# A study on the performance of statistical machine translation and neural machine translation for prefix resolution in software engineering

by

**Hung Phan**

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2021

# DEDICATION

I would like to dedicate this thesis to my family without whose support I would not have been able to complete this work. I would also like to thank my friends for their encouragement during the writing of this work.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Ali Jannesari for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members, Dr. Carl Chang and Dr. Wei Le for their evaluations and suggestions to this work.

## ABSTRACT

Neural Machine Translation (NMT) is the current trend approach in Natural Language Processing (NLP) to solve the problem of automatically inferring the content of target language given the source language. The ability of NMT is to learn deep knowledge inside languages by deep learning approaches. However, prior works show that NMT has its own drawbacks in NLP and in some research problems of Software Engineering (SE). In this work, we provide a hypothesis that SE corpus has inherent characteristics that NMT will confront challenges compared to the state-of-the-art translation engine based on Statistical Machine Translation. We introduce a problem which is significant in SE and has characteristics that challenges the ability of NMT to learn correct sequences, called Prefix Mapping. We implement and optimize the original SMT and NMT to mitigate those challenges. By the evaluation, we show that SMT outperforms NMT for this research problem, which provides potential directions to optimize the current NMT engines for specific classes of parallel corpus. By achieving the accuracy from 65% to 90% for code tokens generation of 1000 Github code corpus, we show the potential of using MT for code completion at token level.

# CHAPTER 1. OVERVIEW

Deep Learning (DL) has been applied in different Software Engineering (SE) researches and problems[30, 13, 7]. It can contribute to all stages of software development life cycle, from requirements extraction, design, implementation to maintenance [30]. Since DL was vastly applied earlier in Natural Language Processing (NLP), a popular trend of applying DL in SE is to consider the input of SE problems as different types of documentation similar to NLP input. According to [7], there are 3 main types of documentation: Natural Language (NL), Software Documentation (SD) and Programming Language (PL). Based on the requirement of each tasks, the output of research works that used these types of documentation as input is varied by different types of code tokens. For the research works that used NL as input, they tend to find the element of code environment that satisfied the description in NL, which they applied for code search [5, 16] and code synthesis [31]. SD is special type of documentation written in NL but contains information about description of the Application Programming Interfaces (APIs) of different programming languages. There is a work in the literature on representing APIs as vector from SD [50]. For the research works used PL as input, they used deep learning translation between PLs [12] and code suggestion [11].

The output of DL researches in SE problems can not only be the source code or code tokens, but also could be the information from SDs and NLs. Different applications are proposed to translate between each types of documentations using Machine Learning (ML). There are works on generating SD as pseudo-code from code using DL and ML [3, 37], or generating documentation from API specifications [40], or generating commit messages in NL [32]. Neural Machine Translation (NMT), which is a technique relied on advantages of DL, can be assumed as the best translation engine for SE. The ability of NMT relies on the formation on multiple layers of neural network to capture more information for the translation of each elements in the source language [53]. Besides, along with text sequence, NMT can be applicable on a different data structure such as graph or tree, which is suitable for the representation of code [52]. Another advantage of NMT is the performance for inferring the results, which is usually outperform earlier Machine Translation techniques [53].

Before the era of NMT, Statistical Machine Translation (SMT) [10] was the most popular technique for solving SE problems which relied on MT approaches. With the idea of extending the original Bayes rule [49], SMT provides the ability of learning the context in Natural Language for translation between popular languages. Since the source code also embed information of NL [20], SMT is successfully be applicable of SE problems as translation between versions of Python [25] and between different PLs [36]. However, compared to newer trends of translation engines such as NMT, SMT reveals 2 drawbacks. First, it cannot learn information from long sequence of text. An implementation tool of SMT, Phrasal [15] can only process the phrase with maximum length of 7. Secondly, the training and testing time of SMT become worse with large training data and increases exponentially [46]. For these reasons, NMT has replaced SMT in SE problems [3, 37].

Although having many advantages, NMT itself contains some challenges which also appear in researches of different areas along with SE. [34, 51] mention about an important problem of NMT compared to SMT as rare words problem in NLP. Current popular NMT engines, OpenNMT [27] or Google NMT [33], cannot handle large size vocabulary with more than 100000 words. To optimize the problem, researchers considered rare words as Unknown words which their translated results are not counted to the final results. This fact caused NMT performs poorly when rare/unknown words are frequent in the corpus [34]. In SE researches, the problem of Type Inference using MT shows that SMT model provided by [41] has a significant higher accuracy compared to the original NMT approach in [19]. Similarly, for natural language diacritic restoration, [39] shows that SMT outperforms NMT. [33, 19, 41] have the same characteristics of parallel corpus, i.e., the length of source and target pairs are equal and the order of the source and target words are consistent with each other. This leads us to an assumption that if the parallel corpus for training MT has these characteristics, it will affect the accuracy of NMT. In summary, NMT tends to have lower accuracy than SMT due to the methodology of NMT that didn't support the rare words translation and the characteristics of parallel corpus.

In this work, we further investigate the efficiency of NMT vs. SMT in a new research problem that has similar characteristics of parallel corpus as [33, 19, 41]. Instead of focusing on only limited types of tokens in code environment, our problem provide a solution to help developers get the code of all types of code tokens based on its first letters in the form of abbreviation/prefix of tokens. To implement the solution for this problem, we build two spaces of abbreviations of code tokens as source language and code tokens as target language. Then, we implement two machine translation models, Neural Machine Translation and Statistical Machine Translation to learn the mapping from prefixes to code tokens for code suggestion. We also analyze the affect of unknown tokens along with the accuracy on each types of tokens. By the evaluation, we show that SMT outperforms the original NMT. We called our approach PrefixMap and analyze the effective translation on three types of documentation: NL, SD and PL. Overall, this paper provides the following contributions:

1. Proposing the translation based engine for code completion from first letters of tokens.
2. Providing algorithms for extracting parallel corpus of prefixes and tokens in 3 types of documentation used in NLP and SE.
3. Implementing and Optimizing Statistical Machine Translation for PrefixMapping.
4. Implementing and Optimizing Neural Machine Translation for PrefixMapping.
5. Analyzing the accuracy of NMT compared to SMT along with accuracy depending on each types of code tokens.

The structure of this thesis is provided as follow. In the next section, we will describe about the literature review. In this section, we will define important concepts we use for abbreviations, algorithms for data collection, and the overview architecture of the system. The core engines of translation, NMT and SMT, along with their optimization for this problem is also described in the third chapter. In chapter 4, we will provide the accuracy and head-to-head comparison between NMT and SMT in 3 types of documentation. Chapter 4 and 5 provide discussion and future works.

# CHAPTER 2. REVIEW OF LITERATURE

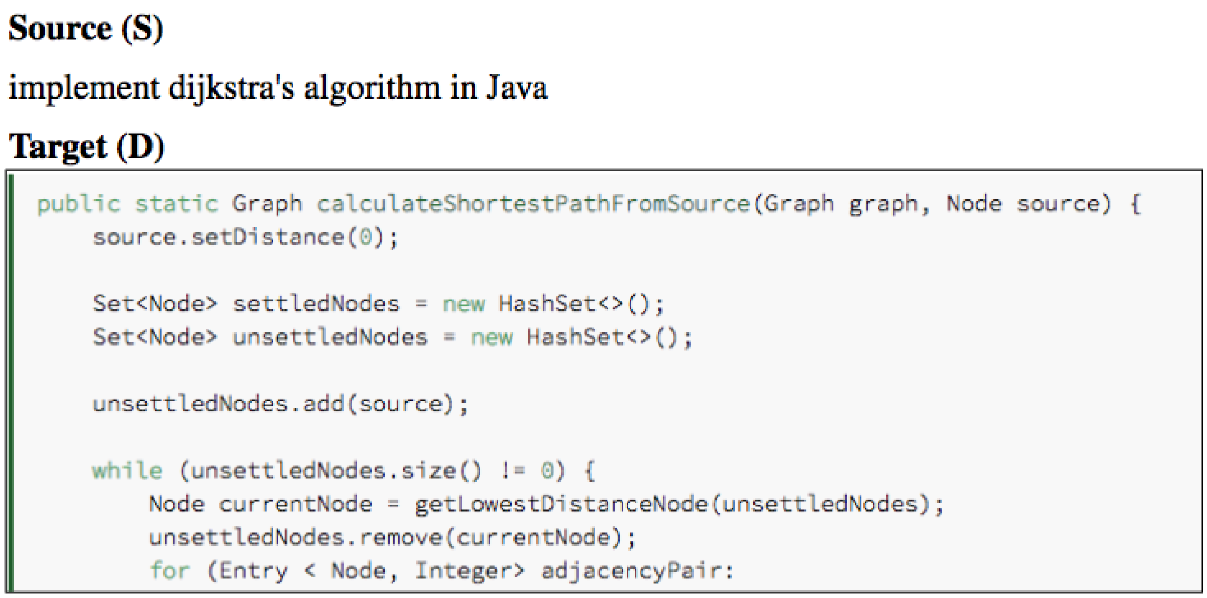
NMT outperforms SMT in popular machine translation problems like natural language translations. However, there is a class of translations that the NMT always perform worse than SMT. In this section, we will introduce this class of the MT problems and prior works on it.

## 2.1 Translation in general

The original machine translation problem is the problem of natural language translation. There are multiple natural languages corpus which were built as parallel versions from source to target language. For example, common languages that their corpuses are published are English, France, German etc. Researchers can work on translating two directions from a parallel corpus, for example translating from English to France and vice versa.

In Software Engineering, there are several problems that has been researched for a long time that inherited the idea of machine translation in NLP. One of the problem is natural language to source code translation. An application of this work is to automatically generating code to inexperienced developers. Another problem is code-to-code translation. Solutions for this problem can help developers to automatically migrate the code from one to another language, such as from Java to CSharp.

One characteristic of the translation problems we mentioned above is that the source and target side of the parallel corpus can be very different from the length of tokens. For example, in the NL to code task defined in Figure 2.1. While the source sequence has 5 words, the target sequence is a program with more than 100 code tokens. Depending on the nature of the research problems, there can be more rules to check the consistency between tokens at the same position of the source and the target language.



