Mid-Term Project: FST and GRM Tools for SLU

Emanuele Viglianisi (187270)
emanuele.viglianisi@studenti.unitn.it

Abstract

This report investigates several configurations in developing a Generative Spoken Language Understanding Module (SLU) given a dataset of conceptually-tagged utterances. The aim of the investigation is to compare results obtained with different methods and parameter congurations in order to find which one performs better when evaluated over a given test set. From our investigation, the best result in terms of Precision, Recall and F1 score is obtained by pre-processing the training set and using a composition between *word2lemma* FST and a language model with 9-gram and Kneser Ney smoothing algorithm.

1 Introduction

Currently, the focus of some of the major software companies is on developing virtual assistant like Apple's Siri and Microsoft's Cortana. They are software able to understand natural language as user input in order to answer questions or perform actions. A core module of these software is represented by a Spoken Language Understanding (SLU) Module to understand the user input.

The aim of this project is to develop a spoken language understanding module for the movie domain using NL-SPARQL Data Set. The module is trained using a statistical approach over a training set of semantically and conceptually annotated utterances. The goal is to predict the meaning of unseen testing utterances. To do this, different methods and configurations have been tested and the results of these test will be analysed in the report.

2 Dataset

The dataset used in all the experiments is Microsoft NL-SPARQL Dataset (Hakkani-Tur et al., 2014). NL-SPARQL is a data set of natural language (NL) utterances to a conversational system in the movies domain. The dataset is divided into:

- training set of **3338** utterances
- test set of **1084** utterances.

Table 1: NLSPARQL.train.feats.txt

word	**pos**	**lemma**
who	WP	who
plays	VVZ	play
luke	NN	luke
on	IN	on
star	NN	star
wars	NNS	war
new	JJ	new
hope	NN	hope

2.1 Description

Informations are stored in files, where utterances are separated by a empty line. The dataset is divided into two files for both training set and test set:

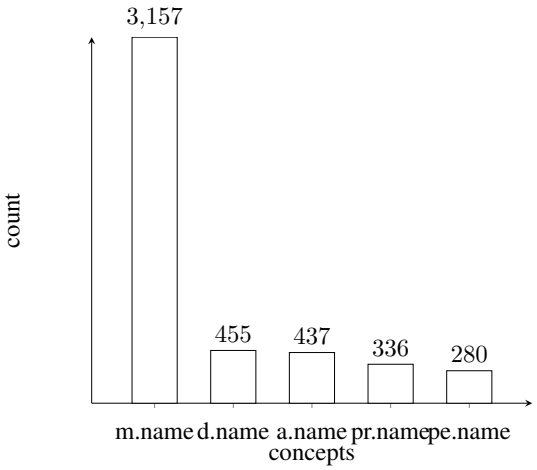
- *NLSPARQL.train.feats.txt* contains, for each utterance, a set of words, with the corresponding lemma, and part-of-speech tag. See Table 1 for an example.
- *NLSPARQL.train.data* contains, for each utterance, a set of words, with the associated concepts. Concepts are expressed using IOB notation. The IOB notation is used to label multi-word spans in token-per-line format. Therefore, we distinguish the terms at the Beginning, the Inside and the Outside of a multi-word span. An example is represented by "star wars new hope" in the following example.

Table 2: NLSPARQL.train.data

word	concept
...	
star	B-movie.name
wars	I-movie.name
new	I-movie.name
hope	I-movie.name

For simplicity, we will refer to *NLSPARQL.train.feats.txt* as *train1.txt* and *NLSPARQL.train.data* as *train2.txt*. During the project this two file have been merged in a single file called *merged.txt* created by appending last column of *train2.txt* to *train1.txt*.

2.1.1 Concept Distributions



The most frequent concept is *movie.name*, followed by *director.name, actor.name, etc*. Over 80% of the concepts represents *out-of-span tag* (marked with "O").

3 Approach

This section describes the tools we used and the strategy employed to create the spoken language understanding module for the project.

3.1 Tools

The tools we used are:

- *openfst* (Ope, a) is an open-source library to build weighted nite-state transducers (WFSTs). A *WFST* is a nite automaton for which each transition has an input label, an output label, and a weight. OpenFST provides also a set of operations that can be applied to FST. For example, it is possible two compose two WFSTs or perform the Union operations between multiple instances.
- *opengrm* (Ope, b) The OpenGrm NGram library is used for making and modifying n-gram language models encoded as weighted finite-state transducers (FSTs).

3.2 Strategy

3.2.1 Lexicon

The first thing that we have to do in order to use the aforementioned libraries is choosing a *lexicon*. A lexicon is a set of words that the FSTs should recognise. The lexicon also includes the symbols *eps* and *unk* that represents respectively epsilon and the unknown symbols. All the words that don't appear in the lexicon will be treated as an *unknown* symbol. To create the lexicon a python script *createLexicon.py* is used. It takes as the input training files and makes a list composed of: words, lemmas, concepts and pos-tags.

