CMPE561 Application Project #1

Mine Melodi Çalışkan
2015705009
minemelodicaliskan@gmail.com

Emrah Budur
2016800036
emrah.budur@yahoo.com

10 May 2017

Introduction

Part of Speech (POS) tagging is a well-known to be a sequence labeling task that labels each word in a sentence with its corresponding tags such as adjective, noun, verb, etc. Hidden Markov Models (HMM) is a generative model that has proven success on POS tagging problem. In this study, we built a POS tagging application that is based on the Viterbi algorithm, which is an example of the HMM model, and evaluated its performance on METU-Sabancı Turkish Dependency Treebank dataset [1, 2, 3]. As a result, we reported 40% accuracy on the test set that is obtaining by randomly selecting 10% of the corpus.

Program Interface

In the Main function the user should enter the corpus file name if different data set other than METU-Sabancı Turkish Dependency Treebank is desired.

This function returns:

- Success statistics for test set
- Word based measurements
- Sentence based measurements
- Tag accuracies

- Confusion matrix

based on the provided corpora.

It is also possible to enter any sentence (in our case Turkish sentence) to predict it's Part-of-Speech tags.

Program Execution

Method

We trained our model using Maximum Likelihood Estimates and predicted Part-of-Speechs using Viterbi algorithm.

Training

The MLE of some parameter of a model M from a training set T is the estimate that maximizes the likelihood of the training set T given the model M.

The estimation is obtained by:

$$\begin{aligned} P_{ML}(t_i|t_{i-1}) &= \frac{c(t_{i-1}t_i)}{c(t_{i-1})} \\ P_{ML}(w_i|t_i) &= \frac{c(t_i w_i)}{c(t_i)} \end{aligned} \tag{1}$$

which is the estimation obtained by getting counts from a corpus, and then normalizing these counts so that they lie between 0 and 1.

Prediction

Viterbi is a decoding algorithm used for HMMs, in this case parf-of-speech tagging.

It takes as input a single HMM and a set of observed words and returns the most probable tag sequence together with its probability.

The algorithm sets up a probability matrix where columns correspond to observations (words) t and rows corresponds to states (tags).

Procedure can be itemized as follows:

- Begin by setting Viterbi values to each cell in the first columns as the product of transition probabilities and the observation probabilities.
- Move column by column, for every state in column i compute the probability of moving into each state in column i+1.
- For each state q_j at time t recursively compute the Viterbi value by taking the maximum over all the path that lead to the current cells with:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t) \tag{2}$$

$v_{t-1}(i)$: best score from Start to tag i until time t-1

a_{ij} : the transition probability from previous tag q_i to current tag q_j

$b_j(o_t)$: the state observation likelihood of the observation word o_t given the current tag j

Input and Output

Input

We used METU-Sabancı Turkish Dependency Treebank dataset as the input corpus [1, 2, 3]. This is a sentence-based corpus where each word in the sentences is labeled with their morpohological units including but not limited to POS tags. Each sentence is separated from each other by blank lines as illustrated in Figure 1.

```
1   Peşreve peşrev  Noun    Noun    A3sg|Pnon|Dat   2   OBJECT  _   _
2   başlamalı   başla   Verb    Verb    Pos|Neces|A3sg  3   SENTENCE    _   _
3   .   .   Punc    Punc    _   0   ROOT    _   _

1   Ama ama Conj    Conj    _   5   S.MODIFIER  _   _
2   annemin anne    Noun    Noun    A3sg|P1sg|Gen   3   POSSESSOR   _   _
3   şartları    şart    Noun    Noun    A3pl|P3sg|Nom   4   SUBJECT _   _
4   vardı   var Verb    Verb    Pos|Past|A3sg   5   SENTENCE    _   _
5   .   .   Punc    Punc    _   0   ROOT    _   _

1   Sanal   sanal   Adj Adj _   2   MODIFIER    _   _
2   _   parçacık    Noun    Noun    A3pl|Pnon|Nom   3   DERIV   _   _
3   parçacıklarsa   _   Verb    Zero    Cond|A3sg   7   SUBJECT _   _
4   bunların    bu  Pron    DemonsP A3pl|Pnon|Gen   5   POSSESSOR   _   _
5   hiçbirini   hiçbiri Pron    Pron    A3sg|P3sg|Acc   6   OBJECT  _   _
6   _   yap Verb    Verb    _   7   DERIV   _   _
7   yapamazlar  _   Verb    Verb    Able|Neg|Aor|A3pl   8   SENTENCE    _   _
8   .   .   Punc    Punc    _   0   ROOT    _   _
```

Figure 1: An example of METU-Sabancı Turkish Dependency Treebank dataset.

Data preprocessing

In order to parse the input file, we converted each line of the file, which are delimited by a tab character, into disjoint fields. Below are the steps that we taken during data preprocessing phase.

