INVESTIGATING THE EFFECT OF CLUSTER-BASED PREPROCESSING ON SOURCE-TO-SOURCE CODE TRANSLATION

INVESTIGATING THE EFFECT OF CLUSTER-BASED PREPROCESSING ON SOURCE-TO-SOURCE CODE TRANSLATION

BY AKILA LOGANATHAN, B.Eng.

A THESIS SUBMITTED

TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

MASTER OF SCIENCE (2021) McMaster University
(Department of Computing and Software) Hamilton, Ontario, Canada

TITLE: Investigating the Effect of Cluster-Based Preprocessing on Source-to-Source Code Translation

AUTHOR: Akila Loganathan

B.Eng., (Computer Science Engineering)

Anna university, Chennai, India

SUPERVISOR: Dr. Richard Paige

NUMBER OF PAGES: ix, 85

To my Bava, Athamma, Suresh Babai and Amma, Naana

Abstract

Numerous programming languages have been proposed over the last 60 years. Programming languages, like other software systems, can become obsolete: their compilers, virtual machines, interpreters and libraries are no longer fit for purpose. As such, programs written using obsolete programming languages may need to be *modernized*, relying instead on modern languages, libraries and tools. Modernization is both a technical and social process; in this thesis, we focus on the technical aspects of modernization, particularly software migration, wherein a program written in one programming language is transformed into an equivalent or similar program written in a different language. Migration happens because many software systems that were developed decades ago can no longer be maintained and need to be overhauled to make it possible to implement new processes that can take advantage of new technologies recently developed.

Migrating an existing codebase to a more efficient and modern programming language is often expensive, and there are different types of risks involved; for example, many functionalities may not be implemented properly after migration, i.e., the migration is inaccurate; or concerns for code quality may not be considered until the end of the migration; and for large code bases, the migration process may be slow, and may demand substantial resources to implement. Recent advancements in Artificial Intelligence in natural language translation have been widely accepted but their application to programming language translation have been limited due to the scarcity of parallel data (i.e., the collection of equivalent phrases in source language and their translations in a target language). This thesis explores a preliminary investigation into the use of unsupervised

learning methods – specifically, a newly proposed K-Means clustering approach for preprocessing and analyzing the source code – prior to rule-based code translation. The thesis investigates such a process both generally and abstractly, and specifically, in the context of a concrete migration from C++ to Java. The thesis also presents a test set for evaluating such an approach, based on open source, which can be used as a general resource for both validating migration approaches and assessing their performance. The test results and our experiments show that our proposed translation approach based on unsupervised machine learning for preprocessing has a very good translation accuracy score of 77.89% and 81.34% when compared against an alternative approach with accuracy score of 33.24% and 59.96%, and also when compared with rule-based translation that excludes the preprocessing step with accuracy score 37.39% and 41.26%.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor Dr. Richard Paige, whose expert advice and guidance motivated me to achieve excellence in my work. I would like to thank him for his valuable inputs throughout the course of my research.

I am most thankful and grateful to my supervisory committee Dr. Alan Wassyng, and Dr. Spencer Smith for all the time they took to read this thesis.

I am thankful to the Department of Computing and Software, for providing me with an opportunity to pursue my research at McMaster University.

I really feel grateful to my friends who shared their experiences with me and guided me throughout this thesis journey.

Last, but by no means least, I would like to thank my husband Raagul for his support and encouragement, as well as Chandramouli, my mother-in-law for her unequivocal support throughout this research, and my uncle Suresh for supporting me throughout my educational journey. You people have always been my motivation to go further in life. I really wanted to thank my parents for showering me with their blessings.

CONTENTS

Table of Figures

Table of Tables

CHAPTER 1: Introduction

Software can become obsolete: it may be written in a programming language that is no longer supported by a vendor; it may make use of inefficient or outdated libraries; it may run on outdated hardware that can no longer be repaired. But there may still be requirements for obsolete software to execute and be used by diverse stakeholders. As such, there are requirements for supporting software *modernization*, the process of transforming obsolete software into software that is behaviorally *equivalent* yet executes in a modern environment. Software modernization is both a social and technical process: the new version of the software must satisfy relevant stakeholders and be useful and usable within existing processes. But it also involves deep technical challenges, particular at scale, when large code bases, programs and documentation must be updated.

Software migration is the technical process of transferring software data, accounts and functionality between operating environments; it may also involve the porting of a legacy software system to a modern computer programming language [Smartsheet, 2017]. Software migration happens increasingly often [Derras et al, 2021] because many software systems that were developed decades ago need to be overhauled to make it possible to implement new processes that can take advantage of technologies recently developed. With rapid technological enhancement, companies may need to move their software from one platform to another platform [Mustafa et al, 2017] as the software systems-built decades ago face technology stasis and can no longer be easily updated or improved. Moreover, in some organizations, processing needs may vary between departments, and software may have been developed without considering the challenges of use in different contexts. For instance, a certain department may need more storage space in the system, while another

department may need more processing speed. We may need to be able to modernize existing systems so that it works under both sets of conditions while producing optimum output for each [Satani et al, 2020].

The fast-changing market trends and the constantly growing business and customer needs increase the pressure for technology innovations in hardware and software-based industries. Developers make substantial efforts to adapt their software and hardware products to the latest processes, technologies and materials to take advantage of new features that may improve security, operational efficiency, performance etc. However, even minor design or implementation changes often require rerunning all necessary verification, validation, and assurance processes to establish that the updated system still meets its functional and non-functional requirements while complying with national and international regulations and standards. This can be a long and costly process that can cause these systems to fall into "technology stagnation". Due to the challenges associated with technology stagnation, complex systems within critical application domains are likely to face the problem of obsolescence [Paige et al, 2018].

Source code translation is an important approach used in software migration, and hence modernization. It is a technical process that is used to migrate legacy code in one programming language to a different language [Chen et al, 2018]. This may include migrating code from a legacy language such as PL/I to a more modern language such as C#, to migration between versions of programming languages (e.g., C++98 to C++17). Migrating code from a legacy programming language such as COBOL or APL, to a modern

alternative like Java or C++, is a complex, time-consuming process that necessitates expertise in both the source and target languages, as well as the libraries and execution environments (including virtual machines) for each. Code translation processes and tools are costly to both build and use because the ultimate translation process is time-consuming, and time-consuming processes can be very expensive [Lachaux et al, 2020]. For example, the Commonwealth Bank of Australia paid $750 million and took five years to convert its platforms from COBOL to Java using program translation.

As development techniques, paradigms and platforms typically evolve far more quickly than domain applications, software modernization and migration is a constant challenge to software engineers [Fleurey et al, 2007]. There are different types of risks involved during migration. One example is that after migration, functionalities or features are not implemented identically [Mustafa et al, 2017]; another is that the performance of the migrated code differs from the original (note that code that executes faster is not necessarily desirable, especially when timing constraints must be satisfied). Typically, in program translation any concerns for code quality are also not considered until the end of the translation, to simplify an already complex process. It has been argued that prioritizing quality in software migration above many other issues has many benefits [Fabry et al, 2019]. By performing migration activities correctly some of these risks might be reduced or mitigated. Due to the absence of resources to support migration - such as workforce, time, and budget - software migration is often not performed in an optimized way.

1.1 Motivation

As mentioned earlier, migrating source code from one platform or programming language to another is generally very expensive, and usually requires substantial programmer expertise and knowledge of existing systems. Traditionally, software migration is supported and implemented in a rule-based manner: rules are specified indicating how concepts and properties in a source language are to be translated into concepts and properties in a target. The rules may act as the specification of actions to be carried out by a programmer or software engineer; or may be executable and used by an automated program translator. The rules are used to document and support either a wholesale rewrite of the code base in a new language or for a new hardware platform, or support the execution of a program translator, the results of which may be manually adjusted. Rule-based translation relies on millions of bilingual dictionaries for each language pair; these dictionaries specify the concepts in each programming language, very much like a bilingual dictionary does for natural languages such as English and German. The software uses these complex rule sets and dictionaries and transfers the grammatical structure and semantics of the source language into the target language [Systran, 2021].

A compiler's purpose is translation, i.e., converting a high-level language into an intermediate or low-level language, such as an abstract computer language or an assembly-like language [Thain et al, 2020]. The compiler's correctness is ensured by the correctness of many such translations. Translations are used to compare the expressiveness of various languages or programming models (and to obtain corresponding expressiveness results). Unfortunately, creating a translator is difficult in practice because different languages have

different syntax and semantics, and use different platform APIs and standard library features. The majority of programming language translator tools are currently rule-based, and typically come with restrictions meant to cope with semantic variation points (e.g., different interpretations of short-circuiting Boolean operators in different versions of the C language), or to make the ultimate program translation fully automatable (e.g., by restricting features of the source language that can be translated).

Machine translation (MT) is the process of translating a text from a source language to its counterpart in a target language [John et al, 2014]. The predominant approach to MT is corpus-based. These approaches use large aggregations of parallel data (i.e., the collection of phrases in source language and their translations in target language) as the origin of knowledge. *Statistical* machine translation, which is also referred as SMT, is a type of machine translation that uses predictive algorithms to train a model how to translate text. These models are created from parallel text corpora and used to create the most probable output, based on different bilingual examples. A parallel text is a text specified with its translation, where the pair are used to create a statistical model of translation. Using this already translated text, a statistical model predicts how to translate text. One drawback is that this SMT system needs parallel data, and this can be challenging to find (especially for large programs). Creating these parallel texts is very time-consuming and labor-intensive [Shofner et al, 2021] and can be an error-prone process.

Neural machine translation is not a dramatic conceptual step beyond what has been traditionally done in statistical machine translation. Neural machine translation is

considered a modern approach for corpus-based machine translation. Neural machine translation (NMT) is typically used to translate words from one language to another. NMTs encompass all types of machine translation where an artificial neural network is used to predict a sequence of numbers when provided with a sequence of numbers. In the case of translation, each word in the input sentence is encoded as a number to be translated by the neural network into a resulting sequence of numbers representing the translated target sentence. Each number in the input and output represents a word in the parallel corpus [i.e., the collection of phrases in source language and their translations in target language] and is always encoded and decoded accordingly [Sam et al, 2021]. Again, the drawback is that this needs parallel data, and it can be hard to find such parallel content. Creating these parallel texts can be time-consuming and labor-intensive and error-prone, as noted earlier.

Professional translators who rely on automatic computer translation mechanisms have largely embraced recent advances in Artificial Intelligence in the context of natural language translation, particularly unsupervised learning. Unsupervised learning is a machine learning method that learns patterns and recognize trends from unlabeled data, making it ideally suited for this application: source code (without intervention) by default comes in an unlabelled format and thus any inherent classifications are implicit.

Recent developments in neural machine translation have gained widespread acceptance when applied to natural language, even among experienced translators, who are increasingly relying on automatic machine translation systems [Lachaux et al, 2020]. However, due to the shortage of parallel data in this domain (programming language

translation), the application of neural machine translation to code translation has been limited. We discuss this further in Chapter 2.

The use of unsupervised machine learning methods for the translation of programming languages has been limited so far, owing to a scarcity of parallel data in this domain. By exploiting C++, C, Java, Python, we extended recent techniques in unsupervised machine learning specifically clustering in this research work to preprocess and analyze source code before starting with the translation process (rule-based translation). Our hypothesis was that preprocessing will help us to improve the accuracy of translation (and give a better accuracy score) using the already available methods used by other researchers in this field. As such, our focus in software migration is addressing the challenge of validating the results of the translation.

1.2 Translation of Programming Languages

Programs are the main tool for building computer applications. Various programming languages have been invented to facilitate programmers to develop programs for different applications. At the same time, the variety of different programming languages also introduces a burden when programmers want to combine programs written in different languages together. Therefore, there is a tremendous need to enable program translation between different programming languages. Nowadays, to translate programs between different programming languages, typically programmers manually investigate the correspondence between the grammars of the two languages, then develop a rule-based translator. However, this process can be time consuming and inefficient [Song et al, 2018].

In this thesis, we propose to use unsupervised machine learning methods on the source code within the software translation process. Specifically, we aim to investigate the use of unsupervised machine learning in the *preprocessing phase* of software translation. Our rationale for this is that preprocessing – particularly in terms of clustering related software components, modules and methods – may be used to better manage the translation of complex code bases, reduce the time associated with translation, and improve the ultimate readability and understandability of the migrated source code. This last point in turn may make it easier to validate the results, and this is important especially when working with large code bases: a program translator is effectively a code generator, and generated code can be very unreadable and difficult to manage and understand. A preprocessing approach may be used to help manage these challenges – and thus, we believe, make it easier to validate the results of migration.