**Source (S)**

implement dijkstra's algorithm in Java

**Target (D)**

```
public static Graph calculateShortestPathFromSource(Graph graph, Node source) {
    source.setDistance(0);

    Set<Node> settledNodes = new HashSet<>();
    Set<Node> unsettledNodes = new HashSet<>();

    unsettledNodes.add(source);

    while (unsettledNodes.size() != 0) {
        Node currentNode = getLowestDistanceNode(unsettledNodes);
        unsettledNodes.remove(currentNode);
        for (Entry < Node, Integer> adjacencyPair:
```

Figure 2.1: Example of Natural Language to Code mapping

## 2.2 Property Translation

The property translation is a new concept that we defined in this project. Property translation solved as a class of machine translation problems. It treats source sequence as tokens. The MT models will learn to translate to the list as properties of tokens in the target language in which each token in a sequence will have a respected target token as the source token's property. Though it is a new concept, there are several problems from our knowledge can be classified as problem of Property Translation.

## 2.3 Part of Speech Tagging

Part of Speech (POS) is a category of words that have common grammatical properties. We can have several popular POSs like NN (noun), IN (preposition) or verbs (VBP, VBN) which they are already defined in famous linguistic processing engines such as nltk [1] in Python programming language. The steps of translating from words to POS can be considered as Property Translation, since the property of this problem is the POS and the length of the words should be equal to the length of the POSs. An example of POS translation can be shown in Figure 2.2. Another problem which can be considered as property translation in NLP is diacrictic resolution in some natural languages like Vietnamese [39].

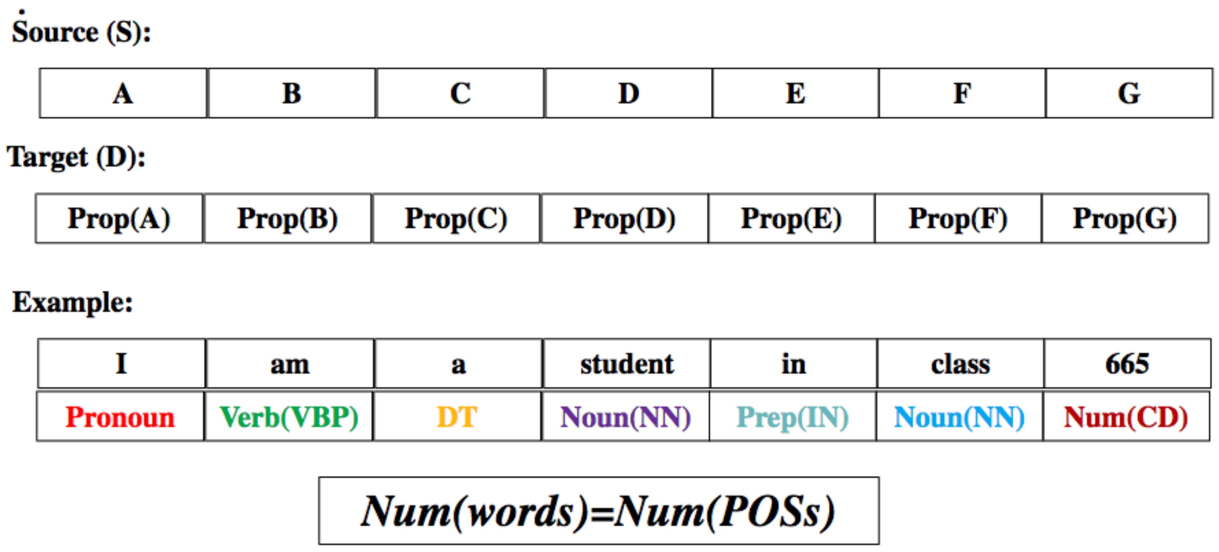


Figure 2.2: Example of POS translation

## 2.4 Type Inference

In SE, Type Inference is the problem of inferring the Application Programming Interface (API) packages given the input as the code snippets inside Online QA Forums for software developers, which contains no information about types. Similar to POS in NLP, Type Inference can be treated as the machine translation problem. The source language is the set of sequences of class names for each code tokens, while the target language is the set of type information for each class name. The Machine Translation models can be learned from the parallel corpus of complete source code project and will be able to infer the type information for online code snippet.

An example of Type Inference can be shown in Figure 2.3. In this example, we can infer the types required inside a File constructor in Java by machine translation. First, from the incomplete code, we infer the list of class names for tokens. Next, the machine translation will help us to infer the types of tokens. In this example, $java.io.File$ and $java.lang.String$ are 2 required types as the output. Differ from POS translation, in Type Inference we can check the consistent of tokens at the same position of source and target languages. To do that, we check the token of the target language has the postfix as the token of the source or not. If not, we need to provide reordering like Figure 2.4.

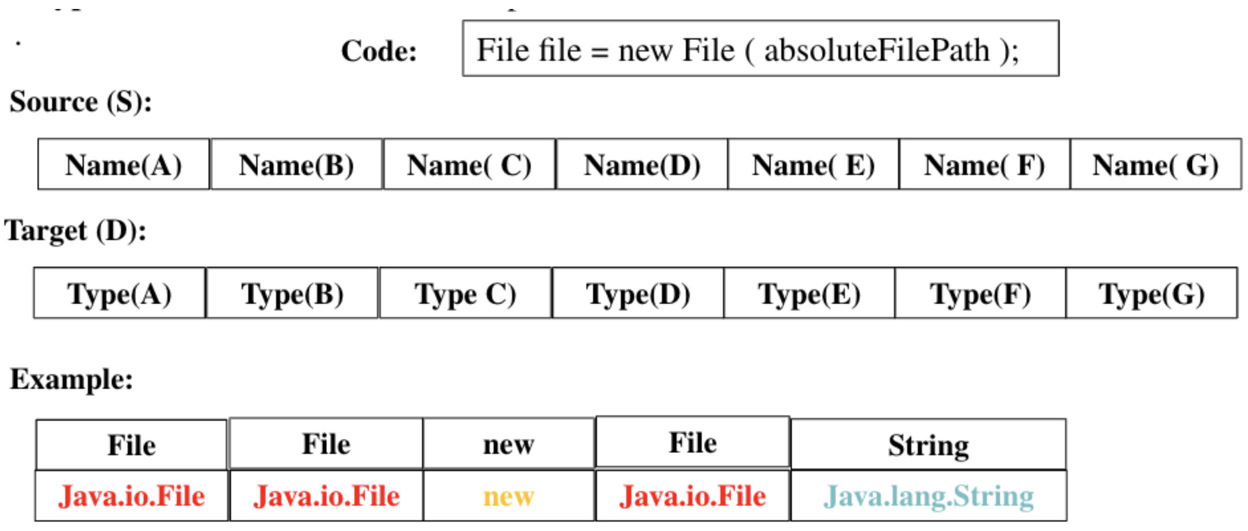


Figure 2.3: Example of Type Inference in Java

In our work, we consider two state of the art works for type inference using machine translation are StatType [41] and DeepTyper [19]. StatType uses Statistical Machine Translation that made the accuracy as 97% in F1 score for top-1 translation in Java. DeepTyper achieved 56% as the accuracy for top-1 on every types of code tokens in Javascript. To increase the accuracy, DeepTyper extends top accuracy to top-5 accuracy. Moreover, they adjust the layers of neural network to integrate the information of the consistency of Property Translation. Third, they combine their approach with another approach JSNice, which they can build DeepTyper as the tool for suggesting the cases that JSNice couldn't cover. By the improvement, they achieve over 95% accuracy on top common API. In summary, the original NMT in [19] performs with lower accuracy than StatType [41].

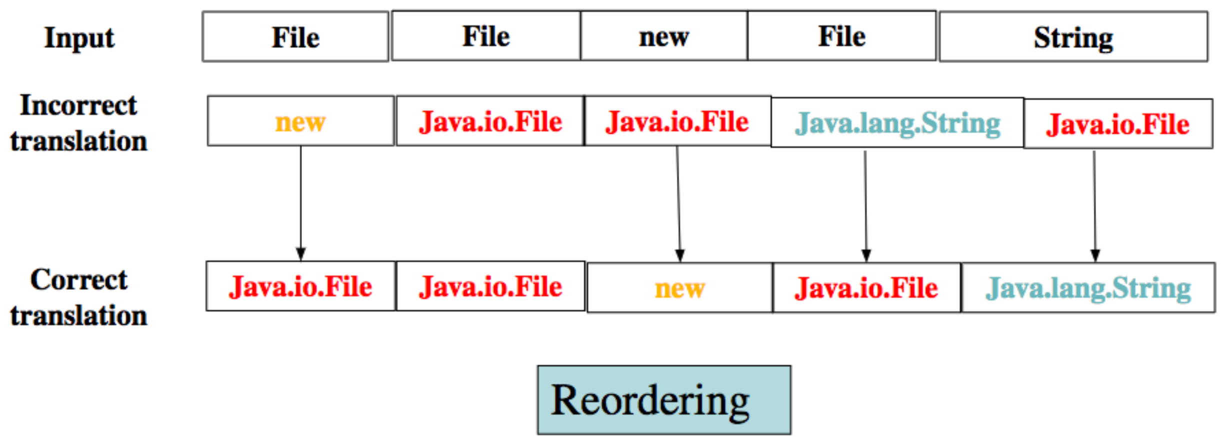


Figure 2.4: Example of reordering tokens in Type Inference

## 2.5 Drawbacks of Machine Learning and Deep Learning in SE

The characteristics of our parallel corpus appeared in other SE problems [19, 41]. In these works, SMT outperforms NMT in accuracy. In general, SE researches have specific characteristics of corpus, which brings rooms for deep learning and machine learning to improve their approaches. Other researches show drawbacks of Machine Learning in SE. [23] and point out the drawbacks of machine learning approach for method name recommendation that is usually suggest too simple method names. [46] shows that the original SMT has problems of exponential time increasing with big data. For the code suggestion area, other research works focuses on a specific types of code tokens. [54] suggested method name based on Hierachical Attention Networks, and [6] suggested method name and class name. In our work, we intend to generate all types of tokens based on writing the abbreviations or prefixes. For natural language to code translation, we have researches on applying NMT to pseudo code -to-code translation from SPOC data set [29]. This work applies the original NMT model using Long Short Term Memory (LSTM) and requires from 100 to 3000 times to run and validate the output of translation.