- We used the second and fourth fields as the word and the label respectively.
- We skipped those lines having the charaacter '_' in its second column as described in the problem definition.
- We relabeled those lines, which were wrongly labeled as 'satın', as Noun.
- We processed the data sentence-wise, that is, we seperated each sentences by an end of sentence term (</s>) after parsing the input.
- The sentences in this corpus were splitted into two parts as **training data** and **test data** such that 90% of the sentences are in the training set and 10% are in the test set. It should be noted that the data were not splitted word-wise but split sentence-wise to preserve the dependencies between words which is an important factor for statistical POS tagging.

Figure 2 shows the resulting format of the dataset which is fed into our application as the actual input.

```
Peşreve→Noun
başlamalı→Verb
.→Punc
</s>
Ama→Conj
annemin→Noun
şartları→Noun
vardı→Verb
.→Punc
</s>
Sanal→Adj
parçacıklarsa→Verb
bunların→Pron
hiçbirini→Pron
yapamazlar→Verb
.→Punc
</s>
```

Figure 2: The parsed form of the input dataset.

Output

The output of our application can be reviewed in two categories such as training phase outputs and prediction phase output.

In the training phase, we measured the performance of the application on training/test data and output a list of evaluation metrics as described below.

- The success rate of our tagger in terms of the number of words correctly tagged in both part of the dataset.
- A sentence-based success rate, which is the ratio of the number of correctly tagged sentences to the total number of sentences.
- The accuracy of each tag along with a confusion matrix

In the prediction phase, our application outputs the predicted sequence of tags that corresponds to a given sentence.

Program Structure

The application is mainly composed of two phases as mentioned previously. In the training phase, we calculated two set of probabilities, namely transition probability and word likelihood probabilities which are explained below.

By transition, we mean the tag sequence of two words. For example, given a sequence of two words having tag1 and tag2 respectively, we say there is a transition from tag1 to tag2. So, in order to calculate transition probabilities, we first iterated over the training sentences and counted the raw frequencies of transitions between each existing pair of tags. We used *a dictionary of dictionary* data structure to store these frequencies and converted it into the data type as known as 'Pandas dataframe'. The resulting data type is a 14×15 matrix of tags as illustrated in Figure 3. In this representation, the columns refers to conditioned tags (previous tag). The additional tag in columns is '<s>' since it appears only as the conditioned tag.

Then, we converted our frequency matrix into probability matrix by implementing maximum likelihood estimation formula. The resulting form of the matrix is

illustrated in Figure 4. We asserted that the sum of the probabilities for each conditioned tag is 1.0 which can also be checked in the column of tag 'Zero' easily in Figure 4.

	<s>	Adj	Adv	Conj	Det	Dup	Interj	Noun	Num	Postp	Pron	Punc	Ques	Verb	Zero
Adj	519.0	471.0	422.0	268.0	130.0	1.0	1.0	2113	63.0	144.0	143.0	543.0	7.0	104.0	0.0
Adv	513.0	76.0	235.0	114.0	11.0	0.0	6.0	1135	11.0	54.0	155.0	305.0	5.0	145.0	0.0
Conj	218.0	133.0	140.0	79.0	7.0	0.0	2.0	876	9.0	28.0	125.0	142.0	3.0	273.0	0.0
Det	336.0	488.0	114.0	103.0	4.0	0.0	0.0	357	0.0	42.0	60.0	227.0	2.0	75.0	0.0
Dup	2.0	1.0	1.0	0.0	0.0	8.0	0.0	6	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Interj	38.0	2.0	0.0	4.0	0.0	0.0	2.0	11	0.0	1.0	0.0	24.0	0.0	9.0	0.0
Noun	2342.0	2906.0	745.0	836.0	1558.0	4.0	27.0	7467	424.0	316.0	586.0	1826.0	20.0	485.0	0.0
Num	98.0	64.0	35.0	30.0	8.0	0.0	0.0	203	68.0	20.0	17.0	76.0	0.0	5.0	0.0
Postp	6.0	62.0	22.0	5.0	12.0	0.0	0.0	679	1.0	5.0	77.0	108.0	0.0	52.0	0.0
Pron	542.0	41.0	111.0	146.0	6.0	0.0	2.0	377	0.0	44.0	109.0	319.0	13.0	169.0	0.0
Punc	117.0	352.0	396.0	238.0	7.0	2.0	39.0	2732	30.0	129.0	242.0	224.0	109.0	4756.0	2.0
Ques	0.0	17.0	10.0	2.0	0.0	0.0	0.0	49	1.0	1.0	8.0	0.0	0.0	101.0	0.0
Verb	339.0	316.0	534.0	210.0	65.0	4.0	12.0	3534	17.0	245.0	357.0	511.0	30.0	202.0	0.0
Zero	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 3: Tag transition frequencies. Columns refer to the conditioned tags.