1.3 Research Objectives

Professional translators who rely on automatic translation mechanisms have largely embraced recent advances in Neural Machine Translations, which are normally applied to natural languages. Their application to programming language translation is currently very limited, owing to a scarcity of parallel data in this domain. We propose to extend techniques in unsupervised machine learning methods in parts of the translation process in this thesis. Our specific objectives are:

- Support preprocessing, and code analysis in software migration, using unsupervised learning methods, to provide means to validate code translation and enhance migrated code quality.
- To build a component based on unsupervised learning methods to support preprocessing. We chose the K-Means unsupervised clustering approach; a particular novelty with this clustering approach is that it does not require us to change the number of clusters every time for different datasets. We explain the importance of this in later chapters, but informally this means that there are greater opportunities for automating the preprocessing step.
- To demonstrate that the preprocessing step before translation provides better performance (good accuracy score of translations) and requires less human effort to validate the translated code.
- Generate a test set composed of corpus of programs (i.e., Human translated, and Machine translated code) with the help of open-source data to evaluate and check the accuracy of translations done using our proposed framework.

1.4 Organization of Thesis

In the following sections, the proposed translation approach and implementation details is discussed. Chapter 2 presents the literature review and related work where we discuss about syntax directed translations, source to source translations, machine translations etc. In Chapter 3 we present an approach to software migration, based on use of a preprocessing step using unsupervised learning method before rule-based translation in the translation of programming languages. The proposed method involves three phases, preprocessing,

analyzing and translation. We present the method generically, and then describe a concrete instance of the method for C++ to Java transformation. In Chapter 4 we describe how we have gathered data for creation of a corpus of test data for evaluating translations and present our experiments (and experimental method/setup), which include performance and accuracy studies on a number of end-to-end translations. This chapter also presents a comparison between the approach proposed in this thesis with an existing translator, as well as a rule-based approach that excludes preprocessing. We analyze the results and make several observations. Chapter 5 gives a summary of the dissertation and discusses future research directions.

1.5 Summary

Programming language translators are primarily used for interoperability and to port codebases written in a deprecated or obsolete language to a modern one. The overall translation process is time-consuming and necessitates knowledge of both the source and target languages, making code-translation projects costly. Although neural models outperform rule-based counterparts in natural language translation, their applications to transcompilation have been very less and limited due to a lack of parallel data in this domain.

We propose an approach where unsupervised learning methods are used to preprocess, analyze the source code before rule-based translation in the translation of programming languages. Specifically, we use clustering approaches from the unsupervised learning methods. We use this clustering approach on the source code prior to translation in such a way that each cluster share a similar semantic meaning.

The context of this work is preprocessing the source code and analyzing the code before translation. Our work is divided into three phases. First, we use the K-Means algorithm to create global clustering of the entire source data where each cluster share a similar semantic meaning, and this helps in preprocessing. Second, we analyze the code based on clusters formed earlier and at third phase we perform program translation. We generalized the first two phases of this approach to various programming languages like C, C++, Java, Python and at the third phase we built a rule-based translator for demonstrating the program translation from C++ to Java where datatypes, functions, operators, selection statements, iteration statements, classes, input and output statements, main function are translated to equivalent in Java. We also built a test set composed of corpus of programs (i.e., Human translated, and Machine translated code) with the help of open-source data, to be used to evaluate and check the efficiency of translations done using our proposed approach.

CHAPTER 2: Literature Review

This chapter discusses the concepts, problems, and methodologies used in programming language translation in the literature, and presents a comparative analysis that categorizes the different technical and conceptual approaches to program translation. The overview of the literature also attempts to identify some of the properties that need to be checked when translation is carried out, such as correctness, timing, and the overall performance of the translator.

2.1 Related Work

This section summarizes and analyzes the key literature related to software migration and program translation.

2.1.1 Syntax directed Translation

Several studies have investigated different approaches to translating programming languages based on definitions and manipulation of the syntax of the source (and sometimes target) language. A syntax directed translation is typically based on a context free grammar [Jose et al, 2012]. A context-free grammar is a set of rules used to generate all possible patterns of strings in each (programming) language [Gerald et al, 2012]. Parsing is a method of analyzing a sentence (i.e., set of strings) to determine its structure based on the grammar, and to generate a parse tree showing their syntactic relation to each other. If *w* is a word in language *L* generated by context free grammar *G*, one obtains a particular syntax directed translation of *w* by constructing a parse tree for *w* in *G* [Ullman et al, 2007]. A syntax-directed translator typically consists of two components, a source language parser and a recursive converter which is usually modeled as a top-down tree-to-

string translator [Gecseg et al, 1984]. For instance, [Irons et al, 1961] developed a syntax-directed translation model, where the source-language input is first parsed into a parse-tree, which is then recursively converted into a string in the target-language. [Pyster et al, 1978] used semantic-syntax-directed translator that maps parse trees of source sentences to parse trees of sentences in the target language as a function of both the syntactic structure of the source parse tree and the values of attributes of its nodes. [Chiang et al, 2010] presents a syntax-directed translation approach that uses an implicit definition of formal syntax, that is syntactic structures for the source and target that are discovered on the basis of a bilingual corpus, but without resort to an externally motivated parser. There are also approaches such as [Yamada et al, 2006] that use an external parser on the target only, or other approaches such as [Quirk et al, 2005] on the source only, [Hasan et al, 2005] that only uses a parser on the target and attempts to improve the accuracy of the translation produced. Finally, there are approaches such as [Cowan et al, 2006] that use external parsers both on the source and on the target. But syntax-directed translations are expensive [Marc et al, 2004] as we must build parsers and grammars in turn which are easily prone to errors and difficult to maintain etc.

2.1.2 Source-to-Source Translation

A transcompiler is a source-to-source translator that transforms between programming languages at equivalent (or very similar) abstraction levels [Ackerman et al, 2016] i.e., from high level programming language (e.g., C++ or Java) to another high-level programming language (e.g., Python). A transcompiler is different from traditional compilers that build executables by translating source code from a high-level programming

language (e.g., C, C++) to a lower-level programming language (e.g., assembly language) [Taylor et al, 2019].

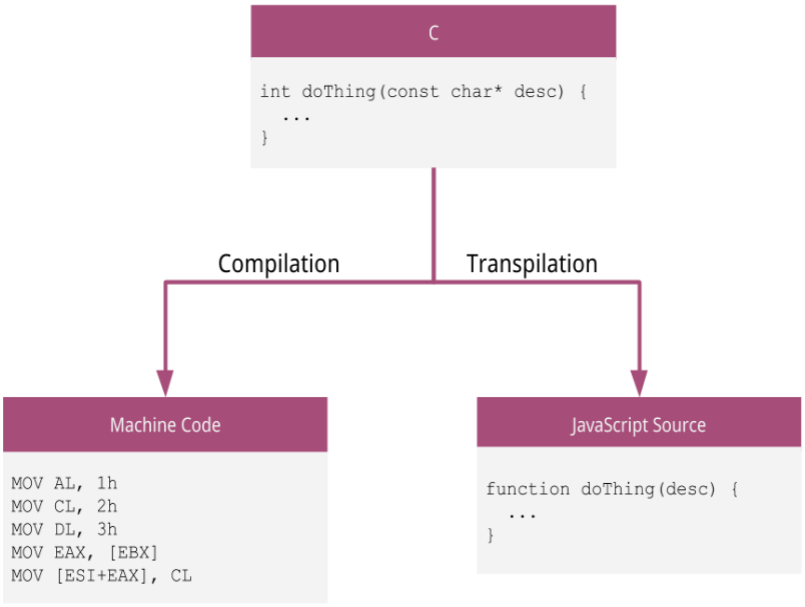

Figure 1: Transcompiler and Traditional Compiler

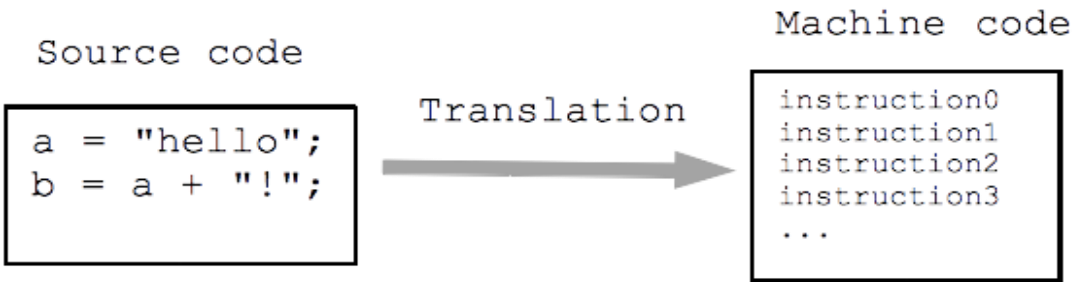

Figure 2: Traditional Compiler

Originally, transcompilers were created to translate source code from one platform to another (for example, converting source code intended for the Intel 8080 CPU to make it compatible with the Intel 8086 CPU). Another example was earlier implementations of the C++ language, which transcompiled C++ source code into C source code (removing object-oriented features such as inheritance – relying on the delegator pattern – and dynamic binding – relying on virtual function tables). More recently, new languages have been developed (e.g., CoffeeScript, TypeScript, Dart, Haxe) along with dedicated transcompilers that convert them into a popular or omnipresent languages (e.g., JavaScript). Using a transcompiler and manually tweaking the output source code instead of rewriting the entire codebase for an application from scratch might be a faster and less expensive approach [Lachaux et al, 2020].

In theory, it is always feasible to translate source code from one Turing-complete language to another, but it can be challenging in practice since different languages have distinct syntaxes and use distinct platform APIs and standard library methods. The majority of transcompilation tools are now rule-based; they tokenize the input source code and transform it to an Abstract Syntax Tree (AST) on which custom rewriting rules are applied. They can take a long time to make and need considerable knowledge of both the source and target languages [Lample et al, 2017], including both their syntax and their semantics.

A more recent development in source-to-source translation relies on neural network-based techniques; these are generally called neural machine translation and have garnered uptake in natural language processing applications. Their application to programming language translations have been limited due to the scarcity of parallel data in this domain [Roziere et al, 2020]. Parallel data, recall, is a corpus of data with equivalent programs written in

two (or more) programming languages, so that this corpus can act as the basis for testing a transcompiler. We discuss these approaches in more detail in the next section.

There are a number of existing open source translator tools that translates programs from C++ to Java; one is the C2J translator [Laffra, 2000], while another is the commercial C++ to Java rule-based converter [Tangible, 2018] etc. Later, in Chapter 4, we will be comparing our translation approach specifically with Tangible Solutions converter as it is a rule-based translator which works similarly to our approach, i.e., based on the rules written by human experts. It also has a free edition that we can use to test and gather results.

2.1.3 Machine Translation approaches

Machine translation (MT) is the task of translating a text from a source language to its counterpart in a target language; MT is normally applied to natural language texts [John et al, 2014], e.g., to support translation from English to German. The predominant approach to MT is corpus-based (i.e., parallel data). These approaches use large aggregations of parallel data (i.e., the collection of phrases in source language and their translations in target language) as the origin of knowledge. MT approach in usually applied to natural language processing. Several researchers have explored the possibilities of using machine translation to translate programming languages, for example, [Nguyen et al, 2013] used a Java-C# parallel corpus (i.e., the collection of phrases in source language and their translations in target language) to train a Phrase-Based Statistical Machine Translation (PBSMT) model. They built their dataset by combining the Java and C# implementations of two open-source projects, Lucene and db4o. [Chen et al, 2019] used the Java-C# dataset

of [Nguyen et al, 2013], to use tree-to-tree neural networks to decode JavaScript. They use a transcompiler to build a CoffeeScript-JavaScript parallel corpus. [Karaivanov et al, 2014] created a program to mine parallel corpus from ported open-source applications in a similar way. [Aggarwal et al, 2015] trained Moses on a Python 2 to Python 3 parallel corpus created with 2to3, a Python library developed to port Python 2 code to Python 3. [Chen et al, 2019] utilised the Nguyen et al. Java-C# dataset to translate code using tree-to-tree neural networks. They also employ a transcompiler to construct a CoffeeScript-JavaScript parallel corpus. [Oda et al, 2015] trained a PBSMT model to generate pseudo-code. To create a training set, they hired programmers to write the pseudo-code of existing Python functions. Barone and Sennrich built a corpus of Python functions with their docstrings from open-source GitHub repositories. They showed that a neural machine translation model could be used to map functions to their associated docstrings, and vice versa. Similarly, [Hu et al, 2018] proposed a neural approach, DeepCom, to automatically generate code comments for Java methods.

