2019

A Topic Modeling approach for Code Clone Detection

Mohammed Salman Khan
University of North Florida, n01232330@ospreys.unf.edu

A TOPIC MODELING APPROACH FOR CODE CLONE DETECTION

by

Mohammed Salman Khan

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2019

The thesis "A Topic Modeling Approach for Code Clone Detection" submitted by Mohammed Salman Khan in partial fulfillment of the requirements for the degree of Master of Science in Computing and Information Sciences has been

Approved by the thesis committee: Date:

Dr. Sandeep Reddivari
Thesis Advisor and Committee Chairperson

Dr. Sanjay P. Ahuja

Dr. Karthikeyan Umapathy

Accepted for the School of Computing:

Dr. Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Dr. William Klostermeyer
Interim Dean of the College

Accepted for the University:

Dr. John Kantner
Dean of the Graduate School

ACKNOWLEGEMENTS

I would like to first thank my thesis advisor, Dr. Sandeep Reddivari, for his patience and helpful advice in the completion of my research work. This thesis would not have been possible without his direction and support. I would also like to thank my committee members, Dr. Sanjay P. Ahuja and Dr. Karthikeyan Umapathy, for their valuable suggestions. This thesis provided me a great learning experience on Software Engineering and its applications. It also taught me real patience in troubleshooting when something does not go as expected, a skill which is useful in the software field outside of academics. Last but not the least, special thanks to my parents for supporting me throughout my studies at University of North Florida. Without their support, I wouldn't have been able to finish my Master's degree successfully. This thesis is written using LaTeX.

CONTENTS

FIGURES

TABLES

ABSTRACT

In this thesis work, the potential benefits of Latent Dirichlet Allocation (LDA) as a technique for code clone detection has been described. The objective is to propose a language-independent, effective, and scalable approach for identifying similar code fragments in relatively large software systems. The main assumption is that the latent topic structure of software artifacts gives an indication of the presence of code clones. It can be hypothesized that artifacts with similar topic distributions contain duplicated code fragments and to prove this hypothesis, an experimental investigation using multiple datasets from various application domains were conducted. In addition, *CloneTM*, an LDA-based working prototype for code clone detection was developed. Results showed that, if calibrated properly, topic modeling can deliver a satisfactory performance in capturing different types of code clones, showing particularity good performance in detecting Type III clones. *CloneTM* also achieved levels of performance comparable to already existing practical tools that adopt different clone detection strategies.

CHAPTER 1

Introduction

As the size and complexity of a software system increases, it becomes important to develop a high-quality, cost-effective software. Also, dealing with large software systems is a great challenge for companies who maintain them. In general, the software engineering life cycle comprises of two major parts; initial software development, followed by active software maintenance. Unlike other industries where development costs are considered a major part of the lifetime cost of the project, in software development, it has been found that the maintenance is critical from cost perspective and might comprise up to 60% [63] or even 80% [3] of the overall cost of the project. Maintenance costs increase with poor software design, incomprehensible code, bad assumptions, sloppy practices, inflexible data structures, etc. There has been a lot of work [3, 64, 37, 49, 50] on improving process models, tool and language support, etc., but poor maintainability can always be traced back to poor code which is difficult to modify, understand, or more error-prone. Code cloning is one of the factors that affect software maintenance and make the process more difficult [71].

Code clones are similar code fragments that appear in a software system [20]. It is estimated that a typical mid-size industrial system contains up to 20% of duplicated code [3, 71, 33]. Clones are often produced by the copy-&-paste practice of programmers [49]. Rather than rewriting the working code fragments from scratch, programmers prefer to copy, and perhaps slightly modify the working

code that has already been tested before [64]. This practice amongst developers is common, especially while developing driver software for hardware devices of operating systems where algorithms are similar [18]. The main assumption is that simply making a copy of a working code is faster and is less likely to introduce new bugs, especially when a deadline is approaching [18]. An important point to consider is that in a software's post-development phase it might be difficult to state which fragment is original or cloned [35]. However, from a refactoring perspective, code clones are considered a major code smell [45, 20]. Code smell is any characteristic in the source code of a program that conceivably demonstrates a deeper problem providing a hint that something has gone wrong in the developed code [20].

Code clones significantly increase the maintenance cost and the error-proneness of the code [13, 2]. In particular, inconsistent changes to code duplicates can lead to unexpected behavior [33]. Therefore, clones must be kept in sync during maintenance [54, 79]. For instance, when a bug is fixed in an instance of a cloned code, all other duplicates must be altered as well [63]. In addition, clones decrease the modularity of the system and its level of encapsulation, as well as unnecessarily increase the size of the code, thus complicating future maintenance tasks and reducing understandability [64, 54]. The ability to change the representation of an abstraction (data structures, algorithms) without disturbing any of its subjects is the primary objective of encapsulation that helps us to keep our code modular. Modularity is closely tied to encapsulation that maps the encapsulated abstractions into real and physical models. Therefore, code clones must be refactored whenever detected [55, 58].

Table 1.1 offers insight to the terminologies that would be frequently encountered

Terminology	**Meaning**
Clone Class [58]	It refers to the set, containing clone pairs that have relative cloning among them.
Clone Pair [58]	It refers to a situation where there is a clone relation in a pair that is considered to be a fragment. It is labeled to be the pair code fragment with mutual or similar clones.
Code Clone Types [56]	Refers to the types of cloning and there are four types of code clones: Type I, Type II, Type III, and the Type IV. More details about the types of code clones are presented in Section 2.
Code Fragment [56]	Code Fragment is considered as the code line sequence with multiple similarities, from a source code, among them. Code Fragment(s) can be present in the source code as a comment or the main function.

Table 1.1: Key Terminologies in Code Clone

in this thesis work.

Software refactoring or refactoring is one of the popular and promising techniques to eliminate the problem of Code Clones [20]. Refactoring is the sequence of code changes which improves the internal structure of the code (quality of design) without changing the external structure of the code (behavior of software). In general, refactoring is a systematic way to clean up code that minimizes the chances of introducing new bugs into source code. Refactoring, when applied to code clones, can improve the efficiency, performance, reuse, and maintainability of the software programs. Popular IDEs (Integrated Development Environments) such as Eclipse, NetBeans, and Visual Studio provide automated refactoring support by their respective refactoring engines. Table 1.2 presents the popular software editors and IDEs that support software refactoring.

Refactoring Tools	**Supporting Languages**
IntelliJ IDEA	Java, JSP, CSS, HTML, XML and JavaScript
Eclipse	Primarily: Java To lesser extent: C++, PHP, JavaScript, R, C and COBAL
NetBeans	Java
CloneDR	C#
Visual Studio	.NET, C++, C#, J#, VB and ASP .Net
XCode	Objective C
AppCode	Objective C, C++

Table 1.2: Software tools and IDEs that support refactoring

A plethora of clone detection tools have been proposed in the literature [5, 58]. Such tools usually support a large variety of programming languages, and adopt different clone detection strategies, at different levels of complexity, designed to target various types of clones. However, despite these major advances, the usage of clone detection tools is still not pervasive in the industry [13, 16]. In general, most of these tools are still far from achieving optimal accuracy. This requires developers working with such tools to manually classify and verify the detected candidate clones [5, 58], a process that is often described as time-consuming and error-prone [51]. In addition, there is still a lack of adequate support for large-scale systems, where clones are likely to spread over several code modules [37, 79].

To address these issues, this thesis proposes a novel approach, based on topic modeling, to facilitate a more accurate, language-independent, and scalable clone detection process. In particular, we experiment with Latent Dirichlet Allocation (LDA), the most commonly used technique for topic modeling in Natural Languages Processing (NLP) [9]. LDA is a probabilistic approach for estimating a topic distribution over a text corpus [9]. The main assumption is that documents

in a text collection are generated using a certain generative model as random mixture over latent topics. Therefore, the set of topics that are likely to have generated the documents in the corpus can be inferred from the corpus by simply inverting the corpus's generative model [30].

The success of LDA in NLP tasks [27, 73], in addition to its scalability [9], and its ability to work with multiple programming languages [38], have motivated a large body of research to investigate its potential applications in several software engineering activities. Such activities include software testing [67], modeling evolution in software repositories [15, 65, 66], automated traceability [28], managing software defects [70], and mining semantic topics from source code [31]. Following this line of work, this thesis, investigated the potential benefits of LDA as a clone detection technique. The main conjuncture is that the probabilistic distribution of topics over a certain code base gives an indication of the presence of code clones. In particular, the contributions in this thesis work can be described as follows:

- An effective, language-independent, and scalable approach for detecting code clones based on topic modeling.
- An experimental benchmark for calibrating LDA parameters and evaluating its performance in detecting various types of code clones.
- A working prototype with visualization support and friendly user interface that implements our main findings in this research.

The rest of the thesis is organized as follows. Chapter 2 briefly introduces topic modeling and code clones. Chapter 3 presents the previous research works done in the field of Code clone detection and the tools developed to detect code clones. Chapter 4 describes our main research assumptions, experimental framework, and

research methodology. Chapter 5 presents our experimental analysis and results. Chapter 6 discusses the benefits and the potential limitations of our approach. Chapter 7 discusses threats to our study's validity. And finally, Chapter 8 concludes the thesis and suggests venues of future work.

CHAPTER 2

Background

This chapter provides (i) background information about LDA, (ii) background information about Code Clones, (iii) LDA applications in Software Engineering, and (iv) discussion about related approaches aiming at Code Clone detections.

2.1 Code Clones

Code clone can be defined as reusing a code fragment by copying it from one section of the software and then pasting with or without minor modifications or adaptations into another section of the software. An example of code clone is seen in Figure 2.1.

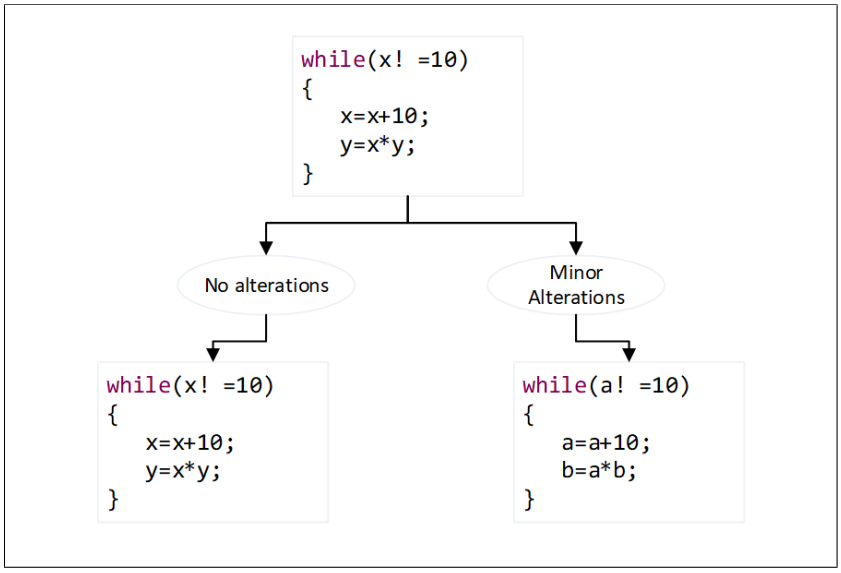
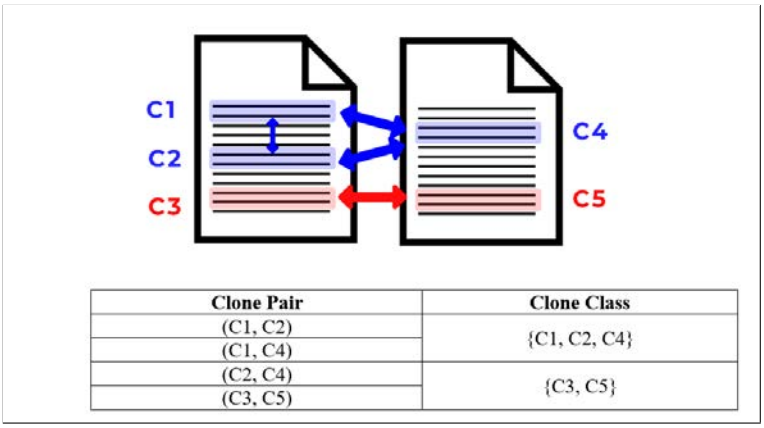


Figure 2.1: An example of a Code Clone

A clone relation is defined as an equivalence relation between two code segments forming a clone pair or clone classes [56]. An equivalence relation exists if it agrees to hold all reflexive, symmetric and transitive relations. A clone relation exists between two different code segments if (and only if) they have same sequences, like original strings, it may be strings with white spaces, sequences of token type and transformed token sequence. For a given clone relation, clone pair is defined as a pair of code portions matching code segments, and an equivalence class of clone relation is called clone class. That is, each code segment in a clone class forms a clone pair with other code segments of that class if the relation holds between any pair of code portions. Figure 2.2 shows the concept of clone pair and clone class.



Clone Pair	Clone Class
(C1, C2)	{C1, C2, C4}
(C1, C4)	
(C2, C4)	
(C3, C5)	{C3, C5}

Figure 2.2: A portray of clone pair and clone class

2.1.1 Code Clone Types

Four types of clones can be identified based on the notion of similarity considered between code fragments [46, 58, 56]. These clones based on their similarities are categorized as textual (type I, II, III) and functional (type IV). In detail, these

types are:

Type I: Exact clones in which the same fragment of code is copied without modification in its semantic or syntactic structure (except for spacing and comments) as shown in Figure 2.3.

Code Segment 1	Code Segment 2

`int a=0, b=1, c=2;`
`while (c>0)`
`{`
`  a= a+b;`
`  b= b*c;`
`  c= c-1;`
`}`
	`int a=0, b=1, c=2;`
	`while (c>0)`
	`{`
	`  a= a+b;`
	`  b= b*c;`
	`  c= c-1;`
	`}`

Figure 2.3: Type -1 (or) exact clones

Type II: Clones that are syntactically identical fragments except for slight variations, such as different identifiers names, literals, types, or spacing. Figure 2.4 demonstrates an example for Type II clones.

Code Segment 1	Code Segment 2

`int a=0, b=1, c=2;`
`while (c>0)`
`{`
`  a= a+b;`
`  b= b*c;`
`  c= c-1;`
`}`
	`int a=0, b=1, d=2;`
	`while (d>0)`
	`{`
	`  a= a+b;`
	`  b= b*c;`
	`  d= d-1;`
	`}`

Figure 2.4: Type-2 (or) renamed clones

Type III: Clones that have been slightly modified by addding, removing, or reordering statements, in addition to Type I and Type II modifications as shown in

Figure 2.5 are called Type III or near miss clones.

Code Segment 1	Code Segment 2

`int a=0, b=1, c=2;`
`while (c>0)`
`{`
`  a= a+b;`
`  b= b*c;`
`  c= c-1;`
`}`
	`int a=0, b=1, c=2, e=5;`
	`while (c>0)`
	`{`
	`  a= a+b;`
	`  b= b*c;`
	`  e= a+b;`
	`  c= c-1;`
	`}`

Figure 2.5: Type-3 (or) near miss clones

Type IV: Functional clones which refer to code fragments that perform similar operations, but their syntactic and semantic structures are different. Figure 2.6 shows an example for Type IV clones.