3.2.2 Develop of unigram-based Concept-Tagger

The second step is to create Weighted Final State Transducers. The aim is to associate a concept-tag to each word. To do this, we have multiple possibilities that consists in a single or multiple WFST. A WFST is a machine with a single state 0 that is both initial and final state for each transition. To each transition is associated a costs (weight). The way this machine is created and the costs are calculated are explained in depth in the section *Methods*.

3.2.3 Language Model

A statistical language model is a probability distribution over sequences of words. Given a sequence of length m, it assigns a probability $P(word_1, \ldots, word_m)$ to the whole sequence. Since we have to deal with data sparsity, we can apply the Markov assumption and consider that the probability of a word only depends on the previous k words instead of the entire sequence m. Suppose we want to compute a 2-grams model, the probability of a sequence takes the form:

$$P(X_1 = x_1, \ldots, X_m = x_m) = \prod_{1}^{m} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

In order to create a language model we have to extract all the concepts from the training set. This is done using the python script *extract_concepts.py*. Then, the concepts are given as

input to the library *OpenGrm* that creates a language model WFST.

In addiction, it is possible to specify the number of grams to be used, the smoothing algorithm, or other options like *backoff*. When backoff is enabled, lower order n-gram distributions are only used if the higher order n-gram was unobserved in the corpus.

3.2.4 Composition

In order to improve the model, we exploit the composition of the input string represented as fsa/fst with the unigram-based Concept-tagger and then with Language Model.

$$inputString_{fsa} \circ conceptTagger_{fst} \circ LM_{fst}$$

At the end, *fstshortestpath* is used to get the most probable concept sequence.

3.2.5 Evaluation

An important step is represented by the evaluation of the model. To evaluate the model we extract the test sentences from the test set using the script *extract_sentences.py*. These sentences are then processed by the model. The script *conneval.pl* takes as input the results of the model and a list of annotated concepts from the test set and prints out a report that contains the values of accuracy, recall and f1-score of the model.

4 Methods and tests

In this section are analysed different tests that have been performed trying to improve the scores. For each method there will be a brief description and a report of the results obtained.

4.1 Baseline

The baseline method consists of one single FST *word2concept*. It maps the word to the associate concept. At each transition is associated a non-zero cost. This method doesn't use additional features such as part-of-speech tags or lemmas. The automata is composed of a single state that is both initial and final.
Since the most probable sequence of concept to be associated is defined as:

$$c_1, c_2, \ldots, c_m = argmax_{c_1 \ldots c_m} \prod_{1}^{m} P(word_i|c_i)P(c_i)$$

We consider as cost for each transition the negative log of the probability $P(word|concept)$ that is calculated as:

$$\frac{count(word, concept)}{count(concept)}$$

Finally we compose the test string with *word2concept* and then, it is possible to use the program *fstshortestpath* to obtain the most probable sequence of concepts.

4.2 Method 1

The first method uses two FSTs. The first one is called *word2lemma*. It maps the word to the associated lemma. At each transition is associated a cost because it is possible that the same word is associated to different lemmas. For example, *'s* could represent the verb *to be* or *to have*. The cost is the negative log of the probability $P(word|lemma)$ and it is calculated as:

$$\frac{count(word, lemma)}{count(lemma)}$$

The second FST is *lemma2concept*. It associates a concept to each lemma. Part-of-speech tags are used as additional features to train the FST and are considered in the costs that is defined as the negative log of the probability $P(lemma, pos|concept)$ that is calculated as:

$$\frac{count(lemma, pos, concept)}{count(concept)}$$

The test string is composed with both *word2lemma*, *lemma2concept* and finally with the *language model*. After that, the most probable sequence of concepts is given using the program *fstshortestpath*.

4.3 Method 2

Method 1 uses two FSTs to construct an unigram-based concept tagger. However, the composition between them can generate some noise that can reduce the accuracy of the model. The idea is to use a single FST that maps directly from word to concept, as in the baseline method. However, lemmas and pos-tags are used as additional features. The cost associated with each transition is the negative log of the probability $P(word, lemma, pos|concept)$, computed as

Table 3: Method 2 - word2concept

init	final	word	concept	costs
0	0	ferrell	I-person.name	1.7687
0	0	july	I-movie.name	4.3791
0	0	all	O	6.5970
0	0	direct	O	6.5970
...				

$$\frac{count(word, lemma, pos, concept)}{count(concept)}$$

The final FST has the format represented in Table 3.

Unknown words are also taken into account. There is also a transition for each couple *(unknown, concept)*. The cost could be calculated using the negative log of the prior probability of that concept $P(concept_i)$ however best results are obtained using simply a cost of 0.

The test string is composed only with *word2concept.fst* and with the language model. The results show that this method represents a big improvement compared to the baseline and the method 1.