All of these methods rely on the presence of parallel data; finding such a large parallel corpus for training or building is challenging and expensive, requiring substantial human intervention. Furthermore, several studies predominantly use the BLEU score to assess the quality of their translations. The BLEU score is the most commonly used metric to evaluate machine translations [Hindel et al, Barone et al, Vechev et al, Pharaoh et al]. Prior to the development and acceptance of the BLEU score, researchers relied on human evaluation by experts in both source and target languages. These experts would place the original sentence with its machine translation side-by-side and they would rate the accuracy of translation. This process is, of course, labor intensive, expensive and error prone. BLEU,

by comparison, helps in automatically evaluating the performance of a machine translation by comparing the text in translated code to reference translations (human generated translations) line by line. But the BLEU score might be flawed because the generated code might be a readable or good translation despite differing from the reference: in general, the text in translated code and the reference code might differ by extra spaces and even an empty line. Since the comparison takes place line by line (i.e., line 1 in translated code is compared to the line 1 in reference code and so on) if line 1 is empty in either of the translated code or reference code the BLEU score for that line will be very low and this will in turn affect the score of all other lines. Issues pertaining to the automatic evaluation of translated code are addressed in the validation section of our work.

2.1.4. Unsupervised Machine Translation

The methods described in Section 2.1.3 are supervised, and as such they rely on the existence of parallel data (i.e., a corpus of equivalent program examples in source and target programming languages). Unsupervised learning, by contrast, is a machine learning method that learns patterns and recognize trends from unlabeled data [Sanatan, 2017], hence it is ideally suited for application to program translation, though there are issues.

In natural language processing, recent advances in neural machine translation have been widely accepted [Lachaux et al, 2020]; it may be that we can learn lessons from successes in neural machine translation and unsupervised learning as applied to natural languages and apply them to programming languages. Machine translation systems can in principle attain near-human performance in certain languages, but their success is heavily reliant on

the availability of enormous numbers of parallel phrases, limiting its application to the vast majority of language pairings [Conneau et al, 2018]. Through recent breakthroughs in deep learning, machine translation has lately reached amazing results. Many attempts have been made to extend these accomplishments to low-resource language pairings, but this would need tens of thousands of parallel phrases. The existing learning algorithms' reliance on vast parallel corpora is a key difficulty. Unfortunately, the great majority of language pairings have very little, if any, parallel data: to make MT more generally applicable, learning methods must better exploit monolingual data. There is a huge body of literature on using monolingual data to improve translation performance when minimal supervision is available [Lample et al, 2016] and [Denoyer et al, 2018]. [Karaivanov et al, 2014], [Nguyen et al, 2013] and Aggarwal et al, 2015] built few a parallel corpuses to train their models by hiring a few programmers which was really very expensive and time consuming. Creating a parallel corpus for testing is much less expensive and it can be done using the open-source data available on web. Data scarcity is one among the many challenges for training a Neural Machine Translation (NMT) model, and this challenge is exacerbated for programming languages, as there are only a few corpuses of equivalent programs available for training and testing.

Despite the progress made for a high-resource language, where parallel data is readily available (for instance: English-German pair), most languages are characterized by the absence of parallel data to train an NMT system. For most languages, parallel resources are rare or nonexistent. Since creating parallel corpora (i.e., the collection of phrases in source language and their translations in target language) for training is not realistic where creating a small parallel corpus for evaluation is already challenging and time consuming

[Koehn et al, 2007]. In this thesis we are interested in investigating the use of unsupervised learning techniques - specifically clustering, which is generally used for the analysis of the data set, to find insightful data among huge data sets and draw inferences from it. They are many types of clustering algorithms. For instance, hierarchical clustering follows top-down, and bottom-up approaches were in bottom-up approach, each data point acts as a cluster initially, and then it groups the clusters one by one, whereas in top-down approach we start off with all the points into one cluster and divides them to create more clusters [Rohit et al, 2019]. *Expectation maximization* is an iterative process with two steps, expectation and maximization. Expectation assigns each data point to a cluster probabilistically. Maximization updates the parameters for each cluster. It estimates the mean and standard deviations for each cluster to maximize the likelihood of observed data, DBSCAN is a density-based clustering algorithm, it groups datapoints that are close to each other i.e., with many nearby neighbors and the data points with low nearby neighbors are considered outliers [Ram et al, 2010]. K-Means clustering partitions the data points into *k* clusters based on the distance metric. The value of *k* is to be defined by the user. The data point which is closest to the centroid of the cluster gets assigned to that cluster [Kasthuri et al, 2020] etc.; there are many more clustering algorithms too. In this thesis will be carrying out experiments on our source code datasets using these algorithms as the basis of our preprocessing and analysis phases in the next chapter.

2.2. Comparative Analysis

There have been a number of approaches presented in the literature for supporting source code translation between high-level programming languages. These approaches have a

variety of motivations – e.g., to more adequately reuse existing code, to enhance developer productivity when migrating software applications, to reduce the number of bugs in migrated code. All researchers note the difficulties in building automated translation tools, even between languages that conceptually share semantic commonalities (e.g., C++ and Java, which both include concepts of classes, inheritance and overriding), in part because of subtle semantic differences between the languages. Table 1 attempts to summarize the research objectives and outcomes of a substantial part of the past work on programming translation. We present this in order to attempt to give an overview of the state of the art and to highlight gaps that exist in the process of automating parts of the translation process.

Authors	**Research Objectives**	**Outcomes**
Qiu, L. (1999).	Discuss the subtleties of automating the majority of the translation from C to C++, as well as the challenges experienced. Also, discussed talks about the experience of manually porting Java programs to C++, and identifies some of the issues and difficulties in automating this translation process.	Designing devices to permit high level specification of translation rules and adequately incorporate human interaction is a generic approach to any language translation issue, which is an intriguing exploration issue to investigate.

Papineni et al. (2002)	Proposed a method (BLEU) of automatic machine translation assessment that is speedy, cheap, and language-independent, that corresponds exceptionally with human assessment, and that has minimal negligible expense per run.	BLEU's strength is that it associates exceptionally with human decisions by averaging out individual sentence judgment errors over a test corpus than endeavouring to divine the specific human judgment for each sentence: amount prompts quality.
Fleurey et al. (2007)	Introduces an original model-driven process created at Sodifrance for migration of software.	They showed that regardless of whether the process is not completely automated and needs manual adaptation from one project to the other as well as manual execution of certain parts of the final application.
Fuhr et al. (2013)	Designed and execute a Service-oriented Architecture (SOA) and assesses abilities to expand the method by model-driven software migration methods.	The assessment of the extensions has shown that these can possibly support SOMA in future SOA design by utilizing legacy

		frameworks dealing with in model-driven manner. Regardless of whether parts of the methods were not feasible in practice, they have proved to be very useful in designing and realizing a SOA.
Oda et al. (2015)	Proposed a method to automatically generate pseudo-code from source code, explicitly employing the statistical machine translation framework.	These methods produce grammatically correct pseudo-code for the python to English and English to Japanese programming language pair and recognized that proposing pseudo code with source code enhances the code comprehension of programmers for unfamiliar programming languages.
Grieger et al. (2016)	Introduce a situational method engineering framework to direct the development of model-driven	They assessed the framework by an industrial project in which the legacy system was

	migration techniques by collecting predefined buildings blocks.	migrated from the domain of real estate to a new environment.
Vavrova et al. (2017)	Developed the primary level 4 grammar of Python, which is equipped for parsing both Python 2.x and Python 3.x code. It is a consequence of a significant grammar programming effort, merging previously existing grammars (made for various dialects and written in various notations) and language documentation.	The detected design defect density in Python was compared to results that DECOR yielded for Java. Generally, the density measured in Python (average 6.07 design defects per 10,000 lines of code) was slightly less than the density measured in Java (average 8.37 design defects per 10,000 lines of code).
Shoaib et al. (2017)	Investigated the process, activities, challenges and their solutions for the software migration are framework.	A typical system for software migration was helpful for effective understanding of the structure and size of the data, which will decrease the migration, also it will improve the appropriateness and effect in real business environment.

Chen et al. (2018)	A tree-to-tree neural network to translate a source tree into a target one, in which a sub-tree of the source tree is translated into the corresponding target sub-tree at each step	This proposed method will improve the past state-of-the-art program translation approaches by a margin of 20 points on the translation of real-world objects.
Candel et al. (2019)	A software process to execute a model-driven re-engineering. This process incorporates a TDD-like approach to incrementally develop model transformations with three sorts of validations for the produced code.	Designed and implemented a re-engineering methodology for a migration of Oracle Forms code to MVC architecture dependent on Java structures.
Wlodarski et al. (2019)	Discussed a modernisation project of mBank, a big Polish bank, were bad smell detection, elimination, automated testing and refactoring to ensure and maintain end results.	They concluded enhancing the quality of the code prior to migrating it to another language was therefore pivotal to a successful migration effort.

Table 1: Comparative study

2.3 Summary

In this chapter, we have presented a literature review related to syntax-directed translation approaches, source-to-source translation approaches and machine translation approaches in the domain of source code translation. Through recent breakthroughs in deep learning, machine translation has lately achieved many significant results in natural language translation, but they rely on the existence of vast parallel data. Many attempts have been made to extend these accomplishments to low-resource language pairings, but this would need at least tens of thousands of parallel phrases. The existing learning algorithms reliance on vast parallel corpora is a key difficulty. Unfortunately, the great majority of language pairings have very little, if any, parallel data: to make machine translations more generally applicable, learning methods must better exploit monolingual data. In this thesis, we propose to extend recent techniques in unsupervised machine learning methods to preprocess and analyze the source code using clustering approaches before the code translation process, believing it will help us to improve the accuracy of translation and gives a better accuracy score using the already available metrics used.

In the next chapter, we present an approach that follows three distinct stages in source-to-source translation: a preprocessing phase, a code analysis phase, and finally a code translation phase. After presenting the approach we evaluate it on a number of test cases and analyze the results.

CHAPTER 3: Technical Approach

This chapter presents a novel approach to program translation that relies on both rule-based source-to-source translation and unsupervised machine learning. The novel architecture of the approach distinguishes a *preprocessing phase* from an *analysis phase* and a *rule-based translation phase*. This separation of concerns has a number of consequences.

- It allows solutions for each phase to be varied with a degree of independence. For example, it allows us to experiment with different preprocessing approaches without having to modify (significantly) the source-to-source translation phase.
- It allows us to independently assess the effect of preprocessing on program translation. Previous work on software migration has largely focused on end-to-end performance of the program translator/transcompiler, making it challenging to fully assess where any potential performance bottlenecks may reside. By separating the process into distinct phases, we can more easily evaluate performance and obtain fine-grained performance and effectiveness information.
- Validation of the end results is potentially easier as we can take information from earlier phases (such as preprocessing, analyzing) and use it in constraining what needs to be evaluated in successive phases.

In general, we are interested in understanding the effect that preprocessing in program translation can have on the accuracy, efficacy and performance of program translation. Our hypothesis is that preprocessing of information – particularly clustering of program concepts – can make the translation and migration process faster, and more accurate, which has important implications for automating migration for large programs.

More specifically, this chapter presents an approach that follows three distinct phases in code translation: (i) an unsupervised learning method for preprocessing the source code; (ii) source code analysis using a code analyzer (i.e., to ensure that the syntactic program structure is retained in the translated code when compared with the actual input source code); and (iii) rule-based code translation. The following sections present these phases in detail, with some emphasis on the first phase.

3.1 Proposed Approach

Figure 3 diagrammatically presents the proposed approach.