# CHAPTER 3. METHODS AND PROCEDURE

## 3.1 Prefix Resolution

[19, 41] provide a translation approach considered the source side language as partial class name (PCN) and the target language as Fully Qualified Name (FQN) of APIs. [39] treated the source language as a word without diacritic information and the target language as a word with diacritic information. In other words, both of these research works build a parallel corpus with the same length of source sequence and target sequence to each pair. The orders for source and for target sequences are also consistent. In summary, the problem of translation in [19, 41, 39] has the common in characteristics of parallel corpus.

In our work, we design an inferring system based on translation inspiring from [41]. While [41] focuses on the class name of APIs as the source language, we provide our source language as types of abbreviation for each words in a documentation corpus. Depending on different level of abbreviation, we build multiple translation models based on different length of the first letters for each words in parallel corpus. We have some definitions of elements in our context of translation problem.

**Definition 1** ***Prefix****: Given a word or code token, the prefix of a word is a word that combined by the set of first letters which developer can input to the code editor.*

Table 3.1: Example of source target tokens for Prefix Mapping at n-letter(s) prefixes

| | | |
|---|---|---|
| Source | 1-letter prefix | s ( m ) { L . l ( " ( ) _ + m + " ) ; f ( I l : m ) { l . n ( ) ; } } |
| | 5-letters prefix | synch ( mList ) { LogUt . logW ( "Requ ( ) _list + ... |
| | 9-letters prefix | synchroni ( mListener ) { LogUtils . logW ( "RequestQ ( ) _listener + ... |
| Target | Code tokens | synchronized ( mListeners ) { LogUtils . logW ( "RequestQueue.notifyOfItemInRequestQueue ( )... |

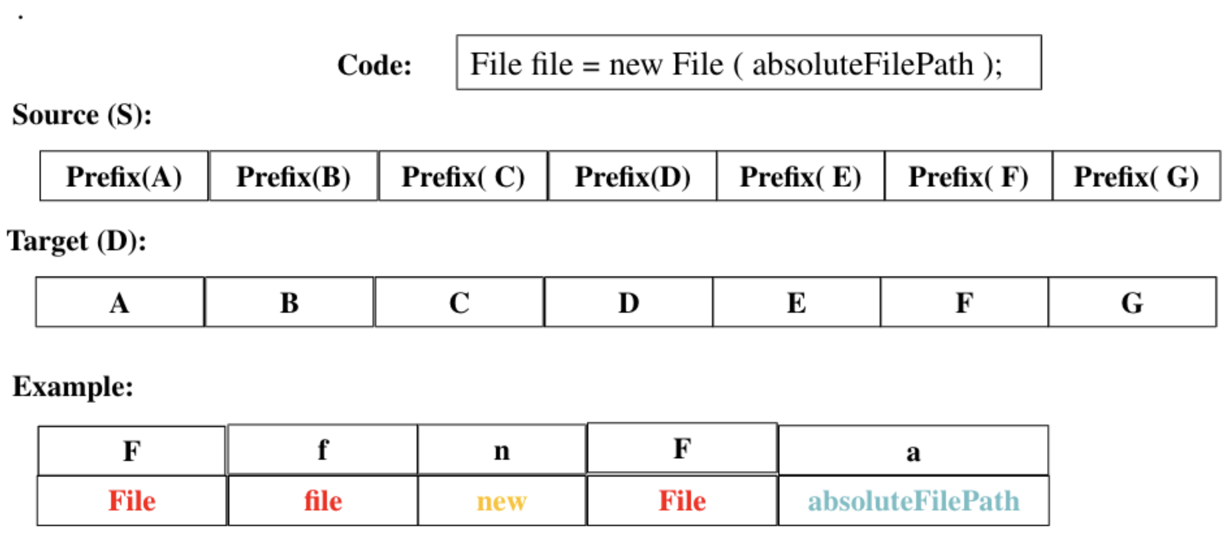


Figure 3.1: Example of Prefix Map in Java

**Definition 2** ***n-letter(s) Prefix****: The n-letter prefix of a word/ code token is the prefix that has length of n letter.*

While Type Inference related to the postfix of API packages, the prefix resolutions works as the translation from list of prefixes or abbreviation to the list of code tokens. The idea of prefix resolution can be applicable for code suggestion and reduce the size of the code corpus. An illustration of PrefixMap can be shown in Figure 3.1.

We see another detailed example of these definitions in PL in the figure 3.2. This code snippet is the method declaration of event function *fireQueueStateChanged()* from [8]. This function is to handle the event for Android version of Vodaphone devices. From this function, we see that there are many kinds of code tokens, including program keywords, class name, method name and variable names. Look at the token *notifyOfItemInRequestQueue()* as an example. In this token, its 1-letter prefix is `n`. Its 3-letters prefix is `not` and 9-levels prefix is *notifyOfI*. We provide a code suggestion that allows developers to write multiple prefixes at 1 level, 3 levels and 9 levels. Then, in the next step, the machine translation will translate from the

```
protected void fireQueueStateChanged() {
 synchronized (mListeners) {
  LogUtils.logW("RequestQueue. notifyOfItemInRequestQueue () listener["
    + mListeners + "]");
  for (IQueueListener listener : mListeners) {
   listener. notifyOfItemInRequestQueue() ;
  }
 }
}
```

Figure 3.2: Example of code snippet [8]

area of n-levels prefixes to suggest the full tokens. We do not restrict on any kinds of code tokens. In the other words, we support suggesting tokens for all types of prefixes.

To implement the solution, we do the following steps. First, we collect the data for software projects. Next, we extract the information from source as prefixes and target as the code tokens by the visiting code at Abstract Syntax Tree (AST) tree structure. Depending on the how many letters of the input prefix, the application will load the training models for the same letters prefixes inference. In the final step, the suggestion of code tokens from prefixes is provided. Example of tokens from source and target language at different n-letters is provided by Table 3.1.

## 3.2 Architecture Overview

The architecture overview of our tool, PrefixMap is provided as follows. We provide a code editor that accept developers to write abbreviation of code tokens in the form of first letters. After this step, we have the input as the mix of full code tokens and prefixes. The length of prefix can be varied. Then the information of code will be parsed by an Abstract Syntax Tree (AST) Parser to collect the code to sequence of prefixes. Each prefixes of code tokens will be encoded into the sequence as the source sequence for translation. Then,

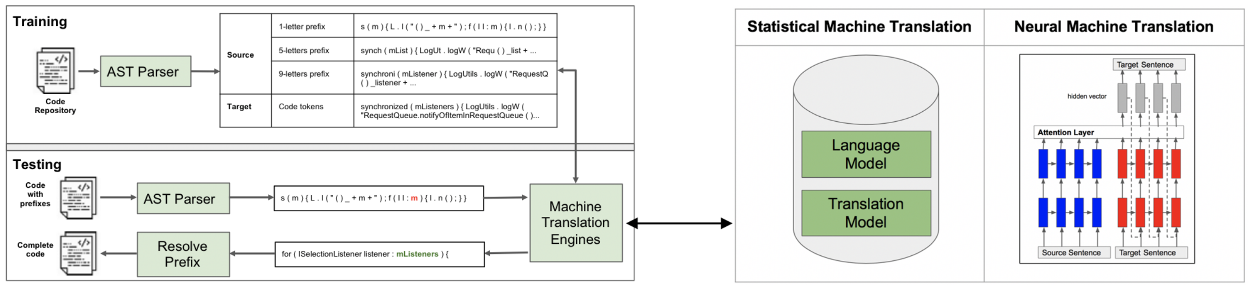

Training
Code Repository
AST Parser
Source
1-letter prefix
s ( m ) { L . l ( " ( ) _ + m + " ) ; f ( l l : m ) { l . n ( ) ; } }
5-letters prefix
synch ( mList ) { LogUt . logW ( "Requ ( ) _list + ...
9-letters prefix
synchroni ( mListener ) { LogUtils . logW ( "RequestQ ( ) _listener + ...
Target
Code tokens
synchronized ( mListeners ) { LogUtils . logW ( "RequestQueue.notifyOfItemInRequestQueue ( )...
Testing
Code with prefixes
AST Parser
s ( m ) { L . l ( " ( ) _ + m + " ) ; f ( l l : m ) { l . n ( ) ; } }
Machine Translation Engines
Complete code
Resolve Prefix
for ( ISelectionListener listener : mListeners ) {
Statistical Machine Translation
Language Model
Translation Model
Neural Machine Translation
Target Sentence
hidden vector
Attention Layer
Source Sentence
Target Sentence

Figure 3.3: PrefixMap Architecture Overview

the MT engines will convert the sequence of prefixes to sequence of tokens, which shows suggestions for each input prefixes. For example, `mList` and `notif` prefixes in Figure 3.2 will be translated to the expected tokens which are `mListeners` and method name `notifyOfItemInRequestQueue`. To be able for inference in the testing phase, we provide the training phase with the source language as prefixes and target language as code tokens from 1000 Github Corpus we collect from MSR 2013 [4].

From the architecture overview, we can see two important points we need to address compared to other neural machine translation works. First, we considered the both 2 MT engines for fixing the prefixes. We show the strength and the disadvantages of each MT models for this problem. Secondly, the PrefixMap was trained based on the parallel corpus with consistent in length of tokens and order of source and target tokens. Third, in our approach, each source tokens needs to be mapped with a target tokens, means the cases of unknown tokens will be affected since the MT cannot provide the suggestion. We will study about the affect in the Evaluation section.

**PrefixMap versus Type Inference** Both PrefixMap and Type Inference are in property translation problem and both have rules to check the consistent between tokens in the same position of source and target language. Type Inference tends to infer the prefix of the class name (which is fully qualified name). PrefixMap takes the source language as several first letters of the code tokens and try to infer the code token from the prefix of the token. In other words, Type Inference generates the word with its prefix, while PrefixMap generates from prefix to code.

## 3.3 Machine Translation Engines

In this section, we discuss about the algorithm for data extraction and the algorithm for SMT and NMT.

