre-processing is the first step in code clone detection process. The aim of the step allows code pieces to be shaped in the same format in order to obtain more comparable input fragments [11]. There are three substeps in pre-processing.

2.2.1.1 Remove uninteresting parts

There may be various languages and technologies used in the same software system. For instance, a java project might include both java and sql statements. It may be good idea to eliminate uninteresting parts, especially if the code clone detector is language dependent. Similarly, generated code and sections of source code that are likely to produce many false positives (such as table initialization) can be removed [11].

2.2.1.2 Determine source units

In this step, the size of code pieces that can be returned as clone should be determined. Source code should be divided into units considering the possibility of being clone of two units with irrelavant size [11]. It is unlikely that two fragments with different size such as method and class are duplicate.

2.2.1.3 Determine comparison units/granularity

Comparison units are different from source units. Regarding the comparison technique of the tool, source unit should be separated into smaller unit. While source units are returned code pieces as clone, comparison units are used in match detection step [11]. For instance, two methods (source unit) may be detected as a clone pair by comparing line by line (comparison unit).

2.2.2 Transformation

Second step is transformation. If the clone detection approach is based on a technique other than text-based, source code is represented with tokens, trees or graphs. These are called extraction. In addition, some normalization methods may be applied before or after the extraction phase [11].

2.2.2.1 Extraction

There are three different extraction techniques to transform source code into suitable input format for match detection algorithm. These techniques are tokenization, parsing and control and data flow analysis. One or more extraction technique can be applied [11].

Tokenization: In case of token-based approaches, source code are represented as token sequences. Source code is transformed into tokens based on lexical rules of the programing interest [11].

Parsing: In parsing, source code is transformend into parse-tree or abstract syntax tree (AST). Syntactic approaches apply parsing and search for subtrees for detecting duplicate code portions [11].

Control and data flow: In this technique, source code is transformend into program dependency graph (PDG). In PDG, statements and conditions are represented as node and control and data dependencies are represented as edge. Semantic approaches use PDG and search for ismorphic subgraphs subtrees for detecting duplicate code portions [11].

2.2.2.2 Normalization

This step helps to reduce slight differences between candidate clone couples.

Removal of whitespace: Whitespace is a slight difference that may cause to reduce accuracy rates of the clone detector.

Removal of comments: Comments are another slight differences that may cause to reduce accuracy rates of the clone detector. A programmer may copy a code portion

and add only comments to the copied portion. These comments may cause the misclassification of clones [11].

Normalizing identifier: In Type-2 clones, the only difference between two code pieces is replaced identifiers. Thus, identifier may be replaced by a same identifier [11].

Structural transformations: In order to eliminate slight differences between code portions, some structural transformations such as removing static keyword in C may be applied.

2.2.3 Match detection

After transformation and eliminating slight differences, the final version of the source code is given as input to diverse range of comparison algorithms such as string comparison algorithm or deep learning based algorithm [11]. A list of clone pairs in the transformed version is the final output of match detection phase.

2.2.4 Formatting

Formatting is a back transformation operation on the list of clone pairs. Each pair is converted into original source code, and coordinates of the pair are identified in the source code [11].

2.2.5 Post-processing/Filtering

Manual analysis: Defined clones are ranked or eliminated manually by human expert.

Automated heuristics: Defined clones are ranked or eliminated automatically using heuristics.

3. PROPOSED APPROACH

3.1 Dataset

BigCloneBench is a large benchmark that contains six million validated code clones. It was built by mining and manually validating IJaDataset-2.0 which is an inter-project repository in Java language [13]. The dataset was generated using 2.5 million Java classes and their more than 22 million methods. To produce the dataset, Svajlenko et al. [13] mined IJaDataset by classifying methods with respect to the frequently used ten functionalities without using any clone detector. They used search heuristics to automatically separate code blocks regarding implemented functionalities. Later, these code blocks are manually checked and labeled as true or false positives of the functionalities. In this process, code blocks were also populated as true and false clones.

BigCloneBench consists of six extended clone types: T1(Type 1), T2(Type2), VST3 (Very Strong Type 3), ST3(Strong Type 3), MT3(Moderately Type 3) and WT3/4(Weak Type 3 or 4). VST3, ST3, MT3 and WT3/4 types were generated by reinterpreting Type-3 and Type-4 clone types. Svajlenko et al. [13] measured syntactical similarity and generated a ratio between two code snippets called similarity ratio. VST3, ST3, MT3 and WT3/4 clone types were classified based on the similarity ratio between two code snippets. Table 3.1 shows similarity ratios and the distribution of samples according to these ratios. If similarity ratio between two clone candidates is more than 0.9, they are labeled as WST3. If the similarity ratio is between 0.7 and 0.9, they are ST3. When the ratio is lower than 0.7 and higher than 0.5, they become MT3. If the similarity is lower than 0.5, they are WST3/4. Table 3.1 shows the distribution of clones based on their types. The whole dataset consists of WST3/4 samples (6,158,975) and there are rarely T2 (3787) and VST3 (2083) samples.

When we analyze BigCloneBench deeply, we may declare that more than 90% of classes is formed with around ten or less methods. Figure 3.1 shows the percentage of methods that each class has, and Figure 3.2 shows the mean value of method counts that each class has. Moreover, Figure 3.3 demonstrates the distrubutions of clone samples based on similarity rule. Note that T1 and T2 clones are also included with similarity ratio 1 in Figure 3.3.