	<s>	Adj	Adv	Conj	Det	Dup	Interj	Noun	Num	Postp	Pron	Punc	Ques	Verb	Zero
Adj	0.102367	0.095557	0.152622	0.131695	0.071903	0.052632	0.010989	0.108132	0.100962	0.139942	0.076104	0.126103	0.037037	0.016311	0.0
Adv	0.101183	0.015419	0.084991	0.056020	0.006084	0.000000	0.065934	0.058083	0.017628	0.052478	0.082491	0.070831	0.026455	0.022742	0.0
Conj	0.042998	0.026983	0.050633	0.038821	0.003872	0.000000	0.021978	0.044829	0.014423	0.027211	0.066525	0.032977	0.015873	0.042817	0.0
Det	0.066272	0.099006	0.041230	0.050614	0.002212	0.000000	0.000000	0.018269	0.000000	0.040816	0.031932	0.052717	0.010582	0.011763	0.0
Dup	0.000394	0.000203	0.000362	0.000000	0.000000	0.421053	0.000000	0.000307	0.000000	0.000000	0.000000	0.000232	0.000000	0.000000	0.0
Interj	0.007495	0.000406	0.000000	0.001966	0.000000	0.000000	0.021978	0.000563	0.000000	0.000972	0.000000	0.005574	0.000000	0.001412	0.0
Noun	0.461933	0.589572	0.269439	0.410811	0.861726	0.210526	0.296703	0.382120	0.679487	0.307094	0.311868	0.424059	0.105820	0.076066	0.0
Num	0.019329	0.012984	0.012658	0.014742	0.004425	0.000000	0.000000	0.010388	0.108974	0.019436	0.009047	0.017650	0.000000	0.000784	0.0
Postp	0.001183	0.012579	0.007957	0.002457	0.006637	0.000000	0.000000	0.034747	0.001603	0.004859	0.040979	0.025081	0.000000	0.008156	0.0
Pron	0.106903	0.008318	0.040145	0.071744	0.003319	0.000000	0.021978	0.019293	0.000000	0.042760	0.058010	0.074083	0.068783	0.026506	0.0
Punc	0.023077	0.071414	0.143219	0.116953	0.003872	0.105263	0.428571	0.139809	0.048077	0.125364	0.128792	0.052020	0.576720	0.745922	1.0
Ques	0.000000	0.003449	0.003617	0.000983	0.000000	0.000000	0.000000	0.002508	0.001603	0.000972	0.004258	0.000000	0.000000	0.015841	0.0
Verb	0.066864	0.064110	0.193128	0.103194	0.035951	0.210526	0.131868	0.180851	0.027244	0.238095	0.189995	0.118672	0.158730	0.031681	0.0
Zero	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000102	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0

Figure 4: Tag transition probabilities. Columns refer to the conditioned tags, i.e. P(row_tag |col_tag) (No smoothing)

By word likelihood, we mean the probability of *a given tag* being labeled to *a given word*. So, in order to calculate word likelihood probabilities, we first iterated over the training sentences, and counted the raw frequencies of each existing pairs of words and tags. As a result, we obtained a 16321×14 matrix of frequencies as illustrated in Figure 5. Then, we converted the raw frequencies into probabilities by implementing maximum likelihood estimation formula. The resulting form of the matrix is shown in Figure 6.

When we wanted to apply add-k smoothing on the probabilities, we only added k to all of the raw frequency values and calculated the probabilities based on the ordinary maximum likelihood estimation formula.

After calculating the transition probabilities and word likelihood probabilities, we implemented viterbi algorithm to predict the POS tag sequence of a given sentence

	Adj	Adv	Conj	Det	Dup	Interj	Noun	Num	Postp	Pron	Punc	Ques	Verb	Zero
01-Feb	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
10'unun	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
118'lere	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1854'te	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
19.8'lik	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1912'de	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1950'de	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1956'da	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1970'li	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1971'de	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20'li	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2002'de	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2005'e	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
28'lik	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3'lük	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30'lu	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30-40	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
375'e	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4'ten	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40'ı	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
şostakovi	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
şov	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
şu	0.0	0.0	0.0	30.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0
şubat	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
şube	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
şubesi	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
şunlar	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0

Figure 5: Word likelihood frequencies. Columns refer to the conditioned tag.

	Adj	Adv	Conj	Det	Dup	Interj	Noun	Num	Postp	Pron	Punc	Ques	Verb	Zero
01-Feb	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.001603	0.0	0.000000	0.000000	0.0	0.000000	0.0
10'unun	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
118'lere	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
1854'te	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
19.8'lik	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
1912'de	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
1950'de	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
1956'da	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
1970'li	0.000203	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
1971'de	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
20'li	0.000203	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
2002'de	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000102	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
2005'e	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
28'lik	0.000203	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
3'lük	0.000203	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
30'lu	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
30-40	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.001603	0.0	0.000000	0.000000	0.0	0.000000	0.0
375'e	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
4'ten	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
40'ı	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
şostakovi	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.000157	0.0
şov	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
şu	0.000000	0.000000	0.0	0.016593	0.0	0.0	0.000000	0.000000	0.0	0.002661	0.000000	0.0	0.000000	0.0
şubat	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000102	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
şube	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000154	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
şubesi	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000051	0.000000	0.0	0.000000	0.000000	0.0	0.000000	0.0
şunlar	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.000532	0.000000	0.0	0.000000	0.0

Figure 6: Word likelihood probabilities. Columns refer to the conditioned tag, i.e. $P(w_i|t_i)$ (No smoothing)

for the prediction phase. We iterated over each sentence in the test set and predicted their sequence of tags.