Code Segment 1	Code Segment 2

`int a=0, b=1, c=2;`
`while (c>0)`
`{`
`  a= a+b;`
`  b= b*c;`
`  c= c-1;`
`}`
	`int a=0, b=1, c=2, f=5;`
	`while (c>0)`
	`{`
	`  a= a+b;`
	`  b= b*c;`
	`  f= a+b;`
	`  c= c-1;`
	`}`

Figure 2.6: Type-4 (or) semantic clones

Detecting different types of clones requires different levels of sophistication. While Type I and Type II can be relatively easy to detect using lexical-based analysis, other types (Type III and Type IV) require a higher level of complexity to match operationally identical code fragments.

2.1.2 Reasons for Cloning

There are various reasons for code cloning. The topmost reason is for code reuse, where the developer finds it easier to copy the working code from one software system and paste it to another rather than developing the code from scratch. Another reason for code cloning is when the developers are asked to complete a project within a strict time frame. However, the most important of all is when the developer may duplicate a particular block of code because of lacking an understanding of the code, but they have the code intended to do the required task's functionality and so they reuse them. Figure 2.7, taken from Roy and Cordy [58], gives a tree diagram for the factors responsible for code cloning. The important factors from all the reasons listed from a developers' perspective to use code clone are discussed below.

Reuse: The primary factor responsible for code duplication is the code reuse approach. It is the easiest way, saves time and is very popular amongst developer communities. It is simple for a developer to copy and paste the already existing code with or without some changes.

Developer's Inadequate Knowledge: It happens more than usual that when developers do not have sufficient knowledge regarding the problem and the domain that they are supposed to solve, they take the existing solution of the problem, transform the solution according to the requirements and present a new solution.

Time Limit: After reuse, the main reason, which is responsible for the usage and existence of code clones is the time constraint. When the developers are assigned

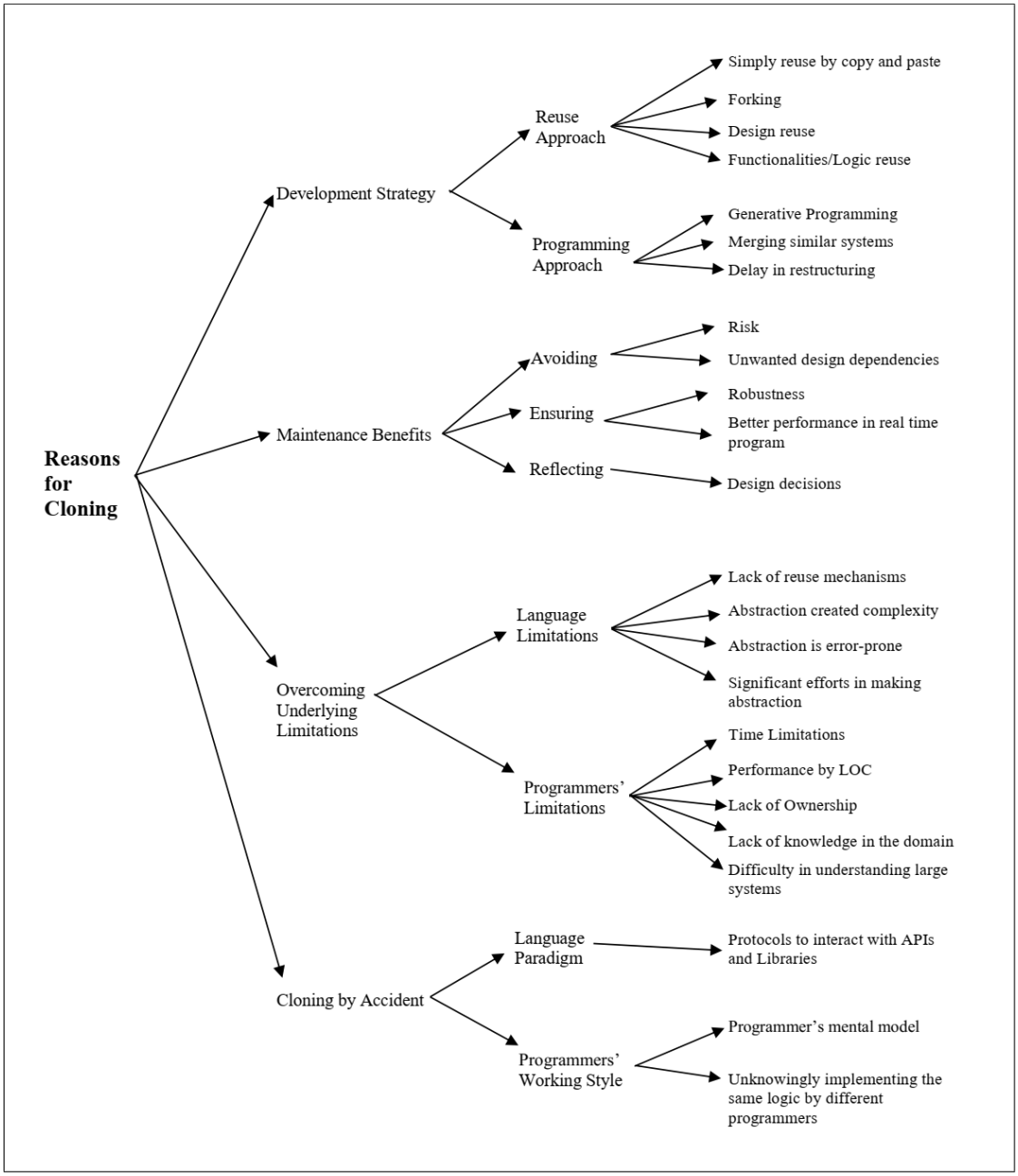


Figure 2.7: Reason for Cloning

a project and are supposed to complete the project within a short amount of time, developers clone their codes from existing software components and use in their current project according to the project requirements to get the project done within

the required time.

By Accident: Code cloning may be performed accidentally [56]. There may be a case that for a problem, two software developers may come with the same solution. While technically these multiple copies are not clones, but for clone detection tools, they are identified as code clones.

Measuring Developer's Performance using LOC: There are times when a developer's productivity and efficiency is computed by the number of Lines of Code (LOC) they write within a given amount of time. In such scenarios, developer's primary focus is to increase the quantity of their code by increasing the number of LOC. To do so, developers use copy-paste approach on the existing code, multiple times, to increase the value of LOC.

Integration of Two Identical Systems: In a larger extensive software system, developers may integrate two different software systems, performing similar tasks to create a new system. Even when different team members have developed these systems, the integrated system may result in producing clones.

Using Generative Programming: In generative programming, the same template is used to produce similar logic. This approach generates code that may result in a large amount of code clones.

2.1.3 Advantages of Code Clones

If needed, clones are introduced in the software systems mostly after refactoring, to obtain several maintenance benefits. Some of the advantages that clones provide are discussed below.

- Risk in Writing Fresh Code: When a developer prefers to avoid risks of writing new code, the developer ends up using existing code. There are chances of errors and bugs in writing new code, but the existing code is already tested [46]. Cordy [59] reports that in a financial software system, although financial products do not often change, especially within the same financial institution, clones do occur frequently. Mainly because of the frequent updates and enhancements that are needed to be performed on the existing system as to support similar, but new functionality. In a scenario like this, the developer is often asked to reuse existing block of code and adapt the code according to the requirements of the new product. This is mostly because of the high risks (software errors in an organization can be very costly) involved with the introduction of software errors found in new fragments of code while in the case of existing code, the code is already well tested.

- Clean and Understandable Software Architecture: It is intended to introduce clones into the system as it will promote clean and understandable software architecture [30].

- Maintenance Speedup: In a multi-cloned system, two cloned code fragments are independent of each other regarding both semantics and syntax and can evolve at a different pace in isolation without affecting one another and testing can also be performed and required to the modified fragments. Main-

taining cloned fragments in a system may speed up maintenance, especially when automated regression tests are absent [37].

- Ensuring Robustness of Life-critical Systems: Redundancy or Cloning is intentionally incorporated in the design of life-critical systems. As life-critical systems have to maintain safety features and needs to work perfectly, and should not fail, multiple teams work on the same functionality as to reduce the chances of errors.

- High Cost of Function Calls in Real-time Programs: In real-time programs, function calls may seem to be too costly. Inlined functions run a little faster than the normal functions as function-calling-overheads are saved, however, there is a memory penalty. If a function is inlined 10 times, there will be 10 copies of the function inserted into the code. Without inline functions, the compiler decides which functions to inline automatically, and if it does not, then this needs to be done by the developer manually to write the code that would have gone in the function at what would have been the function's call site and consequently, there will be clones.

2.1.4 Disadvantages of Code Clones

It may be easy to develop software systems by applying code cloning, but code cloning may be critical for both maintenance and quality of software system [63]. Problems caused due to cloning are listed [46]. Below is a brief discussion of these problems from developer's perspective.

- Increased Probability of Defects: If the original code contains a bug, then its

clone will undoubtedly contain the same bug [20]. Hence, duplication of the code may increase the probability of a bug in the system.

- Increased Resource Requirements: With the increase of code clones, the system size may also increase and the compilation time in addition to memory requirements for the system may also get increased, this may be a factor resulting in an expensive software and hardware upgrades.

- Increase Maintenance Effort and Cost: During the maintenance process, code cloning multiplies the effort required for a software system [54]. During the maintenance phase if an error or bug is found in one fragment, then all of its corresponding clones should be examined first for presence of the same error or bug and then the error has to be solved, resulting in an increased maintenance effort.

- Increased Chances of Bad Design: With the increase of code clones, the system size may also increase and the compilation time in addition to memory requirements for the system may also be increased. This may be a factor resulting in expensive software and hardware upgrades.

2.1.5 Code Clone Detection

In general, the methods for detecting different types of clones in software systems can be classified into several categories as listed below:

- Textual Analysis: Text-based code clone detection techniques are the traditional and easiest way of detecting clones. Such methods analyze the lexical

structure of source code, looking for similar textual patterns that might indicate cloning. In particular, the string-based detection techniques are often language independent [64] and are used for detecting either identical sections of code (Type I) or clones with slight modifications, such as renamed variables (Type II). In this technique, strings are compared line by line without any code transformation. However, lately, some text-based techniques involve transforming code by removing comments and whitespaces from the code. Since these techniques do not require an examination of code in terms of semantical or syntactical, the performance is fastest in case of these techniques compared to other techniques. Dup is an example of a tool that uses textual analysis for clone detection [3].

- Token Based Technique: In token-based clone detection techniques, at first, the code is transformed into different tokens and then compared against matching tokens in the series. In general, a lexical analyzer is used to transform the code into tokens [3]. As each code is tokenized, this technique performs slower as compared to text-based technique, as much amount of time is consumed in tokenization. CCFinder is an example of tool that uses token analysis for clone detection [71].

- Code Metrics Analysis: Such tools identify duplicates and semi-duplicates in source code by comparing pieces of code based on various metrics extracted from source code [33]. In particular, certain metrics are used as signatures that classify and represent code fragments [4]. The underlying assumption is that if two or more code fragments are clones, then they share a number of characteristics that can be effectively captured by these metrics. Therefore, matching signatures indicate potential cloning [41]. As compared to other clone detection techniques, code metrics methods use less time to detect

code, are less complex and are easy to use. Covet is a tool that follows the metrics-based clone detection technique [33].

- Parsing Techniques: Such techniques work by matching parts of the abstract syntax trees (AST) of source code. In particular, similar subtrees in the system's AST indicate cloning [55, 52]. In addition to detecting exact clones, such techniques can also identify Type II clones, given that variables' names and literal values are ignored when constructing the system's AST. In general, when this technique is applied on a very large source code, it takes a lot of time. CloneDR is a tool that uses the AST-based approach for clone detection [31].

- Graph Analysis: Under this approach, similar code fragments in a program are identified by matching similar subgraphs in a program dependency graph (PDG). A PDG represents the structure of a program and its data flow. Such technique enhances the AST parsing technique by considering not only the syntactic structure of programs but also the data flow [43]. This particular approach can capture clones in which matching code statements have been reordered, and clones that are intertwined with each other [40]. Duplix is a tool that applies PDG-based clone detection [43].

- Hybrid Technique: This technique can be diverse in its operations. Here, programmers typically bring two or more of the aforementioned techniques together to detect clone codes. The processes can be sophisticated, such that they are done in stages, and an entire technique would be the first stage, and then another would be the following step. Kosche et al. postulated an approach that was eventually worked on [55]. The team compared the tokens of Abstract Syntax Tree (AST) nodes with making a direct comparison between each AST nodes. Tairas et al. also developed a method that could

Technique	Clone Type	Efficiency	Portability	Integrality
Text Based [3, 64]	Type - I	High	High	Low
Token Based [3, 71]	Type – I, II	Low	Medium	High
AST Based [31, 55, 52]	Type – I, II, III	High	Low	Low
PDG Based [40, 43]	Type – I, II, III	High	Low	Medium
Metric Based [40, 33]	Type – I, II, III	High	Depends on Metrics used	Medium

Table 2.1: Abilities and Properties of clone detection techniques

detect existing, functional cones in a software; it combines suffix tree and AST based techniques [57].

Table 2.1 gives details for efficiency, portability and integrality for different techniques based on detected clone types.

2.1.6 Benefits of Code Clone Detection

There are advantages associated with code clones if they are detected early. In addition to improving the quality of code through refactoring process of duplicated code, code clones can be used for:

- Detection of Library Candidates: If a functionality needs to be used at different locations in a software system, then the code's usability is demonstrated when the code block segment is duplicated and reused many times [58]. This code segment can be made a library candidate and used across different locations in the system.
- Helps in Reducing Code Size: In a software system, if the cloned code is replaced by function calls to a generic code segment that performs the same

functionality as of the code clones, then it results in reducing the size and complexity of the software system [56]. Also, it improves the readability and maintainability of the code.

- Helps in Finding Usage Patterns: It is easy to find the usage patterns for a piece of code when all cloned segments of a block of code are detected.

- Better Understanding of Code: If working of a cloned segment is understood, then it is easy to understand the working of all the duplicated code segments of that code clone.

- Detects Malicious Software: Code clone detection technique can be used to detect malicious software codes. With the different classification of the clones, it may be easy to match the malicious piece of code with other malicious code.

2.2 Topic Modeling

In recent years, we have been witnessing exponential growth in data volumes. This situation poses a greater challenge on how to extract hidden knowledge and relations from these data. Topic model is a probabilistic generative model that has been used widely in the field of Computer Science applications focused specifically on text mining and information retrieval. Topic models originated from the field of natural language processing (NLP) have been receiving much attention because of their interpretability. Since these models were first introduced, they have gained a lot of attention among researchers from different research fields. In addition to

text mining, topic modeling has been successful applied in the fields of computer vision, social networks and population genetics.

