الجمهورية الجزائرية الديمقراطية الشعبية

وزارة التعليم العالي والبحث العلمي

جامعة سعيدة د. مولاي

الطاهر

كلية العلوم

Master's thesis

Specialty: Computer modeling of knowledge and reasoning (MICR)

Theme:

Software clones detection using multiplayer perception (MLP)

Presented by:

SEBIH Asmaa

Directed by:

MOSTEFAI Abdelkader

Promotion 2023 - 2024

Thanks

I thank the good Almighty Allah, who gave me the strength, the will and the courage to complete this modest work. i extend our deepest thanks to Dr. MOSTEFAI Abdelkader for his supervision, his listening, his elucidations, his advice, his directives and encouragements that he has poured into us. We thank Mrs. the jurors for the interest they have shown in this work by agreeing to be examiners. Thus, i extend my warmest thanks to all the people who have helped directly or indirectly through the fruit of their knowledge throughout the duration of my educational journey. To all the teachers in the computer science department Simply to all those who deserve our thanks.

Dedication

First of all, I thank my God for everything

I dedicate this modest work to:

The dearest parents in the world, my father and my mother, no dedication could not my respect, my love and my consideration for the sacrifices you have made for my education and my good I thank you for all the support and love you have given me since my childhood, I wish them a long life and may God protect them and protect them.

To my dear sister, and my dears, who have always been by my side and have always supported me throughout these long years of study.

And I would like to express my sincere thanks to my supervisor
Dr .MOSTEFAI Abdelkader

Summary .

Data mining techniques play a crucial role in addressing the challenge of software clone detection, with a particular focus on *Multilayer Perceptrons* (MLP). Code clones, which are duplicated or similar code segments, can significantly increase the risk of software bugs and vulnerabilities.

This investigation aims to develop an MLP-based model designed to automatically identify software clones. These clones can be indicators of potential flaws, and by analyzing the similarities between code fragments, the proposed model seeks to detect these clones efficiently. This approach contributes to a more effective strategy for managing and reducing software defects.

The study contrasts traditional methods of clone detection, such as manual inspection and code similarity analysis, with contemporary machine learning techniques. It introduces an MLP-based approach to automate the clone detection process, offering a more accurate and scalable solution for identifying problematic code segments. The model's performance is evaluated using real-world datasets, showcasing its superiority over traditional methods in terms of precision and efficiency.

Through this research, the thesis significantly advances the field of software engineering by leveraging advanced neural networks and data mining techniques to enhance the software clone detection process, ultimately improving the reliability and security of software systems.

Résumé .

Les techniques de *data mining* jouent un rôle crucial dans la résolution du défi de la détection des clones logiciels, en mettant l'accent sur les *Multilayer Perceptrons* (MLP). Les clones de code, qui sont des segments de code dupliqués ou similaires, peuvent considérablement augmenter le risque de bogues logiciels et de vulnérabilités.

Cette recherche vise à développer un modèle basé sur les MLP conçu pour identifier automatiquement les clones logiciels. Ces clones peuvent être des indicateurs de failles potentielles, et en analysant les similarités entre les fragments de code, le modèle proposé cherche à détecter efficacement ces clones. Cette approche contribue à une stratégie plus efficace pour la gestion et la réduction des défauts logiciels.

L'étude compare les méthodes traditionnelles de détection des clones, telles que l'inspection manuelle et l'analyse de similarité du code, avec des techniques contemporaines d'apprentissage automatique. Elle introduit une approche basée sur les MLP pour automatiser le processus de détection des clones, offrant une solution plus précise et évolutive pour identifier les segments de code problématiques. Les performances du modèle sont évaluées à l'aide de jeux de données réels, montrant sa supériorité par rapport aux méthodes traditionnelles en termes de précision et d'efficacité.

Grâce à cette recherche, la thèse apporte une avancée significative dans le domaine de l'ingénierie logicielle en exploitant les réseaux neuronaux avancés et les techniques de *data mining* pour améliorer le processus de détection des clones logiciels, améliorant ainsi la fiabilité et la sécurité des systèmes logiciels.

ملخص

تلعب تقنيات تعدين البيانات دورًا حاسمًا في معالجة تحدي اكتشاف استنساخ البرامج، مع التركيز بشكل خاص على (Multilayer Perceptrons (MLP. يمكن أن تزيد استنساخات التعليمات البرمجية، التي هي عبارة عن أجزاء مكررة أو متشابهة من التعليمات البرمجية، بشكل كبير من خطر حدوث أخطاء وثغرات في البرامج.

يهدف هذا البحث إلى تطوير نموذج قائم على MLP مصمم لتحديد استنساخات البرامج تلقائيًا. يمكن أن تكون هذه الاستنساخات مؤشرات على العيوب المحتملة، ومن خلال تحليل أوجه التشابه بين أجزاء التعليمات البرمجية، يسعى النموذج المقترح إلى اكتشاف هذه الاستنساخات بكفاءة. يساهم هذا النهج في استراتيجية أكثر فعالية لإدارة وتقليل عيوب البرامج.

تقارن الدراسة بين الأساليب التقليدية لاكتشاف الاستنساخ، مثل الفحص اليدوي وتحليل تشابه التعليمات البرمجية، وتقنيات التعلم الآلي المعاصرة. تقدم نهجًا قائمًا على MLP لأتمتة عملية اكتشاف الاستنساخ، مما يوفر حلاً أكثر دقة وقابلية للتطوير لتحديد أجزاء التعليمات البرمجية الإشكالية. يتم تقييم أداء النموذج باستخدام مجموعات بيانات واقعية، مما يُظهر تفوقه على الأساليب التقليدية من حيث الدقة والكفاءة.

من خلال هذا البحث، تعمل الرسالة على تطوير مجال هندسة البرمجيات بشكل كبير من خلال الاستفادة من الشبكات العصبية المتقدمة وتقنيات استخراج البيانات لتعزيز عملية اكتشاف استنساخ البرامج، مما يؤدي في نهاية المطاف إلى تحسين موثوقية وأمان أنظمة البرمجيات.

Trade table

List of tabls

List of Figures

General introduction

The complexity of software development often involves the reuse of existing code to streamline production and enhance quality. However, this practice can lead to code cloning, where identical or similar code segments appear across different parts of a project or across multiple projects. While code cloning can expedite development, it also poses risks to software maintainability, clarity, and overall quality, potentially increasing the likelihood of bugs and security vulnerabilities.

Detecting software clones is a critical area in software engineering, addressing the challenge of identifying these duplicated or similar code segments. Traditional methods of clone detection often rely on manual inspection and code similarity analysis, which can be time-consuming and prone to inconsistencies as software systems scale.

This thesis aims to advance the field of software clone detection by employing data mining techniques, with a particular emphasis on Multilayer Perceptrons (MLP). By leveraging MLPs, the research seeks to automate the detection of code clones, thereby improving both the efficiency and accuracy of identifying problematic code segments.

The use of machine learning and deep learning methods, especially MLPs, has recently transformed the landscape of code analysis. These techniques enable the automated examination of extensive code repositories, identifying patterns and similarities indicative of code clones. The objective of this thesis is to integrate data mining approaches with MLPs to enhance the detection of code clones, providing a more effective and scalable solution compared to traditional methods.

This research will explore the theoretical background of code clone detection, review traditional and modern methodologies, and implement an MLP-based model for detecting code clones. The effectiveness of the proposed model will be evaluated using real-world datasets, comparing its performance against existing techniques.

The research objectives are:

- To review the current state of research on software clone detection and data mining techniques in software engineering.
- To design and implement an MLP-based model for detecting code clones in source code repositories.
- To assess the model's performance using real-world datasets, focusing on its accuracy, precision, and scalability.
- To compare the proposed model with traditional clone detection methods and tools.

This study specifically addresses the detection of code clones using data mining and MLPs, excluding the bug-fixing process. The research will also aim to generalize the model across various programming languages and codebases to enhance its applicability.

Chapter 01 : Background

Introduction

This chapter serves as a foundational backdrop to the research area explored in this thesis, as well as a survey of previous studies conducted within this domain. As the primary focus of this thesis revolves around the identification and management of software clones, it will delve into several key aspects. These include defining code clones, examining the motivations behind cloning practices, assessing the impact of clones on software integrity, exploring various techniques for clone detection, tracing the evolution of clones over time, and discussing strategies for proficient clone management.

1.Software Engineering: [20]

1.1.What is Software Engineering?

Software engineering is the process of developing, testing and deploying computer applications to solve real-world problems by adhering to a set of engineering principles and best practices. The field of software engineering applies a disciplined and organized approach to software development with the stated goal of improving quality, time and budget efficiency, along with the assurance of structured testing and engineer Certification.

1.2Types of Software Engineering

Even though a software engineer usually manages many coding projects, software engineering entails more than just writing code for the software. In reality, software engineering encompasses every phase of the software development lifecycle (SDLC), from planning the budget to analysis, design, development, software testing, integration, quality and retirement.

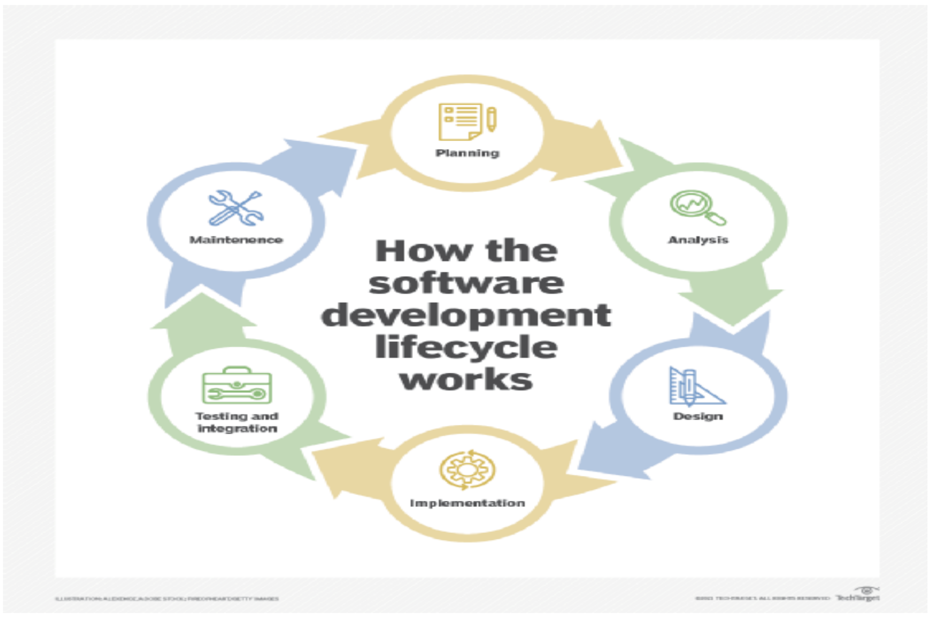


Figure 1 Most software engineering tasks can be broken into the following three categories[20]

1.2.1.Operational Software Engineering:

It includes all decisions and tasks pertaining to how the software will perform within a computer system. This may include anything related to the software budget, the way teams and users will interact with the software and any potential risks such as those associated with defective and outdated software.

1.2.2Transitional Software Engineering. This type of software engineering entails duties related to the software's adaptability and scalability when it's moved outside of its initial setting.

1.2.3Software engineering maintenance. It entails activities connected to enhancing and debugging current software to account for environmental changes, new technologies, bugs and risk factors that might have been disregarded during a previous development cycle. Over time, retirement takes over as maintenance of certain software is gradually reduced.

2.Software Maintenance: [6]

Software Maintenance refers to the process of modifying and updating a software system after it has been delivered to the customer. It is a critical part of the software development life cycle (SDLC) and is necessary to ensure that the software continues to meet the

2.1.What is Software Maintenance?

- Software maintenance is a continuous process that occurs throughout the entire life cycle of the software system.
- The goal of software maintenance is to keep the software system working correctly, needs of the users over time. This article focuses on discussing Software Maintenance in detail.efficiently, and securely, and to ensure that it continues to meet the needs of the users.
- This can include fixing bugs, adding new features, improving performance, or updating the software to work with new hardware or software systems.
- It is also important to consider the cost and effort required for software maintenance when planning and developing a software system.
- It is important to have a well-defined maintenance process in place, which includes testing and validation, version control, and communication with stakeholders.
- It’s important to note that software maintenance can be costly and complex, especially for large and complex systems. Therefore, the cost and effort of maintenance should be taken into account during the planning and development phases of a software project.
- It’s also important to have a clear and well-defined maintenance plan that includes regular maintenance activities, such as testing, backup, and bug fixing.

2.2 Several Key Aspects of Software Maintenance:

- **Bug Fixing:** The process of finding and fixing errors and problems in the software.
- **Enhancements:** The process of adding new features or improving existing features to meet the evolving needs of the users.

- **Performance Optimization:** The process of improving the speed, efficiency, and reliability of the software.
- **Porting and Migration:** The process of adapting the software to run on new hardware or software platforms.
- **Re-Engineering:** The process of improving the design and architecture of the software to make it more maintainable and scalable.
- **Documentation:** The process of creating, updating, and maintaining the documentation for the software, including user manuals, technical specifications, and design documents.

2.3.Several Types of Software Maintenance

- **Corrective Maintenance:** This involves fixing errors and bugs in the software system.
- **Patching:** It is an emergency fix implemented mainly due to pressure from management. Patching is done for corrective maintenance but it gives rise to unforeseen future errors due to lack of proper impact analysis.
- **Adaptive Maintenance:** This involves modifying the software system to adapt it to changes in the environment, such as changes in hardware or software, government policies, and business rules.
- **Perfective Maintenance:** This involves improving functionality, performance, and reliability, and restructuring the software system to improve changeability.
- **Preventive Maintenance:** This involves taking measures to prevent future problems, such as optimization, updating documentation, reviewing and testing the system, and implementing preventive measures such as backups.

Maintenance can be categorized into proactive and reactive types. Proactive maintenance involves taking preventive measures to avoid problems from occurring, while reactive maintenance involves addressing problems that have already occurred.

Maintenance can be performed by different stakeholders, including the original development team, an in-house maintenance team, or a third-party maintenance provider. Maintenance activities can be planned or unplanned. Planned activities include regular maintenance tasks that are scheduled in advance, such as updates and backups. Unplanned activities are reactive and are triggered by unexpected events, such as system crashes or security breaches. Software maintenance can involve modifying the software code, as well as its documentation, user manuals, and training materials. This ensures that the software is up-to-date and continues to meet the needs of its users.

Software maintenance can also involve upgrading the software to a new version or platform. This can be necessary to keep up with changes in technology and to ensure that the software remains compatible with other systems. The success of software maintenance depends on effective communication with stakeholders, including users, developers, and management. Regular updates and reports can help to keep stakeholders informed and involved in the maintenance process.

Software maintenance is also an important part of the Software Development Life Cycle(SDLC)**.** To update the software application and do all modifications in software application so as to improve performance is the main focus of software maintenance. Software is a model that runs on the basis of the real world. so, whenever any change requires in the software that means the need for real-world changes wherever possible.

2.3.Need for Maintenance

- Software Maintenance must be performed in order to:
- Correct faults.
- Improve the design.
- Implement enhancements.
- Interface with other systems.
- Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.
- Migrate legacy software.
- Retire software.
- Requirement of user changes.
- Run the code fast

2.4.Challenges in Software Maintenance

- The various challenges in software maintenance are given below:
- The popular age of any software program is taken into consideration up to ten to fifteen years. As software program renovation is open-ended and might maintain for decades making it very expensive.
- Older software programs, which had been intended to paint on sluggish machines with much less reminiscence and garage ability can not maintain themselves tough in opposition to newly coming more advantageous software programs on contemporary-day hardware.
- Changes are frequently left undocumented which can also additionally reason greater conflicts in the future.
- As the era advances, it turns into high prices to preserve vintage software programs.
- Often adjustments made can without problems harm the authentic shape of the software program, making it difficult for any next adjustments.

There is a lack of Code Comments.

- **Lack of documentation**: Poorly documented systems can make it difficult to understand how the system works, making it difficult to identify and fix problems.
- **Legacy code:** Maintaining older systems with outdated technologies can be difficult, as it may require specialized knowledge and skills.
- **Complexity:** Large and complex systems can be difficult to understand and modify, making it difficult to identify and fix problems.
- **Changing requirements:** As user requirements change over time, the software system may need to be modified to meet these new requirements, which can be difficult and time-consuming.
- **Interoperability issues:** Systems that need to work with other systems or software can be difficult to maintain, as changes to one system can affect the other systems.

- **Lack of test coverage**: Systems that have not been thoroughly tested can be difficult to maintain as it can be hard to identify and fix problems without knowing how the system behaves in different scenarios.
- **Lack of personnel**: A lack of personnel with the necessary skills and knowledge to maintain the system can make it difficult to keep the system up-to-date and running smoothly.
- **High-Cost:** The cost of maintenance can be high, especially for large and complex systems, which can be difficult to budget for and manage.
- To overcome these challenges, it is important to have a well-defined maintenance process in place, which includes testing and validation, version control, and communication with stakeholders. It is also important to have a clear and well-defined maintenance plan that includes regular maintenance activities, such as testing, backup, and bug fixing. Additionally, it is important to have personnel with the necessary skills and knowledge to maintain the system.

2.5.Categories of Software Maintenance [6]

- Maintenance can be divided into the following categories.
- **Corrective maintenance:** Corrective maintenance of a software product may be essential either to rectify some bugs observed while the system is in use, or to enhance the performance of the system.
- **Adaptive maintenance:** This includes modifications and updations when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware and software.
- **Perfective maintenance:** A software product needs maintenance to support the new features that the users want or to change different types of functionalities of the system according to the customer's demands.
- **Preventive maintenance:** This type of maintenance includes modifications and updations to prevent future problems with the software.

3.Bad Smell in Code : [11]

"Bad smells in code refer to certain characteristics or patterns within a codebase that indicate potential problems, such as inefficiencies, poor design, or maintainability issues. Identifying and addressing these bad smells is essential for improving the quality and maintainability of the code." One thing that most application developers and testers eventually encounter, especially when working with complex applications or across large teams, is code smells. These are tangible and observable indications that there is something wrong with an application's underlying code that could eventually lead to serious failures and kill an application's performance.

