Automating Dictionary Mappings for Translating Natural Language Descriptions to Domain Specific Language Programs

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Vineet Hingorani
Y9227654

under the guidance of

Dr. Subhajit Roy
Dr. Amey Karkare

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
May 2014

CERTIFICATE

It is certified that the work contained in this thesis entitled "**Automating Dictionary Mappings for Translating Natural Language Descriptions to Domain Specific Language Programs**", by **Mr. Vineet Hingorani (Roll No. Y9227654)**, has been carried out under our supervision and this work has not been submitted elsewhere for a degree.

Dr. Subhajit Roy	**Dr. Amey Karkare**
Assistant Professor,	Assistant Professor,

Department of Computer Science & Engineering,

Indian Institute of Technology, Kanpur

Kanpur, 208016.

ABSTRACT

Reducing manual effort for monotonous activities has always been the goal of Computer Engineers and Programmers. However end-users often struggle with the automation softwares that require large amounts of manual inputs.

We address the problem of automating the mappings from the natural language words to a domain specific language tokens given a training data of pair of NL sentences and DSL programs. Constructing word-to-token maps manually consumes a lot of time and manual effort. We propose an algorithm using natural language processing tools and domain knowledge. The algorithm semi-automates the dictionary mappings construction. We next describe improvements in the algorithm, making it domain independent to reduce the manual effort.

The algorithm has been validated on an existing framework (implementing "Programming by Natural Language") and we were able to automate the dictionary mappings with minimal manual effort achieving a precision that is about 4% less than the precision achieved using manually constructed dictionary maps. In the end, we quantify the estimated manual effort in constructing the dictionary and using the proposed algorithm to compare the trade-off between the drop in precision versus the manual effort.

Acknowledgements

This work would not have been possible without the invaluable guidance and support of my guide, **Dr. Subhajit Roy** and my co-guide, **Dr. Amey Karkare**. I am truly indebted and express my earnest thanks to them. Many ideas in this work are due to the fruitful discussions we had. I also express my sincere thanks to **Dr. Sumit Gulwani** and **Dr. Mark Marron** for providing their valuable time whenever needed. I also thank Aditya Desai, Nidhi Jain and Sailesh Ramayanam, who, besides being brilliant project partners, were very dear friends. I genuinely feel honored to have worked with so many great minds.

I must surely thank the department of Computer Science and Engineering, IIT Kanpur, for providing me with an excellent environment for research.

I would like to thank my parents, my brother and all family members for their immense support and trust. There are many others and it is difficult to mention the names of everyone but I must mention Mr. Prakhar Banga, Mr. Nitesh Vijayvargiya, Mr. Rahul Ajmera, Ms Ravilla Srikavya, Mr. Jardheela Jain and Ms Tanushree Srivastava without whose love and support I would not have made such a fruitful journey. Though some differences came in between some of us but they taught me about tackling the life hurdles. I thank all my friends at IIT Kanpur for making my stay here one of the most memorable ones. I would also like to give special thanks to Mr. Prateek Singh and Mr. Vijay Dodwani for helping me understand the value of relations and constantly reminding me the importance of academics and career over other stuff.

Last, but not the least, I thank almighty Babaji, for showering me with all the above blessings.

Dedicated to my

Father: *Anil Hingorani*

Mother: *Poonam Hingorani*

Grand-Father: *Harnaam Tahiliani*

Grand-Mother: *Sheela Tahiliani*

Brother: *Ashish Hingorani*

family, teachers and friends

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

With computers taking on a ubiquitous and important role in various fields and applications, programming seems to be becoming a necessary skill across the professional spectrum. A central aspect to the use of computers is to reduce manual effort to get redundant and tough tasks completed. The prerequisite ability of translating ideas to code via languages like C, Java, etc may often be challenging depending on the application at hand and hence there arises a need for easier ways.

With a rise of social networking in today's society, there is a flood of information present in natural languages. This has given rise to the rapid development in the field of Natural Language Processing (NLP). As Wikipedia quotes, "NLP is a field of Computer Science and Linguistics concerned with the interactions between computers and human (natural) languages"[13]. Interactions between computers and human languages include understanding the natural languages, natural language generation, machine translation, etc. Among these, Machine Translation (MT) is a field dealing with translating one natural language to another using Artificial Intelligence. The problem of MT has been described as AI-complete[20] meaning that it is very hard to solve the problem using one specific algorithm.
Machine Translation as described in 1960 by Delavenay et al.[4] is the problem of automatic translation of text or phrases from a natural language (like English) to another natural language by the machine. NLP aims at understanding a natural language by the computer. Using Machine Translation, a similar problem can be used to translate a Natural Language to a Computer Lan-

guage which could easily be understood by the machines. Use of Natural languages as a means of programming has been the goal of researchers for many years.

Consider an example of an employee at a Data Analytics firm. She wants to make some changes in a word document of *10,000* lines. One such change is a simple replacement task given a condition. To make the example concrete consider the formal description of the task description below:

Example: *Replace all the occurrences of "minutes" by "seconds" that are immediately preceded numbers*

Suppose she starts doing her task by finding an occurrence of 'minutes' (can be done by simple FIND command in any text-editor, e.g. MS-Word, Open-Office Writer, etc.) and then proceeds to replace it with 'seconds' (again by using a simple REPLACE command) only after checking that 'minutes' is preceded by a number. To repeat this task manually could take hours given the possible size of the document.

Now suppose she hires a programmer, pays him to write a script for her to do this simple task. The programmer writes a script that helps reduce the manual effort to just a few minutes.

Now since hiring programmers can be expensive, suppose there exists an editor that gives the functionality of performing the above task using simple English sentence (as written above) and generates a program automatically (in a matter of milliseconds) that can be run on the text-document. This automatic generation of a program comes under the field of Program Synthesis.

1.1 Program Synthesis

Program synthesis is a technique of synthesizing programs automatically. As quoted by Wikipedia, "the goal is to construct, automatically, a program that provably satisfies a given high-level specification" ([14]). There are many ways of giving these specifications, Natural Language being the recent rapidly developing approach. An English statement describing the constraints for the program to be synthesized is the input comes under this category of specification. Though the specifications are not 'complete', an English statement can easily describe the conditions for a program to be synthesized. It has also an advantage of concise description. There are various ways of using natural languages as a means of synthesizing programs. Examples of variations of a task has been described in 1.1.

1. Prepend the line containing “P.O. BOX” with “*”
2. Add a “*” at the beginning of the line in which the string “P.O. BOX” occurs
3. Put a “*” before each line that has “P.O. BOX” embedded within it
4. If sentence contain “P.O.” as a substring add “*” before the sentence
5. Insert “*” at the start of every line which has “P.O. BOX” word
6. Put “*” in front of lines with “P.O. Box” as a substring

Table 1.1: Sample variations in NL for description of the same task

Recent works in Program Synthesis using Natural Language as specifications have shown impressive results ([5][12]). They have addressed the problem of designing a generic framework for synthesizing Domain-Specific Language (DSL) Programs using Natural Language (NL) descriptions. The input to their framework includes a DSL definition, a benchmark suite and a set of many-to-many dictionary mappings from the NL words to DSL tokens.
In this work, we have addressed the problem of constructing the many-to-many dictionary mappings automatically given the benchmark set of pair of English statements and Domain-Specific Language Programs. The dictionary contains maps from English words to DSL tokens. As the problem is similar to Word based Machine Translation, we have used various techniques of Natural Language Processing to build the maps. We have targeted the following domains as a means to validate our approach over the framework described in the chapter “Background Work”3.

1.2 Motivating Scenarios

1.2.1 Text-Editing

End-users working with text-editors, in order to handle enormous amount of data, are the target users in this domain. The domain includes editing text documents with operations like replacements, insertions, deletions, etc. This domain's data has been constructed after studying various help-forums where users often discuss performing such operations. The simple functionalities used in these operations are provided in editors like MS-Word/Excel but it is often difficult for non-experts to utilize these functionalities. Also, not all users are familiar with regular expressions

1. Replace “&” with “&&” unless it is inside “[” and “]”.
2. Add “$” at the beginning of those lines that do not already start with “$”.
3. Add “..???” at the last of every 2nd statement.
4. In every line, delete the text after “//”.
5. Remove 1st “&” from every line.
6. Add an “!” followed by a “?” at the end of every word.
7. Delete all but the 1st occurrence of “Cook”.
8. Substitute the 4th occurrence of “foo” with “bar” on each line.

Table 1.2: Sample benchmarks for text-editing domain

which are needed in text-editing tasks.

The benchmark suite is built by performing a user study of 21 text-editing tasks, collected by studying the help-forums, Excel-books, etc. The user study involved sophomore college students. Some of the benchmarks are also obtained from an independent corpus [9].

Consider an example of an operation found on one of these forums:

Example: *Print the* 2^{nd} *word after pattern “ABC” on every line*

This example is more than a simple find/print operation provided in the editors. It has a condition defined for the string (after pattern “ABC”) to be ‘printed’ (2^{nd} word). It also contains a constant string which gives scope of the required string. The DSL in which the program would be synthesized has been carefully designed to take into account these constraints and is powerful enough to express most of the tasks found on the forums. Table 1.2 shows some of the benchmarks from Text-Editing Domain.

An example of a statement-program pair looks like this:

Example: *Remove the first word of the line which starts with a number*

```
REMOVE(SelectString(WORDTOKEN(), ALWAYS(), INTEGER(1)),
          IterationScope(LINESCOPE(), STARTSWITH(NUMBERTOKEN()), ALL()))
```

1.2.2 Air Travel Information Systems

ATIS has long been used in Natural Language Processing and Speech Processing communities as a benchmark suite for baseline comparisons. It consists of English queries on the flight database.

1. I need information on a flight from San Francisco to Atlanta that would stop in Fort Worth.
2. What is the earliest flight from Washington to Atlanta leaving on Wednesday September fourth.
3. Okay we're going from Washington to Denver first class ticket I would like to know the cost of a first class ticket.
4. What ground transportation is there from the airport in Atlanta to downtown.