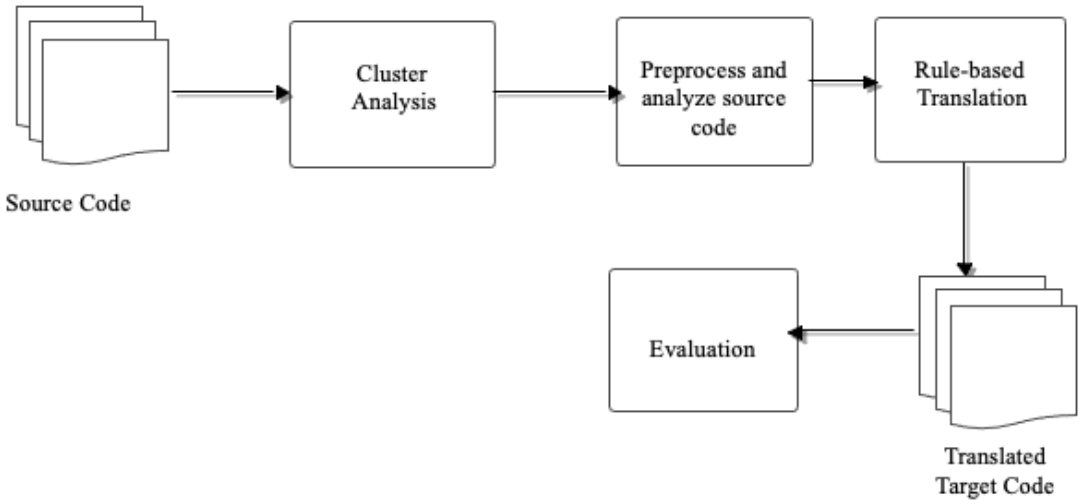

Figure 3: Proposed approach separating preprocessing from rule-based translation phase

In this thesis, we are focusing on preprocessing the source code and analyzing the source code by using unsupervised machine learning techniques (clustering) before code

translation and then using rule-based machine translation (the translation technique based on rules and structural transfer) for translating code. Later in this chapter we will explicitly demonstrate the approach on a concrete translation from C++ to Java.

3.2 Preprocessing

Once again, our general problem is source code translation, and our solution involves decoupling preprocessing from code analysis from code translation. We first discuss the preprocessing phase, which makes use of unsupervised machine learning techniques. Our initial observation – which partly motivated our interest in exploring preprocessing – is that in data analytics, it is commonplace to preprocess data prior to statistical analysis. This helps to remove unwanted or erroneous data (in a process typically called "data cleaning"), which consequently allows the user to have a dataset to contain more valuable information. We hypothesize that this approach may prove to be valuable – in terms of improving quality – for machine translation of source code, too. In software modernization projects prioritizing code quality above many other issues has many benefits; arguably, removal of unwanted or unnecessary code before migration can help in improving code quality [Fabry et al, 2019]. With this thought, we decided to experiment with a preprocessing step on the source code before sending it for code translation. In particular, our preprocessor focuses on removal of unwanted lines of code for the purposes of migration, while also clustering the language features used in the source program. For example, in the concrete translation instance presented later in this chapter, and evaluated in detail in Chapter 4, we are translating the source code from C++ to Java. In such a translation, the header files,

comment lines, and empty lines are not useful in the translation process, and the preprocessor removes these lines of code before translation.

3.2.1 Choosing Clustering Algorithms

Clustering is a division of data into groups of similar objects. In data analytics and machine learning, clustering is a task of dividing the data sets into a certain number of clusters in such a manner that the data points belonging to a cluster have similar characteristics. Clusters are nothing but the grouping of data points such that the distance between the data points within the clusters is minimal [Rohit et al, 2017].

An extensive investigation was done to evaluate different clustering algorithms on the source code written in programming languages like C, C++, Java, Python. For the purposes of this thesis, we finally chose an open-source software framework, WEKA, the Waikato Environment for Knowledge Analysis [Frank et al, 2016]. WEKA is public domain open-source software that provides tools for implementation of several machine learning algorithms to enable data analytics.

The reasons behind choosing WEKA are as follows.

- It is the most widely used software system for implementing different data clustering algorithms.
- It is an expressive and full-featured system that makes it straightforward to implement a wide variety of data clustering algorithm.
- It implements the algorithms that we have chosen to conduct our experiments, and these algorithms in turn are commonly used in the machine learning community, thus

potentially making it easier to conduct experiments and compare the results against existing baselines.

In this thesis we have chosen to use the K-Means algorithm to preprocess the source code. But before arriving at a decision to use K-Means we did some experimental work analyzing other clustering algorithms to determine which was most likely to provide useful results for source-to-source translation. Four Clustering algorithms were considered for experimentation: namely, K-Means clustering, Hierarchical clustering, DBSCAN clustering, and EM clustering. In the literature review chapter of this thesis, we have briefly explained these algorithms. The general reason for considering these algorithms is their widespread use. The chosen clustering algorithms usually work with numerical datasets and mixed datasets (text and numbers) but doesn't work well without numerical data [Manoj et al, 2016]. Therefore, we can't cluster the source code directly without numbers. For the purposes of clustering in this thesis we exploit the linguistic information found in the input source code, namely the use of syntactic language features such as classes, conditions, loops etc.; for each linguistic feature we assign numerical index values to build the dataset for each input source code to make it suitable for clustering and we refer to them as data points. We investigated and did several experiments using those chosen clustering algorithms on the source code dataset and they are described below.

The processing time for four different algorithms to cluster different source code datasets is presented below in table 2-4: We experimented with each of these clustering algorithms (namely Hierarchical, DBSCAN, K-Means and EM) firstly on a source code with 1480 lines. As shown in the experimental results summarized in Table 2, we observed that all these algorithms effectively clustering the source code, but each was taking different

processing times to return the clustered results. Specifically, DBSCAN returned the clusters in 305ms, Hierarchical in 534ms, K-Means in 20ms, and EM 3570ms. From these results we also observe that K-Means clustering takes less processing time compared to other algorithms. We repeated the experiment and applied the clustering algorithms on a smaller source code with 158 lines to check the processing time of these algorithm again and these results are presented in Table 3.

Clustering Algorithm	Hierarchical	EM	DBSCAN	K-Means
Number of Lines	1480	1480	1480	1480
Processing Time (Milliseconds)	534	3570	305	20
Number of lines (After Preprocessing)	1395	1395	1395	1395

Table 2: Processing time for clustering the source code with 1480 lines.

As we can observe in Table 3, the ranking of the algorithms does not change: we observe that DBSCAN returned the clusters in 72ms, Hierarchical in 88ms, K-Means in 2ms and EM in 211ms. We repeated the experiment a final time on an even smaller source code with 25 lines to check the processing time of these algorithm and the results are presented in Table 4.

Clustering Algorithm	Hierarchical	EM	DBSCAN	K-Means
Number of Lines	158	158	158	158
Processing Time (Milliseconds)	88	211	72	2
Number of lines (After Preprocessing)	117	117	117	117

Table 3: Processing time for clustering the source code with 148 lines.

As we can see in Table 4, the ordering does not change once again: the processing times are: DBSCAN returned the clusters in 74ms, Hierarchical in 53ms, K-Means in 1ms and EM in 165ms.

Clustering Algorithm	Hierarchical	EM	DBSCAN	K-Means
Number of Lines	25	25	25	25
Processing Time (Milliseconds)	53	165	74	1
Number of lines (After Preprocessing)	18	18	18	18

Table 4 Processing time for clustering the source code with 25 lines.

By taking processing time factors into account and comparing these algorithms, we determine that the K-Means algorithm is best suited for the purposes of preprocessing source code as it has a better processing time when compared to other clustering algorithms. Thus, we are proposing to use the K-Means clustering algorithm as the basis for our preprocessing and code analysis phases. But we faced few challenges while using the general K-Means clustering algorithm hence we modified the K-Means algorithm and proposed the modified K-means clustering algorithm, which are all explained below in section 3.2.3. Before we demonstrate how this is done, we first briefly explain the general K-Means algorithm.

3.2.2 The K-Means Algorithm:

We briefly describe the general working of the K-Means algorithm. K-Means is based on clustering; clustering is a technique for discovering cluster structure in a data set that is portrayed by the best similarity within the same cluster and the greatest dissimilarity between different clusters [Kasthuri et al, 2020]. It is known that the K-Means algorithm is the oldest and most widely used partitional technique which partitions the dataset into *K* clusters based on the similarities of the data where *K* represents the number of groups.

The K-Means algorithm classifies the data into *K* different cluster through an iterative process. The generated clusters of K-Means are independent. The K-Means clustering algorithm works in two different parts. Firstly, it selects a *K*-value, where *K* is the number of clusters, and we also choose *K* centroids. Another part is to consider each data point to the nearest center. After completing the first step then calculate the Euclidean distance between the data point to *K* centroids (i.e., the center for each cluster) [Kasthuri et al, 2020].

Then all the data points are used to create some group. This process will continue until the center of clusters remain unchanged.

The K-Means algorithm is presented below, where in the algorithm

- *K* - number of clusters.
- *D* is the data set which contains *n* data objects.

Step-1: Select *K* data objects from *D* and decide the number of clusters and initial cluster centroids.

Step-2: Calculate the distance between each data object *d* and all *K* cluster centers. Assign data object *d* to the nearest cluster.

Step-3: For each cluster, recalculate the cluster center.

Step-4**:** Repeat Step 2 and Step 3 until the center of clusters remains unchanged.

Step-5**:** Stop when all the data objects have been assigned to clusters and the center of clusters remains unchanged.

Clustering algorithms in general are used to group similar data points together. Data points that are similar are assigned a value that represents the average value of all points in that cluster. If additional data points are collected, they can be compared to the average values of other clusters and assigned to the closest one [Kasthuri et al, 2020].

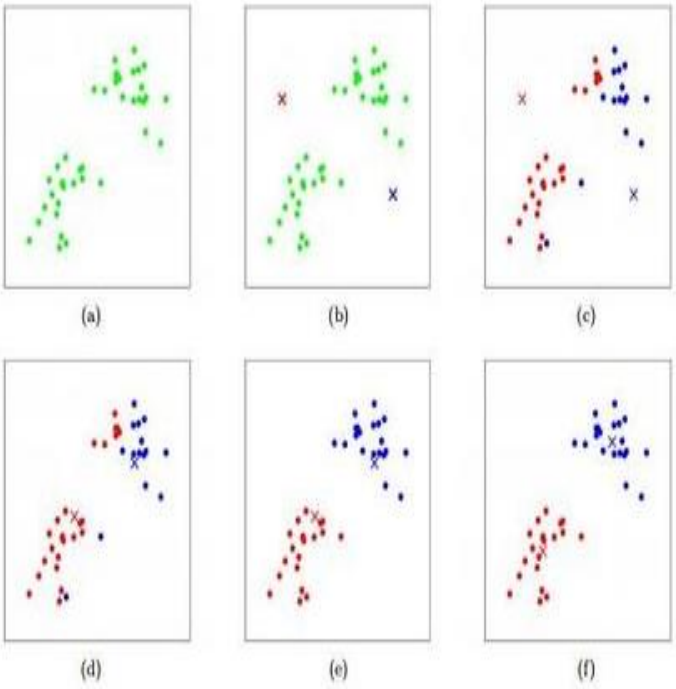

Figure 4: K-Means algorithm illustration [Piech et al, 2012]

In Figure 4, the dots represent data points and the x represents centroids; the above example identifies two clusters. (a) is the original dataset and (b) shows the randomly assigned centroids. (c - f) shows the process of adjusting the centroids until error is minimized.

3.2.3 Proposed K-Means Clustering Approach

K-Means Clustering is an Unsupervised Learning algorithm, which groups an unlabeled dataset into different clusters. Here *K* defines the number of pre-defined clusters that need to be created in the process; so if *K=2*, there will be two clusters, and for *K=3*, there will be three clusters, and so on. It allows us to cluster the data into different groups and provides a convenient way to discover the categories of groups in the unlabeled dataset on

its own without the need for any training. The algorithm takes the unlabeled dataset as input, divides the dataset into *K*-number of clusters, and repeats the process until it does not find the best clusters. Generally, the value of *K* should be predetermined.

The proposed K-Means algorithm in this thesis is built entirely in Java and it works like the general K-Means algorithm already available but differs in the way of choosing the centroids. When clustering, the source code has to be grouped, where classes, loops, functions, conditions, meaningless lines, header files etc., are classified into separate similar groups. From the source language specifications (i.e., the definitions of C++ and Java), we will know the maximum divisions the source code will have based on source language specifications i.e., we know that the source code in a particular language will have classes, functions, loops, conditions, comment lines, header files etc., and this count will be the maximum number of clusters a source code will have. But the problem over here is not all the input source codes we give will have all these divisions mentioned above, so the number of resulting clusters might differ for different source code input, and we might have to keep changing the predetermined number of clusters for different inputs which involves more time and human effort every time. Here the centroid values are chosen based on the distinct elements in the dataset and if they are less than the maximum divisions then the remaining centroid values are set to zero hence there is no need to change the number of clusters every time for different datasets.

Step-1: Take distinct elements in the dataset and choose the cluster centroids.

