ANALYZING AND CHARACTERIZING CODE CLONE LIFETIME

MD. JUBAIR IBNA MOSTAFA
Master of Science in Software Engineering
Institute of Information Technology, University of Dhaka
Class Roll: MSSE 0602
Session: 2018-19
Registration Number: 2013-912-011

A Thesis
Submitted to the Master of Science in Software Engineering (MSSE) Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

Analyzing and Characterizing Code Clone Lifetime

MD. JUBAIR IBNA MOSTAFA

Approved:

Signature	*Date*
Student: Md. Jubair Ibna Mostafa	
Supervisor: Dr. Kazi Muheymin-Us-Sakib *Professor*	
Examination Committee Head: Dr. Zerina Begum *Professor*	

To
Md. Golam Mostafa, my father
and
Mossammat Rokshana Tasmim, my mother

whose enormous love and courage is the living of my life

Abstract

Code clone is the duplication of source code, and cloning is a commonly used practice in software development. It has been found that a software repository may have 7 to 59% cloned code. Clone code may create maintenance problems like bug duplication and propagation, inconsistency among cloned codes etc. On the other hand, it can be used as a design choice to reuse stable features. Studying cloning practices is necessary to understand such phenomena.

In a repository, a clone code may exist in the production or test code, or different locations of source code. Knowing the evolving nature of clones from these aspects help to synthesize cloning practices. Moreover, a clone may exist in the repository for many versions or may be removed after some versions due to maintenance activities. Understanding clone, by analyzing the clone lifetime depicts the facts, that can be associated to characterize clone to identify which clones can be avoided and which require more attention during maintenance.

The aim of this research is to understand clone evolutionary facts to characterize clone lifetime. To do so, an empirical study has been conducted on four popular Java repositories such as Flink, Maven, Pig and Pulsar. In the study, clone occurrences are observed from source code type, clone location, and clone lifetime variability based on clone type and clone location. By detecting clones in subsequent versions and extracting clone genealogies, clone evolution history is categorized on those criteria. It has been found that test code clones are more in number than main code clones and the number of Intra-File clones is higher than Inter-File clones. Clone volatility based on the clone type, it has been found that Type-3 clones are more volatile than Type-1 and Type-2 clones, whereas based on file location, Intra-File clones are more volatile than Inter-File clones in clone genealogies. These observations facilitate to characterize clone lifetime using different metrics.

To characterize clone lifetime, clones are divided into two groups, namely, short-lived and long-lived based on a threshold of the number of versions a clone persists in the system. Then, some metrics, for example, interface similarity, clone context, co-change history, code type, clone location etc. are proposed to characterize clones according to their lifetime. By using these metrics, a Random Forest classifier is used to classify clones as short-lived and long-lived. The model achieved 0.80 – 0.92 AUC (Area Under the Curve) score to correctly classify short-lived clones for four software. Number of commits where all code fragments changed, method name similarity, number of parameters, clone locations are identified as influential metrics. The proposed metrics combined with existing metrics such as the line of code, number of clone siblings etc. increases classifier performance 2% – 7% than using only existing metrics for the studied software. Practitioners can leverage the proposed metrics to better characterize whether a clone will be short-lived or long-lived in the clone maintenance activities.

Acknowledgments

"All praise to almighty Allah"

I am grateful to Allah for getting the opportunity to complete my research. I am blessed to have professor Dr. Kazi Muheymin-Us-Sakib as my thesis supervisor. I would like to express my uninhibited gratitude to my supervisor for his direction, guidance, motivation, and advice throughout the research.

I would like to express my humble gratitude to the committee members of Masters program. Their continuous feedback helped me to research more rigorously and organize my thesis adequately. Along with committee members, I am also thankful to Distributed Science and Software Engineering (DSSE) research group members whose rigorous discussion on this research helped me to think from different perspectives and explore many things.

I am also thankful to Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No 56.00.0000.028.33.002.19.3 Date: 09-Jan-2019

List of Publications

1. *"An Empirical Study on Clone Evolution by Analyzing Clone Lifetime"* in Proceedings of the 13th IEEE International Workshop on Software Clones (IWSC) co-located with the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, pp. 20-26, February 24, 2019.
   DOI: 10.1109/IWSC.2019.8665850

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Code clones are duplicated code fragments in the source code. It is referred as one of the most prevalent code smells in software development [1]. In a software repository, 7 to 59% cloned code may exist [2]. Copying a code fragment and pasting it is a common practice in software development as developers try to reuse existing implementation without following a proper design. A clone can be created due to the developer's ignorance about the language features, having limited time to do it properly or not even caring about future maintenance problems [3]. It can also be created due to not having code authorship (using a third party library) [4].

As software evolves by incorporating new features and fixing bugs, clone code also evolves. So, the number of clone classes in subsequent versions should also be increased. Whether the increase is due to the main code clone or test code clone needs to be analyzed, since the test code clone is not as concerning as the main code clone. During this evolution, some clones may have gone through similar updates keeping consistency among clones, where as some may not, and again some may be refactored as well. Depending on the changes over clone, a clone may live for a small or large number of versions. Clone lifetime is referred as the number of versions a clone lived in the version history. Analyzing clone lifetime is important to know why clone lifetime varies for different clones. It can be helpful to characterize clones which require more attention during maintenance.

Evolutionary researches have found that removing all clones from a repository is neither beneficial nor feasible [5]. Research has found that 30 to 87% clones are short-lived and less likely to be changed consistently [6]. Whereas, long-lived clones changed consistently 25 to 37% times. Understanding clone by characterizing the clone lifetime can be helpful to decide which clones should get more attention during maintenance and which not. In this research, the evolutionary behaviors of clones are studied to know the factors of clone lifetime variations, and relevant metrics are proposed to characterize clone lifetime.

In this chapter, the motivation behind this work, research questions and contribution of the research have been included. Along with these, an overview of how the thesis has been organized is mentioned at the end.

1.1 Motivation

Conventional wisdom about clone is that the number of clones increases in subsequent versions. However, in the presence of test code, whether the clone increases in test code or main code is not investigated. To know the underlying cloning practices, this research focuses on clone evolution based on the main code and test code clone.

Similarly, re-using existing stable functionalities, code authorship, avoiding coupling between sub-systems are considered as the reasons for cloning. These reasons provide intuition that clones likely to happen among different modules or files. Whether cloning practices hold this intuition, this research focuses on location based clone evolution.

Research has found that many code clones exist in the system for only a short period of time. Extensive refactoring of such short-lived clones may not be beneficial as those clones could be introduced for experimental purposes [5]. Besides, if clones are changing independently without maintaining consistency among clones, those can also be avoided during maintenance activities [5, 6]. Characterizing clones based on clone lifetime helps to identify such clones. Researcher tried to incorporate different metrics in this characterization [7, 6]. Whether clone location, co-change history, interface similarity and clone contextual dependency are essential metrics to this characterization, is not yet been investigated.

A real life example is presented to understand clone lifetime and why characterizing clones will be beneficial for clone management is discussed. A clone class is lived for only 2 versions whereas a clone class is lived for 20 versions in Maven software. In the experimented environment, two code fragments (CF-1[1] and CF-2[2]) introduced as a Type-1 clone at Version 5. Listing 1.1 and Listing 1.2 show the duplicated fragments. These two code fragments survived as a clone up to Version 6. In the 7th version, the duplicated fragment is removed to another package

[1] `~/cloneLifetime/maven_versions/ver05-maven-3.0-alpha-6/maven-core/src/main/java/org/apache/maven/lifecycle/DefaultLifecycleExecutor.java952966`

[2] `~/cloneLifetime/maven_versions/ver05-maven-3.0-alpha-6/maven-core/src/main/java/org/apache/maven/lifecycle/DefaultLifecycleExecutor.java88103`

by refactoring [3]. So, the cloning relation is removed, and the clone class has the lifetime of 2 versions in the repository.

Listing 1.1: A Clone Class Lived for 2 Versions (CF-1)

```java
private RepositoryRequest getRepositoryRequest( MavenSession session, MavenProject
    project )
{
    RepositoryRequest request = new DefaultRepositoryRequest();

    request.setCache( session.getRepositoryCache() );
    request.setLocalRepository( session.getLocalRepository() );
    if ( project != null )
    {
        request.setRemoteRepositories( project.getPluginArtifactRepositories() );
    }
    request.setOffline( session.isOffline() );
    request.setForceUpdate( session.getRequest().isUpdateSnapshots() );
    request.setTransferListener( session.getRequest().getTransferListener() );

    return request;
}
```

Listing 1.2: A Clone Class Lived for 2 Versions (CF-2)

```java
private RepositoryRequest getRepositoryRequest( MavenSession session, MavenProject
    project )
{
    RepositoryRequest request = new DefaultRepositoryRequest();

    request.setCache( session.getRepositoryCache() );
    request.setLocalRepository( session.getLocalRepository() );
    if ( project != null )
    {
        request.setRemoteRepositories( project.getPluginArtifactRepositories() );
    }
```

[3] ~/cloneLifetime/maven_versions/ver07-maven-3.0-beta-1/maven-core/src/main/java/org/apache/maven/artifact/repository/DefaultRepositoryRequest.java88103

```java
    request.setOffline( session.isOffline() );
    request.setForceUpdate( session.getRequest().isUpdateSnapshots() );
    request.setTransferListener( session.getRequest().getTransferListener() );

    return request;
}
```

However, two code fragments (CF-1[4] and CF-2[5]) are introduced as a Type-1 clone at Version 2. Listing 1.3 and Listing 1.4 show the clone fragments. After 20 versions, one code fragment is removed from the *DefaultArtifactRepository.java* file at Version 21 and another code fragment exists up to the last version means evolve independently. As a result, this clone class has a lifetime of 20 versions.

This scenario can be happened for many clone classes. Different factors, for example, contextual coupling, clone location, change information etc. can be associated to this phenomena. Analyzing such clones and characterizing their lifetime can be helpful to know whether an introduced clone will be short-lived or long-lived. By this research, the clone management efforts will be well utilized.

Listing 1.3: A Clone Class Lived for 20 Versions (CF-1)

```java
public String toString()
{
    StringBuilder sb = new StringBuilder();

    sb.append( "      id: " ).append( getId() ).append( "\n" );
    sb.append( "     url: " ).append( getUrl() ).append( "\n" );
    sb.append( "  layout: " ).append( layout != null ? layout.getId() : "none"
        ).append( "\n" );

    if ( snapshots != null )
    {
        sb.append( "snapshots: [enabled => " ).append( snapshots.isEnabled() );
        sb.append( ", update => " ).append( snapshots.getUpdatePolicy() ).append(
```

[4]~/cloneLifetime/maven_versions/ver02-maven-3.0-alpha-3/maven-core/src/main/java/org/apache/maven/artifact/repository/MavenArtifactRepository.java130151

[5]~/cloneLifetime/maven_versions/ver02-maven-3.0-alpha-3/maven-compat/src/main/java/org/apache/maven/artifact/repository/DefaultArtifactRepository.java

```
            "]\n" );
    }

    if ( releases != null )
    {
        sb.append( " releases: [enabled => " ).append( releases.isEnabled() );
        sb.append( ", update => " ).append( releases.getUpdatePolicy() ).append( "]\n"
            );
    }

    return sb.toString();
}
```

Listing 1.4: A Clone Class Lived for 20 Versions (CF-2)

```
public String toString()
{
    StringBuilder sb = new StringBuilder();

    sb.append( "      id: " ).append( getId() ).append( "\n" );
    sb.append( "     url: " ).append( getUrl() ).append( "\n" );
    sb.append( "  layout: " ).append( layout != null ? layout.getId() : "none"
        ).append( "\n" );

    if ( snapshots != null )
    {
        sb.append( "snapshots: [enabled => " ).append( snapshots.isEnabled() );
        sb.append( ", update => " ).append( snapshots.getUpdatePolicy() ).append(
            "]\n" );
    }

    if ( releases != null )
    {
        sb.append( " releases: [enabled => " ).append( releases.isEnabled() );
        sb.append( ", update => " ).append( releases.getUpdatePolicy() ).append( "]\n"
            );
    }

    return sb.toString();
}
```

1.2 Research Questions

To analyze and characterize clone lifetime, how clone evolves in the version history is necessary to be studied. Besides, which metrics can characterize clone lifetime also need to be identified. Thus, this research tackles the following Research Questions (RQ) –

- **RQ1:** How do clones evolve in subsequent versions based on various clone features like code type, clone location and clone type?

  Four sub-questions are associated with this question, which are as follows.

  - **SQ1:** How do clones evolve in terms of the number of clones with respect to main or test code throughout the version history?

    From a software repository, code clones need to be detected. Clones are required to be categorized into main code and test code clones. From these categorized clones, version wise evolution will be observed.

  - **SQ2:** Is there any difference between Intra-File and Inter-File clones in terms of occurrences in the repository?

    From the previously detected clones, all clones need to be categorized into Intra-File and Inter-File clones. In these clones, version wise occurrence will be observed.

  - **SQ3:** Which type of clone tends to be more volatile in the clone genealogies?

    Clone genealogies are required to be extracted from already detected clones of a repository. Volatile clones need to be identified. These volatile clones will be categorized based on clone type.

  - **SQ4:** How does clone location impact volatile clone lifetime in the clone genealogies?

    From previously extracted clone genealogies, lifetime of each volatile clones needs to be calculated. These volatile clones will be categorized based on the location.

- **RQ2:** Which metrics can be introduced to characterize clone lifetime and how the metrics perform independently and combined with state-of-the-art metrics?

  To solve this problem, relevant metrics need to be identified. Performance evaluation and impact of metrics on performance also need to be calculated. This research question can be solved by answering the two sub-questions.

  - **SQ1:** Which metrics can be introduced to characterize clone lifetime and how the metrics perform to classify clones as short-lived and long-lived?

    The following steps will be followed to answer this sub-question.

    - Code clone detection need to be performed in a repository. At the same time clone genealogy need to be extracted. Clone lifetime will be calculated, and volatile clones need to be extracted from the genealogy.
    - Volatile clones will be labeled as short-lived and long-lived by performing clustering based on the lifetime.
    - Relevance metrics will be used to fit a classifier. Performance will be measured, and important metrics will be identified.

  - **SQ2:** How the classifier performs using the proposed metrics and the state-of-the-art metrics?

    Using the previously extracted metrics and the state-of-the art metrics a classifier will be built and results will be reported.

1.3 Contribution and Achievement

The contribution of this research is empirical findings of clone evolution and how well metrics perform to better characterize clone lifetime. Through an empirical study, clone lifetime is analyzed from the clone type, clone location and source code. After that, clone lifetime is characterized by incorporating some metrics like clone location, clone source code type and co-change information etc. Briefly, the major contributions of this research include –

- Four observations about clone evolution and clone lifetime. (Chapter 4)

  - The number of clone classes follows an increasing pattern with the development of the software. However, test code clones contribute the most to this increasing number of clones than the main code clone.

  - In location-based clone evolution, *Intra-File* clones occurred more in a repository than *Inter-File* clones.
  - Type-3 clones tend to be more volatile than other types (Type-1 and Type-2) clones in the repository.
  - *Intra-File* clones are more volatile than *Inter-File* clones in clone genealogies and their lifetime has a high frequency for 1 version.

- 16 metrics from 4 perspectives are proposed to characterize clone lifetime. Using these metrics, a random forest classifier can classify short-lived clones with 80 to 92% correctly, where clones' co-change information, method name dissimilarities, the number of parameters, clone located in the same file are identified as influential metrics to characterize clone lifetime. (Chapter 5)
- The proposed metrics along with state-of-the-art metrics can classify short-lived clones more accurately by increasing AUC score 2 to 7% than only existing metrics. (Chapter 5)

1.4 Organization of the Thesis

In this section, the organization of this thesis is provided to get a research roadmap. The organization of the thesis chapters has been mentioned in the followings.

Chapter 2: Background Study

This chapter describes the definitions of clone terminologies such as Code Fragment, Code Clone, Clone Pair and Clone Class. The types of clones, such as Type-1, Type-2, Type-3, and Type-4 are also defined. Along with these, clone evolution related concepts, for example, Clone Genealogy, Clone Lifetime etc. are briefly described to understand the evolution of clone. To facilitate understanding of clone lifetime characterization, concepts of clustering, classifier and performance measures are also described in this chapter.

Chapter 3: Literature Review

To know the state of clone evolutionary studies to understand cloning practices, analyze and characterize clone lifetime, existing works in the literature are discussed. The findings of those research along with their pros and cons are also mentioned. This chapter decorates with the presences of clone in large dataset, clone evolutionary patterns, clone changes, clone lifetime analysis and clone management related research in the literature.

Chapter 4: An Empirical Study on Clone Evolution

This chapter presents an empirical study on clone evolution from clone location, clone source code type. This also describes the lifetime of clone based on clone type, clone location which facilitate clone lifetime characterization using different metrics. In the end, the findings of the evolutionary study and the threats to validity are reported.

Chapter 5: Characterizing Clone Lifetime using Metrics

This chapter contains proposed metrics to characterize clones as, short-lived and long-lived clones, and the performance of those metrics to classify accordingly. Besides, influential metrics to this regards are also analyzed. It presents the comparison between existing metrics and proposed metrics to classify clones as short-lived and long-lived. At last, the threats to validity also mentioned in the chapter.

Chapter 6: Conclusion

This chapter summarizes the whole thesis, discusses the outcome of this research and highlights future research directions which can be performed to comprehend clone practices and support clone management activities.

Chapter 2

Background Study

Usually, in software development, code clone is a technique to take advantages of existing implementations. The process of duplicating a source code is called code cloning and duplicated code fragments are called cloned code. It can be created due to organizational limitations like time limitation of code delivery, or code ownership issues [4]. Although it helps in the development phase, during maintenance it incurs problems to extend functionalities, modify existing features or fix a bug for duplication. It also costs more effort and time to inspect all cloned code which requires the same updates. Understanding clone by observing clone evolution provides evolutionary facts that can support maintainer to take decisions which clone needs more maintenance efforts.

To maintain clone, different clone management activities need to be performed, for example, clone tracking, refactoring etc. Since a repository may contain 19 to 59% clone code, it is essential to know which clones are important and which are not during maintenance. Analyzing clone lifetime can help to characterize such clones. Characterization of clones help to build a classifier which can be used to predict clone life expectancy. In this chapter, terminologies of code clone, clone types and evolutionary concepts of clone are described. Along with these, some performance measure concepts, classifier and feature importance are also described.

2.1 Concepts of Code Clone

Code clone related concepts are essential to understand the development and maintenance of clone. In this section, code clone related terms and terminologies like code fragment, code clone, clone types, clone genealogy, clone lifetime etc. are described.

Code Fragment

A consecutive sequence of statements denotes a Code Fragment (*CF*) [8] as shown in Listing 2.1. It can be a function, block or sequence of statements. In a code fragment, zero to many comments can be found. A *CF* is identified by its position in the source code, – can be identified by the start and end line number within a file. If 'f' is a file, 's' and 'e' is the start and end line number respectively, a code fragment is denoted by a triple *CF=(f, s, e)* [9].

Listing 2.1: Code Fragment (CF)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0;
    for (int i=1; i<=n; i++){
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

2.1.1 Code Clone

If a code fragment is similar to another code fragment for a given function of similarity, for example, exact similar, partially similar etc., it is called a code clone [10]. If CF_1 and CF_2 are two code fragments and ϕ is the similarity function, triple (CF_1, CF_2, ϕ) denotes a clone. Here, ϕ is responsible for creating different types of clone. Listing 2.2 and Listing 2.3 show an example of code clone.

