

# PART OF SPEECH TAGGER FOR TURKISH

by

Levent Altunyurt & Zihni Orhan

Submitted to the Department of Computer Engineering

in partial fulfilment of the requirements

for the degree of

Bachelor of Science

in

Computer Engineering

Boğaziçi University

June 2006

# 1. ABSTRACT

**Project Name** : Part Of Speech Tagger for Turkish

**Term** : 2005/2006 2. Semester.

**Keywords** : Part of Speech Tagger for Turkish, POS Tagging, Part of Speech Tagging

**Summary** : This document explains the procedures, methodologies, design decisions that we made while we are building a part of speech tagger for Turkish language. The aim of the project was bringing something new to this topic which is relatively not researched very much. There are lots of part of speech taggers for English for example but it is very rare for Turkish. We tried to build our own project with our new ideas which we think can improve the performance of the current systems and can be a basis for the future work in this area.

## Table Of Contents :

1. ABSTRACT .....	- 2 -
Table Of Contents :.....	- 3 -
2. INTRODUCTION .....	- 4 -
2.1    What is Part-Of-Speech Tagging ?.....	- 4 -
2.2    Literature Overview.....	- 4 -
3. A NEW APPROACH TO POS TAGGING.....	- 7 -
3.1    Overview.....	- 7 -
3.1.1    Description .....	- 7 -
3.1.2    Input .....	- 7 -
3.1.3    Output .....	- 8 -
3.1.4    Resources .....	- 9 -
3.1.5    Process .....	- 10 -
3.2    Methodology.....	- 11 -
3.2.1    High Level Description.....	- 11 -
3.2.2    Middle Level Description .....	- 20 -
3.2.3    Low Level Description .....	- 20 -
4. RESULTS AND CONCLUSSION.....	- 30 -
5. REFERENCES .....	- 32 -
6. APPENDIX : LISTING OF THE PROGRAM (important parts) .....	- 33 -

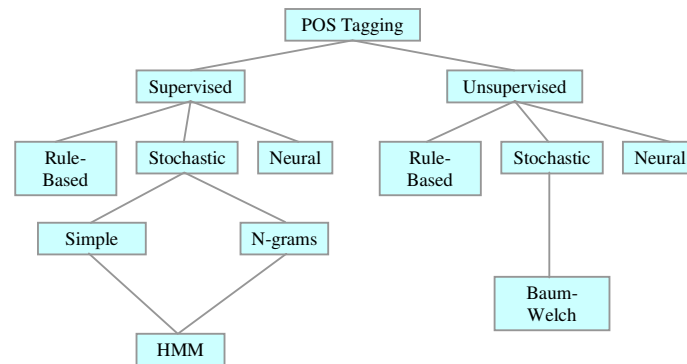
## 2. INTRODUCTION

### 2.1 What is Part-Of-Speech Tagging ?

**Part-of-speech tagging** is the process of marking up the words in a text with their corresponding parts of speech reflecting their syntactic category.

### 2.2 Literature Overview

There are many approaches to automated part-of-speech tagging, but the commonly approved ways will be discussed in this document, as an introduction.



**Figure 1 : Classification of POS Taggers**

**Supervised vs Unsupervised** distinction between POS taggers comes from the distinction of processes they use for tagging in terms of the degree of automation. <sup>[2]</sup> Supervised taggers typically rely on pre-tagged corpora<sup>1</sup> to serve as the basis for creating any tools to be used throughout the tagging process, for example: the tagger dictionary, the word/tag frequencies, the tag sequence probabilities and/or the rule set. Unsupervised models, on the other hand, are those which do not require a pretagged corpus but instead use sophisticated computational methods to automatically induce word groupings (i.e. tag sets) and based on those automatic groupings, to either calculate the probabilistic information needed by stochastic taggers or to induce the context rules needed by rule-based systems. <sup>[1]</sup>

---

<sup>1</sup> corpus (singular form corpora) is a large and structured set of texts (usually electronically stored and processed)

**Rule based POS taggers** use contextual information to assign tags to unknown or ambiguous words.<sup>2</sup> [3] These rules are often known as context frame rules. As an example, a context frame rule might say something like: If an ambiguous/unknown word X is preceded by a determiner and followed by a noun, tag it as an adjective.

$$det - X - n = X/adj$$

In addition to contextual information, many taggers use morphological information to aid in the disambiguation process. One such rule might be: if an ambiguous/unknown word ends in an -ing and is preceded by a verb, label it a verb (depending on your theory of grammar, of course).