Table 1.3: Sample benchmarks for ATIS domain

The programs in the DSL designed for this domain resembles SQL like queries and the grammar for the DSL is flat instead of having a deep structure among the terminals and non-terminals.

A program in the designed DSL for the following benchmark looks like:

I would like the time of your earliest flight in the morning from Philadelphia to Washington on American Airlines

```
ColSet(AtomicColSet(COL_DEPARTURE_TIME()),
EXTRACT_ROW_MIN_T(COL_DEPARTURE_TIME(), AtomicRowPredSet(AtomicRowPred(
EQ_DEPARTS(CITY(philadelphia), Unit_Time_Set(TIME(morning)), ANY(), ANY(), ANY()),
EQ_ARRIVES(CITY(washington), ANY(), ANY(), ANY(), ANY())), EQ_AIRLINES(AIRLINES(american)))))
```

1.2.3 Automata

This is a domain targeting academicians working in the field of Finite State Automata. The synthesizer that converts an English language description to a program can be used by Intelligent Tutoring Systems to: i) automatically check the solutions of the students submissions for Assignments/Exams, ii) generate sample solutions for the examples taken in the class while teaching or iii) help Tutoring Systems in generating feedbacks.

The DSL for Automata Construction Domain has been designed using the descriptions given in [2]. Some of the programs in the DSL designed are given below:

- *Consider the set of all binary strings where the difference between the number of "0" and the number of "1" is even*

  ```
  ISEVEN(DIFF(COUNT(STRING(0)), COUNT(STRING(1))))
  ```

- *Let L1 be the set of words w that contain an even number of "a", let L2 be the set of words w that end with "b", let L3 = L1 intersect L2*

  AND(ISEVEN(COUNT(STRING(a))), ENDSWITHP(STRING(b)))

The DSL includes predicates over strings, quantifiers, boolean connectives, etc.

1.3 Our Contributions

The contributions made in this thesis are:

- Designing the algorithm and heuristics for building Dictionary Maps
- Implementation of the Algorithm using various tools
- Experimenting with different techniques to come up with right heuristics
- Validating our algorithm using the framework design in [5][12]

The thesis is organized in the following way. Chapter 2 gives the literature survey in the Translation systems using Natural Language and some of the tools used in the project. Background work has been briefly described in Chapter 3, one can find the details in [5][12]. Chapter 4 describes the algorithm and the approach taken for Automating Dictionary Mappings. Chapter 5 discusses various experimental results. We conclude with the discussions regarding future work in Chapter 6.

Chapter 2

Literature Survey and Tools Used

This chapter gives an overview of some of the related works in the literature and the research being done currently in this area. We also briefly describe some of the tools used in the implementation of the project.

2.1 Literature Review

Machine Translation is a wide field of research and the problem of translating text from one Natural Language to other Language [15] has been there in the Natural Language community since 1950s. There are different approaches to tackle the problem of Machine Translation some of which are:

- *Rule-based*: Based on some rules and the semantics information of the languages in hand, this approach converts the input language to target language. It often makes use of the grammars and dictionaries for translation.

- *Statistical*: This approach uses statistical methods and a corpus (for a domain) to translate a language's text to another.

- *Hybrid Approach*: As the name suggests, it is a hybrid approach towards machine translation using the advantages of both Statistical methods and rule-based methods.

2.1.1 Rule-Based Machine Translation

Rule-based Machine Translation is a field where linguistic information is used about the two natural languages[16]. Using the structure of sentences in a language and the rules of the grammar generating them, pattern matching is done for the rules of both languages. Domain knowledge is also essential to disambiguate the sentences[17].

For any Natural language, there is a particular way sentences are formed. A context-free-grammar of English would be advantageous in various ways helping in the translation of English texts. CFGs are powerful to express relation between the words in a sentence[6]. Using such knowledge about a language can help in using Rule-based MT.

2.1.2 Statistical Machine Translation

Early notable works in Statistical translation goes back to 1999 by Curin et al.[1] when they described the challenges while tackling the problem through statistical methods in a summer workshop at John Hopkins. For such a translation, the computer must have the knowledge of synonyms and semantics of the two languages. Later in 2008, Lopez[8] tackled the problem of Machine Translation using statistical methods treating it as the translation of natural language using Machine Learning techniques.

Word-based Machine Translation[18] is a sub-field of Statistical MT where the units of translation are words from one natural language to the other. This type of translation is somewhat analogous to the contribution in this thesis, except a few differences:

- Word-based Translation happens between two natural languages, whereas the target language in our case is a more formal language with context-free-grammar that can also be used as a computer language.
- The corpus needed for statistical machine translation tools is huge, whereas we have a corpus of about 500 benchmarks per domain which is too small to successfully use any statistical methods.

- The automation of dictionary building involves single word to single token translation, whereas Word-based MT can be used to translate a single word in one language to a phrase in another if required.

The approach used in this contribution is more of a hybrid approach[7] involving Statistical methods and Knowledge based approaches. As the corpora available is less, statistical methods alone do not perform well and hence some semantic knowledge about the natural language and the domains involved is used to tackle the problem.

2.2 Tools used for Automating Dictionary Mappings

A few Natural Language technique tools have been used during the automation of Dictionary Mappings.

2.2.1 WordNet

WordNet [10] is a lexical database for English Language. There are various synonym sets called *syn-sets*. The WordNet groups the English words in these syn-sets, giving short definitions about the words and defining the semantic relation between the synonym sets for Textual Analysis[19]. We have used it for obtaining the similarity measure between various words in a syn-set and across the syn-sets. The similarity between two words is calculated using the *path similarity* between the synonym-sets. Path similarity is the score denoting the similarity between the two syn-sets based on the shortest path connecting these senses (synonym-sets). We refer to our similarity calculating tool as Sim_{Path}.
NLTK is the Natural Language Toolkit [3] library in python. A python implementation of WordNet API has been used in the work. It is accessed as a reader of the NLTK corpus in python.

2.2.2 GIZA++

GIZA++[11], a Word based Statistical Machine Translation tool has been used in the initial part of the algorithm. It takes a huge set of pair of sentences of different languages and gives the similarity scores between the words of the two languages. It takes into account the joint occurrences of the words in the two languages and assigns the tuple an occurrence measure. Word based Machine Translation has been done using various training models in GIZA++.

In our case GIZA++ returns the joint occurrence measure between the English words and the Domain Specific Language terminals. It iterates over a benchmark set and checks which of the terminal occurs most when a word in English statement is present. Observing this, it calculates the score of similarity between each word in English sentence and every DSL token.

Chapter 3

Background Work

This chapter describes background work over which the work contributed in this thesis is based. We describe the framework on which the algorithm for automating dictionary mappings has been tested. The algorithm-designs and implementations shown in this chapter are not an outcome of this thesis but of the previous works[5][12]. Chapter 4 and onwards describes the contribution of this thesis project.

3.1 Synthesis Framework and Algorithm

This section describes the overall framework, for which Automating the Dictionary Mappings is needed, with some important definitions giving an overview of the algorithms used to build the synthesizer.

3.1.1 Inputs

The Natural Language to Domain-Specific Language synthesizer takes as input a DSL definition, a Dictionary and a Benchmark Suite.

1. DSL Definition: It includes a Context Free Grammar G which in turn contains terminals that are of two types and rules generating them.

    - Active Terminals: These are terminals that take other terminals as arguments.

- Passive Terminals: These are terminals that are complete in themselves i.e. they do not take any arguments.

2. Dictionary: It is a many-to-many mapping between Natural Language words and DSL terminals. D can be thought as a set of (w, t) pairs.

3. Benchmark Suite: A huge set of pair of English statement S and its correct translation P in Domain-specific-language.

Inputs to the framework is given by the developer who develops an NL-to-DSL synthesizer for a new domain. A context free grammar (CFG) has to be designed for a domain that expresses the type of tasks described by the Natural Language one-liners. A benchmark suite of NL statements is translated to DSL programs using the CFG. Building of mappings, manually, from NL words to DSL tokens is done to construct the Dictionary is again a great deal of effort that needs automation. To ease the Natural Language translation, automation of relation between the entities of one language to another is described in chapter 4.
When the framework is given an English statement, the first step of the algorithm is to generate all the valid programs in Domain-Specific Language, related to the given English statement. The set of words in the statement maps to a particular set of terminals in the Dictionary. This particular set of terminals is used for building the *valid* programs in the DSL. The second and final step is to rank these programs using various scores defined over the usual rules of English grammar and the structure of the sentences and programs.

All of the above is done in testing phase 1 i.e. when the framework is deployed. We will describe the training phase in later part of this chapter.

3.1.2 Bag Algorithm

Before describing the first step of the algorithm, we start-off with some definitions.

- **Consistent Map**: A consistent map M for an English statement S is a one-to-one mapping from a sub-set of all the words in S to terminals in DSL L using the dictionary D.

- **Valid Program**: A program P in a DSL $L = (G, SC)$ is valid according to an English statement S if $\exists$ a consistent map M such that the terminals $T_1, T_2, ..., T_i$ in the program P

are either in the range of M or are included in P by default (i.e. they are not translated from a word in S).

- **Correct Map**: A consistent map M is correct with respect to an English statement if and only if
  $Range(M) = TerminalSet(P_{correct}) \bigcup DefaultTerminalSet.$
- **Sub-program**: An expression that is a combination of terminals of the grammar.

The bag algorithm generates all the valid programs corresponding to an English statement using the consistent maps.

Sub-Programs, generated using bag of words from English sentence, are combined based on some rules. Two sub-programs p_1, p_2 can be combined if p_1 takes p_2 as an argument or vice-versa or p_1 and p_2 are both arguments to a production rule of G.