Table 3.1 : Distrubution of clones

Type	Similarity(s) Ratio	Count	%
T1	-	16185	0.24
T2	-	3787	0.06
VST3	0.9 <= s <= 1	2083	0.03
ST3	0.7 <= s < 0.9	10031	0.15
MT3	0.5 <= s < 0.7	55106	0.85
WT3/4	0 <= s < 0.5	6158975	94.63
NOT CLONE	-	262465	4.03

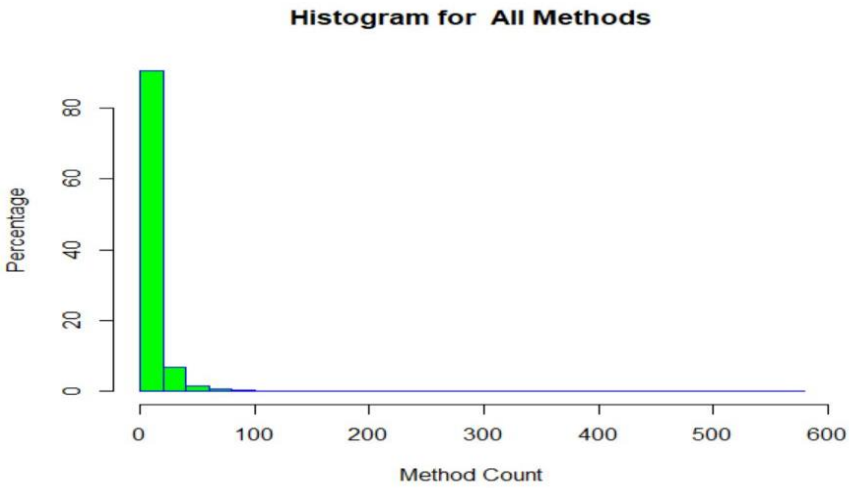


Figure 3.1 : Histogram of methods each class has.

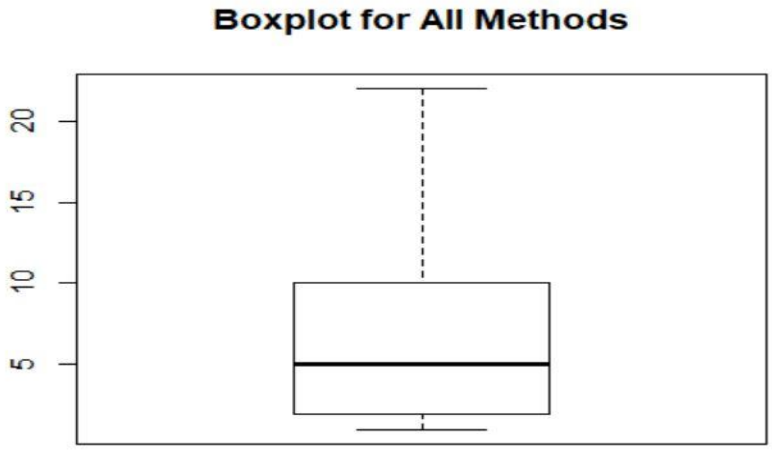


Figure 3.2 : Box plot for method number of classes.

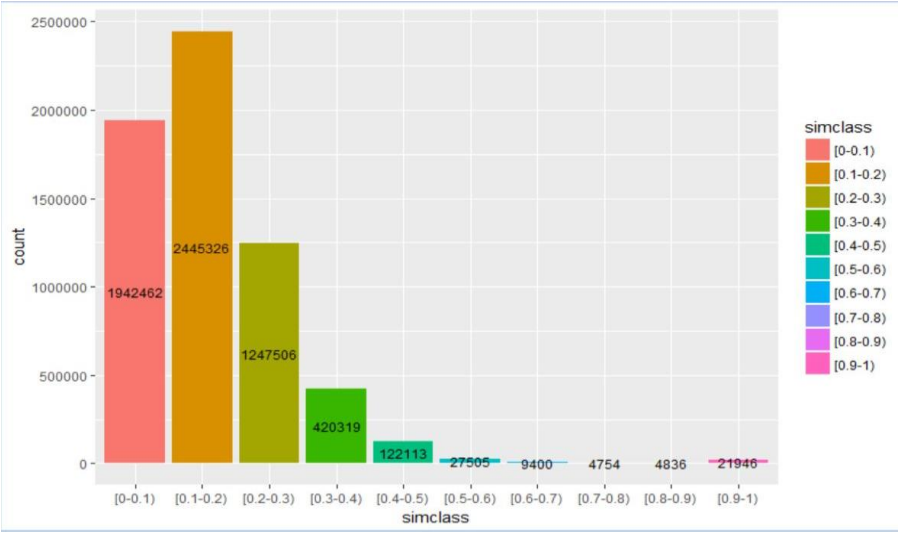


Figure 3.3 : Bar chart showing distributions of methods pair with respect to their similarity ratio.

3.2 Tokenization and Input Features

CNN gets image data as input. To acquire reasonable input for CNN, each method pair must be transformed into image data. With the lexical analyzer tool ANTLR [15], each method pair (method1, method2) is separated into tokens. ANTLR offers pre-defined 107 tokens and the corresponding unique IDs so that code portions can be transformed into token sequences or IDs. Table 3.2 demonstrates every token and its corresponding token ID.

Each method is represented as an array of size 107 (total number of tokens) in the tokenization. Indices of the array correspond to token ID and value at that index shows the number of times that the token appears in the method. Tokens that never show up are counted as 0. After representing method1 and method2 as arrays, these two arrays are merged to acquire one single method pair array, which would be the feature set. The final feature set corresponds to a token array of size 214. The first 107 values of the array show the number of occurrences of tokens in method1 and the last 107 values represent method2. This token array is later saved as an image in jpeg format.

Figure 3.4 demonstrates an example procedure of tokenization. There are two methods (a and b) named as “copy” and “cloneFile”. Both makes file copy operation in different ways so that they are clones of each other. There are also their corresponding token arrays (c and d). In the first array (c), the token “If” seems just once, thereby the value at index 22 is 1. So also, the value at index 22 in the second

array (d) is 2, because “If” appears two times in the method “cloneFile”. Similarly, the values at index 3 for both arrays are 0 since the token “Boolean” never appears. (e) shows merged version of these two arrays.

Table 3.2 : Tokens and corresponding unique IDs.