Listing 2.2: Code Clone (CF-1)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0; //C
    for (int i=1;i<=n;i++){
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

Listing 2.3: Code Clone (CF-2)

```
void sumProd(int n) {
    float sum=0.0;
    float prod =1.0;
    for (int i=1;i<=n;i++)
    {
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

Clone Pair

A pair of code fragments which are similar to each other formed clone pair. In other words, A group of two similar CFs is called a clone pair. It can be found most frequently in a repository because developers frequently copy some fragments one or few times. A clone pair can be specified by a set of code fragments S$\{CF_1, CF_2\}$ where both fragments are clone to each other. Here, Listing 2.4 and Listing 2.5 represent a clone pair.

Listing 2.4: Clone Pair (CF-1)

```
void sumProd(int n){
    float s=0.0; //C1
    float p =1.0;
    for (int j=1;j<=n;j++){
        s=s + j;
        p = p * j;
        foo(s, p);
    }
}
```

Listing 2.5: Clone Pair (CF-2)

```
void sumProd(int n){
    float s=0.0; //C1
    float p =1.0;
    for (int j=1;j<=n;j++){
        s=s + j;
        p = p * j;
        foo(p, s);
    }
}
```

Clone Class

A set of code fragments syntactically similar or identical to each other is called clone class. It can also be called clone group. A clone class can be specified by the tuple $(CF_1, CF_2, CF_3, ...,$

CF_n, ϕ), where, ϕ is the similarity function. Here, each pair of the distinct fragment is a clone pair: (CF_i, CF_j, ϕ), i, j ϵ 1...n, i$\neq$j [9] of the clone class.

A clone class is created due to duplication of a code fragment. A clone class is important to know because it provides information about all clone instances of the code fragment. In case of change consistency, it eases the task to update all clone instances. Listing 2.6 – 2.9 present a clone class that consists of four clone fragments. The members of a clone class are called clone siblings to each other.

Listing 2.6: Clone Class (CF-1)

```
void sumProd(int n){
    float s=0.0; //C1
    float p =1.0;
    for (int j=1;j<=n;j++){
        s=s + j;
        p = p * j;
        foo(s, p);
    }
}
```

Listing 2.7: Clone Class (CF-2)

```
void sumProd(int n){
    float s=0.0; //C1
    float p =1.0;
    for (int j=1;j<=n;j++){
        s=s + j;
        p = p * j;
        foo(p, s);
    }
}
```

Listing 2.8: Clone Class (CF-3)

```
void sumProd(int n)
{
    int sum=0; //C1
    int prod =1;
    for (int i=1;i<=n;i++)
    {
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

Listing 2.9: Clone Class (CF-4)

```
void sumProd(int n)
{
    float sum=0.0; //C1
    float prod =1.0;
    for (int i=1;i<=n;i++)
    {
        sum=sum + (i*i);
        prod = prod*(i*i);
        foo(sum, prod);
    }
}
```

2.1.2 Clone Type

Code fragments can be similar to each other based on syntactic and/or semantic similarity. Syntactically similar clones mean code fragments are similar to each other based on the source code structure without considering their functionalities. On the contrary, semantically similar clones mean code fragments are similar based on the functionality those performed, not the code structure of those fragments. Basically, the syntactic similarity is the result of copying a code fragment and pasting it into another location of source code. Based on syntactic similarity, code clones are divided into three types namely Type-1, Type-2 and Type-3. The semantically similar clone is the fourth type (Type-4) clone. Definition and example of these four types of clone are given below.

Type-1

Two code fragments those are identical to each other excluding the differences in white-space, layout, and comments are called Type-1 clones [10]. It is also called Exact clone as clone fragment is the exact copy of the original code fragment. Type-1 clone often occurs, because, developers simply copy a required code fragment and paste it without any changes. Listing 2.10 and Listing 2.11 denote a Type-1 clone.

If CF_1 and CF_2 are two code fragments, tuples(CF_1, CF_2, ϕ) represents the exact clone if ϕ is a function of exact similarity without white-space, layout and comments. CF_1 and CF_2 are called Type-1 clone instances.

Listing 2.10: Clone Type-1 (CF-1)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0; //C
    for (int i=1;i<=n; ++){
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

Listing 2.11: Clone Type-1 (CF-2)

```
void sumProd(int n) {
    float sum=0.0;
    float prod =1.0;
    for (int i=1;i<=n;i++)
    {
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

Type-2

If two code fragments are similar to each other except the differences in identifier names, literal values, white-space, layout and comments are called Type-2 clones [10]. It is also referred to as Parameter-Substitute or Renamed clone. Usually, developers create Type-2 clones by copying a code fragment and modifying it accordingly in the required context. Listing 2.12 and Listing 2.13 show an example of Type-2 clone where variables are different from one fragment to another.

If CF_1 and CF_2 are two code fragments, tuples(CF_1, CF_2, ϕ) represents renamed clone if ϕ is a function of similarity which allows differences in identifier names, literal values, and Type-1 differences. These CF_1 and CF_2 are called Type-2 clones.

Listing 2.12: Clone Type-2 (CF-1)

```
void sumProd(int n){
    float s=0.0; //C1
    float p =1.0;
    for (int j=1;j<=n;j++){
        s=s + j;
        p = p * j;
        foo(p, s);
    }
}
```

Listing 2.13: Clone Type-2 (CF-2)

```
void sumProd(int n) {
    int sum=0; //C1
    int prod =1;
    for (int i=1;i<=n;i++){
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
    }
}
```

Type-3

Two code fragments are syntactically similar enough to each other considering variations at the statement level (one or more statement differences between two CFs) including Type-1 and Type-2 clone differences is called Type-3 clone [10]. This clone also termed as Gapped clone [2] as some statements can be absent to one of the fragments. Modification in statement level creates this kind of clones, for example, one fragment may have statements added, deleted or modified with respect to other ones. Listing 2.14 and Listing 2.15 show an example of a Type-3 clone where one line is deleted in the second fragment.

If CF_1 and CF_2 are two code fragments, tuples(CF_1, CF_2, ϕ) represents Gapped clone if ϕ is a function of similarity which allows differences at statement level along with Type-1 and Type-2. These CF_1 and CF_2 are called Type-3 clones.

Listing 2.14: Clone Type-3 (CF-1)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0;
    for (int i=1;i<=n;i++)
    {   sum=sum + i;
        prod = prod * i;
        foo(prod);
    }
}
```

Listing 2.15: Clone Type-3 (CF-2)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0;
    for (int i=1;i<=n;i++)
    {
        sum=sum + i;
        //line deleted
        foo(sum, prod*i);
    }
```

Type-4

Two code fragments perform the same functionality but the implementation of those fragments are similar and/or different syntactically is called Type-4 clone [10]. This type of clone refers as semantically similar clones. Listing 2.16 and Listing 2.17 represent a Type-4 clone where both fragments perform the same functionality but those differ in syntactically.

If CF_1 and CF_2 are two code fragments, tuples(CF_1, CF_2, ϕ) represents semantic clone if ϕ is a function of semantic similarity, that is, functionally similar. These CF_1 and CF_2 are called Type-4 clone instances.

Listing 2.16: Clone Type-4 (CF-1)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0;
    for (int i=1;i<=n;i++)
    {
        prod = prod * i;
        sum=sum + i;
        foo(sum, prod);
    }
}
```

Listing 2.17: Clone Type-4 (CF-2)

```
void sumProd(int n) {
    float sum=0.0; //C1
    float prod =1.0;
    int i=0;
    while (i<=n){
        sum=sum + i;
        prod = prod * i;
        foo(sum, prod);
        i++ ;
    }
}
```

Clone types define how similar clone fragments are with each other. Depending on the clone type, clone codes stability may differ from one another. Moreover, different clone types reflects the cloning practices of developers. To know the evolutionary implications, it is required to better understand clone types.

2.2 Clone Evolutionary Terminologies

Clone evolution denotes version wise overall scenario of clone for a software. At the same time, it represents the development and maintenance of clones. In case of clone evolution, a cloned code fragment is denoted by its version number, file name, start and end line number to next version location of that fragment. In successive versions, clone location can be changed due to new code addition, existing code deletion or modification. As an example, Figure 2.1 shows clone evolution for three versions where circles represent clone fragments and rectangles represent clone classes. Below clone evolution and related concepts such as Clone Genealogy, Clone Lifetime etc. are briefly discussed.

2.2.1 Clone Genealogy

A Clone Genealogy is a set of clone lineages which describe how an individual clone or a clone class evolves throughout the version history [11]. A clone genealogy holds evolution history of a clone along with clone class. Figure 2.1 shows an example of clone genealogies for three clone classes. From the figure, it can be seen that 1st clone class evolves from the first version to the third version, whereas 2nd clone class evolves up to the second version.

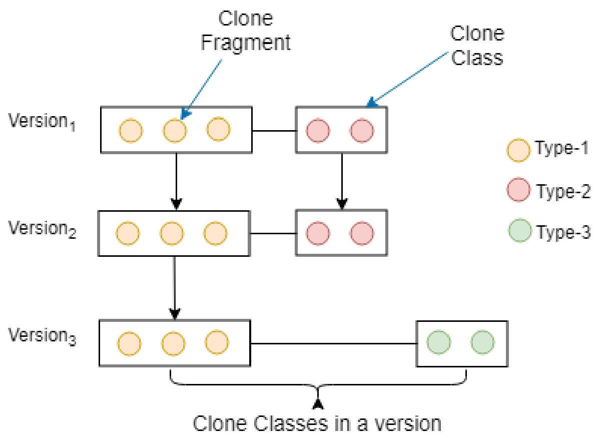


Figure 2.1: Demonstration of Clone Genealogy

Volatile Clone

A code clone which has been removed or dead from its introduce version to before the last version in the version history is called a Volatile Clone [11]. Figure 2.2 denotes one volatile clone which lived for 2 versions. Analysis of volatile clones is important to know the underlying reason that is why the clone has gone through such activity.

Clone Lifetime

The number of versions a clone exists in the repository is called its lifetime. A clone removal from the repository can be happened by refactoring or subsequent changes within clone that ultimately loss clone relation with other fragments of that class. For both cases, clone mapping (version to version) is lost for a clone and denotes as a dead genealogy. Figure 2.2 represents an example of clone lifetime where 2nd clone class has lifetime of 2. Similarly, 6th clone has the lifetime of only one version.

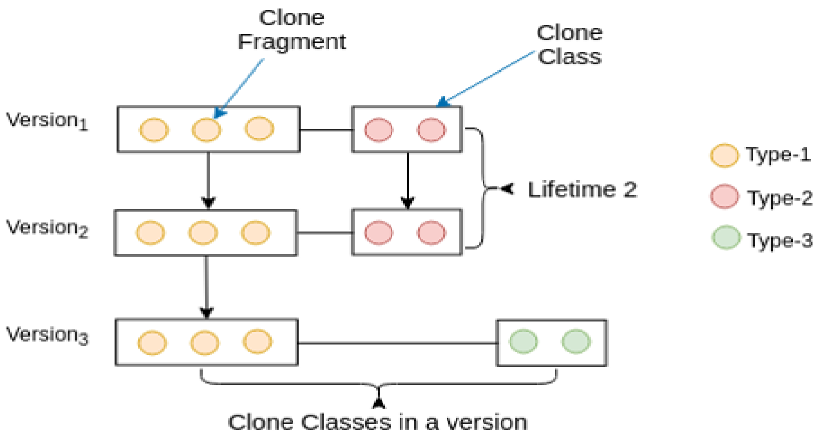


Figure 2.2: Demonstration of Clone Lifetime

2.2.2 Clone in Main and Test Code

Source code is a list of human-readable instructions that developers write to perform some functionalities by a computer. In software development, source code is the ultimate products which serve the customer needs. Developers write both main and test code where the main code is delivered to the customer and test code is written to test the implemented functionalities. Clone in main code vs test code may differ based on maintenance perspective. Below main code clone and test code clone is defined –

Main Code Clone

The part of the source code that contains the logic of the project and is run in the production means the end product that will be used to interact with the users is called Main Code or Production Code [12]. If two code fragments from the main code are clone to each other then it is called Main Code Clone. For example, Listing 2.18 and Listing 2.19 show an main code clone from Apache Pig Software [1] [2]. This is an example of Type-2 clone as identifier types are different at line 7.

Listing 2.18: Main Code Clone (CF-1)

```
1   public Double exec(Tuple input) throws IOException {
2       if (input == null || input.size() < 2)
3           return null;
4
5       try{
6           Double first = DataType.toDouble(input.get(0));
7           Double second = DataType.toDouble(input.get(1));
8           return Math.max(first, second);
9       } catch (NumberFormatException nfe){
10          System.err.println("Failed to process input; error - " + nfe.getMessage());
11          return null;
12      } catch(Exception e){
13          throw new IOException("Caught exception in MAX.Initial", e);
14      }
15
16  }
```

Listing 2.19: Main Code Clone (CF-2)

```
1   public Double exec(Tuple input) throws IOException {
2       if (input == null || input.size() < 2)
3           return null;
4
5       try{
6           Double first = DataType.toDouble(input.get(0));
```

[1] pig/contrib/piggybank/java/src/main/java/org/apache/pig/piggybank/evaluation/math/MAX.java 69 88
[2] pig/contrib/piggybank/java/src/main/java/org/apache/pig/piggybank/evaluation/math/SCALB.java 72 87

```java
            Integer second = DataType.toInteger(input.get(1));

            return Math.scalb(first, second);
        } catch (NumberFormatException nfe){
            System.err.println("Failed to process input; error - " + nfe.getMessage());
            return null;
        } catch(Exception e){
            throw new IOException("Caught exception in MAX.Initial", e);
        }
    }
```

Test Code Clone

The part of the source code that contains the tests which verify if the Production Code work as expected in other words the application works as expected is called Test Code [12]. If two code fragments from Test Code are clone to each other then it is called Test Code Clone. For example, Listing 2.20 and Listing 2.21 shows an test code clone from Apache Pig Software [3] [4]. This is an example of Type-3 clone.

Listing 2.20: Test Code Clone (CF-1)

```java
//check that the foreach projection map has null mappedFields
LOForEach foreach = (LOForEach)lp.getLeaves().get(0);
ProjectionMap foreachProjectionMap = foreach.getProjectionMap();
assertTrue(foreachProjectionMap.changes() == true);

MultiMap<Integer, Pair<Integer, Integer>> foreachMapFields =
    foreachProjectionMap.getMappedFields();
assertTrue(foreachMapFields != null);
```

Listing 2.21: Test Code Clone (CF-2)

```java
//check that the foreach projection map has non-null mappedFields
LOForEach foreach = (LOForEach)lp.getLeaves().get(0);
ProjectionMap foreachProjectionMap = foreach.getProjectionMap();
```

[3] pig/test/org/apahce/pig/test/TestProjectionMap.java 1023 1029
[4] pig/test/org/apahce/pig/test/TestProjectionMap.java 1289 1295

```
    assertTrue(foreachProjectionMap.changes() == true);

    MultiMap<Integer, Pair<Integer, Integer>> foreachMapFields =
        foreachProjectionMap.getMappedFields();
    assertTrue(foreachMapFields != null);
```

2.2.3 Clone Location

In a software repository, clone code can be found within module, package, file, class, and method. The location of clone denotes the sparsity of duplication. In the following, the clone is defined based on the file location as Chapter 4 presents some evolutionary observations.

Intra-File Clone

A clone class which consists of clone siblings located in only one file is called Intra-File Clone. For example, Listing 2.22 and Listing 2.23 show an Intra-File code clone from Apache Maven Software [5] [6]. This kind of clones can be happened within a large file which contains many similar functionalities.

Listing 2.22: Intra-File Clone Fragment 1

```
public void testModelAndFactory()
{
    MavenSession session = mock( MavenSession.class );
    MavenExecutionRequest executionRequest = new DefaultMavenExecutionRequest();
    Map<String, List<ToolchainModel>> toolchainModels = new HashMap<String,
        List<ToolchainModel>>();
    toolchainModels.put( "basic", Arrays.asList( new ToolchainModel(), new
        ToolchainModel() ) );
    toolchainModels.put( "rare", Collections.singletonList( new ToolchainModel() )
        );
    executionRequest.setToolchains( toolchainModels );
    when( session.getRequest() ).thenReturn( executionRequest );

```

[5]maven/maven-core/src/test/java/org/apache/maven/toolchain/DefaultToolchainManagerTest.java 102 112
[6]maven/maven-core/src/test/java/org/apache/maven/toolchain/DefaultToolchainManagerTest.java 118 128

```
    List<Toolchain> toolchains = toolchainManager.getToolchains( session, "rare",
        null );

    assertEquals( 1, toolchains.size() );
}
```

Listing 2.23: Intra-File Clone Fragment 2

```
public void testModelsAndFactory()
{
    MavenSession session = mock( MavenSession.class );
    MavenExecutionRequest executionRequest = new DefaultMavenExecutionRequest();
    Map<String, List<ToolchainModel>> toolchainModels = new HashMap<String,
        List<ToolchainModel>>();
    toolchainModels.put( "basic", Arrays.asList( new ToolchainModel(), new
        ToolchainModel() ) );
    toolchainModels.put( "rare", Collections.singletonList( new ToolchainModel() )
        );
    executionRequest.setToolchains( toolchainModels );
    when( session.getRequest() ).thenReturn( executionRequest );

    List<Toolchain> toolchains = toolchainManager.getToolchains( session, "basic",
        null );

    assertEquals( 2, toolchains.size() );
}
```

Inter-File Clone

A clone class which consists of clone siblings spanning more than one file is called Inter-File Clone. For example, Listing 2.24 and Listing 2.25 show an Inter-File code clone from Apache Maven Software [7] [8]. Inter-File clones likely to be happened by copying other modules features.

Listing 2.24: Inter-File Clone Fragment 1

[7]maven/maven-aether-provider/src/main/java/org/apache/maven/repository/internal/DefaultVersionRangeSolver.java 238 249

[8]maven/maven-aether-provider/src/main/java/org/apache/maven/repository/internal/DefaultVersionSolver.java 304 315

```java
private void invalidMetadata( RepositorySystemSession session, Metadata metadata,
    ArtifactRepository repository,
                              Exception exception )
{
    RepositoryListener listener = session.getRepositoryListener();
    if ( listener != null )
    {
        DefaultRepositoryEvent event = new DefaultRepositoryEvent( session, metadata );
        event.setException( exception );
        event.setRepository( repository );
        listener.metadataInvalid( event );
    }
}
```

Listing 2.25: Inter-File Clone Fragment 2

```java
private void invalidMetadata( RepositorySystemSession session, Metadata metadata,
    ArtifactRepository repository,
                              Exception exception )
{
    RepositoryListener listener = session.getRepositoryListener();
    if ( listener != null )
    {
        DefaultRepositoryEvent event = new DefaultRepositoryEvent( session, metadata );
        event.setException( exception );
        event.setRepository( repository );
        listener.metadataInvalid( event );
    }
}
```

2.3 Clustering

Clustering is a Machine Learning technique that is used to group data points [13]. In theory, data points of a group should have similar properties, while data points in different groups should have highly dissimilar properties. It is a common method to reduce the number of data points or to infer similar properties to a group of data. Different clustering methods such as K-means clustering [13], Agglomerative Hierarchical clustering [13] etc. exist to cluster data points. In this research, clustering will be used to label volatile clones based on their lifetime in Section 5.2.2. Below K-means clustering with the best number of cluster is briefly discussed.

K-means Clustering K-means clustering is one of the most commonly used techniques for its simplicity and easy to use [14]. Broadly speaking, it works by selecting as many points as the number of desired clusters to create initial centers. Then, each observation is associated with the nearest center to create temporary clusters. Temporary clusters gravity center is calculated and each observation is reallocated to the nearest cluster which has the closest center. This procedure is iterated until convergence. K-means clustering is used to label volatile clone instances as a short or long lived clone in Section 5.2.2.

Best Number of Cluster To cluster dataset, determining the best number of cluster is difficult and randomly setting a number may lead to not optimal cluster. To determine the best number of clusters, R library provides NbClust function which comprises of 30 indices for determining the number of clusters. It provides the best clustering scheme from the different results obtained by varying all combinations of the number of clusters, distance measures, for example, Euclidean distance, and clustering methods, for example, K-means [15]. This NbClust is used to get the best number of cluster based on the lifetime of each volatile clone.

2.4 Performance Measures

To characterize clone lifetime, some performance measures of data science, for example, True Positive Rate, False Positive rate and Area Under the Curve etc. are used. Below each of these measures is described briefly.

TRP True Positive Rate (TRP) is the measure of predicting positive class instances as positive from the total number of positive class instances. For example, if the total number of short-lived clones is 100 and a model predict 90 short-lived clones as short-lived, the TRP is 90%. It is also known as *Recall* or *Sensitivity* [9]. The equation of TPR is given below –

[9] https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology

$$TPR/Recall/Sensitivity = \frac{TP}{(TP + FN)} \tag{2.1}$$

FPR False Positive Rate (FPR) is the measure of predicting a positive class instance as positive while it is actually a negative instance 9. For example, if the total number of long-lived clones is 100 and a model predicts 10 clones as short-lived, the FPR is 10%. The equation of TPR is given below –

$$FPR = \frac{FP}{(TN + FP)} \tag{2.2}$$

TNR True Negative Rate (TNR) is the measure of predicting negative class instances as negative while it is actually negative instances. For example, if the total number of long-lived clones is 100 and a model predicts 80 clones as long-lived, the TNR is 80% 9. It is also known as *Specificity.* The equation of TPR is given below –

$$Specificity = \frac{TN}{(TN + FP)} \tag{2.3}$$

ROC ROC (Receiver Operating Characteristic) is a probability measure curve. It is generated by plotting the True Positive Rate (y-axis) against the False Positive Rate (x-axis) by defining various threshold 9.