The basis for the development of the topic model is latent semantic indexing (LSI) introduced by Deerwester et al. in 1990 [62]. However, LSI is not a probabilistic model and hence cannot be considered as an authentic topic model. In 2001, based on LSI, a probabilistic Latent Semantic Ananlysis (PLSA) was proposed by Hoffman [30]. After PLSA, David Blei in 2003 proposed Latent Dirichelt Allocation (LDA) which is an extension of PLSA and a more complete probabilistic generative model [9]. All topic models have been introduced initially for analyzing texts for unsupervised topic discovery in a corpus of documents as shown in Figure 2.9.

To better understand how to use a topic model, we first describe the ideas behind topic modeling as shown below in Figure 2.8. Assuming there are N documents, W words and K topics, Figure 2.8 illustrates the important steps of topic modeling including the bag of words (BoW), a model training, and a model output.

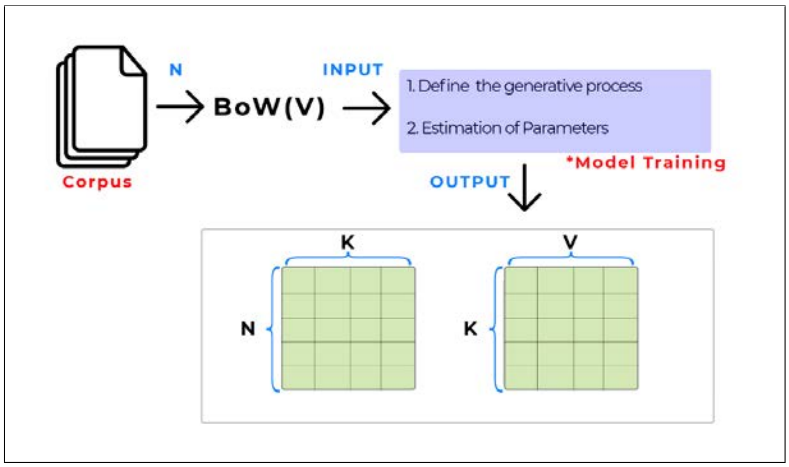


Figure 2.8: Topic modelling with a bag of words

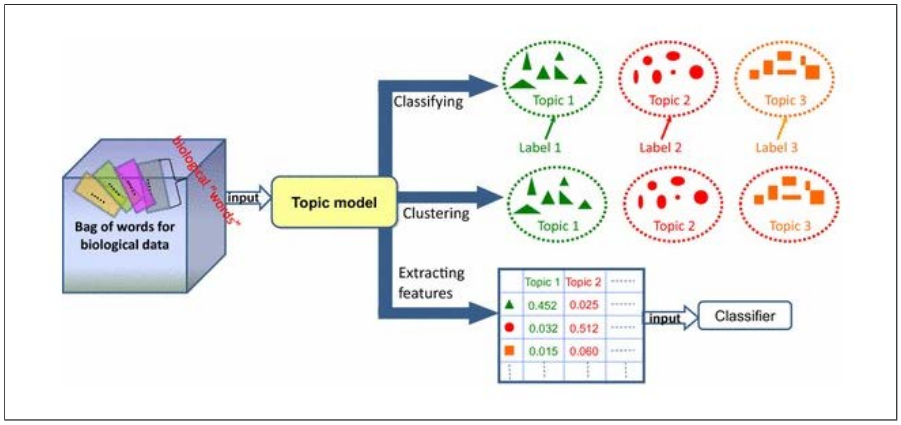


Figure 2.9: An illustration of topic modeling [9]

2.2.1 Bag-of-Words (BoW)

In NLP, a document is actually represented by a word-document matrix, known as Bag-of-Words. An example of BoW is shown in the Table 2.2.

Table 2.2 shows the BoW for a corpus of five documents ($d_1 - d_5$) and four words (Dog, Paw, Fur, and Tail). Value w_{ij} in the matrix represents the frequency of word i in the document j. For example, $w_{3,2} = 3$ means that the frequency of the word "fur" in the document d_2 is 3. The number of words in a corpus is fixed and the collection of these words combined constitutes a vocabulary. In a nutshell, the corpus is represented by the BoW it contains. So, if we want to process a programming data rather than a corpus, we must represent the data as a BoW.

	d_1	d_2	d_3	d_4	d_5
Dog	5	0	2	0	0
Paw	0	7	0	0	0
Fur	1	3	0	0	0
Tail	0	0	5	7	0

Table 2.2: Bag-of-words, a word-documented matrix

After BoW is constructed, it serves as the input to the next step in topic modeling. As assumed N documents with W words in a corpus, BoW of this corpus will be a $N \times W$ matrix. The documents in a corpus are independent as there is no relation among the documents and we can deduce that while representing BoW in a document, the order of words does not matter. Implying, that the words in a document are exchangeable and this is the basic assumption of a topic model. While these assumptions are made available in both LDA and PLSA, there are other variants of topic modes where this basic assumption is relaxed.

2.2.2 Model Training

Bag-of-Words contain only the words of the original texts and the number of words(word space dimensionality) may be very large. In contrast, humans are more concerned about the theme of the document rather than the individual words associated with it. Hence, the aim of topic modeling is to discover the underlying theme of the corpus by analyzing the words of the original texts. These themes are called "topics". Classic topic models are usually developed using unsupervised algorithms (without prior labeling of the documents) and the "topics" are discov-

Topics	Cancer	Protein	Computation
Words	Tumor	Cell	Model
	Cancer	Protein	Algorithm
	Diseases	DNA	Data
	Death	Gene	Computer
	Medical	Polypeptide	Mathematical

Table 2.3: An example article from a medical corpus.

ered during the model training processes. Topic, in topic modeling, is considered a probability distribution over a fixed vocabulary.

Table 2.3 illustrates the top five most frequent words from three different topics that were discovered in a corpus [9]. Table 2.3 shows each word sorted in descending order based on the probabilities. The most frequent top five words reflect related concepts of each "topic": "Topic 1" is about Cancer, "Topic 2" is about protein, and "Topic 3" is about computation. As each topic is a collection of words in a vocabulary, similarly, in topic modeling, each document is a collection of topics. Above all, the main idea behind topic modeling is that the documents show multiple topics and hence to discover the word distribution over each topic, and topic distribution over each document represented by $K \times W$ matrix and $N \times K$ matrix respectively, where K is the number of topics.

In the generative process, documents are considered as a mixture of probabilistic topics and topics as a mixture of words. The completeness of the generative process for documents is achieved by considering Dirichlet priors on the document distributions over topics and on the topic distributions over words. LDA or PLSA can be selected for the generative process as these are relatively simple topic models. LDA was used for this research as in recent years, more or less, most topic models

were related to LDA. As the goal of topic modeling is to automatically discover the underlying topics in a collection of documents. The documents are themselves examined, whereas the topic structure (topics, per-document topic distributions, and per-document per-word topic assignments) is hidden. The topic modeling task of how to use documents under the study to infer hidden topic structure can be thought of as a "reversal" of the generative process and hence the task of parameter estimation can be considered as, given a corpus, estimating the posterior distribution of hidden variables and unknown model parameters. If PLSA, LDA or any other existing topic models are used for model training, then the inference algorithm of parameters is readily available and only the parameters need to be initialized.

2.2.3 Model Output

The output of the model includes two matrices: word distribution over each topic represented by a $K \times W$ matrix, and topic distribution over each document represented by a $N \times K$ matrix. If the number of topics were specified as K, then K topics could be obtained using a model training and the word term space of documents is transformed into "topic" space. It is obvious that the topic space is smaller than the word space ($K < W$) and examining a document at topic level instead of at word level is useful for identifying meaningful structure of the document. The output of a topic model is a cluster of corpus as documents with similar topic probability distribution can be grouped together. In addition to topic model used for clustering, topic model can interpret clustering results by word probability distributions over different topics rather than a single topic. These characteristics

may be crucial in various applications.

2.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) was first introduced by David Blei et al. as a statistical model for automatically discovering topics in large corpora of text documents [9]. The main assumption is that documents in a collection are generated using a mixture of latent topics, where a topic is a dominant theme that describes a coherent concept of the corpus's subject matter. LDA as a topic model intends on detecting the thematic information of documents and can be adapted as to find thematic patterns in generic data, social networks, images, and bioinformatics.

2.3.1 LDA Introduction

The primary objective of LDA is to discover a smaller description of collection's individuals, keeping the necessary statistical dependencies to facilitate efficient processing of the data but at the same time reducing the extensive initial information into a smaller interpretable space. Considering text corpora as our collection, Latent Dirichlet Allocation assumes that each document of the collection exhibits multiple topics, with the topic probabilities providing an explicit representation of the corpus. LDA assumes that all documents in the collection share the same set of topics, but each document exhibits those topics with different proportion. We can better understand LDA's objective in Blei's representation of the hidden

topics in a scientific document in Figure 2.10 below.

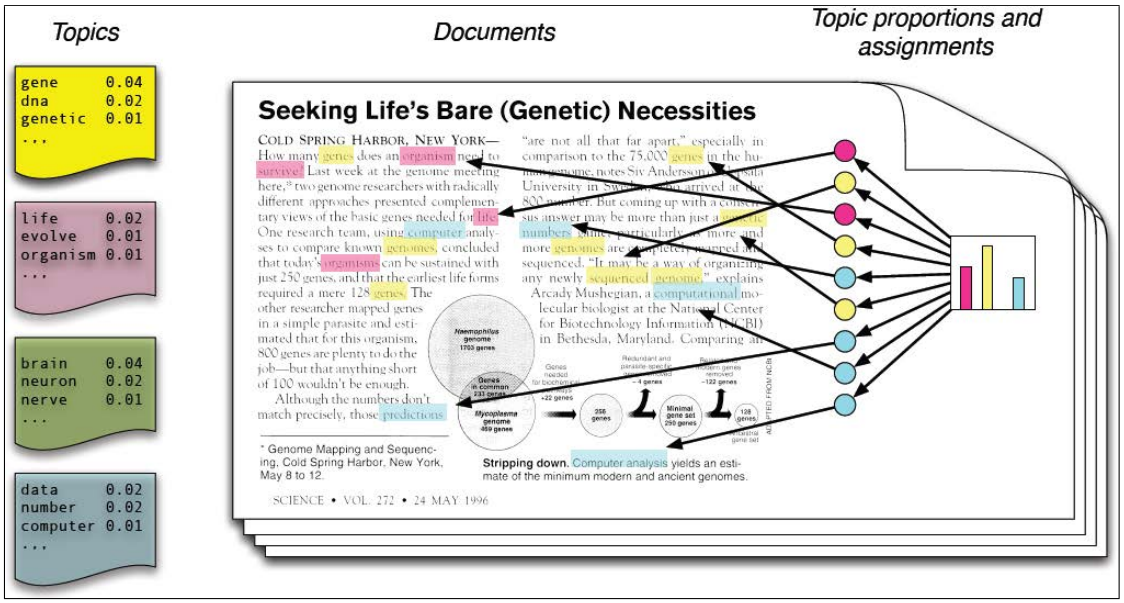


Figure 2.10: An illustration of hidden topics in a document [9]

In Figure 2.10, words from different topics are highlighted with different colors. For example, yellow represents a topic with a bag-of-words about genetics, pink words about evolutionary life, green words about the brain and blue words about computational analysis, declaring the nature of the article and the mixture of multiple topics which is the intuition of the model. LDA as a generative probabilistic model treats data as observations that arrive from a generative probabilistic process that includes hidden variables. These latent or hidden variables reflect the thematic structure of the collection, the topics. The histogram at the right in Figure 2.10 represents the document's topic distribution. Figure 2.10 also shows, the word mixture of each topic with each word's probability exhibition in every topic.

This topic model procedure assumes that the order of words does not necessarily

matter. It is not concerned about the documents becoming unreadable with the words shuffled, but it can discover the thematically coherent terms, which is a bag-of-words (BoW). Same words can be observed in multiple topics but with different probabilities and every topic contains a probability for every word. It can be stated that though one word might not have a high probability in one topic, it can indeed have a higher probability in another.

In Figure 2.10 above, one can notice that documents in LDA are modeled by assuming the known topic mixture of all documents and the word mixture of all the topics. Having this knowledge, the model assumes that a document is empty at first, it then chooses a topic from the topic mixture followed by choosing a word from the word mixture of that topic. This process is repeatedly followed until the document is complete. This exact procedure is followed for every document in the corpus. Given some hidden variables, this process produces the generative character of the LDA model, namely its attribute of generating observations.

2.3.2 Generative Model

In software systems analysis, the utilization of LDA is based on the assumption that a software system can be analogues to a text corpus, where software artifacts represent documents of the corpus and code identifiers represent words. Such textual content can be analyzed to produce a unique perspective of the hidden themes in the system, revealing otherwise unseen relations between its artifacts [9]. LDA as a probabilistic graphical model shows the statistical assumptions, behind the generative process described above, which relies on Bayesian Networks. In

Figure 2.11, the rectangle box plates represent loops. The outer plate represents the documents, while the inner plate represents the repeated choice of topics and words within a document.

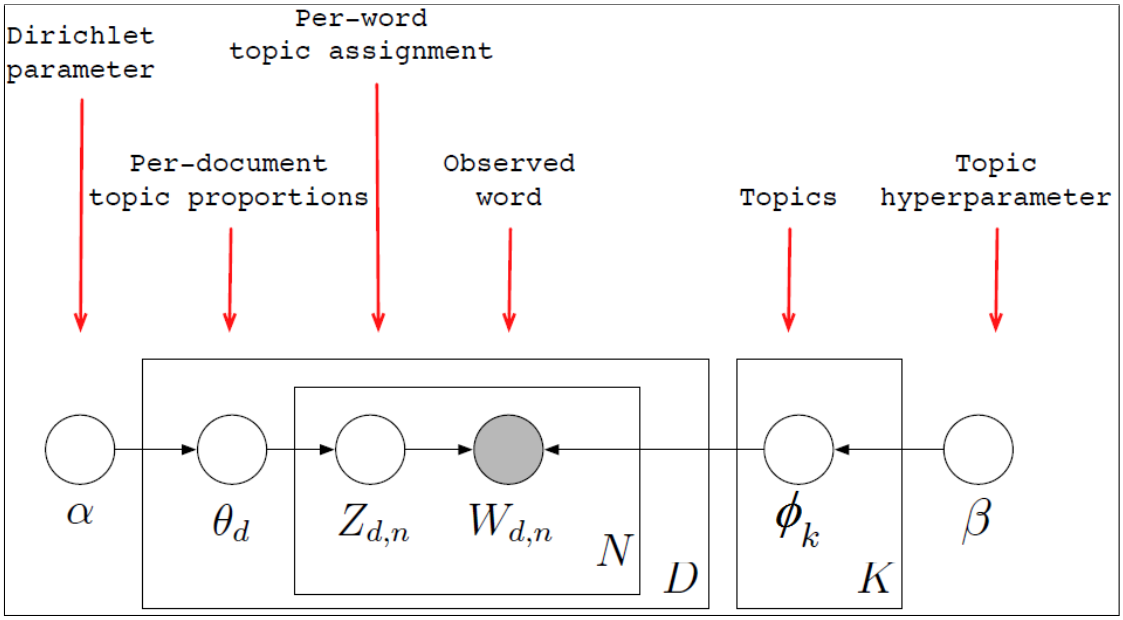


Figure 2.11: LDA Graphical model [9]

Formally, a topic model can be described as a hierarchical Bayesian model that associates with each document d_j in the collection D, a probability distribution over a number of topics T. In particular, each document d_j in the collection ($d_j \in D$) is modeled as a finite mixture over T drawn from a Dirichlet distribution with parameter α, such that each d_j is associated with each ($t_i \in T$) by a probability distribution of θ_{ji}. On the other hand, each topic t_i in the identified latent topics ($t_i \in$ T) is modeled as a multidimensional probability distribution, drawn from a Dirichlet distribution β, over the set of unique words in the corpus (W), where the likelihood of a word from the corpus ($w_k \in W$) to be assigned to a certain topic t_i is given by the parameter $\emptyset_{ki}$. LDA takes the documents collection D, the number

of topics K, and α and β as inputs. LDA parameters α, β, and K are known as hyper-parameters (i.e. a set of parameters that effect the topic model generated). In the context of LDA, these parameters have a high effect on the implementation as follows:

- K is the number of topics to be identified from the data. It can be considered as an equivalent to the number of clusters in a clustering algorithm.
- α influences the topic distribution per document. A higher α value results in a better uniform distribution for each document.
- β affects the words' distribution per topic assigned. A higher value results in a more uniform distribution of words per topic.