Some systems go beyond using contextual and morphological information by including rules pertaining to such factors as capitalization and punctuation. Information of this type is of greater or lesser value depending on the language being tagged. In German for example, information about capitalization proves extremely useful in the tagging of unknown nouns.

Rule based taggers most commonly require supervised training; but, very recently there has been a great deal of interest in automatic induction of rules.

**The stochastic taggers**<sup>3</sup> disambiguate words based solely on the probability that a word occurs with a particular tag. In other words, the tag encountered most frequently in the training set is the one assigned to an ambiguous instance of that word<sup>[4]</sup>. The problem with this approach is that while it may yield a valid tag for a given word, it can also yield inadmissible sequences of tags.

An alternative to the word frequency approach is to calculate the probability of a given sequence of tags occurring. This is sometimes referred to as the **n-gram approach**, referring to the fact that the best tag for a given word is determined by the probability that it occurs with the n previous tags. The most common algorithm for implementing an n-gram approach is known as the *Viterbi Algorithm*<sup>[5]</sup>, a search algorithm which avoids the polynomial expansion of a breadth first search by "trimming" the search tree at each level

---

<sup>2</sup> An unknown / ambiguous word is a word that can belong to different syntactic categories in different context. For instance "yakacak" may belong to one of adjective, noun or verb syntactic categories according to context in Turkish.

<sup>3</sup> Any part of speech tagging model which somehow incorporates frequency or probability, i.e. statistics, may be properly labelled stochastic.

using the best N **Maximum Likelihood Estimates** (where n represents the number of tags of the following word).

The next level of complexity that can be introduced into a stochastic tagger combines the previous two approaches, using both tag sequence probabilities and word frequency measurements. This is known as a **Hidden Markov Model**.<sup>[6]</sup>

Hidden Markov Model taggers and visible Markov Model taggers may be implemented using the Viterbi algorithm<sup>[7]</sup>, and are among the most efficient of the tagging methods discussed here. HMM's cannot, however, be used in an automated tagging schema, since they rely critically upon the calculation of statistics on output sequences (tagstates). The solution to the problem of being unable to automatically train HMMs is to employ the *Baum-Welch Algorithm*, also known as the *Forward-Backward Algorithm*. This algorithm uses word rather than tag information to iteratively construct a sequences to improve the probability of the training data.

## **3. A NEW APPROACH TO POS TAGGING**

### **3.1 Overview**

#### **3.1.1 Description**

The aim of this project is to develop a Turkish part-of-speech tagger which not only uses the stochastic data gathered from Turkish corpus but also a combination of both morphological background of the word to be tagged and the characteristics of Turkish.

#### **3.1.2 Input**

The input of the system is;

§ 85 percent of the corpus, namely 6000 sentences of 7000 in METU-Sabancı Turkish Treebank.

§ 15 % of the corpus ,the remaining 1000 sentences in METU-Sabancı Turkish Treebank, which the program will try to analyze the POS of each word in it. Words will be analyzed as a whole, in other words the morphological root of the word will not be considered. For instance, for the word “evdekiler” the main consideration is on the word “evdekiler” while gathering stochastic data, not morphological root of it “ev”. Although any word sequence can be given as input, for ease of analyzing the success of program part of speech tagged word sequence – a part of corpus - is used.

§ Detailed information about the corpus will be given in “Resources” section.

##### **3.1.2.1 Assumptions**

§ As most of the stochastic based part-of-speech taggers, the success of our new system mainly depends on the percent of the word found in the training corpus. It is assumed that at least 50 percent of the input words found in the corpus.

##### **3.1.2.2 Restrictions**

§ The main restriction is the incompetence of a large enough training corpus. As mentioned, the size of training corpus has direct effect on the success. Although a very

large amount of whole corpus is used for training purposes the size of the training corpus is still very small when compared to studies in some other languages.

§ There are some restrictions because of the characteristics of Turkish. For instance, in English it is sufficient to find “go” or “goes” in the corpus, but on the other hand for only the verb “gitmek”, “gittim”, “gittin”, “gitti”, “gittik”, “gittiniz”, and “gittiler” should be found in order to tag this word appropriately.