Step-2: Calculate the distance between each data object d and all K cluster centers.

Step-3: Assign data object d to the nearest cluster.

Step-4: For each cluster j, recalculate the cluster center.

***Step-5*:** Repeat Step 2, 3 and 4 until all the data objects have been assigned to clusters.

With the help of proposed K-Means clustering approach we clustered the source code into various similar groups and then started preprocessing the source code. The source code formed into various clusters using proposed K-Means algorithm is now further processed using the python library Pandas. Pandas is an open-source python library used for data manipulation and analysis which has its own set of commands for data analysis. Using Pandas, we process the clustered source code where we can remove any particular cluster which is not useful for the purposes of migration. While the notion of what clusters are useful or useless for a specific migration usually strongly depends on the target language, one can make some general inferences. In particular, for the purposes of translating e.g., C++ into Java, the header files, comment lines, empty lines etc., in the source language are not going to be translated to the target language and therefore we remove these particular clusters. The clustering table where source code is grouped into clusters is presented in Figure 6.

The preprocessing step was initially done for two programming languages under the same family, i.e., object-oriented programming languages, namely C++ and Java; we also chose these as the source and target languages in the rule-based translation process in order to demonstrate an end-to-end example. We briefly investigated if the preprocessing phase can be generalized to other programming languages: the preprocessed code can be taken as input to any source-to-source translator (not just rule-based translators), and the code analyzer can be helpful in validation (i.e., to ensure that the syntactic program structure is

retained in the translated code when compared with the actual input source code). As part of this we built preprocessors for two more languages namely C and Python but did not investigate their effectiveness or performance any further, as this would have required building further rule-based translators as well. Nevertheless, constructing such new preprocessors was not difficult. We discuss further extensions to this in Chapter 5.

As an illustration, considering the following example. The original source code and the preprocessed source code to find fibonacci numbers is presented in Figure 5. Appendix A presents the original and preprocessed source code for additional programs respectively.

Original Source code

```
//Fibonacci Series using
Recursion
#include<bits/stdc++.h>
using namespace std;

int fib(int n)
{
     if (n <= 1)
          return n;
     return fib(n-1) + fib(n-
2);
}

int main ()
{
     int n = 9;
     cout << fib(n);
     getchar();
     return 0;
}

// This code is contributed
// by Akanksha Rai
```

Preprocessed Source code

```
int fib(int n)
{
if (n <= 1)
return n;
return fib(n-1) + fib(n-
2);
}
int main ()
{
int n = 9;
cout << fib(n);
getchar();
return 0;
}
```

Figure 5: C++ Original and Preprocessed source code

As an illustration in Figure 6, we presented the source code to find fibonacci numbers clustered into various similar groups where header files, functions, conditions etc., grouped into a number of similar clusters.

LineNum	Source Lines	Char Count	Cluster
1	#include	8	Cluster 4
2	using namespace std;	18	Cluster 4
3	int fib(int n)	12	Cluster 3
4	{	1	Cluster 1
5	if (n <= 1)	8	Cluster 2
6	return n;	8	Cluster 1
7	return fib(n-1) + fib(n-2);	24	Cluster 1
8	}	1	Cluster 1
9	int main ()	9	Cluster 3
10	{	1	Cluster 1

Figure 6: C++ Cluster Analysis: Original source code

3.3 Code Analyzer

The code analyzer we present in this thesis is built to help us to analyze the source code based on cluster analysis using the proposed K-Means clustering approach. Specifically, it takes the clusters provided by the preprocessing and automatically generates lists of Classes, Functions, Conditions, Loops. In the previous phase, the entire source code is grouped into clusters and from those clusters we determine the list of Classes, Functions, Conditions, Loops. The translated code - the output from a translation model - can also be

analyzed in the same way based on cluster analysis. Hence the code analyzer will be useful in validating the translated code after the code translation (i.e., to ensure that the syntactic program structure is retained in the translated code when compared with the actual input source code). Code analysis was initially done for two programming languages namely C++ and Java and we also chose this as the source and target languages in the translation process in this thesis. While investigating if the code analyzer phase can be generalized to other programming languages, as the code analyzer can be helpful in validation (i.e., to ensure that the syntactic program structure is retained in the translated code when compared with the actual input source code) we later experimented and built the code analyzer for two more languages namely C and python but did not investigate their effectiveness or performance any further. We discuss further extensions to this in Chapter 5.

For example, the source code in C++ and Java are analyzed based on the clusters and the results are presented in the table 5 – 6. From table 5 we observe that the fibonacci program in C++ has zero classes, loops, and has one condition and two functions. Similarly in table 6, we observe that the fibonacci program in Java has one class, zero loops, one condition and two functions.

From table 5, we observe that number of functions, conditions, loops are same in both the source and translated code and we can see there is a difference in number of classes in C++ and Java code analyzer. We know that in Java, programs are written within a class, hence if the source language (C++) program doesn't contain a class, then a new wrapper class is created in the translation process. While analyzing the translated Java code the class count will increase to one. Thus, we can conclude that the syntactic program structure is (likely) retained in translated code (target language) from the code in the source language. In the

same way we can easily understand the code analyzer of all programs. The tables in appendix B presents the preprocessed and translated code for a few more programs.

File name	Number of classes	Number of functions	Number of conditions	Number of loops
fibonacci.cpp	0	2	1	0
fibonacci.java	1	2	1	0

Table 5: C++ and Java Code Analyzer

3.4 Rule-based Translation using Preprocessed source code

We have come through the preprocessing and code analysis phases. The original source code has gone through these two phases and now we have a preprocessed and analyzed source code as input to our rule-based translator. Preprocessing and Code Analysis was initially done for two programming languages under the same family i.e., C++ and Java. We also chose this as the source and target languages for the rule-based code translation phase. Given an input in some source language, a Rule based machine translation (RBMT) system generates the output in some target language based on morphological syntactic information and set of rules. In a rule-based machine translation system the input is first analyzed morphologically and syntactically to obtain the structural information governing the ordered use of appropriate use words and symbols for writing code in a particular programming language. This information is then refined to a more abstract level emphasizing the parts relevant for translation and ignoring other types of information.

In this thesis with respect to the code translator phase we built a rule-based translator to translate programs from C++ to Java where datatypes, functions, operators, selection statements, iteration statements, classes, input and output statements, main function are translated to equivalent in Java, as we were running out of the limited time, we didn't look into building more on the translator to handle arrays, vectors, pointers, inheritance in detail and much more. The overall translation process in this thesis is presented in figure 7.

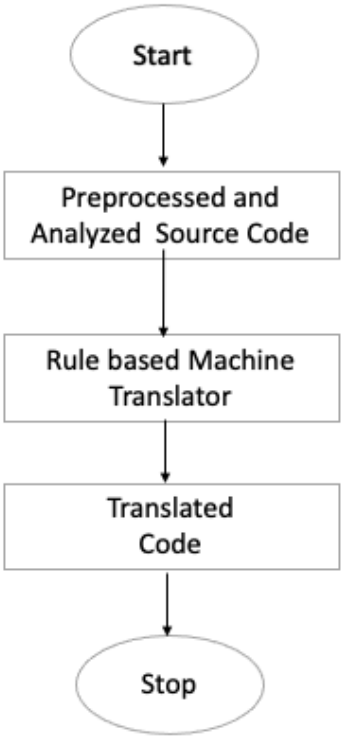

Figure 7: Translation Process

The code translator in this thesis follows the *transfer-based* approach of rule-based machine translation which works based on the linguistic rules (i.e., the language rules)

defined by human experts. The translator takes input in a source language and transfers it to the target language based on linguistic rules**.**

To begin with the translation, we have the preprocessed and analyzed source code as input to the translator. We analyze the source language input into a transfer structure which abstract away many of the grammatical details of the source language. After analysis, the source language structure is transferred into a corresponding target language structure using the rules defined by humans through extensive string matching with some rearrangement of the target string for conformance with the target language structure. A few rules to translate parts of the program from C++ to Java is presented shortly.

The rule-based translator also handles C++ programs without class structure and wraps them in a nominal class using the filename when translated to Java. More specifically, we calculate the class count of source program from code analyzer and if the class count is zero then we take the filename (Fibonacci.cpp) and spilt it using ".", store it in string array and take the first index of that string array (Fibonacci) and store it under that classname to create a class. The result is that even C++ programs without class and object structures can be translated using this approach.

For instance, the rule to create a class while converting C++ programs without class structure to Java is below:

```
if(classcnt<=0)
{
      out.println("class "+classname);
      out.println("{");
}
```

The standard input and output statements in C++ are converted to Java through string matching, where we rearrange the target string. As another example, the rule to convert an output statement in C++ to Java is as below:

```
if(ac.contains("cout<<"))
{
ac=ac.replaceAll("cout<<","System.out.println(").replac
eAll("<<", "+");
ac=ac.substring(0,ac.length()-1)+");";
System.out.println("Print Here="+ac);
}
```

The main function in C++ is converted to Java through string matching where we rearrange the target string. As a third example, the rule to convert the main function in C++ to Java:

```
if(ac.contains("main")&&ac.contains("(")&&ac.endsWith(")")||ac
.contains("main")&&ac.contains("(")&&ac.endsWith("{"))
          {
               ac="public static void main(String args[])";
          }
```

Through extensive string-matching other parts of the source code is also translated from C++ to Java. Thus, based on the rules the preprocessed source code (C++) given as input to the code translator is then translated to the target language (Java). The translated code is stored separately as machine translated code and it is also analyzed using the code analyzer to ensure that the syntactic program structure is retained in the translated code when compared with the actual input source code.

As an illustration, the actual preprocessed input in the source language and the translated code in the target language using the code translator in this thesis is presented in Fig 8. The figures in appendix C presents the preprocessed and translated code for a few more programs.

Preprocessed input code **Translated code**

```
int fib(int n)
{
if (n <= 1)
return n;
return fib(n-1) +
fib(n-2);
}
int main ()
{
int n = 9;
cout << fib(n);
getchar();
return 0;
}
```

```
class fibonacci
{
int fib(int n)
{
if (n <= 1)
return n;
return fib(n-1) + fib(n-2);
}
public static void
main(String args[])
{
int n = 9;
System.out.println( fib(n));
getchar();
}
}
```

Figure 8: Original and Translated Code

3.5 Summary

The technical implementation of the proposed translation approach for translating from C++ to Java was discussed in this chapter. The setup, gathering data to create test set, evaluating translations produced by the proposed approach, and comparison of the proposed approach with alternative approaches, is to be discussed in the next chapter.

CHAPTER 4: Results

In the previous chapters, we have presented an approach to source-to-source code translation comprising a preprocessing phase (supported by a K-Means clustering approach), a code analysis engine that supports different aspects of validation, and a rule-based machine translator for mapping elements of the source program to elements of the target program. This was described first generically, and then with a specific example of rule-based translation from C++ to Java presented in the last chapter.

In this chapter, we evaluate the approach we presented in Chapter 3. We execute it against number of examples and measure the results based on a selection of metrics – metrics that are standard in the machine learning community. As part of this evaluation, we produce a set of parallel test data suitable for assessing such translations.

We carry out the evaluation as follows. Firstly, we describe the data used, the set-up for the experiments, and the process of gathering of data. Then we describe the generation and analysis of the results. As part of the analysis of results, we compare the approach presented in Chapter 3 against an alternative (commercial and proprietary) translation solution. We also evaluate the effectiveness of the translator both with and without the preprocessing step, in order to more precisely understand the effect of the preprocessor on the translation.

4.1. Setup

The proposed translation approach has been implemented as a web application (as it can be accessed from anywhere without any system requirement issues) in the form of a set of Java server pages which runs on Tomcat server; these were developed using the Eclipse Integrated

Development Environment and exploited a number of existing Java and Apache libraries, specifically Java string similarity and Apache text similarity, for computing translation accuracy scores. The approach makes use of a number of clustering algorithms to support preprocessing. Each clustering algorithm described in this thesis was implemented in eclipse on using the Waikato Environment for Knowledge Analysis (WEKA) library. WEKA is an open-source framework that implements a collection of machine learning algorithms for data analysis tasks. It is available free-for-use under GPL (General Public License). However, the proposed K-Means algorithm, on which our thesis relies, is implemented directly in Java by the author.