### 3.3.1 Data Extraction

The algorithm of extracting source side and language side is provided in figure 3.4. The core of implementation is done by an AST parser we extended from the source code of Eclipse JDT [22]. In this parser, we enhance the default `visit()` function which accepts an ASTNode object to a new function `visitAndExtract()`. This function behaves specifically to the MethodDeclaration node and the ASTNode objects inside this MethodDeclaration. The function for other AST objects accepts the node, the level of prefixes and the pair object. For each pairs, they contain one sequence of tokens as source and one sequence of tokens as target. Inside this function, first it will extract all tokens inside the ASTNode. Next, it will extract the prefix at n-levels for each tokens by a for loop. Final, the source and target sequence are included in the a new Pair object before adding the new object to the list. The `visitAndExtract()` function for MethodDeclaration accepts a list of pairs as input. It will create a new Pair object, visiting each ASTNode objects in the sub tree of this MethodDeclaration and extract the prefixes and related code tokens in training Github projects.

For example in figure 3.2, the algorithm figure 3.4 will visit the `fireQueueStateChanged()` MethodDeclaration to extract information all ASTNode inside this function. The information provides us the content of varies of types of ASTNode, including the MethodInvocation, for loop and inside variables and inside MethodInvocation. The target side for each tokens is actually the code content, while the source side contains the prefix of the code tokens, which can be inputted by developers in the suggestion phase. There is a corner cases that the size of the required prefix is greater than the length of the code tokens. If that case happened, the prefix will encode the information of the whole token. For instance, the token `for` has the prefix at 9-letters as `for`. In the next section, we will discuss about important elements of MT engines we used.

```
class PrefixMapVisitor extends ASTVisitor{
     ...
 void visitAndExtract(MethodDeclaration node,int level,List<Pair> list) {
      Pair p=new Pair();
      node.getBody().visitAndExtract(node,level,p);
 }
 ...
 void visitAndExtract(ASTNode node,int level,Pair p) {
  String[] tokens= getTokens(node);

  for t in tokens{
       String sToken=getPrefix(t,level);
       p.getSource().append(sToken);
       p.getTarget().append(t);
          }
 }
 ...
}
```

Figure 3.4: Algorithm to extract the source and target sequences by ASTParser in PrefixMap

### 3.3.2 Statistical Machine Translation for Prefix Map

Our implementation of SMT for PrefixMap is based on a well-known toolkit Phrasal [15] from StanfordNLP group. We call the source sequence as `Prefixes` and the target tokens as `Codes`. We call $prefix_i$ as the $i^{th}$ index prefix in the source sequence and $code_i$ as the $i^{th}$ index token in the target sequence. The purpose of SMT, along with NMT, is to calculate probability of each prefix is translated to target token given a context of code by different machine learning directions. For SMT, this probability contains 2 elements : the Language Model (LM) and Translation Model (TM),

**Language Model**. In SMT, the LM is calculated for the target language, means the sequence of code tokens in our problem [9]. Recently, there are newer LM models that encoded neural network as mentioned in [24]. In SMT, it used the statistical language model approach called n-gram. The n-gram language model will assign probability for code sequences of the whole MethodDeclaration along with the sequence of code tokens inside the body of method. In general, the most useful purpose of LM is to calculate the probability of the last word of n-gram sequence given the previous code tokens. The LM approach is also applied in building code suggestion tool from very large code database such as [4] .

Theoretically, to predict the next code tokens, the larger number of n-gram produces the better of code token suggestion. However, large n-gram will cause exponential time increasing in the performance. In practical, Phrasal restricts the maximum size of n-gram as 7. The fastest n-gram, uni-gram, is not usually used since I provide the estimation only by the number of appearance for that code tokens. The intuition of calculating the n-gram for code tokens is estimating the probability of the current n word given n-1 words by calculating the ratio between the number of appearance of sequence of n words per the number of appearance in sequence of n words. Smoothing is the technique that the LM are required if a word appeared in the unseen context to avoid the LM to assign zero probability. In our SMT implementation, we use the Kneser-Ney smoothing method, which is proposed by KenLM [17, 9, 18]. We have the probabilistic model $P_{LM}$ as follow:

$$\begin{aligned} &P_{LM}(code_n|code_1^{n-1}) = \\ &\quad u(code_n|code_1^{n-1}) + b(code_1^{n-1}) * P_{LM}(code_n|code_2^{n-1})) \end{aligned} \tag{3.1}$$

In Formula 3.1, the language model probability for the $n^{th}$ code token is calculated recursively by the respected probability of the $(n)^{th}$ token given (n-2) tokens. This probability has 2 other elements for normalization, called pseudo probability `u()` and backoff metric `b` [18]. The recursion will stop at the unigram distribution in Formula 3.2. In this Formula, `[]` is the empty sequence. the code tokens which are unseen in data will have the probability of the second operand, since `u()` is equals to Zero.Example of how LM represented in PrefixMap in line 5 example **??** is shown in Table 3.2.

$$P_{LM}(code_n|[]) = u(code_n|[]) + b(\epsilon) * \frac{1}{|vocOfTokens|}) \tag{3.2}$$

Table 3.2: Example of 6-grams LM probability in figure 3.2

| **n-1 gram** | **nth token** | **PLM** |
|---|---|---|
| for ( IQueueEnqueue listener : | list | 0.82 |
| | mListeners | 0.78 |
| | indices | 0.66 |
| | ... | ... |

In this example, we see that the output of $P_{LM}$ suggests the list of variables given 5 previous tokens in the 6-grams LM. It shows the most popular popular variable as the `list` variable which is the first candidate suggested by the LM. However, the inference between source and target language also depends on another probability along with the probability of the target language. We have the second element of the SMT, the translation model.

**Translation Model**. The translation model integrates the information from LM, the mapping probability of the prefixes given the code tokens (called the phrase translation probability) and the reordering score between translated phrase and input phrase. The final output of this step provided by Phrasal is a data structure called Phrase Translation Table, which contains probability for each phrase as candidate for translation. There are three stages in producing the phrase table. The first stage is word alignment, which is done by IBM model 1 by default [28]. Next, the phrase pairs of source and target languages are extracted. Final, the scoring phrase pairs process is done to assign the probability to phrase table. The probability of translation from prefixes to code tokens can be provided as follow:

$$Codes_{best} = argmax_{codes} p(Prefixes|Codes) * p_{LM}(Codes) \tag{3.3}$$

In Formula 3.3, the phrase translation probability $p(Prefixes|Codes)$ contains information about the association between source and target tokens and the probability for reordering the phrases order between source and target language. The association between the phrase of source given the phrase of target language can be scored by relative frequency of their co-occurrence in the training set [28]. There are several ways for estimating the phrase translation probability. In newest version of Phrasal, it used the Log Linear method, which calculate the translation model by Formula 3.4:

$$p(x) = exp(\sum_{i=1}^{3} \lambda_i * h_i(x)) \tag{3.4}$$

In this formula, the final translation probability will depend on the exponential of three features functions and 3 hyper parameters. They are log($\theta$) as the logarithm of relative frequency, log(d) as the log of reorder penalty, and log($P_{LM}$) as the logarithm of language model probability. The input variable x is a random variable contains information about the candidate phrases and the position of phrases in source language.

Table 3.3: Phrase table result of Line 5 figure 3.2

| **source** | **target** | **p(t\|s)** |
|---|---|---|
| for ( I l : m | for ( IQueueEnqueue listener : mListeners | 0.88 |
| for ( I l : m | for ( Index listItem : m | 0.62 |

The translation probability extracted by the phrase table after training the data of Line 5 in figure 3.2 is shown in Table 3.3. In this table, we see that the probability for the phrase contained the sequence `f ( I l :` returns the phrase that contains the correct translation result as `mListeners`.

**Optimization of SMT for PrefixMap**. Since this corpus for PrefixMap has chracteristics of consistent length and consistent order between source and target language, we alternate the original SMT model for suit with our problem. First, we create our own alignment of source and target tokens instead of using IBM Model 1. Second, with the original output from SMT, we provide an algorithm to reverse the reordered phrases. These steps ensure the output of SMT always be consistent in length and order with the input prefixes.

### 3.3.3 Neural Machine Translation for Prefix Map

For NMT, we implement the solution for PrefixMap based on well-known tool Google NMT [33]. The strength of NMT relies on the Encoder-Decoder architecture. As its name imply, the Encoder and Decoder layer provide an intermediate layers to convert the input sentence to a vector by an encoder model and convert from output vector to sequence of tokens in target language. These vector, called thought vector, can represent the meaning of sentence which capture other structures of sentences between source and target language. There are several details architecture of the Encode-Decoder model. Convolutional Neural Network (CNN) was proposed to learn for image processing and unsupervised learning [2, 14]. Graph Convolutional Network (GCN) is another architecture which can be used for graph structure data and semi-supervised classification [26]. Recurrent Neural Network (RNN) is the architecture used in sequence to sequence translation [47]. Long Short Term Memory (LSTM) is an improvement of RNN which help the training process to memorize the context efficiently [47]. Based on characteristics of PrefixMap, we select RNN model along with LSTM as recurrent unit and Attention mechanism to apply for this problem.

**Recurrent Neural Network**. Instead of splitting the sentence as prefixes by phrases, the RNN create a sequence of vectors represented for each prefixes and provide the translated output one by one. The information about the memory of translation will be represented as a hidden state vector $h_i$. Compared to n-gram which is usually be limited by the length of phrase, the hidden state vector can learn the information of previous prefixes at a very long distance. RNN provides a strategy to calculate the hidden state vector at time step t based on the hidden state of previous time step by the following formula:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1}) \tag{3.5}$$

$$p_t = softmax(W_{hy}h_t) \tag{3.6}$$

In formula 3.5, the $\sigma$ function is the non linear functions such as signmoid and tanh. The probability over the set of candidate translation at time step t will be calculated by formula 3.6. In this formula, $x_i$ is an embedded vector representation for $prefix_i$ in the sequence of source sentence. There are 3 weight matrices that can be learned from the training. They are the recurrent weight $W_{hh}$, the feed-forward weight $W_{xh}$ and the output weight $W_{hy}$. The LSTM is selected as recurrent unit which contributes to layer of RNN.

**Attention Mechanism**. The attention model provide the better context embedding which allow to get more information from the source representation instead of used only at the initialization of hidden state. There are 2 types of attention, global attention which considers all previous prefixes and local attention which considers subset of input prefixes. In this research, we use the global attention as the information.