### 3.1.3 Output

There are two types of output :

§ Tagged Text - “assignedTags.txt”

After completing the tagging process for each word in the Text To Be Tagged file - “input.txt” the “Tester” will write the result to the file “assignedTags.txt”.

The format is following;

Word	Assigned Tag
Bu	Adjective
Yılın	Noun
Yakacakları	Noun
Alındı	Verb
.	Punctuation
.....	

Figure 2: Tagged Text Output File

§ The Success Rate - “taggingSuccess.txt”

The evaluation of the result is in following format;

Number of Words	1500
Words Assigned Correctly	1165
Word Assigned Wrong	335



Success Rate	77.6
--------------	------

Figure 3: Success Rate Output File

### 3.1.4 Resources

METU-Sabancı Turkish Treebank is used as corpus in the system.

§ The corpus is a .xml document in the following format:

```
<?xml version="1.0" encoding="windows-1254" ?>
- <Set sentences="1">
- <S No="1">
  <W IX="1" LEM="" MORPH="" IG="[(1,"gir+Verb+Neg+Imp+A2sg")]" REL="[4,1,
    (SENTENCE)]">Girme</W>
  <W IX="2" LEM="" MORPH="" IG="[(1,"su+Noun+A3sg+Pnon+Dat")]" REL="[1,1,
    (OBJECT)]">suya</W>
  <W IX="3" LEM="" MORPH="" IG="[(1,"oğul+Noun+A3sg+P1sg+Nom")]" REL="[1,1,
    (VOCATIVE)]">oğlum</W>
  <W IX="4" LEM="" MORPH="" IG="[(1,"",+Punc")]" REL="[5,1,(COORDINATION)]">,</W>
  <W IX="5" LEM="" MORPH="" IG="[(1,"git+Verb+Pos+Prog1+A1pl")]" REL="[6,1,
    (SENTENCE)]">gidiyoruz</W>
  <W IX="6" LEM="" MORPH="" IG="[(1,".+Punc")]" REL="[,( )]">.</W>
</S>
</Set>
```

where;

- 1) **sentences**: <Set>Total number of sentence.
- 2) **No**: <S> Sets the place of the sentence in the queue
- 3) **IX**: <W> Sets the words place in the sentence
- 4) **LEM**: <W>The root in the Turkish lexicon.
- 5) **MORPH**: <W> Morphological root of the word.
- 6) **IG**: <W> Morphological analyze of the word.
- 7) **REL**: <W> Relative dependence to other words.
- 8) **ORG\_IGN**: <W> Analyze of word sequence.

The tag set is taken also from the same study;

§ The main tags are;

- +Noun**: Noun or derived noun
- +Adj**: Adjective or derived adjective
- +Adv**: Adverb or derived adverb
- +Verb**: verb or derived verb
- +Pron**: Pronoun or derived pronoun

- +Conj:** Conjunction
- +Det:** Determinant
- +Postp:** Postpronoun
- +Ques:** words written apart coming after question adjuncts
- +Interj:** Interjection
- +Num:** Numbers
- +Dup:** Duplicate reflectives i.e. açık saçık
- +Punc:** Punctuations

§ The corpus will be used both for training the POS Tagger and as a resource for testing the rate of success.

§ Tag set gives all possible tags that any Turkish word can be set.

Stochastic models need lexicon in order to assign possible tags to do words at the beginning of the tagging process. Since there is no available lexicon in Turkish, a need for a tool appeared for initial tag assignment. For this purpose we have used “*PCKimmo*”.

§ PCKimmo not only gives the probable initial tags for a word but also the morphological background of the word which is also used in the middle tagging process. The output taken from the PCKimmo is turned into the format :

*Yakacak N N V Adj*

Means “Yakacak” word is 50 percent Noun 25 percent Verb and 25 percent Adjective morphologically.

### **3.1.5 Process**

#### **§ Statistical Information Gathering**

“Statistics Analyzer” is responsible for constructing “Statistics Database” by using the “Training Corpus”. The process is called “Training” and at the end of the process the unigram, bigram, trigram and statistics (consisting the tag probabilities of the words at the beginning and at the end of the sentences.) are constructed.