After predicting the sequence of tags of each sentence, we compared the predicted values with the actual values given in the test set. Then, we calculated an accuracy score both for all words and all sentences in the test set.

k	Word-based accuracy	Sentence-based accuracy
0	0.71	0.36
1	0.75	0.29
0.75	0.76	0.32
0.5	0.78	0.33
0.25	0.79	0.36
0.125	0.80	0.38
0.0625	0.80	0.39
0.03125	0.80	0.39
0.015625	**0.80**	**0.40**

Table 1: Accuracy results

Packages

- Numpy
- Pandas
- codecs

Results

We calculated an accuracy score on test dataset in terms of both the number of words which were correctly tagged and the number of sentence which were correctly tagged as a whole. In addition we applied add-k smoothing and we conducted a series of experiment on the effect of k values on the accuracy scores. The results were reported in Table 1. So, by applying add-k smoothing where $k = 0.015625$, we improved the test score of word-based accuracy from 71% to 80% and also we improved the test score of sentence-based accuracy from 29% to 40%.

In addition, we calculated an overall accuracy for each tag where $add - k = 0.015625$. The results were reported in Table 2.

Finally, we also calculated a confusion matrix of tags for our best score where $add - k = 0.015625$. The results were reported in Figure 7. Note that '</s>' is an augmented tag which refers to end of sentence which can be safely ignored.

Confusion matrix rows are conditioned, P(predicted_tag|correct_tag))

	</s>	Adj	Adv	Conj	Det	Interj	Noun	Num	Postp	Pron	Punc	Ques	Verb
</s>	565	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Adj	0.0	349	19	0.0	0.0	0.0	112	0.0	3	0.0	16	0.0	23
Adv	0.0	16	251	2	2	1	35	0.0	9	4	3	0.0	13
Conj	0.0	0.0	0.0	202	0.0	0.0	2	0.0	0.0	1	0.0	0.0	6
Det	0.0	0.0	0.0	0.0	222	0.0	0.0	0.0	0.0	4	0.0	0.0	0.0
Dup	0.0	1	0.0	0.0	0.0	0.0	1	0.0	0.0	0.0	1	0.0	0.0
Interj	0.0	0.0	2	3	0.0	12	1	0.0	0.0	0.0	0.0	0.0	0.0
Noun	0.0	56	1	0.0	1	0.0	1962	0.0	0.0	4	52	0.0	116
Num	0.0	0.0	0.0	0.0	15	0.0	7	59	0.0	0.0	0.0	0.0	0.0
Postp	0.0	4	6	0.0	0.0	0.0	4	0.0	128	0.0	0.0	0.0	0.0
Pron	0.0	0.0	0.0	0.0	5	0.0	3	0.0	0.0	205	2	0.0	2
Punc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1050	0.0	0.0
Ques	0.0	0.0	0.0	0.0	0.0	0.0	1	0.0	0.0	0.0	0.0	37	1
Verb	0.0	12	1	1	0.0	0.0	84	0.0	3	1	11	0.0	611

Figure 7: Confusion matrix of tags.

Algorithm

Algorithm 1 *Viterbi Algorithm*

function VITERBI(observations of len T, state-graph of len N)
  create a path probability matrix $viterbi[N+2, T]$
  for each state s from 1 to N **do** ▷ Initialization step
    $viterbi[s, 1] \leftarrow a_{0,s} * b_s(o_1)$
    $backpointer[s, 1] \leftarrow 0$
  end for
  for each time step t from 2 to T **do**
    for each state s from 1 to N **do**
      $viterbi[s, t] \leftarrow \max_{s'=1}^{N}\{viterbi[s', t-1] * a_{s',s} * b_s(o_t)\}$
      $backpointers[s, t] \leftarrow argmax_{s'=1}^{N}\{viterbi[s', t-1] * a_{s',s}\}$
    end for
  end for
  $viterbi[q_F, T] \leftarrow \max_{s=1}^{N}\{viterbi[s, T] * a_{s,q_F}\}$ ▷ termination step
  $backpointer[q_F, T] \leftarrow argmax_{s=1}^{N}\{viterbi[s, T] * a_{s,q_F}\}$ ▷ termination step
  for each time step t from T to 1 **do** ▷ backtrace the path
    s $\leftarrow backpointer[s, t]$
    $statesArray[t] \leftarrow s$
  end for
  return reverse of the statesArray
end function

Tag	Accuracy
Adv	0.747
Noun	0.895
Det	0.982
Postp	0.901
Pron	0.945
Ques	0.949
Verb	0.844
Interj	0.667
Punc	1.000
Conj	0.957
Num	0.728
Adj	0.668

Table 2: The overall accuracies of each tag

Examples

Input Sentence: "Aradığımı buldum sandım."