Example: Put ":" before the 1st " " in every line

INSERT(STRING(:), Position(BEFORE(STRING( )), INTEGER(1)), IterationScope(LINESCOPE(), ALWAYS(), ALL()))

In the above example, the subprograms *INSERT()* and *STRING(:)* are related to the production rule of G such that p_1 (i.e. *INSERT*) takes p_2 (*STRING(:)*) as argument. The subprograms *STRING( )* and *INTEGER(1)* are related to a rule such that they both are arguments to a same terminal *BEFORE()*.

Bag algorithm usually has a very high recall i.e. the set of programs generated by it contains the correct program but it also generates a number of unnecessary programs that are either semantically meaningless or are generated due to different placement of the terminals in the program. An example to explain this is given below:

Statement

Replace the first occurrence of “dog” in a line having “mouse” with an occurrence of “cat”

Valid Programs

```
1. REPLACE(SelectString(STRING(cat), ALWAYS(), FIRSTONE()),
            BY(STRING(mouse)),
            IterationScope(LINESCOPE(), CONTAINS(STRING(dog)), ALL()))
```

2. REPLACE(SelectString(STRING(mouse), ALWAYS(), FIRSTONE()),

   BY(STRING(cat)),

   IterationScope(LINESCOPE(), CONTAINS(STRING(dog)), ALL()))

3. REPLACE(SelectString(STRING(cat), ALWAYS(), FIRSTONE()),

   BY(STRING(dog)),

   IterationScope(LINESCOPE(), CONTAINS(STRING(mouse)), ALL()))

4. REPLACE(SelectString(STRING(dog), ALWAYS(), FIRSTONE()),

   BY(STRING(mouse)),

   IterationScope(LINESCOPE(), CONTAINS(STRING(cat)), ALL()))

As per the 1^{st}, 2^{nd} and 3^{rd} programs above, the placements of the constant strings “dog”, “mouse” and “cat” are different but none of them are correct. The 4^{th} program has the correct placement of the strings “dog”, “cat” and “mouse”. Programs like these are generated through the ‘bag of words’ approach of bag algorithm.

3.1.3 Ranking Programs

The intuition behing the algorithm is to look at various components of English statements and DSL programs to get different scores to rank the programs. Three types of scores have been used to rank the programs:

1. **Coverage Score**: The essence of this score is using all the information provided in the sentence. To define it formally, the notion of Usable Words should be clear. **Usable Words** are the set of all the words that are in the domain of the Dictionary D. Coverage Score is essentially the ratio of the number of Used words by the bag to the number of Usable words. A simple example where this score can be useful is:

   i would like to fly continental airlines between boston and philadelphia

   Two of the programs generated by the bag algorithm are: [1]

[1] The Domain specific language programs have been beautified for the ease of readers. Exact programs have other terminals which are unnecessary to the discussion here.

(a) *AtomicRowPredSet(BETWEEN_CITIES(Eq_Departs_IMP(CITY(boston), ...), Eq_Arrives_IMP(CITY(philadelphia), ...)), EQ_AIRLINES(AIRLINES(continental)))*

(b) *AtomicRowPredSet(BETWEEN_CITIES(Eq_Departs_IMP(CITY(boston), ...), Eq_Arrives_IMP(CITY(philadelphia), ...)))*

The first program contains the information about the airlines whereas the other program does not. Our ranking algorithm places the first program above the second one in ranking as the Coverage Score is higher for it.

2. **Mapping Score**: An English word may map to different terminals in D depending upon the different use of the word in sentences. For example, the word 'before' can map to the terminals START and BEFORE. A classifier C_{map} is constructed using the POS tags of the word in S to predict the probability of the terminal that should be present in the correct program of the sentence.

Example: Append a ":" after an integer

- **Mapping 1 Program**

  INSERT(STRING(:), Position(AFTER(NUMBERTOKEN()), ALL()), DOCUMENT())

- **Mapping 2 Program**

  INSERT(STRING(:), Position(AFTER(DIGITTOKEN()), ALL()), DOCUMENT())

Word	**Mapping 1**	**Mapping 2**
Append	INSERT	INSERT
":"	STRING(:)	STRING(:)
after	AFTER	AFTER
integer	NUMBERTOKEN	DIGITTOKEN

Here, the difference between the two mappings is due to the terminals NUMBERTOKEN and DIGITTOKEN mapping to the same word "integer". Clearly, what the user intended is the first program and hence, the value of its Mapping Score is higher than the second.

The score is also very useful once the Dictionary mappings are automated (described in later chapter). It assigns the most probable terminal to the correct word in the sentence which helps even if a terminal is mapped to some superfluous words (which come during automation process) in the dictionary.

3. **Structure Score**: This score helps in deciding the placement of the various terminals and sub-programs in the translated program. Structure Score uses various features depending upon how the sub-programs *combine* in a program.

   Connection: Consider the production rule $R =: N \rightarrow N_1...N_i..N_j...N_k$, the tuple (R, i, j) where $1 \leq i, j \leq k$, $i \neq j$ is called a Connection.

   Combination: Consider the program $P = (P_1, P_2, ..., P_k)$ generated using the production rule $R =: N \rightarrow N_1N_2...N_k$ of grammar, such that N_i generates P_i for $1 \leq i \leq k$. We say the pair of sub-programs (Pi, Pj) is combined using connection (R, i, j) and this combination is denoted as $Conn(Pi, Pj)$.

   The Structure Score of a program is the geometric mean of the all the combination probabilities of sub-programs.

$$StructureScore(P, S, M) = GeometricMean(CombProb(P, S, M))$$
$$CombProb(P, S, M) = \bigcup_{Conn(P_i, P_j)\ \epsilon\ P} C_{str}[Conn].predict(\vec{f}, 1)$$

   where $\vec{f} \equiv (f_{pos1}, f_{pos2}, f_{lca1}, f_{lca2}, f_{order}, f_{over}, f_{dist})$

   The various features used to construct the Structure score are described in [5][12].

A simple example to illustrate the use of Structure score is:

Replace " " by ":" and " "

Bag algorithm generates the following two programs:

1. REPLACE(SelectString(STRING( ), ALWAYS(), ALL()), BY(CONCATENATE(STRING(:), STRING( )), DOCUMENT())

2. REPLACE(SelectString(CONCATENATE(STRING( ), STRING(:)), ALWAYS(), ALL()), BY(STRING( )), DOCUMENT())

Due to the Structure Score, the first program is ranked higher than the second. There is an order mis-match in the English sentence and the sub-programs (referring to the CONCATENATE terminal) in the second program. There is also an overlap between the terminal connection BY and CONCATENATE in the program i.e. not correct in second program as per the parse tree of English statement. This score is much more useful in Ranking Programs once the dictionary mappings has been automated.

```
Data: Statement $S$, Dictionary $D$
Result: Ranked Set of Programs List, $RankRes$
$R_0 \leftarrow \phi$;
for each word's index $i$ in $S$ do
    $T \leftarrow D(S[i])$;
    for each $t \epsilon T$ do
        $R_0 \leftarrow R_0 \bigcup (t, Map(i,t))$;
    end
end
$Res \leftarrow Bag(R_0)$;
for each program $(P, M)$ in $Res$ do
    $s_{cov} \leftarrow CoverageScore(P, S, M)$;
    $s_{map} \leftarrow MappingScore(P, S, M)$;
    $s_{str} \leftarrow StructureScore(P, S, M)$;
    $score(P) \leftarrow max(\omega_{cov} \times s_{cov} + \omega_{map} \times s_{map} + \omega_{str} \times s_{str})$;
end
$RankRes \leftarrow orderedsetofprogramsinResbasedonscore$
return $RankRes$;
```

Algorithm 1: NL to DSL Synthesis Algorithm

3.1.4 Training Phase

The section describes the training process of the various classifiers mentioned in the Synthesis Algorithm above and how to combine these classifiers.

Mapping Score Classifier (C_{map})

The mapping score classifier predicts the probability of a word mapping to a terminal in the program P using the POS tag of the word. Naive Bayes Classifier has been used to train C_{map}. The algorithm involves generating all the consistent maps that are used to construct the program and then selecting the most likely map based on the *Likeability Score*.

$$LikeabilityScore = (Domain(M), Disjointedness(P, S, M))$$

$$Disjointedness(P, S, M) = \sum_{P' \, \epsilon \, SubProgs(P)} \sigma(P') \;\; where$$

$$\sigma((P1, ..., Pn)) = \begin{cases} 1 & \text{if } \forall \, Pi, \; Pj, \; Pi \cap Pj = \phi \\ 0 & \text{otherwise} \end{cases}$$

3.1.5 Structure Score Classifier (C_{str})

The aim of this classifier is to predict the probability that a combination *Comb* is an instance of connection *Conn* using the features of *Comb*. Learning of C_{str} is done using Naive Bayes Classifier. The algorithm generates all the consistent programs using Bag Algorithm (name it B) and uses all the combinations present in $P' \, \epsilon \, B$ but not present in P as negative examples and those combinations present in P as positive examples. In this way the training data for the Structure Score classifier is generated.

Combining various Classifiers

Weighted sum of the various scores is one standard way of combining the components. The problem of maximizing an optimization function is tackled by learning the weights.

Optimization Function: Total number of benchmarks in the training set for which the correct program is at rank 1

Stochastic Gradient Descent method is used to maximize the optimization function. An error function F_{error} has been defined to maximize the Optimization Function. The weights for the consecutive iterations uses F_{error} for the update.

$$w_{n+1} = w_n - \eta \Delta F_{error}(w_n)$$

where $n = 1, 2, ...,$ η is the learning rate and Δ is the gradient. The process of changing the weights is stopped when the change is below a threshold ϵ.