TOKEN	ID	TOKEN	ID	TOKEN	ID	TOKEN	ID
ABSTRACT	1	INTERFACE	28	StringLiteral	55	SUB	82
ASSERT	2	LONG	29	NullLiteral	56	MUL	83
BOOLEAN	3	NATIVE	30	LPAREN	57	DIV	84
BREAK	4	NEW	31	RPAREN	58	BITAND	85
BYTE	5	PACKAGE	32	LBRACE	59	BITOR	86
CASE	6	PRIVATE	33	RBRACE	60	CARET	87
CATCH	7	PROTECTED	34	LBRACK	61	MOD	88
CHAR	8	PUBLIC	35	RBRACK	62	ARROW	89
CLASS	9	RETURN	36	SEMI	63	COLONCOLON	90
CONST	10	SHORT	37	COMMA	64	ADD_ASSIGN	91
CONTINUE	11	STATIC	38	DOT	65	SUB_ASSIGN	92
DEFAULT	12	STRICTFP	39	ASSIGN	66	MUL_ASSIGN	93
DO	13	SUPER	40	GT	67	DIV_ASSIGN	94
DOUBLE	14	SWITCH	41	LT	68	AND_ASSIGN	95
ELSE	15	SYNCHRONIZED	42	BANG	69	OR_ASSIGN	96
ENUM	16	THIS	43	TILDE	70	XOR_ASSIGN	97
EXTEBDS	17	THROW	44	QUESTION	71	MOD_ASSIGN	98
FINAL	18	THROWS	45	COLON	72	LSHIFT_ASSIGN	99
FINALLY	19	TRANSIENT	46	EQUAL	73	RSHIFT_ASSIGN	100
FLOAT	20	TRY	47	LE	74	URSHIFT_ASSIGN	101
FOR	21	VOID	48	GE	75	Identifier	102
IF	22	VOLATILE	49	NOTEQUAL	76	AT	103
GOTO	23	WHILE	50	AND	77	ELLIPSIS	104
IMPLEMENTS	24	IntegerLiteral	51	OR	78	WS	105
IMPORT	25	FloatingPointLiteral	52	INC	79	COMMENT	106
INSTANCEOF	26	BooleanLiteral	53	DEC	80	LINE_COMMENT	107
INT	27	CharacterLiteral	54	ADD	81		

```
static void copy(String src, String dest) throws IOException {
    File ifp = new File(src);
    File ofp = new File(dest);
    if (ifp.exists() == false) {
        throw new IOException("file '" + src + "' does not exist");
    }
    FileInputStream fis = new FileInputStream(ifp);
    FileOutputStream fos = new FileOutputStream(ofp);
    byte[] b = new byte[1024];
    int readBytes;
    while ((readBytes = fis.read(b)) > 0) fos.write(b, 0, readBytes);
    fis.close();
    fos.cloes();
}
```

(a)

```
private static final void cloneFile(File origin, File target)
                                        throws IOException {
    FileChannel srcChannel = null;
    FileChannel destChannel = null;
    try {
        srcChannel = new FileInputStream(origin).getChannel();
        destChannel = new FileOutputStream(target).getChannel();
        destChannel.transferFrom(srcChannel, 0, srcChannel.size());
    } finally {
        if (srcChannel != null) srcChannel.close();
        if (destChannel != null) destChannel.close();
    }
}
```

(b)

```
[0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,1,0,0,0,0,1,0,0,0,6,0,0,0,0,
 0,0,1,0,0,0,0,0,1,1,0,0,1,0,1,3,0,1,
 0,2,0,14,14,2,2,2,2,10,3,5,6,1,0,0,0,
 0,0,1,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,40,0,0,0,0,0]
```

(c)

```
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 1,1,0,0,2,0,0,0,0,0,0,0,2,0,1,0,0,
 0,0,1,0,0,0,0,0,0,1,0,1,1,0,0,1,0,0,
 0,0,4,11,11,3,3,0,0,7,3,6,4,0,0,0,0,
 0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,29,0,0,0,0,0]
```

(d)

```
[0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,6,0,0,0,0,0,
 0,1,0,0,0,0,0,1,1,0,0,1,0,1,3,0,1,0,2,0,14,14,2,2,2,2,10,3,5,6,1,0,0,0,0,
 0,1,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,40,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,2,0,0,0,0,0,0,0,0,2,0,1,0,0,
 0,0,1,0,0,0,0,0,0,1,0,1,1,0,0,1,0,0,0,0,4,11,11,3,3,0,0,7,3,6,4,0,0,0,0,0,
 0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,29,0,0,0,0,0]
```

(e)

Figure 3.4: Preprocessing on methods. Two java method copy (a) and cloneFile (b) and their corresponding arrays (c) and (d), also their merged version (e).

3.3 Training and Test Set Construction

We worked on three different approaches to build the training data. Initially, we used real data ratios. As summarized in Table 3.1, BigCloneBench has diverse number of samples at each type of clones. It consists of 0.24% T1, 0.06% T2, 0.03% VST3, 0.15% ST3, 0.85% MT3, 94.63% WT3/4 and 4.03% false clone samples. Keeping these proportions the same, we built the training data with a sum of 50 thousands of samples. The first line of Table 3.3 demonstrates the number of samples on different clone types that form the training data. There are 124 T1, 29 T2, 16 VST3, 77 ST3, 423 MT3, 47314 WT3/4 and 2016 false clone samples, which are same distribution ratios in the BigCloneBench.

Table 3.3: Number of samples for real data ratios.

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Train	124	29	16	77	423	47314	47983	2016
Test#1	25	6	3	15	85	9463	9597	403
Test#2	25	6	3	15	85	9463	9597	403
Test#3	25	6	3	15	85	9463	9597	403
Test#4	25	6	3	15	85	9463	9597	403
Test#5	25	6	3	15	85	9463	9597	403
All Test	125	30	15	75	425	47315	47985	2015

Secondly, we tried micro-sampling, i.e. undersampling the classes based on the minority class ratio. We construct training data picking 2000 samples for each clone type because VST3 has only 2083 samples which is the smallest number of samples in BigCloneBench. Thus, our training data consists of 14 thousands of samples. Table 3.4 shows the distrubution of micro sampling training data.

Table 3.4: Number of samples for micro sampling.

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Train	2000	2000	2000	2000	2000	2000	12000	2000
Test#1	25	6	3	15	85	9463	9597	403
Test#2	25	6	3	15	85	9463	9597	403
Test#3	25	6	3	15	85	9463	9597	403
Test#4	25	6	3	15	85	9463	9597	403
Test#5	25	6	3	15	85	9463	9597	403
All Test	125	30	15	75	425	47315	47985	2015

In the third strategy, we constructed the training data by choosing random number samples for each clone type. We selected 50 thousands of method pairs ignoring clone types and built the training set. We ended up having 3119 T1, 697 T2, 417 VST3, 1913 ST3, 6912 MT3, 12721 WT3/4 and 25000 false clone samples as illustrated in Table 3.5.