#### § **Assigning Tag Probabilities To Each Word**

This is the core part of whole system and “Tagger” is responsible for the process with the help of “Tag Set Finder”. By means of “PCKimmo” and “Statistics Database” “Tagger” assigns the probability of tag for some word. For instance, “Yakacak” is “Adjective” with probability  $p_1$  and is “Noun” with probability of  $p_2$  and is “Verb” with probability of  $p_3$ .

#### § **Computing The Most Probable Path**

The next phase is computing the most probable path for each sentence. The attention is on sentence in this part and “Tagger” is also responsible the maintenance of this process. Firstly number of possible paths is computed. Then by multiplying the probabilities of each tag for each word the probability of the path found.

#### § **Assigning Tags To Words**

According to most probable path tags are assigned to each word. This is the last responsibility of the tagger. And the result is added to “Tagged Text” file.

#### • **Testing The Result**

After tags are assigned to each word to “Text To Be Tagged” , “Tester” will compare the result found by the POS Tagger and the actual result and give a success rate. And also a statistical result will be written to a result file explained above.

The detailed information about the process will be given in the following section.

## **3.2 Methodology**

### **3.2.1 High Level Description**

The project consist of several modules which have different tasks in whole tagging process. Since our model is a supervised, stochastic model, the process begins with geting stochastic data from the training corpus. *Statistics analyzer* is responsible for preparing necessary files for the subsequent steps. The constructed files and construction period is defined detaily below:

#### § **Computing Unigram Probabilities:**

Each word will be counted according to their appearance in the training corpus. How many times it appeared total, how many of them was with the tag “Noun”, “Adj” ... is calculated for each word in the corpus. As mentioned word encountered as a whole, for instance the main consideration point is “selamlaşma” in a word, it is not root of “selamlaşma” “selam”. The constructed file is in the following format:

	Total	Noun	Adj	Adv	Verb	Pron	Conj	...	Dup	Interj
yaz	1000	280	120	0	600	0	0	...	0	0
ben	500	215	0	0	285	132	0	...	0	0
oku	863	203	0	0	660	0	0	...	0	0
yarım	124	245	60	0	25	0	0	...	0	0

**Figure 4: Counting in Unigram Process**

Then, we calculate the probability for a word  $w$  given a tag  $t$ :  $P(w|t)$ . This is done by counting the occurrences of word  $w$  tagged with  $t$  in the treebank divided by the number of occurrences of tag  $t$ , as stated above:

$$P(w|t) = C(w, t) / C(t)$$

Then calculated probabilities for each word is written into “unigram\_prob.txt” file. The format is the following:

```
YaSamInIz Noun 33
YaSamInIz Adj 0
YaSamInIz Adv 0
YaSamInIz Verb 66
YaSamInIz Pron 0
YaSamInIz Conj 0
YaSamInIz Det 0
YaSamInIz Postp 0
YaSamInIz Ques 0
YaSamInIz Interj 0
YaSamInIz Num 0
YaSamInIz Dup 0
YaSamInIz Punc 0
....
```

**Figure 5: A part of unigram\_prob.txt**

## § Computing Bigram Probabilities

Each word couple will be counted according to their appearance in the training corpus, in a similar way done in finding unigram probabilities. Their tags will be hold also in the occurrence. For example “Ali gel” occurred 23 times together and in 20 of them “Ali” appeared as “Noun” and “gel” appeared as “Verb”. Since every combination of tags (if there are 30 tags for example, the probable occurrences are  $\text{comb}(30,2)$  which is a very big number) are so big that we cant hold in a table(it will be inefficient), we will have a table holding the frequencies of the tags that the words took separately. The format will look like the following :

	Ali gel		Eve gitti		Sarı gelin	
Total	45		23		36	

	Ali	gel	eve	gitti	sarı	gelin
Noun	12	34	3	20	23	2
Adj	32	23	12	1	2	12
Adv	0	2	7	2	12	13

**Figure 6: Counting in Bigram Process**

Then, for each word couple probabilitiy is computed with the following fomula :

$$P(t_i|t_{i-1}) = C(t_{i-1}t_i) / C(t_i)$$

The results are written into bigram\_prob.txt. A part from the file given below. For instance “Dükkancım bunlarI Noun Pron 50 ” means that 50 percent of “Dükkancım bunlarI” couple have “Noun Pron” as tags.

```

dersten cekilme Noun Noun 100
Dükkancım bunlarI Noun Pron 50
bunlarI söyledikten Pron Verb 100
söyledikten YaSamInIz Verb Noun 100
YaSamInIz boyunca Noun Postp 100
boyunca birçok Postp Det 100
birçok korkunuz Det Noun 100
Dükkancım bunlarI Noun Verb 50
korkunuz oldu Noun Verb 100
oldu Bir_ara Verb Det 100
.....