Output Tags: 'Noun', 'Verb', 'Verb'

Improvements and Extensions

Improvement 1: Smoothing

We tried add-k smoothing to overcome the sparse data problem in probability matrices. However, it would be possible to apply more advance methods for this problem. This section includes alternative sophisticated algorithms to compute smoothed probabilities.

Good-Turing Discounting : This approach treats unseen events as the events that are seen only once to estimate their counts.

Counts are adjusted as:

$c^* = (c+1)\frac{N_{c+1}}{N_c}$

where N_x is the frequency of frequency x.

For c=0, which corresponds to unseen species:

$c^* = \frac{N_1}{N}$

where $N = N_0$ is the number of unseen species.

Interpolation: This approach uses linear combination of estimations in N-gram models with increasing order and each of these estimations are weighted by a constant factor.

For example Interpolated Trigram Model:

$P(w_n|w_{n-2}, w_{n-1}) = \lambda_1 P(w_n|w_{n-2}, w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$

where $\sum_i \lambda_i = 1$

Back-off (Katz backoff with GT smoothing): If data for higher-order model is unavailable, i.e count is zero, recursively back-off to weaker models until data is available.

Improvement 2: Handling Unknown Words

Hapax Legomana: Idea of Hapax Legomana,as in the Good Turing Smoothing, is to treat the probability distribution of tags over unknown words similar to the distribution of tags over words that occur only once.

Then estimate $P(w_i|t_i)$ for an unknown word w_i as:

$$avg\big[P(w_j|t_i) \textit{for all singleton } w_j \textit{ in the training set}\big] \tag{3}$$

Use of morphological and orthographical information: This approach groups words in the training data into categories using morphological and orthographical information (e.g ends with "able") and estimates probabilities of categories which are then used for the calculations of the likelihoods.

Difficulties Encountered

Problem 1: Bad labels

In METU-Sabancı Turkish Dependency Treebank dataset we encountered Multi-word expressions tag (e.g "satın") which is not considered in our task and effected the training phase.

Applied solution:

We accepted "satın" word as "Noun".

Problem 2: Sparse data

Using MLE in the training phase, zero probabilities of unseen word-tag and tag-tag combinations gave rise to sparse data problem.

Applied solution:

To avoid such situations we used Laplace Smoothing.

Laplace Smoothing: In Laplace Smoothing we add 1 to all frequency counts as if every tag-tag and word-tag are seen once more and then re-normalize the probability space as follows:

$$\begin{aligned} P_{Laplace}(t_i|t_{i-1}) &= \frac{c(t_{i-1}t_i)+1}{c(t_{i-1})+V} \text{ where V is the tag size.} \\ P_{Laplace}(w_i|t_i) &= \frac{c(t_i,w_i)+1}{c(t_i)+V} \text{ where V is the tag size.} \end{aligned} \tag{4}$$

.

Problem 3: Unknown words

Viterbi algorithm requires a dictionary with parts-of-speech probabilities of every word and since it's not possible to acquire every possible word in a dictionary, there is a need to develop a method for guessing the tags of unknown words.

Applied solution:

We adapted our tagger by implementing following approach to unknown words:

Whenever $v_t(j) = 0$ for all j, because of an unknown word at position t, assume that $b_j(w) = 1$ for all j, i.e treat each unknown word as ambiguous among all possible tags, with equal probability.

This sets $v_t(j) = \max_i v_{t-1}(i)a_{ij}$ and allows to interpolate the missing POS tag based on the transition probabilities alone.

Conclusion

In this project we implemented a HMM-based part-of-speech tagger for Turkish using the METU-Sabancı Turkish Dependency Treebank dataset.We introduced and applied solutions to the difficulties that we encountered "Bad labels", "Sparse data" and "Unknown words". We improved the test score of word-based accuracy from 71% to 80% and also we improved the test score of sentence-based accuracy from 29% to 40%.

A Source Code

A.1 Parsing

A.1.1 Extract the sentences in the corpus.

```
1  import codecs
2
3  def parse_input_file(input_filename):
4
5      output_filename = input_filename + '.parsed.txt'
6
7      input_file = codecs.open(input_filename, 'r', encoding='utf-8')
8      output_file = codecs.open(output_filename, 'w', encoding='utf-8')
9
10     for line in input_file:
11         fields = line.strip().split('\t')
12         if len(fields) <=1:
13             output_file.write('</s>\n')
14             continue
15         if fields[1] == '_':
16             continue
17
18         word = fields[1]
19         if len(word)>1 and word <> u'...' and word <> u'..':
20             word = word.strip(',.\'?-')
```

```
21
22          word = word.lower()
23          tag = fields[3]
24          if tag.upper() == u'SATIN':
25              tag='Noun'
26          output_line = '\t'.join([word, tag]) +'\n'
27
28          output_file.write(output_line)
29
30      input_file.close()
31      output_file.close()
32      print 'parsed input file into', output_filename
33      return output_filename
```