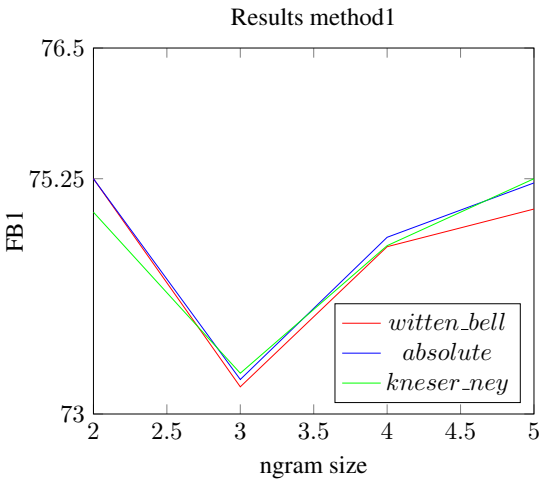
4.4 Method 3-4

In the preceding methods, one of the problems was the mislabelling of the out-of-span tags. Methods 3 and 4 are an extension of method 2 that use an additional phase of pre-processing. The script *pre-processing.py* appends the *word* to the corresponding O concept. In that way, the pre-processing phase makes it possible to train the model to better distinguish between out-of-span concept and other concepts. Methods 3 uses all the features to compute the costs, instead method 4 only computes the costs from the probability $P(word|concept)$. Adding the pre-processing steps in Method 3 and 4 significantly increased the performance with respect to the previous methods.

5 Results

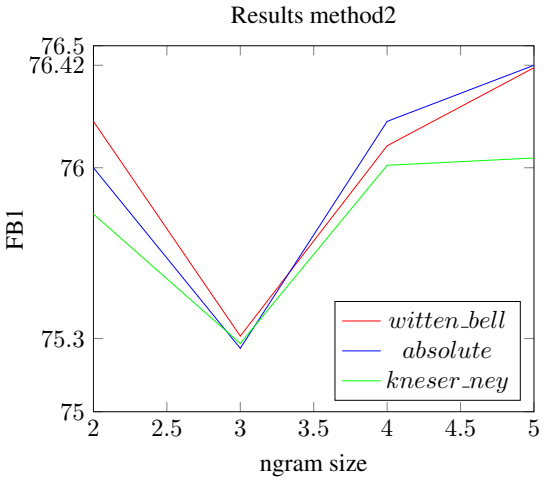
During the tests *frequency cutoff* and *–backoff option of ngrammake* were applied without any remarkable improvement. Because of this, their results are not reported in the following report. However, the report contains the results for all the possible combinations of n-grams (1 to 5-grams) and smoothing algorithms *witten_bell, absolute, katz, kneser_ney, presmoothed, unsmoothed*.

5.1 Method 1



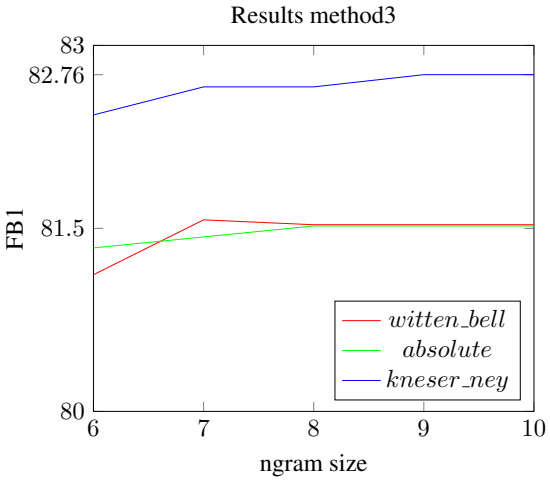
Best F1 score is **75.25** and it is obtained using 2-grams with *witten_bell*, *absolute* and 5-grams with the algorithm *kneser_ney*.

5.2 Method 2



Best F1 score is **76.42** and it is obtained using 5-grams with the algorithm *absolute*.

5.3 Method 4



Best F1 score is **82.76** and it is obtained using 5-grams with the algorithm *kneser_ney*.

6 Conclusion

The report describes different methods to predict concepts for each word of unseen testing utterances. The best score in terms of F1 and accuracy is obtained from the method 4 applying a pre-processing phase of the training data and using a large ngram size and *kneser_ney* smoothing algorithm. Method 4 doesn't use additional features such as part-of-speech tags and lemmas. The use of these additional features only represents a small improvement in the performances probably because they led to overfitting.

References

Openfst. http://www.openfst.org/. Last Update: 2017-03-14.

Opengrm. http://opengrm.org/. Last Update: 2017-01-24.

Dilek Hakkani-Tur, Asli Celikyilmaz, Larry Heck, Gokhan Tur, and Geoff Zweig. 2014. Probabilistic enrichment of knowledge graph entities for relation detection in conversational understanding. In *Proceedings of Interspeech*. ISCA - International Speech Communication Association, page 1. research.microsoft.com/apps/pubs/default.aspx?id=219835.