Each document in the corpus is represented as a bag-of-words d = $< w_1,w_2,...,w_n >$. Since these words are observed data, Bayesian probability can be used to invert the generative model and automatically learn $\emptyset$ values for each topic t_i, and θ values for each document d_j. In particular, using algorithms such as Gibbs sampling, an LDA model can be extracted [9]. This model contains for each t_i the matrix $\emptyset = \emptyset_1,\emptyset_2,\emptyset_3,\emptyset_4,\emptyset_5 ...,\emptyset_n$ representing the distribution of t_i over the set of words $< w_1,w_2,...,w_n >$ and for each document d_j the matrix $\theta = \theta_1, \theta_2, . . ., \theta_n$ representing the distribution of d_j over the set of topics $< t, t, . . ., t_n >$. Several methods have been proposed in the literature to approximate near-optimal combinations of LDA parameters (α, β, K) in software systems. Such methods can be categorized as follows:

- Manual Approach: Under this approach, the values of (α, β, K) are specified manually. For instance, in their assisted model, Maskeri et al. concluded that determining K manually by a domain expert or based on the folder structure of the system is more adequate for LDA applications in source code [22].

Similarly, in their study of statistical debugging, Andrzejewski et al. chose the number of topics for each program according to domain expert advice [8].

- Experimentally Determined: in the experimental approach, LDA parameters are tuned experimentally until a configuration that achieves acceptable results over a certain quality measure is reached. For instance, Asuncion et al. experimented with different values of K in automated tracing, with the objective of minimizing the number of incorrectly retrieved traceability links (false positives) [28].

- Fully Automated: under this approach, LDA settings are discovered automatically. For instance, Grant and Cordy proposed a method to automatically estimate the optimal number of latent topics in source code by consulting the file and folder structure of source code and the location of each method in that structure and their conceptual similarity [26]. In a more recent paper, Panichella et al. proposed a fully automated approach based on genetic algorithms to calibrate LDA parameters [1]. The fully automated approach often achieves acceptable approximations in software engineering tasks. However, it adds another level of sophistication to the process.

In this research, LDA was calibrated experimentally. In particular, multiple settings of LDA parameters were experimented to approximate near optimal values that achieve the best performance in the domain of code clone detection. Chapter 4 describes the calibration procedure in greater detail.

CHAPTER 3

Related Work

A growing body of work on code cloning has emerged in recent years, from the conception of cloning, to variations of codes as a form of variation, and the software development to detect these code clones. This chapter offers insight into the previous works, methods, propositions and deduction by computation done by various computer engineers, scientists and software developer over the years.

Any code clone detection technique is always a two phase process [44]. These phases are the transformational and comparison phases. In the first phase, the source text is converted into an internal format that can be easily worked with the comparison algorithm. In the second phase, which is the comparison phase, the actual code matches are detected. It is reasonable to classify detection techniques according to their detection format [44].

3.1 Text-based detection

Text-based techniques or string-based techniques were considered to be the earliest to be developed and provided easiest method of clone detection. The program is considered a sequence of statements. The detection works for a direct replica of a command statement. The text-based technique uses basic string transforma-

tion and comparison algorithms which makes them independent of programming language as the code is compared line by line and lines are in the form of simple strings. Detection techniques in this category differ in the underlying text comparison algorithm. Comparing calculated signatures per line is one of the possible ways of identifying matching strings. Line matching can be categorized into two variations based on string manipulations techniques. They use a small fraction or nothing, from the original source code, before they are compiled and made operative. In text-based detection, the source code is directly compared against the potential cloned code during the clone detection process. Examples of such techniques are seen in NICAD [59] and Simian 1 [29]. Some of the popular research in text-based detection are listed below.

Baker et al. presented a non-graphical user interface tool named Dup that locates instances of duplication or near-duplication code in a software system [3]. Dup gives feedback on text identical sections with the exception of variable name substitutions. The approach is both text-based and line-based while ignoring comments and white spaces to find maximum number of sections of code over an ideal threshold length. Dup has proven to be both fast and efficient at locating duplication of code as it can process a million lines of code within seven minutes using one R3000 40MHZ processor. However, Dup ignores indentation in a program added by the editor and cannot generate procedures with parameters from the input.

Ducasse et al. presented a language-independent approach for detecting duplicated code in a tool called DUPLOC, running under VISUALWORKS 2.5 [64]. DUPLOC is based on simple line-based string matching by visual presentation of the duplicated code, and detailed textual reports from which overview data can be synthesized. Using DUPLOC, clones can be easily identified between several files,

within the same file, cloned files and evolution files. Additionally, data generated from DUPLOC were useful in re-engineering tasks and software maintenance.

3.2 Token-based detection

Token-based detection is also called Lexical Approach, that converts the source code into tokens by utilizing compiler style lexical analyses [71]. The generated token sequence is then used as an overlay scan, using the duplications that are not required from the source code to form the new clone. The method has proven more effective for variation detection in the code with the use of renaming, formatting, and spacing of the source code.

Kamiya et al. proposed a new clone detection technique by transforming source code rules and using a token-based comparison [71]. CCFinder uses a lexical analyzer rule-based transformation to the token sequence formed from the input code with a suffix-tree matching algorithm [5] to compute matching. This is the reason that the tool detects clones that are not detected by other methods, specifically against the line-by-line method approach where CCFinder outperforms line-by-line method by more than 23% [8]. CCFinder does not accept source files written in two or more programming languages.

Asuncion et al. presented an initial investigation into an automated technique that combines traceability links with topic modeling using a machine learning approach [28]. This approach automatically records traceability links during the software development process and, over time, learns a probable topic model over artifacts. This approach was tested by implementing different tools such as a

topic-based artifact search engine, Topically-Rich Artifact Search Engine (TRASE) for artifact search, prospective capture tool, Architecture-Centric Traceability for Stakeholders (ACTS) for recording and topic-enhanced architecture visualization tool, and Topic-Enhanced Architecture Mashup (TEAM) for visualization [28]. At first, user's interactions with TRASE were prospectively recorded with ACTS, which produced trace links. These produced trace links were then augmented with topical information and were visualized within TEAM. That work employed a scalable approach to trace links with topic modeling in order to aid users in analyzing the semantic nature of artifacts in addition to the entire software architecture. The TEAM tool was presented to the ArchStudio developers [11, 12], and the response was mostly positive. The topic visualization has aided the developers in analyzing the correctness of the related artifacts. Visualization also enabled the developers to quickly identify whether or not the captured links were accurate or if there were missing links. In general, developers thought that the captured links were accurate. Results from the experiments suggested that LDA is competitive or superior to LSI [62] and that this approach can scale to handle larger numbers of artifacts. There are, however, limitations to this approach. First, the existence of architecture was assumed. This is not an unrealistic assumption since every system has an underlying architecture if it is explicitly documented. If the architecture is not explicitly documented or incomplete, virtual components to correspond to the source code can be created. Additionally, topic analysis on text-based artifacts was performed, and the topic model algorithm ignored the non-text artifacts.

Bellon et al. presented an experiment that evaluates six clone detector tools such as, Dup, CloneDR, CCFinder, Duplix, CLAN, and Duploc [60]. The authors evaluated these clone detectors' clone candidates as an independent third party. The selected techniques work on text, software metrics, syntactic and lexical in-

formation, and program dependency graphs and cover the whole spectrum of the state-of-the-art in clone detection. This experiment presented some astonishing results as the two token-based techniques (Dup and CCFinder) and the text-based technique (Duploc) performed similarly and had higher recall [60]. A function metrics tool (CLAN) [34] and AST-based tool (CloneDR) [31] had higher precision but they also had higher execution time. Individual tools missed many injected secret clones and in particular lacked abstraction. For instance, if two blocks of code are identical except that one block uses a literal in several places while the other block uses a function call, then the current tools may be able to find the many type-1 and type-2 identical clones between the two blocks of code fragments. However, the tools failed to notice that one block of code fragment can be transformed into the other by consistently using a function call instead of literals. Therefore, they failed to identify the fact that the fragments are nothing but clones.

In 2012, Yuan and Gao used the token-based detection technique for Boreas clones [76]. Boreas is a novel approach that uses a counting-based method to characterize the matrices of program segments in order to detect clones. Boreas was developed to have three terms used to categorize its elements; the Counting Environments (CE), Count Vectors or Characteristic Vector (CV), and the Count Matrix (CM) [76]. The CE is divided into three stages, which explains the pattern of the variable. The stages include the Naïve counting stage, instatement counting stage, and inter statement counting stage. The naïve counting stage is characterized by the presence of used variables. The in-statement counting stage is used for variables that are found in the instances of array subscript, if-predicates, any place where operations are directed to apply to variables, or where variables are expressed with constants. In the in-statement stage, the counting environment is set as the variable in the first level loop, second level loop, or deeper level loop [32].

The CV of various dimensions can be formed, by using the corresponding CEs. For instance, a CV with n dimension, an n CE would be used. For a CV with *int* dimension, an *int* CE would be used, that is, the number of counts of the *int* variable in the *int* CE. When an n variable is formed from a CV with the m dimension, a count matrix with the m*n is formed. The CM will contain abstracts of code fragments of n and m. The similarity between the two fragments is measured, using the characterization of their CMs and CVs. Given two occurrence counts a and b ($a >= b$), similarities between two CVs are calculated using the below formula:

$$ProSim = \frac{1}{a+b} + \frac{b}{a+1}$$

3.3 Tree-based detection

The Tree-based clone detection technique is also called the Syntactic Approach, or Abstract Syntax Tree Approach [31]. The technique uses a mode that converts a source program from syntax trees, also known as parse trees, into a mode that can be processed by structural metrics or tree matching to detect the presence of clones. The conversion is done by a parser. This operation technique is found in CloneDR [31]. They used three sub-processes in their work. The first was an algorithm to detect any form of subtree cloning. Every subtree is subjected to a scan with another subtree, and clones become identified on the basis of shared

similarities.

The second algorithm was developed to identify a continuum of clones, or the track for subtrees that may be present in a sequence. The algorithm is able to detect fragments and identify them by the sequence. If a single clone is found within a tree, the clone is labeled as a single clone. In the third algorithm, more complex detections are implemented to detect complex nearmiss clones [31]. The scan is done throughout the entire code to detect if a parent is a nearmiss clone. Jiang et al. also used an algorithm that detects clones with numerical vectors [36]. It operated with subtree characterization, which is responsible for the vector detection.

Yang employed a technique that was able to differentiate between two versions of an exact program, using the syntax tree [77]. Wahler et al. worked with the technique to detect subtree parameters and exact clones [72]. They were able to achieve this by transforming the source code into an XML format. Thierry Lavoie et al. employed the combination of the token based and metric based detections [47]. They used the Levenshtein distance to find the presence of clone codes. In order to calculate an accurate Levenshtein distance calculation, metric trees and Manhattan distance are needed. The calculation works by comparing strings the calculation of the number of insertions, alteration, and swapping of characters [71]. Levenshtein works by extracting the token from the source code, using the lexical analyzer. The second step sets up a frequency vector. A unique ID is generated for each of the tokens with a given dynamics. The basic idea is that if there is any introduction of a new item, it would be detected, and the generated ID will be provided for the exact token. If there is any change in the string, the dynamic will be changed accordingly. In the third step, T. Lavoie and Merlo explained that a basic metric tree is built with frequency vectors [47]. The

L1 is chosen because of its speed and precision and the lightness of its size. The final step is called the tree query step. The distance between the trees in both strings is measured by the Manhattan distance. If the calculated distance is less than the threshold, the clones are known and confirmed.

Rochelle et al. worked on clone detection using the semantic Input, Output, Effects (IOE) behavior [53]. Semantic clones, also known as behavioral clones [14], are detected by a basic input-output code examination. For every action (input-output inclusion) done, the effects are noted and examined for change in the heap's state. This technique worked well especially for Java codes [17]. *"In input value of parameters passed to function and heap's state at invocation time of method. In the output, the behavior is determined with the returned value of the function, possible and persistent change into heap,"* Khanna et al. [32] reported. The IOE semantic clone detection included four sub-processes: abstraction, filtering, testing, and collection.

- Abstraction: In the Abstraction processes, an Abstract Syntax Tree (AST) is formed so that the properties of two varying methods can be obtained.
- Filtering: Two filters are used. The first follows a definite method type, using syntactical information as the input and output method. The return type has its parameters as candidate clones. The second filter takes the semantic information and refines the candidate clones by a functional deletion of a command with a different effect. After the filtering stage, methods in an equivalent class would have the same types and effects.
- Testing: In this sub-process, the dynamic behavior of the codes are checked, using developed test files.

- Collection: The test driver that runs the files executes the actions and groups the clones into equivalent categories.

After the processes are complete, the code fragments or clones are identified. Kong et al. used a software program that operated with the K-nearest algorithm for clone detection [10]. The Abstract Syntax Tree (AST) induced technique combines the dependency graph and controls the k-nearest neighbor of a clustering algorithm [6]. Two fragments are considered to be a functional equivalence if there are two existing permutations, say P_1 and P_2.

$$OC_1(P_1(I)) = OC_2(P_2(I))$$

where OC_1 and OC_2 indicate the output set of C_1 and C_2 code fragments, respectively. The process operates in such a way that the code is first passed through a lexical analyzer and syntax analyzer to get the control dependency, by generating AST and Program Dependency Graph (PDG) information. The functional code is then detected by comparing the nearest cluster algorithm in the tree series. This way, variables in the input-output are identified. The next step transforms the ungrouped code into an executable command. The input that has been automatically generated is aligned with the new, executable command and is tested for dynamics. The output results determine the grouping when equivalent functions are found to match that those of the inputs. This grouping is called a cluster.