The error function is defined in the following way:

$$F_{error} = \sum_{\forall \; benchmarks \; S} f(w, S)$$

$$f(w, S) = \frac{1}{1+e^{-c(\lambda-1)}} \;\; where \; \lambda = \frac{v_{wrong}}{Score_{P_{correct}}} \wedge c > 0$$

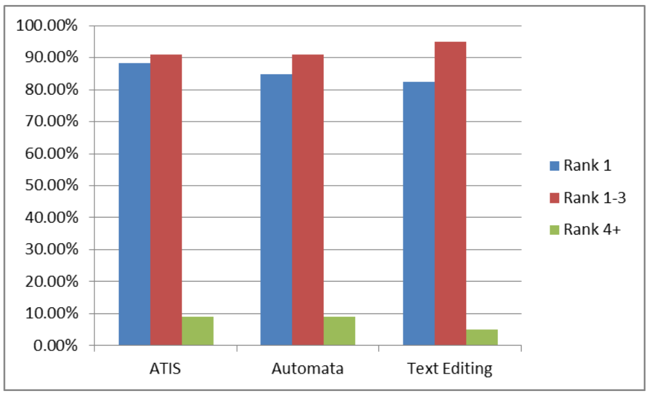


Figure 3.1: Ranking precision of algorithm on all domains.

$$P_{correct} = \text{ correct program } P \text{ of benchmark } S$$

$$v_{wrong} = max(Score(P) \mid P \epsilon Bag(S) \wedge P \neq P_{correct})$$

The error function is piece-wise continuous and differentiable. Though it is mostly well behaved, max function makes it discontinuous at some points. An approximation of the max function has been used to make it continuous.

$$max(a,b) \approx \frac{log(e^{ca}+e^{cb})}{c} \text{ where } c \geq 1 \text{ if } a << b \vee b << a$$

To ensure the correctness of gradient descent, we select large values of c when the scores between the correct program and incorrect programs become nearly equal (probably due to the large number of programs generated by bag).

The synthesizer was experimented on three domains in which users specify different types of tasks. The benchmark set used had over 1200 benchmarks. The precision of algorithm is over 80% for the top rank in all domains and for the top three ranks it was more than 90% 3.1. The average time 3.2 taken by the benchmarks of Text-Editing domain is 0.68 seconds, for ATIS domain it is 1.38 seconds and for Automata the average time for producing the correct formula was 1.78 seconds.

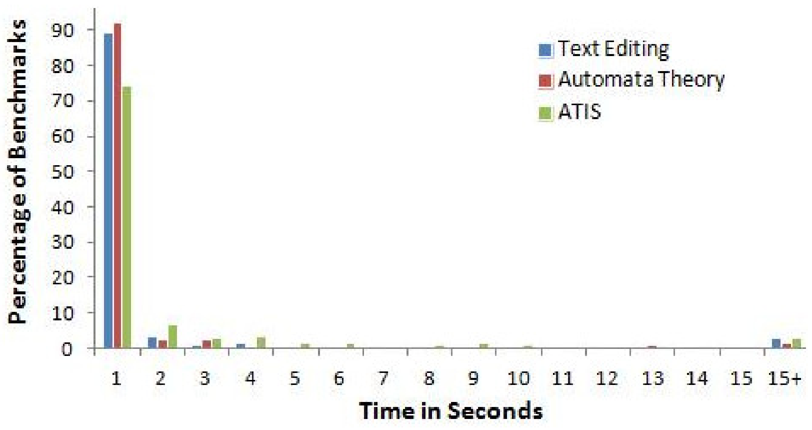


Figure 3.2: Timing performance of algorithm on all domains.

Chapter 4

Automating Dictionary Mappings

In this chapter, we describe the work contributed during this thesis. The dictionary that maps English words with DSL tokens is used in the bag algorithm to generate all the programs that are consistent with the English statement as an input to the synthesizer. We tackle the problem of reducing the effort to generate the dictionary mappings. In this chapter, we next describe a real world example showing the need of the automation in input building in the present framework.

The synthesizer framework described in the last chapter helps a developer to implement the NL-to-DSL synthesizer to a domain according to her needs. Imagine a Mathematics teacher at high school wanting to develop such a synthesizer for Geometry problems domain. She (or someone who is hired for the job) would have to design a Grammar for the domain which itself is going to take a lot of manual effort and knowledge about the context-free-grammars and the intricacies in designing one for the Geometry domain. The benchmark suite of English statements can be easily collected on-line or through various high-school textbooks, etc. The English sentences should then be translated to Domain grammar programs manually. The dictionary mappings to be used by the bag algorithm of the synthesizer have to be constructed by iterating over the benchmarks. Amidst all this manual effort, automation is needed wherever possible. After all, the goal of the whole project is to make programming an easier task through automation.

The automation of dictionary mappings construction is helpful in reducing the benchmark suite iteration (which consumes a lot of time) for getting correct mappings of each terminal. The domains we targeted had on an average 68 terminals [75 (Text Editing) + 42 (ATIS) + 88 (Automata)] and

159 mappings [164 (Text Editing) + 183 (ATIS) + 131 (Automata)] for over 1200 benchmarks (all domains added). Automating the work for the developer of NL-to-DSL synthesizer of the dictionary mappings building for all the terminals would prove very useful.

The automation of dictionary is not only specific for the NL-to-DSL Synthesis algorithm framework, it is also a generalized way of how Word based Machine translation can be done. We describe a general way to create a dictionary automatically (with minimal manual intervention) from a Natural language words to a Formal grammar programs, that can be understood by the Computer. Given a benchmark suite of Natural Language (NL) sentence and Formal language (FL) program pair, the algorithm is able to map NL words to FL tokens. The next section formally describes the goal of the thesis project.

4.1 Objective

The aim of this chapter is to automate the construction of mappings from English words to DSL tokens such that the results with the Automated Dictionary (AD, i.e. constructed with minimal manual effort) are not affected too much as compared to the results of the benchmarks' ranks with the Manual Dictionary (MD, i.e. constructed manually by iterating over every benchmark). The subjective notion of "results not getting affected too much" lies in the trade-off between the time-taken by a developer to construct the Dictionary for a new domain synthesizer and the difference between the results using the Dictionary constructed manually and automatically.
To achieve the above goal, we try to aim for constructing the set AD such as the mapping set is equal to the mappings set in MD.

$$MD \equiv AD$$

This chapter describes the algorithm used to semi-automate the Dictionary building process in the framework.

4.2 Pure Seeding

We will introduce this section by defining a few terminology:

- **Seeding**: A terminal T_i of the Domain Specific Language is mapped with some English words $w_{ia}, w_{ib}, ...$ This process of assigning maps for all the terminals is known as *Seeding* and the English words are known as *Seeds*.
- **Terminal Maps**: The English words that are mapped to the Terminals in the dictionary are referred to as Terminal Maps (TM).

$$TM(T_i) = w_{ia}, w_{ib}, ...$$

The algorithm 2 starts with seeding the terminals manually. The algorithm then iterates over the training set of pair of sentences and their correct translations. It tries to find out the words in the English statement corresponding to each terminal in the correct translation of that benchmark such that the word should be in the synonym-set of any of the words in the terminal's maps in the dictionary.
A dictionary D keeps on building as the English words are added in the terminal maps. The synonym-sets are found using WordNet. The path similarity is calculated using Sim_{Path} (referred in 2), giving the similarity scores of the words already present in the Terminal maps and the words of the English sentence. If the similarity score is above a threshold value ϵ, the word (from the sentence) is added to the present terminal map of the dictionary. An example for the process is shown below:

Insert "NM-" before 1^{st} *letter in every line*

INSERT(STRING(NM-), Position(BEFORE(CHARTOKEN()), INTEGER(1)), IterationScope(LINESCOPE(), ALWAYS(), ALL()))

Suppose above benchmark is encountered while iterating over the suite, the algorithm tries to find out a word for the terminal CHARTOKEN in the English sentence. Assume the present Terminal Map of CHARTOKEN contains the words "symbol, character, anything". The path similarity is measured between all words of this set with all the words of the sentence. The word 'letter' is found to be most similar to the word 'character' and Sim_{Path} gives 0.5 as the path similarity measure using WordNet. Hence, 'letter' is added in the Terminal Map of CHARTOKEN.

```
Data: Manually Seeded Dictionary $MSD$ in the form of terminal maps $TM_{MSD}$
Result: Automated Dictionary, $D1$ with $TM_{D1}$ as terminal map
$D1 \leftarrow MSD$;
$\epsilon \leftarrow WordNet\_Similarity_{threshold}$;
$TL \leftarrow EmptyList$;
/* TL is the list of tuples of the type (English word, DSL terminal, score value) */
for each benchmark $(S, P)$ do
    for each terminal $t$ in $P$ do
        for each word $w$ in $S$ do
            for each word $w'$ in $TM_{D1}(t)$ do
                $score(w) \leftarrow WordNet\_Similarity(w, w')$;
                if $score(w) > \epsilon$ then
                    Add $(w, t, score(w))$ to $TL$;
                end
            end
        end
    end
    $TL_{unique} \leftarrow$ A unique list from $TL$ (based on the word $w$ of $S$ and maximum score);
    for each tuple $(w, t, value)$ in $TL_{unique}$ do
        Add $(t, w)$ to $D1$;
    end
end
return $D1$;
```

Algorithm 2: Pure Seeding: Constructing Dictionary using Synonym Sets

The similarity threshold value ϵ is kept low (less than 0.5) in the above process. If the value is kept higher, the words having less similarity scores with the terminal maps are not included in the dictionary making some essential maps to be left out. If the dictionary misses these essential maps, even the Bag fails to generate the correct program and its recall value falls. Thus, to make Bag generate the correct program, we keep the ϵ value low.