AUC AUC is a performance measurement for the classification problem at diverse thresholds settings. It represents the measure of separability between classes [16]. It depicts how much a model is capable of distinguishing between classes. It ranges from zero (0) to one (1). Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s. A model has excellent separability if its AUC value is near to 1 whereas AUC near to 0 means it has the worst measure of separability. And when AUC is 0.5, it means a model has no separability. For example, if a model has AUC 0.8, it means there is an 80% chance that the model will be able to distinguish positive class as a positive and negative class as a negative instances. In this research, AUC is used to measure the performance of a Random Forest classifier to classify a clone as short-lived and long-lived in Section 5.2.4.

2.5 Classification Concepts

Classification is the technique of predicting the class of given data points [10]. Predicting classes are also called as targets/ labels or categories. Classification belongs to the category of supervised learning where the classes also provided with the given data. This can be binary, ternary or multi-class classification. There are different classification models exist in the literature, for example, Decision Tree (DT), Random Forest (RF) [17] etc. Below a short description of RF classifier and Feature importance are given –

Random Forest Classifier

A random forest classifier is a combination of multiple decision tree classifiers where each tree uses a random sub-samples of the dataset to fit a predictive model. Each tree votes for a class and the most voted class is the outcome of the random forest classifier [17]. It is robust from over-fitting and outlier problem [18] with good overall accuracy. In this research, RF classifier is used to classify volatile clones in either short-lived or long-lived clones.

Feature Importance

The feature importance score measures the contribution from the feature to the classifier. It denotes how important the feature is to discriminate multiple classes. In this analysis, feature importance is used to compute metrics influence. It can be calculated based on the impurity reduction of the class due to the feature or mean decrease in accuracy of the classifier for removing the feature [18]. For example, a classifier is built to classify clones as short-lived and long-lived by using only three features namely, clone type, number of co-change and line of code. From feature importance, it can be known which of these features are best to classify clones. In this research, feature importance will be used to know the importance of proposed metrics in Subsection 5.2.3 to classify clones as short-lived and long-lived in Subsection 5.2.4.

2.6 Summary

To understand this research and existing research in the literature, it is required to have a preliminary knowledge of code clone related concepts. Code clone, types of clone and clone evolutionary terminologies are described. Besides, random forest classifier, feature importance,

[10]https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

performance measure concepts are briefly described to ease the understanding process. In the proceeding chapter, clone evolutionary research and the state-of-the-art techniques for characterizing clone lifetime are described elaborately.

Chapter 3

Literature Review

Since code clone has controversial impacts on development and maintenance, researchers conducted several studies on clone from different perspectives such as clone detection [9], clone evolution [5], clone management [19, 20] etc. Clone detection research focuses on how accurately clone can be detected from the source code to know the presence of clone in different systems [21, 22, 9]. Whereas clone evolutionary research identifies how clones evolve throughout the version history to establish observations [23, 5, 6]. These observations can be used to manage the cloned code better. So, research community focuses on efficient clone management by incorporating evolutionary observations [7]. Besides, clone management research focuses on clone tracking [24], refactoring and prioritizing clones for refactoring [25, 26, 27, 28] etc. This research has also been conducted towards clone management by analyzing and characterizing clone lifetime [6]. To do so, clone genealogy, clone changes and clone lifetime related research need to be studied. In this research, exhaustive literature survey has been done, and, the whole knowledge base is divided into four following perspectives –

- Clone Occurrences Analysis
- Clone Genealogy Analysis
- Clone Change Analysis
- Clone Lifetime Analysis

In clone occurrences research, the existences of clone in large, open source and commercial systems are observed to know cloning practices in different systems. Clone genealogies research tries to extract evolutionary facts, laws and observations to understand clone evolution.

Whereas, in clone change analysis, researcher debates about clone changes, error-proneness and harmfulness of cloning towards clone management. Lastly, clone lifetime analysis focuses on clone survivability to better characterize clone based on management activities. Each of these perspectives is elaborately discussed, limitations of those studies are investigated and relevance or differences with this research are mentioned in the following sections.

3.1 Clone Occurrences Analysis

Clone occurrences research uses clone detection tools and explores clone existences in different systems empirically [29, 30, 21, 9, 31]. This researches find whether cloning is prevalent to a system or it is negligible to consider as a threat. Moreover, these studies discover, which type of application, language or modules of a software are more prone to cloning. Clone occurrences research facilities practitioners by providing evidences that clone should be considered as a bad (sometimes good [32]) practice, which should get attention during maintenance activities.

To understand the presences of code clones, several studies have been conducted to find the number of clones exists in large software [29, 33, 31]. Kamiya et al., developed the popular *CCFinder* clone detector to know the presence of clone in large systems like Linux, FreeBSD and NetBSD. CCFinder transforms source code into streams of tokens by using some transformation rules and a language specific lexical analyzer. For this transformation, the tool is capable of detecting clone in different languages such as C, C++, Java and COBOL etc by over coming syntactic differences. It substitutes identifiers/parameters by a special token which allows detecting Type-2 clones. Afterward, token sequences are matched with each other to detect clone pairs. They found that FreeBSD and NetBSD operating systems have 40.1% clone files whereas Linux and NetBSD have 3.1% clone files.

Using CCFinder, Raihan et al., conducted empirical study on 9 text-editors and 8 X-Windows window managers written in C and C++ to validate clone as a re-using technique [33]. They found that on an average 8% files in the text-editors and 2.2% files in window managers contain cross project clones. They identified most of those cloning were happened accidentally or to keep consistency with the libraries. On the other hand, recently, a large scale open source C and C++ projects were analyzed by Koschke et al. [34]. They studied 7800 C, C++ projects on Github, and found that 80% of those projects consist of Type-2 clones [34], denotes a significant number of projects contain clones.

Lopes et al., studied 4.5 million Java, C++, Python, and JavaScript projects on Github to establish a mapping of clones in open source community [31]. They found that 70% of the code on GitHub consists of clones of previously created files. They also studied language wise clone occurrences, to know whether the number of clones variate in different languages. They found that JavaScript systems have a larger number of clones than Java systems where only 6% of the JavaScript files were distinct [31]. While these studies focused on clone coverage in different systems, this study focuses on evolutionary aspects of clone from main code clones and test code clones.

Research has also conducted to identify the reasons behind code cloning. Studies have found that following an already implemented solution is better than creating a new design and abstracting duplicates into a single one [33, 32, 4]. Sometimes developers create clones for learning and experimentation purposes [4]. It has also been found that, unintentional clones happened for following a given API or set of libraries, which require similar invocations [33]. Time restriction to deliver the product or following a proper design, also incur cloning. Whereas those research focused on reasons for cloning, this research focuses on cloning practices in evolving software based on location and source code type.

3.2 Clone Genealogies Analysis

Clone genealogies analysis research focuses on deriving evolutionary patterns of clone [23, 5, 35]. To extract clone genealogy, each clone is required to map subsequent versions. From the genealogy, a code fragment's (which is copied multiple times) introduce as a clone, changes with its similar ones, and removal from the system can be explored. These researches characterize clone evolution within a system.

The first clone evolution research was performed by Antoniol et. al., in Linux Kernel [23] by analyzing 19 releases (from 2.4.0 to 2.4.18). They used a metric based clone detector and observed clone ratios in those releases. In the study, authors found that Linux Kernel contain low number of clones and the number of clones was stable across releases. This indicates a stable structure of Linux and clones evolve without degradation. However, this study did not consider the evolving pattern of clone in a clone class or lifetime of a clone genealogy.

Kim et al. defined a model of code clone genealogies and provided patterns of clone evolution [5]. In this study, authors derived six patterns of clone evolution namely same, add, subtract, consistent change, inconsistent change and a shift in clone class. These six patterns are mostly related to the number of siblings in a clone class. They also divided entire clone evolution into consistently changing clones, volatile clones, locally unfactorable clones and long-lived clones. Observing two systems namely carol and dnsjava, authors found 38% and 36% clone genealogies followed the consistently changing pattern. The stability behavior of clone in those subject systems showed that clones were volatile. Within 5 to 10 versions, almost 50% clone disappeared and changed independently. Using standard refactoring techniques like pull up a method, replace conditional with polymorphism etc., can be used to refactor locally. Whereas, 64% in carol and 49% in dnsjava comprised locally unfactorable clone class due to some constraints such as language limitation. Most of the clone genealogies of long-lived clones were locally unfactorable.

Bakota et al. also studied the clone evolution pattern by tracking clone in subsequent versions [36]. They defined Vanished Clone Instance (VCI), which indicates clone instance was not mapped onto any subsequent versions, Occurring Clone Instance (OCI), smell which means a clone has found in a version that was not derived from the previous version, Modified Clone Instance (MCI), means a clone instance might have been modified in a way which became a clone instance to another class, and lastly, Migrating Clone Instance (MGCI), means clone instance move into another clone class from its original clone class for some changes.

Thummalapenta et al., analyzed clone evolution in four different Java and C software, to know, whether clones evolve independently or consistently [35]. They identified the relationship of these patterns with clone radius, clone size, and the kind of change within clones i.e., corrective maintenance or enhancement. Using CCFinder, Simscan [29, 37] as clone detectors and diff [38] as clone change tracker, they quantitatively studied those systems. They defined four evolution patterns, namely, COnsistent Evolution (CO), Late Propagation (LP), Delayed Propagation and Independent Evolution (IE). They found that over 80% of the clones undergo either IE or CO for different software and IE is more frequent than CO for class and method level clones. They did not find any strong relation with the clone radius or size with the evolution patterns. Lastly, they found that bug fixing changes (corrective maintenance) occur for clones exhibiting late propagations.

Although these studies focus on the clone classes evolutionary patterns by means of the number of clone instances and how they interactively changed with the system, this thesis focuses on the overall number of clone occurrences in subsequent versions by considering source code type, clone location. Besides, volatility of clones are studied based on their clone types and clone location. Moreover, this thesis focuses on characterizing clone's lifetime by incorporating different metrics where the findings of the evolutionary study also used.

3.3 Clone Change Analysis

During the clone evolution, clone codes may be changed or remain the same from its origin. A clone's origin to end is denoted as the lifetime of the clone. Depending on the aliveness, a clone can be live for a short period of time, termed as, short-lived clone and live a long period of time, termed as, long-lived clone. Moreover, clones in their lifetime may face consistent and inconsistent changes, denoted as, consistently-changed clones. Literature focuses on these changes to facilitate clone change management tasks such as tracking clones.

Several research studied clone changes in evolving software [39, 40, 8]. Krinke analyzed consistent and inconsistent changes in clone by studying five open source software [39]. The author showed that usually half of the changes in clone classes were inconsistent changes. To select desirable clones for refactoring, consistently changed clones are more important than inconsistently changed clones [41, 42] because consistent changes require extra effort to update a change to all similar clones. In this study, the author used CVS repositories and took 200 weeks of version history to analyze clone changes. The simian tool [43] was used to identify the clone and clone classes. To get the version wise change information of a repository, Diff tool [38] was used. After analyzing those repositories, the author found that roughly 45% - 50% changes in clone class were consistent and others were inconsistently changing clone classes. The author concluded that inconsistently changed clones require tracking to monitor and update changes into all clone instances and consistently changed clones should be refactored if possible.

Barbour et al. conducted an empirical study on late propagation of changes [44]. In a situation where one of the clone fragments of a clone class undergoes inconsistent changes then other fragments need to be changed accordingly to re-synchronize the changes is referred to as late propagation. They studied two open source software namely ArgoUML and Ant where, they identified 8 types of late propagation. From these, 2 types of late propagation are most prone to induce bugs, one is, when a clone experiences inconsistent changes and then a re-synchronizing change without changes to other fragments, and another one is, when two clones undergo an inconsistent modification followed by a re-synchronizing change that modifies both the clone fragments in a clone pair.

Considering revision level as chaotic and experimental development process, Bettenburg et al. conducted a release level empirical study on clone changes for long-lived clones [45]. Authors used two open source software namely Apache Mina and jEdit to study a coarse-grained level long-lived clones characteristics, effects of inconsistent changes to clones and types of cloning patterns at release level. In that study, authors used SimScan [37] to detect clones in different releases which uses Clone Region Descriptor (CRD) [24] to map clone classes in subsequent versions. CRD uses an abstract representation of a clone region. It is more robust to map changes of clone classes between subsequent versions. Changes were classified into one of the patterns of forking, templating etc reported by Kapser et al. [32]. Authors findings confirmed that many clones were short-lived and changed independently over time. Long-lived clone class survived on an average 6.79 releases and contained 2.34 clones.

Although these studies focused on clone changes considering consistent, inconsistent and late propagation changes, they did not analyze and characterize clone lifetime. However, these clone change studies facilitate considering clone change information as a metric class in characterizing clone as short-lived and long-lived.

However, different results have been found by Göde et al. [8]. They analyzed clone evolution from the perspective of clone changes frequency and harmfulness of changes [8]. After analyzing three mature software, the authors found that most of the clones were rarely changed and unintentional inconsistent changes to clones were also small. Addressing two questions, they wanted to know how many clones required careful attention during software maintenance time and how many of those unintentionally happened during software development. This study divided changes to clones into three categories namely never changed, changed once and changed more than once. Studying on three subject systems, authors found 47.5% of all genealogies were never changed, 40.3% changed once and less than 8% of all genealogies were changed more than once. The change frequency of clones provides knowledge of which clones should require more attention. These findings lead to consider clone change history as metrics in this research.

Although these studies focused on how clone changes in successive versions, this research investigates clone evolution from the perspective of source code type, that is, main and test code clone, clone location, that is, Inter-File and Intra-File clone, and volatile clones lifetime. Moreover, those studies did not consider clone lifetime characterization.

3.4 Clone Lifetime Analysis

Clone lifetime analysis research focuses on clone aliveness (originating and removing duration) in the system [7, 6]. Researchers tries to identify reasons, why clones have been removed immediately or after many versions from its introduction as a clone in the system. The characteristics of a clone, surrounding context, maintenance activities etc. which may have influence for such aliveness. This kind of research facilitates developers by suggesting clones for maintenance.

A few researches have been conducted on clone lifetime analysis. Cai et al., studied long-lived clones to selectively identify clones for refactoring [7]. They used statistical binning mechanism to categorize clone lifetime based on the number of days a clone class lived in the repository. Afterward, different spatial and historical metrics were used to classify clones into their lifetime. They found that evolutionary information like addition, deletion, same, shift, consistent and inconsistent change of clone siblings in a clone class are important metrics to characterize long-lived clones whereas spatial metrics like the size, line of code, the number of clone siblings etc. are not important characteristics for such clone lifetime characterization.

Thongtanunam et. al., conducted research on clone life expectancy by characterizing whether a clone will be short-lived or long-lived [6]. They used k-means clustering to label volatile clones as short or long-lived clones. After that, using several metrics such as the number of clone siblings, cyclomatic complexity, FanIn, FanOut etc. they formulated the problem as a supervised machine learning problem and performed classification by using a random forest classifier [17]. They achieved 0.63 to 0.92 AUC (Area Under the Curve) scores to classify clones as short-lived clones. Although their study used various metrics to characterize clone life expectancy, they did not consider interface similarities, source code type, the contextual coupling of clone etc.

These research are helpful to know the importance of analyzing and characterizing lifetime of clones. This helps to identify beneficial clones which should be refactored and less beneficial clones which can be avoided to improve the efficiency of clone management efforts.

3.5 Summary

In the literature, many research exist on clone evolution and management. These researches focused on the presences of clone in different systems up to the management of those clones. Many studies tried to validate different evolutionary patterns, changes of clones, and many identify characteristics of clone for such patterns. The summary of related work such as clone occurrences studies in large scale dataset, clone evolutionary studies, changes of clone and clone lifetime analysis are discussed in this chapter. However, these research neither analyze clone evolution from source code type, clone type, clone location nor do they characterize clone lifetime using contextual information, interface similarities and co-change history. The following chapter discusses clone evolution by analyzing clone lifetime.

Chapter 4

An Empirical Study on Clone Evolution

According to Fowler et al., Code clone is the most prevalent code smells in the source code [1]. It helps developers to implement functionalities easily and quickly by re-using existing similar ones. With the evolution of source code, cloned codes start to create maintenance problems like bug duplication and propagation, inconsistency among cloned codes, etc. It has several negative impacts on software maintenance, so researchers conducted many research to better understand clone evolutionary implications. Most of those research focused on the ratio of cloned and non-cloned codes in the source code [46, 47], clone change consistency, that is, consistent and inconsistent changes among clones [39, 40], and cloning patterns [32, 36], etc. On the contrary, a small amount of research focused on clone genealogies evolution [5, 7] and clone lifetime [6]. Neither those studies consider clone evolution from source code type and clone location nor clone volatility from clone type considering their lifetime. Analyzing clone evolution from these perspectives and obtained observations can be helpful to support clone maintenance activities like refactoring, clone life expectancy, etc. Moreover, one of the goals of empirical research in software engineering is to establish facts about software from experimental data using statistical methods [30]. In this research, an empirical study has been performed on clone evolution by considering source code types, clone location, as well as clone volatility and their lifetime. In the study, it has been found that

- Test code clones exist more and evolve increasingly than main code clone, which infers that increase in the number of clone classes is not always harmful as test code clone less likely to be maintained.
- *Intra-File* clones occurred in a repository more than *Inter-File* clones, which infers that developers tend to clone in the same file than different files.

- *Type-3* (gaped) clones are more volatile than Type-1 (exact similar) and Type-2 (parameterized) clones.

- *Intra-File* clones are more volatile than *Inter-File* clones in clone genealogies. Most of these clones have lived for one version only.

4.1 Introduction

Code cloning is the process of duplicating source code with or without modification [10]. It is the common practice in software development for faster implementation without using proper design. As developers have a strict schedule and less time to implement a properly designed functionality, they rely on existing similar implementations without considering future maintenance implications.

Duplicating a single code fragment multiple times creates a clone class which may require consistent updates across all clone fragments within this clone class. It can also be the reason for bug propagation [48], late propagation [44] unintended clone inconsistencies [49, 50], and risk inducer by increasing change frequency [51, 8], etc. So that, managing such clones require more efforts than non-cloned code [3]. Besides these negative impacts, it has some positive impacts too. Research has found that all clones are not harmful, as cloning sometimes fulfill particular, for example, templating design issues [32]. Therefore, understanding clone by observing evolutionary behavior is beneficial from the clone management perspective.

Clone evolutionary studies observe how clones evolve throughout the version history, for example, the number of clone increases or decreases in subsequent versions, the number of bugs in the cloned code and non-cloned code, etc. With the development of software, the number of clones may also increase. However, it is important to know whether this clone increases in main code or in test code. Because the test code clone is not as important as the main code clone from clone maintenance activities. An increase in main code clone depicts developers are not concern about managing production code clones whereas test code clones are less likely to be changed in case of future change and feature improvement.

As clones evolve, some clones disappeared after some versions due to multiple updates within clone fragments. These clones are termed as volatile clones and the number of versions a clone alive in the version history is denoted as clone lifetime [5]. Research has found that some clones exist in the repository for a short number of versions and some may alive for multiple versions.

The volatility of clone may depend on clone type, clone location, etc. Studying these volatile clones can help to better characterize why clones have been going through such evolution.

To observe how clones evolve, this study focuses on clone evolution in terms of clone occurrences in different (a) source code types namely main code clone, and test code clone, (b) clone location namely Intra-File clone and Inter-File clone, and (c) clone volatility based on clone type and their (d) lifetime. Formally, this research aims to investigate the following four research questions.

1. RQ4.1: How do clones evolve in terms of the number of clones with respect to main or test code throughout the version history? — Investigating clone evolution in subsequent versions provide knowledge about how clone changes in the version history. It can be assumed that with the increases of the source code, the number of clone code also increases. However, this question aims to investigate which type of source code clone (main and test code clone) is responsible for such an increase.

2. RQ4.2: Is there any difference between Intra-File and Inter-File clones in terms of occurrences? — It can be assumed that developers like to clone existing functionalities from a different module or file to another module or file due to lack of code ownership, and not having much time to understand other module functionalities [4]. So it is a common perception that clone should occur more in Inter-File than Intra-File. Here, clone occurrences are investigated depending on file level location of clone to validate this assumption.

3. RQ4.3: Which type of clone tends to be more volatile in the clone genealogies? — Evolving clones may be removed by refactoring or disappeared by subsequent changes incorporated into some of the clone instances within a clone class. Knowing which type of clone (from Type-1, Type-2 and Type-3) is more volatile can help to decide which type of clones require more attention during refactoring activities.