Maryand et al. introduced a technique that identifies function level clones from a large software code system [34]. DatrixTM, an analyzer tool set, which was used to extract a set of metric parameters from the source code and was developed as part of a research project. In the initial stage had the source code was transformed into AST, which was later transformed into a control flow graph and data flow

graph as an intermediate representation. Metrics were eventually calculated from the intermediate representation. The detected clones were analyzed to be recorded on a cloning scale range, which can read exact copy clone codes to distinct clones. This technique proved to be useful in terms of improving the maintainability of large software, and also in reducing the number of false positive to a negligible quantity.

Kontogiannis [42] performed experiments with some specific program features in order to get standard metrics that could be used as signatures to detect code clones. This was initially presented as a tree-based detection approach from which a computation was made based on the AST representation. 5D space metrics were computed and the Euclidean distance between the points was plotted. The spaces formed the basic signature that was used for the detection of code clones. This approach is fast and easy to use. It is capable of detecting clones if there are changes in the variable names of the source code. Geetika provided a table that shows the characteristic proficiency of the various code clone detection techniques that have been used in various software over the years [24]. This table outlines each technique's portability, efficiency, and integrality. Kodhai et al. worked with a hypothetical mix of both metric based and text based techniques [39]. The detection was carried out, using a code written in C programming language. The source code in the system was manually parsed to a standard format. After parsing, metric values were computed and the methods were compared on order to detect clone candidates. Additionally, a standardized code was developed to detect potential clones using text based comparison for the parsed standardized code. The mutual characters in both strings of code were identified as cloned codes. Based on the comparison of this technique with other clone detection methods, it is was considered to be less complex. Nevertheless, it was proven efficient and has prospects for

further application. Figure 3.1 shows an example of the standard parsing method.

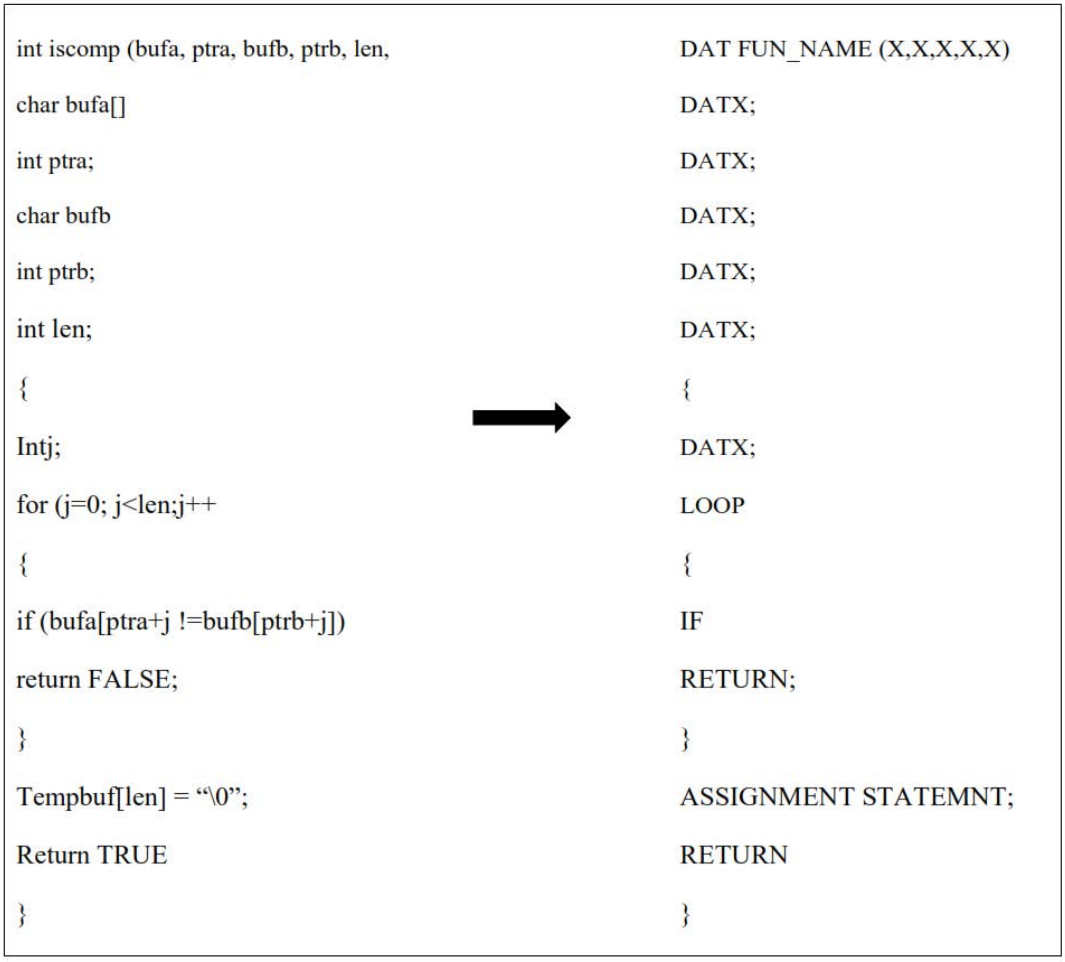


Figure 3.1: An example of parsing of a sample method [39]

Shawky et al. worked on a sequence of cluster, called sequential clustering, in clone determination [68]. This technique evaluated some metrics that were used for the detection of clones, and generated a sequence of metrics with 100% precision. The techniques were deemed acceptable to work as a preliminary stage in metrics based code clone detection; however, as noted, it's limitation was that it only detects an optimal metrics sequence for two types of clone systems. Shawky et al. concluded with a prospect that further metrics can be generated, such that it will be compat-

Authors	Techniques Used	Scalability	Portability	Complexity
Baxter et al. [31]	Abstract Syntactic Technique (AST) or Tree based approach	Less scalable	Uses parser. The parser transforms the source code into executable metrics. Technique is entirely language dependent.	It is complex
Yuan et al. [76]	Token Based technique	More scalable	The technique used was not language dependent	Less complex
Lavoie et al. [47]	Levenshtein Distance calculation that works with the combination of Token Based and Metric Based techniques	Less scalable	It was used on a 'C' language code	More complex
Rochelle et al. [53]	The technique worked by considering the behavioral code fragments with a semantic IOE	More scalable	It was applied and found applicable for Java language	Less complex
D. Kong et al. [10]	k-nearest algorithm is determined by the combination of the Tree Based and the Graph Based techniques	Less scalable	Used parser, which was language dependent	Less complex
Maryand et al. [34]	AST, and Graph Based Techniques	Less scalable	Made use of AST, control flow and data flow graphs	More complex
Kontogiannis [41]	AST	More scalable	Used the standard metrics and AST computation. (Procedural language)	More complex
Kodhai [39]	AST and the parsing technique	Less scalable	Used a standardized parsing code. (C language)	Less Complex
Shawky [68]	Metric based technique	More scalable	Made use of sequential clustering and metric clones. (C Language)	Less complex

Table 3.1: A comparison of different code cloning techniques

ible for all clone type systems [68]. A comparative analysis of various techniques that were introduced in this chapter are presented in Table 3.1.

The Table 3.2 below shows a number of highlighted types of clone detection command techniques and their properties. The properties of each of the detection technique were analyzed.

Jia et al. published a report about their software development for clone detection that used a newly developed algorithm called KClone [74]. KClone uses a novel,

Properties	Text Based	Token Based	Tree Based	PDG Based	Metric Based
Representation	Normalized source code	In the representation of tokens	Shown in the abstract syntax tree form	Considered as program dependency graph set	Metrics value set
Transformation	Eliminates comments and whitespace	Token is produced from the source code	AST is produced from source code	It is produced from the source code	For finding the metrics value AST is produced from the source code
Comparison dependent	Token of line	Token	Node of tree	Program dependency graph node	Metrics value
Computational Complexity	Algorithm dependent	Linear	Quadratic	Quadratic	Linear
Refactoring Opportunities	Better for exact matches	Some post-processing required	Better for refactoring as it finds syntactic clones	Better for refactoring	Manual inspection is needed
Dependency language	Ease for adaptability	It requires a lexer but no syntactic knowledge is needed	Parser is needed	Edge syntactic knowledge and PDG is needed	Parser is needed

Table 3.2: Comparative analysis of code clone detection techniques w.r.t varied properties

integrated combination of lexical and local dependence analytical comparison to achieve precision and was able to detect types 1, 2, and 3 clones. To develop this method, Jia et al. explained the problems that must be overcome while working on detecting each of the types their model was developed for [74]. Type-1 can be easily detected; therefore, it posed no problem to them. Although type-2 is also easy to detect, it involves having to navigate around two code fragments, f1 and f2, where there will be a corresponding yield in f2 for every uniform identifier replaced in f1 [74]. For the Type-3 clone, one more string statement can be eliminated, added, or altered. There is also the possibility for change in a code fragment, slightly varying from the original. This kind of clone is known to pose difficulties in detection so it was necessary to increase the sophistication of the algorithms to achieve higher degree of precision in order to detect high clones. The postulation for increasing the sophistication of the algorithm brought an additional burden in computational cost. The algorithm followed three strict steps

- Step 1: The transformation of the code to an internal representation.
- Step 2: It is responsible for the detection of the parts from which the clone

pairs were created. NOTE: It is the longest stage.

- Step 3: The detected clone pairs were then categorized into clone classes

This worked effectively for C, C++, and Java programs. A comparative analysis was done between KClone and CCFinder and it was noted that "The key benefit of this combination is an improvement in both inclusiveness of all clone types and performance."

One of the foremost experiments of clone detection was done by Bailey and Burd as they conducted comparative analyses on three tested and widely known detection techniques and two other plagiarism detection tools [5]. The team started the procedure by conducting a scan that validates all the clone candidates present in a source code, which had been called to form what they called the "human oracle." This was then used to form the standard comparison tool for the different metric based techniques. Baily and Burd's intention was to develop a preventive and effective maintenance method that can influence a clone validation method [5]. They were able to verify all the candidate characters present in a clone code system; however, their method was limited. They worked with a single subject system and modest system size, which demonstrated the subjectivity of their validation method and showed it to be less than a definitive tool [58]. Despite this finding, it became a model.

Roy and Cordy did a review of the work, stating that Bellon's experiment [60, 74] presented problems in terms of precision, recall, speed, RAM and the types of clones detected for each of the tools [58]. The data is reported at an ordinal scale {, -, +, ++ where { is worst and ++ is best (exact measures can be found in the corresponding papers). In some cases, a question mark was used to indicate

that they were unsure about that particular measure. As in Burd and Bailey's study, a human oracle was formed with the candidate clones from all the tools, then manually verified. Bellon was only able to verify 2% of the candidate clones due to time constraints. It is noted that for each of tool employed, precision and recall relate paradoxically; the only exemption is the PDG based Duplix, where both qualities are in their worst expressions. It is also noted that the majority of the tools are efficient and able to detect Type I and Type II clones, except Duplix. CCFinder, CLAN, and Duploc are efficient tools, able to detect Type III clones; however, Type III detection appears to be Duplix's strong point. CLAN, the metric-based tool, exhibited better precision overall. Bellon compared the tokens and textual images of the functions with which the metrics values were analyzed [61].

Rysselberghe and Demeyer worked on a project that evaluated prototypes of three representative clone detection techniques [19]. The results of their project was presented in comparative terms that showed the portability of the techniques used, the kinds of clone detection implemented, the scalability of the technique, the number false matches present in the clone, and the number of useless matches. Their work only evaluated small subject systems under 10KLOC so they did not make a reference set. Additionally, the parameters that validated the detected clones was not considered a reliable process. Rather than quantitative evaluation of the detection techniques, their intention was to determine the suitability of clone detection techniques for a particular maintenance task. They also conducted an evaluation of the techniques with respect to:

- Suitability: To determine whether or not a refactoring tool can manipulate

a candidate.

- Relevance: To see to the priority in which the matches would be refactored
- Confidence: The reliability of the results yielded by the code cloning tool, to determine whether or not it can work on its own without manual interference.
- Focus: The possibility of concentrating on a single class or an entire project assessment.

Considering the various techniques, there was no significant difference noticed with respect to relevance and focus. The technique demonstrated that, for an entire project or grained entities, there was no difference with respect to focus. For simple line matching, only one technique did not lack confidence. Other techniques may yield false matches that would require a manual inspection. All of the techniques tested only provided clone length and the file names of the returned clones.

Brutink et al. treated several clone detection techniques using homogenous implementations, to discover crosscutting concerns in C programs [48]. Three programs with three different detection models were used: 1) The token-based CCFinder 2) AST-based CCDIML that was a variant of Baxter's Clone DR, and 3) The PDG-based PDG-DUP. These programs were used as a means of finding the cross-cutting concerns in C programs that the study addressed, which consisted primarily of errors such as handling, tracing, and pre- and post-condition checking. In their study, CCDIML and CCFinder found null-pointer and error handling concerns without manual investigation. Ccdiml was more holistic in its function as it was able to find the complete range of the concerns. However, PDG-DUP was able to find the

cross-cutting concerns that ccdiml and CCFinder were unable to discover.

Jiang et al. demonstrated that Deckard was able to detect more clones than CloneDR (AST technique), and CPMiner (token-based technique) [36]. There has been no independent comparison that shows a program against all other programs as all-encompassing self-sufficient in all properties of clone detection capabilities. It is commonly agreed that there is no approach or tool that outperforms other [58].

Various researchers have proposed, developed, and worked with different models of clone detection, which includes text-based detection, token-based detection, metric-based detection, AST, PDG, and hybrid techniques. Each one of the researchers subjected various clone programs, under various programming languages, to scalability, portability, language dependency, and complexity checks.

CHAPTER 4

Research Methodology

This chapter describes our experimental investigation, including experimental datasets, implementation, main research assumptions, and evaluation techniques.

4.1 Datasets

To conduct the experimental analysis, four software systems from different application domains were used. These systems were selected based on their design and sizes that were most appropriate for the analysis being conducted. These systems include:

- iTrust: An open source medical application, developed by software engineering students at North Carolina State University (USA). iTrust is developed in Java and it provides patients with a means to keep up with their medical records and to communicate with their doctors. Version 15.0 was used for this analysis.
- Apache Ivy: The Ivy project is a sub-project of the Apache Ant project, the project with which Ivy works to resolve project dependencies by focusing on flexibility and simplicity. The system is written in Java and it provides auto-

Dataset	VER.	CLS	LANG	SLOC	CLOC
iTrust	15.0	299	Java	20.7K	9.6K
Apache Ivy	2.3.0	451	Java	49.9K	16.7K
QuantLib	1.3.0	874	C++	178.8K	22.3K

Table 4.1: Experimental datasets

matic retrieval of transitive dependency definitions and automatic integration to public artifact repositories. Version 2.3.0 was used for this analysis.

- QuantLib: A free open-source C++ project that is aimed at providing a comprehensive software framework for quantitative finance, including modeling, trading, and risk management in real-life. Version 1.3.0 was used in this analysis.

Table 4.1 describes the characteristics of the different experimental datasets including: the size of the system in terms of lines of source code (SLOC), lines of comments (CLOC), implementation language (LANG), version (VER), and a number of classes (CLS).