Table 3.5: Number of samples for random ratio.

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Train	3119	697	417	1913	6912	12721	25779	25000
Test#1	25	6	3	15	85	9463	9597	403
Test#2	25	6	3	15	85	9463	9597	403
Test#3	25	6	3	15	85	9463	9597	403
Test#4	25	6	3	15	85	9463	9597	403
Test#5	25	6	3	15	85	9463	9597	403
All Test	125	30	15	75	425	47315	47985	2015

We build five distinctive test data for each training in order to get more reliable results. Unlike training data, we follow only one methodology. We used real data ratios strategy to be more realistic since we believe clone ratios of real-life software is very close this ratio The instances in the training set are not included into these test sets to avoid sampling bias. Eventually, each test has 25 T1, 6 T2, 3 VST3, 15 ST3, 85 MT3, 9463 WT3/4 and 403 not clone samples. Because of constructing five different test set, we ended up with 125 T1, 30 T2, 15 VST3, 75 ST3, 425 MT3, 47315 WT3/4 and 2015 not clone samples. Table 3.3, Table 3.4 and Table 3.5 show test sets besides their training sets.

3.4 Convolutional Neural Network

Convolutional Neural Networks are kinds of multi-layer neural networks, which are specially designed for visual pattern recognition. They consist of neurons that can take input and perform dot products and they are trained with a backpropagation algorithm like every other neural network [12, 39]. The difference between CNN and ordinary neural networks is that all neurons are not connected to all neurons in the next layer. Neurons are connected to next layer's neurons based on localization of the neurons. This means that each neuron takes input as small portion of the image [40]. The most important layers and components of CNN are described below.

3.4.1 Input Layer

Input layer holds the raw pixel values of input images.

3.4.2 Convolution Layer

Convolotion layer consists of set of filters (neurons) named as kernel. Each neuron receives some portions of the input image and apply convolution operation. Convolution operation is performed by computing dot product between input and kernels [12]. Kernels are learnerable weight matrices. In each pass of traning, weight matrices are updated. When an input image has shape W1*H1*D1 and with K filters (F*F*D1), it produces W2*H2*D2 where

$W2=(W1-F)/S+1$ (S is stride size)
$H2=(H1-F)/S+1$
$D2=K$

3.4.3 Pooling Layer

Pooling layer performs downsapling operation to reduce the dimensional heavy computational load. It reduces the number of paremeters, so it helps to restrict overfitting. Generally, it is inserted after convolutional layers. Pooling makes the networks less sensitive to the exact location of pixels [12, 40]. When an input image has shape W1*H1*D1 and with filters size F*F*D1, it produces W2*H2*D2 where

W2=(W1−F)/S+1 (S is stride size)
H2=(H1−F)/S+1
D2=D1

There are two kind of pooling technique: max pooling and average pooling.

3.4.3.1 Max pooling

It takes the maximum pixel value of the input region. Figure 3.5 demonstrates an example of max pooling operation. The input iamge has shape 4*4*1 and filter size is 2*2 with stride 2, so output with shape 2*2 are generated.

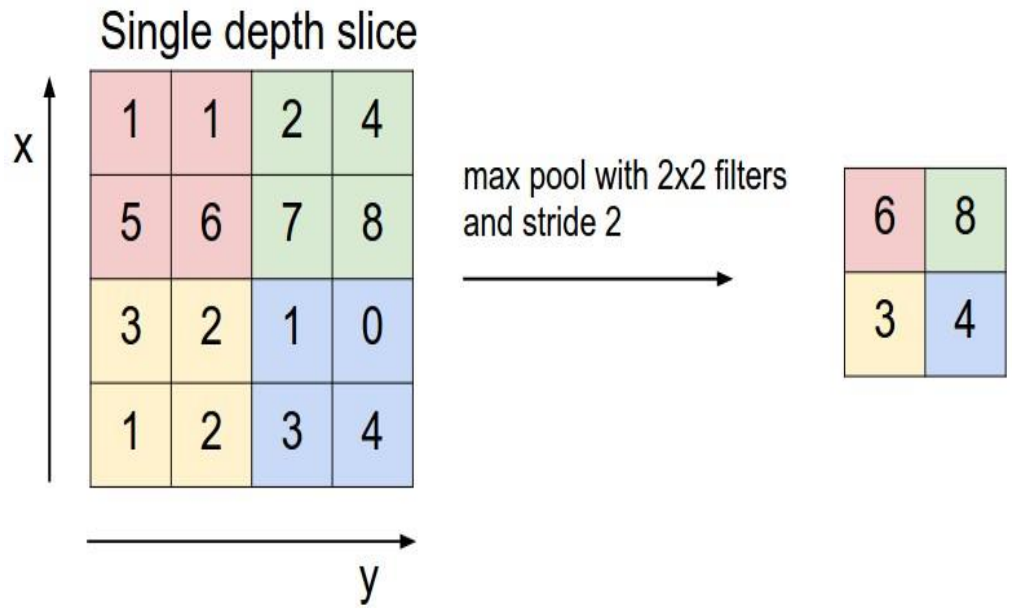


Figure 3.5: Max pooling with 2x2 filters [12].

3.4.3.2 Average pooling

Average pooling takes the average pixel value of the input region.

3.4.4 Activation Functions

The activation function is used to define output of the neuron. It transforms the result value of the neuron to desired range [40]. Commonly used activation functions are sigmoid, relu and tanh.

3.4.4.1 Sigmoid

The sigmoid function squashes the result value of the neuron into range between 0 and 1. It computes the function $\sigma(x)=1/(1+e^{-x})$. In sigmoid function, large negative numbers are transformed into 0 and large positive numbers are transformed into 1 [12]. Figure 3.6 shows the sigmoid function.