The attention model creates connections between source hidden state and target hidden state. The heart of this problem is to design a context vector $c_t$. The new hidden state layers for target decoder can be estimated by the `tanh()` function:

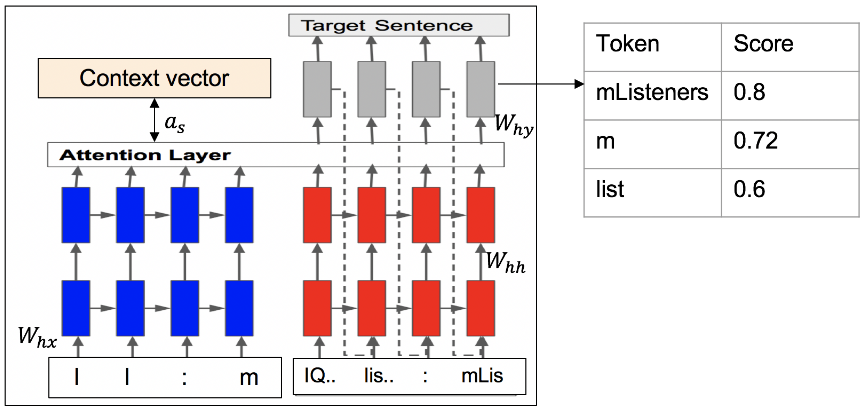
$$\overline{h}_t = tanh(W_c|c_t; h_t) \tag{3.7}$$

To calculate the context vector, Google NMT propose an internal vector called variable length alignment weight vector $a_t$ which embeds the information of both source states and target states. Given the source state s, the alignment weight vector is calculated as formula 3.8 from [33]:

$$a_t(s) = \frac{exp(score(h_t, \overline{h}_s))}{\sum(exp(score(h_t, \overline{h}_{s'})))} \tag{3.8}$$

The score function in formula 3.8 can be calculated by dot or concatenation by vectors [33]. To see how RNN and Attention work, we can look at the illustration on figure 3.2. In Figure 3.5, the step of translation is done by following steps. First, a sequence of input prefixes will be translated to a sequence of vectors $x_1, ..., x_k$. In the training of RNN, $W_{xh}$ represents for the relation between input vector and the hidden source, weight $W_{hh}$ propagates the relation between different hidden vector, while $W_{hy}$ represents the weight of output vector and hidden target state. The final score for each candidate of prefix `m` are shown in 3.5.

**Optimization of NMT for PrefixMap**. From formula 3.6, we see that the NMT provides a distribution for all posible output, mean all possible tokens in the vocabulary. So that, it cannot work with too large vocabulary with more than 40000 words [41]. To overcome this challenge, we replace the prefixes/ tokens with less than 10 times appearance as $Unknown$ token. We apply the same algorithm to reverse the order of the output of NMT like SMT model.



| Token | Score |
|---|---|
| mListeners | 0.8 |
| m | 0.72 |
| list | 0.6 |

Figure 3.5: NMT translation result of figure 3.2

# CHAPTER 4. RESULTS

## 4.1 Research Questions

We implement the tool PrefixMap with 2 Machine Translation engines.Since the idea of PrefixMap can also be applicable in different types of documents, we provide our experiment from all 3 types of documentations. They are Natural Language, Software Documentation and Programming Language. We also discuss on our results on comparing between Neural Machine Translation versus Statistical Machine Translation on this problem. We target to have the information about the prefix resolution based on different types of code tokens. In general, we want to answer the following research questions:

1. RQ1: Is prefix resolution important in programming language?
2. RQ2: Does NMT outperform SMT on PrefixMap by NLP translation evaluation metrics?
3. RQ3: Does NMT outperform SMT on PrefixMap by SE translation evaluation metrics?
4. RQ4: How SMT and NMT perform with different types of code prefixes?
5. RQ5: How SMT and NMT perform with ambiguous code prefixes?

## 4.2 Metrics for MT Evaluation

We use 2 metrics reflect the view point of NLP and SE: the BLEU score and the exact matching accuracy.

**BLEU**. Bilingual Evaluation Understudy score, called the BLEU score, is the fundamental metric for comparing the actual output and expected output of the MT problem [38]. The BLEU score will take input as pair of expected and translated result, it will output as the score from 0 to 1 to reflect the similarity at n-gram level of word. The higher of BLEU, the better of translation engine performed. This score is calculated based on the co-occurrence at of n-gram between expected and translated sentence along with strategies for penalty and smoothing [38]. We select the n-gram at 4-gram, and using the BLEU score implementation from Google NMT [33].

**Exact Match Accuracy**. In our MT problem, the requirement of good translation is not only restricted at the similar at n-gram. Similar to [41], another translation based approach in Software Engineering, we evaluate the Exact Match Accuracy at words level. Given the ith index of source sequence $prefix_i$ and of the expected sequence $expect_i$ and of the translated sequence $translate_i$, we compare the match between the $expect_i$ and $translate_i$. Since we have training data and testing data, we have Out of Vocabulary (OOV) cases. OOV has 2 types: Out of Source (OOS) means $prefix_i$ didn't appear in the training data and Out of Target (OOT) means $expect_i$ didn't exist in the training data. Since the prefix mapping process should suggest meaningful code token, we avoid evaluating the cases that the expected token was the same with prefix token.

**Training and Test Sets**. For each types of documentation, we split the data by 80% of training, 10% of validation and 10% of testing. The requirement of validation data is important in machine translation for tuning the training model to get optimized hyper parameters [33]. The selection of validation and testing is done randomly from the list of pairs of prefixes and tokens.

## 4.3 Corpus Preparation

We do the evaluation on 3 types of documentation:

**Natural Language (NL)**. We collect all English sentences from the large scale corpus of English-German translation in NLP in [33]. This corpus contains 1,15 millions sentences.

**Software Documentation (SD)**.We use the Conala corpus from [55]. This corpus contains Python software documentation as 116000 English sentences.

**Programming Language (PL)**.We collect 1000 Java projects from MSR 2013 corpus [4]. We extract 560000 pairs of source and target tokens. The algorithm for extracting source and target language can be found in section 3.

The application for PrefixMap in SD and PL is code completion for code like terms in SD and code tokens in PL. We also apply PrefixMap in popular corpus of NL for word completion, to reveal the differences of accuracy when applies PrefixMap in natural language and in programming language. For the variation of n-letter prefixes, we do the evaluation based on 3 levels: 1-letter prefix, 5-letters prefix and 9-letters prefix. To select these 3 types of prefix, we analysis the average length of each tokens for corpus, which brings the result on Table 4.3. It shows that the average length of code tokens is 18 letters, which means the developers need the auto code completion when they write 9-letters prefix or any prefix with less than 9 letters.

### 4.3.1 MT Models Configurations

To train SMT and NMT models, we use a high end computer with 32GB of RAM and an Nvidia RTX 2080 card with 8GB of GPU.

**Statistical Machine Translation**. We use the default configuration suggested by Phrasal [15].The details of configuration is shown in 4.1.

**Neural Machine Translation**. Number of units in each hidden layer affects the accuracy [45]. Increasing number of hidden unit can improve the accuracy but reduce the time performance since we need to change the batch size. So for NMT, we have 2 configurations as shown in Table 4.2.

Table 4.1: Configuration of Statistical Machine Translation Model

| **Key** | **Value** |
|---|---|
| MAX_PHRASE_LEN | 7 |
| Memory | 26 GB |
| ttable-limit | 250 |
| distortion-limit | 50 |
| stack | 100 |

## 4.4 RQ1: Analysis on Length of Tokens in NL, SD and PL

To answer RQ1, we analyze the average length of tokens in the target language of our NL, SD and PL corpus. The result is shown in Table 4.3. By this table, we show that the average length of words in NL is over 6 letters in English. This is consistent with our hypothesis since we consider that NL usually intend to

Table 4.2: Configuration of Neural Machine Translation Models

| **Key** | **Config A (512 units)** | **Config B (1024 units)** |
|---|---|---|
| attention | normed_bahdanau | normed_bahdanau |
| attention_architecture | gnmt_v2 | gnmt_v2 |
| **batch_size** | **32** | **16** |
| beam_width | 10 | 10 |
| decay_scheme | luong10 | luong10 |
| dropout | 0.2 | 0.2 |
| encoder_type | gnmt | gnmt |
| **infer_batch_size** | **32** | **16** |
| infer_mode | greedy | greedy |
| init_op | uniform | uniform |
| init_weight | 0.1 | 0.1 |
| learning_rate | 1 | 1 |
| length_penalty_weight | 1 | 1 |
| num_decoder_layers | 2 | 2 |
| num_encoder_layers | 2 | 2 |
| **num_train_steps** | **340000** | **680000** |
| **num_units** | **512** | **1024** |
| optimizer | sgd | sgd |
| share_vocab | FALSE | FALSE |
| src_max_len | 255 | 255 |
| steps_per_stats | 100 | 100 |

describe a single word in a token. Besides, the size of vocabulary in in NL corpus is around 20000, which is feasible for NMT to train and get the result without removing any unknown tokens.

For the SD and PL corpus, the result shows different characteristics of these data. For the average length, the PL corpus gains highest length of letter per separate words at over 16. The PL also reveals a high average length of letters as over 11. This is the expected result, since in SD and PL, sentences are usually made by developers. Unlike NL, developers have to mention about AST elements such as method names and class names, which increases the number of letters per each tokens. This means the code completion tool that allows getting code from prefix is needed.

The other points from Table 4.3 also shows that the vocabulary size of each corpus are varied. In contrast to NL, the SD and PL corpus contains a remarkably bigger vocabulary. In our assumption, this fact is due to

2 reasons. First, the PL and SD contains tokens that mentioned several words. It means that they can be combination of tuple or triple of words instead of uni-word per token like NL. Secondly, the SD and PL can be developed by many developers with different code naming style, which cause many rare words appeared in the corpus.

**Answer of RQ1:** Code tokens in Java usually have more number of letters than popular NL and SD corpus.

Table 4.3: Analysis on Length of Tokens in NL, SD and PL

| Target Languages | Vocab size | Average token length |
|---|---|---|
| Natural Language | 19038 | 6.08 |
| Software Documentation | 134886 | 11.77 |
| Programming Language | 541275 | 18.38 |

## 4.5 RQ2: BLEU Score Evaluation for Prefix Mapping in NL, SD and PL

In this experiment, we provide the translation for both NL, SD and PL corpus for SMT and configuration A of the SMT. The results are shown in Table 4.4 and Table 4.5. Along with these experiments, we have a verification experiment, which we run the configuration A on the English-German translation by [33]. We got the accuracy of BLEU score at **29.74** compared to **29.9** of Google NMT, which shows the validity of our NMT configurations. For NL SD, we use 1-letter prefix as the source language.