4. RQ4.4: How does clone location impact volatile clone lifetime in the clone genealogies? — As stated earlier some clones may have resided in the repository for many versions and some may be removed after a while. Therefore, clone lifetime can vary depending on the clone location. Knowing clone lifetime based on inter-file and intra-file clone may help to characterize life expectancy of a clone. Here, this research investigates whether clone lifetime depends on clone location or not.

To answer these questions, an empirical study has been performed on five open source software namely Maven[1], Flink[2], Pulsar[3], Spring-boot[4] and Pig[5]. During the analysis, release versions of each software are used to detect clones and observe their evolution. Firstly, the Github repository is traversed to extract only official releases and create experimented versions. Then, code clone is detected in every version by using iclones – incremental clone detector [22]. From different clone granularities, for example, file level, method level and block level clones, this study uses block level clones because block level clones occur most in repositories. It also provides the version wise clone evolution mapping termed as clone genealogies by using change information of subsequent versions. From the clone result, source code wise and location wise clones are retrieved for every version. Lastly, volatile clones which do not have evolution mapping from its origin to the last version are extracted from all clones. Along with volatile clones, the lifetime of each volatile clone class is also calculated by counting the number of versions a clone class lived in the version history. Using these data above mentioned research questions are answered.

The following sections of this chapter are organized as follows. Section 4.2 describes the whole approach, Section 4.3 presents empirical study, observations and answers of research questions, Section 4.4 mentioned the threats to validity in the empirical study, and finally, Section 4.5 summaries the research.

4.2 Approach

This study analyzes clone evolution to answer the research questions mentioned in Section 4.1. The overview of the total approach is shown in Figure 4.1. It consists of three steps, which are *(i)* Version Creation, *(ii)* Clone Genealogies Extraction and *(iii)* Clone Evolution Analysis. For each of the experimented software, these steps are applied. In Version Creation step, the Git repository of experimented software is traversed to select only official release tags to identify versions. In the second step, clones are detected in every versions and clone genealogies are extracted from the versions of the target software. From these volatile clones, the lifetime of each volatile clone will be calculated. Lastly, clone evolution is analyzed by the number of

[1]https://github.com/apache/maven
[2]https://github.com/apache/flink
[3]https://github.com/apache/pulsar
[4]https://github.com/spring-projects/spring-boot
[5]https://github.com/apache/pig

clone classes with respect to (a) source code type, which are, main code or test code, (b) clone occurrences with respect to clone location, which are, Intra-File clones and Inter-File clones, (c) clone volatility with respect to clone type and (d) the lifetime of volatile clones. Each of these steps is described below.

4.2.1 Version Creation

To analyze clone evolution, this study investigates code clone in each release of the software. In this step, official releases are selected and versions are created. As a target of the investigation, release level versions are selected because revision level versions may have experimental development. Releases also represent more stable products than revision level products. From repositories of the software, all branches except the main branch are removed to echo the original clone evolution (normally *master* branch of a git repository). This is required as developers often create different branches to develop different features which may have been merged into the main branch after a while.

To collect software, popular git based Open Source Software (OSS) container Github is used. From a Git repository, all releases are identified by using *git tag* command. These tags contain information of official releases along with release candidates. To get these release tags sequentially with the creation time, each Github repository is traversed using *log*[6] command. Then, using the regular expression, all release candidates with keywords like 'RC', 'alpha', 'beta' etc. are removed. Remaining tags are considered as official releases and selected version tags for the analysis. Finally, versions are created by using *reset*[7] command with the previously selected tag names. These versions are used in the next step to identify clones in every version along with their clone genealogies.

4.2.2 Clone Genealogies Extraction

From the previously created versions, code clone is detected for every version and each clone fragment is mapped from their origin version up to the alive version successively. To detect clone, an incremental clone detection tool namely iClones [22] is used. iClones performs well to detect all three types of clone [10] and which was previously used in clone evolution-related research [41, 6]. It uses a generalized suffix tree and Baker's *pdup* algorithm [52] to detect clones

[6]git log –pretty='%ci,%H'

[7]git reset –hard $< sha >$

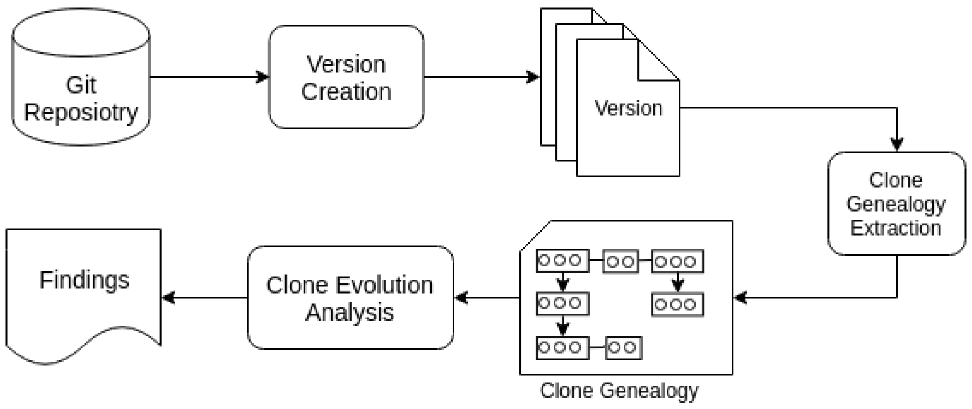


Figure 4.1: Overview of Clone Evolution Analysis

from each version. In this step clone genealogy for every clone fragment is also identified. It uses the change information of file like modified, added, and deleted to map clone genealogies from one version to another. The suffix tree is updated using the change information of consecutive versions. From the updated tree, *pdup* is used to report only new tagged clones.

To extract clones along with their genealogies from selected versions, firstly, all three types of clones (Type 1, Type 2, and Type 3) are detected by using 'bazrafshan' mapping which allows detecting Type-3 clones by merging Type-1 and Type-2 clones [53]. Clones are detected at minimum token size as 100 and near-miss clone merge for Type-3 clone as 20 tokens. Late propagation is allowed to get consistent and re-synchronized [54] clone genealogies. Remaining options are set as default to get more accurate clones [55].

4.2.3 Clone Evolution Analysis

Clone evolution is analyzed with respect to, *(a)* source code type, *(b)* clone location, *(c)* the volatility of clone based on clone types, and *(d)* the lifetime of volatile clones. Analyzing clone evolution regarding these perspectives will help developers for maintenance tasks like predicting clone life expectancy, selecting refactoring candidate clone and understanding evolutionary facts about clone, etc.

From the previously generated clone genealogies which provides a Rich Clone Format (RCF) [56], clones are divided into main code clones and test code clones based on their source code type. If a clone class consists of only test code clone siblings, it is referred to as a test code clone. Otherwise, clones are labeled as the main code clone. These clones will be used to answer RQ1. Furthermore, all clones are divided into Intra-File (clone classes having clone siblings located in only one file), and Inter-File clones (clone classes having clone siblings spanning different files) based on the clone locations of all cloned fragments within a clone class. These clones will be used to answer RQ2.

To collect volatile clones, all clones which do not have a successor mapped clone in subsequent versions are selected. If clones are originated in the last version or mapped to up to the last version from the origin then those are excluded from selected volatile clones. This study excluded these clones because such clones may evolve further versions with software development which is out of the research scope. These volatile clones labeled with their lifetime (the number of versions each clone exists in the version).

Volatile clones are categorized according to their clone type that is Type-1 (exact similar clone), Type-2 (parameterized clone) and Type-3 (gaped clone). This data will be used to answer the RQ3. Using statistical measurements like percentage, more volatile prone clone type is identified. Besides this, volatile clones with their lifetime, clones are also divided into Intra-File and Inter-File volatile clones. This categorization is created based on the location of the volatile clone siblings. These will be adopted to answer RQ4. Observing the lifetime of these Inter and Intra-File volatile clones provide insight to which types of clones are more volatile in the repository and how these clones evolve throughout the clone genealogies.

4.3 Empirical Study

To know the evolutionary characteristics of clones with respect to source code type, clone location and clone lifetime, an empirical study has been performed on five Open Source Software (OSS) namely *Maven, Flink, Spring-boot, Pig* and *Pulsar*. These software are selected based on the number of commits and releases to cover variation in experimented software. The details of the studied software are given in Table 4.1. The study is organized with research questions, experimentation results and observations.

Table 4.1: Overview of the Studied Software

Software	**Study Period**	**#Releases**	**#Commits**
Maven	2003-09-01 to 2018-06-17	54	10,420
Flink	2010-12-15 to 2018-10-17	73	15,002
Spring-boot	2014-03-21 to 2018-10-16	112	19,330
Pulsar	2016-09-06 to 2018-10-12	59	2,447
Pig	2009-03-05 to 2016-06-07	57	2,857

4.3.1 Studied Software

Open source software is a great source of software repositories to conduct empirical studies. In this study, following well known Github repositories are studied to answer above mentioned research questions.

Maven Apache Maven is a software project management and comprehension tool, popularly used to build java projects[1]. It is based on the concept of a Project Object Model (POM). It can manage a project's build, reporting and documentation by using pom.xml. It helps to share JAR files across the project. From Table 4.1, it can be seen that, Maven has 54 releases with more than 10 thousand commit history within 15 years.

Flink Flink is an open source stream processing framework which provides powerful stream and batch processing capabilities[2]. It supports very high throughput and low event latency with the fault-tolerance mechanism. It has elegant and fluent APIs in Java and Scala. Moreover, it provides compatibility layers for Hadoop MapReduce and integration with YARN, HDFS, HBase, and other components of the Apache Hadoop ecosystem. From Table 4.1, it can be seen that, within 8 years, Flink has 73 releases with more than 15 thousand commit history.

Spring-boot Spring Boot is a framework to create stand-alone, production-grade Spring based Applications with all project build functionalities[4]. It simplifies build configuration by removing code generation and XML configuration. It also provides non-functional features like embedded servers, security, metrics, health checks, etc to observe the performance. From Table 4.1, it can be seen that, Spring-boot has 112 releases with more than 19 thousand commit history within 4 years.

Pulsar Pulsar is an open source distributed server to server messaging platform[3]. It supports multiple clusters and provides seamless geo-replication of messages across clusters. It also provides client API with bindings for Java, Go, Python and C++. It is originally created by Yahoo and now-a-days maintained by Apache Foundation. From Table 4.1, it can be seen that, Pulsar has 59 releases with more than 2 thousand commit history within 2 years.

Pig Pig is a dataflow programming environment for processing very large complex data[5]. It transforms data using relational algebra style operations such as join, filter, project and functional programming style operators such as MapReduce etc. It is commonly used in Hadoop MapReduce program. From Table 4.1, it can be seen that, Pig has 57 releases with more than 2 thousand and 8 hundred commit history within 7 years.

Table 4.2: Clone Summary in the Studied Software

Software	**#Selected Releases**	**Clone Classes**					
		Type 1		**Type 2**		**Type 3**	
		Min	Max	Min	Max	Min	Max
Spring-boot	66	7	96	8	82	14	101
Maven	37	28	56	67	77	83	97
Flink	15	182	1657	141	1112	309	4902
Pulsar	12	116	566	125	398	447	1328
Pig	15	276	988	421	942	446	1627

Using the above described approach in Section 4.2, 12 versions of Pulsar, 15 versions of Pig and Pulsar, 37 versions of Maven and 66 versions of Spring-boot are created. Table 4.2 represents the number of versions, the minimum and the maximum number of clones with three types of clones in those software. From the table, it can be seen that Spring-boot and Maven have a lower number of clones than other software. Flink software has the maximum clone for all three types of clone.

4.3.2 Experimented Result

In the following, research questions are answered using experimented data.

Research Question 4.1: How do clones evolve in terms of the number of clones with respect to main or test code throughout the version history?

The aim of this question was to investigate whether the number of clone classes follows any pattern and whether the number of clone classes deviates for different source codes in clone evolution. One may assume that with the increase of source code, clone codes also increase. As an example, Figure 4.2 depicts the number of clone classes evolves in the successive versions at Flink. From Figure 4.2, it can be found that the number of clone classes follows an increasing pattern. However, the pattern is not smoothly increasing in successive versions as some clone maintenance activity may have been performed with the development of software. Here, the investigation proceeds on how this increasing pattern evolves for main code clone and test code clone throughout the versions.

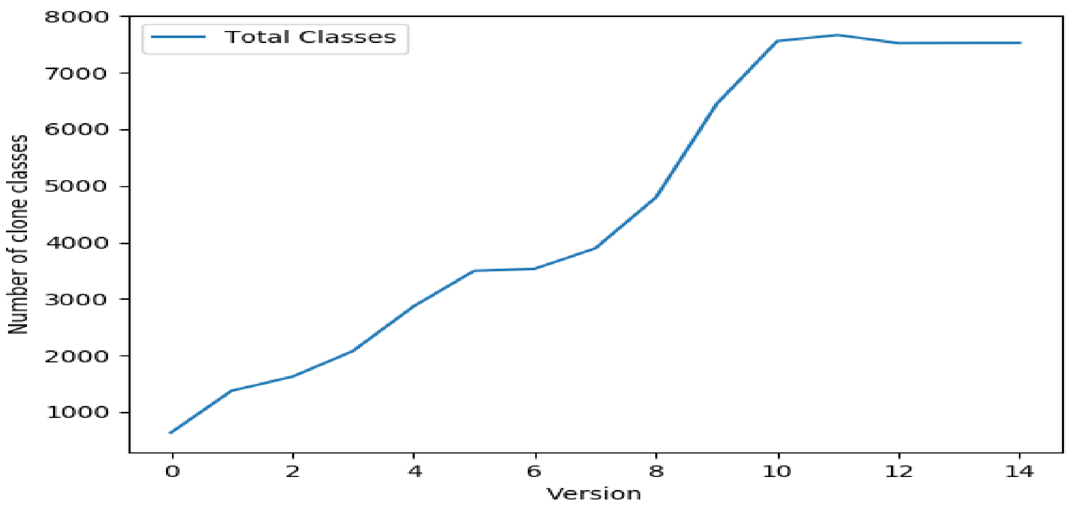
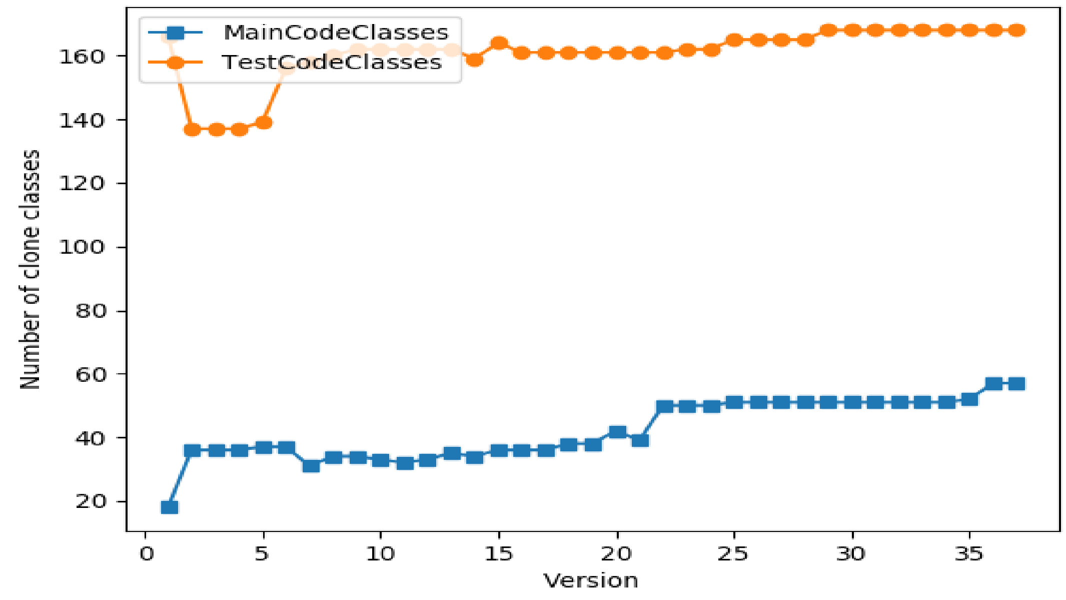
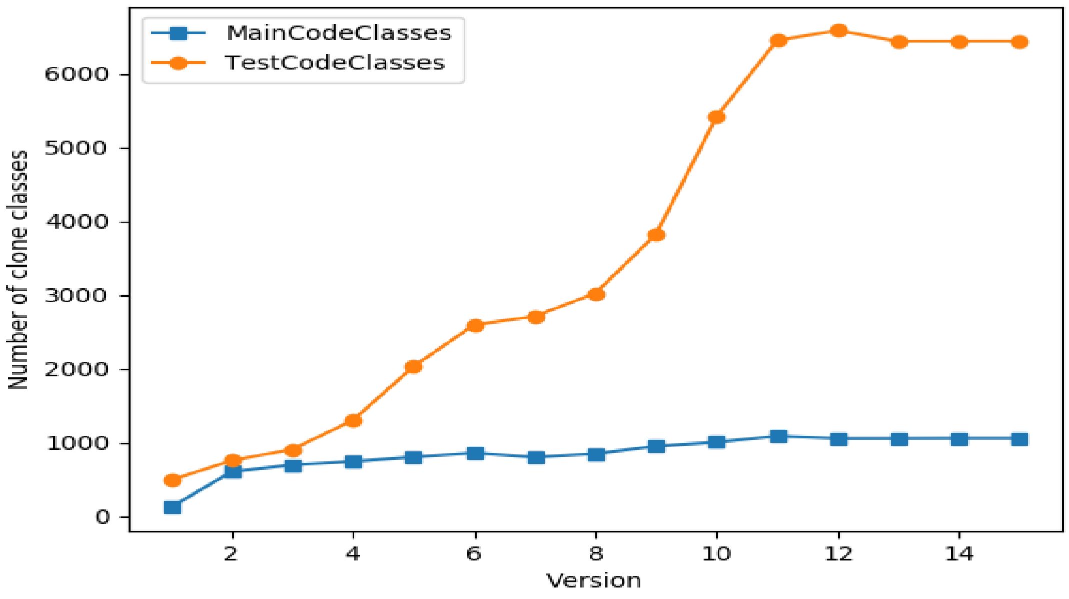


Figure 4.2: Clone Evolution at Flink

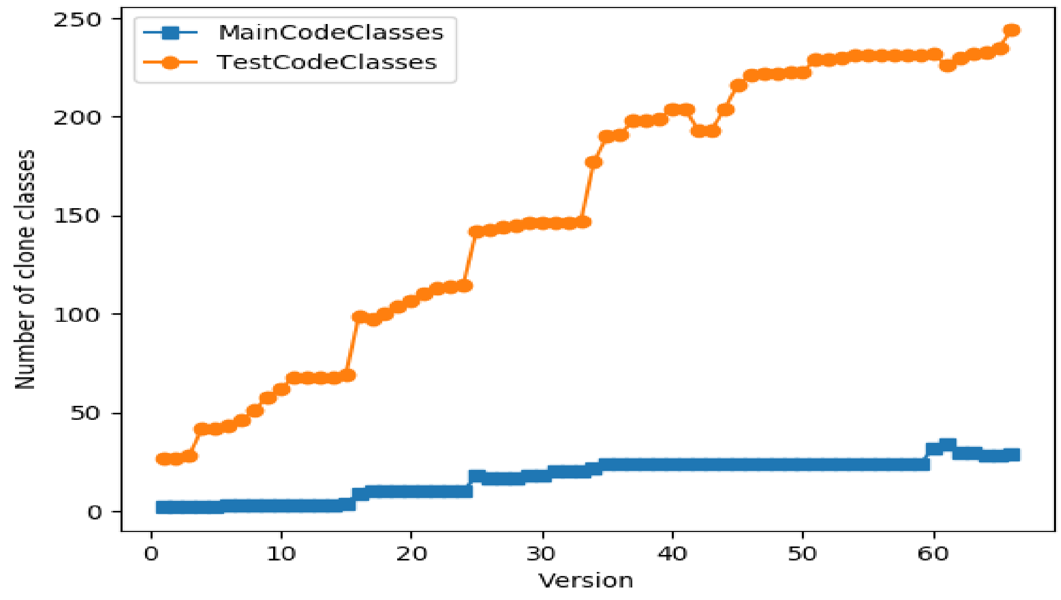
The assumption was that the clone exists in both types of source code and increases similarly in successive versions. That is why the number of clone increases with the code base. Figure 4.3a–4.3e show the code clone evolution of main code clones and test code clones at each analyzed system. In the figure, X-axis represents the consecutive versions and Y-axis represents the number of clone classes for each system. The square marked line (blue line) represents clone evolution main code clones and the circle marked line (orange line) represents clone evolution of test code clones. From the figure, it can be seen that test code clones exist more in a code base than the main code clone at four software, which are, Maven, Flink, Spring-boot and pig. Meanwhile, Pulsar software initially has a lower number of test code clones than main code clones. However, all analyzed software has a higher increasing rate for test code clones than main code clones. This has also happened in Pulsar software evolution (Figure 4.3d). In the figure, it can be seen that the number of test code clones increases along with main code clones and exceeds after 8th version which is v1.21.0-incubating.