3.4.4.2 Relu

The rectifier linear units have been proposed as an activation function and become very popular in last few years. It has mathematical form $f(x)=\max(0,x)$. In ReLU, activation is thresholded at zero [12, 40]. Figure 3.6 shows the ReLU function.

3.4.4.3 Tanh

In tanh, the result value of the neuron is squashed into range between -1 and 1. It applies the function $\tanh(x)=2\sigma(2x)-1$ [12]. Figure 3.6 shows the tanh function.

3.4.5 Fully Connected Layer

Fully connected layer is a typical neural network. All neurons are fully connected to previous layer's neurons. It generates probability distrubution of each classes. Fully connected layer mostly located at the end of the network and used for classification. If there exist ten classes, this layer should produce ten different probabilities.

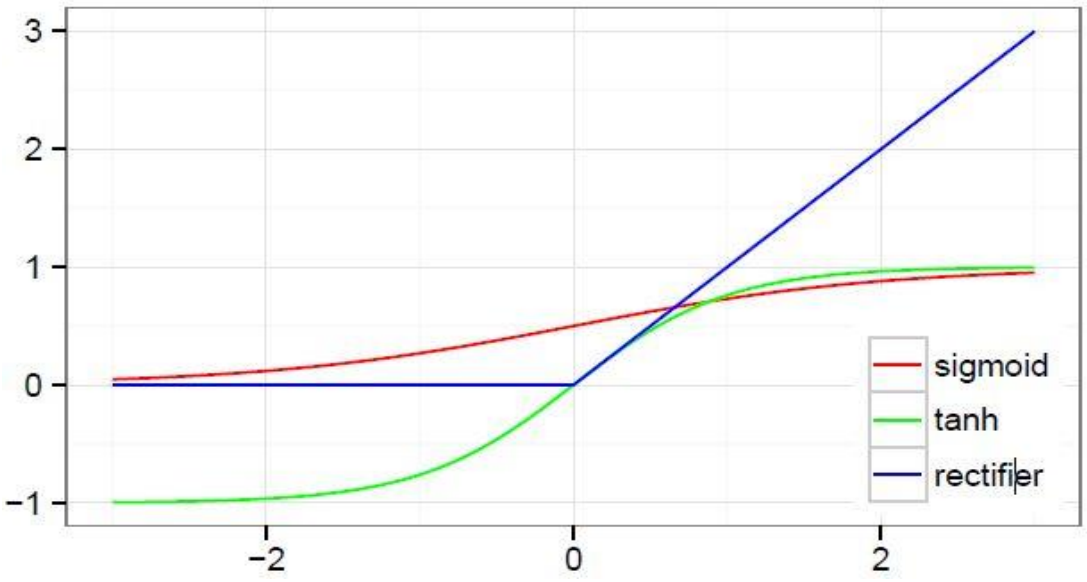


Figure 3.6: Activation functions [40].

3.5 CNN Architecture

We used Lenet [14] architecture that has two convolutional layers. Every convolution is followed by a pooling layer. First convolution has 20 filters with size 5*5 and stride 1, second convolution has 50 filters with size 5*5 and stride 1. Each pooling layer is max pooling with kernel size 2*2 and stride 2. After the second pooling layer, we have two fully connected layers. The first one consists of 50 hidden layers and 500 output layers and second one has 50 hidden layers with 2 two outputs. Also, we have a RELU between this two fully connected layers. Lastly, the final layer connected to a softmax to calculate loss. As a shorthand notation C(20,5,1) -> P(2,2) -> C(50,5,1) -> P(2,2) -> FC(50,10) -> R() -> FC(50,2) where C(n,f,s) indicates a convolution layer with n filter, f*f spatial size, s stride; R indicates RELU; P(f,d) indicates pooling with f*f spatial size; s stride and FC(h,o) indicates fully connected layer with h hidden, o output nodes. Figure 3.7 shows the illustration of the model.

3.6 Initialization and Optimization of Parameters

We applied fine-tuning parameters for initialization and optimization in the CNN. All the weights in the fully connected and convolution layers are initialized with Xavier initialization [21]. We also used Adam momentum update [22] with momentum 0.9 and learning rate 0.01. Batch size was 64.

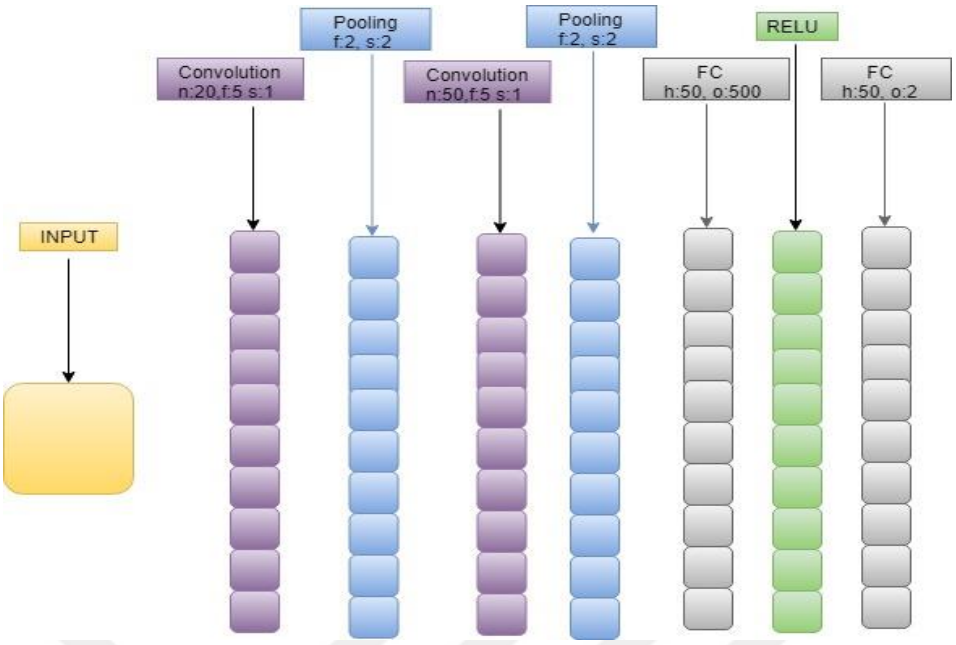


Figure 3.7: Illustration of CNN architecture.

