Mapping sentences onto logical form

Dennis Laks
D04, Lund Institute of Technology
Sweden
d04dl@student.lth.se

Olof Olsson
D04, Lund Institute of Technology
Sweden
d04oo@student.lth.se

Abstract

This paper gives a short introduction to statistical machine translation, logical forms and parsing. It describes a project which goal was to recreate the results of previous work in the field of statistical machine translation. In this project we have made use of the pre written software GIZA++ and and fair deal of pearl programming. The result of our work is still far from the level of previous work but manages to translate parts of sentences correctly.

1 Introduction

As the amount of computer users grow we find a need for high usability in computer software. A possible way of giving unexperienced computer users direct control of software is to let them ”speak” to it. If we can teach the software to understand the users natural communication we can achieve very high usability and help users do what they intend. As an example we could imagine a software that could answer a plain text question about the weather such as: ”Is it going to rain tomorrow?” The software could then find information from a weather forecast and give the user information directly answering the question. Previous work in this field by L. Zettlemoyer and Michael Collins (Zettlemoyer, 2009) has managed with a good result to build a software that answers plain text questions about air traffic. In our project we try to recreate the result of aforementioned software and by doing this, learn the problems of the field. We will in this paper describe our work process, present our results and finally discuss about possible future improvements.

2 Corpus

The corpus used in our work is an ATIS corpus containing about 4500 sentences. The sentences are utterances used in a flight booking system where the user often wants to get a flight from one airport to another. The collection of sentences was manually annotated by L. Zettlemoyer and Michael Collins (Zettlemoyer, 2009) into lambda calculus form. In this particular syntax the usual lambda form features of variables or entities and relations between them are supported by symbols $0, $1 ... and parantheses. During our work we changed the corpus somewhat for our experiments. This is discussed more later.

3 The Logical Form and the Lambda Form

The logical form can be said to be an abstraction level in which one represents the essence of a natural language sentence using formal grammar. This is often used in order to remove unnecessary words or information and replace the essence with formal symbols. Logical forms is mostly used in computers and is necessary if the goal is for the computer to understand the semantic of information. Common logical forms are programming languages, such as Java, C or SQL.

The lambda form is one type of logical form that is widely used in, amongst other, computational linguistics and is suitable to use in computers. We used the lambda form as well for this project. The choise of our used syntax was based on available corpora we could use for our dictionary training and translation. The corpus we got from Zettlemoyer proved to be very uselful, with a straight forward syntax.

The lambda form describes variables and relations by grouping variables together with some keywords. The keywords can be a name of a place for example an airport or a city. It can also be a

direction like to or from.

To give a quick understanding, we give an example of a plain text sentence and its lambda translation equivalent:

"List all flights from Chicago to Milwaukee"

(lambda $0 e (and (flight $0) (from $0 chicago:ci) (to $0 milwaukee:ci)))

4 Giza++

GIZA++ is an extension of the program GIZA which was developed by the Statistical Machine Translation team in 1999 at the Center for Language and Speech Processing at Johns-Hopkins University. GIZA is a Statistical machine translation (SMT)toolkit that takes two corpora as input and maps words from the two corpora together. GIZA then outputs a dictionary containing all mappings from the two corpora. To use GIZA++, one first needs to transform the corpus to GIZA++ format, and then use the program mkcls in order to produce word classes. When this is done GIZA++ can begin to create a dictionary that maps one language to the other.

In the resulting mapping from GIZA++, each input sentence from the corpus outputs three lines:

1. Information about sentence lengths and alignment score.
2. The translation sentence (in lambda form) into which GIZA++ tried to align a sentence.
3. A list of the plain text words and to which translation words this word has been mapped. The mappings are shown as a list of numbers relating to word indices in the translation sentence.

An example of how the output from GIZA looks like can be seen in Figure 1. This example is on the sentence "what airlines from washington dc to columbus".

5 Our Parser

When we now had the bilingual mapping from GIZA++, we were to write the program that can take an arbitrary natural language sentence as input and, using the mappings, produce the logical translation in lambda form. It may be noted that since the GIZA translation only used sentences from an ATIS corpus, the dictionary is of course also suited to be used to translate equivalent sentences. Inputing another arbitrary sentence, e.g "I wish Santa Claus were here today", would only produce nonsense.