```

**Figure 7: A part of bigram\_prob.txt**

## § Computing Trigram Probabilities

The computing in triples is similar to bigram probability finding process, each word triple will be counted according to their appearance in the training corpus. For example, consider the sentence “ali akşam eve gitti.” The word triples here are “ali akşam eve” and “akşam eve gitti.” There will be an entry for this two triples and the format of the table will be similar to the one that we draw for the previous option but this time a little bit bigger.

Since the Turkish Treebank used in training process is not big enough, almost all of the triples and assigned tags have the 100 percent, when the probability calculated using the following formula:

$$P(t_i|t_{i-2}t_{i-1}) = C(t_{i-2}t_{i-1}t_i)/C(t_{i-2}t_{i-1})$$

At the end the results are written into the file trigram\_prob.txt :

```
hep birileri yanInIzda Adv Pron Noun 100
birileri yanInIzda olsun Pron Noun Verb 100
yanInIzda olsun istediniz Noun Verb Verb 100
Sonra ne oldu Adv Ques Verb 100
ne oldu BabanIz Ques Verb Noun 100
oldu BabanIz bir Verb Noun Det 100
BabanIz bir hastanede Noun Det Noun 100
bir hastanede tüm Det Noun Det 100
hastanede tüm sevdikleri Noun Det Verb 100
tüm sevdikleri yanIndayken Det Verb Verb 100
.....
```

Figure 8: A part of trigram\_prob.txt

## § Computing the Tag Probabilities of the Words at the beginning or at the end of a Sentence

Another statistics table has some grammatical information that we use if we don't have any information about a word that has never occurred in the training corpus. This table have entries like the following; what is the most probable tag of a word that appears in the beginning of a sentence and what is the most probable tag of a word that is at the end of a sentence? These kinds of information is kept for the words that we don't have much

information about it. After counting and computing process, the data is written into the “statistics.txt” file in the following format:

first_noun 36	last_noun 2
first_adj 0	last_adj 0
first_adv 17	last_adv 2
first_verb 9	last_verb 95
first_pron 17	last_pron 0
first_conj 7	last_conj 0
first_det 7	last_det 0
first_postp 0	last_postp 0
first_ques 2	last_ques 0
first_interj 2	last_interj 0
first_num 0	last_num 2
first_dup 0	last_dup 0
first_punc 0	last_punc 0

**Figure 9: The statistics.txt file**

These statistical operations will be done once and the results will be used many times by the program. But if any new corpus arrives, the operation may be repeated and new tables may be hold.

The next and curicial part of the whole project is tagger part. The *Tagger* uses several methods to decide the most probable tag of a word. The steps are as follows;

1. First of all, The “text to be tagged” is written from input file “input.txt”. The processing is sentence by sentence. For each word in the input sentence, each possible tag is assigned to word is computed by using “pckimmo” program. The results are written into “tags.txt”. A part of the created file is given below:

```
yeni N N ADJ
bir N
dOneme N V
kayIt N V
zamanInda N N
ders N
ekleme N V N
bIrakma V N
sUresinde N ADJ
veya CON
dersten N
Cekilme V N V N
ders N
....
```

**Figure 9: A part of tags.txt file**

2. As seen above file PCKimmo is not only giving only possible tags but also different ways of producing the tag for the word. We add this morphological data about word into consideration, also. The morphological background has a direct effect on the computed probability. The probabilities for each tag are also computed from the pckimmo output file and written into tagProbabilities.txt file. The format of the file is given below:

yeni	66.0	33.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bir	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dOneme	50.0	0.0	0.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
zamanInda	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ders	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ekleme	66.0	0.0	0.0	33.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bIrakma	50.0	0.0	0.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sUresinde	50.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
veya	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dersten	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Cekilme	50.0	0.0	0.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
.....													