However, keeping it low has a disadvantage of its own. While trying to include a terminal's essential maps in the dictionary, some superfluous maps also get included. This is what happens in practice when we try to cover a subset of the benchmark suite by lowering the threshold (thence making essential maps to be included in the dictionary).

$$\epsilon = 0.2$$

$$ManualSet = 164 \text{ maps}$$

$$AutomatedSet = 178 \text{ maps}$$

$$Manual - AutomatedSet = 11 \text{ maps}$$

$$Automated - ManualSet = 25 \text{ maps}$$

$$\epsilon = 0.8$$

$$ManualSet = 164 \text{ maps}$$

$$AutomatedSet = 133 \text{ maps}$$

$$Manual - AutomatedSet = 38 \text{ maps}$$

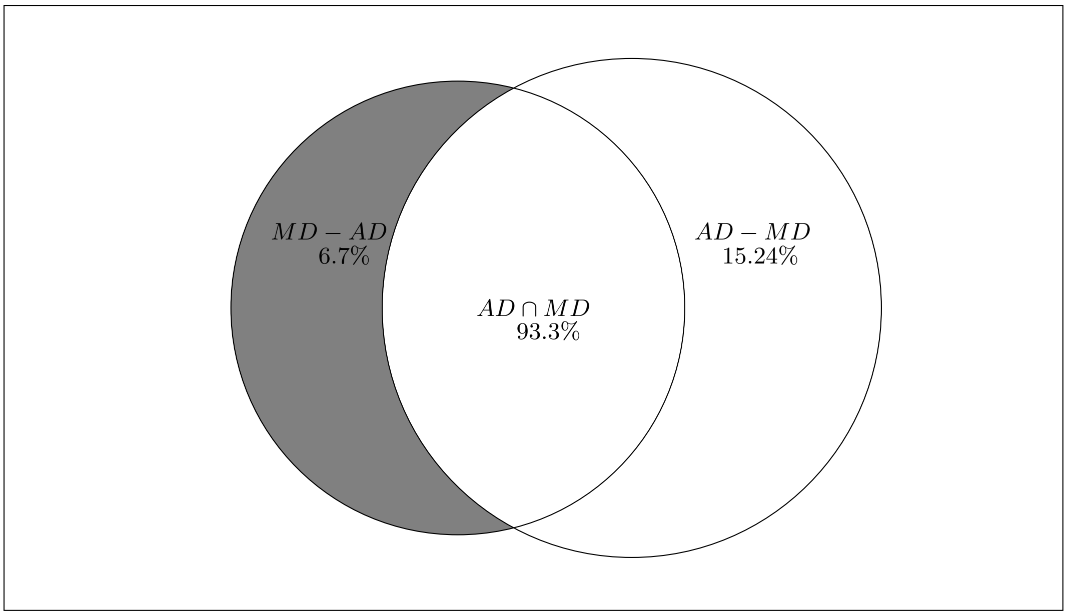
$$Automated - ManualSet = 14 \text{ maps}$$

The figure 4.1 shows the size of the sets - Manual Dictionary maps (MD) and Automated Dictionary maps (using Pure Seeding) (AD) with the Path Similarity threshold (referred to as $WordNet\_Similarity_{threshold}$ in the code) as 0.2 and 0.8 respectively for Text-Editing Domain. Shaded region is the set $Manual - Automated$. Comparing the figures, shows how the AD set changes with change in ϵ. As the example given above, Sim_{Path} gives path similarity measure between the words 'letter' and 'character' as 0.5. Automation could not have found the mapping of CHARTOKEN and *letter* if the threshold had been more than 0.5.
There is also a difference between the sets $AD - MD$ for the two thresholds. This is because such a low threshold also allows some spurious maps to be added to the dictionary and that becomes a disadvantage for us in the automation process.
We targeted for making the two sets (Automated Dictionary AD and Manual Dictionary MD) similar. But due to the above drawback, the set AD becomes larger than MD and hence we have to remove/*weed-out* the extra-unnecessary maps from the set AD. Here, we introduce the notion of *Weeding*.

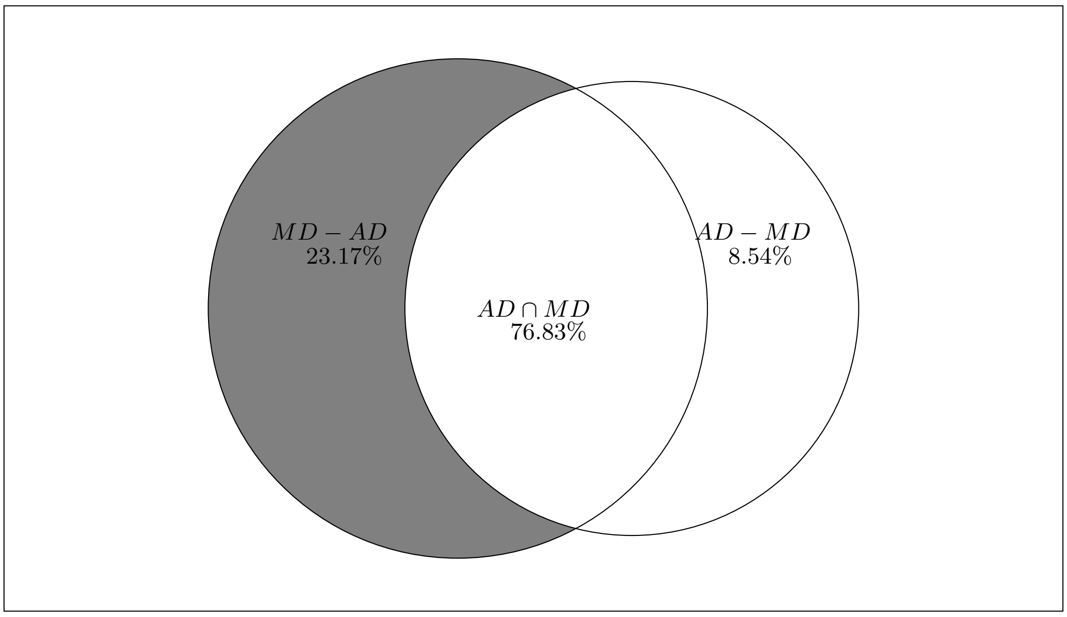
Weeding: Removing the unnecessary maps manually using domain knowledge.

Some of the maps are extremely unnecessary and can be removed manually just by iterating over the maps with the knowledge of benchmarks in hand. Weeding is done over the set D_1 after it is returned by the above algorithm.

Example: In ATIS Domain the terminal TIME gets mapped to "now" which is useless as per the grammar design. This mapping came because of the benchmark "*thank you very much now can you help me with information on flights from boston to atlanta*". We can manually weed out



$\epsilon = 0.2$



$\epsilon = 0.8$

Figure 4.1: Manual Dictionary (MD) and Automated Dictionary (AD) for Text-Editing Domain

the map as it can be of no use to the domain. Similarly, in Text-Editing domain, the terminal LINESCOPE gets mapped to "string". We can remove this mapping by using the domain knowledge that the user never refers to a statement as "string", rather they may refer to a single word as string.

The results even after the weeding did not look promising. Also, there is still a large amount of manual effort in seeding and weeding and so we had to differentiate between the importance of words in the seeded dictionary and the words learned.

4.3 Levelized Learning

The seeds given initially represent the terminals better than the words learned later through WordNet. This is because of the path similarity threshold value that is kept less. The words for which the similarity measure given by the Sim_{Path} is less are examples that they does not represent the seeds better. In this way, the unnecessary words or the less important (in terms of similarity) words learned due to less ϵ, give rise to more useless maps or even less important words and hence increasing the number of unwanted maps.

Example: *add "*" in beginning of all the lines wherever 'P.O. BOX' is present*

INSERT(STRING(*), START(), IterationScope(LINESCOPE(), CONTAINS(STRING(P.O. BOX)), ALL()))

In the above example, Sim_{Path} fails to give high similarity between the word 'present' and the words in the Terminal Maps of CONTAINS. Whereas, the words 'has' (learned during the method) and 'is' have a similarity measure of 0.33 as returned using the Sim_{Path} and hence it is added to the Dictionary. We can clearly see the user intention of using the word 'present' in the above benchmark.

If somehow we could keep a check on the addition of these superfluous words in Terminal Map list of terminals, we could reduce the size of AD thereby making it closer to MD.

The *more similar* words can be given importance in the following ways:

- **Level of Learning**: We define the '*level of learning*' in the following way. $Level_0$ of learning is the step where a word is learnt by taking the similarity measure from the seeds (let us call the seeds as L_0 words and the words learned through $Level_0$ learning as L_1 words). $Level_1$

learning is the step of learning words using similarity measure with L_1 words. Words learned through $Level_1$ are referred to as L_2 words. Similarly, L_{i+1} words are learned in $Level_i$ learning where the similarity measures are taken with L_i.

We give less importance to L_i words as compared to L_{i-1} words where $i > 0$ by using a $\delta_{importance} < 1$ and multiplying it to the score of the new words.

- **Pruning using word scores**: One more level of pruning can be done on the words learned by defining the score of seeds as absolute 1.0 and taking the score of learned word w (i.e. $score(w)$) as similarity measure with the word w' already present in dictionary times the score of w'. This helps in taking account of the measure of the word within the group L_i. Calculation of score of a word becomes a recursive function:

$$score(w) = \begin{cases} WordNet\_Similarity(w, w') \times score(w') & \text{if } w' \text{ in } SeededDictionary \\ WordNet\_Similarity(w, w') \times score(w') \times \delta_{importance} & \text{otherwise} \end{cases}$$

After using the Levelized Learning, words like in the above example would not be added to the dictionary and hence reducing the size of extra maps in automated dictionary. The above methodology also helps in the process of manual weeding. We illustrate the above by showing the unnecessary maps that were generated using *Pure Seeding* and were removed in *Levelized Learning.* The reason for the useless maps is the low threshold value and the various synonym-sets used by WordNet that gives high path similarity measure of L_1 words with seemingly random words. *Levelized Learning* tries to tackle this problem of *Pure Seeding.* Table 4.1 shows the maps that are not present in Automated Dictionary when it was constructed using Levelized Learning but were there in Pure Seeding methodology built Dictionary.