3.7 Performance Evaluations

We use recall, precision, false positive rate and accuracy as our performance metrics in this work. Since our dataset consists of two labels which are "clone" and "not clone", we defined our metrics for both clone and not clone samples.

Recall (R), measures how many of clones are detected and how many of not clones are detected.

$$R = \frac{\#\text{ of Clones Truly Predicted}}{\text{Total \# of Clones}} \quad (1)$$

$$R = \frac{\#\text{ of Not Clones Truly Predicted}}{\text{Total \# of Not Clones}} \quad (2)$$

Precision (P), measures how many of detected clones are actually clone and how many of detected not clones are actually not clone.

$$P = \frac{\#\text{ of Clones Truly Predicted}}{\text{Total \# of Predicted Clones}} \quad (3)$$

$$P = \frac{\#\text{ of Not Clones Truly Predicted}}{\text{Total \# of Predicted Not Clones}} \quad (4)$$

False Alarm Rate (F), measures how many of the actual not clones are predicted as clone, and how many of the actual clones are predicted as not clone.

$$\mathrm{F} = \frac{\#\ \text{of Clones Falsely Predicted}}{\text{Total \# of Not Clones}} \qquad (5)$$

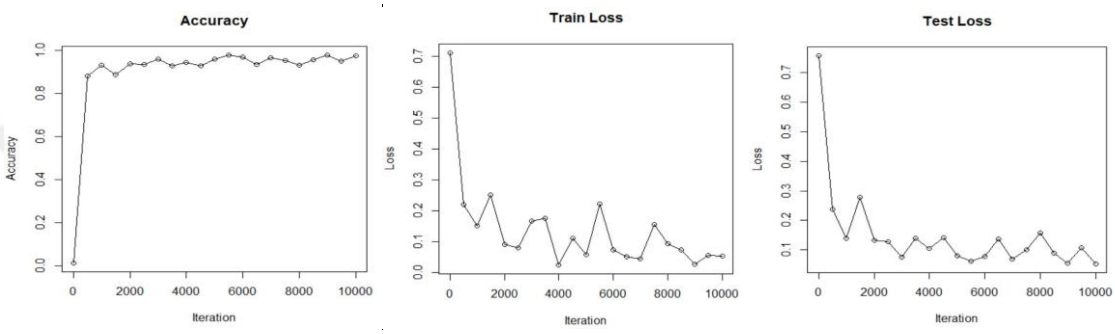
$$F = \frac{\#\ \text{of Not Clones Falsely Predicted}}{\text{Total \# of Clones}} \qquad (6)$$

Accuracy (A), is an overall measure to compute how many of samples are truly classified.

$$\mathrm{A} = \frac{\#\ \text{of Samples Truly Predicted}}{\text{Total \# of Samples}} \qquad (7)$$

4. RESULTS

We performed experiments on three different training sets and five different test sets corresponding to each training set. Training took about 10 minutes with maximum 10000 iterations. For every 500 iterations, model is built and accuracies are taken. We took 0.95 accuracy with 0.05 training loss and 0.05 test loss. Figure 4.1 shows accuracy and loss plots for the architecture trained in the experiment performed with micro sampling methodology.



(a) (b) (c)

Figure 4.1 : Loss and accuracy plots. (a) accuracy (b) train loss (c) test loss

We calculated the performance metrics for each of the experiments. Tables 4.1 -4.7 except Table 4.4 shows the computed performance metrics. While Tables 4.1, 4.2 and 4.3 report recall rates with respect to clone types, and overall recall for each experiment, Tables 4.5, 4.6 and 4.7 report precision, false alarm rate and accuracy for each experiment. We compared our work with CCLearner because CCLearner is another work that applies a deep learning technique to the same dataset. Table 4.4 reports the performance of CCLearner to compare our findings with the recent approach.

We achieve to get 100% recall and 96% precision for the clone samples in the trial utilizing real data ratios, as in Table 4.1 and Table 4.5 respectively. However, the recall value for not clone samples is lower (0.7%). While the network can effectively predict the clone samples, it can not succeed in predicting not clone samples. Our model couldn't learn the shape of the not clone samples, because we didn't give adequate not clone samples while keeping the real ratio of clone types.

We accomplished more encouraging outcomes in the other two trials experiments. Table 4.2 and 4.3 report the recall for each clone type and overall recall. Additionly, Table 4.6 and 4.7 report precision, false alarm rate and accuracy for these two experiments. We took 95% accuracy for micro sampling and 93% accuracy for randomly selected samples. Also, while recall values remain approximately the same, precision for not clone samples exceeds 70 percent. This demonstrates that our sampling techniques function admirably in predicting both clone and not clone classes. When we compare these two techniques, we can state that randomly selected training set has better recall values while micro sampling has better precision for not clone samples.

Table 4.1: Recall for real data ratio (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Test#1	100	100	100	100	100	100	100	0,7
Test#2	100	100	100	100	100	100	100	0,7
Test#3	100	100	100	100	100	100	100	0,7
Test#4	100	100	100	100	100	100	100	0,7
Test#5	100	100	100	100	100	100	100	0,7
Overall	100	100	100	100	100	100	100	0,7

Table 4.2 : Recall for micro sampling (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Test#1	100	100	100	100	95	95	96	77
Test#2	100	100	100	100	92	96	96	76
Test#3	100	100	100	100	98	95	96	75
Test#4	96	100	100	100	99	95	95	72
Test#5	100	100	100	93	95	96	96	74
Overall	99	100	100	99	96	96	96	75

Table 4.3 : Recall for random data ratio (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
Test#1	100	100	100	87	87	92	92	98
Test#2	100	100	100	93	93	92	92	98
Test#3	100	100	67	100	86	93	93	99
Test#4	100	100	100	80	87	92	92	98
Test#5	100	100	100	87	87	92	92	97
Overall	100	100	93	89	88	92	92	98

Table 4.4 : Recall for CClearner (%)