**Figure 10: A part of tagProbabilities.txt file**

3. After the tag probabilities for each word in the input sentence is computed, the unigram effect state starts.

a. If the word is appeared in the corpus, it has an unigram probability for each tag. The new score for each word is computed by the following formula:

$$pckimmo+unigram\_tag\_probability = 0.5 * pckimmo\_tag\_probability + 0.5 * unigram\_tag\_probability$$

b. Else If the word is not appeared in the corpus but it is the first word in the sentence, the new score is computed by the formula :

$$pckimmo+unigram\_tag\_probability = 0.5 * pckimmo\_tag\_probability + 0.5 * statistics\_first\_tag\_probability$$

c. Else If the word is not appeared in the corpus but it is the last word in the sentence, the new score is computed by the formula:

$$pckimmo+unigram\_tag\_probability = 0.5 * pckimmo\_tag\_probability + 0.5 * statistics\_last\_tag\_probability$$



- d. Else If the word is not appeared in the corpus, the following formula is used:

$$p_{kimmo+unigram\_tag\_probability} = p_{kimmo\_tag\_probability}$$

The results are updated and written into kimmoUnigramProbabilities.txt file in the following format :

```
kayIt Det 3.5
kayIt Postp 0.0
kayIt Ques 1.0
kayIt Interj 1.0
kayIt Num 0.0
kayIt Dup 0.0
kayIt Punc 0.0
yeni Noun 33.0
yeni Adj 66.5
yeni Adv 0.0
yeni Verb 0.0
```

**Figure 11: A part of kimmoUnigramProbabilities.txt file**

4. After the tag probabilities and unigram effect for each word in the input sentence is computed, the bigram effect state starts.

- a. If the word(i) and word(i+1) is appeared in the corpus, it has an bigram probability with the tags tag(i) and tag(i+1). The new score for each word is computed by the following formula:

$$p_{kimmo+unigram+bigram\_tag\_probability} = 0.5 * p_{kimmo+unigram\_tag\_probability} + 0.5 * bigram\_tag\_probability$$

- b. Else ( the word has no bigram probability ) , the following formula is used:

$$p_{kimmo+unigram+bigram\_tag\_probability} = p_{kimmo+unigram\_tag\_probability}$$

The results are updated and written into kimmoUnigramBigramProbabilities.txt file in the following format :

```

kayIt Postp 0.0
kayIt Ques 1.0
kayIt Interj 1.0
kayIt Num 0.0
kayIt Dup 0.0
kayIt Punc 0.0
yeni Noun 33.0
yeni Adj 83.25
yeni Adv 0.0
yeni Verb 0.0
yeni Pron 0.0

```

**Figure 12: A part of kimmoUnigramBigramProbabilities.txt file**

5. After the tag probabilities and unigram and bigram effect for each word in the input sentence is computed, the trigram effect state starts.

a. If the word(i) , word(i+1) , word(i+2) is appeared in the corpus, it has an trigram probability with the tags tag(i) , tag(i+1), tag(i+2). The new score for each word is computed by the following formula:

$$p_{kimmo+unigram+bigram+trigram\_tag\_probability} = 0.5 * p_{kimmo+unigram+bigram\_tag\_probability} + 0.5 * trigram\_tag\_probability$$

b. Else ( the word has no bigram probability ) , the following formula is used:

$$p_{kimmo+unigram+bigram+trigram\_tag\_probability} = p_{kimmo+unigram+bigram\_tag\_probability}$$