4.2 Gibbs Sampling

Gibbs Sampling is a technique that is used to analyze complicated actuarial models by reducing them to simpler models that are easier to manage. In light of this work, Gibbs Sampling is used as a link to try the Markov Chain Monte Carlo (MCMC) technique by running an inference with code models in text related instances to check for clone detection. In simulation approaches such as code cloning,

the technique becomes important when running analysis that can determine the iterated sample in an array of algorithms.

When considering a set of algorithms to work with Gibbs sampling, no samples are taken directly from posterior distribution. Rather, samples are simulated by running a sweep through all the posterior conditions randomly and one at a time. The initialization of algorithm codes with random values will simulate samples and, depending on the codes during early iterations, the outcome might not be representative of an actual posterior distribution. There is, however, a guarantee that the target joint posterior of interest is the stationary distribution of the samples generated under the algorithm [78]. Owing to this point, MCMC algorithms are used to analyze an array of large iterations, focusin on the target posterior. As a result, the early clones sampled from the iterations are discarded as being part of the "burn-in" period. The Gibbs Sampler is a special case of Metropolis-Hasting sampling that works around the acceptance of random sampling. The objective of the procedure is to analyze and process a chain whose value would converge at the target distribution. Initially, this method was proposed for image processing by Geman [25]. The technique uses a univariate conditional distribution (i.e., all the random variables are treated as fixed with the exception of one). In doing so in a chain, it becomes easier to simulate these. As Walsh [7] noted, "Thus, one simulates n random variables sequentially from the n univariate conditionals rather than generating a single n-dimensional vector in a single pass using the full joint distribution." Here is the breakdown of the Gibbs Sampler algorithmic process.

1. Randomly choose an attack spell.

2. To choose the buff conditional on the attack, the accept-reject algorithm

would be used.

3. Discard the attack spell chosen in the first step (1). Choose a new attack spell and use the accept-reject algorithm conditional on the buff in the second step (2) above.

4. Repeat the run for a long period of time, up to as many as 10,000 iterations.

5. The algorithm with the focus on the last iteration is the sample.

The Special instance of Metropolis-Hastings, the Gibbs Sampling would be applied with an MCMC algorithm as shown below:

- Consider distribution p(z)=p(z_1,...,z_m) from which the sample is taken.
- Choose an initial state for the Markov chain
- For each step, there is a replacement of value of one variable by a value drawn from p($z_j|z_i$) where symbol z_i denotes z_1,..,z_m with z_i omitted
- The procedure is repeated by iterating through variables in a considered pattern.

Steps for Distribution of p(z_1,z_2,z_3) over three variables is as follows

- At step t, the selected values are z_1 (t), z_2 (t) and z_3 (t)
- The next step z_1 will take a leap from t, with a new value, z_1 (t+1) obtained by sampling from p($z_1|z_2$ (t) ,z_3 (t))

- z_2 (t) by will assume a new value z_2 (t+1) by sampling from p($z_2|z_1$ (t+1),z_3 (t)) – The new value for z_1 is substituted immediately.
- z_3 is updated with a sample z_3 (t+1) drawn from p($z_3|z_1$ (t +1),z_2 (t+1))
- Analyses are run through the three variables in respective turns.

It is paramount that, in any of the MCMC Sampler iterations, the steps should be repeated until the chain approaches stagnancy (the burn-in period). Typically, the first 1000 to 5000 elements are discarded [72]. There are various convergence tests used to check whether or not 'stationarity' has been reached in a chain. Any one of them may be used, as long as it's suitable for the sampling circumstance; however, the observed limitation using Gibbs Sampling, according to Kazuhito Shida [69], is that it is vulnerable to local optima trapping. As a result, when conducting code clone detection using the Gibbs Sampling method, it is seen that by taking a large random text from a code as its conditional posterior, distributions only appear as though the samples were taken from their respective marginal distributions. The generated random samples are then analyzed to determine their properties with a code fragment or a code segment of interest. This will give a verification of cloning if striking similarities are observed. Sampling from the marginal distributions in an array of clustered codes is extremely difficult without Gibbs Sampling. When considering the results, Colton Gearheart [23] defined Gibbs Sampling as "The realization that as the number of iterations approaches infinity, the samples from the conditional posterior distributions converge to what the actual target distribution is that could not be sampled from directly."

4.3 Implementation and Tool Support

To implement topic modeling in this paper, *JGibbLDA*, a Java implementation of LDA was used. This particular implementation uses Gibbs Sampling for parameter estimation and inference [41]. To integrate *JGibbLDA* in the analysis, a C# based tool is implemented upon the current Java implantation. This prototype is referred to as *CloneTM*, a code clone detection tool based on Topic Modeling. Our interface provides options to tune the underlying LDA model (α, β, K), as well as the visualization support for LDA results. For instance, stacked charts and bar charts are available for visually comparing topic distributions of multiple artifacts. Figure 4.1 shows the main interface of *CloneTM*.

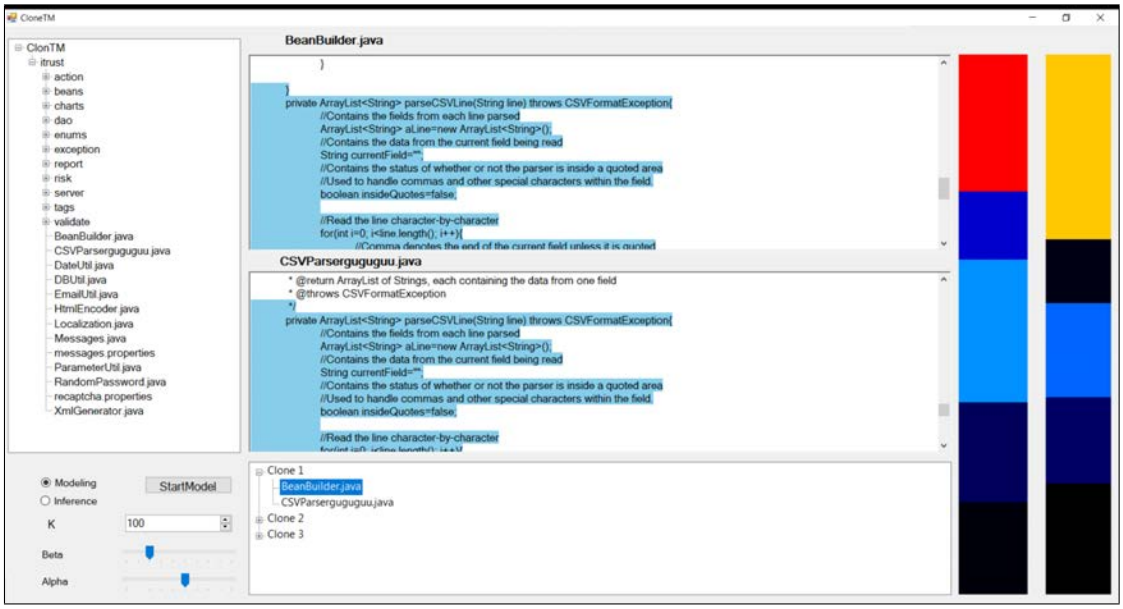


Figure 4.1: *CloneTM*, an LDA-based code clone detection tool

Modeling: In order to build the new LDA model with the Corpus, the user clicks

the radio button “Modeling”. Then the button “StartModel” is activated. Click the button “StartModel” and select the Corpus which is used in building of LDA model as shown in Figure 4.2. At this time, the user can set the LDA parameters such as α, β,and K.

Inference: Inference is selected by default. Before the user selects two source files to detect the clone codes, the “inference” button must be checked. Then, the user selects the source files in the tree view control as shown in Figure 4.1. From there, the app inferences the topic models with two selected files, and detects the clone codes based on the inferred topic model.

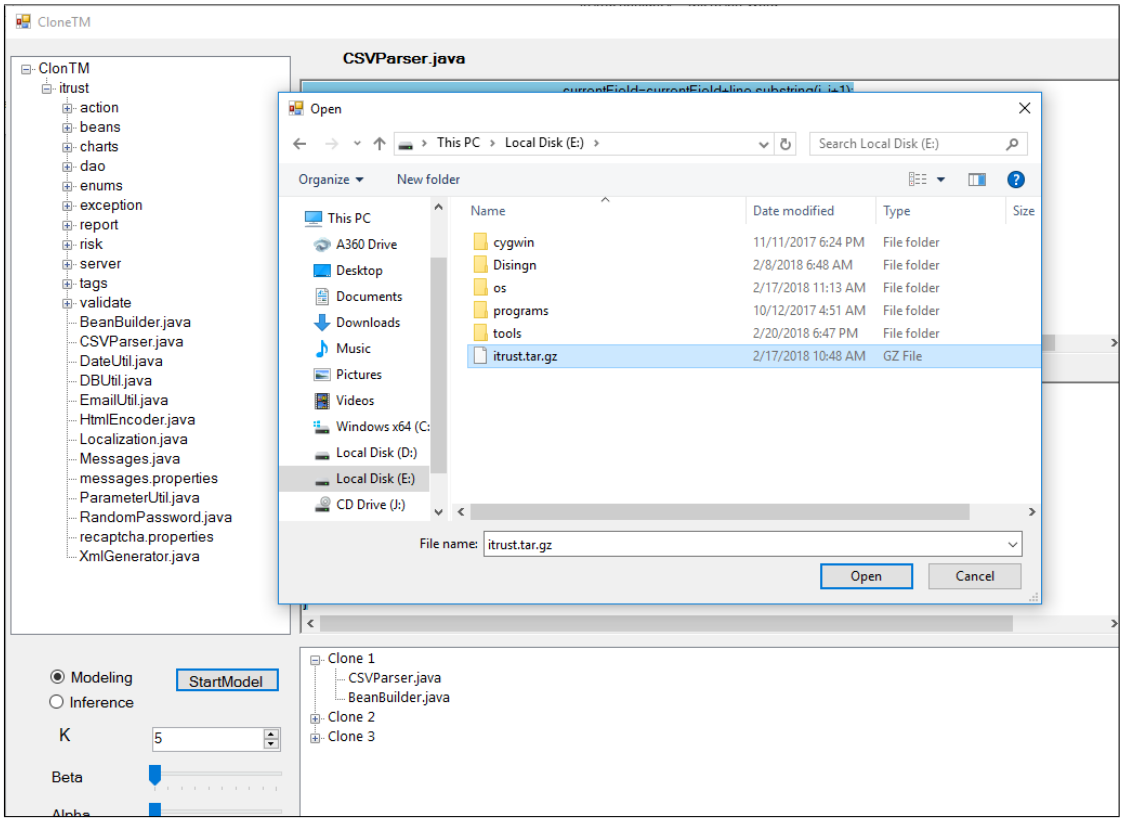


Figure 4.2: Dataset Modeling selection

Whenever a project is selected for analysis, the tool starts by indexing the source

code. Indexing extracts textual content (splitting identifiers names, removing stop words, and stemming) from each artifact in the system [50]. By adopting a broad range of regular expressions, *CloneTM* provides support for multiple programming languages including Java, C++, and C#. In the analysis, work was conducted at class granularity level; therefore, an artifact refers to a code class, and both terms are used interchangeably throughout this paper.

The output of the indexing process is a compact content descriptor, or a profile, which is usually represented as a vector space model that is fed to the LDA model as input, along with other LDA parameters including α, β, K. The tool then proceeds to generate the document-topic and topic-word distributions explained earlier. The dominant topics are shown by the stack bars on the right end of the tool as shown in Figure 4.3. Further explanation of CloneTM's code clone detection is provided in Chapter 5.

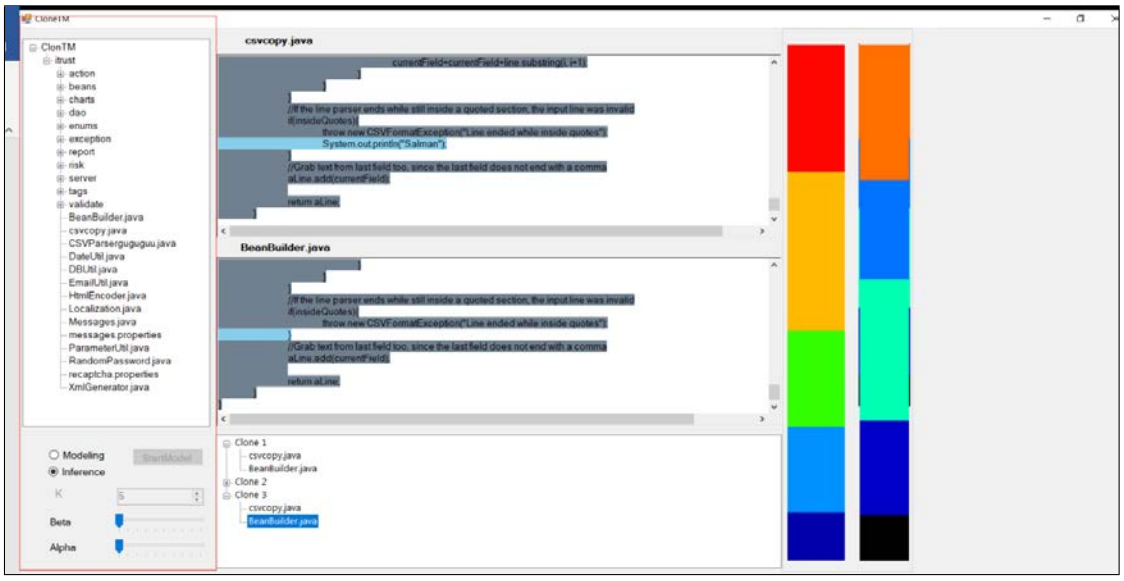


Figure 4.3: Code clone detection by CloneTM.

4.4 Dominant Topic Analysis

The analysis was started by investigating the effect of a different number of topics (K) over the topic distribution generated for each artifact in each of the experimental systems. It is important to point out that the complexity of the study grows exponentially with the inclusion of other LDA parameters such as α and β; therefore, at this stage of the analysis, the values of these parameters are fixed. This strategy is often used in related research to control for such variables' effect [26, 75, 65]. In particular, values of $\alpha = 50/\text{K}$ and $\beta = 0.1$ are used. These heuristics have been shown to achieve satisfactory results in related literature [27, 73].