As our evaluation relies on high-quality data (in our case – programs with source code), we require support for data cleaning and transformation. To facilitate this from an architectural point of view, we make use of two frameworks. The first of these is Jupyter Notebooks, an open-source web application that is used for data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more. In addition to Jupyter notebooks, we use an open-source Python library called Pandas. Pandas is one of the tools in Machine Learning which is used for data analysis. It has features which are used for exploring, cleaning, and transforming the data. Hence Jupyter notebook and Pandas library was used to preprocess (clean) the source code dataset. We export the clustered source code into an Excel file from our web application that runs on Tomcat server, and then further process it using a set of commands provided by Pandas library in the Jupyter notebook. We then export and use this preprocessed file for code translation.

The following subsections explain the results, comparison and evaluation of our approach and the future work.

4.2. Gathering Data

In this section we explain the data collection process for our experiments. We intended to evaluate our source-to-source translation approach on a concrete translation from C++ to Java. As such, we needed a corpus of equivalent C++ and Java programs that could be used.

GeeksforGeeks is a platform for searching useful articles related to computer science and programming languages. It includes many coding problems and presents solutions in several programming languages in a well-explained and well-written manner. From the GeeksforGeeks repository, we extracted a set of programs in C++ and Java, to create test sets. We particularly wanted programs that would (a) exercise the full extent of our preprocessor, code analyzer and the rule-based translator in terms of coverage of language features; and (b) offered complexity in program logic to challenge the preprocessor and translator.

We have built a concrete rule-based translator to translate programs from C++ to Java where datatypes, functions, operators, selection statements, iteration statements, classes, input and output statements, main function are translated to their equivalents in Java. Due to limited time, we excluded further features from the translator, particularly arrays, vectors, pointers, inheritance. Nevertheless, in Chapter 5 we comment on extension of the translator to these further features. Hence, we extracted programs from GeeksforGeeks which made use of the supported language features listed above, and created the test sets to evaluate our translation approach.

In Figure 9, we show an example of C++ and Java source code to find fibonacci numbers, extracted from GeeksforGeeks.

C++

```
//Fibonacci Series using
Recursion
#include<bits/stdc++.h>
using namespace std;

int fib(int n)
{
     if (n <= 1)
          return n;
     return fib(n-1)
+fib(n-2);
}

int main ()
{
     int n = 9;
     cout << fib(n);
     getchar();
     return 0;
}

// This code is contributed
// by Akanksha Rai
```

Java

```
//Fibonacci Series using
Recursion
class fibonacci
{
     static int fib(int n)
     {
     if (n <= 1)
     return n;
     return fib(n-1)
+ fib(n-2);
     }

     public static void
main (String args[])
     {
     int n = 9;
     System.out.println(fi
b(n));
     }
}
/* This code is
contributed by Rajat
Mishra */
```

Figure 9: Example of parallel implementation of programs from our test set. We extracted a corpus of programs from GeeksforGeeks to create test sets. Here, we have the implementations of the same program in both C++, Java which determines whether a given number is Fibonacci.

4.3 Evaluation

As presented in Chapter 3, we firstly preprocess and analyze the input source code and then send it for rule-based translation. The translated code from our approach will be referred to as Machine Translated (MT) code; it is stored separately for every program in the test set. We will be referring to the Java code stored from GeeksforGeeks as Human Translated (HT) code.

We will be evaluating the translation approach by computing the accuracy scores of translations. In the past, to evaluate the performance of Machine translation researchers made use of human evaluation, by placing the original sentence with its machine translation side-by-side and human experts in both source and target languages rate the accuracy of translations; of course, this is expensive and time-consuming. More recently, approaches have been introduced to automatically evaluate the machine translations using human generated translations, and we make use of one of these.

A key question is how to actually compare the human translated and machine translated code. One approach would be to take differences of the two (e.g., using Git's diff tool) and if the difference is in some syntactic way minimal (e.g., using tree and string matching), claim that the two programs are syntactically identical. This approach, while feasible, does require some expensive calculations of syntax trees and tree differences, which can take a great deal of time for larger programs. The typical approach taken in the source code translation literature uses the Bilingual Evaluation Understudy Score (BLEU) to evaluate the quality of generated source code [Hindel et al 2015, Barone et al 2017, Vechev et al 2014, Pharaoh et al 2004]. The BLEU score compares the machine translation and human translation and gives an accuracy score. For example, below is the machine translation and human translations of a sentence.

Machine translated 1: It is a guide to action which ensures that the military always obeys the commands of the party.

Machine translated 2: It is to ensure the troops forever hearing the activity guidebook that party direct.

Human translated: It is a guide to action that ensures that the military will forever heed Party commands.

Now, we compare the Machine translated 1 and Machine translated 2 separately with Human translated sentence. An automatic evaluation will rank Machine translated 1 higher than Machine translated 2 by comparing matches of words between each machine translation and the human translation. When the matches are high the scores will be high.

The BLEU score compares the text in translated code to one or more reference translations line by line. This score might be misleading because the generated code might be a readable or good translation despite syntactic or layout differences from the reference: the text in translated code and the reference code might differ by extra spaces and even empty lines. Since the comparison takes place line by line (i.e., line 1 in translated code is compared to the line 1 in reference code and so on) if line 1 is empty in either of the translated code or reference code the BLEU score for that line will be very low and this will in turn affect the score of all other lines. To overcome this, using clustering within the preprocessing step, we cluster the source code into similar groups and then remove any particular cluster which contains unnecessary lines: comment lines, empty lines, header files from the Human Translated code and Machine translated code.

Once preprocessed, the two programs can be compared on a purely syntactic basis. We do this using two metrics. The first is the BLEU score, which we described earlier. The second metric we use is the Jaccard similarity score, which is a similarity metric that helps to assess the performance of machine translations [Baghel et al, 2019], by computing similarity between the translated code and the reference code [Mageed et al, 2020]. The Jaccard similarity measures the similarity between two text documents (for instance: A and B) by taking the intersection

of both and dividing it by their union i.e., dividing the number of words in common in the two documents (their intersection) by the union of words in the two documents. For instance:

A: She likes mango lassi and mango juice.

B: She likes mango.

Now, we will calculate the intersection and union of these two sets of words and measure the Jaccard similarity between A and B as below:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Using the above formula, the Jaccard score of the two sentences A and B is calculated as follows:

$$J(A, B) = \frac{\{she, likes, mango, lassi, and, mango, juice\} \cap \{she, likes, mango\}}{\{she, likes, mango, lassi, and, mango, juice\} \cup \{she, likes, mango\}}$$

$$J(A, B) = \frac{\{she, likes, mango\}}{\{she, likes, mango, lassi, and, mango, juice\}}$$

$$J(A, B) = 3 / 7 = 0.428 = 42.8\%$$

The Jaccard score for the above example with two sentences A and B is 42.8%. Similarly, the Jaccard score is calculated for the programs in our test.

Table 7 summarizes the BLEU and Jaccard similarity baseline score of our translation approach on our test set with preprocessed input source code. We noted the BLEU and Jaccard

scores of our translation approach for all the programs in our test with preprocessed input source code and derived the baseline score for our translation approach. Baseline score acts like a reference score i.e., we calculate the baseline score of our proposed approach and if any of the other approaches chosen for comparison achieves a score higher than the baseline of our proposed approach, then that approach is better than ours. We calculate the baseline score by taking the mean and standard deviation of all the scores, calculate the standard error by dividing the standard deviation with the square root of the number of datapoints, multiply the standard error by two and add it to the mean and this value is the baseline score. The individual BLEU and Jaccard scores of our translation approach for programs in our test set is presented in Table 1 in appendix D.

Baseline	C++ → Java
BLEU score	77.89%
Jaccard similarity score	81.34%

Table 6: Scores with Preprocessed input

The BLEU and Jaccard similarity scores range from 0-100. From our interpretation after an extensive investigation, we observed that the BLEU scores greater than 50 describes the translations as good high-quality translations and less than 20 as hard to understand and requires high level editing to finalize the translations, which is also supported by the literature [Lavie et al, 2019]. Whereas the higher the Jaccard similarity score means the machine

translated code is more similar to the human translated code. Our score values are not 100 because they are not same as the human translated code, and they are only similar to the human translated code to an extent which is good and understandable. Thus, from table 6 we conclude that our translation approach produces good translations that is readable and understandable for all the programs in our test set. We will compare the baseline score of our translation approach with the baseline scores of the existing approach and our translation approach excluding preprocessing in the next sections to evaluate the efficiency of our translation approach.

Apart from the above-mentioned metrics used to evaluate the translation, in this thesis we have a code analyzer built earlier based on the cluster analysis to analyze the source code for number of functions, loops, conditions, classes. Using the code analyzer, we can analyze the translated code and actual input source code and compare these both to validate i.e., to ensure the syntactic program structure is retained in the translated code from the actual input code. Tables in appendix A presents the analysis of input source code and translated code for programs in our test set in both source language (C++) and target language (Java), respectively.

4.4 Comparison with alternative approaches

In this section, we compare our translation approach with alternative approaches. We will do the evaluation in two ways: by comparison against a commercial translation; and by evaluating performance with and without the preprocessor. The comparison is done in the next section.

We attempt to show that we can obtain the better accuracy results by placing a preprocessing step before translation and by excluding the preprocessing step we attempt to present the results

which show lower accuracy scores. We also compare our translation approach with results produced by the commercial Tangible Software Solutions converter (for C++ to Java); as discussed in Chapter 2, the Tangible approach is also rule-based, where the rules are written by human experts. Comparison with the output of Tangible's software allows us to both assess accuracy as well as performance in contrast to commercial offerings. We chose Tangible's software for comparison because it is a rule-based translator and it also has a free edition which can be executed as black-box.

We created a test set earlier to evaluate our translation approach with preprocessing step. Again, we use the same test set to compute the BLEU and Jaccard scores by testing the programs using the Tangible software solutions C++ to Java converter. We compare the code translated by Tangible with Human Translated code compute the BLEU score and Jaccard similarity score accordingly. The table with BLEU and Jaccard similarity baseline scores for Tangible software solutions converter is presented in table 7.

Baseline	C++ $\rightarrow$ Java
BLEU score	33.24%
Jaccard similarity score	59.96%

Table 7: Baseline Scores of existing approach

From the above table, we note that the baseline BLEU score is 33.24% and the Jaccard score is 59.96% for the Tangible solutions C++ to Java converter.

In table 8 the baseline BLEU and Jaccard scores of our translation approach but excluding preprocessing is presented.

Baseline	C++ → Java
BLEU score	37.39%
Jaccard similarity score	41.26%

Table 8: Baseline Scores of our translation approach excluding preprocessing

From the above table, we note that the baseline BLEU score is 37.39% and the Jaccard score is 41.26% for the Tangible solutions C++ to Java converter.

Now, we compare the scores of our translation approach with preprocessing, our translation approach excluding preprocessing, and Tangible's C++ to Java converter. The table with the comparison of BLEU and Jaccard similarity scores of these approaches is presented in Table 9. The individual BLEU and Jaccard scores for programs in our test set is presented in Table 2 in appendix D.

Translation Approach	**BLEU Score**	**Jaccard Score**
With Preprocessing	77.89%	81.34%
Tangible	33.24%	59.96%
Without Preprocessing	37.39%	41.26%

Table 9: Comparison of proposed and existing approach accuracy scores

From Table 9, we observe that our proposed translation approach with preprocessed input outperforms the baselines of existing approach and our translation approach excluding preprocessing in terms of accuracy with 77.89% and 81.34% compared to the scores 33.24% and 59.96% of the existing approach and 37.39% and 41.26% of our translation approach excluding preprocessing. The proposed translation approach outperforms the baselines of the existing approach by a BLEU score value of 44.65%, Jaccard score value of 21.38% and our translation approach excluding preprocessing by a BLEU score value of 40.5%, Jaccard score value of 40.08%. Thus, we conclude that the performance of our translation approach with preprocessing is high compared to the alternative approaches.

4.5 Threats to validity

There are several threats to validity that apply to the experiments that we have carried out. We summarize these briefly.

- Only one full translation from C++ to Java was studied empirically. More studies on other programming language translations using the proposed approach would provide more data, but we decided to focus on one concrete source-to-source translation in depth rather than several in breadth. It is also expensive to build rule-based translators. As a result, we have thorough empirical data for that one translation. We leave it to future work to look at other programming langue translations using the proposed approach. We should point out that the architecture of our technical solution allows modular decomposition, and as such replacing our C++ to Java rule-based translator with another can be done following a systematic process.

- A relatively limited corpus of examples was used for experiments. More examples would obviously be better, but we were limited in terms of time (we had to build infrastructure as well). The empirical results show our translation approach is promising, certainly nothing in the results suggests that it is unnecessary or invalid to use unsupervised machine learning methods for preprocessing or preprocessing itself for source code translation.