From Table 4.4, we show that the increasing of BLEU score from NL, SD to PL. The NL has the lowest BLEU score. The reason is that we are doing the mapping from a context of 26 letters in NL to the target tokens which contains 19000 words, which caused challenges for the MT models. The PL corpus at 1-letter prefix returns surprisingly higher BLEU score. It shows the potential of capturing the code tokens based on code context can be better than in NL and SD. The BLEU score increases if we change the length of the prefixes.

We talk about the comparison between MT models. We got the accuracy of SMT outperforms the NMT with all of the corpus. This fact shows the strength of SMT to resolve the characteristic of consistent order between source and target language. We have a comparison of BLEU score between configuration A and B. It shows that the accuracy in configuration B increases, which shows the important of increasing number of hidden units.

**Answer of RQ2:** SMT outperforms NMT significantly in accuracy of prefixes' translation measured by BLEU score.

Table 4.4: BLEU Score Evaluation using SMT and NMT in NL, SD and PL

| Corpus | SMT | NMT (config A) |
|---|---|---|
| NL | 13.36 | 8.67 |
| SD | 53.11 | 24.09 |
| PL (1-letter) | 63.4 | 53.99 |
| PL (5-letters) | 84.9 | 64.33 |
| PL (9-letters) | 92.61 | 74.42 |

Table 4.5: BLEU Score Comparison between Config A and B of NMT

| Corpus | NMT (Config A) | NMT (Config B) |
|---|---|---|
| PL (1-letter) | 53.99 | 52.78 |
| PL (5-letters) | 64.33 | 73.39 |
| PL (9-letters) | 74.42 | 79.12 |

## 4.6 RQ3: Exact Match Accuracy Comparison of Prefix Mapping in NL, SD and PL

The exact match accuracy are shown in Table 4.6 and Table 4.7. From these tables, we show the accuracy of the PrefixMap varied depending on the types of n-letters prefixes. We don't have the OOS cases in the NL, SD and PL with 1-letter prefix. For the SMT, we achieve over **65%** of precision score, showing the strength of SMT in this type of problem. For 9-letter prefixes, the accuracy gained to **90%**, means if developers wrote 9 letters of the code token, there are 9 per 10 cases the tool suggested correctly. We didn't include the non-useful suggestions counted to this accuracy. In the other words, the expected tokens need to be longer in letter than the prefixes.

For the NMT, the result is remarkably lower for config A but is improved in config B. We got the accuracy ranged from 61% from 1-letter prefix to 74% for 9-letters prefixes for config A of NMT. For config B, the precision ranked from 58.62% to 82.55%. Though the gap between configuration B and SMT is only 8%, another problem of the NMT is the OOV tokens. For 9-prefixes letter, we got the OOV of NMT as high as third times the OOV of SMT. This is caused by the fact that there is a set of words required to change to Unknown words, which can badly impact the total accuracy of NMT.

**Answer of RQ3:** SMT outperforms NMT significantly in accuracy of prefixes' translation measured by F1 score.

Table 4.6: Exact Match Accuracy Comparison between SMT and NMT Config A

| **Corpus** | **SMT** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **Correct** | **Incorrect** | **OOS** | **OOT** | **OOV** | **Prec** | **Rec** | **F1** |
| NL | 12671 | 35890 | 0 | 196 | 196 | 26.09% | 98.48% | 41.25% |
| SD | 40656 | 20176 | 0 | 1855 | 1855 | 66.83% | 95.64% | 78.68% |
| PL (1-letter) | 53868 | 28529 | 0 | 4090 | 4090 | 65.38% | 92.94% | 76.76% |
| PL (5-letters) | 40164 | 7897 | 540 | 3337 | 3877 | 83.57% | 91.20% | 87.22% |
| PL (9-letters) | 20554 | 2207 | 1374 | 1753 | 3127 | 90.30% | 86.80% | 88.51% |
| | **NMT (Config A)** | | | | | | | |
| | **Correct** | **Incorrect** | **OOS** | **OOT** | **OOV** | **Prec** | **Rec** | **F1** |
| NL | 10487 | 38074 | 0 | 196 | 196 | 21.60% | 98.17% | 35.40% |
| SD | 26237 | 31700 | 0 | 4750 | 4750 | 45.29% | 84.67% | 59.01% |
| PL (1-letter) | 44510 | 28711 | 0 | 13266 | 13266 | 60.79% | 77.04% | 67.96% |
| PL (5-letters) | 25565 | 13895 | 3758 | 8720 | 12478 | 64.79% | 67.20% | 65.97% |
| PL (9-letters) | 11778 | 4217 | 7263 | 2630 | 9893 | 73.64% | 54.35% | 62.54% |

Table 4.7: Exact Match Accuracy Comparison of NMT model Config B

| **Corpus** | **NMT (Config B)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Correct | Incorrect | OOS | OOT | OOV | Precision | Recall | F1 |
| PL (1-letter) | 42919 | 30302 | 0 | 13266 | 13266 | 58.62% | 76.39% | 66.33% |
| PL (5-letters) | 30853 | 8607 | 3758 | 8720 | 12478 | 78.19% | 71.20% | 74.53% |
| PL (9-letters) | 13204 | 2791 | 7263 | 2630 | 9893 | 82.55% | 57.17% | 67.55% |

## 4.7 RQ4: Analysis on Types of Tokens in Naming Conventions for Prefixes

To setup this experiment, we provide a module that check the regular expression of each tokens in the testing data set. There are regular expressions on checking if a numeric, a class name, a variable, a method name or a string literal. We found and add the regular expressions for checking at well-known online resource as [48, 44]. Considering the Configuration B as a higher accuracy NMT model compared to configuration A. The results are shown in Table 4.8 and Table 4.9.

The result shows the consistent between NMT and SMT configuration. The two types of token received lowest accuracy is Numeric and String literal. The numeric tokens got precision as 34% in 1-letter prefix of SMT and got 24.71% in 1-letter prefix of NMT configuration B. The String literal got 28% in each configuration. This fact reveals the challenge of code suggestion for numeric and string literal, since the they are not only depend on the tokens but also depend on the control flow or data flow graph of the program. For main types of tokens, we got the highest accuracy for class name while the variable name and constanct tokens achieved the equal results in both SMT and NMT. It is explainable since the good class names can be reused popular by developers which helps the SMT model to learn the prediction. Method name, in the other hands, varied based on the purpose of each developers.

**Answer of RQ4:** For prefixes' translation, numeric and string literal caused challenges for get good accuracy while class name achieved highest accuracy.

## 4.8 RQ5: Analysis on the Accuracy of PrefixMap on Ambiguous Tokens

In the last experiment, we analyze whether the number of prefixes to code token mappings can affect the accuracy. For each prefix in the training set, we run a program to check how many distinct code tokens are mapped to that prefix in the target language. The prefix that has more mapped tokens can be considered as more ambiguous. The result can be shown in Table 4.10 for SMT and Table 4.11 for NMT configuration B.

The first observation we got is that in 1-letter prefix, there is no case of mapping 1-1 to tokens in both SMT and NMT. This fact is explainable since the vocabulary of the source language contains only letters in alphabet and number digits, which are less than 100 prefixes. Besides, there are a large percentage of the prefixes has more than 100 mappings with 1-letter prefix corpus. For the 1-letter prefix, the SMT got accuracy about 70% while the NMT got 59% for very ambiguous tokens. For 5-letters and 9-letter prefixes, the accuracy of SMT decreases from unambiguous tokens to very ambiguous tokens. In the NMT with 9-letters prefix and for very ambiguous tokens, these tokens are considered as Unknown due to their rarely appeared in the data set, cause the NMT to decrease. In general, the SMT outperforms the NMT for almost all of accuracy experiments.

**Answer of RQ5:** For the translation of most ambiguous tokens, the SMT outperforms NMT in both 3 types of prefix. In NMT, sparsity tokens with more than 9 letters are usually be converted to Unknown tokens which caused low accuracy for this MT engine.

Table 4.8: Analysis Result on types of tokens for PrefixMap by SMT

| **SMT (1-letter prefix)** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Type of Tokens** | **Cor...** | **Incor...** | **OOS** | **OOT** | **OOV** | **Prec** | **Rec** | **F1** |
| Total: | 53868 | 28529 | 0 | 4090 | 4090 | 65.38% | 92.94% | 76.76% |
| NumericType: | 163 | 304 | 0 | 28 | 28 | 34.90% | 85.34% | 49.54% |
| ClassNameType: | 6154 | 2793 | 0 | 317 | 317 | 68.78% | 95.10% | 79.83% |
| VariableType: | 26937 | 14888 | 0 | 1503 | 1503 | 64.40% | 94.72% | 76.67% |
| MethodNameType: | 12908 | 6695 | 0 | 501 | 501 | 65.85% | 96.26% | 78.20% |
| StringLiteralType: | 628 | 1585 | 0 | 841 | 841 | 28.38% | 42.75% | 34.11% |
| ConstanctType: | 1321 | 821 | 0 | 281 | 281 | 61.67% | 82.46% | 70.57% |
| OtherType: | 5757 | 1443 | 0 | 619 | 619 | 79.96% | 90.29% | 84.81% |
| **SMT (5-letters prefix)** | | | | | | | | |
| Total: | 40164 | 7897 | 540 | 3337 | 3877 | 83.57% | 91.20% | 87.22% |
| NumericType: | 81 | 65 | 7 | 12 | 19 | 55.48% | 81.00% | 65.85% |
| ClassNameType: | 6986 | 755 | 40 | 270 | 310 | 90.25% | 95.75% | 92.92% |
| VariableType: | 17271 | 3861 | 258 | 1096 | 1354 | 81.73% | 92.73% | 86.88% |
| MethodNameType: | 10924 | 2057 | 23 | 474 | 497 | 84.15% | 95.65% | 89.53% |
| StringLiteralType: | 1219 | 389 | 100 | 730 | 830 | 75.81% | 59.49% | 66.67% |
| ConstanctType: | 1182 | 404 | 29 | 243 | 272 | 74.53% | 81.29% | 77.76% |
| OtherType: | 2501 | 366 | 83 | 512 | 595 | 87.23% | 80.78% | 83.88% |
| **SMT (9-letters prefix)** | | | | | | | | |
| Total: | 20554 | 2207 | 1374 | 1753 | 3127 | 90.30% | 86.80% | 88.51% |
| NumericType: | 11 | 3 | 2 | 0 | 2 | 78.57% | 84.62% | 81.48% |
| ClassNameType: | 4279 | 233 | 116 | 160 | 276 | 94.84% | 93.94% | 94.39% |
| VariableType: | 6436 | 950 | 511 | 431 | 942 | 87.14% | 87.23% | 87.19% |
| MethodNameType: | 6963 | 557 | 185 | 279 | 464 | 92.59% | 93.75% | 93.17% |
| StringLiteralType: | 868 | 179 | 288 | 456 | 744 | 82.90% | 53.85% | 65.29% |
| ConstanctType: | 923 | 155 | 104 | 114 | 218 | 85.62% | 80.89% | 83.19% |
| OtherType: | 1074 | 130 | 168 | 313 | 481 | 89.20% | 69.07% | 77.85% |