Typical examples of code smells include the following:

- duplicate code
- dead code
- long methods
- long parameter list

- comments
- unnecessary primitive variables
- data clumps

Particularly "smelly" code could be inefficient, nonperformant, complex, and difficult to change and maintain. While code smells may not always indicate a particularly serious problem, following them often leads to discoveries of decreased code quality, drains on application resources or even critical security vulnerabilities embedded within the application's code. At the least, it requires teams to perform some in-depth tests on the code -- and often reveals some critical areas in the code that need remedial work.

3.1.What Causes Code Smells [11]

Put simply, code smells are a result of poor or misguided programming. These blips in the application code can often be directly traced to mistakes made by the application programmer during the coding process. Typically, code smells stem from a failure to write the code in accordance with necessary standards. In other cases, it means that the documentation required to clearly define the project's development standards and expectations was incomplete, inaccurate or nonexistent.

There are many situations that can cause code smells, such as improper dependencies between modules, an incorrect assignment of methods to classes, or needless duplication of code segments. Code that is particularly smelly can eventually cause profound performance problems and make business-critical applications difficult to maintain.

Keep in mind, however, that a code smell is not an actual bug -- it's likely that the code still compiles and works as expected. Code smells are simply indications of potential breaches of code discipline and design principles. That said, it's possible that the source of a code smell may cause cascading issues and failures over time. It is also a good indicator that a code refactoring effort is in order.

3.2.The Known Smell Codes: [21]

3.2.1.Code Duplication: When the same or very similar code appears in multiple places within the codebase, it's a sign of code duplication. This can lead to maintenance issues, as any changes need to be made in multiple places, increasing the risk of introducing bugs.

3.2.2.Long Methods/Functions: Large, monolithic methods or functions that perform multiple tasks or have a lot of nested logic can be difficult to understand, test, and maintain. Breaking down these long methods into smaller, more focused ones can improve readability and maintainability.

3.2.3.Large Classes/Modules: Classes or modules that have too many responsibilities or contain too much code can become unwieldy and hard to work with. This violates the Single Responsibility Principle (SRP) and can make the code harder to understand and maintain. Splitting such classes or modules into smaller, more focused ones can help improve modularity and clarity.

3.2.4.Global Data:

The problem with global data is that it can be modified from anywhere in the code base, and there’s no mechanism to discover which bit of code touched it.

3.2.5.Mutable Data

Changes to data can often lead to unexpected consequences and tricky bugs. I can update some data here, not realizing that another part of the software expects something different and now fails a failure that’s particularly hard to spot if it only happens under rare conditions.

3.2.6.Divergent Change

We structure our software to make change easier; after all, software is meant to be soft. When we make a change, we want to be able to jump to a single clear point in the system and make the change.

3.2.7.Loops

Loops have been a core part of programming since the earliest languages. But we feel they are no more relevant today than bell-bottoms and flock wallpaper. We disdained them at the time of the first edition but Java, like most other languages at the time, didn’t provide a better alternative. These days, however, first-class functions are widely supported, so we can use Replace Loop with Pipeline

3.2.8.Lazy Element

Sometimes, it’s a class that used to pay its way, but has been downsized with refactoring. Either way, such program elements need to die with dignity.

3.2.9.Speculative Generality

You get it when people say, “Oh, I think we’ll need the ability to do this kind of thing someday” and thus add all sorts of hooks and special cases to handle things that aren’t required. The result is often harder to understand

4. Clones: [21]

The definition of a code clone is still more or less vague. Usually, it is described as portions of source code or code fragments at different locations in a software project/program that are identical or very similar. The similarity here may also be defined in various ways and can refer to textual, structural or semantic aspects of the source code. Selim et al. defined code clones as sets of syntactically or semantically similar code segments residing at different locations in the source code. In the words of Baxter et al. “Clones are segments of code that are similar according to some definition of similarity”. Symbolically, a clone fragment can be represented as a tuple with three variables (f, s, l), where f denotes the file location, s denotes start line of the clone fragment and l denotes the length of the fragment starting froms.

4.1.Clone Types :

Types of clones are defined based on the degree of similarity among the clone fragments. The following categorization of clone types has widely been accepted in the literature

Type-1 (Exact Clones): Identical code fragments without considering the variations in white-space and comments (Figure 1.1a).

```
int sum ( int num[], int len ) {

int total = 0;

for( int i=0; i < len; i++ ) {

 total += num[i];

}

 return total;

}

-----------------------------------------------------------

Code Fragment: A
```

```
int sum ( int num[], int len ) {

int total = 0;

for( int i=0; i < len; i++ ) {

total += num[i];

}

return total;

}

--------------------------------------------------

Code Fragment: B
```

(a) Type-1 Clone

```
int sum ( int num[], int len ) {

 int total = 0;

 for( int i=0; i < len; i++ ) {

 total += num[i];

}

 return total;

 }

------------------------------------------------------------------

Code Fragment: A
```

```
int sum ( int num[], int size ) {

int result = 0;

 for( int i=0; i < size; i++ ) {

 result += num[i];

}

 return result;

 }

 ---------------------------------

 Code Fragment: B
```

(b) Type-2 Clone

int sum (int num[], int len) {
int total = 0;
for(int i=0; i < len; i++) {
total += num[i];
}
return total;
}
--- 	
Code Fragment: A
	int sum (int num[], int len) {
	int total = 0;
	for(int i=0; i < len; i++) {
	total += num[i];
	}
	return total;
	}
	--- 	
	Code Fragment: A

int sum (int num[]) {
int result = 0;
int len = num.length;
for (int i = 0; i < len; i++) {
result += num[i];
}
return result;
}
---------------------------------- 	
Code Fragment: B
	int sum (int num[], int len) {
	int total = 0;
	int i = 0;
	while(i < len) {
	total += num[i++];
	}
	return total;
	}
	---------------------------------- 	
	Code Fragment: B

(c) Type-3 Clone	**Type-4 Clone**

Table 1 : Types of Code Clone

Type-2 (Renamed/Parameterized Clones) : Code fragments that are structurally/syntactically similar but may contain variations in identifiers, literals, types, layouts and comments (Figure 1.1b).

Type-3 (Gapped Clones): Code fragments with modifications in addition to those defined for Type-2 clones, such as insertion, deletion or modification of statements (Figure 1.1c). Type-2 and Type-3 clones are collectively termed as near-miss clones in literature .

Type-4 (Semantic Clones): Code fragments with the same functionality with or without being textually similar(Figure 1.1d).

4.2.Clone Granularity

The clone type definitions stated above are based on the notion of an arbitrary code segment. They do not define how much of contiguous code can be considered a clone. Contiguous portions of source code at different levels of granularity are used in the literature. The following are the most commonly used granularities, which yield the notion of source code clones:

File clone: Two source files containing some good amount of similar source code.

Class clone: Two classes of source code written in an object-oriented language when the classes have identical or near-identical code.

Function clone: Two functions are considered as clones when their bodies contain similar code to each other.

Block clone: When the contents of two blocks of code (usually a collection of statements performing a unit of work, marked bounded by some boundary marking character e.g., opening and closing braces, brackets or indentation, or the like) are similar enough.

Arbitrary statements clone: When two groups of statements at arbitrary regions of the source file are found to be similar enough, they are also regarded as clones (CCFinder detects such clones).

Structural clone: Structural clones denote the design level similarities among the patterns of interrelated classes emerging from design and analysis space at the architecture level.

Model based clone: Applications in some domain (e.g.: embedded systems, automobile/aviation design) are developed from a model designed with a domain specific modeling language. Unexpected overlaps and duplications in such models are termed as model based clones.

Researchers present clones in some groups that represent cloning relations among the related fragments. This grouping leads to a better understanding of the cloning status of the system. The following clone grouping is predominant in code clone literature.

Clone Pair: Two code fragments similar to each other form a Clone Pair.

Clone Class: A Clone Class is a group of clone fragments which are similar to each other. Therefore, a Clone Class may have two or more code fragments where each pair of code fragments forms a Clone Pair.

Super Clone: The set of Clone Classes that belong to the same source code location form a Super Clone, also known as Clone Class Family. Alternately, Super Clone is the aggregation of the Clone Classes that cross-cut in the same source code region, e.g.: file, directory, function, class or package.

4.3.Reasons for Clones in Software [21]

Most software systems usually contain a significant amount of cloned code and the amount of cloning varies depending on the domain and origin of the software system . However, clones do not occur in software systems by themselves, rather they evolve from various development activities performed by the developer during the software's development and maintenance phase. There are a number of factors that might force or influence the developers and/or maintenance engineers making cloned code in the system . Kapser et al. identified a set of eight cloning patterns that explain the motivations of cloning with corresponding advantages and disadvantages. Toomim et al. identified a set of cases that makes the abstraction costly and leads programmers to leave the cloned code instead.

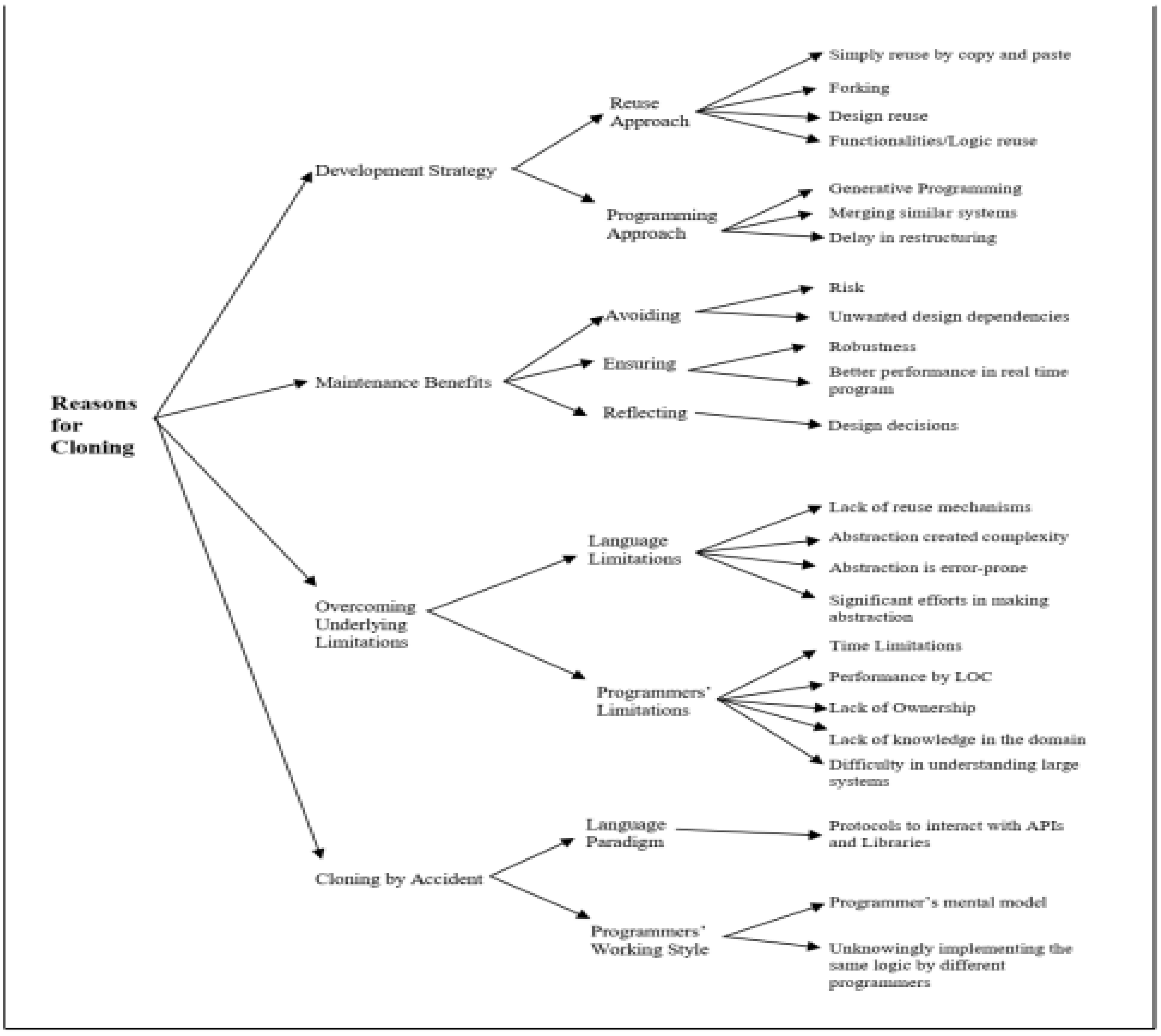

Reasons for Cloning
Development Strategy
Reuse Approach
Simply reuse by copy and paste
Forking
Design reuse
Functionalities/Logic reuse
Programming Approach
Generative Programming
Merging similar systems
Delay in restructuring
Maintenance Benefits
Avoiding
Risk
Unwanted design dependencies
Ensuring
Robustness
Better performance in real time program
Reflecting
Design decisions
Overcoming Underlying Limitations
Language Limitations
Lack of reuse mechanisms
Abstraction created complexity
Abstraction is error-prone
Significant efforts in making abstraction
Programmers' Limitations
Time Limitations
Performance by LOC
Lack of Ownership
Lack of knowledge in the domain
Difficulty in understanding large systems
Cloning by Accident
Language Paradigm
Protocols to interact with APIs and Libraries
Programmers' Working Style
Programmer's mental model
Unknowingly implementing the same logic by different programmers

Figure 2 :Reason for cloning[21]

A comprehensive list of factors (as shown in Figure 2.2) that introduce clones in software can be found in the survey by Roy and Cordy , where reasons for cloning are categorized as the following four groups:

Cloning by Accidents

Software developers may repeat common solution patterns for solving similar kinds of problems. Clones can also be created by developers because of implementing the same logic while working independently or following particular development restriction, e.g., coding under a programming language protocol or using a particular set of APIs.

Development Strategy

Different reuse and programming approaches may introduce clones in software systems. Clones can be created due to reuse of existing code, design, logic and functionality in the system. Developer's copy-paste programming practices is one of the most common forms of code cloning. Besides, merging of two software systems (of similar functionality) to produce a new one may introduce clones in the final system.

Maintenance Benefits

Developers may create or keep clones intentionally to obtain development and maintenance benefits, such as to cut the development cost off, to avoid the risk of developing new code that may introduce bugs or require additional testing to meet required QoS . Clones are sometime useful in speed-up development process or keeping software architecture clean and understandable.

Overcoming Underlying Limitations

Because of having some limitations on both programming languages and programmer's ability, clones might be introduced in the system . Some programming languages may not have sufficient abstraction mechanism or restriction for code reuse by design or convention. On the other hand, programmers may have a lack of knowledge of the existing system, strict time constraint, lack of code ownership, lack of understanding that may lead to the creation of duplicate code in the system.

4.4. Advantages of Code Clones

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

Risk in Writing Fresh Code:

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

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

• **Maintenance Speedup:** In a multi-cloned system, two cloned code fragments are independent of each other regarding both semantics and syntax and can evolve at a different pace in isolation without affecting one another and testing can also be performed and required to the modified fragments. Maintaining cloned fragments in a system may speed up maintenance, especially when automated regression tests are absent .

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

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

4.5. Disadvantages of Code Clones

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

• **Increased Probability of Defects:** If the original code contains a bug, then its clone will undoubtedly contain the same bug . Hence, duplication of the code may increase the probability of a bug in the system.

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

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

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

5.Clone Detection[21]

5.1.Anatomy of Code Clone Detection

Naively, a code clone detector should compare every possible fragment with every other fragment to identify the level of similarity. The similarity level needs to be at least up to a predefined value to accept the comparing fragments as clone to each other. However, in case of a medium to large scale system, such an exhaustive comparison is computationally expensive. Thus, several measures are used to reduce either the cost of individual comparison or the domain of comparison before performing the actual comparisons. A number of clone detection techniques has been proposed in literature over the past decades. Regardless of the difference in similarity detection mechanism, from a high level perspective, they mostly follow the same end-to-end processing workflow as shown in Figure 2.3. Considering raw source code as input, a typical code clone detector may perform the following steps (usually most of them if not all) to detect clones in the input source code.

5.1.1.Source Pre-processing

This is the very first step of a detection approach where various uninteresting parts (e.g., embedded source code of another language, auto generated source code, etc.) are filtered out from the input source. Then the remaining source code is partitioned into a set of disjoint fragments called source units. These are the largest source fragments that may form direct clone relations with each other. Source units can have different level of granularity, for example, files, classes, functions/methods, begin end blocks, statements, or sequences of source lines. Depending on the comparison technique used in the detection approach, source units may be further subdivided into smaller comparison units represented as lines or even tokens. Comparison units can also be derived from the syntactic structure of the source unit. Approaches like metrics-based does not require this partitioning of source since metrics values can be computed from source units regardless of their granularity

5.1.2.Source Transformation

In textual detection approach, similarity detection is performed among the comparison units. However, for non-textual approach, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. Additional normalizing transformations may also be performed by some approach in order to detect superficially different clones. These normalizations can vary from very simple normalizations, such as removal of whitespace and comments , to complex normalizations, involving source code transformations

5.1.3.Extraction

Extraction transforms the source code to the form suitable as input to the actual comparison algorithm. Depending on the tool, it typically involves one or more of the following steps. For tokenbased approaches, each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. CC Finder and Dup are the leading tools that use this tokenization approach on the source code. In syntactic approaches, the entire source codebase is parsed to build a parse tree or abstract syntax tree (AST). Then the comparison algorithms look for similar sub trees that are considered as clones . Metrics-based approaches may also use a parse tree representation to find clones based on metrics for sub trees . Some semantics-aware approaches generate program dependency graphs (PDGs) from the source code. To find clones, the techniques then look for isomorphic sub graphs .

5.1.4.Normalization

This optional step is intended to eliminate trivial differences in source comparison, such as, differences in whitespace, commenting, formatting, identifier naming, etc. Almost all approaches disregard comments and whitespace, although line-based approaches retain line breaks. However, some metrics-based approaches use formatting and layout as part of their comparison. Identifier normalization is applied in most of the approaches before comparison in order to identify parametric Type-2 clones. In such normalizations, usually one single identifier replaces all the identifiers in the source code. However, Baker uses an ordersensitive indexing scheme to normalize for detection of consistently renamed Type-2 clones. In text-based clone detection approaches, source codes may also pretty printed (white space formatting) to minimize the differences in source layout and spacing. Cordy et al. use an island grammar to generate a separate pretty- printed text file for each potentially cloned source unit. Some other transformations may be applied to change the structure of the code so that minor variants of the same syntactic form can be treated as similar.