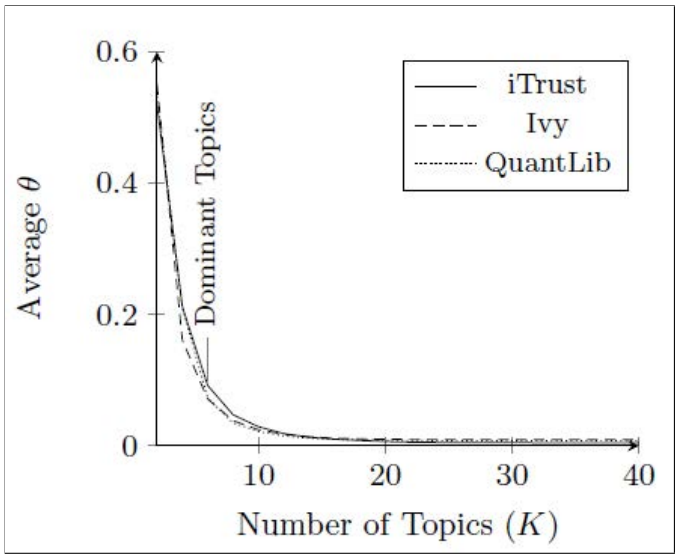


Figure 4.4: Average θ values in the document-topic matrix arranged in a descending order for K =40

Topic analysis shows that for each artifact $d_j \in$ D regardless of K, there are always a few numbers of topics that stand out from the rest of the topics in the document-

topic matrix. Such topics have relatively larger θ_{ji} values. To demonstrate this effect, LDA was run using K = 40 over our experimental datasets. For each artifact d_j in the document-topic matrix of each system, the 40 topics were sorted in descending order according to their θ_{ji} value, so that topics at rank (r =1) have the highest θ_{ji}. These values were then averaged for all artifacts (N) in the system over each rank r, producing A_r.

$$A_r = (\frac{1}{N}) \sum_{j=0}^{N} d_j(\theta_r) \forall_r = 1, 2, 3, ..., 40 \tag{4.1}$$

The results are shown in Figure 4.4 which demonstrates that, in the topic distribution of each artifact, only a small portion (≈5) of the topics has a θ_i value larger than a certain threshold value (λ). These topics with relatively larger θ_i are known as dominant topics [75, 1]. In an attempt to specify λ, further empirical analysis was conducted over the experimental, open source systems using different values of K. Results showed that the topics' probability distribution for each artifact seemed to consistently follow a regular distribution. In general, three categories of topics, based on the empirically observed λ, can be identified as follows:

- λ_1 (θ => 0.1) : Topics in this category are known as the absolute dominant topics, or the most probable topics in the document-topic matrix of each artifact. Analyzing the topic word distributions of these topics shows a large overlap ratio between their representative terms, and the terms representing the artifacts they dominate. In object-oriented software systems, each class often implements a single semantically coherent concept, so generally, 1 or 2 topics can be classified under this category [75].

- λ_2 ($0.1 > \theta => 0.01$): An average of 4 to 8 topics for each document. The topics in this category are referred to as the dominant topics. Term overlap analysis shows that these topics share a smaller number of terms with the class, usually centered around certain code fragments.

- λ_3 ($\theta < 0.01$): Most of the topics in the document topic distribution of each artifact fall into this category. These topics have a very small number of common terms with the class, most often generated by a single term overlap.

Observations regarding the distribution of system artifacts' topics, and the term-overlap analysis were used to derive the main hypothesis set forth in this paper. An assumption that documents sharing similar code fragments might also share a similar dominant topic distribution was experimentally tested and the results are outlined in the next chapter.

CHAPTER 5

Detecting Code Clones

In this chapter, we investigate the effect of code clones on the latent topic structure of software artifacts in order to derive the main hypothesis in this paper. The following example was used to guide the analysis:

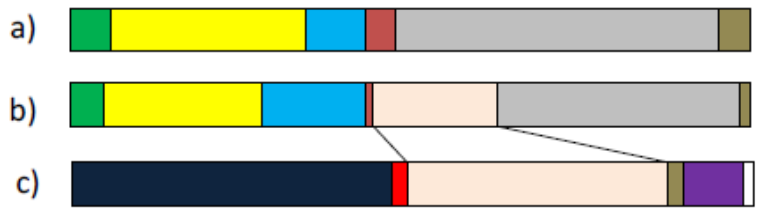


Figure 5.1: Stacked bars of dominant topic distributions of a) AddHCPAction.java b) AddHCPAction.java after cloning c) PersonnelDAO.java.

Example 1: Class AddHCPAction.java in the iTrust system calls the public method editPersonnel from class PersonnelDAO.java. A document matrix for both classes using K = 120 was generated. Then the method call in AddHCPAction.java was replaced with the function body from PersonnelDAO.java, thus introducing a code clone. The topic distribution of AddHCPAction.java is then re-generated and matched with the topic distribution of PersonnelDAO.java. Figure 5.1 shows the dominant topic distributions ($\theta_i >= 0.01$) of all three cases in a stacked chart format. Stacked charts display the contribution of several data items into a total, represented by bars stacked one on top of, or next to each other. The figure shows that injecting the code clone in AddHCPAction.java has changed its latent

	iTrust		Ivy		QuantLib	
Type	NO. C	CLS	NO. C	CLS	NO. C	CLS
Type I	20	42	20	85	40	86
Type II	20	44	20	17	40	92
Type III	15	30	15	45	25	50

Table 5.1: Injecting Code Clones into our systems

topic structure. Specifically, a new topic (t_{19}) has emerged as a dominant topic. Now both classes match on this particular dominant topic. Examining the high-probability terms representing t_{19} shows a high overlap ratio with the unique set of terms representing the method editPersonnel. This leads to the main research hypothesis in this paper that "matching in some of the dominant topics between two artifacts in a software system indicates the presence of code cloning."

To test this hypothesis, an experimental benchmark was devised to analyze the performance of LDA in detecting code clones. Code clones Types I, II, and III were manually injected in each of the experimental datasets. Type IV clones were excluded in this study. Manually injecting and verifying code smells for refactoring studies is a common practice in related research, especially in proof-of-concept studies [55, 60, 41, 5]. Additionally, since we are working with class granularity level, the analysis in this paper is limited to to cross-class or cross-file clones.

Table 5.1 shows the characteristics of the injected clones, including the number of clones injected of each type in each system (NO. C) and the number classes affected (CLS). Since QuantLib is a relatively larger system, more clones were able to be injected into it. Injecting Type I and II clones is a straightforward process. For instance, to produce a Type I clone, a method call is replaced by the method itself, and changes in spacing and comments are made. A Type II clone can be injected in a similar way, but by changing code identifiers' names. Injecting Type III clones is a

more challenging task. In particular, code statements have to be reordered, added, and removed. To achieve this, random sequences of certain operation-preserving transformations were applied into copied code fragments. These transformations include:

Conditional Statements: Break and merge certain if and while statements into if else statements and vice versa. For example, the code segment:

```
if(validteUsrNm(uName) && validPwd(uPwd))
    return true;
```

can be broken down into:

```
if(validteUsrNm(uName))
    if(validPwd(uPwd))
        return true;
```

Loops: Certain for statements were converted into while loops and vice versa. For example, the following loop statement:

```
for(line=br.readLine(); line!=null; line=br.readLine())
```

is transformed to the following while statement.

```
line = br.readLine();
    while (line != null)
```

```
            line = br.readLine()
```

Re-order: Certain statements were reordered in such a way that does not change the structure of the code. For example, in the following code segment, variable fBloodPressure declaration can be moved above the method call setPatientRecords(patientID) without affecting the functionality of the code.

```
    line = br.readLine();
        while (line != null)
            line = br.readLine()
```

The second step in the analysis is to define the notion of matching between dominant topics. In general, a set-matching procedure was followed. If any two classes had the same topic appearing in their set of topics with $\theta_i > .01$, this case is considered to be a candidate instance of code clones. Here, the word candidate means that it is suspected that, in some cases, matching may also happen without the presence of cloning. In that case, a false positive is given. The procedure for our LDA-based clone detection approach can be described as shown above.

Standard recall and precision metrics of information retrieval were used to assess the effectiveness of LDA in capturing instances of different types of clones. These metrics are often used to assess the performance of clone detection tools [5, 60, 58]. Recall measures coverage, and is defined as the percentage of clones that are correctly identified by the tool. Precision measures accuracy, and is defined as the

```
Detect Clones: Input D, α, β
 1. K = 40;
 2. Doc_Topic_Matrix = Generate_Topic_Model (D,α, β, K)
 3. FOR EACH d_i ∈ D IN Doc_Topic_Matrix
 4.       FOR EACH t_j ∈ d_i
 5.            IF θ_j < 0.01 THEN Remove t_j
 6. FOR EACH d_i ∈ D
 7.       FOR EACH d_j ∈ D
 8.            IF i≠j
 9.                 IF (Match (Doc_Topic_Matrix (d_i ,d_j )) > 0)
 10.                RETURN TRUE
 11. K +=40
 12. GOTO 2
```

Figure 5.2: Procedure for LDA based Clone detection

percentage of identified clones that are correct.

CHAPTER 6

Model Calibration and Evaluation

Our approach was evaluated by testing *CloneTM* over two systems: the open source system, iTrust, and the industrial proprietary system, WDS. The objective was to assess the usefulness and the scope of applicability of this approach and, more importantly, to identify limitations and areas for improvement.

To run the analysis, K was initially set to be 40 topics. The document topic distribution of each artifact in the system was generated, and a pair-wise comparison was conducted to capture matching in the latent topic structures of different classes. Results were then evaluated against the answer set using recall, or the number of injected clones the tool managed to identify. The value of K was then increased by 40, and the process was repeated. This particular step size has been found to yield noticeable changes in the recall values. A hill climbing approach was followed to monitor the changes in the recall. Our objective was to identify or approximate near-optimal K settings to detect clones. Optimality was tied to recall in this paper; therefore, in this analysis, recall has been emphasized over precision. The rationale used is that errors of commission (false positives) are easier to deal with than errors of omission (false negative). In other words, it is easier for a developer to discard instances that were misclassified as clones, rather than

to deal with clones that were not detected by the tool.

Since different types of clones were injected, a separate recall curve for each type was produced, as shown in Figure 6.1. As for precision, a single precision chart, which shows the percentage of misclassified cases, was produced for each system as shown in Figure 6.2. The list of candidate clones generated for each system was scanned for pre-existing cross-file clones before injecting our own clones, so such clones were excluded from our precision calculations. Our evaluation benchmark was also implemented in *CloneTM*. Candidate clones were displayed to the user and the lines of code, which include words from the matching topics, were highlighted in the class view window of each class. Results show that in all three systems, the recall seems to converge to a local maximum at the range of K = [160, 200] topics for all systems.

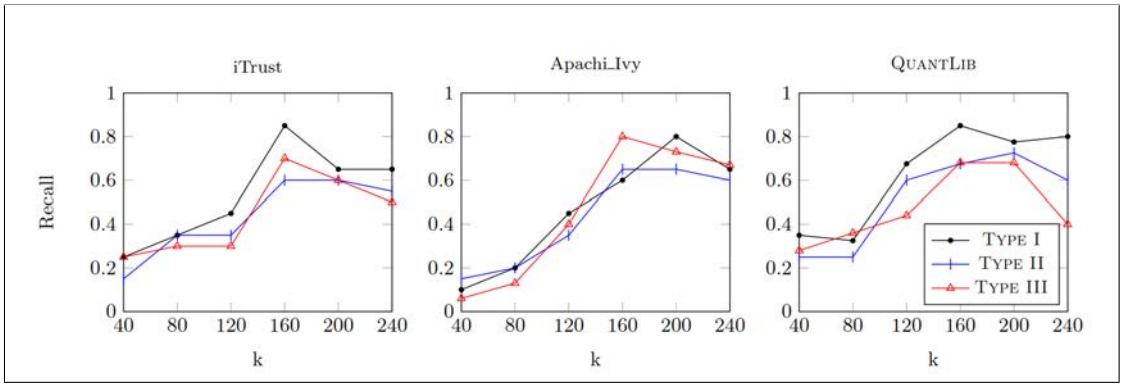


Figure 6.1: Recall values of different types of code clones at different values of K

The precision values also show satisfactory performance at this level. This can be explained based on the observation that at this range of K, topics tend to be more distinguishable from each other which makes this particular number of topics seem to be the most optimal for code clone detection. At lower values of K, topics tend to have less density, and are generally spread all over the class, whereas at higher K

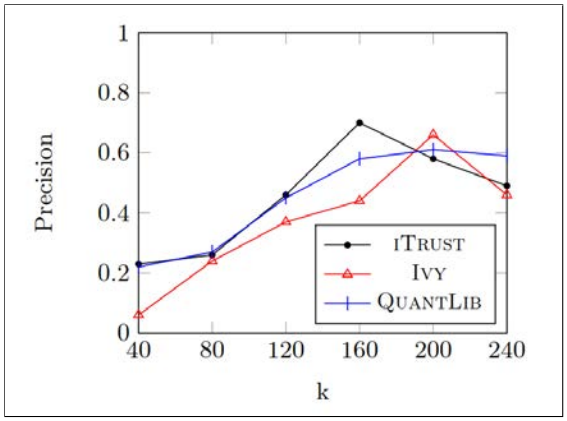


Figure 6.2: Precision values at different values of K.

Recall									
	TYPE I			TYPE II			TYPE III		
System	CloneTM	CCFinder	CloneDR	CloneTM	CCFinder	CloneDR	CloneTM	CCFinder	CloneDR
ITRUST	0.85	0.9	0.2	0.6	0.8	0.2	0.7	0.4	0.13
IVY	0.8	0.8	0.2	0.65	0.85	0.15	0.73	0.67	0.27
QUANTLIB	0.775	0.7	0.35	0.575	0.7	0.25	0.68	0.53	0.32

Table 6.1: Comparing Recall Values of CloneTM, CCFinder and CloneDR.

values (i.e., > 200) topics tend to be very specific, unable to cover code fragments with meaningful size. In general, the results show that this LDA-based approach was able to capture most types of clones, showing particularly good performance in detecting Type I and Type III clones. Results show that the precision, while satisfactory, is still far from being optimal. To put the performance of *CloneTM* in perspective, its recall and precision were compared with other clone detection tools such as CCFinder [71] and CloneDR [21]. Table 6.1 shows the results of the tool comparison.

For each type of clone in each system, the recall values were also compared. Results show that *CloneTM* is able to achieve comparable levels of recall to other tools

in all systems. In particular, results demonstrate that this LDA-based approach managed to outperform CCFinder in Type III refactoring, which can be explained based on the fact that the sequential analysis of code statements in CCFinder makes it sensitive to statement reordering and code insertion. In general, many other token-based detection approaches do not detect clones with reordered statements [15]; however, the fact that LDA treats a class as a bag of words makes it immune to such changes. In contrast, results show that CCFinder was more successful in detecting Type II clones, based on the fact that token-based methods are immune to name changing. On the other hand, LDA can be very sensitive to the information value embedded in the identifiers names and comments, so inconsistency in such information is expected to lower accuracy. Results also show that, in comparison to CloneTM and CCFinder, CloneDR captured the smallest number of clones in all different types of clones. That might be explained based on the fact that this tool tends to do better in cross-method rather than cross-file clone detection [4].

CHAPTER 7

Threats to Validity

This study has several limitations that might affect the validity of the results. In terms of external validity, the results of this study might not generalize beyond the underlying experimental settings. For instance, only four systems were employed in the entire process of our analysis. Nevertheless, we believe that using four datasets from different domains, including a proprietary software product, helped to mitigate these threats. In fact, we believe that using these heavily-used, open source tools and systems increases the reliability of our results as it makes it possible to independently replicate our results. Other threats to the external validity might stem from specific design decisions, such as using heuristic values for α and β. Nevertheless, as mentioned earlier, it was not feasible to evaluate the effect of all LDA parameters in this study due to the exponential complexity of the problem. It should be noted that the heuristics used in this analysis have been proven to achieve satisfactory performance in other related researches, which provides confirmation of validity to reasonable and logical extents.