Table 4.9: Analysis Result on types of tokens for PrefixMap by NMT

| **NMT in Config B (1-letter prefix)** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Type of Tokens** | **Cor...** | **Incor...** | **OOS** | **OOT** | **OOV** | **Prec** | **Rec** | **F1** |
| Total: | 42919 | 30302 | 0 | 13266 | 13266 | 58.62% | 76.39% | 66.33% |
| NumericType: | 86 | 262 | 0 | 147 | 147 | 24.71% | 36.91% | 29.60% |
| ClassNameType: | 4554 | 3343 | 0 | 1367 | 1367 | 57.67% | 76.91% | 65.91% |
| VariableType: | 21774 | 16366 | 0 | 5188 | 5188 | 57.09% | 80.76% | 66.89% |
| MethodNameType: | 10742 | 7065 | 0 | 2297 | 2297 | 60.32% | 82.38% | 69.65% |
| StringLiteralType: | 354 | 902 | 0 | 1798 | 1798 | 28.18% | 16.45% | 20.77% |
| ConstanctType: | 875 | 586 | 0 | 962 | 962 | 59.89% | 47.63% | 53.06% |
| OtherType: | 4534 | 1778 | 0 | 1507 | 1507 | 71.83% | 75.05% | 73.41% |
| **NMT in Config B (5-letters prefix)** | | | | | | | | |
| Total: | 30853 | 8607 | 3758 | 8720 | 12478 | 78.19% | 71.20% | 74.53% |
| NumericType: | 40 | 18 | 84 | 23 | 107 | 68.97% | 27.21% | 39.02% |
| ClassNameType: | 5473 | 1245 | 444 | 889 | 1333 | 81.47% | 80.41% | 80.94% |
| VariableType: | 13575 | 4153 | 1402 | 3356 | 4758 | 76.57% | 74.05% | 75.29% |
| MethodNameType: | 8843 | 2366 | 227 | 2042 | 2269 | 78.89% | 79.58% | 79.23% |
| StringLiteralType: | 512 | 202 | 862 | 862 | 1724 | 71.71% | 22.90% | 34.71% |
| ConstanctType: | 727 | 223 | 305 | 603 | 908 | 76.53% | 44.46% | 56.25% |
| OtherType: | 1683 | 400 | 434 | 945 | 1379 | 80.80% | 54.96% | 65.42% |
| **NMT in Config B (9-letters prefix)** | | | | | | | | |
| Total: | 13204 | 2791 | 7263 | 2630 | 9893 | 82.55% | 57.17% | 67.55% |
| NumericType: | 6 | 3 | 6 | 1 | 7 | 66.67% | 46.15% | 54.55% |
| ClassNameType: | 3082 | 580 | 827 | 299 | 1126 | 84.16% | 73.24% | 78.32% |
| VariableType: | 3632 | 1244 | 2645 | 807 | 3452 | 74.49% | 51.27% | 60.74% |
| MethodNameType: | 5254 | 673 | 1335 | 722 | 2057 | 88.65% | 71.86% | 79.38% |
| StringLiteralType: | 251 | 75 | 1168 | 297 | 1465 | 76.99% | 14.63% | 24.58% |
| ConstanctType: | 456 | 91 | 632 | 117 | 749 | 83.36% | 37.84% | 52.05% |
| OtherType: | 523 | 125 | 650 | 387 | 1037 | 80.71% | 33.53% | 47.37% |

Table 4.10: Analysis Result on how PrefixMap can handle Ambiguous tokens by SMT

| **SMT (1-letter prefix)** | | | | | | |
|---|---|---|---|---|---|---|
| **NumofMap** | **1** | **2-10** | **11-20** | **21-50** | **51-100** | **>100** |
| | **Precision** | **Precision** | **Precision** | **Precision** | **Precision** | **Precision** |
| NumericType: | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 34.90% |
| ClassNameType: | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 68.78% |
| VariableType: | 0.00% | 0.00% | 0.00% | 100.00% | 0.00% | 64.40% |
| MethodNameType: | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 65.85% |
| StringLiteralType: | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 28.38% |
| ConstanctType: | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 61.67% |
| OtherType: | 0.00% | 40.00% | 97.35% | 83.18% | 0.00% | 70.68% |
| Total of tokens: | 0 | 10 | 2416 | 221 | 1 | 79749 |
| Percentage: | 0.00% | 0.01% | 2.93% | 0.27% | 0.00% | 96.79% |
| **SMT (5-letter prefix)** | | | | | | |
| NumericType: | 100.00% | 57.78% | 71.43% | 36.84% | 60.00% | 0.00% |
| ClassNameType: | 100.00% | 91.83% | 88.90% | 87.06% | 90.34% | 89.09% |
| VariableType: | 100.00% | 80.09% | 78.40% | 77.36% | 82.91% | 81.70% |
| MethodNameType: | 100.00% | 94.60% | 89.77% | 90.62% | 89.03% | 80.35% |
| StringLiteralType: | 100.00% | 85.04% | 75.63% | 74.70% | 68.45% | 65.50% |
| ConstanctType: | 100.00% | 83.66% | 79.35% | 75.09% | 66.67% | 58.60% |
| OtherType: | 100.00% | 87.09% | 84.58% | 90.91% | 91.85% | 82.38% |
| Total of tokens: | 1980 | 6425 | 3375 | 6411 | 7017 | 22853 |
| Percentage: | 4.12% | 13.37% | 7.02% | 13.34% | 14.60% | 47.55% |
| **SMT (9-letter prefix)** | | | | | | |
| NumericType: | 100.00% | 100.00% | 50.00% | 75.00% | 0.00% | 0.00% |
| ClassNameType: | 100.00% | 94.22% | 90.73% | 89.12% | 89.56% | 83.10% |
| VariableType: | 100.00% | 84.45% | 77.86% | 72.23% | 74.80% | 57.25% |
| MethodNameType: | 100.00% | 92.64% | 87.54% | 91.82% | 85.10% | 80.07% |
| StringLiteralType: | 100.00% | 84.38% | 69.66% | 68.29% | 71.74% | 59.85% |
| ConstanctType: | 100.00% | 80.74% | 66.27% | 72.46% | 64.71% | 66.67% |
| OtherType: | 100.00% | 88.58% | 81.52% | 90.79% | 88.46% | 53.33% |
| Total of tokens: | 6360 | 9952 | 1808 | 2550 | 1302 | 789 |
| Percentage: | 27.94% | 43.72% | 7.94% | 11.20% | 5.72% | 3.47% |

Table 4.11: Analysis Result on how PrefixMap can handle Ambiguous tokens by NMT

| **NMT in Config B (1-letter prefix)** | | | | | | |
|---|---|---|---|---|---|---|
| **NumofMap** | **1** | **2-10** | **11-20** | **21-50** | **51-100** | **>100** |
| | **Precision** | **Precision** | **Precision** | **Precision** | **Precision** | **Precision** |
| NumericType: | 0.00% | 0.00% | 66.67% | 29.79% | 30.77% | 23.16% |
| ClassNameType: | 0.00% | 0.00% | 25.00% | 62.50% | 0.00% | 57.68% |
| VariableType: | 0.00% | 0.00% | 0.00% | 0.00% | 19.64% | 57.15% |
| MethodNameType: | 0.00% | 0.00% | 0.00% | 0.00% | 50.00% | 60.33% |
| StringLiteralType: | 0.00% | 0.00% | 0.00% | 0.00% | 0.00% | 28.18% |
| ConstanctType: | 0.00% | 0.00% | 66.67% | 0.00% | 0.00% | 59.96% |
| OtherType: | 0.00% | 89.25% | 50.00% | 66.29% | 35.29% | 58.76% |
| Total: | 0 | 2661 | 32 | 324 | 90 | 70114 |
| Percentage: | 0.00% | 3.63% | 0.04% | 0.44% | 0.12% | 95.76% |
| **NMT in Config B (5-letters prefix)** | | | | | | |
| NumericType: | 76.47% | 66.67% | 64.29% | 0.00% | 0.00% | 0.00% |
| ClassNameType: | 79.67% | 84.24% | 79.10% | 76.19% | 84.81% | 0.00% |
| VariableType: | 70.64% | 74.03% | 78.03% | 85.35% | 64.47% | 57.97% |
| MethodNameType: | 91.36% | 88.80% | 82.14% | 81.87% | 74.68% | 63.74% |
| StringLiteralType: | 75.74% | 67.24% | 77.88% | 70.83% | 0.00% | 0.00% |
| ConstanctType: | 87.64% | 76.84% | 75.58% | 47.73% | 53.42% | 0.00% |
| OtherType: | 92.73% | 75.98% | 73.26% | 88.68% | 77.80% | 83.33% |
| Total: | 3324 | 10580 | 6002 | 10938 | 5283 | 3333 |
| Percentage: | 8.42% | 26.81% | 15.21% | 27.72% | 13.39% | 8.45% |
| **NMT in Config B (9-letters prefix)** | | | | | | |
| NumericType: | 0.00% | 66.67% | 0.00% | 0.00% | 0.00% | 0.00% |
| ClassNameType: | 83.72% | 86.38% | 76.40% | 68.18% | 0.00% | 0.00% |
| VariableType: | 77.08% | 72.33% | 70.32% | 68.24% | 0.00% | 0.00% |
| MethodNameType: | 92.88% | 88.57% | 80.41% | 67.58% | 77.50% | 0.00% |
| StringLiteralType: | 78.17% | 71.43% | 87.10% | 0.00% | 0.00% | 0.00% |
| ConstanctType: | 88.08% | 73.86% | 50.00% | 0.00% | 0.00% | 0.00% |
| OtherType: | 73.49% | 78.28% | 91.72% | 66.67% | 0.00% | 0.00% |
| Total: | 7403 | 6748 | 1300 | 504 | 40 | 0 |
| Percentage: | 46.28% | 42.19% | 8.13% | 3.15% | 0.25% | 0.00% |