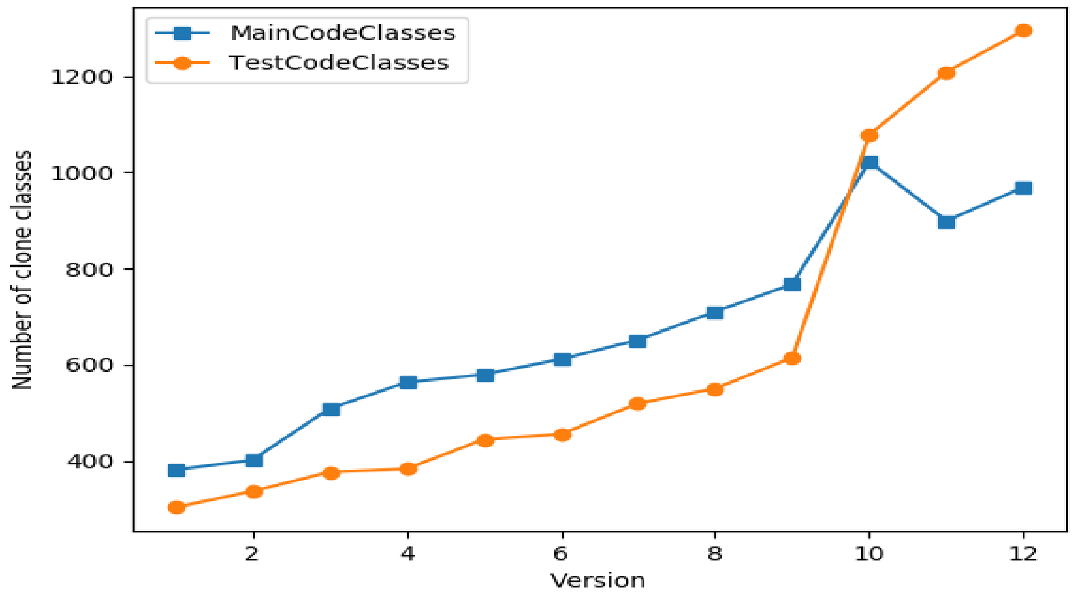
(a) Clone Evolution of Main Code and Test Code at Maven



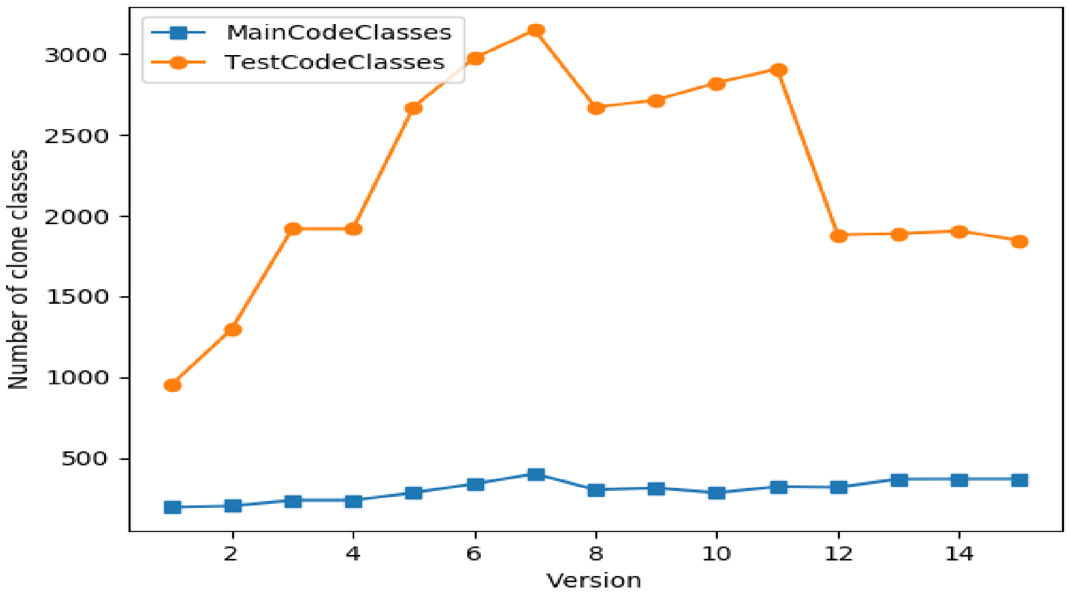
(b) Clone Evolution of Main Code and Test Code at Flink



(c) Clone Evolution of Main Code and Test Code at Spring-boot



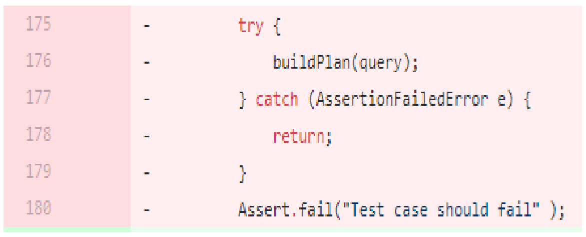
(d) Clone Evolution of Main Code and Test Code at Pulsar



(e) Clone Evolution of Main Code and Test Code at Pig

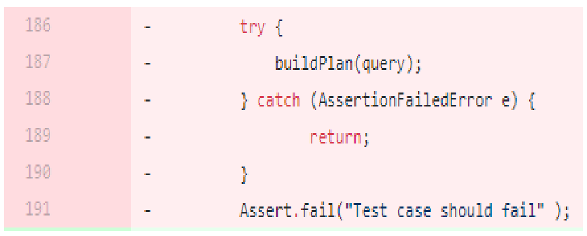
Figure 4.3: Evolution of Clone Classes with respect to Main and Test Code

Pig software shows different pattern than other software in Figure 4.3e for extensive maintenance happened on test code clones. Here, the number of test code clones decreases after Version 6. To know the reason behind such deviation, some commits were manually analyzed. These commits have large number of addition and deletion of codes. it has been found that a lot of maintenance activities, like refactoring, have been performed at those commits, for example, *sha1*[8], *sha2*[9]. As an evidence of maintenance activitiy, Figure 4.5 shows test code clone removal. The commit message of *sha1* also confirms the refactoring by stating *PIG-3087: Refactor TestLogicalPlanBuilder to be meaningful.*



```
-           try {
                buildPlan(query);
-           } catch (AssertionFailedError e) {
-               return;
-           }
-           Assert.fail("Test case should fail" );
```

(a) Removed Test Code Clone Fragment 1



```
-           try {
-               buildPlan(query);
-           } catch (AssertionFailedError e) {
-                   return;
-           }
-           Assert.fail("Test case should fail" );
```

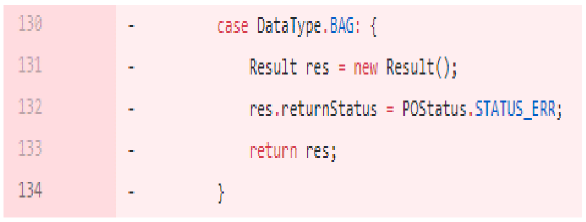
(b) Removed Test Code Clone Fragment 2

Figure 4.5: Test Code Clone Removal

Similarly, Figure 4.7, presents an clone refactoring on *sha2* where clones are refactored

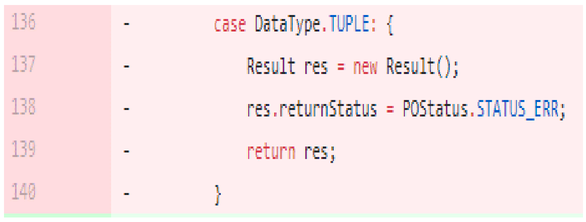
[8]https://github.com/apache/pig/commit/b47249340f8410172ec2c7db5c12f762ad92d36d
[9]https://github.com/apache/pig/commit/a760df0b20425eef2820b2526baa617c81358ce4

by using the Extract Method refactoring technique. There common clone fragment has been removed to another method named as *error()*. The *error()* method is invoked from the original snippets where the cloned common code resided.



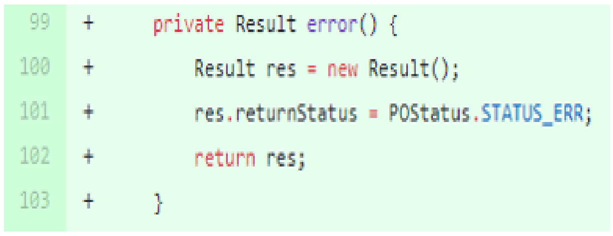
```
-         case DataType.BAG: {
-             Result res = new Result();
-             res.returnStatus = POStatus.STATUS_ERR;
-             return res;
-         }
```

(a) Clone Fragment 1 (before refactoring)



```
-         case DataType.TUPLE: {
-             Result res = new Result();
-             res.returnStatus = POStatus.STATUS_ERR;
-             return res;
-         }
```

(b) Clone Fragment 2 (before refactoring)



```
+     private Result error() {
+         Result res = new Result();
+         res.returnStatus = POStatus.STATUS_ERR;
+         return res;
+     }
```

(a) Extracted Method Fragment (for refactoring)

Figure 4.7: Code Clone Refactoring using Extract Method Refactoring

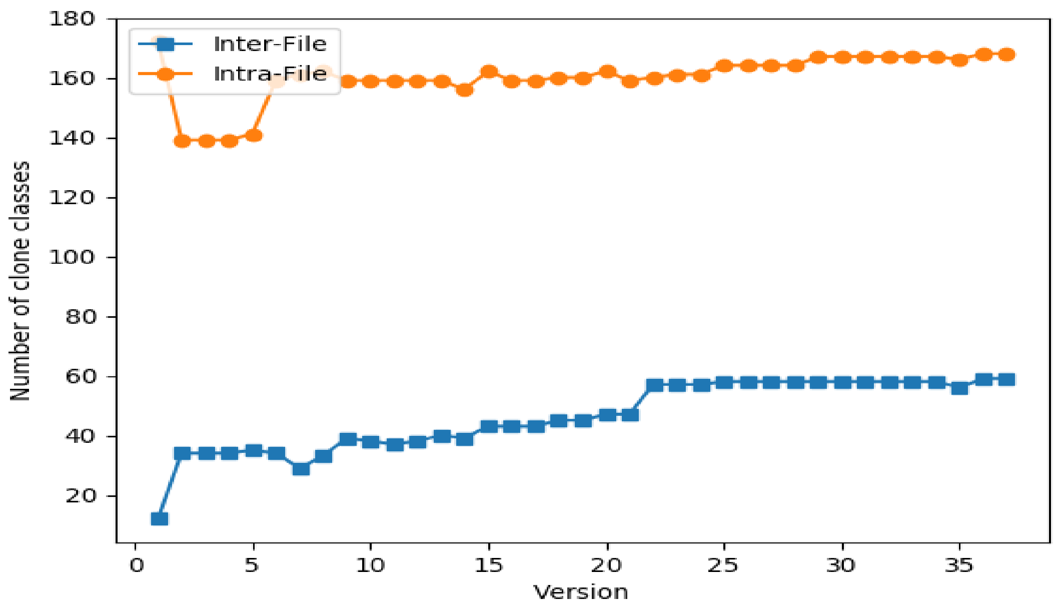
Observation 1: The number of clone classes follows an increasing pattern with the development of the software (as source code increases with the version). However, test code clones contribute the most to this increasing number of clones.

Research Question 4.2: Is there any difference between Intra-File and Inter-File clones in terms of occurrences?

This research question aims to investigate whether clones at a single file (Intra-File) occur more than multiple files (Inter-File) or vice versa. This helps to figure out which clones should get more attention during clone maintenance activity like refactoring which depends on clone location [1]. For example, Intra-File clones can easily be removed by applying an extract method refactoring whereas Inter-File clones require high level refactoring such as Extract Super Class, Extract Utility Class, Pull Up Method, and Form Template Method, etc. which are difficult to carry out for more cloned fragments.

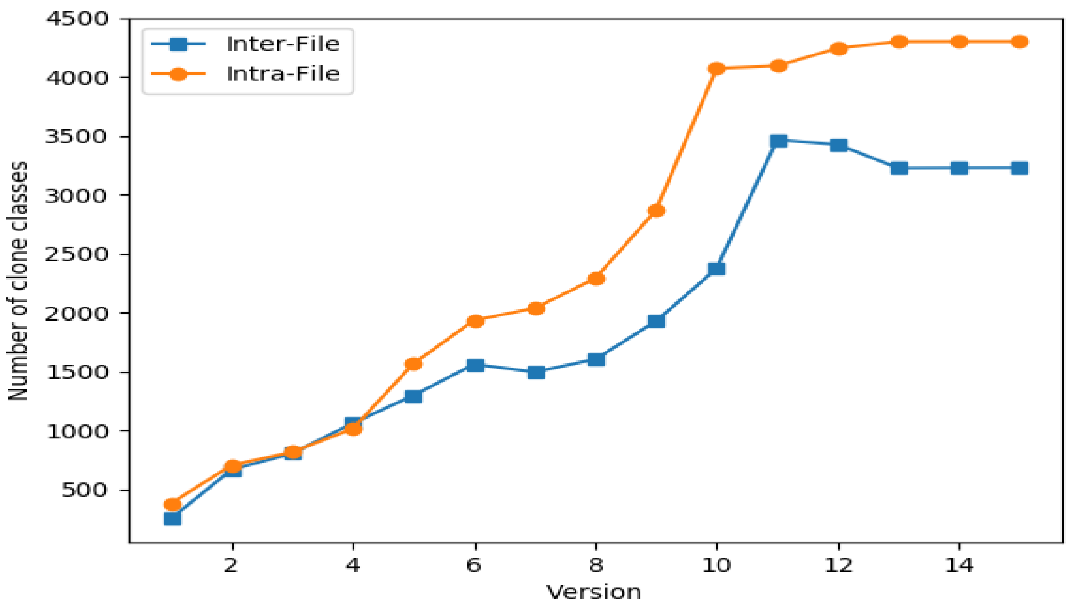
Table 4.3: The Min, Max Difference of Inter-File and Intra-File Clone Classes

Software	Min	Max
spring-boot	22	109
maven	103	160
flink	168	1013
pulsar	438	1175
pig	13	1693

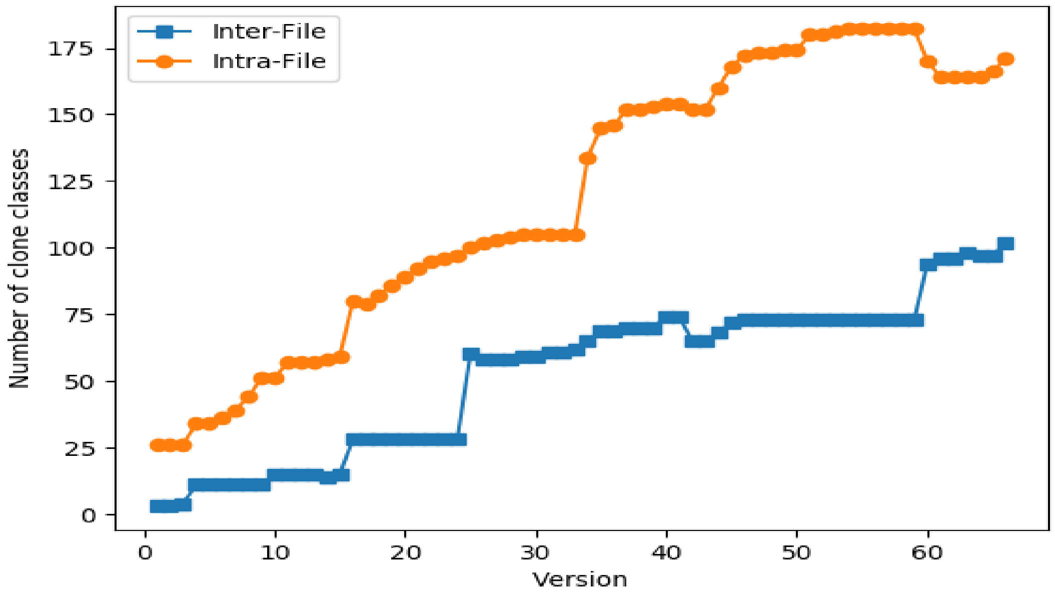


(a) Clone Evolution of Inter-file and Intra-File Clones at Maven

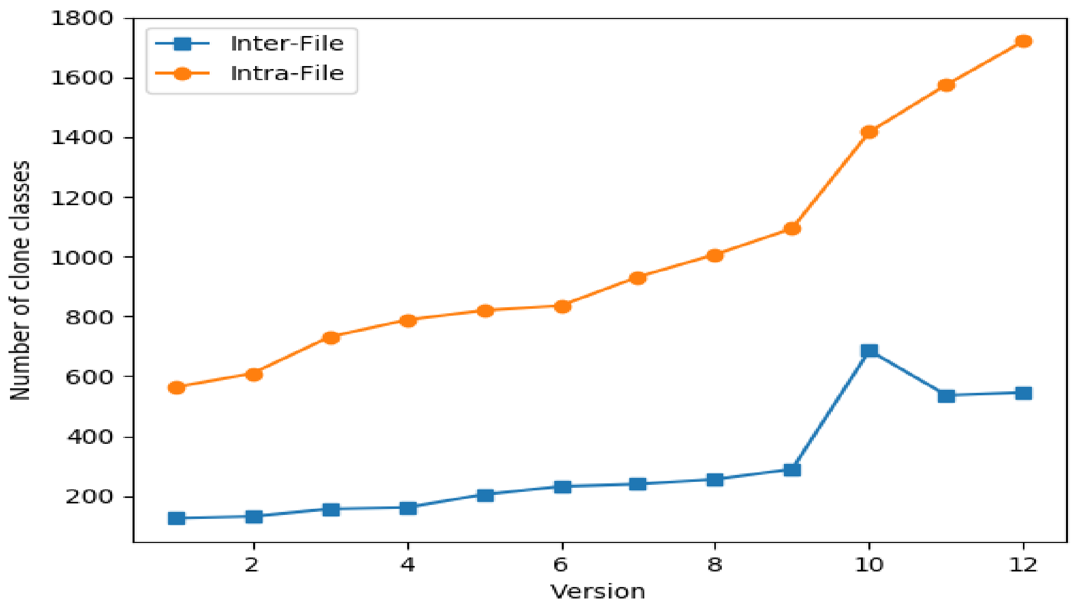
To answer this question, the differences between Inter-File clones and Intra-File clones in the number of clone classes are calculated for every release versions of analyzed software. For every software, it has been found that the number of Intra-File clones exists more than Inter-File clones. Table 4.3 shows the minimum and maximum differences in the number of clone classes between Inter-File and Intra-File clones at those analyzed software. From the table, it can be seen that, Intra-File and Inter-File clones deviate at least 22 to 438 and at maximum 160 to 1175 number of clone classes. Figure 4.8a–4.8e represent Inter-File and Intra-File clone evolution where X-axis represents the subsequent number of versions and Y-axis represents the number of clone classes. From Figure 4.8, it can be seen that the number of Intra-File clones increases more than Inter-File clones. The reason behind Intra-File clones outnumbering Inter-



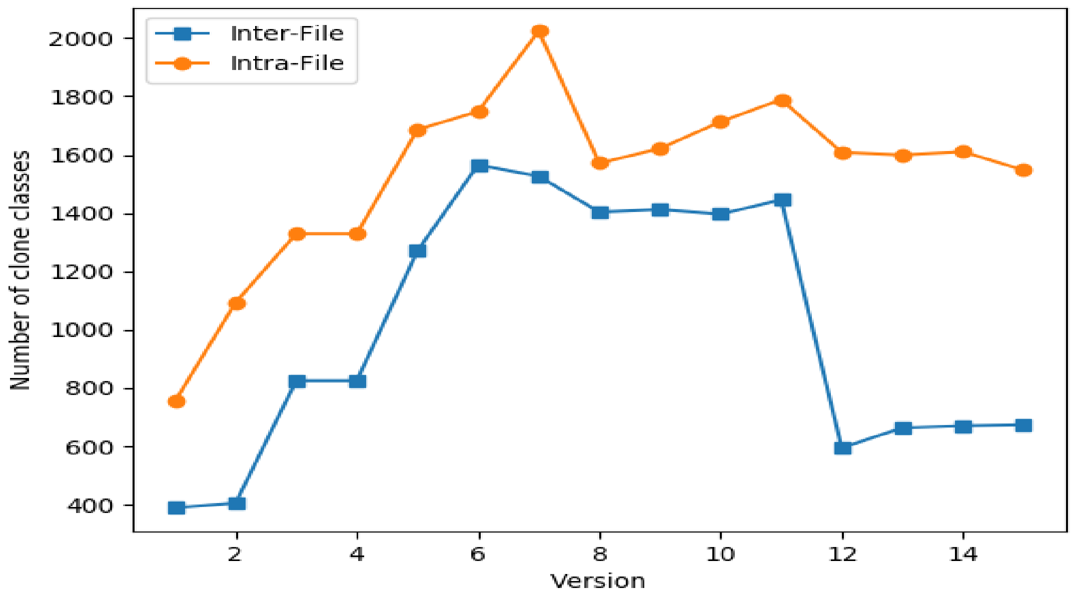
(b) Clone Evolution of Inter-file and Intra-File Clones at Flink



(c) Clone Evolution of Inter-file and Intra-File Clones at Spring-boot



(d) Clone Evolution of Inter-file and Intra-File Clones at Pulsar



(e) Clone Evolution of Inter-file and Intra-File Clones at Pig

Figure 4.8: Inter-File and Intra-File Clone Classes Evolution

File clones can also be answered from the observation of RQ1. As the analyzed system contains more test code clones than main code clones, clones exist more in the same file.