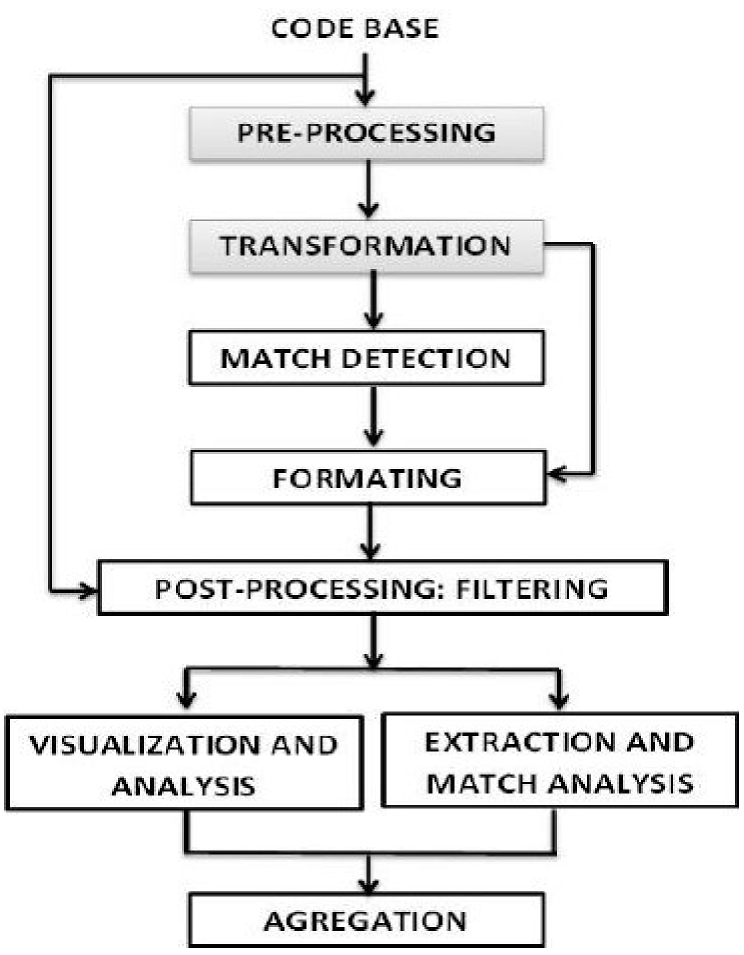

CODE BASE
PRE-PROCESSING
TRANSFORMATION
MATCH DETECTION
FORMATING
POST-PROCESSING: FILTERING
VISUALIZATION AND ANALYSIS
EXTRACTION AND MATCH ANALYSIS
AGREGATION

Figure 3 :Generic clone detection process[21]

Techniques ➡ Parameters ⬇	Text Based	Token Based	Abstract Syntax Tree (AST) Based	Program Dependency Graph (PDG) Based	Metric Based	Hybrid
Portability (Ability to run on multiple platforms)	High	Medium	Low	Low	Metrics dependent	Depends on type of technique used
Efficiency (Quality of results)	High	Low	High	High	High	High
Integrality(Level of difficulty involved in integrating the technique in current environment)	Low	High	Low	Medium	Medium	Medium
Transformation (Method of pre-processing)	Removes white space and comments	Tokens are generated from source code	AST is generated from source code	PDG is generated from source code	Metrics value are calculated after AST is generated from source code	Depends on the hybrid technique
Comparison Based (Type of input taken by clone detection technique)	Lines of code	Token	Nodes of tree	Nodes of program dependency graph	Metrics value	Depends on the hybrid technique
Computational Complexity(upper bound on time taken for clone detection)	Depends on algorithm	Linear	Quadratic	Quadratic	Linear	Depends on the hybrid technique
Refactoring Opportunities (Refactoring can be done easily or not)	Good for exact matches	Good	Good for refactoring as it finds syntactic clones	Good for refactoring	Manual inspection required	Depends on the hybrid technique
Representation (How source code will be represented)	Normalized source code	In the form of tokens	In the form of abstract syntax tree	In the form of program dependency graph	Set of metrics values	Depends on the hybrid technique
Language Dependency (Language targeted by clone detection technique)	Adaptable	It needs a laxer but no syntactic knowledge required	Parser required	Parser required	Parser required	Depends on the hybrid technique
Type of Clone Detected	Type 1,2 and 3	Type 1,2 and 3	Type 1,2,3 and 4	Type 1,2,3 and 4	Type 1,2,3 and 4	Depends on the hybrid technique
Output Type	Clone pairs and Clone Classes	Clone pairs and Clone Classes	Clone pairs, Clone Classes, AST nodes	Clone pairs and Clone Classes	Clone pairs, Clone Classes, metrics value	Depends on the hybrid technique

Table 2:Comparative Review of Clone Detection Techniques[21]

5.3.Clone Management[21]

The combination of various activities like clone classification, refactoring, visualization and tracking is known as clone management. In short, it includes all the activities that includes detecting, avoiding and removing the present clones. Clone management comes with a number of benefits like improvement in customer satisfaction and software system quality. It also helps developers during various activities like debugging and modification.

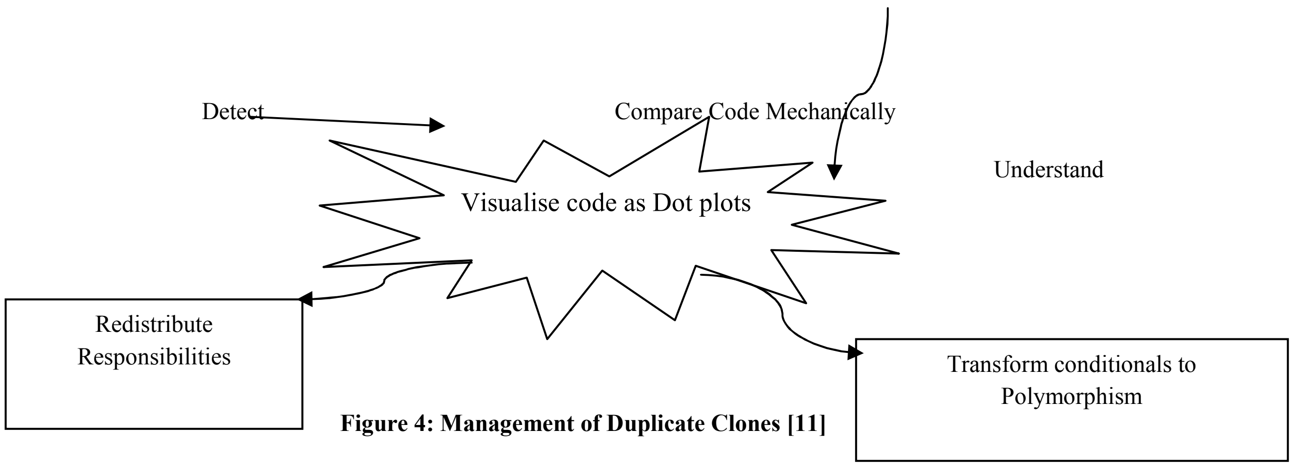
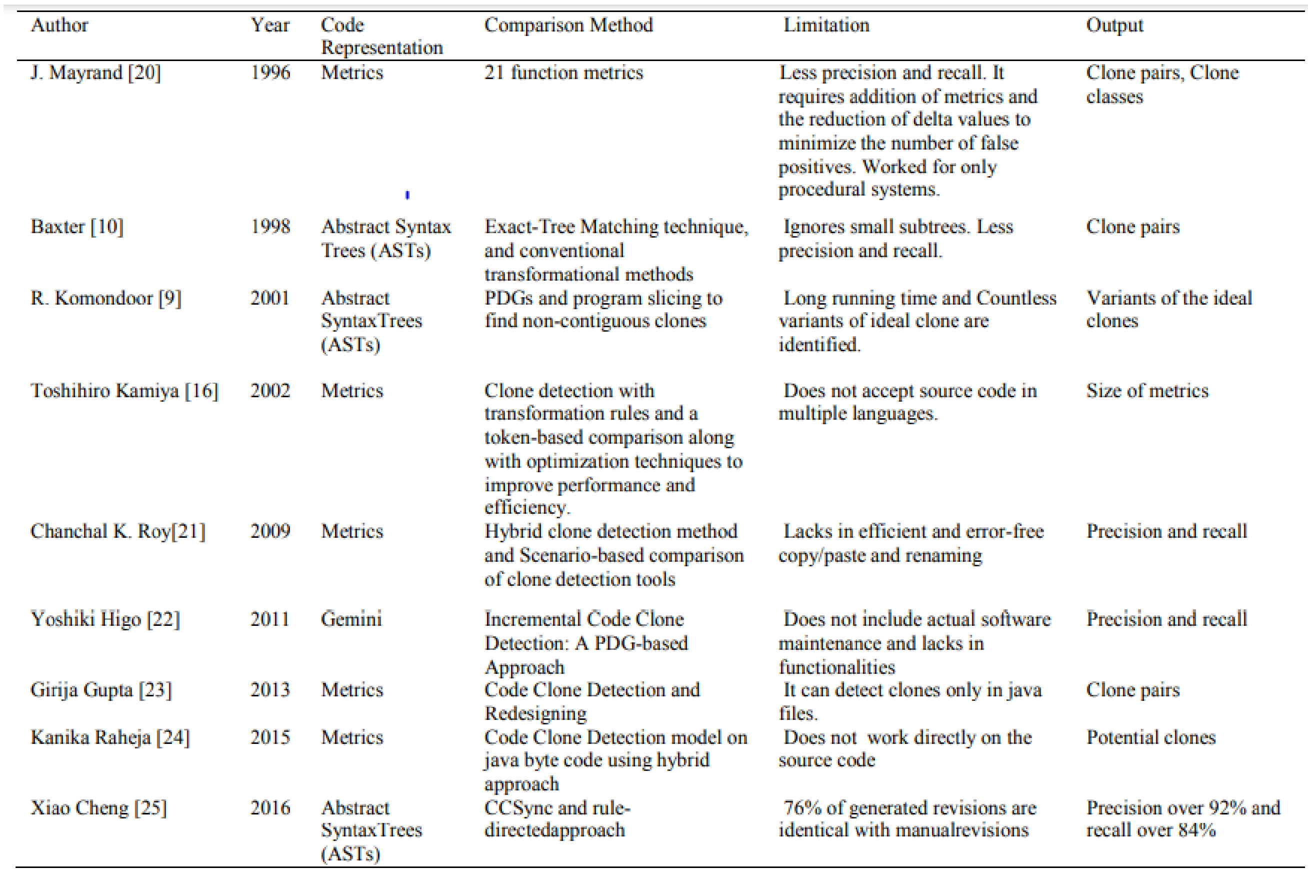

Detect
Compare Code Mechanically
Understand
Visualise code as Dot plots
Redistribute Responsibilities
Transform conditionals to Polymorphism

Figure 4: Management of Duplicate Clones [11]



Author	Year	Code Representation	Comparison Method	Limitation	Output
J. Mayrand [20]	1996	Metrics	21 function metrics	Less precision and recall. It requires addition of metrics and the reduction of delta values to minimize the number of false positives. Worked for only procedural systems.	Clone pairs, Clone classes
Baxter [10]	1998	Abstract Syntax Trees (ASTs)	Exact-Tree Matching technique, and conventional transformational methods	Ignores small subtrees. Less precision and recall.	Clone pairs
R. Komondoor [9]	2001	Abstract SyntaxTrees (ASTs)	PDGs and program slicing to find non-contiguous clones	Long running time and Countless variants of ideal clone are identified.	Variants of the ideal clones
Toshihiro Kamiya [16]	2002	Metrics	Clone detection with transformation rules and a token-based comparison along with optimization techniques to improve performance and efficiency.	Does not accept source code in multiple languages.	Size of metrics
Chanchal K. Roy[21]	2009	Metrics	Hybrid clone detection method and Scenario-based comparison of clone detection tools	Lacks in efficient and error-free copy/paste and renaming	Precision and recall
Yoshiki Higo [22]	2011	Gemini	Incremental Code Clone Detection: A PDG-based Approach	Does not include actual software maintenance and lacks in functionalities	Precision and recall
Girija Gupta [23]	2013	Metrics	Code Clone Detection and Redesigning	It can detect clones only in java files.	Clone pairs
Kanika Raheja [24]	2015	Metrics	Code Clone Detection model on java byte code using hybrid approach	Does not work directly on the source code	Potential clones
Xiao Cheng [25]	2016	Abstract SyntaxTrees (ASTs)	CCSync and rule-directedapproach	76% of generated revisions are identical with manualrevisions	Precision over 92% and recall over 84%

Figure 5: Related Research and Limitations [11]

In Clone Management, first step is to document the code segments and clone relationships occurring in those code segments. After that, there is a need to track the code base consistently during software development and incorporate the changes in documentation. Then clone documentation is analyzed properly using some visualization technique to remove or keep the clones [19]. In the next step refactoring is performed. If refactoring leads to change in program behavior, then it should be roll backed. Another way to manage clones is clone prevention by adopting minimal copy paste use.

5.4.Clone Detection and Management Process

There are some general steps which are required for clone detection and management. Those steps have been summarized in the form of pseudocode given in Table 4. For e.g. if we want to detect and manage code clones in two files α and β, then α and β will be supplied as input to the program for detecting and managing clones. The program will make use of some clone detection technique π for detecting the clones on the basis of threshold value Ω. Threshold value sets the lower limit on the degree of similarity between two fragments of code. If the detected degree of similarity i.e. £ between two fragments of code c1 and c2 is greater than or equal to Ω, then c1 and c2 will be considered as clone pair. Detected clone pair will be stored in the database DB. Similarly all the clone pairs will be detected and stored in the database DB. After that clone pairs will be categorized in to harmful and useful clones. Useful clones will be retained and harmful clones will be refactored or removed using some algorithm.

```
Function Clone Detection And Management (α, β, π, Ω, DB)
Inputs:
 • Two source code files to be compared i.e. α and β.
 •Detection Technique to be used i.e. π.
 •Threshold value for clone detection i.e. Ω
 •Database to store clones
Outputs:
 • Clone pairs c1 and c2
 • Clone detected in percentage i.e. £
 • Maintenance Overhead of clones
 • Rank wise Clones
 • Refactored source code file α and β
Step I: Input source code files α and β to clone detection and management system.
Step II: Use a Clone detection approach π for detecting clones in source code file α and β.
Step III: If (£>=Ω) then go to step 4 else go to step 8.
Step IV: Export the detected clones to database DB and update DB.
Step V: Find maintenance overhead of clones.
Step VI: Compare and rank clones according to maintenance overhead.
Step VII: Refactor or remove clones.
Step VIII: Stop
```

Table3: Pseudocode for Software Clone Detection and Management [11]

6.Machine learning : [11]

6.1.What is Machine Learning?

Machine Learning, often abbreviated as ML, is a subset of artificial intelligence (AI) that focuses on the development of computer algorithms that improve automatically through experience and by the use of data. In simpler terms, machine learning enables computers to learn from data and make decisions or predictions without being explicitly programmed to do so.

At its core, machine learning is all about creating and implementing algorithms that facilitate these decisions and predictions. These algorithms are designed to improve their performance over time, becoming more accurate and effective as they process more data.

In traditional programming, a computer follows a set of predefined instructions to perform a task. However, in machine learning, the computer is given a set of examples (data) and a task to perform, but it's up to the computer to figure out how to accomplish the task based on the examples it's given.

This ability to learn from data and improve over time makes machine learning incredibly powerful and versatile. It's the driving force behind many of the technological advancements we see today, from voice assistants and recommendation systems to self-driving cars and predictive analytics.

6.2.The Importance of Machine Learning: [12]

In the 21st century, data is the new oil, and machine learning is the engine that powers this data-driven world. It is a critical technology in today's digital age, and its importance cannot be overstated. This is reflected in the industry's projected growth, with the US Bureau of Labor Statistics predicting a 21% growth in jobs between 2021 and 2031.

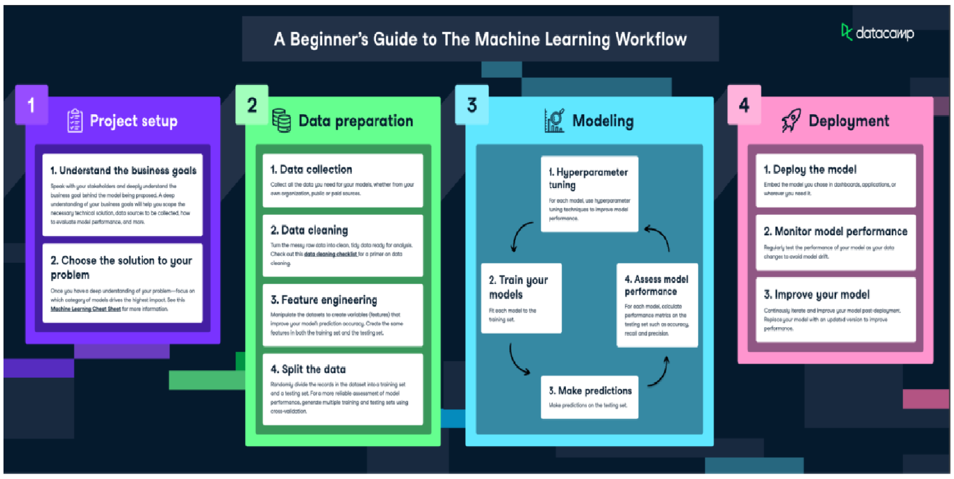
Here are some reasons why it’s so essential in the modern world:

- **Data processing.** One of the primary reasons machine learning is so important is its ability to handle and make sense of large volumes of data. With the explosion of digital data from social media, sensors, and other sources, traditional data analysis methods have become inadequate. Machine learning algorithms can process these vast amounts of data, uncover hidden patterns, and provide valuable insights that can drive decision-making.
- **Driving innovation.** Machine learning is driving innovation and efficiency across various sectors. Here are a few examples:
- **Healthcare**. Algorithms are used to predict disease outbreaks, personalize patient treatment plans, and improve medical imaging accuracy.
- **Finance**. Machine learning is used for credit scoring, algorithmic trading, and fraud detection.
- **Retail**. Recommendation systems, supply chains, and customer service can all benefit from machine learning.
- The techniques used also find applications in sectors as diverse as agriculture, education, and entertainment.
- **Enabling automation**. Machine learning is a key enabler of automation. By learning from data and improving over time, machine learning algorithms can perform previously manual tasks, freeing humans to focus on more complex and creative tasks. This not only increases efficiency but also opens up new possibilities for innovation.