A.1.2 Divide the corpus into two parts.

```
1  def split_corpus(input_filename):
2      test_filename = input_filename + '.test_set.txt'
3      train_filename = input_filename + '.train_set.txt'
4
5      input_file = codecs.open(input_filename, 'r', encoding='utf-8')
6      test_file = codecs.open(test_filename, 'w', encoding='utf-8')
7      train_file = codecs.open(train_filename, 'w', encoding='utf-8')
8
9      eos = '</s>'
10     current_sentence = []
11     np.random.seed(1357)
12     for line in input_file:
13         current_sentence.append(line)
14
15         if line.strip() == eos:
16             if np.random.rand()<=0.9:
17                 for tmp_line in current_sentence:
18                     train_file.write(tmp_line)
19             else:
```

```
20                for tmp_line in current_sentence:
21                    test_file.write(tmp_line)
22            current_sentence = []
23
24
25    input_file.close()
26    test_file.close()
27    train_file.close()
28    print 'test␣set␣file␣into', test_filename
29    print 'train␣set␣file␣into', train_filename
30    return train_filename, test_filename
```

A.2 Smoothing Factor

```
1  def get_smoothing_factor():
2      return 0.015625
```

A.3 Maximum Likelihood Estimates

```
1  import pandas as pd
2  import numpy as np
3
4  def calculate_maximum_likelihood_estimation(freqs):
5      freqs += get_smoothing_factor()
6      probs = freqs / freqs.sum(axis=0)
7      assert probs.sum(axis=0).all() == 1
8      return probs
9
10 def calculate_viterbi_probabilities(input_filename):
11     input_file = codecs.open(input_filename, 'r', encoding='utf-8')
12     bos = '<s>' # beginning of sentence
13     eos = '</s>' # end of sentence
14
15     tag_transition_freqs = {}
16     word_likelihood_freqs = {}
17
```

```
18    prev_tag = bos
19    for line in input_file:
20
21        fields = line.strip().split('\t')
22        if fields[0] == eos:
23            prev_tag = bos
24            continue
25
26        curr_word = fields[0]
27        curr_tag = fields[1]
28
29        if prev_tag not in tag_transition_freqs:
30            tag_transition_freqs[prev_tag] = {}
31
32        if curr_tag not in tag_transition_freqs[prev_tag]:
33            tag_transition_freqs[prev_tag][curr_tag] = 0
34
35        if curr_tag not in word_likelihood_freqs:
36            word_likelihood_freqs[curr_tag] = {}
37
38        if curr_word not in word_likelihood_freqs[curr_tag]:
39            word_likelihood_freqs[curr_tag][curr_word] = 0
40
41        tag_transition_freqs[prev_tag][curr_tag] = tag_transition_freqs[
              prev_tag][curr_tag] + 1
42        word_likelihood_freqs[curr_tag][curr_word] = word_likelihood_freqs[
              curr_tag][curr_word] + 1
43
44        prev_tag = curr_tag
45    input_file.close()
46    tag_transition_freqs_dataframe= pd.DataFrame.from_dict(
          tag_transition_freqs)
47    tag_transition_freqs_filled=tag_transition_freqs_dataframe.copy()
48    tag_transition_freqs_filled=tag_transition_freqs_filled.fillna(0.0)
```

```
49
50      word_likelihood_freqs_dataframe= pd.DataFrame.from_dict(
            word_likelihood_freqs)
51      word_likelihood_freqs_filled=word_likelihood_freqs_dataframe.copy()
52      word_likelihood_freqs_filled=word_likelihood_freqs_filled.fillna(0.0)
53
54      tag_transition_probs = calculate_maximum_likelihood_estimation(
            tag_transition_freqs_filled)
55      word_likelihood_probs = calculate_maximum_likelihood_estimation(
            word_likelihood_freqs_filled)
56
57      print('tag_transition_probs.shape', tag_transition_probs.shape)
58      print('word_likelihood_probs.shape', word_likelihood_probs.shape)
59
60      return tag_transition_freqs_filled, word_likelihood_freqs_filled,
            tag_transition_probs, word_likelihood_probs
```