To prepare the input of our parser, which takes two files, one consisting of the translated sentences and one consisting of their mappings, we split this mappings file into a *lambda* file and a *parse* file using a trivial perl script.

Our main algorithm works in three stages:

- Data collecting
- Sentence building
- Post processing

We will now go through them and describe what they do. The code is located in Appendix and could be handy to view at the same time for a clearer understanding.

5.1 Data collecting

In this first step we build a form of dictionary in a matrix data form. The matrix consists of a list of all possible translations for any given plain-text word. This means that it is used to searching through the matrix for a plain-text word, and if it exists, all possible translations this word has is given as a list.

The dictionary is built by for all plain-text words, reading out the mapping numbers and storing the corresponding lambda translations in the translation list in the matrix for that particular word. When this has been done for all training sentences, the dictionary is built.

5.2 Sentence building

In this step the algorithm takes a plain-text sentence as input and breaks it down into words. For every word, we go through the dictionary and retrieve the list of the word's possible translations. We have here decided to use a probabilistic approach. This means that for a word with all of its translations, we collapse the translation list to only include uniqe translations and add a number that keeps track of the number of occurrences of this translation for this word. The information on the frequency of each possible translation tells us thus how probable a specific choise of translation is. We then take the most probable translation of every word and concatenate into the translated sentence. The resulting lambda form sentence is the per-word most probable translation.

Sentence pair (29) source length 7 target length 19 alignment score : 2.93896e-19 (lambda $0 e (exists $1 (and (from $1 washington:ci) (to $1 columbus:ci) (= (airline:e $1) $0)))) NULL ({ 2 3 6 16 17 18 19 }) what ({ 1 }) airlines ({ 4 5 8 11 13 14 15 }) from ({ 7 }) washington ({ 9 }) dc ({ }) to ({ 10 }) columbus ({ 12 })

Table 1: An example output from GIZA

5.3 Post processing

Since there is no real possibility to translate word by word into lambda form, due to e.g grammar, the lambda form generated by our algorithm is bound to be broken in some way. From our results we concentrated on two main faults:

- Paranthesis closure: In most cases the resulting lambda form sentence tended to be unbalanced and have more opening than closing paranthesis
- Variable insertion: The resulting lambda form sentence tended to lack variables ($0, $1 et.c)

We decided, due to the limited time for this project, to focus on these two faults in our translations and try to incorporate a fast and simple way of accounting for the most obvious ways they occur.

To close off parantheses, we count the number of opening and closing parantheses in a translated lambda form sentence and simply add the differentiating number of closing parantheses at the end of the sentence, given that there were more opening than closing parantheses.

In the lambda form notation used, variables are very often used within closed parantheses to represent a noun within, for example, `(to $1 columbus:ci)` or `(month $0 august:mn)` to represent a place, Columbus, or a month, August. In our post processor, we try to recover the missed variables within closed parantheses by finding occurences in the form `(*** ***)`, meaning "opening paranthesis followed by something, a space, something and a closing paranthesis". In this case we insert a $0 in the middle.

Our post processing is on a rather basic level, and therefore the paranthesis insertion for example is not always correcting a sentence in a semantically correct fashion. This comes from the fact that we always insert the closing parantheses at the very end of the sentence, while they might instead be needed somewhere within the sentence. The variable insertion has a surprisingly effective impact on the result. In more or less any query of this type in lambda form has at least one occurrence of the patterns we search for in order to insert a variable and very often the pattern should include a variable that is almost always missed by our parser. So by just inserting a variable we can be quite confident on that the editing is reasonable with respect to the real lambda form. We discuss improvements on these corrections a bit more later.

6 Results

In order to get a good understanding of our algorithms preformance, we ran all the training set sentences through our program. Later we ran a small set of newly written ATIS sentences as well and found the results to be similar to the corpus sentences.

In Table 2 one can see a few examples of how the algorithm performed. We denote the plain text sentence with Q, the algorithm output with A, and for a comparison, the manually writted translation from the corpus with FACIT. The achieved score is also written.

GIZA's bilingual mapping often mapped parantheses and variables to NULL, meaning no translation. We tried to correct this problem by creating a new GIZA translation from a corpus where we had removed multiple ending parentheses. The translation did map less words to NULL but it didn't make a big difference to the final result. The NULL translations are of course a major concern since we believe it to be a major reason for the somethimes awfully broken lambda syntax the results from out parser. We have unfortunately not come up with any reason for the NULL mappings, but the only thing we can point at is GIZA and that its algorithms may not be fit to translate between natural language and the lambda form syntax used here.