Full statistics for the Automated Dictionary for Text-Editing Domain are:

$$Manual\ List : 164 \text{ maps}$$

$$Automated\ List : 167 \text{ maps}$$

$$Manual - Automated : 10 \text{ maps}$$

$$Automated - Manual : 13 \text{ maps}$$

Terminals	**Maps**
CONTAINS	is
INSERT	substring
PRINT	numbered
END	place

Table 4.1: Example of Useless Maps from Pure Seeding removed in Levelized Learning for Text-Editing Domain

The algorithm 3 for this method is similar to the 2 except for the score updates. The words learned through seeds are given higher weight-age than the words learned through other words.

Weeding is done over $D2$ after the algorithm is run to get the automated dictionary AD. The amount of manual intervention in weeding reduces for this method and hence we gain by automating the process of dictionary construction more than the Pure Seeding method. But still there is a huge amount of effort in seeding the terminals in the dictionary. What if the seeding is also automated!

```
Data: Seeded Dictionary $SD$ in the form of terminal maps $TM_{SD}$
Result: Automated Dictionary, $D2$ with $TM_{D2}$ as terminal map
$D2 \leftarrow SD$;
$\epsilon \leftarrow WordNet\_Similarity_{threshold}$;
$\delta_{importance} \leftarrow 0.9$ $TL \leftarrow EmptyList$;
/* TL is the list of tuples of the type (word', terminal, score value) */
for each benchmark $(S, P)$ do
    for each terminal $t$ in $P$ do
        for each word $w$ in $S$ do
            for each word $w'$ in $TM_{D2}(t)$ do
                $score(w) \leftarrow WordNet\_Similarity(w, w') \times score(w')$ if $w'$ not in $TM_{SD}(t)$
                then
                    $score(w) \leftarrow score(w) \times \delta_{importance} \times score(w')$
                end
                if $score(w) > \epsilon$ then
                    Add $(w, t, score(w))$ to $TL$;
                end
            end
        end
    end
    $TL_{unique} \leftarrow$ A unique list from $TL$ (based on the word $w$ of $S$ and maximum score);
    for each tuple $(w, t, value)$ in $TL_{unique}$ do
        Add $(t, w)$ to $D2$;
    end
end
return $D2$;
```

Algorithm 3: Levelized Learning: Importance of initial level learnings

4.4 Automated Levelized Learning

The seeds usually are derived from the name of the terminals. The terminal AFTER of the Text-Editing domain is given the word "after" as its seed and learning is then done over these words. Table 4.2 shows various seeds for the terminals of ATIS domain. The seeds are of two types in ATIS domain. One is given by the developer as shown in table 4.2 and other type is taken from the airline database shown in table 4.3

GIZA++[11] has been used for seeding the terminals. In the algorithm of Automated Levelized Learning, we get a Seeded Dictionary (*GSD*) returned by the GIZA++ tool. The *GSD* is constructed by having a threshold γ for the similarity (also known as *joint occurrence*) measure of word and terminals used after the GIZA++ gives all the scores. This is done as GIZA++ gives

Terminals	**Maps**
BETWEEN_CITIES	between
BREAKFAST	breakfast
COL_AIRCRAFT_TYPE	aircraft
COL_AIRCRAFT_TYPE	equipment
COL_AIRLINES	airline
COL_ARRIVAL_TIME	arrivals
COL_FARE	expensive
EQ_FOOD	serves

Table 4.2: Example of Seeds for ATIS Domain

Terminals	**Maps**
AIRCRAFT	727
AIRCRAFT	747
AIRLINES	delta
WEEKDAY	Thursday
WEEKDAY	Tuesday
AIRLINES	eastern
CITY	denver
CITY	detroit

Table 4.3: Example of Database Seeds for ATIS Domain

Terminals	**Maps**
COL_AIRLINES	airlines
COL_ARRIVAL_TIME	time
COL_STOPS	stop
EXTRACT_ROW_MAX	highest
EQ_AIRCRAFT_TYPE	aircraft
COL_AIRCRAFT_TYPE	aircraft

Table 4.4: Example of Extra Trivial Maps for ATIS Domain added after Learning

the joint occurrence measure for each word and every terminal. A [automated] preprocessing step comes after this which removes some unwanted words i.e. some unwanted articles, pronouns, etc. from GSD giving us SD which is given to the algorithm to construct the automated dictionary $D3$.

In this method, even though the threshold for Sim_{Path} learning is kept low, there are a few maps (less than 10% of the dictionary maps of a domain) that are missed but are necessary in the dictionary. This is because the seeded dictionary generated by GIZA++ is not equivalent to the set of Manual seeds.

Example 1: *what is the latest evening flight leaving san francisco for washington*[1]

EXTRACT_ROW_MAX_T(COL_DEPARTURE_TIME(),

AtomicRowPredSet(AtomicRowPred(EQ_DEPARTS(CITY(francisco), Unit_Time_Set(TIME(evening)), ...),

EQ_ARRIVES(CITY(washington), ...))))

Example 2: *i need information on ground transportation from the airport in atlanta*

PROJECT(AtomicColSet(COL_TRANSPORT()), AtomicRowPredSet(EQ_AIRPORT(CITY(atlanta))))

The first example above shows that the word 'for' in English statement refers to the place where the flight would be arriving and hence it is mapped to EQ_ARRIVES terminal while seeding the dictionary. Similarly, in the second example sentence the word 'need' has been used which should be mapped to PROJECT and hence, these maps are added to the dictionary. The list of some such maps has been shown in the table 4.5.

More than 50% of the extra maps are trivial as they were included in the seeded dictionary when seeding was done manually in the first two methods. They are called *Trivial maps* in the sense that they are words that are taken directly from the Terminal name 4.4. Few sentences to show the occurrence of these maps are:

Example 1: *i would like to know if i fly on american flight number 813 from boston to oakland if i will stop enroute at another city*

PROJECT(AtomicColSet(COL_STOPS()), AtomicRowPredSet(AtomicRowPred(EQ_DEPARTS(CITY(boston),

[1]The Domain specific language programs have been beautified for the ease of readers. Exact programs have other terminals which are unnecessary to the discussion here.

...), EQ_ARRIVES(CITY(oakland), ...)), Eq_Aircraft_Type_IMP(AIRCRAFT(813)),

Eq_Airlines_IMP(AIRLINES(american))))

Example 2: *show me the flight from detroit to westchester county with the highest one way fare*

EXTRACT_ROW_MAX(AtomicColSet(COL_FARE()),

AtomicRowPredSet(AtomicRowPred(EQ_DEPARTS(CITY(detroit), ...), EQ_ARRIVES(CITY(westchester), ...))))

Example 1 above queries about the stops the flight has and hence the terminal COL_STOPS is mapped to the word 'stop'. In example 2, user asks about the max fare of the flight using the word 'highest', mapping it to the terminal EXTRACT_ROW_MAX.

After the construction of $D3$, we try to remove the unnecessary maps by weeding. It takes another amount of manual effort for weeding. While the initial aim of Automating the Dictionary was to generate the maps without affecting the results much, generated using the manual dictionary. We tried using the $D3$ set as it is, without weeding, for the framework.
We did so because even if there are extra-maps present in the Dictionary, the Bag algorithm would be generating a lot of programs making the process of translation slower. Now, during the ranking of the set generated by the Bag, the scores used by Ranking algorithm helps in giving correct scores to the programs.
For example, the set $AD - MD$ contains a mapping from PROJECT to the word 'ground'. This mapping is an outcome of the similarity given by the Sim_{Path} between the words 'show' and 'ground'. Now, as the mapping score gives more mapping probability of 'ground' to COL_TRANSPORT, the Ranking Algorithm making use of the mapping score ranks the program with correct mapping above the programs having incorrect mappings. The sophistication of the various features used in the Structure score also helps in ranking the correctly mapped programs over others.

We found that the precision 4.2 4.3 for the top rank was affected by 3-5%. The percentage of programs in top three positions also dropped. Though, the recall of the bag algorithm is not affected as there are all the *necessary* mappings needed to generate the correct program as with the manual dictionary but the number of *Valid Programs* generated by Bag increases most of the times (and the number increases exponentially in some cases). Due to the rise in the number of programs in bag set, the scores assigned by the ranking algorithm to the correct program were

Terminals	**Maps**
EQ_DEPARTS	go
COL_AIRCRAFT_TYPE	equipment
COL_ARRIVAL_TIME	time
EQ_ARRIVES	destination
PROJECT	need
PROJECT	find
CLASS	first
EQ_ARRIVES	for

Table 4.5: Example of Extra-Maps for ATIS Domain

almost similar to some incorrectly mapped programs.