A.4 Viterbi

```
1  def viterbi(word_likelihood_probs,tag_transition_probs,sentence=None):
2      if not sentence:
3          string_input=raw_input("Enter␣a␣sentence:")
4          string_input = string_input.decode('utf-8')
5          sentence = string_input.split()
6          sentence = [word for word in sentence]
7
8      tags_for_test=list(tag_transition_probs.columns.values)
9      tags_for_test.remove('<s>')
10
11     conditioned_tag_size = len(word_likelihood_probs.keys())
12     #b_default = float(0.0) #no smoothing
13     b_default = float(get_smoothing_factor()/conditioned_tag_size) #
           smoothing applied
14
15     viterbi={}
```

```
16    pointers={}
17    #initialize
18    for curr_tag in tags_for_test:
19
20        curr_word = sentence[0].lower()
21        a_transition = float(tag_transition_probs['<s>'][curr_tag])
22        b_current = b_default
23
24        if (curr_tag in word_likelihood_probs) and (curr_word in
              word_likelihood_probs[curr_tag]):
25            b_current = float(word_likelihood_probs[curr_tag][curr_word])
26
27        viterbi.setdefault(0,{})[curr_tag]=a_transition*b_current
28        pointers.setdefault(0,{})[curr_tag]='<s>'
29
30    T = len(sentence)
31
32    for j in range(1,T):
33        for curr_tag in tags_for_test:
34            viterbi.setdefault(j,{})[curr_tag]=-1.0 # to make zero
                  probabilities greater
35
36    viterbi_pd=pd.DataFrame(viterbi)
37
38    for i in range(1, T):
39        curr_word = sentence[i]
40        for curr_tag in tags_for_test:
41            for prev_tag in tags_for_test:
42                v_prev = viterbi_pd[i-1][prev_tag]
43                a_transition = float(tag_transition_probs[prev_tag][curr_tag
                      ])
44                b_current = b_default
45
46                if (curr_tag in word_likelihood_probs) and (curr_word in
```

```
                    word_likelihood_probs[curr_tag]):
47                  b_current = float(word_likelihood_probs[curr_tag][
                        curr_word])
48                  cur_viterbi = v_prev * a_transition * b_current
49              else:
50                  cur_viterbi=v_prev*a_transition #treat each unknown words
                        with equal probability
51
52              if cur_viterbi > viterbi_pd[i][curr_tag]:
53                  viterbi_pd[i][curr_tag] = cur_viterbi #takes larger
                        values until it's max for current curr_tag
54
55                  pointers.setdefault(i,{})[curr_tag]=prev_tag #assigns
                        pointer to prev_tag which gives max value
56  pointers_pd=pd.DataFrame(pointers)
57  states=[]
58
59  states.append(viterbi_pd[T-1].argmax()) #take last viterbi value's tag
60  for i in range(T-1, 0, -1):
61      states.append(pointers_pd[i][states[-1]]) #backtrace the pointers'
            path
62
63  return states[::-1] #return reverse order
```

A.5 Success Statistics

```
1  def calculate_success_statistics(file_name):
2      words = []
3      correct_tags = []
4      predicted_tags = []
5
6      with open(file_name) as f:
7          for line in f:
8              word, correct_tag, predicted_tag=line.split()
9              words.append(word)
```

```
10          correct_tags.append(correct_tag)
11          predicted_tags.append(predicted_tag)
12
13      eos = '</s>'
14      matching_tags = [i for i, j in zip(correct_tags, predicted_tags) if i
            == j and i <> eos]
15      true_positives = len(matching_tags)
16      number_of_tags = len(correct_tags)
17      accuracy = float(true_positives) / float(number_of_tags)
18
19      false_positives = len(matching_tags)
20
21      #sentence based measurements...
22
23      sb_true_positives = 0
24      number_of_sentences = 0
25
26      sentence_correct_tags = []
27      sentence_predicted_tags = []
28      sentence_words = []
29      has_match = True
30      for word, correct_tag, predicted_tag in zip(words, correct_tags,
            predicted_tags):
31
32          sentence_words.append(word)
33          sentence_correct_tags.append(correct_tag)
34          sentence_predicted_tags.append(predicted_tag)
35
36          if correct_tag == eos:
37              if has_match:
38                  sb_true_positives = sb_true_positives + 1
39              '''else:
40
41                  print ' '.join(sentence_words)
```

```
42              print 'Correct tags: ', sentence_correct_tags
43              print 'Predicted tags: ', sentence_predicted_tags
44              print('******************\n')
45          '''
46
47          sentence_correct_tags = []
48          sentence_predicted_tags = []
49          sentence_words = []
50
51          number_of_sentences = number_of_sentences + 1
52          has_match = True
53          continue
54
55      has_match = has_match and (correct_tag == predicted_tag)
56
57  sb_accuracy = float(sb_true_positives) / float(number_of_sentences)
58
59  #report the results
60
61  print('----------------------------')
62  print('word based measurements')
63  print('true_positives', true_positives)
64  print('number_of_tags', number_of_tags)
65  print('accuracy', round(accuracy, 2) )
66
67
68  print('----------------------------')
69  print('sentence based measurements')
70  print('true_positives', sb_true_positives)
71  print('number_of_sentences', number_of_sentences)
72  print('accuracy', round(sb_accuracy, 2) )
73
74
75  confusion_matrix={}
```

```
76      for correct_tag, predicted_tag in zip(correct_tags, predicted_tags):
77          confusion_matrix.setdefault(correct_tag,{}).setdefault(predicted_tag
                ,0)
78          confusion_matrix[correct_tag][predicted_tag] = confusion_matrix[
                correct_tag][predicted_tag] + 1
79
80      confusion_matrix_dataframe= pd.DataFrame.from_dict(confusion_matrix)
81      confusion_matrix_filled=confusion_matrix_dataframe.copy()
82      confusion_matrix_filled=confusion_matrix_filled.fillna('0.0')
83
84      #confusion_matrix_filled['Adj']
85
86      tag_accuracy = {}
87
88      for tag in confusion_matrix.keys():
89          if (tag in confusion_matrix) and (tag in confusion_matrix[tag]):
90              tag_accuracy[tag] = float(confusion_matrix[tag][tag]) / sum(
                    confusion_matrix[tag].values())
91
92      print('------------------------------')
93      print('Tag␣accuracies')
94      print(tag_accuracy)
95
96      print('------------------------------')
97      print('Confusion␣matrix␣rows␣are␣conditioned,␣P(predicted_tag|
            correct_tag))')
98      confusion_matrix_filled.transpose()
```