Observation 2: In location-based clone evolution, *Intra-File* clones occurred more in a repository than *Inter-File* clones. So, it can be said that having the opportunity to abstract the same code fragment and calling it from multiple fragments, developers still tend to clone in the same file without following the proper design.

Research Question 4.3: Which type of clone tends to be more volatile in the clone genealogies?

The aim of this question was to investigate clone type wise volatility measurement. As a clone may be removed by successive changes into the cloned fragment without changing others of a clone class or removed by applying extract method refactoring (merging the similar code fragment into a common code fragment).

Table 4.4: Type Wise Volatile Clone Class Percentages in the Experimented Software

Software	Total Volatile Clone Classes	Type 1	Type 2	Type 3
Maven	377	20.20%	33.70%	46.10%
Pig	4213	28.40%	23.40%	48.20%
Spring-boot	488	36.70%	24.80%	38.50%
Flink	8237	22.80%	14.10%	63.10%
Pulsar	1782	15.90%	20.50%	63.60%

To know more volatile clone type, all three types (Type-1, Type-2, and Type-3) volatile clone percentages are calculated. Table 4.4 presents percentages of all volatile clones with the respective software. From the table, it can be seen that, for all software, Type-3 clones are more volatile than Type-2 and Type-1 clones. The reason behind Type-3 clones are more prone to volatile could be, clones may face repetitive changes and loss clone relation with its similar code fragment. After that, Type-1 clone is more volatile than Type-2 clones for Pig, Flink and Spring-boot software. One possible reason for such less volatility rate to Type-2 clone could be that during the creation of Type-2 clones, clones are adapted with its surrounding context. So that, the change of refactoring is less likely from Type-1 and Type-3 clones. However, Type-2 clones are more volatile than Type-1 clone for Maven and Pulsar. In these software, Type-2 clones may also go through refactoring activities or multiple updates within the code fragment.

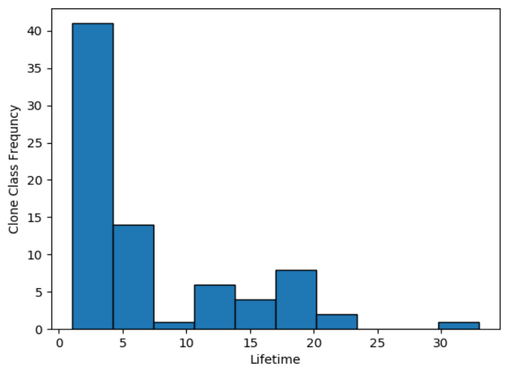
Observation 3: Type-3 clones tend to be more volatile than other types of clones (Type-1 and Type-2) in the repository.

Research Question 4.4: How does clone location impact volatile clone lifetime in the clone genealogies?

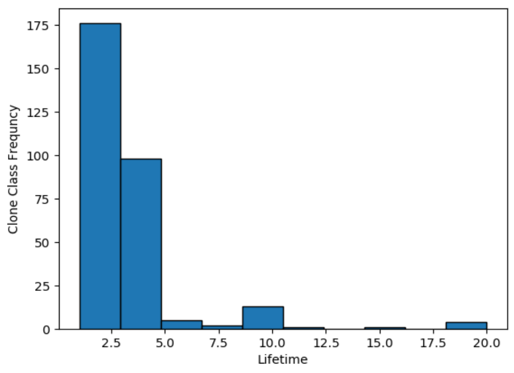
Table 4.5: Genealogies of Volatile Clone Classes

Software	**# Versions**	**# Volatile Clone Classes**	**Genealogies Lifetime**		
			Min	Median	Max
Maven	32	120	1	5	25
Flink	15	7532	1	1	13
Spring-boot	66	241	1	17	65
Pulsar	12	1036	1	1	11
Pig	15	4213	1	5	14

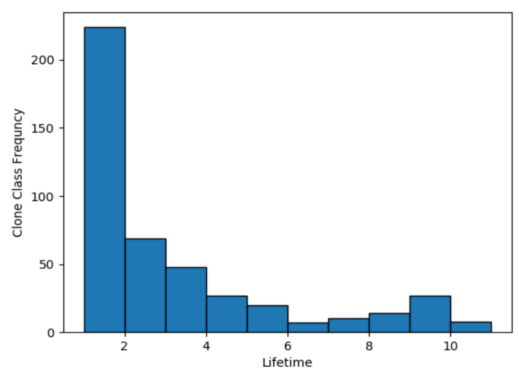
This question aims to investigate whether clone location has any impact on the lifetime of volatile clones. The reason behind analyzing only volatile clones is to understand why clones are removed or changed from the clone genealogies after certain versions. This will be helpful to characterize clone lifetime and predicting clone volatility.



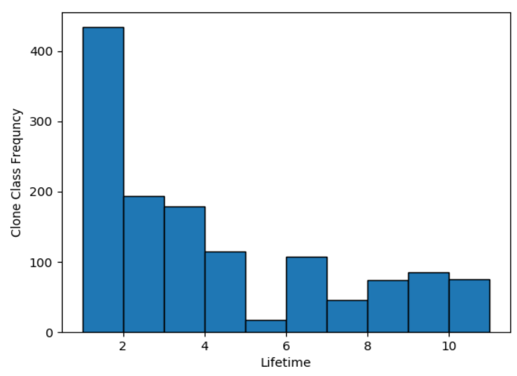
(a) Histogram of the lifetime of the Inter-File volatile clones at Maven



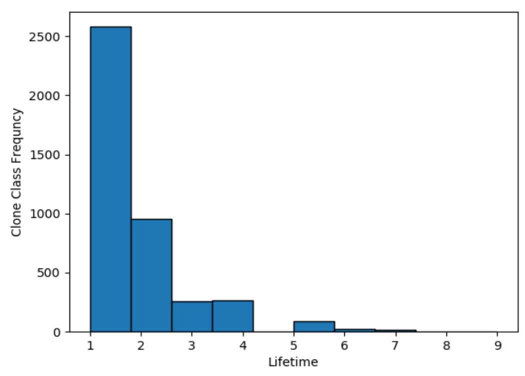
(b) Histogram of the lifetime of the Intra-File volatile clones at Maven



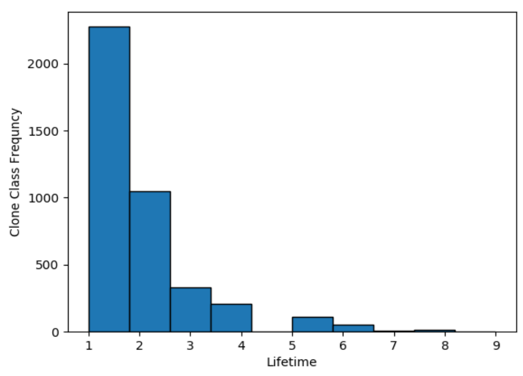
(c) Histogram of the lifetime of the Inter-File volatile clones at Pulsar



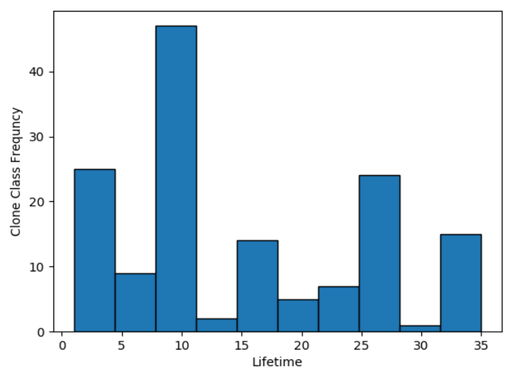
(d) Histogram of the lifetime of the Intra-File volatile clones at Pulsar



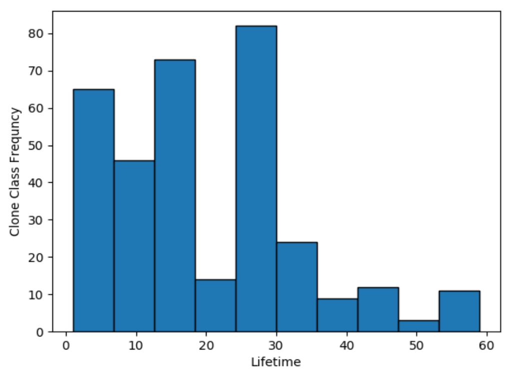
(e) Histogram of the lifetime of the Inter-File volatile clones at Flink



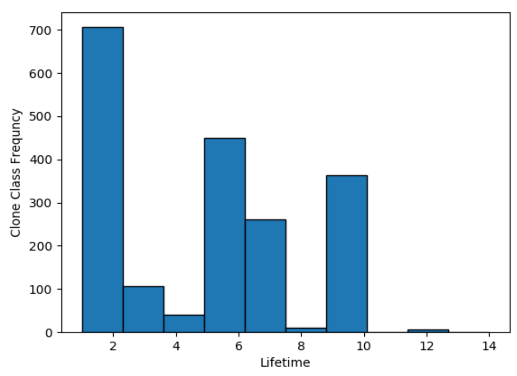
(f) Histogram of the lifetime of the Intra-File volatile clones at Flink



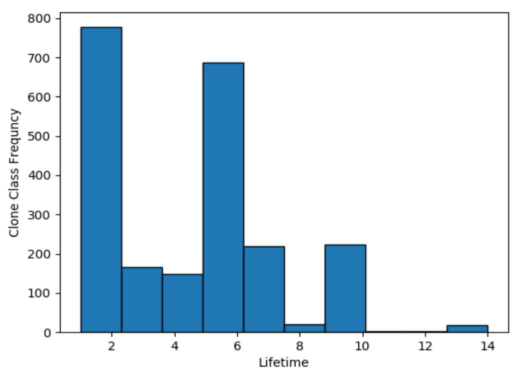
(g) Histogram of the lifetime of the Inter-File volatile clones at Spring-boot



(h) Histogram of the lifetime of the Intra-File volatile clones at Spring-boot



(i) Histogram of the lifetime of the Inter-File volatile clones at Pig



(j) Histogram of the lifetime of the Intra-File volatile clones at Pig

Figure 4.9: Inter-File and Intra-File Volatile Clone Classes Lifetime Frequency

To deeply analyze the lifetime of volatile clones, the number of Inter-File and Intra-File volatile clones are plotted as shown in Figure 4.9. The X-axis of Figure 4.9 represents the lifetime (the number of versions a clone class lived in the clone genealogies) and Y-axis represents the frequency of volatile clone classes. Most of the volatile clones lived one version. However, this lifetime varies depending on the software. As shown in the figure, software Maven, Pulsar, Flink and Pig have the highest frequency for one version whether Spring-boot has the highest frequency during one to five number of versions for Intra-File volatile clones. In Table 4.5, the min, median and maximum lifetime of such volatile clone genealogies are shown. Figure 4.9 also shows that the frequency of Intra-File volatile clones is higher than Inter-File volatile clones. This depicts that, Intra-File clones lived a shorter period of time.

Observation 4: *Intra-File* clones are more volatile than *Inter-File* clones in clone genealogies and their lifetime has a high frequency for 1 version.

4.4 Threats to Validity

Clone evolution analysis by observing clone lifetime with clone location, clone occurrences with source code types has some internal and external threats to validity which are as follows:

Internal Validity As this study heavily depends on the clone genealogies, clone detection and mapping of clones in subsequent versions are the prime concern of the analysis. The accuracy of clone detection and clone genealogies extraction may provide different results which incur internal threats to validate the research. To mitigate this threat, *iClones* tool is used in clone detection and clone genealogies extraction which is popular and was previously used in other researches [41, 6]. It has good precision and recalls in clone detection. Moreover, it uses an incremental way to map clones between subsequent versions.

External Validity The findings of this research cannot be generalized as the findings may vary depending on the analyzed software. The analysis versions can be releases with or without candidate releases or commit wise versions. Moreover, one can consider the time differences between two versions and analyze lifetime of clones by the time duration. The consideration versions can depict different result. In this study, the number of versions is considered as the lifetime. To mitigate external validity, five software repositories are analyzed with a different number of releases. The variation in the analyzed software, different study periods and number of versions also validates the analysis.

4.5 Summary

This chapter presents an empirical study on clone evolution with respect to *(a)* source code (main or test code), *(b)* clone location, *(c)* clone volatility in the clone types and *(d)* volatile clone lifetime. The study has been performed on five open source software with a different number of versions 12 to 66. By the study, it has been found that the number of clone classes increases in the subsequent version where the test code clone contributes most than the main code clone. In clone location based analysis, it has been observed that Intra-File clones occur more than Inter-File clones in a software repository. From the clone type wise volatility, Type-3 clones are more volatile than Type1 and Type-2 clones. Moreover, categorizing volatile clone based on location, it has been also found that Intra-File clones tend to be more volatile than Inter-File clones and have a high frequency for living a small number of versions. These findings will be used to better characterize clone life expectancy in the next chapter.

Chapter 5

Characterizing Clone Lifetime using Metrics

Code cloning is a commonly used practice in software development. In a software repository, 7% to 23% cloned code can be found and it may vary up to 59% [2]. Research has found that removing all clones by refactoring is neither beneficial nor feasible [5], because some clones may be evolved independently, some may be removed after a while, and some may have dependencies with its surrounding context. To selectively identify important clones for refactoring, Cai et al., conducted an empirical study on clone lifetime to characterize long-lived clones [7]. A clone exists in the repository for long, by evolving with other clones, can be a good candidate for refactoring. On the contrary, Patanamon et al. conducted an empirical study to characterize short-lived clones which require less attention during maintenance [6]. Understanding clones characteristics to be long-lived and short-lived will help in clone maintenance by identifying more beneficial clones early. However, their approach did not consider interface similarity, code type, clone location etc. metrics which may have an impact to better characterize clone lifetime. In this research, some metrics, for example, interface similarity, clone context, co-change history, code type, clone location etc. are proposed to better characterize clone life expectancy. By using these metrics, the Random Forest classifier is used to classify short-lived and long-lived clones. The model achieved 0.80 – 0.92 AUC (Area Under the Curve) score to correctly classify short-lived clones for four software. Number of commits all code fragments changed, method name similarity, number of parameters, clone locations are depicted as influential metrics. Moreover, adding the proposed metrics with existing metrics (introduced by Patanamon et al. [6]) increases classifier performance 2% – 7% for studied software. Practitioners can leverage the proposed metrics to better characterize whether a clone will be short-lived or long-lived in the clone maintenance activities.

5.1 Introduction

Code clone is the result of code fragments duplication in the source code. Replication of existing functionality with or without modification is the reason behind code clone creation. To achieve an immediate solution by re-using existing functionalities, it also breaks the DRY (Don't Repeat Yourself) principle [27]. It can create future maintenance problem by incurring similar updates for a clone class or propagating a bug to multiple code fragments. However, all the clones in the repository are not harmful, because, those may be less frequent to be changed [8, 57].

After creation, clone codes may evolve independently or require a consistent update along with other fragments of the clone class [5]. It can also fulfill a design requirement by decoupling different modules [32]. So that, mitigating clones by refactoring requires prioritization or selection of important clones [7, 41, 42] which are more beneficial, cost-effective and less risk prone for future maintenance. Moreover, refactoring clone requires extra time to understand the context and the effects of refactoring.

Previous research has found that after a clone has been introduced, rather than refactoring it immediately, it is better to understand its evolutionary behavior like lifetime and its changing consistency [5, 6]. Refactoring of a long-lived clone is considered beneficial than a short-lived clone. Cai et al. investigated long-lived clones to suggest clones for refactoring [7]. They found that evolutionary characteristics like who modified clones, addition and deletion of clone to its clone group etc. are important to characterize long-lived clones rather than Line Of Code (LOC), the number of clone instances in the same group etc.

Patanamon et al., found that 30% to 87% of clones lived for a short duration and a smaller proportion of short-lived clones (9% to 19%) change consistently. Whereas the long-lived clones consistently changed with their siblings 25% to 37% times [6]. They characterize clone life expectancy by classifying short-lived and long-lived clones using different metrics like complexity, quality, human factors etc. However, they did not consider some metrics like co-change information, interface similarity, contextual coupling etc. which can be considered to classify clone instances as short-lived or long-lived. Using the proposed metrics a classifier performance is measured. Moreover, by adding proposed metrics along with existing ones, the classifier performance is measured to know whether the proposed features can increase performance. Formally, this research aims to investigate the following research questions.

RQ5.1: Which metrics can be introduced to characterize clone lifetime and how the metrics perform to classify clones as short-lived and long-lived? — Clone lifetime can depend on co-change information, clone location, interface similarities etc. This question aims to investigate how the proposed metrics perform to characterize clone lifetime. Using metrics from four perspectives, a random forest classifier achieves an average AUC of 0.80 to 0.92 where clones' co-change information, method name dissimilarities, the number of parameters, clone located in the same file depicts as influential metrics to characterize clone lifetime.

RQ5.2: How the classifier performs using the proposed metrics and the state-of-the-art metrics? — Patanamon et al., conducted a study to characterize clone lifetime using metrics from three dimensions. This question aims to investigate proposed metrics performance comparatively with their metrics performance. The proposed metrics achieve 1 – 6% less AUC value than Patanamon et al., metrics independently. However, combining both metrics improve classifier performance by increasing AUC score 2 to 7%.

To answer these questions, an experiment has been performed on four open source software namely Maven1, Flink2, Pulsar3, and Pig5. Firstly, volatile clone classes are labeled as short-lived and long-lived clones by k-means clustering where short-lived clones are minimum lived clones cluster and other cluster clones are long-lived clones. 16 metrics from co-change history, clone location information, interface similarity and clone context are calculated for each of the clone class. After computing metric values for all labeled clones, a random forest classifier is used to classify the clones as short-lived and long-lived. Bootstrap sample with replacement is used to train and test the classifier. By averaging results of 1000 iterations, Area Under the Curve (AUC) is calculated for short-lived clones and following scores achieved – Flink (0.88), Pulsar (0.80), Maven (0.92) and Pig (0.89). To measure the importance of used metrics, Mean Decrease in Accuracy (MDA) is calculated which depicts that the number of times clone instances of a clone class co-changed as the most important metrics. Thereafter method name similarity, the number of parameters are important metrics for determining clone life expectancy.

Whether proposed metrics perform better than the state-of-the-art metrics of Patanamon et al., a comparative analysis is also conducted. It has been found that using only proposed metrics perform less than 1%(Maven) – 6%(Pulsar). However, combining proposed metrics and Patanamon et al., metrics have achieved 2%(Pulsar) – 7%(Flink) increase in Area Under the Curve. So the developers can incorporate proposed metrics with Patanamon et al., metrics to better characterize clone life expectancy.

The following sections of this chapter are organized as follows. Section 5.2 describes the whole approach, Section 5.3 presents experimented results of four software, Section 5.4 mentioned the threats to validity, and finally, Section 5.5 summaries the research.

5.2 Approach

Characterization of clone lifetime by different metrics broadly consists of 4 parts namely, *(i)* Clone Genealogies Extraction, *(ii)* Clone Labeling *(iii)* Feature Identification and Calculation and *(iv)* Classification. Clone genealogies are extracted for every software by detecting and mapping clones in every version. From that, the lifetime of a clone is also calculated. Secondly, clones are categorized into two groups namely, (a) short-lived and (b) long-lived clone by using a clustering technique. In the third step, proposed metrics are calculated and clones are classified by fitting a random forest model. Figure 5.1 presents the whole approach of clone lifetime characterization. Each of these steps is described below.