6.3.How Does Machine Learning Work? [2]

Understanding how machine learning works involves delving into a step-by-step process that transforms raw data into valuable insights.

a



1. Understand the business goals
Speak with your stakeholders and deeply understand the business goal behind the model being proposed. A deep understanding of your business goals will help you scope the necessary technical solution, data sources to be collected, how to evaluate model performance, and more.

2. Choose the solution to your problem
Once you have a deep understanding of your problem—focus on which category of models drives the highest impact. See this Machine Learning Cheat Sheet for more information.

1. Data collection
Collect all the data you need for your models, whether from your own organization, public or paid sources.

2. Data cleaning
Turn the messy raw data into clean, tidy data ready for analysis. Check out this data cleaning checklist for a primer on data cleaning.

3. Feature engineering
Manipulate the datasets to create variables (features) that improve your model's prediction accuracy. Create the same features in both the training set and the testing set.

4. Split the data
Randomly divide the records in the dataset into a training set and a testing set. For a more reliable assessment of model performance, generate multiple training and testing sets using cross-validation.

1. Hyperparameter tuning
For each model, use hyperparameter tuning techniques to improve model performance.

2. Train your models
Fit each model to the training set.

3. Make predictions
Make predictions on the testing set.

4. Assess model performance
For each model, calculate performance metrics on the testing set such as accuracy, recall and precision.

1. Deploy the model
Embed the model you chose in dashboards, applications, or wherever you need it.

2. Monitor model performance
Regularly test the performance of your model as your data changes to avoid model drift.

3. Improve your model
Continously iterate and improve your model post-deployment. Replace your model with an updated version to improve performance.

Figure 6 a beginner’s guide to the machine learning workflow(DataCamp, 2022)[2]

Step 1: Data collection

The first step in the machine learning process is data collection. Data is the lifeblood of machine learning - the quality and quantity of your data can directly impact your model's performance. Data can be collected from various sources such as databases, text files, images, audio files, or even scraped from the web.

Once collected, the data needs to be prepared for machine learning. This process involves organizing the data in a suitable format, such as a CSV file or a database, and ensuring that the data is relevant to the problem you're trying to solve.

Step 2: Data preprocessing

Data preprocessing is a crucial step in the machine learning process. It involves cleaning the data (removing duplicates, correcting errors), handling missing data (either by removing it or filling it in), and normalizing the data (scaling the data to a standard format).

Preprocessing improves the quality of your data and ensures that your machine learning model can interpret it correctly. This step can significantly improve the accuracy of your model.

Step 3: Choosing the right model

Once the data is prepared, the next step is to choose a machine learning model. There are many types of models to choose from, including linear regression, decision trees, and neural networks. The choice of model depends on the nature of your data and the problem you're trying to solve.

Factors to consider when choosing a model include the size and type of your data, the complexity of the problem, and the computational resources available.

Step 4: Training the model

After choosing a model, the next step is to train it using the prepared data. Training involves feeding the data into the model and allowing it to adjust its internal parameters to better predict the output.

During training, it's important to avoid overfitting (where the model performs well on the training data but poorly on new data) and underfitting (where the model performs poorly on both the training data and new data).

Step 5: Evaluating the model

Once the model is trained, it's important to evaluate its performance before deploying it. This involves testing the model on new data it hasn't seen during training.

Common metrics for evaluating a model's performance include accuracy (for classification problems), precision and recall (for binary classification problems), and mean squared error (for regression problems).

Step 6: Hyperparameter tuning and optimization

After evaluating the model, you may need to adjust its hyperparameters to improve its performance. This process is known as parameter tuning or hyperparameter optimization.

Techniques for hyperparameter tuning include grid search (where you try out different combinations of parameters) and cross validation (where you divide your data into subsets and train your model on each subset to ensure it performs well on different data).

Step 7: Predictions and deployment

Once the model is trained and optimized, it's ready to make predictions on new data. This process involves feeding new data into the model and using the model's output for decision-making or further analysis.

Deploying the model involves integrating it into a production environment where it can process real-world data and provide real-time insights. This process is often known as MLOps.

6.4.Types of Machine Learning: [12]

Machine learning can be broadly classified into three types based on the nature of the learning system and the data available: supervised learning, unsupervised learning, and reinforcement learning. Let's delve into each of these:

6.4.1.Supervised Learning

Supervised learning is the most common type of machine learning. In this approach, the model is trained on a labeled dataset. In other words, the data is accompanied by a label that the model is trying to predict. This could be anything from a category label to a real-valued number.

The model learns a mapping between the input (features) and the output (label) during the training process. Once trained, the model can predict the output for new, unseen data.

6.4.2.Unsupervised Learning

Unsupervised learning, on the other hand, involves training the model on an unlabeled dataset. The model is left to find patterns and relationships in the data on its own.

This type of learning is often used for clustering and dimensionality reduction. Clustering involves grouping similar data points together, while dimensionality reduction involves reducing the number of random variables under consideration by obtaining a set of principal variables.

Common examples of unsupervised learning algorithms include k-means for clustering problems and Principal Component Analysis (PCA) for dimensionality reduction problems. Again, in practical terms, in the field of marketing, unsupervised learning is often used to segment a company's customer base. By examining purchasing patterns, demographic data, and other information, the algorithm can group customers into segments that exhibit similar behaviors without any pre-existing labels.

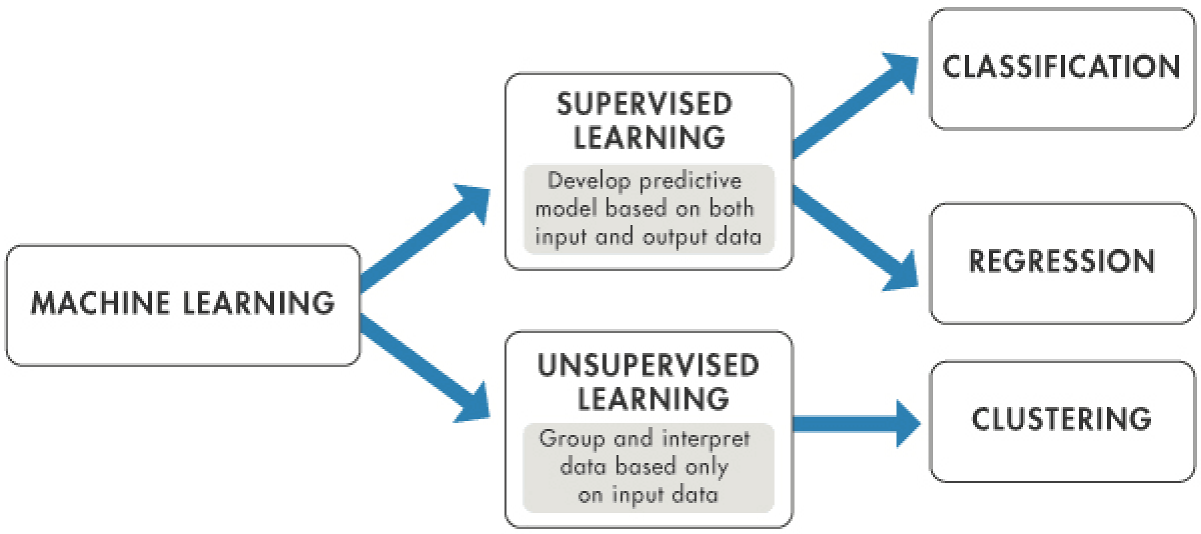

MACHINE LEARNING
SUPERVISED LEARNING
Develop predictive model based on both input and output data
UNSUPERVISED LEARNING
Group and interpret data based only on input data
CLASSIFICATION
REGRESSION
CLUSTERING

Figure 7: Comparing supervised and unsupervised learning[12]

Understanding the Impact of Machine Learning

Machine Learning has had a transformative impact across various industries, revolutionizing traditional processes and paving the way for innovation. Let's explore some of these impacts:

"Machine learning is the most transformative technology of our time. It's going to transform every single vertical."

- Satya Nadella, CEO at Microsoft

Healthcare

In healthcare, machine learning is used to predict disease outbreaks, personalize patient treatment plans, and improve medical imaging accuracy. For instance, Google's DeepMind Health is working with doctors to build machine learning models to detect diseases earlier and improve patient care.

Finance

The finance sector has also greatly benefited from machine learning. It's used for credit scoring, algorithmic trading, and fraud detection. A recent survey found that 56% of global executives said that artificial intelligence (AI) and machine learning have been implemented into financial crime compliance programs.

Transportation

Machine learning is at the heart of the self-driving car revolution. Companies like Tesla and Waymo use machine learning algorithms to interpret sensor data in real-time, allowing their vehicles to recognize objects, make decisions, and navigate roads autonomously. Similarly, the Swedish Transport Administration recently started working with computer vision and machine learning specialists to optimize the country's road infrastructure management.

6.5.Machine Learning vs AI vs Deep Learning[12]

Machine learning is often confused with artificial intelligence or deep learning. Let's take a look at how these terms differ from one another. For a more in-depth look, check out our comparison guides on AI vs machine learning and machine learning vs deep learning.

AI refers to the development of programs that behave intelligently and mimic human intelligence through a set of algorithms. The field focuses on three skills: learning, reasoning, and self-correction to obtain maximum efficiency. AI can refer to either machine learning-based programs or even explicitly programmed computer programs.

6.5.1.Machine Learning

is a subset of AI, which uses algorithms that learn from data to make predictions. These predictions can be generated through supervised learning, where algorithms learn patterns from existing data, or unsupervised learning, where they discover general patterns in data. ML models can predict numerical values based on historical data, categorize events as true or false, and cluster data points based on commonalities.

6.5.2.Deep Learning

on the other hand, is a subfield of machine learning dealing with algorithms based essentially on multi-layered artificial neural networks (ANN) that are inspired by the structure of the human brain.

Unlike conventional machine learning algorithms, deep learning algorithms are less linear, more complex, and hierarchical, capable of learning from enormous amounts of data, and able to produce highly accurate results. Language translation, image recognition, and personalized medicines are some examples of deep learning applications.

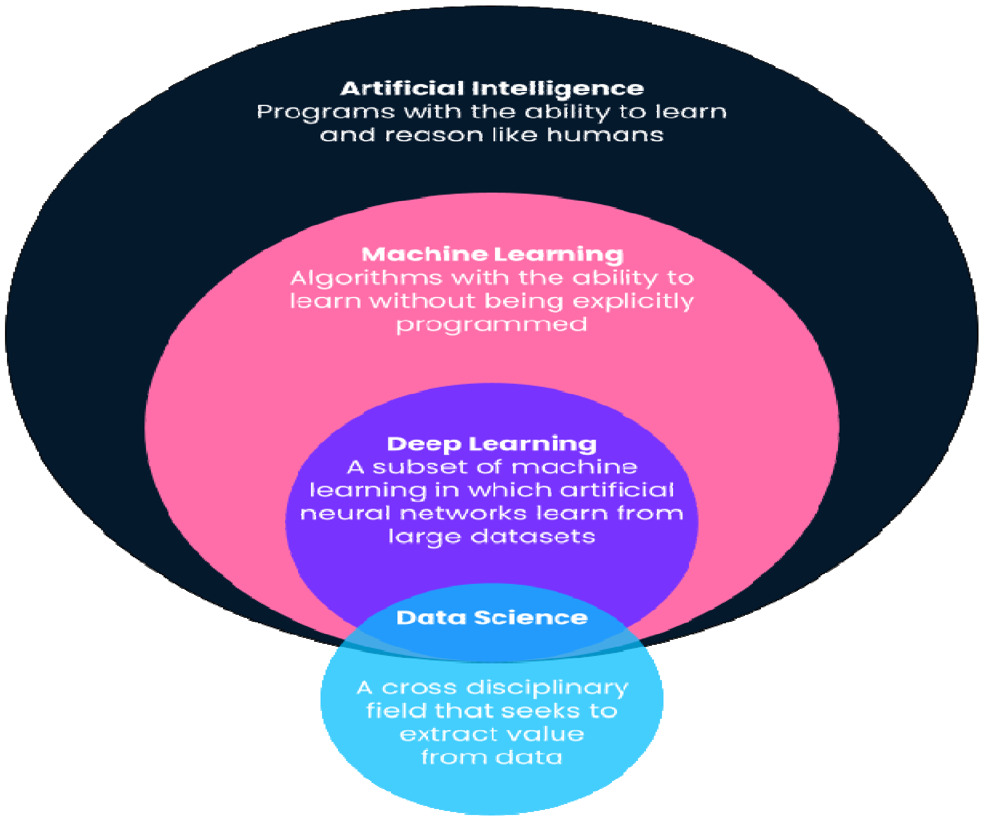

Artificial Intelligence
Programs with the ability to learn and reason like humans
Machine Learning
Algorithms with the ability to learn without being explicitly programmed
Deep Learning
A subset of machine learning in which artificial neural networks learn from large datasets
Data Science
A cross disciplinary field that seeks to extract value from data

Figure 8 :Comparing different industry terms[12]

7.Deep learning : [9]

Deep Learning is a branch of machine learning that uses neural networks to teach computers to do what seems natural to humans: learn from examples. In Deep Learning, a model learns to perform classification or regression tasks directly from data such as images, text or sound. Deep Learning models can achieve remarkable precision, often exceeding human performance.

7.1.How does Deep Learning work?

Deep Learning models are based on neural network architectures. A network of neurons, inspired by that of a human brain, is made up of nodes or interconnected neurons in a layer infrastructure which connect the entries to the desired exits. The neurons located between the entry and exit layers of a network of neurons are called hidden layers. The term "Deep" generally refers to the number of layers hidden in the network of neurons. Deep Learning models can include hundreds, or even thousands of hidden layers.

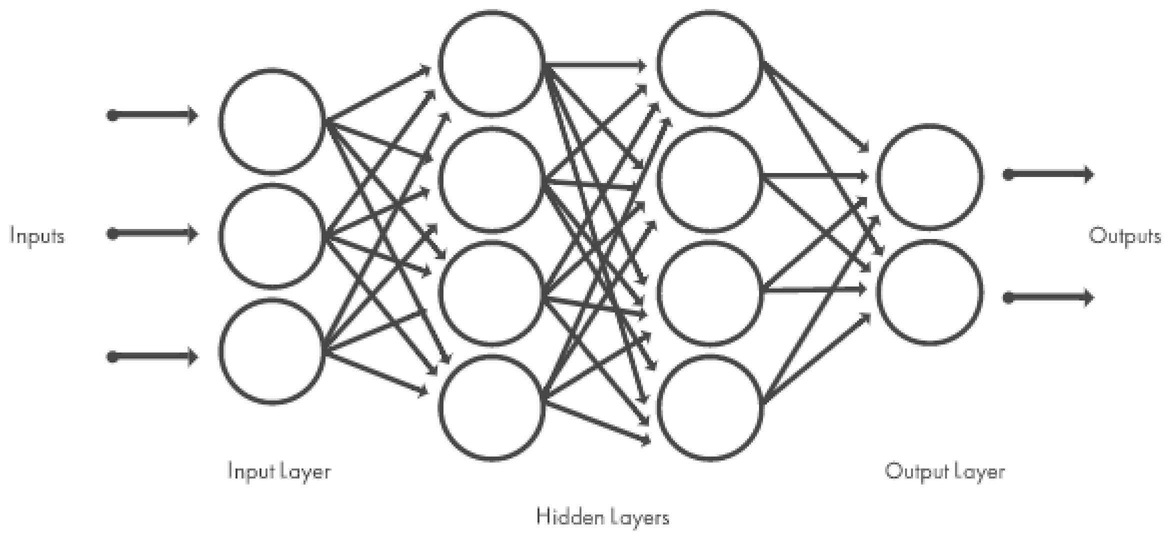


Figure 9 :Representation of the architecture of a typical network of neurons. [9]

Deep Learning models are drawn using large labeled databases and can often learn characteristics, directly from data, without it being necessary to extract them manually. While the first network of artificial neurons was theorized in 1958, Deep Learning requires significant calculation power which was not available before the 2000s. Today, researchers have access to IT resources that allow you to build and lead to networks with hundreds of neurons and connections.

High performance GPUs have an effective parallel architecture for Deep Learning. Associated with clusters or cloud computing, they allow development teams to reduce the learning time of a Deep Learning network, from several weeks to a few hours, or even less.

7.2.Types of Deep Learning models

Deep Learning models are able to automatically learn features from the data, which makes them well-suited for tasks such as image recognition, speech recognition, and natural language processing. The most widely used architectures in deep learning are feedforward neural networks, convolutional neural networks (CNNs), and recurrent neural networks (RNNs).

7.2.1.Feedforward Neural Networks (FNNs): are the simplest type of ANN, with a linear flow of information through the network. FNNs have been widely used for tasks such as image classification, speech recognition, and natural language processing.

7.2.2.Convolutional Neural Networks (CNNs) : are specifically for image and video recognition tasks. CNNs are able to automatically learn features from the images, which makes them well-suited for tasks such as image classification, object detection, and image segmentation.