A.6 Predictions

```
1  import codecs
2
3  def calculate_predictions(input_filename, word_likelihood_probs,
       tag_transition_probs):
4      output_filename = input_filename + '.predictions.txt'
```

```
5     input_file = codecs.open(input_filename, 'r', encoding='utf-8')
6     bos = '<s>' # beginning of sentence
7     eos = '</s>' # end of sentence
8
9     words = []
10    correct_tags = []
11    predicted_tags = []
12
13    current_sentence_words = []
14    sentence_count = 0
15    for line in input_file:
16        #print line
17        fields = line.strip().split('\t')
18
19        if len(fields) == 1:
20            if len(current_sentence_words) > 0:
21                sentence_count = sentence_count + 1
22                        current_sentence_predicted_tags = viterbi(
                              word_likelihood_probs,tag_transition_probs,
                              current_sentence_words)
23                predicted_tags.extend(current_sentence_predicted_tags)
24                current_sentence_words = []
25                current_sentence_tags = []
26
27                eos = '</s>'
28                words.append(eos)
29                correct_tags.append(eos)
30                predicted_tags.append(eos)
31
32                assert len(words) == len(correct_tags)
33                assert len(correct_tags) == len(predicted_tags)
34                if sentence_count % 100 == 0:
35                    print('sentence_count', sentence_count)
36            continue
```

```
37
38          word, tag = fields
39          current_sentence_words.append(word)
40
41          words.append(word)
42          correct_tags.append(tag)
43
44      print('final␣sentence_count', sentence_count)
45
46      print len(words), len(correct_tags), len(predicted_tags)
47
48      output_file = codecs.open(output_filename, 'w', encoding='utf-8')
49
50      for word, correct_tag, predicted_tag in zip(words, correct_tags,
            predicted_tags):
51          line = '\t'.join([word, correct_tag, predicted_tag])+'\n'
52          output_file.write(line)
53
54      output_file.close()
55      print('prediction␣completed')
56      return output_filename
```

A.7 Main Method

```
1  def main():
2      input_filename = 'Data/METUSABANCI_treebank.conll'
3      parsed_corpus_filename = parse_input_file(input_filename)
4      train_filename, test_filename = split_corpus(parsed_corpus_filename)
5      tag_transition_freqs, word_likelihood_freqs, tag_transition_probs,
           word_likelihood_probs = calculate_viterbi_probabilities(
           train_filename)
6
7      print('Success␣statistics␣for␣training␣set␣')
8      training_prediction_filename = calculate_predictions(train_filename,
           word_likelihood_probs, tag_transition_probs)
```

```
9   calculate_success_statistics(training_prediction_filename)
10
11  print('Success␣statistics␣for␣test␣set␣')
12  test_prediction_filename = calculate_predictions(test_filename,
        word_likelihood_probs, tag_transition_probs)
13  confusion_matrix = calculate_success_statistics(test_prediction_filename
        )
14
15  print('Free␣query')
16  tags = viterbi(word_likelihood_probs,tag_transition_probs)
17  print(tags)
```

References

[1] G. Eryigit, T. Ilbay, and O. A. Can, "Multiword expressions in statistical dependency parsing," in *Proceedings of the Second Workshop on Statistical Parsing of Morphologically Rich Languages (IWPT - 12th International Conference on Parsing Technologies)*, (Dublin, Ireland), pp. 45–55, Association for Computational Linguistics, October 2011.

[2] K. Oflazer, B. Say, D. Z. Hakkani-tür, and G. Tür, "Building a turkish treebank," 2003.

[3] N. B. Atalay, K. Oflazer, and B. Say, "The annotation process in the turkish treebank," 2003.

[4] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2000.

[5] M. Haulrich, *Different Approaches to Unknown Words in a Hidden Markov Model Part-of-Speech Tagger*. 2009.