4.6 Summary:

In this chapter we have seen the simulation setup for our experiments, how we gathered data from open sources for experiments and tests of our translation approach. Evaluating our translation approach using the BLEU and Jaccard metrics, comparing our translation approach with the alternative existing approach and we also compared our translation approach by excluding the preprocessing step (i.e., without preprocessing the input source code). By observing the accuracy scores and comparing it against each other here we conclude that our overall translation approach with a preprocessing step before translation has better accuracy results.

CHAPTER 5: Conclusions and Future Work

5.1 Summary

In this thesis, a novel approach to preprocess and analyze source code using unsupervised machine learning method (via K-Means clustering) prior to rule-based code translation was proposed. To contextualize our approach, we demonstrated its application as part of an end-to-end source code translation approach with three phases: preprocessing using K-Means clustering, code analysis, and rule-based code translation. The outputs for each phase were presented, and the overall performance and accuracy of the translation was evaluated.

The novelty of the work was specifically in decoupling preprocessing from rule-based translation and code analysis, and in a detailed evaluation of the use of K-Means to efficiently cluster the input source code; the latter specifically is designed to help in preprocessing and analyzing the source code prior to rule-based code translation. By using this preprocessing step prior to translation, all the unnecessary lines from the source code were removed which in turn helped in improving the accuracy of code translations, as demonstrated in the evaluation.

Our overall translation approach is fully demonstrated for translating programs from C++ to Java, but we also tried partial experiments investigating whether the first two phases (namely preprocessing and code analysis) can be applied to other programming languages; we constructed proof-of-concepts for this for the Python and C languages, but didn't investigate this in depth (i.e., we didn't build rule-based translators involving Python or C) as we had limited time. Our conclusion is that preprocessing has a net positive effect on code translation, as demonstrated in our evaluation, particularly through the accuracy and performance scores

in Chapter 5. Our proposed translation approach very significantly outperforms alternative existing approaches, including an entirely rule-based translation approach.

5.2 Future Work

In this thesis we focused only on translating source code from C++ to Java in an end-to-end translations, and analyzed the results. The results suggest that it is valuable to carry out preprocessing prior to rule-based code translation; it is evident from the results that preprocessing improves the accuracy scores of translations. We briefly investigated if the preprocessing phase and code analyzer can be generalized to other programming languages and as a part of this, we built preprocessors and analyzers for two more languages namely C and Python but did not investigate their effectiveness or performance any further, as this would have required building further rule-based translators as well. As such, an important part of future work is to carry out further studies on these other programming languages. Building the preprocessors and analyzers for any programming language following our approach is not difficult and it can be done by systematically following the source language specifications. The preprocessor could be coupled with any translator to translate programs. We can either build new rule-based translators or find translators that are already available to test and know the effects of preprocessing in accuracy of translations.

As one of our contributions, we produced a set of test data, derived from GeeksforGeeks, to be used in our experiments. This test set could be enhanced with more diverse programs. It would be interesting to run an experiment where developers were given, e.g., C++ programs and asked to hand-translate them to Java, or were given requirements and asked to produce C++ and Java implementations, where the Java implementations could then be compared against the C++ versions. By carrying out the experiments under controlled conditions (where

developer experience and familiarity with the requirements could be managed), we could produce additional quality data for testing.

The experiments focused on evaluating accuracy of the translation and the performance of the translator. There may be other properties that could be considered for study. Accuracy of the translation is really focused on syntactic equivalence. Another property to explore might be timing preservation: if the C++ program demonstrates particular timing characteristics (e.g., worst case timing performance on critical functions), can we demonstrate that the Java program demonstrates equivalent timing characteristics? Or timing behaviour that is at least as performant as the original?

This thesis made an argument for the use of K-Means for clustering in the preprocessor. We did investigate other clustering algorithms in Chapter 3, but showed that K-Means offered the best performance in this context. It may be interesting to explore other algorithms in more detail to dig into the relationship between preprocessing and rule-based translation further.

Finally, we would seek to extend the preprocessor to identify further clusters relevant to the source language under investigation, so that further forms of preprocessing could be supported. Two obvious candidates are inlined functions, which could form their own cluster; and macro instantiation, which would be relevant to languages like C++ or C. Indeed, a preprocessing may be the most appropriate way to translate programs in C++ or C that make use of macros, into a language like Java that does not support them.

References

1. Gerasimou, S., Kolovos, D., Paige, R., & Standish, M. (2018). Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems. Software Technologies: Applications and Foundations, 385–393. https://doi.org/10.1007/978-3-319-74730-9_34
2. Nguyen, A. T., Nguyen, T. T., & Nguyen, T. N. (2013). Lexical statistical machine translation for language migration. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE. https://doi.org/10.1145/2491411.2494584
3. Philipp Koehn, Hieu Hoang, Alexandra Constantin, and Evan Herbst. Moses (2007). Open-source toolkit for statistical machine translation. In Annual Meeting of the Association for Computational Linguistics (ACL).
4. Cheng Fu, Xinyun Chen, Yuandong Tian, (2019), An end-to-end neural program decompiler. In Advances in Neural Information Processing Systems, pages 3703–3714, 2019.
5. Banbara, M., & Tamura, N. (2000). Translating a Linear Logic Programming Language into Java. Electronic Notes in Theoretical Computer Science, 30(3), 20–45. https://doi.org/10.1016/s1571-0661(05)80111-2
6. Software Migration Planning. Smartsheet. (n.d.). https://www.smartsheet.com/all-about-software-migration-planning. Last retrieved June 13, 2021.
7. Xinyun Chen, Chang Liu, and Dawn Song (2018). Tree-to-tree neural networks for program translation. In Advances in neural information processing systems, pages 2547–2557, 2018.

8. Lachaux, M. A., Roziere, B., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages.
9. Fleurey, F., Breton, E., Baudry, B., Nicolas, A., & Jezequel, J. M. (2007). Model-driven engineering for software migration in a large industrial context. https://doi.org/10.1007/978-3-540-75209-7_33
10. Shoaib, M., Ishaq, A., Ahmad, M. A., Talib, S., Mustafa, G., & Ahmed, A. (2017). Software migration frameworks for software system solutions: A systematic literature review. https://doi.org/10.14569/IJACSA.2017.081126
11. Candel, C. J. F., Molina, J. G., Ruiz, F. J. B., Barceló, J. R. H., Ruiz, D. S., & Viera, B. J. C. (2019). Developing a model-driven reengineering approach for migrating PL/SQL triggers to Java: A practical experience. https://doi.org/10.1016/j.jss.2019.01.068
12. Fabry, J., & Zaytsev, V. (2019). Qualify first! a large-scale modernisation report. https://doi.org/10.1109/SANER.2019.8668006
13. Vavrova, N., & Zaytsev, V. (2017). Does Python smell like Java? Tool support for design defect discovery in Python. https://doi.org/10.22152/programming-journal.org/2017/1/11
14. John, T., Bijimol, T. (2014). A study of machine translation methods. https://doi.org/10.1.1.840.3069
15. Eibe Frank, Mark A. Hall, and Ian H. Witten (2016). The WEKA Workbench. Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques.
16. Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., & Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation. https://doi.org/10.1109/ASE.2015.36

17. Daniel, R., Antonio, S., Jose, L. (2012). A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools. https://doi/org/10.2298/CSIS111223022R
18. Qiu, L. (1999). Programming language translation. Cornell University. http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR99-1746
19. Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002). BLEU: a method for automatic evaluation of machine translation. https://doi/org/ 10.3115/1073083.1073135
20. Irons, E. T. (1961). A syntax directed compiler for ALGOL 60. Communications of the ACM. https://doi.org/10.1145/366062.366083
21. Yamada, Kenji. (2006). A Syntax-based Statistical Translation Model. https://aclanthology.org/P01-1067
22. Sanatan, M. (2017). Unsupervised Machine Learning and Data Clustering Analysis. Medium. Last retrieved July 29, 2021.
23. Knight, K. (2006). SPMT: Statistical Machine Translation with Syntactified target language phrases. https://aclanthology.org/W06-1606
24. Gerald, P. (2012). Computational Linguistics. https://doi/org/ 10.1016/B978-0-444-51747-0.50005-6
25. Quirk, C. (2005). Dependency treelet translation: Syntactically informed phrasal SMT. https://doi/org/ 10.3115/1219840.1219874
26. Cowan, B. (2006). A Discriminative Model for Tree-to-Tree Translation. https://doi/org/ 10.3115/1610075.1610110
27. Shofner, K. (n.d.). Statistical Vs. Neural Machine Translation. United Language Group. https://www.unitedlanguagegroup.com//statistical-vs-neural-machine-translation. Last retrieved on June 5, 2021.

28. Gecseg and M. Steinby. (1984). Tree Automata. https://arxiv.org/abs/1509.06233
29. Ott, M., Conneau, A., & Denoyer, L. (2018). Phrase-Based & Neural Unsupervised Machine Translation. https://arxiv.org/abs/1804.07755
30. Kasthuri, R., Indani, Y., & Rahul. (2020). Understanding K-means Clustering. https://www.edureka.co/k-means-clustering/. Last retrieved on May 25, 2021.
31. Joshi, A., & Knight, K. (2006). Statistical Syntax-Directed Translation with Extended Domain of Locality. https://aclanthology.org/2006.amta-papers.8
32. Barone, A. V. M., & Sennrich, R. (2017). A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. https://arxiv.org/abs/1707.02275
33. Marc, M. (2004). Lecture 5: Compiler Theory: Syntax-directed translations. https://www.csd.uwo.ca/~mmorenom/CS447/Lectures
34. Aho, A. V., & Ullman, J. D. (2007). Properties of syntax directed translations. https://doi.org/10.1016/S0022-0000(69)80018-8
35. Pyster, A., William, B. (1978). Semantic syntax-directed translation. https://doi/org/10.1016/S0019-9958(78)90344-3
36. Bragagnolo, S., & Derras, M. (2021). Software Migration: A Theoretical Framework. https://hal.inria.fr/hal-03171124/
37. Chiang, D. (2010). Learning to translate with source and target syntax. https://aclanthology.org/P10-1146.pdf
38. Sam, Y. Neural Machine Translation & How does it work? TranslateFX. (n.d.). https://www.translatefx.com/neural-machine-translation-engine. Last retrieved June 6, 2021.

39. Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev, (2014). Phrase-based statistical translation of programming languages. https://doi/org/ 10.1007/3-540-45751-8_2.
40. Lavie, A. (2019). Evaluating the Output of Machine Translation Systems. https://www.cs.cmu.edu/~alavie/Presentations/MT-Evaluation-MT-Summit-Tutorial-19Sep11.pdf
41. Manoj, S., Avli, S. (2016). Using Categorical Attributes for Clustering. International Journal of Scientific Engineering and Applied Science – Volume-2, Issue-2
42. Karan Aggarwal, Mohammad Salameh, and Abram Hindle, (2015). Using machine translation for converting python 2 to python 3 code. https://doi/org/ 10.7287/PEERJ.PREPRINTS.1459V1
43. Ranzato, M., Lample, G. (2017). Unsupervised machine translation using monolingual corpora only. https://arxiv.org/pdf/1711.00043
44. SYSTRAN. (n.d.). https://www.systransoft.com/systran/translation-technology/what-is-machine-translation/. Last retrieved May 12, 2021.
45. Thain, D. (2020). Introduction to compilers and language design. https://www3.nd.edu/~dthain/compilerbook/compilerbook.pdf
46. Dave, S., Parikh, J., & Bhattacharyya, P. (2001). Interlingua-based English–Hindi machine translation and language divergence. Machine Translation, 16(4), 251-304.
47. Lehman, M. M. (1979). On understanding laws, evolution, and conservation in the large-program life cycle. https://doi.org/10.1016/0164-1212(79)90022-0
48. Chris Laffra. C2J, a C++ to Java translator. https://sep.stanford.edu/oldstep/matt/jest/C2j/c2j.html. Last retrieved July 10, 2021.