5.2.1 Clone Genealogies Extraction

A git repository is selected and all its release tags are identified using git utility[1]. From these tags, official releases are selected by using a regular expression. These release tags are sorted chronologically based on those commit time. Then, versions are created on these selected releases to detect clone in every version. From the clone result, version wise clone mapping is retrieved for clone genealogy.

To detect clone an incremental clone detection tool namely iClones is used [22]. It firstly detects clone for every version, then, the version to version change information is incorporated to report clone genealogies by mapping every clone in subsequent versions. From the clone genealogies, lifetime is calculated for every clone by counting the number of versions a clone alive in the version history. A clone which has been removed after some version from its origin are collected as a volatile clone. This volatile clones along with their lifetime are used in the next step.

5.2.2 Clone Labeling

To classify volatile clones as short-lived and long-lived clones, each clone needs to be labeled. The labeling of a clone depends on the lifetime of that clone. The lifetime can be defined by

[1]git tag

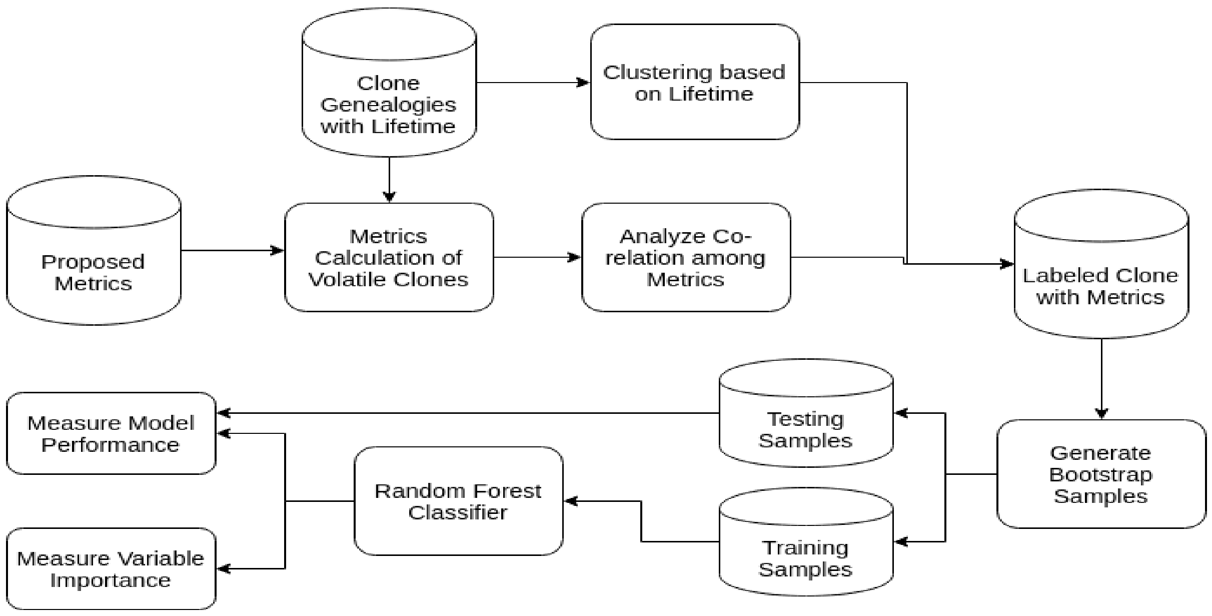

Clone Genealogies with Lifetime
Clustering based on Lifetime
Proposed Metrics
Metrics Calculation of Volatile Clones
Analyze Co-relation among Metrics
Labeled Clone with Metrics
Measure Model Performance
Testing Samples
Random Forest Classifier
Generate Bootstrap Samples
Measure Variable Importance
Training Samples

Figure 5.1: Overview of Clone Lifetime Characterization

a number of versions or time duration or a number of commits etc. For simplicity, clones are clustered based on their version wise lifetime. Then, the cluster which contains the minimum lifetime clones is considered as short-lived clones and other cluster clones are labeled as long-lived clones.

K-means clustering technique is used to cluster clones. Here, identifying optimal K values is difficult. So that, different indices like 'kl', 'ch', 'hartigan', 'ccc', 'scott' etc. from R package are used to find the optimal value of K [58]. This allows to reduce biasness towards a cluster as the best number of cluster is chosen from these methods.

5.2.3 Metric Identification and Calculation

To identify metrics for clone lifetime characterization, volatile clones are studied from different perspectives such as clone location, interface similarity change history and clone context. These metric classes have 16 individual metrics related to cloning. Table 5.1 contains all metric names, definitions and rationale behind considering it as a metric for clone lifetime characterization. Below the overview of these metric classes is given.

Clone Location A clone can be found within a method body, a file, a package and the same class hierarchy, or among method bodies, file, packages and different class hierarchies. Depending on the location, maintenance activities like refactoring and change consistency may vary. For example, refactoring of a clone class (aka clone) where all clone instances located in the same file is easier by extract method refactoring than surrounding on different files. Similarly, clones located in different packages likely to be changed independently and developers may not have access permission to another module's code for maintenance [4]. So that, clone location can be an important factor for clone lifetime characterization. Table 5.1 presents four metrics namely, same package, same file, same class hierarchy and the same method to cover different scopes of clone location. The value of these metrics are binary value which means if all clone instances are located within the method/file/package/ then value is 1 otherwise 0. Similarly, if the distance from the parent class to the child class means the depth of inheritance is same then the value will be 1 otherwise 0.

Interface Similarity A method level clone may have a relation with its method interface. Rakib et al., found that clones are related to its method interface and can be used to detect lightweight clone detection [59]. If clone fragments have different parameters or return type, those could not be merged due to type constraint. As a result, clones may live for a long time. Method name dissimilarity also denotes how much similar implementations clone have. Besides, many differences among clone instances of a clone class depicts merging or refactoring such clones requires a large number of parameters [60]. Parameterizing many differences among clone fragments leads to another code smell namely long parameter list [1]. So, developers may not be interested to refactor or remove such clones. To cover this phenomenon, interface similarity metric class has four metrics namely, minimum Levenshtein distance (Edit Distance) among method names, same return type, same access modifier, and the number of parameter among clone contained methods. Table 5.1 presents these metrics (from 5 to 8) names, definitions along with rationales for considering these metrics to characterize clone lifetime. The value for Edit Distance and the number of parameter is numeric value whereas same return type and same access modifier is binary (1 or 0) value.

Table 5.1: Metric Classes, Names, Descriptions and Rationales

Feature Class	**Name**	**Definition**	**Rationale**
Clone Location	1. samePkg	Clone Fragments of a clone class located in the same package or in different packages.	Clones located in the same package may go through similar changes for their common context. Similarly, clones located in different packages may depict Forking pattern of Kapser et. al. [32] in which clones evolve independently.
	2. sameFile	Clone Fragments of a clone class located in the same file or in a different files.	Similar to metric 1, Clones located in the same file may evolve consistently or may be removed after a while as refactoring clones of a file is easier than clones spanning from different files. For these clones may have lived for a short duration.
	3. sameClassHierarchy	Clone Fragments of a clone class located in the same class hierarchy or in different hierarchy levels of Object Oriented Programming (OOP).	Similar to metric 2, as clones located in the same OOP class hierarchy means clones located in the two child classes of a parent class may easier to refactor through pull-up method than different class hierarchy levels.

Continued on next page

Table 5.1 – *Continued from the previous page*

Metric Class	**Name**	**Definition**	**Rationale**
	4. sameMethod	Clone Fragments of a clone class located in the same method or different methods of an OOP class.	Clones located in the same method may live short duration as extract method refactoring may be applied immediately or refactoring is not possible for contextual coupling.
Interface Similarity	5. minLevenshteinDis AmongMethodNames	Differences among clone contained method names	Similar functionalities likely to have a similar method names. So the lifetime of a clone may vary depending on the similarity among method names.
	6. sameReturnType	The return type of clone contained methods	Similar to metric 6, this feature also characterize similarity among clone methods.
	7. sameAccessModifier	The access modifier of clone contained methods	The access modifier differences among cloned methods may characterize the lifetime as they also reflect the possibility of to be refactored.
	8. NumOfParameter AmongMethods	The number of parameters in the clone contained methods	More the differences among the number of parameters, methods likely to couple with different contexts so their lifetime may vary.
Change History	9. NumOfCommit AllCloneChange	The number of commits all clone fragments changed	The more consistently clones changed in the history the more likely to be lived for a long time.

Continued on next page

Table 5.1 – *Continued from the previous page*

Metric Class	**Name**	**Definition**	**Rationale**
	10. NumOf-Commit No-CloneChange	The number of commits no clone fragments changed	If clones are not changing frequently these may less likely to be removed from the repository.
	11. Nu-mOfCommit 1CloneChange	The number of commits one clone changed	Inconsistent changes in clone fragments of a clone group also, characterize clone lifetime.
	12. Nu-mOfCommit 2CloneChange	The number of commits two clones changed	Similar to metric 11, this also characterizes clones lifetime.
	13. Nu-mOfCommit 3CloneChange	The number of commits three clones changed	Similar to metric 12, this also characterizes clones lifetime where the number of clone fragments are more than 2 in a clone group.
Clone Code Context	14. NumOf-Frag BeginWith-ControlFlow	Clone code starts with a control statement like *for, while* etc.	As clone starts with a control statement, it is less likely to be changed as this clone may have happened unintentionally.
	15. NumOfFrag HasComplete-ControlFlow	Clone code contains control statements like *for, while* etc. completely	If the clone code contains a complete control statement, it can be refactored and likely to be lived a short time in the repository.
	16. codeType	Clone code type (i.e., main code or test code clone)	The type of source code where clones are found may characterize clones lifetime as test code less likely to be changed in the future.

Change History Clone lifetime can depend on previous change histories of clones as change histories contain association among clones [54, 25]. Researcher have also found that clones are related with change couplings [61]. The number of times all clone instances of a clone class changed means co-changed depicts the similarity preserving changes. This relation presents that the clone cannot be refactored so that it has been going through similar updates. Similarly, the number of times no clone instances of a clone class changed tells the no association among clone instances. Despite changing all clone instances of a clone class, some of those instances may be changed in some commits. This relation among clones represents less association and late propagation among clones. Moreover, the number of commits where no clone fragments changed denotes whether the clone class is concerning or not for frequent maintenance updates. Table 5.1 shows the metrics of change history metric class. Covering how much less association exists among clone siblings within a clone class, the number of commits only one/two/three clone fragments changed within that clone class is also considered. From Table 5.1, metrics 11 to 13, characterize these less associations. The value for all of these metrics is numeric value.

Clone Code Context Clone code context measures the contextual dependency among clone instances of a clone class. Previous studies have been found that clone context can be a useful characteristics to measure the harmfulness (whether a clone undergoes many changes with the evolution of the software) of a clone [57]. Moreover, it is also useful to determine the refactoring possibility of a clone [42]. If a clone class consists of clone instances which contain partial control flow of control statements such as for, while block etc. may less likely to be refactored. Whereas clones have complete control flow easier to refactor due to less contextual coupling [42]. Along with these, depending on the source code type that is the main code and test code, the clone may live long or short duration. Table 5.1 shows three metrics from 14 to 16, which may have an impact in clone lifetime characterization. The value for the number of fragment BeginWithControlFlow, HasCompleteControlFlow is numeric value whereas codeType value is binary value. If all clone instances are main code then the value is 1 otherwise 0.

Metric calculation

To characterize clone lifetime, each clone class is considered as an entity. Here, some metrics represent the value of a clone instance, and some metrics represent the value of a clone class. For example, the number of parameters is calculated for a clone instance and the number of times all clones changed is computed for a clone class. So that, metric values need to be abstracted from a single clone instance to clone class. Averaging the metric values of all clone instances, and taking the maximum difference among clone instances can be useful for such abstraction. So that, metric values are calculated by taking average and maximum differences of a clone class. These formulae are applicable for numeric value metrics, for example, the number of parameter among method names. Equation 5.1 and 5.2, computes these values. Equation 5.1 is used to compute the average metric value where C denotes a clone class, c denotes a clone instance, $Metric\_Value(c)$ is the value of a clone instance and $S|C|$ is the number of clone instances within the clone class.

$$Average\_Metric\_Value(C) = \frac{\sum_{c \epsilon S(C)} Metric\_Value(c)}{|S(C)|} \tag{5.1}$$

Equation 5.2 is used to compute the maximum difference among clone instances of a clone class where $M(c_i)$ denotes the metric values of clone instances.

$$Difference\_Metric\_Value(C) = max(\{abs(M(c_1) - M(c_2))|c_1, c_2 \epsilon S(C)\}) \tag{5.2}$$

In the next step, computed metric values for all volatile labeled clones, that is, short-lived and long-lived clones will be used to build a classifier model which can classify clone classes into short-lived and long-lived clones.

5.2.4 Classification Model Construction

Previously calculated metrics for volatile clones which are already labeled in Subsection 5.2.2 are used to build a classification model. Here, 16 metrics will be used as the feature vector or input variables and clone life expectancy will be used as the response variable. Then, Random Forest (RF) classifier is used to build the classification model which can classify clones as short-lived and long-lived clones. Finally, the performance of the classifier is measured based on Area Under the Curve (AUC). This AUC value denotes the separability of the RF classifier. Below the steps are described.

Co-relation Analysis Firstly, to overcome the co-relation problem among metrics, highly co-related metrics are removed from the feature vector by using the pairwise Spearman rank correlation test (ρ) [62]. Highly correlated metrics, whose $|\rho| > 0.7$ are removed because such metrics are redundant. For example, if two metrics, assume, sameFile and codeType has co-relation value > 0.7, co-relation between response variable and codeType, and response variable and sameFile are computed. Which input variable has maximum co-relation with the response variable is kept as independent variables. These variables are used during model construction.

Bootstrap Sample Generation To validate the RF classifier, the Out-Of-Sample Bootstrap (OOB) validation technique is used as it provides less biased performance estimates than n-fold cross validation [63]. OOB generates sample which is the same population size as the original dataset. Here, sample generation with replacement is also used which generates 36.8% new data points that are not present in the bootstrap sample. Since the number of short-lived and long-lived clone classes are not equal, means, there exists a class imbalance problem, OOB bootstrap can be useful for such a situation. Then, bootstrap samples are used as training dataset and the remaining samples that are not used in the training are used as testing dataset. This process is repeated 1,000 times, and the average result is reported as the performance estimation.

Random Forest Classifier Construction A Random Forest (RF) classifier [17] is built by using independent variables of Section 5.2.4. Since the RF has a good accuracy from tree based approach and more robust to noisy data so that RF is chosen as a classifier. The RF classifier uses a combination of Decision Tree (DT)s inherently to decide which class the instance will be. Each tree output a class that is short-lived or long-lived. this output class is considered as a vote for random forest output. The most popular voted class is counted as the outcome of the RF classifier.

5.3 Experimentation and Result Analysis

The above mentioned approach in Section 5.2, is applied to four Open Source Software (OSS) namely Flink, Maven, Pig and Pulsar. Table 5.2 presents the experimented software which are described in Subsection 4.3.1. From these software, clone genealogies are extracted and volatile clones are computed. Each volatile clone is labeled with short-lived or long-lived using K-means clustering technique. $3rd$ column of Table 5.3 presents the optimal number of clusters for each software. From these clusters, one cluster instances are labeled with short-lived and others are labeled with long-lived as stated in Subsection 5.2.2. Then, a Random Forest (RF) classifier is used to classify clones as short-lived and long-lived clones. As RF is an ensemble learning technique and provides good performance in binary classification than other tree-based classifier, so that, RF classifier is used to classify clones. Here, short-lived clones are considered as positive class and long-lived clones as negative class since we likely to identify short-lived clones. From the obtained results, the research questions are answered.

Table 5.2: Overview of the Experimented Software

Software	Study Period	#Releases	#Commits
Flink	2010-12-15 to 2018-10-17	73	15,002
Pulsar	2016-09-06 to 2018-10-12	59	2,447
Pig	2009-03-05 to 2016-06-07	57	2,857
Maven	2003-09-01 to 2018-06-17	54	10,420

5.3.1 Performance of the Proposed Metrics

To investigate the applicability of metrics like NumOfCommitAllFileChange, minimalLevenshteinDistanceAmongMethodNames, sameFile etc. to determine clone life expectancy following research question is used. This research investigates the performance of the classifier using the proposed metrics. Then most influential metrics to determine clone life expectancy are reported.

Research Question (RQ5.1): Which metrics can be introduced to characterize clone lifetime and how the metrics perform to classify clones as short-lived and long-lived?

Table 5.3: Experimented Volatile Clones

software	# versions	# volatile clone	Optimal # clusters	# short-lived clone	# long-lived clone
Flink	15	2526	4	1593	933
Pulsar	12	663	4	326	337
Pig	15	2085	4	1584	500
Maven	37	353	7	212	141

To answer this question, proposed metrics of Table 5.1 are used to fit a random forest classifier and the Area Under the receiver operating characteristic Curve (AUC) is reported to evaluate the performance. Table 5.3 presents the classification dataset where the number of volatile clones, short-lived and long-lived clones are shown. From the table, it can be seen that Flink has the maximum number of clones and Maven has the minimum number of clones. It can also be seen that the data has the class imbalance problem, for example, Pig software has 24% (500/2085) long-lived clones. For this reason, the bootstrap sample with replacement and out-of-sample validation is used to train and test the classifier. Iterating 1000 times, RF classifier is trained, tested and AUC value is computed, to report the average value. To evaluate the performance, AUC value is used because AUC captures an area below the curve of the true positive rate, that is the proportion of correctly classified short-lived clones against false positive rate, that is, the proportion of misclassified long-lived clones. The AUC value is ranges between 0 (worst) and 1 (best), so the AUC value towards 1 is considered as a good performance indication of the classifier.

Table 5.4: AUC, TP Rate and FP Rate of Experimented Software

	Flink	Pulsar	Maven	Pig
AUC	0.88	0.80	0.92	0.89
True positive rate	0.85	0.70	0.92	0.91
False positive rate	0.27	0.24	0.19	0.31

Table 5.4 shows the performance of the fitted RF classifier which achieves an average AUC of 0.80(Pulsar) – 0.92(Maven). These results indicate that the RF classifier performs well to determine a clone life expectancy by using proposed metrics. Table 5.4 also shows that the classifier achieves an average true positive rate of 0.70(Pulsar) – 0.92(Maven). This result indicates that 70% to 92% of the short-lived clones can be identified by the fitted RF classifier. On the other hand, the false positive rate of 0.19(Maven) – 0.31(Pig) indicates that 19% to 31% of long-lived clones are misclassified as short-lived clones.

Observation 1: By using the proposed metrics, a random forest classifier can achieve an average AUC of 0.80(Pulsar) – 0.92(Maven).

To know the important metrics to characterize clone lifetime, feature importance in random forest classifier is computed by using the Mean Decrease in Accuracy (MDA). MDA measures how much accuracy decreases of the classifier by removing one metric and using all other metrics during building the tree. As the classifier fitted 1000 times, each metrics has 1000 MDA values. So, the average value of MDAs is calculated for every metrics. Then, the metrics are prioritized based on MDA, that is which metric has the highest MDA raked first and which metric has the lowest MDA raked last. It mainly computes the sensitivity of a metric to the classifier. Doing so, influential metrics are identified for characterization.

Table 5.5: Metrics Influence to Random Forest Classifier

Software	Rank	Metric	MDA	SD
Flink	1	NumOfCommitAllCloneChange	0.16	0.00613
	2	NumOfCommit2CloneChange	0.10	0.00586
	3	minimalLevenshteinDistanceAmongMethodNames	0.04	0.00385
	4	AvgNumOfParameterAmongMethods	0.03	0.00317
	5	NumOfCommit3CloneChange	0.02	0.00247
Pulsar	1	NumOfCommitAllCloneChange	0.15	0.01285
	2	minimalLevenshteinDistanceAmongMethodNames	0.05	0.00667
	3	codeType	0.05	0.01082
	4	NumOfFragHasCompleteControlFlow	0.03	0.00732
	5	AvgNumOfParameterAmongMethods	0.02	0.00541
Pig	1	NumOfCommit2CloneChange	0.11	0.00671
	2	NumOfCommitNoCloneChange	0.07	0.00519
	3	NumOfCommitAllCloneChange	0.07	0.00479
	4	sameFile	0.04	0.00502
	5	minimalLevenshteinDistanceAmongMethodNames	0.04	0.00506
Maven	1	NumOfCommitAllCloneChange	0.28	0.01682
	2	sameMethod	0.12	0.01775
	3	sameFile	0.04	0.01095
	4	AvgNumOfParameterAmongMethods	0.04	0.00860
	5	DiffInNumOfParameterAmongMethods	0.01	0.00311

Table 5.5 shows the top 5 metrics for every software. From the table, it can be seen that the number of commits all clones co-changed is the most important metric. For Pig software, the number of commits only 2 clone fragments of a clone class changed, depicts as most important. So that, co-change information is the most prominent metrics for clone life characterization. Along with this, several metrics like method name dissimilarities (minimalLevenshteinDistanceAmongMethodNames), the number of parameters (AvgNumOfParameterAmongMethods), clone instances location (sameFile) also depict as important metrics for clone lifetime characterization.