A mean value of the scores from both corpora was calculated. This score is calculated in the fol-

1
Q: can you list all flights from chicago to milwaukee
A: (lambda $0) (lambda e (flight (from $0 chicago:ci) (to)))
FACIT: (lambda $0 e (and (flight $0) (from $0 chicago:ci) (to $0 milwaukee:ci)))
Score: 0.6321
2
Q: what flights are there from minneapolis to newark on continental
A: (lambda (flight) (lambda (from $0 minneapolis:ci) (to $0 newark:ci)))
FACIT: (lambda $0 e (and (flight $0) (airline $0 co:al) (from $0 minneapolis:ci) (to $0 newark:ci)))
Score: 0.7460
3
Q: please list all the takeoffs and landings for general mitchell international
A:) (lambda)) mke:ap)) (to) mke:ap))
FACIT: (lambda $0 e (and (flight $0) (or (from $0 mke:ap) (to $0 mke:ap))))
Score: 0.1022

Table 2: Examples of how the parser performed. The third example is of a specially bad translation. Legend: Q: plain text sentence, A: algorithm output, FACIT: correct translation

lowing manner:

- For every word i, a probability of the chosen translation, $P(i)$, is calculated by dividing the frequency of this translation with the total frequency of the word (i.e sum of the frequencies of all the words translations).
- The total probability for a sentence of length n is calculated by $score = P(0) * P(1) * \ldots * P(n)$

The mean score calculated from all ~ 4500 sentences was:

Corpus including paranthses: 0.3219

Corpus excluding parantheses: 0.2823

This doesn't necessarily mean that the first attempt got a better result. The score means that the the chosen word was chosen by GIZA with a high frequency. In the second attempt we had a higher number of words to choose from in the parsing process, due to less words being mapped to null, and therefore accumulated a lower score. This measurement is far from ideal when measuring the correctness of the translations. Instead the amount of correctly assembled parentheses could be counted and be used as a score. The better measures were unfortunately not implemented since we had very little time for the project.

7 Related Work

Our parser was done in order to see how good results would be achievable with a rather rudimentary statistical algorithm. We were inspired by Zettlemoyer and Collins and their work on mapping context dependent sentences to logical form (Zettlemoyer, 2009), which definitely is a more advanced problem with a need of a more sophisticated solution than our work could comprehend. Yet Zettlemoyer and Collins reached an impressive 83.7% correct recovery of the logical form. This number may, once again, not be fully comparable with our 32.2% since we have used a different method to compute recoverability.

8 Future Work

An important extension would be to first of all implement a method of more accurately calculate the actual performance of this algorithm, for example using the aforementioned method.

8.1 Improvements

There are three major parts to expand in the parser. The parenthesis correction, the variable insertion and a control of keywords.

The parenthesis correction is now only inserting closing parentheses into the end of the sentence. With more logic the parser could place the ending parentheses more accurately. This would greatly improve our result and also help the variable insertion since our resulting syntax is as of now, rather broken due to misplaced closing parantheses.

The variable insertion could be made to insert variables even in non closed parentheses. This is quite often the case but was never achieved by our parser. See the first result example in section Re-

sults.

The parser could also start looking at some common combination of keywords. An example of this would be that places always should be combined with a word like from or to. We know that locations have a high probability of being correctly translated so if the keywords could be inserted together with a variable we should always create at least one correct relation.

9 Possible uses

This method of translating natural language into logical form could work very well with other easier database languages such as SQL. This could greatly improve the usability in webpages and software.

Another future use could be to translate pseudo code into high level code such as java or c#. This could make it possible even for inexperienced users to create their own specialized software.

Acknowledgements

We would like to thank Pierre Nugues for all his help and encouraging throughout the project. We would also like to take the opportunity to thank Luke Zettlemoyer for letting us use his manually translated ATIS corpus.

References

Luke Zettlemoyer and Michael Collins. 2009. *Learning Context-dependent Mappings from Sentences to Logical Form*. MIT CSAIL, Cambridge MA.