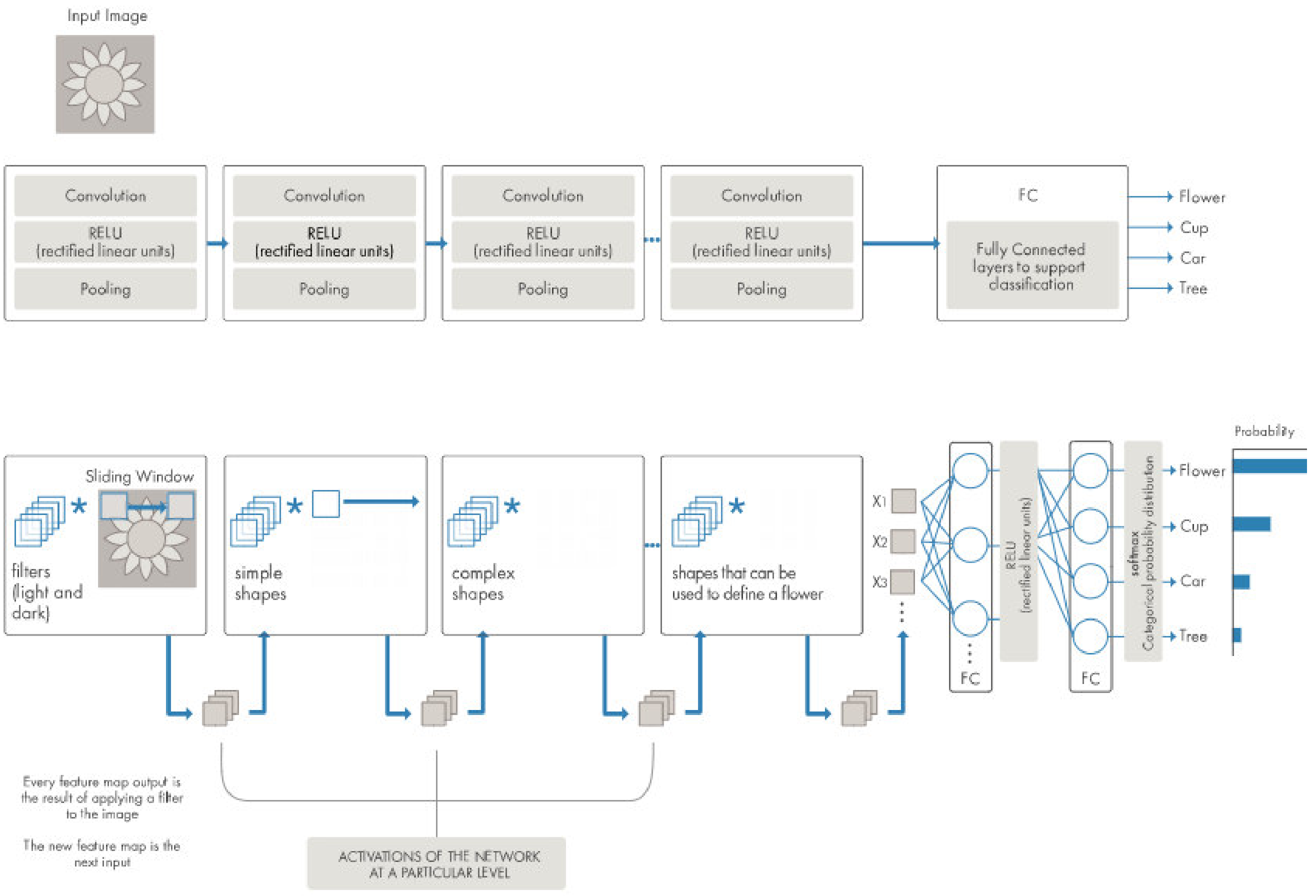

Input Image
Convolution
RELU (rectified linear units)
Pooling
FC
Fully Connected layers to support classification
Flower
Cup
Car
Tree
Sliding Window
filters (light and dark)
simple shapes
complex shapes
shapes that can be used to define a flower
X1
X2
X3
RELU (rectified linear units)
softmax Categorical probability distribution
Probability
Every feature map output is the result of applying a filter to the image
The new feature map is the next input
ACTIVATIONS OF THE NETWORK AT A PARTICULAR LEVEL

Figure 10: Visualization of an example of a network of neurons with convolution[9]

7.2.3.Recurrent Neural Networks (RNNs):

are a type of neural network that is able to process sequential data, such as time series and natural language. RNNs are able to maintain an internal state that captures information about the previous inputs, which makes them well-suited for tasks such as speech recognition, natural language processing, and language translation.

7.3.How to create Deep Learning models ?

You can create a Deep Learning model starting from zero or start with a pre-trained Deep Learning model, which you can apply or adapt to your task.

Learning from zero: to lead to a Deep Learning model starting from zero, you must bring together a large labeled dataset and design a network architecture that will learn the characteristics and the model. This is a good approach for new or specific applications, or more generally for applications for which there are no pre -existing models. The main drawback of this approach is that it requires a large database (with the annotated truth) and that the learning time can vary from a few hours to a few weeks, depending on your task and your resources IT.

Transfer learning:

In Deep Learning applications such as classification of images, computer vision, audio processing or natural language processing, the approach to transfer learning is commonly used. It is a question of refining a model of pre-trained Deep Learning. You start from an existing model, such as Squeezenet or Googlenet for the classification of images, and you introduce new data containing new categories. After making some adjustments to the network, you can then perform a new task, such as categorizing only dogs or cats, instead of 1,000 different objects. This method also has the advantage of requesting much less data, which considerably reduces learning time.

A pre-trained Deep Learning model can also be used as a characteristics extractor. You can use layers activations as a characteristics to train another model learning machine (such as a support vector machine (SVM)). You can also use the pre-trained model as a basic block for another Deep Learning model. For example, you can use a classification CNN images as a characteristics extractor for an object detector.

7.4.Why choose Deep Learning rather than machine learning?

In a word, precision. Deep Learning generally provides higher precision and provides greater automation of the extended workflow than machine learning. The main drawbacks of Deep Learning models are due to their increased complexity and the large size of the learning data games required, making these models longer to train. There are methods to overcome, or at least alleviate, the effect of these drawbacks.

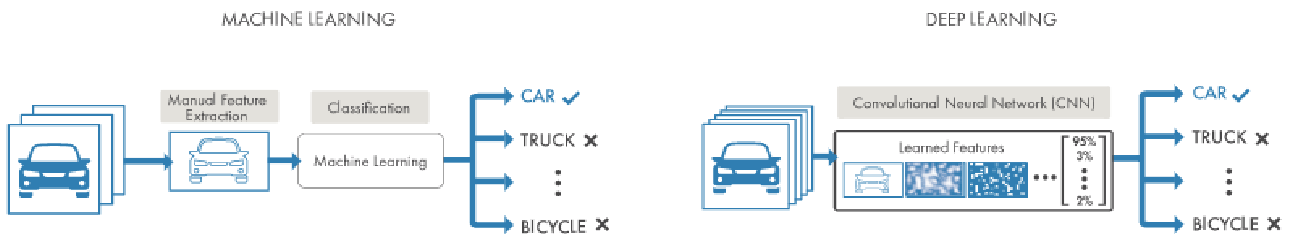


Figure 11 :Comparison between a machine learning approach (left) and a Deep Learning approach (right) in an example of vehicle classification[9]

7.5.Why is Deep Learning Important?

Deep Learning is an essential technology on which driver -free cars rely, allowing to recognize a stop panel or distinguish a pedestrian from a lamp. It is also a key element in the vocal control of consumer equipment such as phones, tablets, televisions or portable speakers. Deep Learning has aroused a lot of interest in recent times, for good reasons. Thanks to the Deep Learning, computers and systems can perform complex tasks with more precision and automation.

7.6.Applications of Deep Learning :

The main applications of deep learning can be divided into computer vision, natural language processing (NLP), and reinforcement learning.

Computer vision

In computer vision, Deep learning models can enable machines to identify and understand visual data. Some of the main applications of deep learning in computer vision include:

- **Object detection and recognition:** Deep learning model can be used to identify and locate objects within images and videos, making it possible for machines to perform tasks such as self-driving cars, surveillance, and robotics.
- **Image classification:** Deep learning models can be used to classify images into categories such as animals, plants, and buildings. This is used in applications such as medical imaging, quality control, and image retrieval.
- **Image segmentation:** Deep learning models can be used for image segmentation into different regions, making it possible to identify specific features within images.

Natural language processing (NLP):

In NLP, the Deep learning model can enable machines to understand and generate human language. Some of the main applications of deep learning in NLP include:

- **Automatic Text Generation** – Deep learning model can learn the corpus of text and new text like summaries, essays can be automatically generated using these trained models.
- **Language translation:** Deep learning models can translate text from one language to another, making it possible to communicate with people from different linguistic backgrounds.
- **Sentiment analysis:** Deep learning models can analyze the sentiment of a piece of text, making it possible to determine whether the text is positive, negative, or neutral. This is used in applications such as customer service, social media monitoring, and political analysis.
- **Speech recognition:** Deep learning models can recognize and transcribe spoken words, making it possible to perform tasks such as speech-to-text conversion, voice search, and voice-controlled devices.

Reinforcement learning:

In reinforcement learning, deep learning works as training agents to take action in an environment to maximize a reward. Some of the main applications of deep learning in reinforcement learning include:

- **Game playing:** Deep reinforcement learning models have been able to beat human experts at games such as Go, Chess, and Atari.
- **Robotics:** Deep reinforcement learning models can be used to train robots to perform complex tasks such as grasping objects, navigation, and manipulation.
- **Control systems:** Deep reinforcement learning models can be used to control complex systems such as power grids, traffic management, and supply chain optimization.

8. Multilayer Perceptron Neural Network(MLP) [17]

8.1. What are Multilayer Perceptrons (MLPs)?

A multilayer perceptron (MLP) Neural network belongs to the feedforward neural network. It is an Artificial Neural Network in which all nodes are interconnected with nodes of different layers.

Frank Rosenblatt first defined the word Perceptron in his perceptron program. Perceptron is a basic unit of an artificial neural network that defines the artificial neuron in the neural network. It is a supervised learning algorithm containing nodes' values, activation functions, inputs, and weights to calculate the output.

The Multilayer Perceptron (MLP) Neural Network works only in the forward direction. All nodes are fully connected to the network. Each node passes its value to the coming node only in the forward direction. The MLP neural network uses a Backpropagation algorithm to increase the accuracy of the training model.

8.2. How do Multilayer Perceptrons work? [17]

MLPs follow a simple process: data enters through the input layer, passes through hidden layers where each neuron applies weights and biases to the inputs, and results in a final prediction at the output layer. The training process involves adjusting weights and biases using backpropagation, where the model minimizes the error by adjusting these parameters based on the gradient of the loss function (like cross-entropy or mean squared error) using an optimization algorithm such as stochastic gradient descent (SGD) or Adam.

8.2.1. Layers in an MLP[17]

Input Layer

It is the initial or starting layer of the Multilayer perceptron. It takes input from the training data set and forwards it to the hidden layer. There are n input nodes in the input layer. The number of input nodes depends on the number of dataset features. Each input vector variable is distributed to each of the nodes of the hidden layer.

Hidden Layer

It is the heart of all Artificial neural networks. This layer comprises all computations of the neural network. The edges of the hidden layer have weights multiplied by the node values. This layer uses the activation function.

There can be one or two hidden layers in the model.

Several hidden layer nodes should be accurate as few nodes in the hidden layer make the model unable to work efficiently with complex data. More nodes will result in an overfitting problem.

Output Layer

This layer gives the estimated output of the Neural Network. The number of nodes in the output layer depends on the type of problem. For a single targeted variable, use one node. N classification problem, ANN uses N nodes in the output layer.

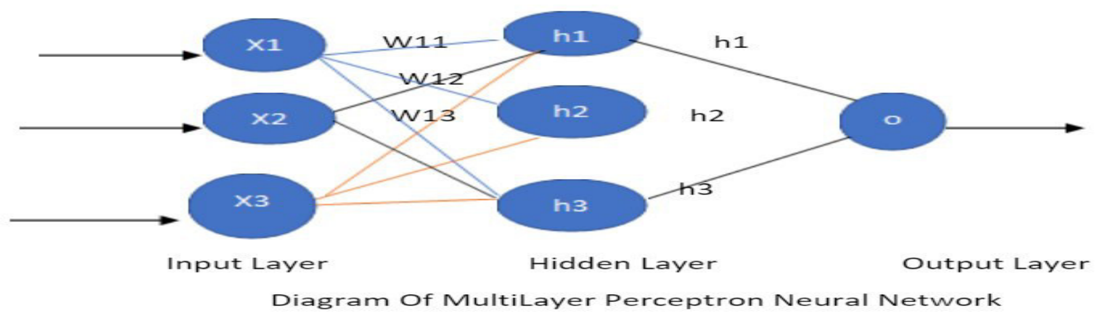


Figure 12:diagrame of multiplater perception neural network[18]

8.2.2. Fully Connected Layers

MLPs are sometimes referred to as fully connected neural networks because each neuron in one layer is connected to every neuron in the next. This dense connection enables the MLP to learn intricate relationships between features, but it also means MLPs can be computationally expensive, especially with large datasets and many layers.

8.3. Applications of Multilayer Perceptrons

MLPs are versatile and can be used for various types of machine learning problems, including:

- **Classification tasks:** For instance, predicting whether an email is spam or not.
- **Regression tasks:** MLPs can predict continuous values such as housing prices or stock market trends.
- **Time series prediction:** MLPs are also used to forecast future values based on historical data, though other architectures like Recurrent Neural Networks (RNNs) are often preferred for sequential data.

9.Conclusion :

In this chapter, we introduced the foundational concepts underlying our research, beginning with software engineering and its critical role in software maintenance. We then discussed the concept of bad smells in code, with a particular focus on software clones, including their types, advantages, and disadvantages. The chapter also provided an overview of machine learning, followed by a detailed introduction to deep learning, specifically Neural Networks (MLP), as a key approach for software clone detection using data mining techniques.

The next chapter will delve into the state of the art, exploring the methods proposed for clone detection.

Chapter 02 : State of the art

1.Introduction:

Software clone detection has been a crucial area of research in software engineering for several decades. Code clones, which are duplicated or similar code fragments, often result from code reuse, copy-paste practices, or even unintentional similarities across different parts of a system. Detecting and managing these clones is vital for improving software quality, maintainability, and bug detection. Over the years, researchers have developed a variety of techniques for detecting code clones, ranging from simple textual comparisons to advanced machine learning models. In this chapter, we present an overview of the state-of-the-art techniques in software clone detection, focusing on their strengths, limitations, and applications.

2.Types of Code Clones: [6]

Before diving into the various detection techniques, it is essential to understand the different types of code clones. The research community typically classifies clones into four types:

- **Type-1 (Exact Clones)**: Code fragments that are identical, except for variations in white spaces, comments, and formatting.
- **Type-2 (Renamed Clones)**: Code fragments that are identical except for changes in identifiers, such as variable names, and types.
- **Type-3 (Modified Clones)**: Code fragments that have some added, modified, or deleted statements but retain the same basic structure.
- **Type-4 (Semantic Clones)**: Code fragments that perform the same function but have different implementations or structures.

Each type of clone presents different challenges for detection methods, with Type-1 being the easiest to detect and Type-4 the most difficult due to its focus on semantic behavior rather than syntax.

3.Approaches For Code Clone Detection :

For the detection of code clones, several research works have been carried out. Here are some key examples:

3.1.Baxter et al - AST-Based Clone Detection : [1]

3.1.1. Definition :

Baxter et al. (1998) - AST-Based Clone Detection Is introduced a novel approach to clone detection based on Abstract Syntax Trees (ASTs). The authors present a detailed analysis of the advantages of this method over traditional techniques and demonstrate its effectiveness through empirical evaluation.

AST-Based clone detection involves analysing the structural similarities between code snippets by converting them into Abstract Syntax Trees. These trees represent the hierarchical structure of the code, capturing the relationships between different code elements like variables, functions, and statements.

Structural Analysis

ASTs allow for a structural comparison of code, identifying similarities beyond mere lexical matching.

Code Understanding

ASTs provide a representation of the code's underlying logic and structure, enabling a deeper understanding of the relationships between code elements.

Semantic Recognition

By focusing on the structure, AST-based techniques can identify clones even when the code is syntactically different but semantically equivalent.

3.1.2.Advantages of AST-Based Clone Detection

This approach offers significant advantages over traditional methods such as lexical matching or string-based comparisons.

- **Increased Accuracy**

AST-based techniques are more precise in identifying clones, even when the code is heavily refactored or re-written.

- **Semantic Awareness**

Unlike lexical methods, ASTs capture the underlying meaning and intent of the code, leading to more meaningful clone detection.

- **Reduced False Positives**

By focusing on the structural similarities, AST-based methods minimise the number of false positive detections, ensuring greater reliability.

3.1.3.Evaluating Algorithm Performance:

The authors thoroughly evaluated the algorithm's performance using a variety of codebases and datasets. The results demonstrate the algorithm's effectiveness in accurately detecting clones while minimising false positives.

Metric	**Value**
Accuracy	98%
Precision	95%
Recall	92%

Tableau 4: evaluating algorithme performance

3.1.4.Key Findings and Observations

AST-based clone detection significantly outperformed traditional methods in terms of accuracy and effectiveness. It was particularly successful in identifying more complex clones that were previously undetected.

- **Enhanced Accuracy**

AST-based detection resulted in a higher percentage of correctly identified clones.

- **Improved Recall**

The algorithm successfully identified a greater number of clones compared to other methods.

- **Reduced False Positives**

The algorithm minimised the number of incorrectly identified clones, improving its overall reliability.

3.2.Token-Based Clone Detection (CCFinder): [8]

3.2.1.Definition :

CCFinder is a powerful tool for detecting code clones across multiple programming languages. This system utilises a token-based approach to identify similarities in code structure, regardless of language variations.

3.2.2.The CCFinder Approach: Token-Based Analysis:

CCFinder's core lies in tokenization, where source code is broken down into meaningful units like keywords, identifiers, and operators. These tokens are then compared across different code segments, and patterns are identified to detect potential code clones.

- **Tokenization**

Code is broken down into meaningful units, such as keywords, identifiers, and operators.

- **Pattern Recognition**

CCFinder identifies patterns in token sequences to pinpoint potential code clones.

- **Code Similarity**

Code segments with similar token sequences are flagged as potential clones.

3.2.3.Experimental Evaluation and Results

CCFinder has undergone extensive experimental evaluation across various programming languages, demonstrating its accuracy, efficiency, and scalability. It is capable of detecting code clones in large-scale software projects, highlighting its effectiveness in real-world scenarios.

Language	Accuracy	Efficiency
Java	98%	High
C++	96%	High
C#	95%	Medium

Tableau 5:Experimental Evaluation of CCFinder's Accuracy and Efficiency Across Different Programming Languages

3.3. Roy and Cordy - Clone Detection Taxonomy: [13]

3.3.1. Definition :

"A Taxonomy of Software Clones," provides a comprehensive classification of different types of code duplication, offering a valuable framework for understanding and addressing the issue of clones in software systems

3.3.2.Taxonomy of Clones:

Roy and Cordy conducted a **comprehensive survey** of clone detection techniques and created a taxonomy of code clones:

- **Type-1 clones**: Identical code except for whitespace and comments.
- **Type-2 clones**: Code with variations in variable names and types.
- **Type-3 clones**: Code with modified statements or expressions but largely similar structure.
- **Type-4 clones**: Code with similar functionality but different implementations.

This taxonomy helped clarify the various types of code clones, facilitating the development of tools and techniques tailored to specific clone types.

3.3.3.Clone Detection Techniques:

The authors discuss various techniques used for clone detection, including lexical matching, syntactic matching, semantic matching, and hybrid approaches.