5.3.2 Comparison between state-of-the-art Metrics

Patanamon et al., conducted an empirical study on clone life expectancy considering 38 metrics from 3 dimension namely clone, product and process dimension. They classify volatile clones as short-lived and long-lived by using those 38 metrics. Their metrics are SiblingCount, CloneLineCount, TokenCount, DirectoryDistance, EditDistance, CloneType, LineCount,, LineCodeCount, LineCodeDeclCount, LineCodeExeCount, StmtCount, StmtDeclCount, StmtExeCount, RatioLineCount, RatioLineCodeCount, RatioLineCodeDeclCount, RatioLineCodeExeCount, RatioStmtCount, RatioStmtDeclCount, RatioStmtExeCount, FanOut, FanIn, Cyclomatic, CyclomaticModified, CyclomaticStrict, Essential, MaxNesting, CommentLineCount, RatioCommentToCode, Churn, LineAdded, LineDeleted, CommitCount, DeveloperCount, MajorDeveloperCount, FixCommitCount, NewFeatureCommitCount and ImproveCommitCount. Using the similar approach as stated in Section 5.2, the performance of the classifier is measured for these state-of-the-art metrics.

Research Question (RQ5.2): How the classifier performs using the proposed metrics and the state-of-the-art metrics?

Table 5.6: Comparative Result between the Proposed vs Existing Metrics

Software	AUC			TP			FP		
	New	Old	All	New	Old	All	New	Old	All
Flink	0.88	0.85	0.92	0.85	0.87	0.87	0.27	0.38	0.22
Pulsar	0.80	0.86	0.88	0.70	0.79	0.81	0.24	0.21	0.19
Pig	0.89	0.94	0.94	0.91	0.96	0.96	0.31	0.30	0.30
Maven	0.92	0.93	0.93	0.92	0.95	0.94	0.19	0.23	0.24

Table 5.6 presents the comparative results of the random forest classifier by using the proposed 16 metrics, existing 38 metrics (proposed by Patanamon et al.) and combining all (16 and 38) metrics. From the table's AUC values, it can be seen that using only proposed metrics the classifier performs 3% better for Flink but performs less 1% for Maven, 5% for Pig and 6% for Pulsar software than Patanamon et al. metrics. The reason classifier performs better for Flink software due to decrease in False Positive rate (11%). However, the proposed metrics did not perform worse than 6% for any experimented software.

Afterward, the proposed and existing metrics are combined to check whether combined metrics perform better than existing ones such as LineCount, LineAdded etc. To do so, all metrics and only existing metrics used to fit the random forest classifier independently. From the AUC column of Table 5.6, it can be seen that combined metrics performed 2%(Pulsar) and 7%(Flink) better than using only Patanamon et al., metrics. Moreover, the False Positive rate is reduced by 16% for Flink and 2% for Pulsar. Performance for other software did not vary. So that, leveraging the proposed metrics with the existing metrics can improve the performance of the classifier means that the life expectancy (short-lived or long-lived) of a newly introduced clone can be determined 2% – 7% more correctly than using only Patanamon et al., metrics.

5.4 Threats to Validity

Characterization of clone using clone location, interface similarity, clone context etc. has some internal and external threats to validity. Threats are described below.

Internal Validity Clone life expectancy (that is short-lived and long-lived clone) characterization, firstly depends on the clone genealogies. Clone genealogy relies on the mapping of clones in subsequent versions. So, the accuracy of clone detection and clone genealogies extraction are important for the following analysis. Inaccurate detection and genealogy extraction may provide different results which incur internal threats to validate the research. To mitigate this threat, *iClones* tool is used. It is an incremental clone detection tool and appropriate for evolutionary research. It provides more accurate clone genealogies by using change information of subsequent versions. It is also popular and was previously used in other researches [41, 6].

Secondly, labeling clone as short-lived and long-lived heavily influence classification performance and predicting clone life expectancy. To labeling clones, K-means clustering technique is used where optimal value for K is chosen by computing different methods of optimal K selection, such as Krzanowski et al. [64], Calinski et al. [65] etc. approaches. Then, the cluster of clones which has the minimum lifetime, are labeled as short-lived and others are labeled as long-lived. This procedure mitigates the labeling problem.

Thirdly, the number of instances for short-lived and long-lived is not equal which creates class imbalance problem. Due to this reason, a classifier may always predict the majority class and get higher accuracy. To reduce bias towards a class, bootstrap samples with replacement is used during training. Then, Out-Of-Bag (OOB) validation technique is used to measure the performance. Since, OOB technique measures the performance by testing newly generated data points, so that, single iteration result may not be reliable. So that, iterating 1000 times, the average result of Area Under the Curve (AUC) value of short-lived clones are reported. This way class imbalance problem and the performance bias are mitigated.

External Validity The performance of the proposed metrics may vary depending on the experimented software. Since the number of versions is considered as the lifetime of a clone, so that defining lifetime differently such as the number of commits, time duration may also depict different results. To mitigate external validity, four software repositories which are previously used in clone related research [6] are analyzed.

5.5 Summary

This chapter presents clone lifetime characterization using different metrics from co-change history, clone location information, interface similarity and clone context. A model is built to classify clone as short-lived or long-lived to identify whether a newly introduced clone will be short-lived or long-lived. To do so, some metrics are proposed that can be incorporated to better characterize clone lifetime. Using proposed metrics with a Random Forest (RF) classifier, the model achieved an average AUC value 0.88, 0.80, 0.92 and 0.89 for Flink, Pulsar, Maven and Pig respectively. This means that using proposed metrics, RF classifier can 88% – 92% correctly identify that a clone will be short-lived. Computing important metrics for characterization, it has found that the number of times clones co-changed is the most influential metrics. The method name dissimilarity, the number of parameters, the number of fragments have complete control flow, clones located in the same file are also important metrics. Moreover, the performance of the classifier is compared using the state-of-the-art metrics taken from Patanamon et al. [6]. Comparing the performance of RF classifier, it has been found that proposed metrics perform 1% – 6% less than Patanamon et al., metrics. However, combining proposed metrics with existing metrics, the classifier performance increases 2% – 7% for Pulsar and Flink and not decreases for Pig and Maven. So, it can be concluded that practitioners can leverage proposed metrics to better characterize clone life expectancy. In the future, a clone maintenance support tool will be developed which can identify clone life expectancy by using these metrics and recommend maintainer clones which require immediate refactoring.

Chapter 6

Conclusion

Code clone is identical or similar code fragments in the source code. It has controversial impacts on maintenance activities. It may be considered as a time saver to developers for reusing existing implementation. On the other hand, it may be responsible for bug duplication, bug propagation, and future maintenance difficulties. Due to these reasons, clones need to be analyzed from different perspectives. Analyzing evolutionary implications of clones can help to establish facts which lead to proper management of clone. At the same time, to ease clone maintenance activities, which clone needs more attention and which can be avoided, needs to be characterized. It helps to build an automated tool that can assist the maintainer. This chapter summarizes the evolutionary implications of code clone by analyzing clone lifetime with respect to clone type, clone location and clone code type. Besides, it describes clone lifetime characterization by using various metrics. At the end, this chapter concludes the research by outlining the possible future research directions.

6.1 Thesis Summary

Code duplication can be harmful or beneficial depending on the context of cloning. So that, special attention is required to better manage cloned codes. Clones, which live for a short period of time, are not as beneficial as long lived clones, because, short lived clones are removed immediately. Different factors can be associated with this phenomena. Analyzing clone lifetime using various factors facilitates understanding about cloning and provides support towards efficient clone management. Moreover, such study validates conventional perception about cloning. To this aim, this research conducted an empirical study where code type, clone location and clone type is used to analyze cloning practices along with clone lifetime.

Whether a clone will be short-lived or long-lived, need to be characterized to better allocate clone maintenance efforts. The factors of clone lifetime analysis along with some others metrics are incorporated to this lifetime characterization.

6.1.1 Clone Evolutionary Discussion

The clone evolutionary study focuses on gathering facts about clone which were not previously explored. The first research question aims to explore such evolutionary observations by analyzing clone lifetime based on various clone features like clone type. In subsequent versions, clone evolution is observed from code type, clone location, and clone volatility based on clone type and clone location. To answer the research question, an empirical study has been performed on four popular Java software. Then, obtained findings are reported and incorporated to better characterize clone lifetime, which is used to answer second research question.

To do so, software repository is traversed and multiple versions are created. Then, clones are detected and mapped in subsequent versions to extract clone genealogies. Categorizing clones into main code and test code, Inter-File and Intra-File some conventional wisdom is validated. Then, clone lifetime is calculated to understand volatile clones and their lifetime frequencies.

The first finding of the main code clone and test code clone refers that developers maintain the main code clone than test code clone. So that, the overall number of clone increases in successive versions in the test code clones more than main code clones. Secondly, knowing that, cloning may create future maintenance problem, developers still copy code fragment within the same file without following a proper design. For that reason, clone exists more in Intra-File than Inter-File. The third finding, Type-3 clone is more volatile than other types of clone, denotes that developers are less likely refactor or maintain exact Type-1 and parameterized Type-2 clones. As Type-3 (gapped) clones by definition have more differences and undergo many updates they are more likely to be volatile. Besides these, most of the volatile clones live for only one version which represents some clones may be created for experimental purposes. These findings help to identify relevant metrics to characterize clone lifetime.

6.1.2 Clone Lifetime Characterization

All clones removal from a repository is not beneficial from the clone maintenance perspectives due to the changing nature of clone. Short-lived clones can be created for experimental purposes and will be removed immediately. So, to improve the efficiency of clone management efforts, clone lifetime needs to be characterized. It helps to understand whether a newly introduced clone will be short-lived or long-lived.

Considering clones co-change information, interface similarity, clone context and clone location 16 metrics are proposed to characterize clones. To know the impacts of metrics, a random forest classifier is used to classify clones as short-lived and long-lived. The classifier achieves 0.80 – 0.92 AUC score to classify a short-lived clone as short-lived. This performance shows that the metrics are useful to characterize clone lifetime. Which metrics are more important to the classifier to achieve such separability of long-lived and short-lived clones, impacts of metrics are also measured. It has been found that, the number of times clones co-changed, method name dissimilarity, the average number of parameters and file location of clones are important metrics for characterize clones.

The performance of the proposed metrics is compared with the existing metrics proposed by Patanamon et al. [6]. The performance of the classifier is measured by using these different metrics independently. Using only existing metrics, the classifier achieves 0.85 – 0.94 AUC value. Afterwards, using all metrics, that is, the proposed and existing metrics, the classifier achieves 0.88 – 0.94 AUC value for studied software. From the results, it has been found that the proposed metrics combined with existing metrics increase 2% – 7% AUC value of the classifier than using only existing metrics. This finding suggests that practitioners can leverage the proposed metrics along with the existing ones to better characterize clone lifetime.

6.2 Future Directions

In this research, clone evolutionary observations are reported to characterize clone lifetime. Some future research scopes exist that can be performed to identify clone evolutionary behaviours. Following points address the potential future directions derived from this research.

- In the evolutionary study in Chapter 4, some test code clones are identified as volatile along with main code clones. A study needs to be performed to know whether the test code clones are refactored intentionally or unintentionally, to comprehend clone maintenance activities in test code. As recent studies also focus on test code clone maintenance [66, 67], empirical investigation can be performed to know test code clone changing history.

- Since it has been found that Intra-File clone exists more than Inter-File clone, an exploratory study needs to be performed to identify reasons why developers opt in cloning a code fragment within a file while a proper design can be followed.

From the characterization of clone lifetime, it has been found that different metrics such as clones co-change history, clone location, clone context etc. can characterize clone life expectancy, that is, lifetime. So, using such metrics automated tool can be developed. A brief description of such tools are given below.

- A clone management support tool can be developed by using the clone lifetime characterization metrics, to tag each clone whether it will be short-lived or long-lived. The number of versions it already lives in the repository by changing consistently or inconsistently can be incorporated. This helps the maintainer to take decisions on which clones require attention during maintenance activities. This tagged information can also inform developers which code fragments should not be copied.
- An automated clone recommendation tool can also be developed, which uses the clone lifetime characterization metrics and recommend beneficial clone for refactoring by learning from previous history and current context of the clone.

Bibliography

[1] M. Fowler, *Refactoring - Improving the Design of Existing Code.* Addison Wesley object technology series, Addison-Wesley, 1999.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 577–591, Sept. 2007.

[3] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1–7, IEEE, 2017.

[4] W. Zhao, Z. Xing, X. Peng, and G. Zhang, "Cloning practices: Why developers clone and what can be changed," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, (Washington, DC, USA), pp. 285–294, IEEE Computer Society, 2012.

[5] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 187–196, ACM, 2005.

[6] P. Thongtanunam, W. Shang, and A. E. Hassan, "Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones," *Empirical Software Engineering*, pp. 1–36, 2018.

[7] D. Cai and M. Kim, "An empirical study of long-lived code clones," in *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, (Berlin, Heidelberg), pp. 432–446, Springer-Verlag, 2011.

[8] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 311–320, ACM, 2011.

[9] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pp. 1157–1168, 2016.

[10] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, (New York, NY, USA), pp. 187–196, ACM, 2005.

[12] H. Noon, "How to deal with test and production code." https://hackernoon.com/how-to-deal-with-test-and-production-code-c64acd9a062. Online; accessed 01 September 2018.

[13] M. Corporation, "The 5 clustering algorithms data scientists need to know." https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68. Online; accessed 04 February 2019.

[14] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: analysis and implementation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 881–892, July 2002.

[15] RDocumentation, "Nbclust." https://www.rdocumentation.org/packages/NbClust/versions/3.0/topics/NbClust. Online; accessed 14 November 2018.

[16] T. D. Science, "Understanding auc - roc curve." https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5. Online; accessed 11 January 2019.

[17] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, pp. 5–32, Oct. 2001.

[18] M. Corporation, "An introduction to random forest." https://towardsdatascience.com/random-forest-3a55c3aca46d. Online; accessed 10 February 2019.

[19] M. F. Zibran and C. K. Roy, "Towards flexible code clone detection, management, and refactoring in ide," in *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, (New York, NY, USA), pp. 75–76, ACM, 2011.

[20] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pp. 18–33, 2014.

[21] "Nicad4 clone detector." http://www.txl.ca/nicaddownload.html.

[22] N. Gde and R. Koschke, "Incremental clone detection," in *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 219–228, March 2009.

[23] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[24] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Proceedings of the 29th international conference on Software Engineering*, pp. 158–167, IEEE Computer Society, 2007.

[25] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic identification of important clones for refactoring and tracking," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, pp. 11–20, 2014.

[26] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[27] D. Silva, N. Tsantalis, and M. T. Valente, “Why we refactor? confessions of github contributors,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pp. 858–870, 2016.

[28] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *Proceedings of the 40th International Conference on Software Engineering*, ICSE ’18, (New York, NY, USA), pp. 483–494, ACM, 2018.

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

[30] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, “An empirical investigation into a large-scale java open source code repository,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM ’10, (New York, NY, USA), pp. 11:1–11:10, ACM, 2010.

[31] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, “Déjàvu: A map of code duplicates on github,” *Proc. ACM Program. Lang.*, vol. 1, pp. 84:1–84:28, Oct. 2017.

[32] C. Kapser and M. W. Godfrey, “”cloning considered harmful” considered harmful,” in *Reverse Engineering, 2006. WCRE’06. 13th Working Conference on*, pp. 19–28, IEEE, 2006.

[33] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, “Cloning by accident: an empirical study of source code cloning across software systems,” in *2005 International Symposium on Empirical Software Engineering, 2005.*, pp. 10 pp.–, Nov 2005.

[34] R. Koschke and S. Bazrafshan, “Software-clone rates in open-source programs written in c or c++,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, pp. 1–7, March 2016.

[35] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones,” *Empirical Softw. Engg.*, vol. 15, pp. 1–34, Feb. 2010.

[36] T. Bakota, R. Ferenc, and T. Gyimothy, “Clone smells in software evolution,” in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 24–33, IEEE, 2007.

[37] E. Duala-ekoko and M. P. Robillard, “Clonetracker: Tool support for code clone management,” in *In Proc. Intl Conf. on Software Engineering (ICSE*, pp. 843–846, ACM, 2008.

[38] “java-diff-utils.” `https://code.google.com/archive/p/java-diff-utils/`. accessed 20 Nov. 2017.

[39] J. Krinke, “A study of consistent and inconsistent changes to code clones,” in *14th Working Conference on Reverse Engineering (WCRE 2007)*, pp. 170–178, Oct 2007.

[40] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, “An empirical study on inconsistent changes to code clones at release level,” in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pp. 85–94, 2009.

[41] W. Wang and M. W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 331–340, 2014.

[42] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," *CoRR*, vol. abs/1807.11184, 2018.

[43] S. Harris, "Simian - similarity analyser." `http://www.harukizaemon.com/simian/`.

[44] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pp. 273–282, 2011.

[45] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Sci. Comput. Program.*, vol. 77, pp. 760–776, June 2012.

[46] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pp. 1227–1234, ACM, 2012.

[47] V. Saini, H. Sajnani, and C. Lopes, "Comparing quality metrics for cloned and non cloned java methods: A large scale empirical study," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 256–266, Oct 2016.

[48] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pp. 227–237, 2017.

[49] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 485–495, IEEE, 2009.

[50] Y. Higo and S. Kusumoto, "How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 222–231, IEEE, 2012.

[51] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *2008 IEEE International Conference on Software Maintenance*, pp. 227–236, Sept 2008.

[52] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM J. Comput.*, vol. 26, pp. 1343–1362, Oct. 1997.

[53] U. o. B. Software Engineering Group, "Iclones." `http://www.softwareclones.org/iclones.php`. Online; accessed 28 August 2018.

[54] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pp. 114–123, 2014.

[55] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, (New York, NY, USA), pp. 455–465, ACM, Saint Petersburg, Russia, August 2013.

[56] J. Harder and N. Göde, "Efficiently handling clone data: Rcf and cyclone," in *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, (New York, NY, USA), pp. 81–82, ACM, 2011.

[57] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can i clone this piece of code here?," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, (New York, NY, USA), pp. 170–179, ACM, 2012.

[58] M. Charrad, N. Ghazzali, V. Boiteau, and A. Niknafs, "Nbclust: An r package for determining the relevant number of clusters in a data set," *Journal of Statistical Software, Articles*, vol. 61, no. 6, pp. 1–36, 2014.

[59] M. R. H. Misu, A. Satter, and K. Sakib, "An exploratory study on interface similarities in code clones," in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pp. 126–133, Dec 2017.

[60] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 164–174, ACM, 2014.

[61] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, "Relation of code clones and change couplings," in *Fundamental Approaches to Software Engineering* (L. Baresi and R. Heckel, eds.), (Berlin, Heidelberg), pp. 411–425, Springer Berlin Heidelberg, 2006.

[62] *Spearman Rank Correlation Coefficient*, pp. 502–505. New York, NY: Springer New York, 2008.

[63] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap.* No. 57 in Monographs on Statistics and Applied Probability, Boca Raton, Florida, USA: Chapman & Hall/CRC, 1993.

[64] W. J. Krzanowski and Y. T. Lai, "A criterion for determining the number of groups in a data set using sum-of-squares clustering," *Biometrics*, vol. 44, no. 1, pp. 23–34, 1988.

[65] A. Casillas, M. T. G. de Lena, and R. Martínez-Unanue, "Document clustering into an unknown number of clusters using a genetic algorithm," in *TSD*, 2003.

[66] W. Hasanain, Y. Labiche, and S. Eldh, "An analysis of complex industrial test code using clone analysis," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 482–489, July 2018.

[67] B. van Bladel and S. Demeyer, "A novel approach for detecting type-iv clones in test code," in *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pp. 8–12, Feb 2019.