49. Tangible. C++ to Java converter [Computer Software] free edition. https://www.tangiblesoftwaresolutions.com/. Last retrieved April 10, 2021.
50. Ram, A., Jalal, S., Jalal, A. S., & Kumar, M. (2010). A Density Based Algorithm for Discovering Density Varied Clusters in Large Spatial Databases. International Journal of Computer Applications, 3(6), 1–4. https://doi.org/10.5120/739-1038
51. Langford, J., & Kavka, C. (2011). Expectation Maximization Clustering. 382–383. https://doi.org/10.1007/978-0-387-30164-8_289
52. Mageed, M., Adebara, I., Nagoudi, E.M.B. (2020). Translating Similar Languages: Role of Mutual Intelligibility in Multilingual Transformers. https://arxiv.org/pdf/2011.05037
53. Baghel, A. S., Malik, P. (2019). Performance Enhancement of Machine Translation Evaluation Systems for English – Hindi Language Pair. https://doi/org/ 10.5815/ijmecs.2019.02.06

A. Original and Preprocessed source code:

Original Source code

```
// C++ Program to find the area
// of triangle
#include <bits/stdc++.h>
using namespace std;

float findArea(float a, float
b,float c)
{
      // Length of sides must be
positive
      // and sum of any two
sides
      // must be smaller than
third side.
      if (a < 0 || b < 0 || c <
0 ||(a + b <= c) || a + c <= b
||b + c <= a)
      {
cout << "Not a valid trianglen";
            exit(0);
      }
      float s = (a + b + c) / 2;
      return sqrt(s * (s - a) *
            (s - b) * (s - c));
}

// Driver Code
int main()
{
      float a = 3.0;
      float b = 4.0;
      float c = 5.0;
cout << "Area is " <<findArea(a,
b, c);
      return 0;
}

// This code is contributed
// by rathbhupendra
```

Preprocessed Source code

```
float findArea(float a,
float b, float c)
{
if (a < 0 || b < 0 || c < 0
||(a + b <= c) || a + c <= b
||b + c <= a)
{
"cout << ""Not a valid
trianglen"";"
exit(0);
}
float s = (a + b + c) / 2;
return sqrt(s * (s - a) *
(s - b) * (s - c));
}
int main()
{
float a = 3.0;
float b = 4.0;
float c = 5.0;
"cout << ""Area is "" <<
findArea(a, b, c);"
return 0;
}
```

Figure 1: Area of Triangle original and preprocessed source code

Original Source code

```
#include <stdio.h>

// Returns the new average after
including x
float getAvg(float prev_avg, int x,
int n)
{
      return (prev_avg * n + x) /
(n + 1);
}

// Prints average of a stream of
numbers
void streamAvg(float arr[], int n)
{
      float avg = 0;
      for (int i = 0; i < n; i++) {
            avg = getAvg(avg,
arr[i], i);
            printf("Average of %d
numbers is %f \n", i + 1, avg);
      }
      return;
}

// Driver program to test above
functions
int main()
{
      float arr[] = { 10, 20, 30,
40, 50, 60 };
      int n = sizeof(arr) /
sizeof(arr[0]);
      streamAvg(arr, n);

      return 0;
}
```

Preprocessed Source code

```
float getAvg(float prev_avg, int x,
int n)
{
return (prev_avg * n + x) / (n + 1);
}
void streamAvg(float arr[], int n)
{
float avg = 0;
for (int i = 0; i < n; i++) {
avg = getAvg(avg, arr[i], i);
printf(""Average of %d numbers is %f
\n"", i + 1, avg);
}
return;
}
int main()
{
float arr[] = { 10, 20, 30, 40, 50,
60 };
int n = sizeof(arr) /
sizeof(arr[0]);
streamAvg(arr, n);
return 0;
}
```

Figure 2: Average Stream original and preprocessed source code

Original Source code

```
// A Program to check whether a
number is divisible by 7
#include <bits/stdc++.h>
using namespace std;

int isDivisibleBy7( int num )
{
      // If number is negative,
make it positive
      if( num < 0 )
            return
isDivisibleBy7( -num );

      // Base cases
      if( num == 0 || num == 7 )
            return 1;
      if( num < 10 )
            return 0;

      // Recur for ( num / 10 - 2
* num % 10 )
      return isDivisibleBy7( num
/ 10 - 2 *
                  ( num - num /
10 * 10 ) );
}

// Driver code
int main()
{
      int num = 616;
      if( isDivisibleBy7(num ) )
            cout << "Divisible" ;
      else
            cout << "Not
Divisible" ;
      return 0;
}

// This code is contributed by
rathbhupendra
```

Preprocessed Source code

```
int isDivisibleBy7( int num )
{
      if( num < 0 )
            return isDivisibleBy7(
-num );
      if( num == 0 || num == 7 )
            return 1;
      if( num < 10 )
            return 0;
      return isDivisibleBy7( num /
10 - 2 *
                  ( num - num / 10
* 10 ) );
}
int main()
{
      int num = 616;
      if( isDivisibleBy7(num ) )
            cout << "Divisible" ;
      else
            cout << "Not
Divisible" ;
      return 0;
}
```

Figure 3: Divisible by seven original and preprocessed source code

Original Source code **Preprocessed Source code**

```
//Fibonacci Series using Recursion
#include<bits/stdc++.h>
using namespace std;

int fib(int n)
{
      if (n <= 1)
            return n;
      return fib(n-1) + fib(n-2);
}

int main ()
{
      int n = 9;
      cout << fib(n);
      getchar();
      return 0;
}

// This code is contributed
// by Akanksha Rai
```

```
int fib(int n)
{
      if (n <= 1)
            return n;
      return fib(n-1) + fib(n-2);
}
int main ()
{
      int n = 9;
      cout << fib(n);
      getchar();
      return 0;
}
```

Figure 4: Fibonacci original and preprocessed source code

Original Source code

```
# include<bits/stdc++.h>

using namespace std;
//c++ implementation
long multiplyBySeven(long n)
{
      /* Note the inner bracket here. This
is needed
      because precedence of '-' operator is
higher
      than '<<' */
      return ((n<<3) - n);
}

/* Driver program to test above function */
int main()
{
      long n = 4;

      cout<<multiplyBySeven(n);

      return 0;
}
```

Preprocessed Source code

```
long multiplyBySeven(long n)
{
      return ((n<<3) - n);
}
int main()
{
      long n = 4;
      cout<<multiplyBySeven(n)
;
      return 0;
}
```

Figure 5: Multiply by seven original and preprocessed source code

B. C++ Code Analyzer for C++ programs in our test set

File name	Number of classes	Number of functions	Number of conditions	Number of loops
fibonacci.cpp	0	2	1	0

Table 1: Fibonacci C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
AreaofTriangle.cpp	0	2	1	0

Table 2: AreaofTriangle C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
avgstream.cpp	0	3	0	1

Table 3: Average stream C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
DayofWeek.cpp	0	2	0	0

Table 4: DayofWeek C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
divisiblebyseven.cpp	0	2	4	0

Table 5: divisiblebyseven C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
Findfibonacci.cpp	0	3	0	1

Table 6: Findfibonacci C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
Luckynumber.cpp	0	2	3	0

Table 7: Luckynumber C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
Minimumnumber.cpp	0	2	3	0

Table 8: Minimumnumber C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
payroll management system.cpp	3	23	104	43

Table 9: payroll management C++ Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
multiplybyseven.cpp	0	2	0	0

Table 10: multiplybyseven C++ Code Analyzer

Code Analyzer for machine translated Java programs in our test set

File name	Number of classes	Number of functions	Number of conditions	Number of loops
AreaofTriangleMT.java	1	2	1	0

Table 11: AreaofTriangle Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
avgstreamMT.java	1	3	0	1

Table 12: Average stream Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
fibonacciMT.java	1	2	1	0

Table 13: Fibonacci Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
divisiblrbysevenMT.txt	1	2	4	0

Table 14: Fibonacci Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
findfibonacciMT.java	1	3	0	1

Table 15: Findfibonacci Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
LuckynumberMT.java	1	2	3	0

Table 16: Luckynumber Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
multiplybysevenMT.java	1	2	0	0

Table 17: Multiplybyseven Java Code Analyzer

File name	Number of classes	Number of functions	Number of conditions	Number of loops
MinimumnumberMT.java	1	2	3	0

Table 18: Minimumnumber Java Code Analyzer

C. Preprocessed and Translated code:

Preprocessed code

```
float findArea(float a,
float b, float c)
{
if (a < 0 || b < 0 || c < 0
||(a + b <= c) || a + c <= b
||b + c <= a)
{
"cout << ""Not a valid
trianglen"";"
exit(0);
}
float s = (a + b + c) / 2;
return sqrt(s * (s - a) *
(s - b) * (s - c));
}
int main()
{
float a = 3.0;
float b = 4.0;
float c = 5.0;
"cout << ""Area is "" <<
findArea(a, b, c);"
return 0;
}
```

Translated code

```
class AreaofTriangle
{
float findArea(float a, float b,
float c)
{
if (a < 0 || b < 0 || c < 0 ||(a
+ b <= c) || a + c <= b ||b + c
<= a)
{
"System.out.println( ""Not a
valid trianglen"");"
System.exit(0);
}
float s = (a + b + c) / 2;
return Math.sqrt(s * (s - a) *(s
- b) * (s - c));
}
public static void main(String
args[])
{
float a = 3.0;
float b = 4.0;
float c = 5.0;
"System.out.println( ""Area is ""
+ findArea(a, b, c));"
}
}
```

Figure 1: Area of Triangle preprocessed and translated code

Preprocessed code **Translated code**

```
long multiplyBySeven(long n)
{
      return ((n<<3) - n);
}
int main()
{
      long n = 4;
      cout<<multiplyBySeven(n)
;
      return 0;
}
```

```
class multiplybyseven {
int multiplyBySeven(int n)
{
return ((n << 3) - n);
}
public static void main
(String arg[])
{
int n = 4;
System.out.println(multiplyByS
even(n));
}
}
```

Figure 2: Multiply by seven preprocessed and translated code

Preprocessed code **Translated code**

```
int isDivisibleBy7( int num )
{
      if( num < 0 )
            return isDivisibleBy7(
-num );
      if( num == 0 || num == 7 )
            return 1;
      if( num < 10 )
            return 0;
      return isDivisibleBy7( num /
10 - 2 *
                  ( num - num / 10
* 10 ) );
}
int main()
{
      int num = 616;
      if( isDivisibleBy7(num ) )
            cout << "Divisible" ;
      else
            cout << "Not
Divisible" ;
      return 0;
}
```

```
class divisiblebyseven
{
boolean isDivisibleBy7( boolean
num )
{
if( num < 0 )
return isDivisibleBy7( -num );
if( num == 0 || num == 7 )
return true;
if( num < 10 )
return isDivisibleBy7( num / 10
- 2 *( num - num / 10 * 10 ) );
}
public static void main(String
args[])
{
int num = 616;
if( isDivisibleBy7(num ) )
System.out.println("Divisible"
);
else
System.out.println( "Not
Divisible");
}
}
```

Figure 3: Divisible by seven preprocessed and translated code

Preprocessed code **Translated code**

```
int fib(int n)
{
      if (n <= 1)
            return n;
      return fib(n-1) + fib(n-2);
}
int main ()
{
      int n = 9;
      cout << fib(n);
      getchar();
      return 0;
}
```

```
class fibonacci
{
int fib(int n)
{
if (n <= 1)
return n;
return fib(n-1) + fib(n-2);
}
public static void main(String
args[])
{
int n = 9;
System.out.println( fib(n));
getchar();
}
}
```

Figure 4: Fibonacci preprocessed and translated code

D. Test result details:

Programs	BLEU score	JACCARD score
Area of triangle	87.82	68.18
Average stream	61.08	66.09
Divisible by seven	98.08	78.72
Fibonacci Numbers	94.95	85.81
Find Fibonacci	70.22	68.67
Lexicographic	69.00	86.58
Lucky number	71.89	73.88
Minimum number	75.63	87.08
Multiply by seven	80.09	85.43
Odd number	74.56	79.87
Smallest palindrome	67.93	73.46
Armstrong number	77.23	84.92
Factorial of a number	88.61	93.04
Prime number	67.35	78.61
Reverse a number	73.77	84.55

Table 1: Our translation approach BLEU and Jaccard score

Programs	BLEU score	JACCARD score
Area of triangle	50.26	49.72
Average stream	0.57	58.94
Divisible by seven	15.84	50.38
Fibonacci Numbers	5.35	30.73
Find Fibonacci	7.46	32.37
Lexicographic	0.41	45.98
Lucky number	0.76	22.3
Minimum number	11.59	38.91
Multiply by seven	2.35	25.35
Odd number	13.64	39.12
Smallest palindrome	0.47	21.34
Armstrong number	31.51	42.29
Factorial of a number	18.13	53.74
Prime number	7.57	38.15
Reverse a number	23.82	44.62

Table 2: Existing approach BLEU and Jaccard score