Parser code

```
($lambda, $parse, $plain) = @ARGV;
open IN1, $lambda || die "readfail lambda";
open IN2, $parse || die "readfail parse";
open IN3, $plain || die "readfail plain";
open OUT, ">>resultat.txt" || die "openfail res";


while ($line = <IN1>){
    $lamb = $line;
    $line2 = <IN2>;
    $par = $line2;
    @words = split(/ /, $lamb);
    @bindings = split(/\) /,$par);
    for ($i = 0; $i <= $#bindings-1; $i++) {
        $word = substr($bindings[$i], 0, index($bindings[$i], ' ', 0));
        $word =~ tr/ //;
        $temp = substr($bindings[$i], index($bindings[$i], '{', 0)+2,
                index($bindings[$i], '}', 0)-index($bindings[$i],
                '{', 0)-2)."\n";
        @indices =  split(/ /,$temp);
        for ($j = 0; $j <= $#indices -1; $j++) {
            $translations[$j] = $words[$indices[$j]-1];
        }
        if(!exists($super{$word})){
            $super{$word} = [ @translations ];
        }
        else{
            push @{ $super{$word} }, @translations;
        }
        undef @indices;
        undef $temp;
        undef @tempA;
        undef @translations;
    }
}
close IN1;
close IN2;

open IN1, $lambda || die "readfail lambda";

while($l = <IN3>){
    $q = <IN1>;
    $sentence = $l;
    $tak = $q;

    @words2 = split(/ /,$sentence);
    @matrix = ();
    $wordnumber = 0;
    for ($i = 0; $i <= $#words2; $i++) {
        @transes = @{$super{$words2[$i]}};
```

```
    %seen = ();
    @translist;
    @freqlist;
    foreach $item (@transes) {
        push(@uniq, $item) unless $seen{$item}++;
    }
    for $item(keys %seen) {
        push(@translist, $item);
        push(@freqlist, $seen{$item});
        $seen{$b} <=> $seen{$a};
    }
    for( $j=0; $j<=$#translist; $j++){
        $matrix[$j][$wordnumber] = $translist[$j]."[".$freqlist[$j]."]";
    }
    undef %seen;
    undef @translist;
    undef @freqlist;
    $wordnumber++;
}
$score =1;
for ($y=0;$y <=$#{$matrix[0]};$y++){
    for ($i=0;$i <=$#matrix; $i++){
        $this_probability = substr($matrix[$i][$y],
                index($matrix[$i][$y], '[', 0)+1,
                index($matrix[$i][$y], ']', 0)-index($matrix[$i][$y],
                '[', 1)-1);
        $totalfrq += $this_probability;
        if($probability < $this_probability){
            $most_prob = $matrix[$i][$y];
            $probability = $this_probability;
        }
    }
    $sent = $sent.substr($most_prob,0,index($most_prob, '[', 0))." ";
    if($totalfrq == 0){
        $slh1 = 1;
    }
    else{
        $slh1 = $probability/$totalfrq;
    }
    $score = $score * $slh1;

    push(@preproc,substr($most_prob,0,index($most_prob, '[', 0)));
    push(@count, $probability);
    push(@slh, $slh1);

    $most_prob = "";
    $probability = 0;
    $totalfrq=0;
}

print OUT "\nFACIT: ".$tak;
```

```
    print OUT "PRE PP: "."@preproc"."\n";
    @preproc = para_add(@preproc);
    $preproc = var_add(@preproc);
    print OUT "POST PP: ".$preproc."\n";
    print OUT "Score: ".substr($score,0,6);
    print OUT "---------------------------------\n";
    push (@scorelist, $score);
    print $loadingbar++;
}

foreach $value (@scorelist) {
    $sum += $value;
}
print OUT "Mean of all scores is: ".$sum/$#scorelist;

################################################################

sub para_add{
    my(@preproc) = @_;
    for($i=0;$i<=$#preproc;$i++){
        if(index(@preproc[$i],'(',0) >= 0){
            $open = $open + 1;
        }
        if(index(@preproc[$i],')',0) >= 0){
            $close = $close + 1;
        }
    }
    $nbr = $open-$close;
    for ($i=0;$i < $nbr;$i++){
        push(@preproc, ")");
    }
    return @preproc;
}


sub var_add{
    my(@preproc) = @_;
    $sentence = "@preproc";
    #$sentence =~ s/(\([^($ ]+?)( )([^($ ]+?\))/\1 $0 \3/;
    $sentence =~ s/(\([^($ ]+?)( )([^($ ]+?\))/$1 \$0 $3/;
    return $sentence;
}
```