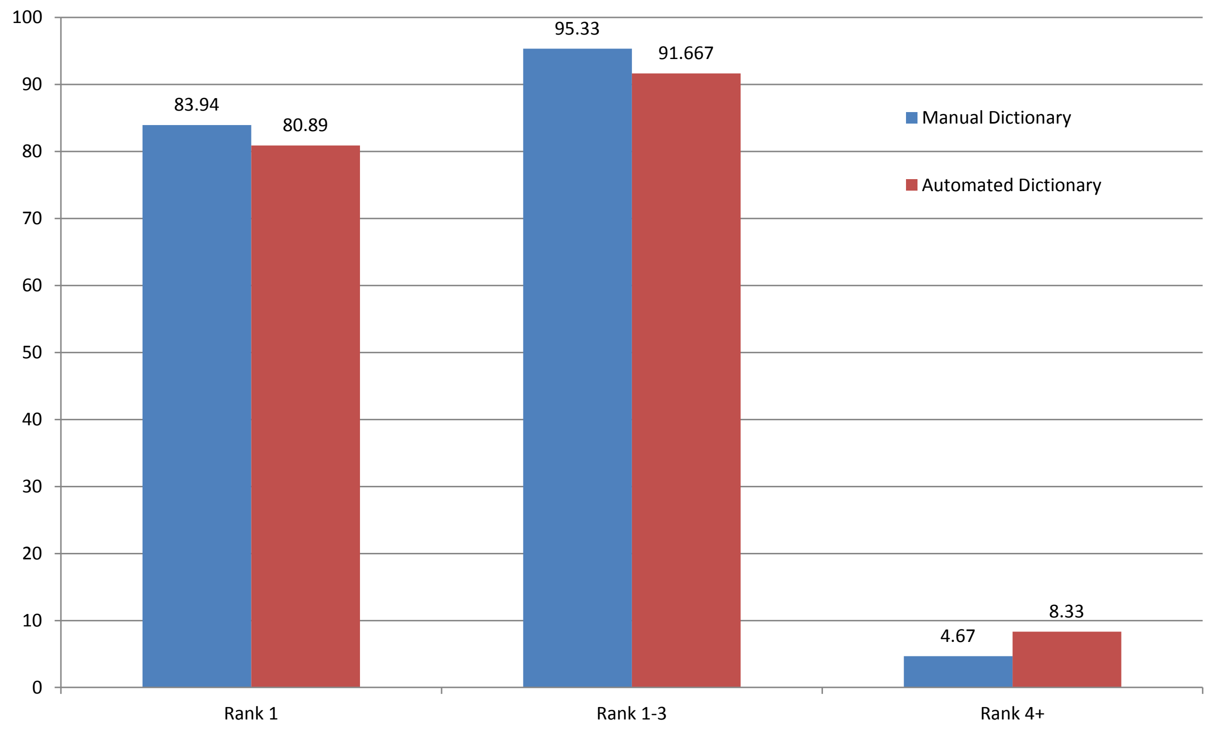


Figure 4.2: Comparison of Rank Results after Automated Dictionary for Text-Editing Domain

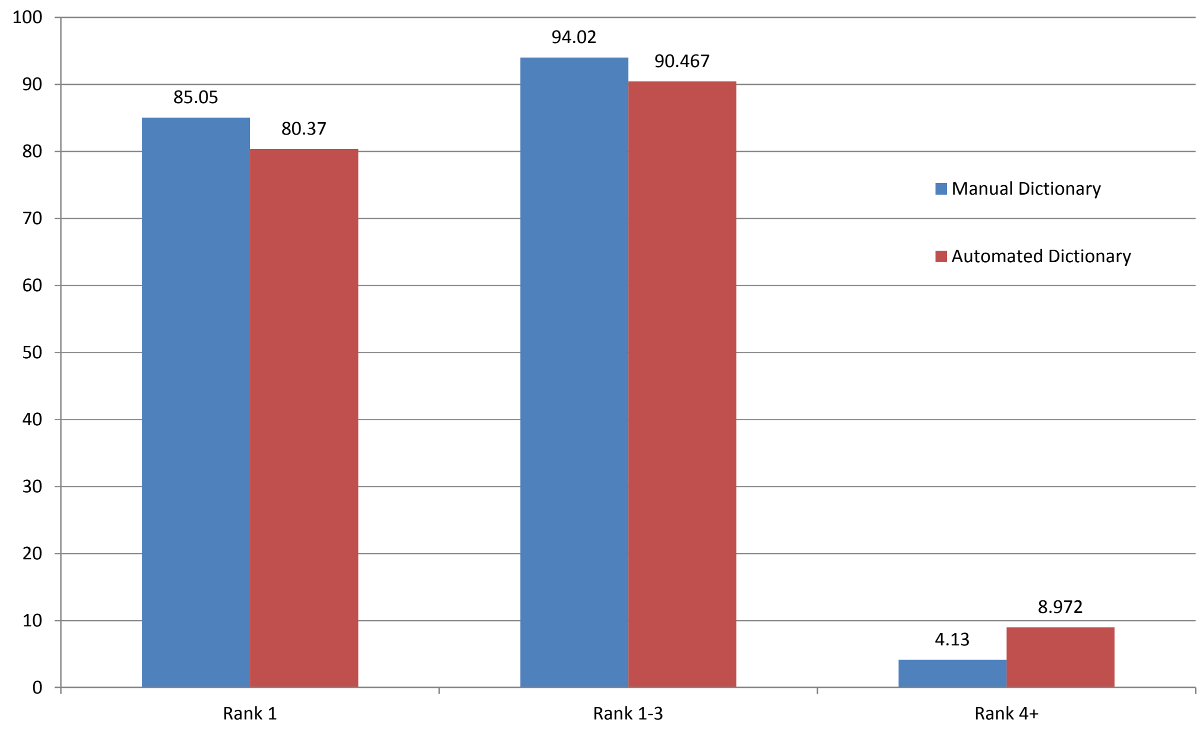


Figure 4.3: Comparison of Rank Results after Automated Dictionary for ATIS Domain

Chapter 5

Results and Discussion

5.1 Experimental Setup

For the implementation of the methods for Automating Dictionary Mappings described above, we have used Python version-2.7.3. The WordNet tool is used as a corpus reader for NLTK library of python which is a platform for building python programs for the Natural Language Processing techniques. Stemming has also been used. As an example, the implementation of the WordNet API (i.e. the tool Sim_{Path}) shows the path similarity between 'work' and 'working' to be 1.0.

5.2 Results

The results of the precision over ranking algorithm were shown in the last chapter. Here we have compared the timing results of both the Ranking and Bag algorithm. The total time taken by the bag algorithm for all the benchmarks increased manifold, but by looking at per benchmark scenario 5.15.2 it shows that the time for most of the outliers (that were already taking a lot of time) have increased. The benchmarks in the 2-5 seconds bucket dropped a little when the automated dictionary was used. The set of inputs in more than one minute bucket increased significantly. There are benchmarks that were taking a lot of time (more than one minute) while using Manual dictionary in Bag algorithm. This is because the users have described these sentences in a very redundant manner. The increase in time for such benchmarks is very high 5.15.2 when

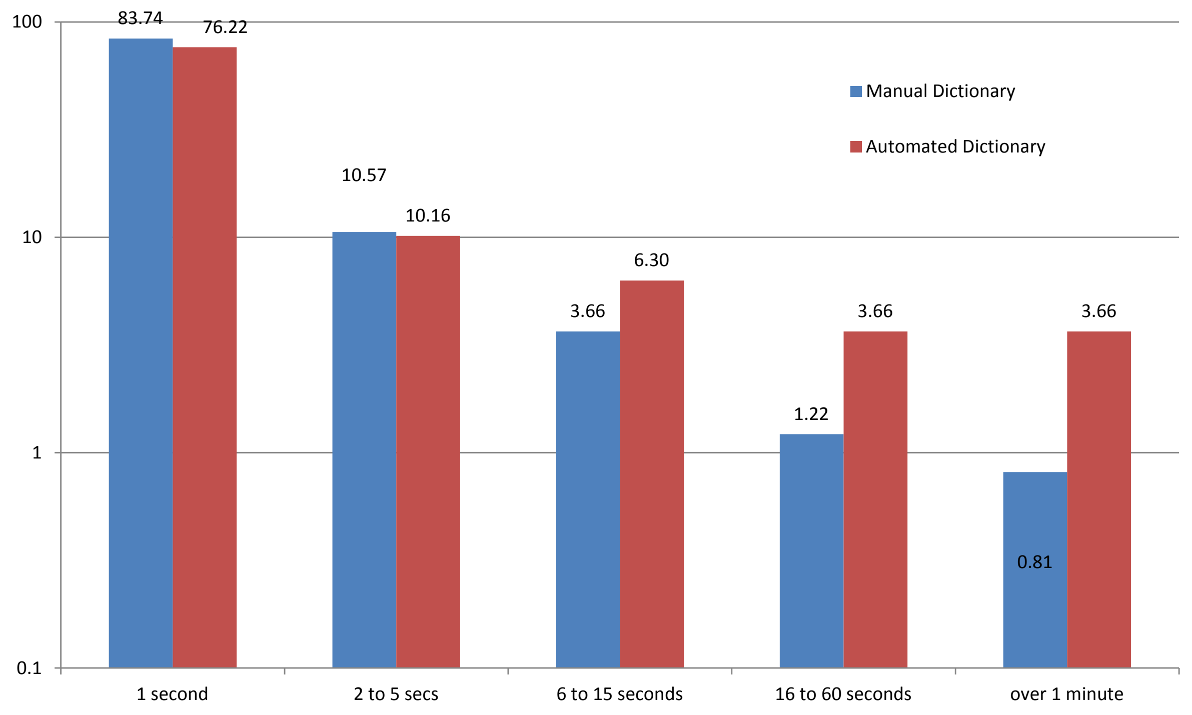


Figure 5.1: Comparison of time taken by the Bag Algorithm using Manual and Automated Dictionaries for text-editing domain

the Automated Dictionary is used. This increase in time for redundant benchmarks was the main reason for the increase in total time of bag algorithm.

We observed that the total time taken by the ranking algorithm also increased by 6-8 times. The main reason behind this was that the set of programs generated by the bag algorithm increased exponentially for the redundant benchmarks. Ranking algorithm had to assign scores to each of these programs, hence increasing the time.