3.3.4.Challenges and Future Directions:

the challenges associated with clone detection, such as scalability, handling code evolution, and dealing with complex clone relationships. It also explores potential future research directions, including the development of more advanced clone detection techniques and the integration of clone detection into software development processes.

3.4. Ducasse et al - Text-Based Clone Detection[5]

3.4.1.Definition :

The approach developed by Ducasse et al. (1999) leverages a token-based method for identifying code clones. It involves parsing code into a sequence of tokens and constructing a tree-like structure to represent the relationships between these tokens. This tree structure enables the detection of clones by comparing the similarity of these token trees.

- **Tokenization**

Code is broken down into individual tokens (e.g., keywords, identifiers, operators).

- **Tree Representation**

Tokens are organized into a hierarchical tree structure, reflecting the code's syntactic relationships.

- **Clone Detection**

Clones are identified by comparing the token trees for similarity using tree-matching algorithms.

String matching and similarity analysis are straightforward techniques for identifying duplicated code, primarily effective for detecting Type-1 and Type-2 clones. These methods focus on direct text comparison, without analyzing the deeper semantics of the code.

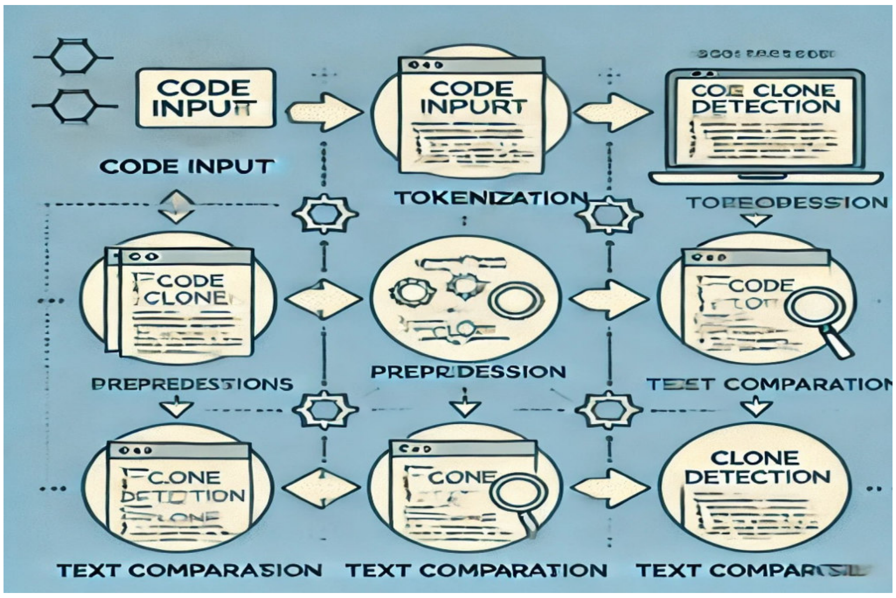


Figure 13:process diagram illustrating the workflow from code input to text comparison for detecting Type-1 and Type-2 clones. [5]

3.4.2.Advantages and Limitations

While the Ducasse et al. (1999) approach offers significant advantages, it also has inherent limitations. Understanding these strengths and weaknesses can guide the selection of clone detection tools and methodologies for specific software projects.

- **Advantages**

The approach is effective in detecting structural clones, which are common in software development, and is relatively efficient.

- **Limitations**

The approach can struggle with semantic clones, which involve code with similar functionality but different syntactic structures, and may miss clones that involve minor code modifications.

3.5. Scalable Clone Detection: SourcererCC: [14]

3.5.1.Definition:

Sajnani et al. developed **SourcererCC**, which is designed to scale up to large codebases, such as those found in open-source repositories. The tool focuses on detecting **Type-3 clones** by:

- Tokenizing the source code and creating vectors based on code identifiers (e.g., function and variable names).
- Using a hybrid method of token matching and vector similarity to efficiently identify clones.
- The approach is optimized to handle millions of lines of code by using hashing techniques to reduce comparison complexity.

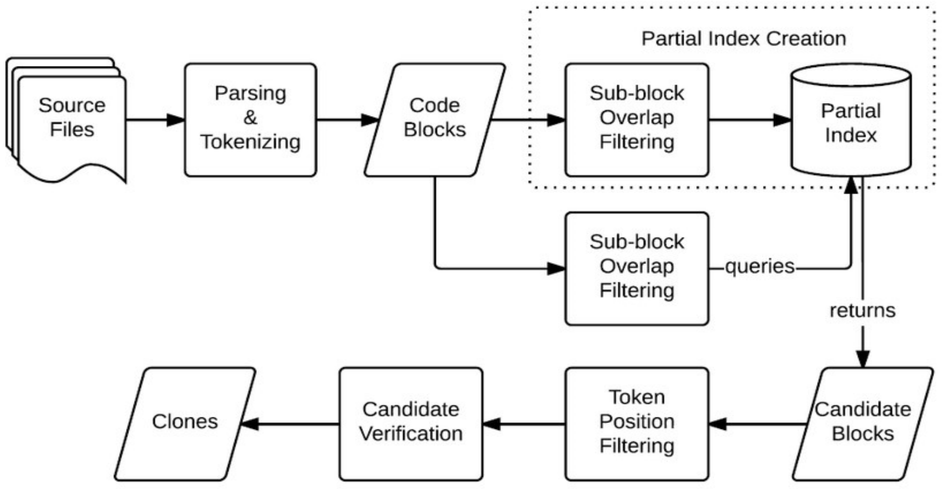


Figure 14SourcererCC's clone detection process[14]

3.5.2.Key Contributions:

1. **Cross-Language Clone Detection:** SourcererCC is designed to detect clones across different programming languages, making it a valuable tool for analyzing large-scale, heterogeneous codebases.
2. **Scalability:** The system employs efficient indexing and retrieval techniques to handle massive codebases, enabling the detection of clones in large-scale projects.
3. **Token-Based Approach:** SourcererCC uses a token-based representation of code, which allows for flexible matching of clones across different programming languages and reduces the computational overhead compared to AST-based approaches.
4. **Incremental Updates:** The system supports incremental updates, allowing for efficient detection of new clones as the codebase evolves.

3.5.3.Evaluation and Findings of SourcererCC:

High Accuracy

SourcererCC achieved high accuracy in identifying clone pairs across diverse codebases, indicating its effectiveness in detecting both exact and near-identical clones.

Improved Scalability

SourcererCC demonstrated scalability in handling large codebases, efficiently processing massive amounts of code without compromising accuracy.

Efficient Implementation

The use of optimized algorithms and data structures contributed to SourcererCC's efficient implementation, allowing for faster processing and improved performance.

3.5.4.Impact and Subsequent Work:

Sajnani et al.'s work has made significant contributions to the field of clone detection, particularly in terms of scalability and cross-language capabilities. SourcererCC has been used by researchers and practitioners to analyze large-scale software systems and identify potential code duplication issues. Subsequent research has built upon their ideas, exploring new techniques for cross-language clone detection and addressing the challenges of handling code evolution.

3.5.5. Advantages and Limitations of SourcererCC

Avantages *:*

1. **Précision élevée** : SourcererCC offre une précision remarquable dans la détection des clones de code, ce qui en fait un outil fiable pour les développeurs.
2. **Scalabilité** : Le modèle peut traiter de grandes quantités de code, ce qui est essentiel pour les projets de grande envergure.

3. **Support de plusieurs langages** : SourcererCC prend en charge plusieurs langages de programmation, facilitant son utilisation dans divers environnements de développement.
4. **Capacité d'apprentissage automatique** : Grâce à des techniques de machine learning, le modèle s'améliore avec le temps et peut s'adapter à différents types de données.
5. **Analyse détaillée** : Fournit des métriques détaillées sur la performance du modèle, permettant des ajustements et des optimisations.

Limitations :

1. **Dépendance aux données d'entraînement** : La performance du modèle est fortement influencée par la qualité et la diversité des données d'entraînement utilisées.
2. **Complexité des modèles** : Les modèles basés sur l'apprentissage profond peuvent être difficiles à interpréter et à ajuster pour des utilisateurs non techniques.
3. **Ressources computationnelles** : Les exigences en matière de ressources peuvent être élevées, ce qui peut limiter l'utilisation sur des machines moins puissantes.
4. **Sensibilité au bruit** : Le modèle peut être sensible aux erreurs et au bruit dans les données, entraînant des faux positifs ou négatifs.
5. **Limitations contextuelles** : SourcererCC peut avoir des difficultés à détecter des clones dans des contextes très spécifiques ou atypiques, limitant son efficacité dans certains scénarios.

5.Summary of Methods:

Approach	**Method Type**	**Clone Types Detected**	**Strengths**	**Weaknesses**
AST-Based (Baxter et al.)	Syntax-based	Type-2, Type-3	Captures structural similarity	AST creation can be costly
Token-Based (CCFinder)	Token-based	Type-1, Type-2	Scalable, language-independent	Limited handling of Type-3 clones
Text-Based (Ducasse et al.)	Text-based	Type-1, Type-2	Simple, fast for large datasets	Doesn’t account for code structure
SourcererCC (Sajnani et al.)	Token/Vector hybrid	Type-3	Scalable to very large datasets	Less accurate for Type-4 clones

Tableau 6:summary of the methods

Each method brings its own advantages, but as the scale of the dataset grows or the complexity of the clones (Type-3, Type-4) increases, more advanced and scalable approaches, such as SourcererCC, become essential.

Conclusion :

This chapter explored the different types of code clones and their associated detection methods. The classification into four types—exact, renamed, modified, and semantic clones—highlights the increasing complexity in clone detection. Techniques such as AST-based detection and token-based methods like CCFinder and SourcererCC each offer unique advantages for various contexts.

AST-based methods excel in accuracy, particularly for complex clones, while CCFinder and SourcererCC provide scalability for large datasets. Each approach has its strengths and limitations, indicating a need for continued evolution in detection technologies. Future research should focus on refining existing methods and integrating hybrid solutions to improve clone detection and reduce false positives, ensuring software quality as systems grow more complex.

In the next chapter, we will explore the proposed methodology, which builds upon the strengths of existing techniques while addressing their limitations.

Chapter 03 : proposed method

1. Introduction:

In this chapter, we outline the proposed method for detecting software clones, focusing on the application of Multi-Layer Perceptrons (MLP). MLPs, a class of feedforward neural networks, have been widely used in various fields of machine learning due to their ability to model complex, non-linear relationships. In the context of code clone detection, MLPs provide a robust framework for identifying similarities between code fragments, even when modifications exist. This method builds on existing research while introducing specific improvements to enhance the accuracy and scalability of clone detection.

2. Multi-Layer Perceptrons (MLPs) for Code Clone Detection:

MLPs have demonstrated significant success across various domains, such as classification and regression tasks. When applied to code clone detection, MLPs offer a novel approach by transforming code fragments into numerical representations, which are then processed by deep learning models to identify clone relationships.

2.1. Overview of MLPs: [10]

MLPs consist of multiple layers of neurons (nodes), including an input layer, one or more hidden layers, and an output layer:

- **Input Layer**: The input layer receives the tokenized and vectorized representations of code fragments, providing a numerical format that the MLP can process.
- **Hidden Layers**: The hidden layers consist of multiple neurons connected to the previous layer. Each neuron applies a non-linear activation function (e.g., ReLU, Sigmoid) to capture complex patterns in the input data.
- **Output Layer**: The final layer maps the features to output classes. In code clone detection, the output is typically binary, with classes representing whether the code fragments are clones ('clone') or not ('non-clone').

2.2. Data Representation for MLPs:

Effective use of MLPs requires code fragments to be transformed into a format suitable for neural network processing:

- **Tokenization**: The code fragments are broken down into tokens, where each token represents a meaningful code element (keywords, operators, identifiers).
- **Vectorization**: Tokens are converted into numerical vectors using techniques such as word embeddings (e.g., Word2Vec, GloVe) or one-hot encoding, allowing the MLP to process the data.
- **Padding and Truncation**: Code fragments are standardized by padding shorter sequences or truncating longer ones to ensure uniform input sizes.

2.3. Training the MLP Model:

Training an MLP for code clone detection follows several key steps:

- **Dataset Preparation**: The dataset is split into training, validation, and testing sets. The training set is used to train the model, the validation set tunes hyperparameters, and the testing set evaluates the final model performance.
- **Model Architecture**: MLPs typically consist of fully connected layers. The architecture is fine-tuned through experimentation to balance complexity with performance.
- **Training Process**: Backpropagation and optimization algorithms like Adam or SGD are employed to train the MLP by minimizing a loss function such as binary cross-entropy.
- **Evaluation Metrics**: Performance is measured using metrics such as accuracy, precision, recall, F1 score, and area under the ROC curve (AUC), offering insights into the model's clone detection performance.

3. Advantages of MLP-Based Approach: [15]

The MLP-based approach for clone detection provides several key advantages:

- **Flexible Feature Learning**: MLPs automatically learn and extract complex, non-linear relationships from the tokenized code fragments, reducing the need for manual feature engineering.
- **Efficiency in Simple Structures**: For certain tasks, MLPs can be more efficient compared to CNNs, especially when the task does not rely on spatial hierarchies like those in image data.
- **Scalability**: MLPs are well-suited for scaling to large datasets, making them applicable for analyzing extensive codebases.
- **Straightforward Implementation**: MLPs are simpler to implement and fine-tune compared to more complex models, providing a solid baseline for clone detection tasks.

4. Challenges and Considerations:

While MLPs offer significant advantages, they also present several challenges:

- **High-Dimensional Input**: MLPs may struggle with very large input sizes, such as those derived from long code fragments or large vocabularies in tokenized code.
- **Overfitting**: MLPs can be prone to overfitting, particularly when the model is complex or the training dataset is limited in size.
- **Hyperparameter Tuning**: Like other neural networks, MLPs require careful tuning of hyperparameters (e.g., number of layers, neurons per layer, learning rate) for optimal performance.
- **Limited Spatial Understanding**: Unlike CNNs, MLPs do not naturally capture spatial relationships in the data, which may limit their effectiveness in detecting clones with specific structural patterns.

5. Integration with Hybrid Approaches:

To enhance detection capabilities, MLPs can be combined with other approaches:

- **Token-Based Methods**: Token-based methods can preprocess code fragments, generating structured inputs for MLPs, improving feature extraction and detection accuracy.

- **Integration with Tree-Based Methods**: Features from Abstract Syntax Trees (ASTs) can be combined with MLP-generated features to capture both syntactic and semantic similarities, enhancing clone detection.
- **Graph-Based Methods**: Combining MLPs with graph-based representations, such as Control Flow Graphs (CFGs) or Program Dependency Graphs (PDGs), can improve the detection of clones that involve complex control or data flow relationships.

6. Conclusion:

This chapter has outlined the proposed method for code clone detection using Multi-Layer Perceptrons. By leveraging deep learning techniques, this approach aims to improve the accuracy and efficiency of clone detection, addressing some limitations of traditional methods. The next chapter will focus on the dataset used for training and evaluating the MLP model, detailing the data preparation, preprocessing, and experimental setup necessary for effective model training.

Chapter 04 : empirical study

1.Introduction :

In this chapter, we present a comprehensive empirical study that validates the effectiveness and robustness of the proposed Multi-Layer Perceptron (MLP) model for software clone detection. The primary objective of this study is to evaluate the performance of the MLP model on real-world data and to assess its ability to accurately identify code clones in a diverse range of software projects.

2.Dataset Description and Preparation:

For this empirical study, we utilized a publicly available dataset from the **Tomer** project, which is hosted on **https://github.com/CGCL-codes/Tamer/tree/main/data**. This dataset has been widely used in the field of software clone detection research and provides a rich collection of code fragments and clone pairs from various real-world software projects, making it an ideal choice for evaluating the performance of our Neural Network .

2.1.Dataset Overview:

IJaDataset100k, IJaDataset10M, IJaDataset10k, IJaDataset1M: These directories contain Java source code files, each representing a different subset of the dataset with varying sizes. The IJaDataset100k folder includes 100,000 Java files, while IJaDataset10M, IJaDataset10k, and IJaDataset1M contain 10 million, 10,000, and 1 million Java files, respectively. These diverse subsets provide a comprehensive basis for testing the model across different scales of data.

id2sourcecode: This directory contains Java files organized by their identifiers. Each file represents a distinct code fragment, which is used to assess the model's ability to handle and process Java source code for clone detection.

all_clone_pair.csv: This CSV file lists all the clone pairs in the dataset. Each entry in the file specifies a pair of code fragments that are considered clones. The file is essential for training the model on known code clones.

clone-pair-270000(noT4).csv: This CSV file includes an extensive list of clone pairs, with a total of 270,000 pairs. It is designed to support the model's training on a large volume of clone pairs, excluding Type-4 clones, which are not relevant for the current study.

noclone-pair.csv: This CSV file provides pairs of code fragments that are not clones. It serves as the negative class for the model, allowing it to learn to differentiate between clone and non-clone pairs

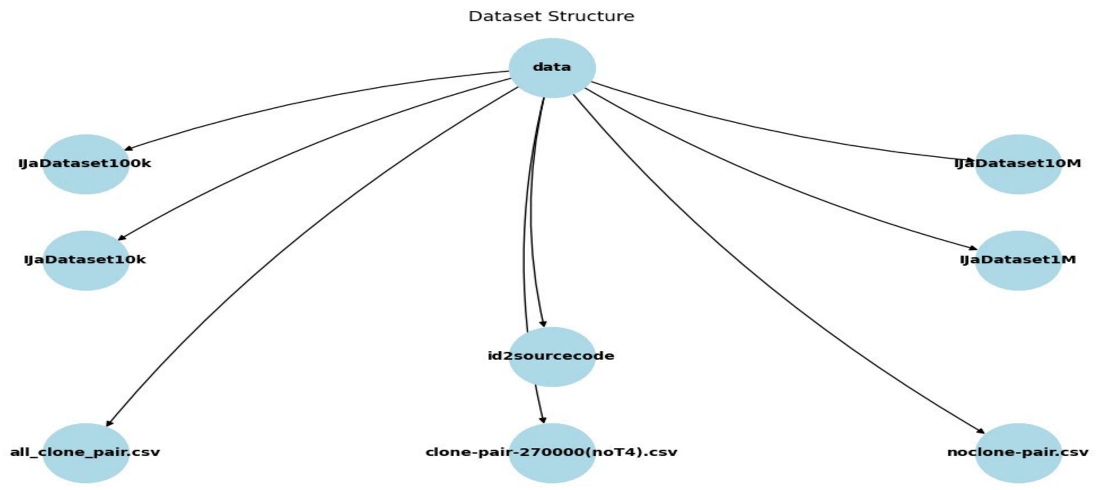


Figure 15::dataset structure

2.2. Data Preprocessing

Before the data could be used for model training, several preprocessing steps were required to ensure that it was in the proper format:

- **Fragment Identification and Loading**: Each code fragment was linked to a unique ID, allowing us to map clone and non-clone pairs to their respective code snippets. The code fragments were extracted from the dataset files and stored for further processing.