	T1	T2	VST3	ST3	MT3	WT3/4	All Clone	Not Clone
CCLearner	100	98	98	89	28	1	-	-

Table 4.5 : Precision-false alarm rate-accuracy for real data ratio (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
Test#1	96	100	100	0	96
Test#2	96	100	99	0	96
Test#3	96	100	99	0	96
Test#4	96	100	99	0	96
Test#5	96	67	100	0	96
Overall	96	93	99	0	96

Table 4.6 : Precision-false alarm rate-accuracy for micro sampling (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
Test#1	99	42	23	4	95
Test#2	99	44	24	4	95
Test#3	99	41	25	4	95
Test#4	99	38	28	5	94
Test#5	99	41	26	4	95
Overall	99	41	25	4	95

Table 4.7 : Precision-false alarm rate-accuracy for random data ratio (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
Test#1	100	35	2	8	93
Test#2	100	35	1	8	93
Test#3	100	36	1	7	93
Test#4	100	34	1	8	92
Test#5	100	34	3	8	92
Overall	100	35	2	8	93

Table 4.8 : Precision-false alarm rate-accuracy for CCLearner (%)

	Precision		False Alarm Rate		Accuracy
	Clone	Not Clone	Clone	Not Clone	
CCLearner	93	-	-	-	-

5. THREATS TO VALIDITY & INTERPRETATIONS

5.1 Threats To Validity

We selected samples from BigCloneBench [13] dataset and evaluated our work relying on the fact that samples in this dataset are labelled correctly. BigCloneBench is the largest dataset in code clone detection and so many works [16,20,41,42] uses this dataset. Thus, we worked with the dataset accepting it is reliable.

Samples which we selected may be the best fitting samples with our model and may effected that the model showed better results. However, we made our experiments by creating five different test sets to overcome this. The results we took for each test set were very close to each other. Thus, we think that our work conducted valid results.

5.2 Interpretations

Considering outcomes, our model accomplishes detecting code clones remarkably. For both clone and not clone samples, the model presents great recall values. Likewise, the model has very encouraging precision results for clone samples although precision value for not clone samples and false alarm rate for clone samples may need further improvements.

We tried our model selecting 10000 test samples and repeating it five times for every experiment, and we observed the achievement of the model on code clone detection. The full execution time for the approach we have applied is less than an hour. It is about minutes for tokenization and preparing input features. Furthermore, CNN runs in about 20 minutes, including training and test phases. We think that the complete execution time is reasonable for applying a deep learning approach. Regarding the execution time and taking into account that our feature vector looks like an image CNN is a good algorithm for detection of code clones.

If we consider the distribution of samples with respect to clone types, the number of samples for each clone types very different from one another. 94.63% of the dataset is formed by WT3/4, but 0.06% of the dataset is consists of T2. Although the WT3/4 dominates the dataset, the model gives very successful results in all clone

types including T2. We think CNN has the capacity to overcome different clone types.

We compared our micro sampling experiment results with CCLearner [16] because of its better and reliable results. CCLearner results are reported in Table 4.4 and 13. Table 4.4 demonstrates recall for each clone type. Table 4.8 shows only precision for clone samples. Recall for not clone samples and other overall metrics are missing (represented as -) because they were not reported by Li et al. [16].

Althought our work and CCLearner took almost same result for simple clone types which are T1, T2 and VST, our model has essentially better outcomes on complicated clone types. While we got 99%, 96% and 96% recall for ST3, MT3 and WT3/4 clone types respectively, CCLearner only has 89%, 28% and 1% recall value for these types. The reason is that CCLearner doesn't include MT3 and WT3/4 clone types in training data. Without learning these sophisticated clone types in training phase, CCLearner could'n succeed in detecting them. In addition, we have better precision on detecting clones. Because of the values that were not reported by Li et al. [16], we are not able to compare our false alarm rate, accuracy and not clone precision results with CCLearner.

6. CONCLUSION

In this work, we propose a clone detection technique which combines tokenization and deep learning practices. The model has good ability to classify java methods as ‘clone’ and ‘not clone’. We took three different sets to train the model and five test sets at each training set. Using these data we trained and tested our CNN. We reported the results comparing the similar approach and saw that our approach has considerable amount of contribution. Compared to prior work, different types of clones (Type 3 and 4) are successfully detected with a recall rate between 93-100% and false positive rate between 72-77%.

In the future, our work can be improved by taking several steps in model construction and dataset. One of them could be varying the dataset size, since we only used a small portion of the whole dataset. By increasing the dataset size, the model could learn the minority clone types better although the majority is still dominated by the MT3 and WT3/4 clone types. A sampling technique would essentially be necessary to keep the balance between different clone types in training set, and therefore, we applied the micro-sampling technique and obtained more successful results. The test set should always reflect the real scenario, as in practice when the model is used to predict whether a method pair is clone, this pair is most likely be a MT3 or WT3/4 clone type.

Second, different CNN architecture may be constructed with different layers and initialization parameters. Our CNN model is currently completing training and prediction along with dataset construction less than an hour. We think this is reasonable, but improvements and scalability works can be done. Further, other machine learning methods may be tried instead of CNN.

Another work can be done on feature selection. We counted tokens and stored them as frequency list, so we lost the places of tokens and relation between them. Instead of frequency list, it may be possible to extract token vectors based on Word embeddings so that the place of each token and its relation to the prior and next token will be considered, although it is challenging to construct equal sized list because of a wide variety of method size. It is also possible that tokens for another languages may be extracted. For instance, other token numbers with different unique IDs may

be defined for SQL so that our work supports both Java and SQL languages. However, increasing number of tokens results in a larger feature matrix and it may cause increasing the training and prediction time. Our future research direction is to combine CNN with text mining approaches such as topic modelling and word embeddings to identify related tokens, and analyze their impact on code clone detection.

REFERENCES

[1] **B. Lague, E.M. Merlo, J. Mayrand, and J. Hudepohl,** ªAssessing the Benefits of Incorporating Function Clone Detection in a Development Process,º Proc. IEEE Int'l Conf. Software Maintenance (ICSM '97), pp. 314-321, Oct. 1997.

[2] **J. Johnson**, Visualizing textual redundancy in legacy source, in: Proceedings of the 1994 Conference of the Centre for advanced Studies on Collaborative research, CASCON 2004, 1994, pp. 171_183.