Internal validity refers to factors that might affect the causal relations established in the experiment. A major threat to this study's internal validity is the fact that manually injected clones were used to calibrate the model. Our cloning models were manually created and configured to assume similarities with a real and existing coding system. We also attempted to manually verify the candidate clones of different tools. This can lead to an experimental bias due to the subjectivity of

this process; however, this particular experiment design decision was necessary to gain more insight into the procedure's performance, in particular, its effectiveness in detecting different types of clones. As reported earlier, in the current state of research, human approval of the outcome of the code clone detection tool or method is inevitable.

CHAPTER 8

Conclusion

In this thesis, a proposition for a novel approach based on topic modeling for code clone detection was made. In particular, the potential benefits of using LDA to identify cross-class similar code fragments in Object Oriented software systems was investigated. The main hypothesis was constructed upon observations related to the latent topic structure of the software artifacts, and the effects that code clones might have on that structure. In particular, it was assumed that matching the dominant topic distribution between individual artifacts indicates cloning. To test this hypothesis, calibration, and the evaluation of our approach, an experimental analysis using four software systems from different application domains was conducted. Additionally, we compared the performance of this approach with other popular clone detection tools that adopt different clone detection strategies including CCFinder and CloneDR.

Results show that LDA can achieve satisfactory levels of recall, showing particularly good performance in detecting Type III clones that other related tools usually tend to miss. It also achieves levels of accuracy that is adequate for practical applications. In the future, we plan to evaluate this approach by testing *CloneTM* using open source software systems to assess the usefulness and the scope of applicability of this approach. We plan to fully implement our finding in the tool, and provide visualization support to allow users to visually compare topic distributions of different classes, as well as accept or reject candidate clones. Also potential ef-

fect of other code smells, such as God Class or Feature Envy [18], on the latent topic structure of software artifacts will be investigated.

REFERENCES

[1] A. Panichella, B. Dit, R. O. M. D. P. D. P., and Lucia, A. D. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In International Conference on Software Engineering (2013), 522–531.

[2] B. Lague, D. Proulx, J. M. E. M., and Hudepohl, J. Assessing the benefits of incorporating function clone detection in a development process. In International Conference on Software Maintenance (1997), pages 314–321.

[3] Baker, B. On finding duplication and near-duplication in large software systems. In Working Conference on Reverse Engineering (1995), 86–95.

[4] Biegel, B., Soetens, Q., Hornig, W., Diehl, S., and Demeyer, S. Comparison of similarity metrics for refactoring detection. In Working Conference on Mining Software Repositories (2011), 53–62.

[5] Burd, E., and Bailey, J. Evaluating clone detection tools for use during preventative maintenance. In International Workshop on Source Code Analysis and Manipulation (2002), 36–43.

[6] C. Lung, X. Xu, M. Z. A. S. Program restructuring using clustering techniques. The Journal of Systems and Software (2006).

[7] Carlo, C. Markov Chain Monte Carlo and Gibbs Sampling, 2002.

[8] D. Andrzejewski, A. Mulhern, B. L., and Zhu, X. Statistical debugging using latent topic models. In European Conference on Machine Learning (2007), 6–17.

[9] D. Blei, A. N., and Jordan, M. Latent dirichlet allocation. Journal of Machine Learning Research (2003), 993–1022.

[10] D. Kong, X. Su, S. W. T. W., and Ma, P. Detect functionally equivalent code fragments via k-nearest. IEEE fifth International Conference on Advanced Computational Intelligence (ICACI) (2012).

[11] Dashofy, E. Supporting Stakeholder-Driven, Multi-View Software Architecture Modeling. PhD thesis, Info and Comp Science, Univ of Calif, Irvine, 2007.

[12] E. Dashofy, H. Asuncion, S. H. G. S. J. G., and Taylor, R. N. Archstudio 4: An architecture-based meta-modeling environment. 29th ICSE, Comp Vol (2007).

[13] E. Juergens, F. Deissenboeck, B. H., and Wagner, S. Do code clones matter? In International Conference on Software Engineering (2009), 485–495.

[14] E. Juergens, F. D., and Hummel, B. Clone detection beyond copy & paste. In Proc. of the 3rd International Workshop on Software Clones (2009).

[15] E. Linstead, C. L., and Baldi, P. An application of latent dirichlet allocation to analyzing software evolution. In International Conference on Machine Learning and Applications (2008), 813–818.

[16] E. Murphy-Hill, C. P., and Black, A. P. How we refactor and how we know it. In International Conference on Software Engineering (2009), pages 287–297.

[17] Elva, R., and Leavens, G. Jsctracker: A semantic clonedetection tool for Java code. PhD thesis, Department of EECS, University of Central Florida, 2012.

[18] F. Rahman, C. B., and Devanbu, P. Clones: What is that smell? MSR (2010), pages 72–81.

[19] F. V. Rysselberghe, S. D. Evaluating clone detection techniques. In Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA'03) (2003).

[20] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison–Wesley, 1999.

[21] G. Bavota, R. Oliveto, M. G. D. P., and Lucia, A. D. Methodbook: Recommending move method refactorings via relational topic models. IEEE Transactions on Software Engineering (2014).

[22] G. Maskeri, S. S., and Heafield, K. Mining business topics in source code using latent dirichlet allocation. In India software engineering conference (2008), 113–120.

[23] Gearhart, C. Implementation of gibbs sampling within bayesian inference and its applications in actuarial science. SIAM (2018).

[24] Geetika. Clone code detection by evaluating combinations of software metrics. Master's thesis, Thaper University, 2014.

[25] Geman, S., and Geman., D. Stochastic relaxation, gibbs distribution and bayesian restoration of images. IEEE Transactions on Pattern Analysis and Machine Intelligence (1984), 721–741.

[26] Grant, S., and Cordy, J. Estimating the optimal number of latent concepts in source code analysis. In International Working Conference on Source Code Analysis and Manipulation (2010), 65–74.

[27] Griffiths, T., and Steyvers, M. Finding scientific topics. In The National Academy of Sciences (2004), 5228–5235.

[28] H. Asuncion, A. A., and Taylor, R. Software traceability with topic modeling. In International Conference on Software Engineering (2010), 95–104.

[29] Harris, S. Simian – similarity analyser, version 2.4. http://www.harukizaemon.com/simian/.

[30] Hofmann, T. Probabilistic latent semantic indexing. In International ACM SIGIR Conference on Research and Development in Information Retrieval (1999), 50–57.

[31] I. Baxter, A. Yahin, L. M. M. S., and BierIn, L. Clone detection using abstract syntax trees. International Conference on Software Maintenance (1998), 368–377.

[32] J. Khanna, R. Singh, R. G. Review of clone detection. International Journal of Computer Science and Network 3, 2 (2014).

[33] J. Mayrand, C. L., and Merlo, E. Experiment on the automatic detection of function clones in a software system using metrics. In International Conference on Software Maintenance (1996), 244–253.

[34] J. Mayrand, C. L., and Merlo, E. M. Experiment on the automatic detection of function clones in a software system using metrics. In Proceedings of the 12th International Conference on Software Maintenance, USA (1996), 244–253.

[35] J. Wang, D. Y., and Shu, X. Detection of code clone, based on source fragment alignment. Journal of Software Engineering (2017), 266–274.

[36] Jiang, L., D. Misherghi, Z. S., and Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering, USA (2007), 96–105.

[37] Jiang, Z., and Hassan, A. A framework for studying clones in large software systems. In International Working Conference on Source Code Analysis and Manipulation (2007), 203–212.

[38] K. Meghan, R., and Poshyvanyk, D. Using latent dirichlet allocation for automatic categorization of software. In International Working Conference on Mining Software Repositories (2009), 163–166.

[39] Kodhai, S. Kanmani, A. K. R. R., and Saranya, B. V. Detection of type-1 and type-2 code clone using textual analysis and metrics. In Proceedings of the 2010 International Conference on Recent Trends in Information, Telecommunication and Computing, Kerala (2010), 241–243.

[40] Komondoor, R., and Horwitz, S. Using slicing to identify duplication in source code. In International Symposium on Static Analysis (2001), 40–56.

[41] Kontogiannis, K. Evaluation experiments on the detection of programming patterns using software metrics. In Working Conference on Reverse Engineering (1997), 44–54.

[42] Kontogiannis, K. Evaluation experiments on the detection of programming patterns using software metrics. In Proceedings of the 3rd Working Conference on Reverse Engineering, Netherland (1997), 44–54.

[43] Krinke, J. Identifying similar code with program dependence graphs. In Working Conference on Reverse Engineering (2001.), 301–309.

[44] Kumar, B., and Singh, S. Code clone detection and analysis using software metrics and neural network - a literature review. International Journal of Computer Science Trends and Technology 3, 2 (2015).

[45] L. Aversano, L. C., and Penta, M. D. How clones are maintained: An empirical study. In European Conference on Software Maintenance and Reengineering (2010), 81–90.

[46] L. Prechelt, G. M., and Philippsen, M. Finding plagiarisms among a set of programs with jplag. Cognitive Psychology 8, 11 (2002), 1016–1038.

[47] Lavoie, and Merlo, E. Automated type-3 clone oracle using levenshtein metric. IEEE Computer Society Press, IEEE (2004).

[48] M. Bruntink, A. Deursen, R. E. T. T. On the use of clone detection for identifying crosscutting concern code. Transactions on Software Engineering 31, 10 (2005), 804–818.

[49] M. Kim, L. Bergman, T. L., and Notkin, D. An ethnographic study of copy and paste programming practices in oopl. In International Symposium on Empirical Software Engineering (2004), 83–92.

[50] Mahmoud, A., and Niu, N. Source code indexing for automated tracing. In International Workshop on Traceability in Emerging Forms of Software Engineering (2011), pages 3–9.

[51] Murphy-Hill, E., and Black, A. P. Breaking the barriers to successful refactoring: Observations and tools for extract method. In International Conference on Software Engineering 1 (2008), pages 421–430.

[52] Paul, S., and Prakash, A. A framework for source code search using program patterns. IEEE Transactions Software Engineering 20, 7 (1994), 463–475.

[53] R. Elva, G. T. L. Semantic clone detection using method ioe behavior. InIWSC, Zurich, Switzerland,IEEE (2012).

[54] R. Geiger, B. Fluri, H. C. G., and Pinzger, M. Relation of code clones and change couplings. In International Conference of Fundamental Approaches to Software Engineering (2006), 411–425.

[55] R. Koschke, R. F., and Frenzel, P. Clone detection using abstract syntax suffix trees. In Working Conference on Reverse Engineering (2006), 253–262.

[56] R. Saha, C. Roy, K. S., and Perry, D. Understanding the evolution of type-3 clones: An exploratory study. In IEEE Working Conference on Mining Software Repositories (2013), 139–148.

[57] R. Tairas, J. G. Phoenix-based clone detection using suffix trees. In Proceedings of the 44th annual Southeast regional conference, USA (2006), 679–684.

[58] Roy, C., and Cordy., J. A survey on software clone detection research. Technical report, School of Computing, Queens University, 2007.

[59] Roy, C. K., and Cordy, J. R. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. ICPC'08 (2008), 172–181.

[60] S. Bellon, R. Koschke, G. A. J. K., and Merlo, E. Comparison and evaluation of clone detection tools. IEEE Transactions of Software Engineering 33, 9 (2007), 577–591.

[61] S. Bellon, G. Antoniol, J. K., and Merlo, E. Detection of software clones tool comparison experiment. ACM (2007).

[62] S. Deerwester, S. Dumais, T. L. G. F., and Beck, L. Improving information retrieval with latent semantic indexing. In Annual Meeting of the American Society for Info. Science 25 (1988).

[63] S. Demeyer, S. D., and Nierstrasz, O. Object-Oriented Reengineering Patterns. Elsevier, 2003.

[64] S. Ducasse, M. R., and Demeyer, S. Language independent approach for detecting duplicated code. In International Conference on Software Maintenance (1999), 109–118.

[65] S. Thomas, B. Adams, A. H., and Blostein, D. Validating the use of topic models for software evolution. In IEEE Working Conference on Source Code Analysis and Manipulation (2010), pages 55–64.

[66] S. Thomas, B. Adams, A. H., and Blostein., D. Modeling the evolution of topics in source code histories. In Working Conference on Mining Software Repositories (2011), 49–158, 173–182.

[67] S. Thomas, H. Hemmati, A. H., and Blostein, D. Static test case prioritization using topic models. Empirical Software Engineering (2012), pages 1–31.

[68] Shawky, D. M., and Ali, A. F. An approach for assessing similarity metrics used in metric-based clone detection techniques. In 3rd IEEE International Conference on Computer Science and Information Technology, China (2010), 580–584.

[69] Shida, K. Gibbsst: A gibbs sampling method for motif discovery with enhanced resistance to local optima. BMC Bioinformatics 7 (2006), 486.

[70] T. Chen, S. Thomas, M. N., and Hassan, A. Explaining software defects using topic models. In IEEE Working Conference on Mining Software Repositories (2012), 189–198.

[71] T. Kamiya, S. K., and Inoue, K. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions Software Engineering 28, 7 (2002), 654–670.

[72] V. Wahler, D. Seipel, J. G., and Fischer, G. Clone detection in source code by frequent itemset techniques. In Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (2004), 128–135.

[73] Wei, X., and Croft, B. Lda-based document models for ad-hoc retrieval. In ACM SIGIR conference on Research and development in information retrieval (2006.), 178–185.

[74] Y. Jia, D. Binkley, M. H. J. K., and Matsushita, M. Kclone: A proposed approach to fast precise code clone detection. IWSC (2011).

[75] Y. Liu, D. Poshyvanyk, R. F. T. G., and Chrisochoides, N. Modelling class cohesion as mixtures of latent topics. In International Conference on Software Maintenance (2009), pages 233–242.

[76] Y. Yuan, Y. Guo, B. An accurate and scalable token-based approach to code clone detection,. In In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (2012), ASE,IEEE.

[77] Yang, W. Identifying syntactic differences between two programs. Software Practice and Experience 21, 7 (1991), 739 – 755.

[78] Yildirim, I. Bayesian inference: Gibbs sampling. Department of Brain and Cognitive Sciences (2012).

[79] Z. Li, L. Shan, S. M., and Zhou, Y. Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering 32, 3 (2006), 176–192.

VITA

Mohammed Salman Khan received a bachelor's degree in Electronics and Communication from Jawaharlal Nehru Technology University, Hyderabad in India in 2013, and aims to receive his Master of Science in Computer and Information Sciences from University of North Florida by April 2019. Dr. Sandeep Reddivari of the University of North Florida is Salman's thesis advisor. After achieving his undergraduate degree, Salman had joined a startup company and played a vital role in making the startup a success. He is also a contributor to multiple open source systems including WordPress and Grafana. During his term at UNF, he had experience working on-campus as a Graduate Teaching Assistant. He is also an Oracle Certified Java Professional(OCJP) and Microsoft Certified Professional(MCP). He currently works as an IT Systems Engineer for CEVA Logistics in Jacksonville, Florida. His expertise includes Hadoop, Cassandra, Elastic Search, SOLR, Angular, Java, C# .Net and Microsoft SQL server. Salman's long-term dream is to start and lead a software company.