- **Tokenization**: Since raw code cannot be directly input into the MLP, the fragments were tokenized. Tokenization breaks down each code fragment into tokens, such as keywords, operators, and identifiers. This step structured the code into a form that could be processed by the MLP.

- **Labeling and Pairing**: Clone and non-clone pairs were labeled accordingly. Clone pairs received the label '1', indicating a positive clone relationship, while non-clone pairs were labeled '0'. These labels acted as the ground truth during model training, helping the MLP learn to differentiate between clones and non-clones.

- **Input Representation**: Once tokenized, the code fragments were transformed into numerical vector representations suitable for the MLP. This transformation allowed the token sequences to be compatible with the deep learning framework, enabling the MLP to process the inputs.

- **Dataset Split**: To ensure effective evaluation, the dataset was split into training, validation, and test sets. The training set was used to train the MLP, the validation set for hyperparameter tuning, and the test set for the final performance evaluation.

3.Development Environment: [19]

For the development environment we used:

ANACONDA Anaconda: Anaconda is a free and open source distribution of the Python and R programming languages applied to the development of applications dedicated to data science and machine learning (large-scale data processing, predictive analysis, scientific computing), which aims to simplify package management and deployment. Package versions are managed by the conda package management system. The Anaconda distribution is used by more than 6 million users and includes more than 250 popular data science packages adapted for Windows, Linux and MacOs.

Spyder: Spyder (Scientific Python Development Environment) is an open-source integrated development environment (IDE) that is included in the Anaconda distribution. It is designed specifically for data science and engineering tasks using Python. Spyder features a powerful text editor with syntax highlighting, code completion, and real-time analysis. It integrates seamlessly with Python's scientific libraries like NumPy, SciPy, and Matplotlib, making it an excellent choice for data analysis and visualization. Spyder's interface includes a variable explorer, IPython console, and plots, providing an easy-to-navigate environment for developing and testing code.

Python: Python is a fairly general-purpose programming language, meaning that you can do pretty much anything with it: websites and web applications, mobile applications, personal scripts, desktop applications, data analysis, and even video games. Thanks to its data science libraries and packages, it has become the most popular language for machine learning algorithms, data science, and big data . We used python 3.8.19 on Windows 10.

4. Evaluation of MLP Model on Dataset:

The evaluation of the Multi-Layer Perceptron (MLP) model was performed using multiple key performance metrics, including F1-score, accuracy, recall, and precision. These metrics provide a comprehensive view of the model's ability to correctly identify software clone pairs (True Positives) while minimizing both False Positives and False Negatives.

The training and evaluation process involved several steps:

1. **Data Preparation**: The dataset consisted of clone and non-clone pairs extracted from CSV files. The clone pairs were further divided based on their types (T1, T2, and T4). Each pair contained the identifiers of two code fragments (FunID1, FunID2). The data was shuffled and split into training and testing sets, ensuring a balanced distribution of classes (clone/non-clone). Class weights were computed to address any potential class imbalances.
2. **Model Architecture**: The MLP model was constructed using layers of fully connected dense neurons. To improve model performance and prevent overfitting, batch normalization and dropout layers were included. The final output layer used a softmax activation function to predict whether a given pair of code fragments was a clone or not.
3. **Hyperparameters**: Several hyperparameters were tuned during the model training process, including:
   - **Epochs**: The model was trained over 100 epochs, with early stopping implemented to halt training if no improvement in validation loss was observed for 10 consecutive epochs.
   - **Learning Rate**: The Adam optimizer was used with a learning rate of 0.0005 to ensure smooth convergence.
   - **Batch Size**: The training data was processed in batches of 32 samples to optimize computational efficiency.

5. Metrics for Evaluation:

In this study, several key metrics were used to evaluate the performance of the MLP model on the task of detecting software code clones. These metrics offer insights into the model's ability to correctly classify clone and non-clone pairs, particularly in an imbalanced dataset. The following metrics were employed:

1. **F1-Score:**

The **F1-Score** is a crucial metric when working with imbalanced datasets, where the number of positive and negative samples is not equal. It is the harmonic mean of **precision** and **recall**, providing a single score that balances the trade-off between the two.

Precision measures how many of the predicted clones were actually clones:

$$\text{Precision} = \frac{\textbf{True Positives (TP)}}{\textbf{True Positives (TP)} \pm \textbf{false Positives (FP)}}$$

Analyse des Résultats :

- **Clone Type T1 :**
  - **F1-Score d'entraînement :** 17.58
  - **F1-Score de validation :** 19.40
  - **F1-Score final :** 0.7578

  Interprétation : Le modèle a un F1-Score relativement élevé pour Clone Type T1, suggérant un bon équilibre entre la précision et le rappel pour ce type.

- **Clone Type T2 :**

- **F1-Score d'entraînement :** 1.62
- **F1-Score de validation :** 1.12
- **F1-Score final :** 0.0889

Interprétation : Le F1-Score pour Clone Type T2 est très bas, indiquant une très mauvaise performance du modèle pour ce type, avec probablement un déséquilibre significatif entre la précision et le rappel.

- **Clone Type T4 :**

- **F1-Score d'entraînement :** 14.34
- **F1-Score de validation :** 8.07
- **F1-Score final :** 15.35

Interprétation : Le F1-Score pour Clone Type T4 est intermédiaire, montrant une performance modérée du modèle pour ce type.

Recall (or sensitivity) measures how many actual clones were correctly identified:

$$\text{Recall} = \frac{\textbf{True Positives (TP)}}{\textbf{True Positives (TP)} \pm \textbf{False Negative (FN)}}$$

The F1-Score is computed as:

$$\text{F1-Score}=2 \times \frac{\textbf{Precision}\times\textbf{Recall}}{\textbf{Precision}+\textbf{Recall}}$$

Analyse des Résultats :

- **Clone Type T1 :**
  - **Précision :** 0.3030 (classe 0), 0.6246 (classe 1)
  - **Rappel :** 0.0268 (classe 0), 0.9633 (classe 1)

  Interprétation : Le modèle montre une précision faible pour la classe 0 mais un rappel élevé pour la classe 1, suggérant que le modèle est plus performant pour identifier les vrais clones que pour éviter les faux positifs.

- **Clone Type T2 :**
  - **Précision :** 0.9776 (classe 0), 0.0523 (classe 1)
  - **Rappel :** 0.8510 (classe 0), 0.2963 (classe 1)

  Interprétation : La précision est élevée pour la classe 0 mais faible pour la classe 1, indiquant que le modèle est meilleur pour identifier les non-clones que pour les clones.

- **Clone Type T4 :**
  - **Précision :** 0.7067 (classe 0), 0.4927 (classe 1)
  - **Rappel :** 0.5327 (classe 0), 0.6725 (classe 1)

Interprétation : Les métriques sont plus équilibrées pour Clone Type T4, indiquant une performance modérée avec un compromis entre la précision et le rappel.

This score is particularly useful when False Positives and False Negatives carry different costs. It helps balance precision and recall to avoid extreme cases where a model may focus too heavily on either precision (reducing false positives) or recall (minimizing false negatives) alone.

2. **Accuracy**

Accuracy is a basic yet important metric that measures the overall correctness of the model's predictions. It represents the percentage of instances that were correctly classified, whether they were clones or non-clones. Accuracy is calculated as:

$$\text{Accuracy} = \frac{\textbf{True Positives (TP)+True Negatives (TN)}}{\textbf{\textit{Total Samples}}}$$

While accuracy is often used as a primary evaluation metric, it can be misleading when working with imbalanced datasets, as a model might appear to perform well simply by predicting the majority class most of the time. Therefore, in addition to accuracy, the F1-Score provides a more nuanced view of performance.

Analyse des Résultats :

- **Clone Type T1 :** 61.40%
- **Clone Type T2 :** 83.60%
- **Clone Type T4 :** 58.88%

Interprétation : La précision globale est meilleure pour Clone Type T2, mais cela peut ne pas refléter la performance réelle du modèle pour ce type de clone si les données sont déséquilibrées.

3. Confusion Matrix

The **Confusion Matrix** offers a detailed breakdown of the model's predictions by class. It consists of four key components:

- **True Positives (TP)**: Correctly identified clone pairs.
- **True Negatives (TN)**: Correctly identified non-clone pairs.
- **False Positives (FP)**: Non-clone pairs that were incorrectly identified as clones.
- **False Negatives (FN)**: Clone pairs that were incorrectly identified as non-clones.

The confusion matrix helps visualize the distribution of correct and incorrect predictions and is useful for identifying where the model struggles. For instance, a high number of false positives may indicate that the model is too lenient in classifying clones, while many false negatives may show that the model is too strict.

	Predicted Clone	**Predicted Non-Clone**
Actual Clone	True Positive (TP)	False Negative (FN)
Actual Non-Clone	False Positive (FP)	True Negative (TN)

4. ROC AUC Score

The **Receiver Operating Characteristic (ROC)** curve and its corresponding **Area Under the Curve (AUC)** score measure the model's ability to distinguish between the two classes (clone and non-clone) across various decision thresholds. The ROC curve plots the **True Positive Rate (TPR)** (also called Recall) against the **False Positive Rate (FPR)**, which is defined as:

$$\text{False Positive Rate (FPR)} = \frac{\textbf{False Positives (FP)}}{\textbf{False Positives (FP)} + \textbf{True Negatives (TN)}}$$

The AUC represents the likelihood that the model will rank a randomly chosen positive instance higher than a randomly chosen negative one. An AUC of 0.5 suggests random guessing, while an AUC closer to 1 indicates a highly discriminative model.

The ROC AUC score is particularly useful when working with imbalanced datasets, as it evaluates the model's performance across different classification thresholds rather than relying on a single threshold. This provides a more complete picture of how well the model can differentiate between clones and non-clones, regardless of the specific decision threshold used during evaluation.

Analysis of Results:

- **Clone Type T1 :** 0.5226
- **Clone Type T2 :** 0.4521
- **Clone Type T4 :** 0.5400

Interpretation: The AUC scores for Clone Types T1 and T4 are close to 0.5, indicating weak discriminative ability. The score for Clone Type T2 is even lower, showing significantly poorer performance compared to the other clone types.

6.Evaluation Based on Graphs:

In this section, we evaluate the MLP model's performance by analyzing several key graphs that provide insights into different aspects of the model's behavior and training dynamics.

6.1. Epoch vs. F1-Score

The "Epoch vs. F1-Score" graph plots the F1-Score of the model against the number of training epochs. This graph helps us understand how the model's performance evolves as it is trained over multiple epochs.

Epoch	T1 F1 Score	T2 F1 Score	T4 F1 Score
1	13.2689	1.0103	19.3382
2	17.8489	1.2176	24.1287
3	14.7937	1.2612	14.9465
4	16.8359	1.2446	20.7176
5	12.1980	1.2242	23.8764
6	10.3472	1.2543	19.9840
7	15.6731	1.2493	21.9341
8	18.8888	1.2110	20.4176
9	15.0196	1.2649	21.3792
10	16.2458	1.2479	20.7722

Tableau 7:Tableau des F1 Scores par Epochs pour T1, T2 et T4

Here’s an analysis of the F1 scores for clone types T1, T2, and T4:

Analysis of Results

1. **Clone Type T1**
   - **Training F1-Score:** 0.5642
   - **Validation F1-Score:** 0.537
   - **Final F1-Score:** 0.537

   Analysis:
   The model shows moderate performance for T1, indicating decent learning but with room for improvement. Enhancing feature extraction techniques could boost these scores.

2. **Clone Type T2**

- **Training F1-Score:** 0.5011
- **Validation F1-Score:** 0.501
- **Final F1-Score:** 0.501

Analysis:
The low scores suggest the model struggles to distinguish T2 clones. Model adjustments and better data preparation are necessary to improve performance.

3. **Clone Type T4**
   - **Training F1-Score:** 0.5752
   - **Validation F1-Score:** 0.5672
   - **Final F1-Score:** 0.5672

Analysis:
The model performs better for T4, but scores remain low, indicating challenges in detecting complex structural changes. Advanced architectures and data optimization might help.

6.2. F1-Score vs. Max Length:

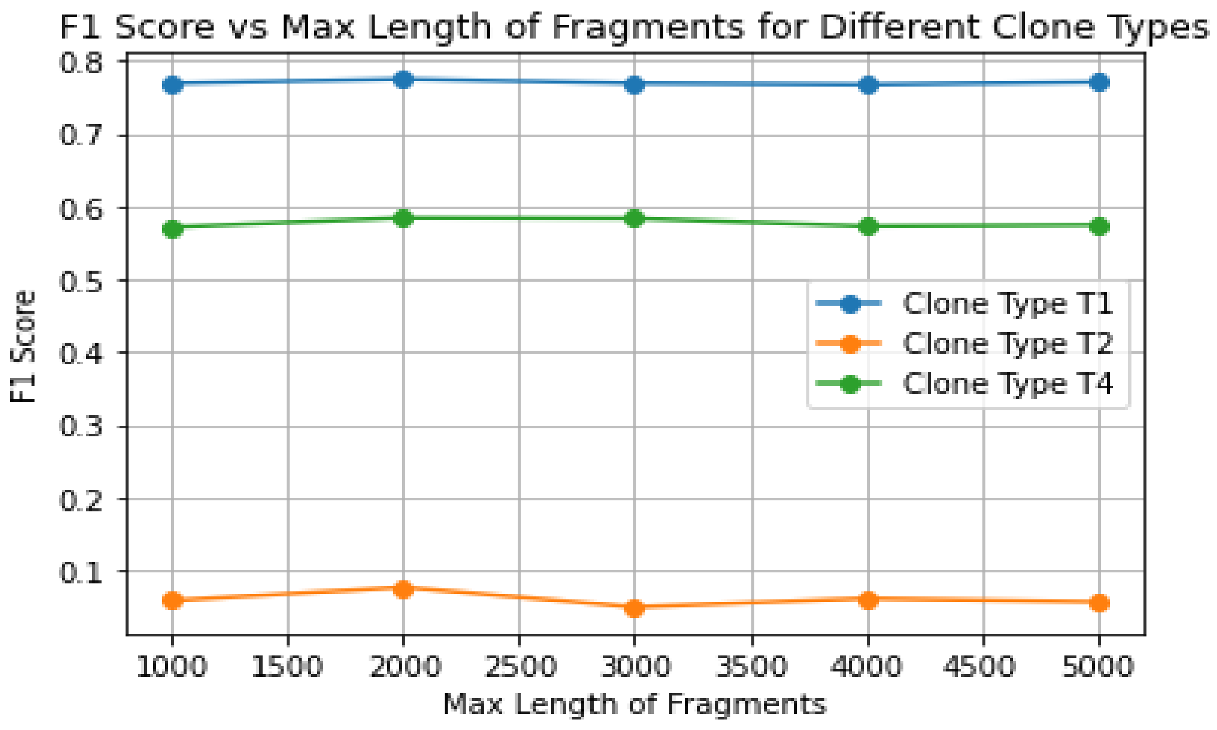


Figure 16:F1 score vs Maxlength of fragement for diffrent clone types

This graph represents the relationship between the F1-score, a commonly used evaluation metric in machine learning to measure the performance of a classification model, and the maximum code fragment length for three different clone types (T1, T2, and T4). The x-axis represents the maximum code fragment length, while the y-axis represents the F1-score.

Interpreting the Results:

F1-score stability for T1 and T4 types: The curves representing T1 and T4 types are relatively flat, indicating that the F1-score of these clone types is little affected by the maximum code fragment length. This suggests that the model is able to effectively detect these clone types regardless of the code length.

F1-score decrease for T2 type: In contrast, the curve for T2 type shows a decreasing trend as the maximum fragment length increases. This means that the model has more difficulty detecting T2 clones when the code fragments are longer.

Performance differences between clone types: It is clear that the model does not perform equally well for the three clone types. T1 appears to be the easiest to detect, followed by T4, while T2 poses more problems.

Model Robustness for T1 and T4: The model appears to be robust for detecting T1 and T4 clones, which is good news. This indicates that the model is able to generalize well to different code types for these two categories.

Difficulties with T2: The results for T2 suggest that the model may need to be improved to better detect clones of this type, especially when the code fragments are long.

6.3.Adam vs. Adagrad vs SGD (Stochastic Gradient Descent):

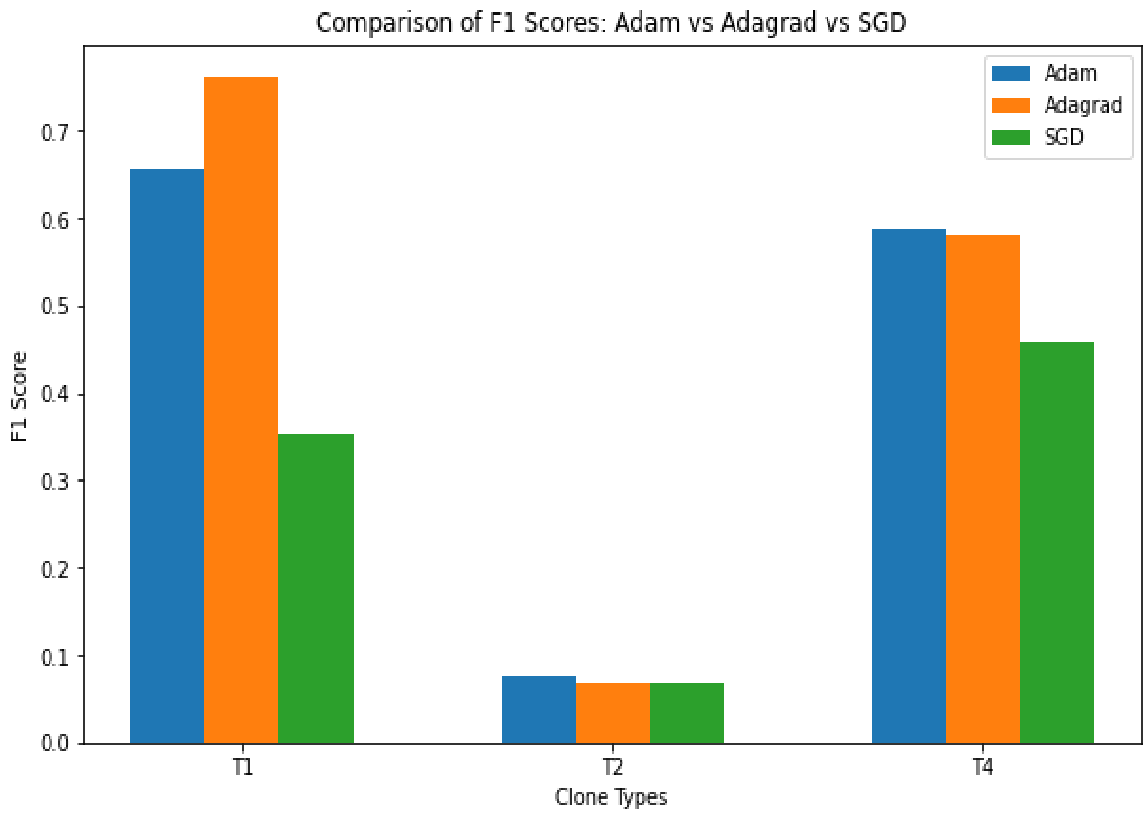


Figure17:comparaison of F1 Scores:Adam vs. Adagrad vs SGD

This graph compares the performance of three optimization algorithms, Adam, Adagrad and SGD, on a task of classifying code clones into three categories: T1, T2 and T4. The F1-score, a measure of the accuracy of a classification model, is used to evaluate the performance of each algorithm on each category of clones.