Reason for time increase: The increase in time for bag after automating the maps is because the Bag algorithm used in the Synthesis Framework described is a brute force algorithm taking a lot of time and generating semantically meaningless programs. If there were some kind of semantic checker which could keep a check over the type of programs generated by bag, then most of the benchmarks from the bag set would not have blown up so badly. This exponential blow up of the

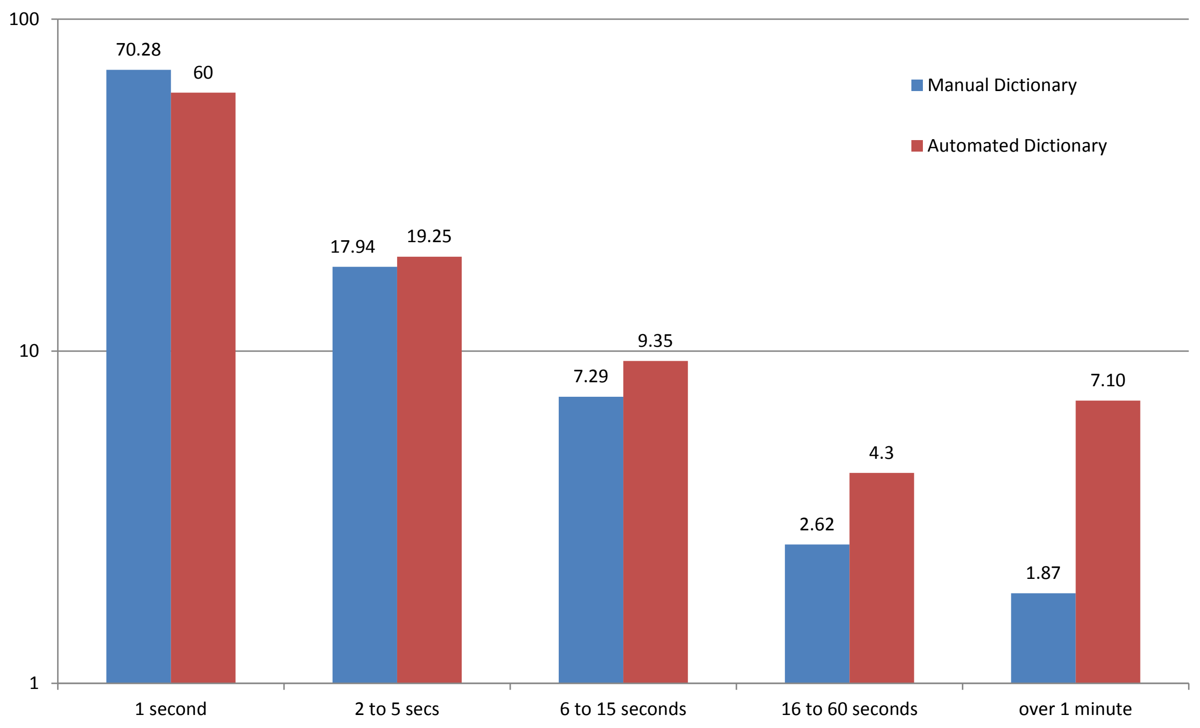


Figure 5.2: Comparison of Time taken by the Bag Algorithm using Manual and Automated Dictionaries for ATIS Domain

Bag set of programs is the reason for such an increase in the overall time taken by the Ranking algorithm.

5.3 Quantifying Manual Effort

We lost the precision for both the Text-Editing and ATIS domain by a few percentages but we gained a lot in terms of time taken by the developer to start with a new domain. Assume that iterating over one benchmark takes unit time κ and adding/removing a map in the dictionary takes unit time ν (where $\kappa > \nu$). We quantify the time taken to develop a domain for various methods as described below: [1]

1. *Manual Dictionary*: The developer has to iterate over all the benchmarks at least once. Assuming there are 500 benchmarks in this domain (and iteration over all the benchmarks takes same unit time, which may not be true but assume that the least time taken by any benchmark is κ) and the total maps in dictionary are about 170 (taken as an average of ATIS and Text-Editing domain), the total time taken by the developer would be $500\kappa + 170\nu$.

2. *Pure Seeding*: Here, the developer has to seed the terminals initially and weed out some of the mappings after total learning. Assuming seeding is about 1.5 maps per terminal (which is a fair assumption based on the domains we have) and there are about 50 terminals in the domain, in total 75 maps would be seeded. Also, assuming the weeding is about 20 maps (a rough estimate) the total time taken by the developer would be 95ν.

3. *Probabilistic Seeding*: The developer still has to seed the domain but assuming the amount of weeding decreases by 50% compared to the Pure Seeding the total time taken by the developer would be 85ν.

4. *Automated Probabilistic Seeding*: The developer does not seed and weed in this process but adds a few necessary maps at the end of the learning. Assuming the set of maps added contains about 15 maps (a fair assumption as ATIS needed 19 maps and Text-Editing domain needed

[1]This estimated quantification is just for the sake of showing the trade-off between manual effort and precision lost. We did not do any user-study to be able to estimate it precisely. The rough estimation done is for the sake of readers.

4 maps to be added after Automated Probabilistic Seeding), the total time taken by the developer would be 15ν.

The above quantification shows that we gain a lot from Pure seeding than in Manual Dictionary and from Automated Probabilistic Seeding than in Probabilistic Seeding.

5.4 Failure for Automata Domain

We described the results for Text-Editing and ATIS domain but there was also an implementation of Automata domain in the Background work. The reasons for failure of the algorithm in Automata Domain include:

- **Benchmark Suite**: The benchmark suite of Automata domain contains only 235 benchmarks which is very less for GIZA++ to perform learning. The tool needs at least 400 benchmarks as experimented to perform better machine translation.
- **Benchmark v/s Terminal Ratio**: The number of terminals in Automata domain is very high (88) i.e. almost one-third of the total number of benchmarks which is not sufficient to do any kind of learning.
- **Maps v/s Terminal Ratio**: While the number of terminals in Automata domain is highest among the three domains, the number of maps is the lowest (131). This makes it even harder for Sim_{Path} to distinguish between the correct mappings and incorrect mappings due to the similarity scores being similar (of the words in terminal maps of wrong terminals and seeded words versus the words in terminal maps of correct terminals and seeded words).
- **Benchmarks' Structure**: Most of the benchmarks of the Automata domain 5.1 do not form proper English queries, rather they are factual statements about the property of particular Finite Automata strings. There is a large difference between these benchmarks and the benchmarks of the other two domains due to this. It is difficult to do the kind of Machine Translation we have targeted while automating dictionary mappings.

Due to the above reasons, the results were extremely poor, thereby removing any possibility of automating the domain.

1. Consider the language L consisting of words containing “010”.
2. A string w belongs to the language L precisely when w contains at least 1 “a” symbol and does not contain any “b” symbols.
3. The set of strings over alphabet 0 to 9 such that the final digit has not appeared before.
4. w has the same number of occurrences of “10” and “01”.
5. Every odd position of w is a “1”.
6. Set of x such that any “a” in x is followed by a “b”.
7. x begins and ends with the sequence “aab”.

Table 5.1: Sample benchmarks for the Automata Domain

Chapter 6

Conclusion and Future Work

We addressed the problem of automating the mappings from the Natural Language words to formal domain specific language tokens. We used two corpora of benchmarks including simple text-editing tasks and the Air Travel Information System benchmarks with their translations in the domain specific language expressions. We used a hybrid approach to automate the dictionary mappings from one language to other. The automation was not equivalent to the manual effort for building the mappings but we showed that the precision was affected only by 4-5%. One aspect of the future work can be incorporating the domain knowledge into the system to automate the mappings such that there are no extra-unnecessary mappings.

The Automated Probabilistic Method requires a little amount of manual intervention for including some of the necessary maps in the dictionary. The reason behind this could be the size of data in a corpus which is very less for doing any statistical translation successfully. An interesting aspect of the future work can be to use semantic information in the algorithm. The semantic information can be provided from both the domain knowledge and the type of sentences to perform the tasks in the domain. Using this may help a lot in automating the dictionary completely without any manual intervention.

The algorithm designed did not perform well in the Automata domain. The reasons for the same were detailed. Future work can focus on the problem of targeting dictionary automation with a little amount of manual effort for a particular domain with properties like this domain. This would prove very useful compared to constructing the complete dictionary manually.

Another interesting problem for the future work could be validating the proposed algorithm on a more sophisticated bag of words approach for a particular domain that uses semantic checking on the programs without blowing up the Bag set.

Bibliography

[1] Yaser Al-Onaizan, Jan Curin, Michael Jahr, Kevin Knight, John Lafferty, Dan Melamed, Franz-Josef Och, David Purdy, Noah A Smith, and David Yarowsky. Statistical machine translation. In *Final Report, JHU Summer Workshop*, volume 30, 1999.

[2] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1976–1982. AAAI Press, 2013.

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python.* O'Reilly Media, Inc., 2009.

[4] Emile Delavenay and Katharine M Delavenay. *An introduction to machine translation.* Thames and Hudson, 1960.

[5] Nidhi Jain. A Generic Framework for Translating Natural Language Descriptions into Domain-Specific Logical Expressions, May 2013. M.Tech Project Report. She is a co-member of this project.

[6] Daniel Jurafsky and H James. Speech and language processing an introduction to natural language processing, computational linguistics, and speech. 2000.

[7] Amir Kamran. Hybrid machine translation. 2013.

[8] Adam Lopez. Statistical machine translation. *ACM Computing Surveys (CSUR)*, 40(3):8, 2008.

[9] Mehdi Manshadi, James Allen, and Mary Swift. A corpus of scope-disambiguated english text. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 141–146. Association for Computational Linguistics, 2011.

[10] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[11] Franz Josef Och and Hermann Ney. Giza++: Training of statistical translation models, 2000.

[12] Sailesh Kumar Raju Ramayanam. Applying Machine Learning to Rank Domain-specific Logical Expressions of Natural Language Descriptions, May 2013. M.Tech Project Report. He is a co-member of this project.

[13] Wikipedia. `http://en.wikipedia.org/wiki/Natural_language_processing`.

[14] Wikipedia. `http://en.wikipedia.org/wiki/Program_synthesis`.

[15] Wikipedia. `http://en.wikipedia.org/wiki/Machine_translation`.

[16] Wikipedia. `http://en.wikipedia.org/wiki/Rule-based_machine_translation`.

[17] Wikipedia. `http://language.worldofcomputing.net/machine-translation/rule-based-machine-translation.html`.

[18] Wikipedia. `http://en.wikipedia.org/wiki/Statistical_machine_translation#Word-based_translation`.

[19] Wikipedia. `http://en.wikipedia.org/wiki/WordNet`.

[20] Yorick Wilks. *An artificial intelligence approach to machine translation.* Springer, 2009.