Bogazici University Computer Engineering Department
CMPE 561 Natural Language Processing
Homework #2 Report

HMM POS Tagging

Dilara Keküllüoğlu
2014700171

April 23, 2016

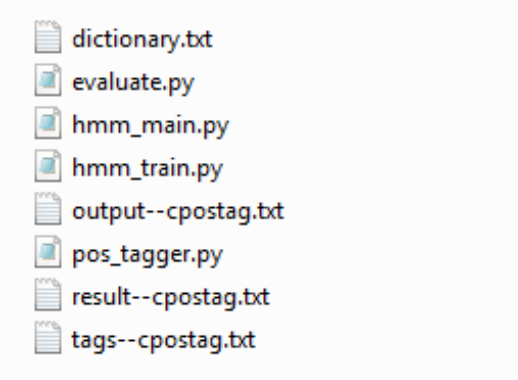

Figure 1: Files

1 Introduction

In this project we were asked to write an HMM Part-of-Speech Tagger using Viterbi algorithm. Every word in a sentence has some properties. They can be adjectives, nouns etc. We are going to use machine learning to assign words their pos tags after training our HMM model. Dataset was created with a joint effort of Metu and Sabanci Universities named *METU-Sabanci Turkish Dependency Treebank*[1] [2] [3]. The code is written in Python.

In Section 2, we will explain the files, their purpose and also the usage of the program. Structure of the program and how we implemented the steps will be explained in Section 3. Evaluation of the tagger is discussed on Section 4 and lastly we conclude our report on Section 5.

You can find the source code in the repository
https://github.com/dilara91/hmm_pos_tagger

2 The Mechanism and Usage

Files for this project is shown in Figure 1.

- *hmm_main.py:* This is the main program which calls necessary parts of the program. It takes five arguments: path to train file, tagset name,path to blind test file, output file and path to gold test file.
- *hmm_train.py:* This file take two parameters; path to train file and the tagset name to train the hmm tagger. It returns the transition,

emission and tags to the caller. That is why this program should be called from the script that assigns pos tags to the test data.

- *pos_tagger.py:* This file calls hmm_train to train the hmm then uses viterbi algorithm to compute the most likely pos tags. It writes the tags and words to the output file as shown in the output_sample.txt file we were given.

- *evaluate.py:* This file takes two arguments; output file and the path to the gold test dataset. It will compare the predictions with the actual tags and writes the results in a text file. The contents of the result file is explained later.

- *dictionary.txt:* This is the words we have in dictionary to keep track of known words so we can evaluate them accordingly.

- *tags–[tagset_name].txt:* This file has the tags used in the tagset. It is created automatically while training the HMM.

- *output–[tagset_name].txt:* This is the output file in the manner given in the sample output file. Every line has a 'word|tag' pair indicating which tag our HMM assigned to the word.

- *result–[tagset_name].txt:* This file has the evaluation results. It firstly gives the confusion matrix. Then writes the precision and recall for every tag. After that micro and macro averaged precision, recall and f-measure is given. Lastly we have accuracy values for all words,known words and unknown words.

To use the program user need to call the hmm_main.py with five arguments; path to train file, tagset name,path to blind test file, output file and path to gold test file.

```
python hmm_main.py
. metu_sabanci_cmpe_561 train turkish_metu_sabanci_train.conll -cpostag
. metu_sabanci_cmpe_561 test turkish_metu_sabanci_test_blind.conll
output-cpostag.txt
. metu_sabanci_cmpe_561 test turkish_metu_sabanci_test_gold.conll
```

After running this program you can find the results in the results file.

3 Program Structure

This program consists of three main steps; training , tagging and evaluation. Our HMM has transition, emission and tag counts. Transition keeps the occurrences where tag1 is followed by tag2 in the training dataset. For example if a 'Adjective' is followed by 'Noun' we increment their occurrence in transition. Emission keeps the occurrence where a word 'w' is assigned a tag 't' so we can calculate the probability of a word being tagged by a tag. Tag counts keeps the count of tags occurred in the training dataset. We use this HMM to tag our test file. To do that we use Viterbi algorithm which is a dynamic programming that uses the assumption that the probability of assigning tag 't' to word 'w' only depends only on the previous state and the P(w|t). We calculate these probabilities and find the maximum likely tag for all words. The tasks are explained in detail in the following sections.

3.1 Task 1

This task is the training phase. It takes two arguments; training file and tag set name. We take the training file and for every line with a legitimate word in it, we do three things. Firstly we record the tag transition with the previous tag. Secondly we record the emission which keeps the tag-word pairs. Lastly we increment the tag count for the current tag. The reason why we do not get every line as a legitimate word is because Turkish has a lot of inflectional groups of words and in the dataset these are represented in some additional lines. We only take the main word for these group of words since it is also represented same way in the output_sample.txt file. The dummy tags <START> and <END> are used for start and end respectively.

After we process the whole file, we create a dictionary with the training file so can keep track of the known words. These words are kept in dictionary.txt file. In addition to that we also create a tag file to keep track of the tags the training file use so we can create the confusion matrix accordingly. Then we send the results to the caller file.