# CHAPTER 5. SUMMARY AND DISCUSSION

In this work, we propose PrefixMap, a code suggestion tool for all types of code tokens in Java programming language. To realize our idea, we propose two Machine Translation models, Statistical Machine Translation and Neural Machine Translation, which learn the information from source language as the space of abbreviation or prefix to the target language as actual code tokens. Our work shows that we got an accuracy from 60% to 90% for SMT and from 59% to 83% for NMT. In the Machine Learning point of view, we reveal a class of parallel corpus which SMT can learn more information and get better accuracy on NMT in Software Engineering. Two of the characteristics of SE parallel corpus are the unknown tokens problem and being consistent on the length and the order of the source and target corpus. While the accuracy on different prefixes shows how SMT outperforms NMT, these characteristics provide initial take-away messages about the reasons of why NMT results low accuracy for this problem.

In the last few years, there are more and more researches regarding improving deep learning and neural machine translation in software engineering. [16] proposes DeepAPI which generates list of APIs given the code documentation. [50] represents the APIs inside software documentation as vectors. [11] designs a deep leanguage model for source code. [12] proposes a module for programming language translation using tree-to-tree model, while [52] implements a tree-based decoder which was augmented for Neural Machine Translation. [34] focuses on another problem of NMT, which is rare word problem. [35] works on applying LSTM for inferring the code a specific type of missing error as variable declarations.[21] proposes DeepRace, a bug finding tool for data race errors in parallel programs using advantages of Convolution Neural Network (CNN). We can think about integrating the new models from these works to NMT to increase the performance of NMT in SE.

# CHAPTER 6. FUTURE WORK

There are few limitations of our work. First, we only select 3 types of prefixes for our evaluation. We will extend our work to evaluate the accuracy of inference based on a range of lengths for prefixes. Secondly, we use an adhoc approach to treat the unknown token for NMT which is based on removing sparsity tokens from the corpus. In future work, we will study how abbreviations are written by developers to support more types of abbreviation suggestions instead of suggesting only by prefix, and apply optimization of NMT in other area such as [51] to improve the accuracy. The artifact of PrefixMap is available at [43, 42].

# BIBLIOGRAPHY

[1] Nltk toolkit. https://www.nltk.org/.

[2] AGHDAM, H. H., ET AL. *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[3] ALHEFDHI, A., ET AL. Generating pseudo-code from source code using deep learning. In *2018 25th Australasian Software Engineering Conference (ASWEC)* (Nov 2018), pp. 21–25.

[4] ALLAMANIS, M., ET AL. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)* (May 2013), pp. 207–216.

[5] ALLAMANIS, M., ET AL. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (2015), ICML'15, JMLR.org, p. 2123–2132.

[6] ALLAMANIS, M., ET AL. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, Association for Computing Machinery, p. 38–49.

[7] ALLAMANIS, M., ET AL. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. 51*, 4 (July 2018).

[8] ANDROID, P. Example of method firequeuestatechanged() in the android project. https://tinyurl.com/ukgnrph, August 2019.

[9] CHEN, S. F., ET AL. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics* (USA, 1996), ACL '96, Association for Computational Linguistics, p. 310–318.

[10] COSTA-JUSSÀ, M. R., ET AL. Statistical machine translation enhancements through linguistic levels: A survey. *ACM Comput. Surv. 46*, 3 (Jan. 2014).

[11] DAM, H. K., ET AL. A deep language model for software code. *CoRR abs/1608.02715* (2016).

[12] DRISSI, M., ET AL. Program language translation using a grammar-driven tree-to-tree model. *CoRR abs/1807.01784* (2018).

[13] FERREIRA, F., ET AL. Software engineering meets deep learning: A literature review, 2019.

[14] GHOSHAL, T., ET AL. Improving performance of convolutional neural networks via feature embedding. In *Proceedings of the 2019 ACM Southeast Conference* (New York, NY, USA, 2019), ACM SE '19, Association for Computing Machinery, p. 31–38.

[15] GREEN, S., ET AL. Phrasal: A toolkit for new directions in statistical machine translation. In *In Proceddings of the Ninth Workshop on Statistical Machine Translation* (2014).

[16] GU, X., ET AL. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, p. 631–642.

[17] HEAFIELD, K. KenLM: faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation* (Edinburgh, Scotland, United Kingdom, July 2011), pp. 187–197.

[18] HEAFIELD, K., ET AL. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Sofia, Bulgaria, Aug. 2013), Association for Computational Linguistics, pp. 690–696.

[19] HELLENDOORN, V. J., ET AL. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2018), ESEC/FSE 2018, Association for Computing Machinery, p. 152–162.

[20] HINDLE, A., ET AL. On the naturalness of software. In *Proceedings of the 34th ICSE 2012* (2012), ICSE '12, IEEE Press, p. 837–847.

[21] JAMSAZ, A. T., KHALEEL, M., AKBARI, R., AND JANNESARI, A. Deeprace: Finding data race bugs via deep learning. *CoRR abs/1907.07110* (2019).

[22] JDT, E. Eclipse jdt. `https://tinyurl.com/yx5hto3f`, February 2020.

[23] JIANG, L., ET AL. Machine learning based recommendation of method names: How far are we. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), pp. 602–614.

[24] JING, K., ET AL. A survey on neural network language models. *CoRR abs/1906.03591* (2019).

[25] K, A., ET AL. Using machine translation for converting python 2 to python 3 code, 2015.

[26] KIPF, T. N., ET AL. Semi-supervised classification with graph convolutional networks. *CoRR abs/1609.02907* (2016).

[27] KLEIN, G., ET AL. OpenNMT: Open-source toolkit for neural machine translation. In *Proc. ACL* (2017).

[28] KOEHN, P. *Statistical Machine Translation*, 1st ed. Cambridge University Press, USA, 2010.

[29] KULAL, S., PASUPAT, P., CHANDRA, K., LEE, M., PADON, O., AIKEN, A., AND LIANG, P. Spoc: Search-based pseudocode to code. *CoRR abs/1906.04908* (2019).

[30] LI, X., ET AL. Deep learning in software engineering, 2018.

[31] LIN, X. V., ET AL. NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system. In *LREC: Language Resources and Evaluation Conference* (Miyazaki, Japan, May 2018).

[32] LIU, Z., ET AL. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, Association for Computing Machinery, p. 373–384.

[33] LUONG, M., ET AL. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt* (2017).

[34] LUONG, T., ET AL. Addressing the rare word problem in neural machine translation. *CoRR abs/1410.8206* (2014).

[35] MOHAN, V. T., AND JANNESARI, A. Automatic repair and type binding of undeclared variables using neural networks. *CoRR abs/1907.06205* (2019).

[36] NGUYEN, A. T., AND OTHERS. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th ICSE* (New York, NY, USA, 2014), ICSE Companion 2014, Association for Computing Machinery, p. 544–547.

[37] ODA, Y., ET AL. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE) 2015* (Nov 2015), pp. 574–584.

[38] PAPINENI, K., ET AL. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (USA, 2002), ACL '02, Association for Computational Linguistics, p. 311–318.

[39] PHAM, T., ET AL. On the use of machine translation-based approaches for vietnamese diacritic restoration. *CoRR abs/1709.07104* (2017).

[40] PHAN, H., ET AL. Statistical learning for inference between implementations and documentation. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, may 2017), IEEE Computer Society, pp. 27–30.

[41] PHAN, H., ET AL. Statistical learning of api fully qualified names in code snippets of online forums. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018), ICSE '18, Association for Computing Machinery, p. 632–642.

[42] Phan, H., and Jannesari, A. Statistical machine translation outperforms neural machine translation in software engineering: Why and how. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages* (New York, NY, USA, 2020), RL+SEPL 2020, Association for Computing Machinery, p. 3–12.

[43] PrefixMap. Prefixmap data. https://pdhung3012.github.io/prefixmap.html, 2020.

[44] RegexTester. Regular expression collection site 2. https://www.regextester.com/97778, August 2019.

[45] ResearchGate. Question of hidden unit in nmt. https://tinyurl.com/j5b2o3r, August 2019.

[46] Saifullah, C. M. K., et al. Learning from examples to find fully qualified names of api elements in code snippets. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2019), pp. 243–254.

[47] Sherstinsky, A. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR abs/1808.03314* (2018).

[48] StackOverflow. Regular expression collection site 1. https://stackoverflow.com/, August 2019.

[49] Stone, J. V. *Bayes' Rule: A Tutorial Introduction to Bayesian Analysis*. Sebtel Press, 2013.

[50] Van Nguyen, T., et al. Characterizing api elements in software documentation with vector representation. In *Proceedings of the 38th International Conference on Software Engineering Companion* (New York, NY, USA, 2016), ICSE '16, Association for Computing Machinery, p. 749–751.

[51] Wang, X., et al. Neural machine translation advised by statistical machine translation. *CoRR abs/1610.05150* (2016).

[52] Wang, X., et al. A tree-based decoder for neural machine translation. *CoRR abs/1808.09374* (2018).

[53] Wu, Y., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR abs/1609.08144* (2016).

[54] Xu, S., et al. Method name suggestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (New York, NY, USA, 2019), PEPM 2019, Association for Computing Machinery, p. 10–21.

[55] Yin, P., et al. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories* (2018), MSR, ACM, pp. 476–486.