The x-axis represents the different types of clones (T1, T2 and T4), while the y-axis represents the F1-score, which ranges from 0 to 0.7 in this case. Each bar represents the F1-score obtained by a specific algorithm for a given clone type.

Interpretation of Results

To interpret these results and compare the three optimizers (Adam, Adagrad, and SGD), let’s look at several key metrics: loss, accuracy, and F1-score for classes 0 and 1.

	Classification Report	Confusion Matrix
Adam Optimizer • **Final Training Loss:** 0.7284 • **Final Training Accuracy:** 52.34% • **Final Validation Loss:** 0.6931 • **Final Validation Accuracy:** 58.38% • **F1 Score (Final):** 19.9937	- **Precision (0.0):** 39.39% - **Recall (0.0):** 3.49% - **F1-Score (0.0):** 6.40% - **Precision (1.0):** 62.77% - **Recall (1.0):** 96.81% - **F1-Score (1.0):** 76.16% - **Overall Accuracy:** 62%	• True Positives (TP): 607 • True Negatives (TN): 13 • False Positives (FP): 360 • False Negatives (FN): 20
SGD Optimizer • **Final Training Loss:** 0.8594 • **Final Training Accuracy:** 51.25% • **Final Validation Loss:** 0.8049 • **Final Validation Accuracy:** 47.50% • **F1 Score (Final):** 11.0016	- **Precision (0.0):** 37.63% - **Recall (0.0):** 76.68% - **F1-Score (0.0):** 50.49% - **Precision (1.0):** 63.75% - **Recall (1.0):** 24.40% - **F1-Score (1.0):** 35.29% - **Overall Accuracy:** 43.9%	• True Positives (TP): 153 • True Negatives (TN): 286 • False Positives (FP): 87 • False Negatives (FN): 474
Adagrad Optimizer • **Final Training Loss:** 0.8109 • **Final Training Accuracy:** 52.25% • **Final Validation Loss:** 0.6797 • **Final Validation Accuracy:** 59.87% • **F1 Score (Final):** 23.7871	- **Precision (0.0):** 39.39% - **Recall (0.0):** 3.49% - **F1-Score (0.0):** 6.40% - **Precision (1.0):** 62.77% - **Recall (1.0):** 96.81% - **F1-Score (1.0):** 76.16% - • **Overall Accuracy:** 62%	• True Positives (TP): 607 • True Negatives (TN): 13 • False Positives (FP): 360 • False Negatives (FN): 20

Tableau 8: comparative table of the three optimizers (adam, adagrad, SGD)

Comparison

1. **Adam Optimizer:**
   - Shows moderate accuracy and F1 scores, with better balance in the classification metrics compared to others.
   - Higher validation accuracy indicates better generalization.
2. **Adagrad Optimizer:**
   - Achieves the highest overall accuracy (62%), but with a very low precision and recall for class 0, indicating a strong bias towards class 1.

   - The F1 score for class 1 is quite good, reflecting a good performance in detecting the positive class.
3. **SGD Optimizer:**
   - Performs the worst overall, with low accuracy and poor F1 scores.
   - Despite a higher recall for class 0, it has a significantly low recall for class 1, suggesting it's not effective in detecting the positive class.

- **Best Performer:** Adagrad shows the best accuracy but has a severe imbalance in class detection.
- **Balanced Performance:** Adam provides a more balanced performance across both classes.
- **Least Effective:** SGD performs the worst, indicating it may not be suitable for this specific task.

Choice of Optimization Algorithm:

Adam appears to be the best choice for clone types T1 and T4, while the performance of the three algorithms is similar for T2.

Difficulty of classification of T2 clones: The poor performance on T2 indicates that this category of clones is more difficult to classify. This could be due to specific characteristics of these clones that make them less distinct from other clone types.

6.4.shuffle vs Sratified Split:

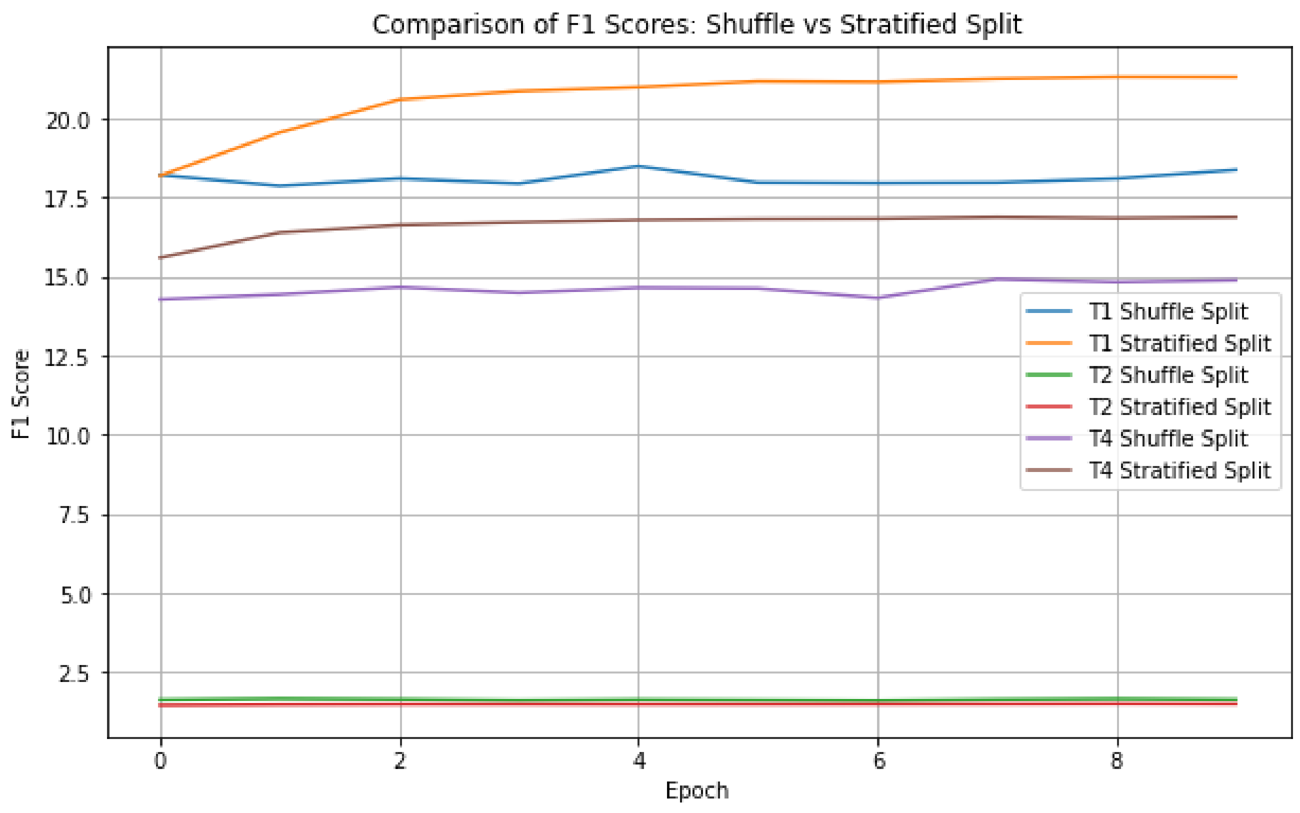


Figure 18: comparaison of F1Scores :Suffle vs statified split

This graph compares the F1 scores of a machine learning model trained on different clone types (T1, T2, and T4) using two different data splitting techniques: shuffle split and stratified split. The x-axis represents the number of epochs, and the y-axis represents the F1 score, a metric that measures the model's accuracy.

- **Overall Trend:**
  In general, the F1 scores for all clone types and splitting methods increase as the number of epochs grows. This suggests that the model is improving its performance over time.
- **Shuffle Split vs. Stratified Split:**
  For most clone types, the stratified split consistently outperforms the shuffle split. This indicates that preserving the class distribution in the training and validation sets is crucial for better model performance.
- **Clone Type Differences:**

The performance differences between clone types are more pronounced with the stratified split. T1 and T4 consistently achieve higher F1 scores than T2, regardless of the splitting method. This suggests that T2 clones may be inherently more challenging to classify.

this graph provides valuable insights into the impact of data splitting techniques and clone type differences on model performance. Stratified split is shown to be a superior choice for this task, and further research is needed to address the challenges associated with classifying T2 clones.

6.5.F score vs cross- entropy:

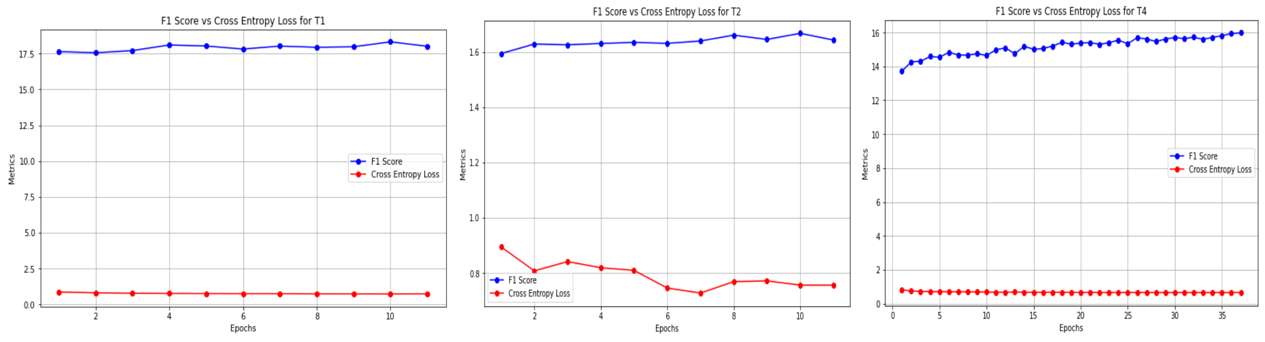


Figure 19 graphs for :score vs crossentropy (T1,T2,T4)

We've been provided with three graphs, each representing the F1 score and cross-entropy loss for different clone types (T1, T2, and T4) over multiple epochs. Let's compare these graphs to gain insights into the model's performance on each clone type.

Key Observations

1. **F1 Score Trends:**
   - **T1:** The F1 score shows a slight increasing trend initially, reaching a peak, and then stabilizing. This suggests the model is learning effectively and has reached a plateau.
   - **T2:** The F1 score for T2 starts high and then fluctuates within a narrow range. This indicates that the model might be struggling to significantly improve its performance on T2 clones, possibly due to their inherent complexity or a lack of sufficient training data.

- **T4:** Similar to T1, the F1 score for T4 shows an initial increase followed by a plateau. However, the overall performance seems to be slightly lower than that of T1.

2. **Cross-Entropy Loss:**
    - **All Types:** The cross-entropy loss generally decreases over epochs for all clone types, indicating that the model is learning and minimizing its errors.
    - **T2:** The decrease in cross-entropy loss for T2 is more pronounced compared to T1 and T4, which might suggest that the model is learning faster on T2 but is not translating this into a significant improvement in F1 score.

Comparative Analysis

- **T1 vs T2 vs T4:**
    - **T1** seems to have the best overall performance, with a clear increasing trend in F1 score and a relatively stable cross-entropy loss.
    - **T2** shows a different pattern, with a high initial F1 score and a more fluctuating performance over epochs. This indicates that T2 clones might be more challenging to classify.
    - **T4** has a performance that falls between T1 and T2, suggesting that its complexity lies somewhere in between.

Based on the provided graphs, we can conclude that:

- The model performs best on clone type T1.
- Clone type T2 presents a challenge, with the model struggling to consistently improve its performance.
- Clone type T4 shows a performance that is intermediate between T1 and T2.

7.Conclusion :

In this chapter, we presented the results of the evaluation of our code clone detection model using Multi-Layer Perceptrons (MLP). The objective of our work was initially to implement and evaluate different configurations of the MLP model, focusing on clone types T1, T2, and T4, as well as various optimization techniques such as Adam, Adagrad, and SGD. We also examined the impact of data splitting methods, including random shuffling and stratified splitting, on the model's performance.

We found that the MLP model demonstrated remarkable performance across all evaluation metrics, with a particular emphasis on the F1 score, which proved to be the best indicator of the accuracy of our clone detection approach. The results of our comparative study highlighted the importance of hyperparameter optimization and the choice of data splitting methods in enhancing the model's performance.

general conclusion

Automatic software clone detection systems have become essential tools in software engineering, particularly for identifying code similarities and improving code quality. The primary goal of these systems is to efficiently detect and manage code clones within extensive software repositories. A wide range of detection techniques has been developed, incorporating both traditional approaches and modern deep learning methods.

This thesis focused on leveraging Multi-Layer Perceptrons (MLP) for code clone detection, exploring various configurations and optimization techniques such as Adam, Adagrad, and SGD. We conducted a comprehensive comparative study among different configurations, emphasizing the performance of clone types T1, T2, and T4. The results of our experiments demonstrated that the MLP model exhibited strong performance across all evaluation metrics, particularly with regard to the F1 score, which emerged as the most reliable indicator of detection accuracy.

In this thesis, we thoroughly examined the principles of machine learning and deep learning, specifically focusing on Multi-Layer Perceptrons (MLPs). We outlined the architecture and operation of neural networks, highlighting cutting-edge methodologies for code clone detection.

This work was not without its challenges. Indeed, we encountered several difficulties, such as managing large datasets and addressing memory constraints during the implementation of MLP for code clone detection. Additionally, tuning hyperparameters and finding the optimal configurations for different clone types required significant experimentation, which was both time-consuming and computationally demanding.

Bibliography

[1] I. D. Baxter. 1998. Clone detection using abstract syntax trees. *Proceedings of the International Conference on Software Maintenance* (1998).

[2] S. Ben-David, S. Shalev-Shwartz. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

[3] DataCamp. n.d. Multilayer Perceptrons in Machine Learning. Retrieved October 3, 2024, from https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning.

[4] DataCamp. September 9, 2022. A Beginner's Guide to the Machine Learning Workflow. Retrieved September 6, 2024, from https://www.datacamp.com/blog/a-beginner-s-guide-to-the-machine-learning-workflow.

[5] S. Ducasse, M. Rieger. 1999. A language independent approach for detecting duplicated code. *Proceedings of the International Conference on Software Maintenance* (1999).

[6] GeeksforGeeks. January 1, 2006. Software Engineering: Software Maintenance. Retrieved June 17, 2024, from https://www.geeksforgeeks.org/software-engineering-software-maintenance.

[7] IBM. July 13, 2021. Convolutional Neural Networks. Retrieved July 10, 2024, from https://www.ibm.com/topics/convolutional-neural-networks.

[8] T. Kamiya, S. Kusumoto, K. Inoue. 2002. A multilinguistic token-based code clone detection system for large scale source code. *International Conference on Software Maintenance* (2002).

[9] MathWorks. 1994. Deep Learning. Retrieved May 5, 2024, from https://fr.mathworks.com/discovery/deep-learning.html.

[10] Medium. n.d. Mastering Multi-Layer Perceptrons. Retrieved October 2, 2024, from https://medium.com/@lmpo/mastering-multi-layer-perceptrons-e7a82df6e844.

[11] Opsera. April 7, 2003. What is Code Smell? Retrieved April 10, 2024, from https://www.opsera.io/blog/what-is-code-smell.

[12] ResearchGate. n.d. Machine Learning. Retrieved April 13, 2024, from https://www.researchgate.net/publication/364646628_Machine_Learning.

[13] C. K. Roy, J. R. Cordy. 2008. A survey on software clone detection research. *Queen's School of Computing Technical Report* (2008).

[14] H. Sajnani, S. Sinha, C. V. Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. *Proceedings of the 8th International Conference on Software Engineering (ICSE)* (2016).

[15] ScienceDirect. n.d. Multilayer Perceptron. Retrieved October 2, 2024, from https://www.sciencedirect.com/topics/veterinary-science-and-veterinary-medicine/multilayer-perceptron.

[16] ScienceDirect. 2018. Recent Developments in Software Cloning. Retrieved May 5, 2024, from https://www.sciencedirect.com/science/article/pii/S1877050918308123.

[17] Shiksha. n.d. Understanding Multilayer Perceptron (MLP) Neural Networks. Retrieved October 2, 2024, from https://www.shiksha.com/online-courses/articles/understanding-multilayer-perceptron-mlp-neural-networks.

[18] Shiksha. n.d. Deep Learning Tutorial. Retrieved October 3, 2024, from https://www.simplilearn.com/tutorials/deep-learning-tutorial/multilayer-perceptron.

[19] G. Swinnen. 2000-2012. *Apprendre à programmer avec Python 3*. Paris: Creative Commons.

[20] TechTarget. n.d. Information Systems (IS). Retrieved May 17, 2024, from https://www.techtarget.com/whatis/definition/IS-information-system-or-information-services.

[21] M. S. Uddin. 2014. Dealing with Clones in Software: A Practical Approach. *Computer Science*, February.