3.2 Task 2

This task is the tagging phase. It takes three files. Training file, tag set name and testing file. The reason this script takes the first two arguments is because it runs the Task 1 code firstly to get the trained HMM. Then for

the every sentence of the test file we call the Viterbi function which returns the predicted tags for the words of the sentence. We write these words with their predicted tags to the output file in a manner that was given in the output_sample.txt file.

Viterbi algorithm gets the trained hmm and the sentence we want to tag. We create a matrix similar to the one that is shown in the class. For every word of the sentence we calculate the maximum probable tag. To do that we enumerate all the tags that word 'w' can take. For every 'w,t' pair we look at the previous words tag possibilities. We calculate the maximum probability as follows;

$P(w_c, t_c) = argmax(P(w_p, t_i) * P(t_i, t_c)) * P(w_c|t_c)$
w_c is the current word and t_c is the tag we currently calculating probability for. w_p is the previous word. For every possible tag we calculated $P(w_p, t_i)$ in the previous step. $P(t_i, t_c)$ is the transition probability from the t_i to t_c We take the product of these two and get the maximum of these products. Lastly we multiple the maximum with the probability of the current word assigned to the current tag, $P(w_c|t_c)$. These probabilities are computed by using Naive Bayes method like the previous project.

$P(t_i, t_j) = \frac{transition[t_i,t_j]}{count(t_i)}$

Transition probability is calculated as shown. The occurrences where such transition took place is divided by the total times first tag is observed in the training file. The possibility of 0 division is really small hence we do not do any smoothing.

$P(w_c, t_c) = \frac{emission[t_c,w_c]}{count(t_c)}$

The probability of a word to be tagged with a tag is calculated as shown. However there may be some words we have not seen in the training files. In that case we will have 0 s in the numerator.All these calculations converted to logs to avoid overflow since with each word the probability gets smaller. That means we cannot have 0 in the numerator. That is why we use Laplace smoothing as follows;

$P(w_c, t_c) = \frac{emission[t_c,w_c]+1}{count(t_c)+count(word)}$
Since we use smoothing for the unknown words,$P(w_c|t_i)$ is same for every tag t_i . So the assigned tag is only selected by the $argmax(P(w_p, t_i) * P(t_i, t_c))$

which is the biggest production of transition and previous tag assignment.

After we reach the end of the sentence we add the transition to <END> . We kept best edges for every possible w,t pair for every word. Best edge is the edge between the w_c, t_c and the w_p, t_i that makes our $P(w_c, t_c)$ maximum. We use these best edges to backtrack the best possible w,t pairs for every word and assign these tags to their respective words.

4 Evaluation

The tags are different for each tagset. –cpostag has following tags;

`Adv,Noun,Det,Zero,Postp,Pron,Ques,Verb,Interj,Dup,Punc,Conj,Num,Adj.`

–postag has following tags;

`Adv,NPastPart,Prop,Pron,Ques,QuesP,Dup,ReflexP,Adj,Real,APastPart,Verb,NFutPart,`
`Range,Postp,Punc,PersP,Zero,Num,Interj,NInf,Distrib,Conj,APresPart.`

We used validation files given to test our program. The results are as follows;

- **–cpostag :**
  accuracy : 0.480332409972
  accuracy known : 0.364836100468
  accuracy unknown : 0.695479777954

- **–postag :**
  accuracy : 0.324376731302
  accuracy known : 0.306087696892
  accuracy unknown : 0.358445678033

It seems like this tagger works better on unknown words which only use the transitions while making decisions. That shows that only taking transition habits to the consideration we can have higher results. This may be the result of small training dataset. Also we have better results with the –cpostag tagset since they have smaller options to choose from.

From the confusion matrices we can see that our tagger have strong bias for some tags. This may come from the occurrences of those words in the

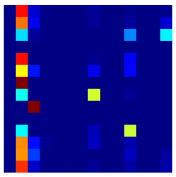

Figure 2: –cpostag confusion matrix

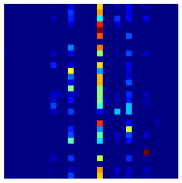

Figure 3: –postag confusion matrix

training dataset. If those tags have more instances then they are likely to be chosen.

Further results can be seen from the github page.

5 Challenges

The biggest challenge was the Turkish specific characters and getting them correctly. There was some misunderstanding on my part about the string encodings that Python uses and it made me spend much time on it. To open the files correctly we used standard library codecs so we can encode the words to 'UTF-8' .

Bibliography

[1] Nart B Atalay, Kemal Oflazer, Bilge Say, et al., *The annotation process in the turkish treebank*, Proc. of the 4th Intern. Workshop on Linguistically Interpreteted Corpora (LINC), Citeseer, 2003.

[2] Gülşen Eryiğit, Tugay Ilbay, and Ozan Arkan Can, *Multiword expressions in statistical dependency parsing*, Proceedings of the Second Workshop on Statistical Parsing of Morphologically Rich Languages, Association for Computational Linguistics, 2011, pp. 45–55.

[3] Kemal Oflazer, Bilge Say, Dilek Zeynep Hakkani-Tür, and Gökhan Tür, *Building a turkish treebank*, Treebanks, Springer, 2003, pp. 261–277.