[3] **S. Ducasse, M. Rieger, S. Demeyer**, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, 1999, pp. 109_118.

[4] **C.K. Roy, J.R. Cordy**, An empirical study of function clones in open source software systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, 2008, pp. 81_90.

[5] **B. Baker**, A program for identifying duplicated code, in: Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, vol. 24, 1992, pp. 49_57.

[6] **T. Kamiya, S. Kusumoto, K. Inoue**, CCFinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654_670.

[7] **Z. Li, S. Lu, S. Myagmar, Y. Zhou**, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176_192.

[8] **T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, H. Iida,** SHINOBI: A real-time code clone detection tool for software maintenance, Technical Report: NAIST-IS-TR2007011, Graduate School of Information Science, Nara Institute of Science and Technology, 2008.

[9] **I. Baxter, A. Yahin, L. Moura, M. Anna,** Clone detection using abstract syntax trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, 1998, pp. 368_377.

[10] **L. Jiang, G. Misherghi, Z. Su, S. Glondu,** DECKARD: Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, 2007, pp. 96_105.

[11] **C. K. Roy, J. R. Cordy, and R. Koschke.** Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Sci. Comput. Program., 74(7):470–495, May 2009.

[12] **F. Li, J. Johnson and S. Yeung,** Convolutional Neural Networks for Visual Recognation class in Stanford University, 2018. [Online]. Available: http://cs231n.github.io/convolutional-networks/

[13] **Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy and Mohammad Mamun Mia**, "Towards a Big Data Curated Benchmark of Inter-Project Code Clones", In Proceedings of the Early Research Achievements track of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), 5 pp., Victoria, Canada, September 2014.

[14] **Y. LeCun,** "Lenet, convolutional neural networks," 2015. [Online]. Available: http: //yann.lecun.com/exdb/lenet/

[15] **URL-1** "ANTLR", http://www.antlr.org date retrived 15.12.2018

[16] **L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder**. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). 249–260. https://doi.org/10.1109/ICSME.2017.46

[17] **S. Ducasse, M. Rieger, S. Demeyer**, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, 1999, pp. 109_118.

[18] **N. Davey, P. Barson, S. Field, R. Frank**, The development of a software clone detector, International Journal of Applied Software Technology 1 (3/4) (1995) 219_236

[19] **R. Komondoor, S. Horwitz**, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, SAS 2001, 2001, pp. 40_56.

[20] **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, 2016

[21] **Bengio, Yoshua and Glorot, Xavier**. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of AISTATS 2010, volume 9, pp. 249– 256, May 2010

[22] **D. Kingma and J. Ba. Adam**: A method for stochastic optimization. ICLR, 2015

[23] **L. Prechelt, G. Malpohl, and M. Philippsen**. Finding plagiarisms among a set of programs with JPlag. J. of Universal Computer Science, 8(11), 2002.

[24] **A. Ahtiainen, S. Surakka, and M. Rahikainen.** Plaggie: Gnu-licensed source code plagiarism detection engine for Java exercises. In Baltic Sea '06: Proceedings of the 6th Baltic Sea Conference on Computing Education Research, pages 141–142, New York, NY, USA, 2006. ACM.

[25] **Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita,** "KClone: a proposed approach to fast precise code clone detection," in IWSC'09, 2009

[26] **A. Sheneamer and J. Kalita,** "Article: A survey of software clone detection techniques," International Journal of Computer Applications, vol. 137, no. 10, pp. 1–21, March 2016

[27] **B. Baker,** On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, 1995, pp. 86_95.

[28] **R. Wettel, R. Marinescu,** Archeology of code duplication: Recovering duplication chains from small duplication fragments, in: Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2005, 2005, p. 8.

[29] **K. Kontogiannis,** Evaluation experiments on the detection of programming patterns using software metrics, in: Proceedings of the 3rd Working Conference on Reverse Engineering, WCRE 1997, 1997, pp. 44_54

[30] **Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk**. 2015. Toward deep learning software repositories. In Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on. IEEE, 334–345.

[31] **Hoa Khanh Dam, Truyen Tran, and Trang Pham.** 2016. A deep language model for software code. arXiv preprint arXiv:1608.02715 (2016).

[32] **Chanchal K Roy and James R Cordy.** 2010. Near-miss function clones in open source software: an empirical study. Journal of Software: Evolution and Process 22, 3 (2010), 165–189.

[33] **Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia.** 2014. Towards a big data curated benchmark of inter-project code clones. In Proceedings of the International Conference on Software Maintenance and Evolution. IEEE, 476–480.

[34] **U. Bandara and G. Wijayarathna,** "A machine learning based tool for source code plagiarism detection," International Journal of Machine Learning and Computing, vol. 1, no. 4, 2011.

[35] **Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel**. Efficient plagiarism detection for large code repositories. Softw. Pract. Exper., 37(2):151–175, 2007

[36] **S. Narayanan and S. Simi**. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In Proc. of 7th International Conference on Computer Science Education, ICCSE '12, July 2012.

[37] **Murakami, Hiroaki, et al**. Folding repeated instructions for improving token-based code clone detection. Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on. IEEE, 2012.

[38] **H. Sajnani, J. Ossher, and C. Lopes,** "Parallel code clone detection using MapReduce", Proc. ICPC, 2012, pp. 261-262.

[39] **Y. LeCun**, "LeNet-5, convolutional neural networks,"November 2013.

[40] **van Doorn**, J. (2014). Analysis of Deep Convolutional Neural Network Architectures.

[41] **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. ACM, 2016.

[42] **I. Keivanloo, F. Zhang, and Y. Zou**. Threshold-free Code Clone Detection for a Large-scale Heterogeneous Java Repository. In SANER 2015.

CURRICULUM VITAE

Name-Surname **:** Harun Dişli

Place and Date of Birth **:** Kilis / 1990

E-Mail **:** disli15@itu.edu.tr

B.Sc. **:** 2014, İzmir Institute Of Technology

PROFESSIONAL EXPRERIENCE AND REWARDS:

- May 2017- Researcher, Tubitak
- November 2014- May 2017 Software Engineer, SFS