The results are updated and written into kimmoUnigramBigramTrigramProbabilities.txt file in the following format:

```

kayIt Interj 1.0
kayIt Num 0.0
kayIt Dup 0.0
kayIt Punc 0.0
yeni Noun 33.0
yeni Adj 91.625
yeni Adv 0.0
yeni Verb 0.0
yeni Pron 0.0
yeni Conj 0.0
....

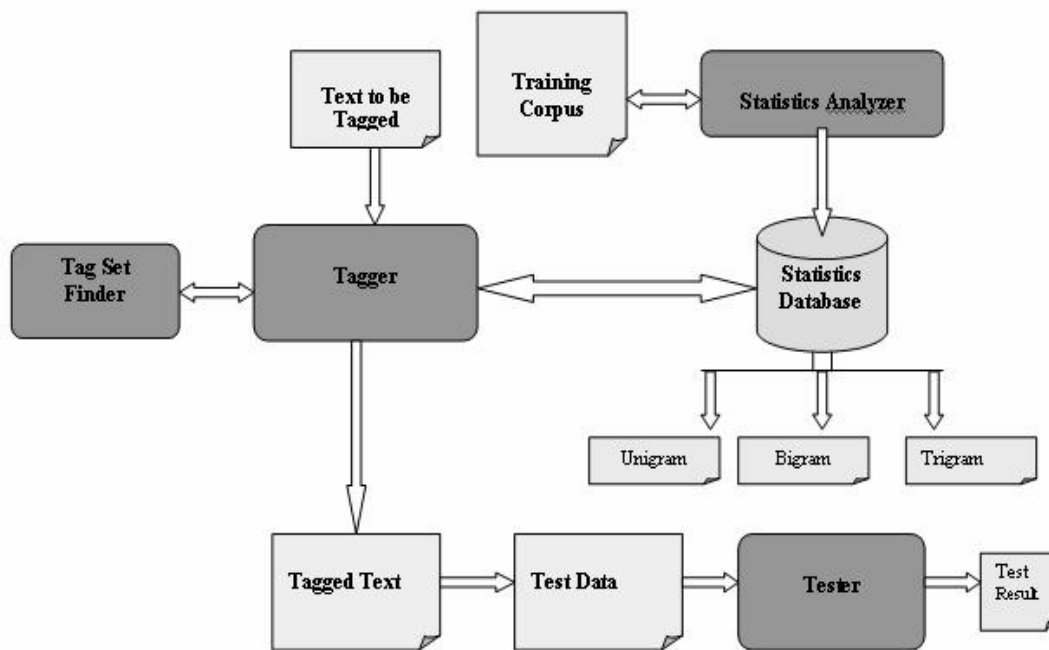
```

**Figure 13: A part of kimmoUnigramBigramProbabilities.txt file**

6. In order to determine the tag, we first look at the table that holds the information for the triples. If the triple occurred in the text, we put the tags in the tag set of the words. We keep on doing this until the end of the sentence.
7. When do the same thing for the couples. Since we cant expect the triple to occur very frequently, we try to find the couple and the possible tags.
8. Lastly we look at the table that holds the information for individual words. We determine the tags from here and put it in the tag set.
9. There can still be some words that never occurred in one of these tables. That have never appeared in the text. This time we have some options. We may look to the lexicon for the words that are in the form of a root. There is another option that we can use a program that determines the tag set of a word. these programs can be used as an API for the tagger.
10. After determining the tag set, now we compute the possible paths for the sentence. The paths are defined as follows. For the sentence “ali eve geldi” a possible path can be “isim isim fiil”. According to the number of tag sets, all possible paths will be calculated.
11. Now is the time for processing all the paths. We will use Hidden Markov Model to determine the probabilities for each word triple, couple or individual table that we already have. After calculating the probabilities, we give a score to each path.
12. Then we compare all the scores and take the one with highest score. According to the path we give the tags and write the sentence to a text file with its tags in the format of Turkish tree bank.

The last part of the project is calculating the performance. There will be a module that takes the tagged text that we processed and another tag text that was tagged before and by comparing word by word, prints a result of the system.

### 3.2.2 Middle Level Description



### 3.2.3 Low Level Description

#### pc\_kimmo\_parser.java

This java file analyses the output of the pc\_kimmo, and creates a file named “tags.txt”. The input for the file is the name of the output file of pc\_kimmo. By using pattern matching, this file selects the words from the output of pc\_kimmo and writes the possible tags of the word. On the command line type;

```
Java pc_kimmo_parser file_name
```

Here, the file\_name is the file created by the pc\_kimmo. The main purpose of the file is learning the all possible tags of a given word.

After executing the file, programme creates a file called “tags.txt” in the same directory as the pc\_kimmo\_parser.java. It includes lines as the following;

```
yeni N N ADJ
```

```
bir N
```

dOneme N V

kaylt N V

Here, first word of each line is the word and each word after it are the possible tags. One tag may appear more than once, meaning that there can be different meanings of that word but in the same morphological category.

### **bigram\_test\_provider.java**

This java file is used for creating a text file for further use in other programmes. Bigram.java , trigram.java and input\_provider.java files uses the output of this file. It simply the xml files which are included in Turkish tree bank and analyses them. After analyzing, takes the words and determines their tags and write them to a file named “bigram\_test.txt”. On the command line type;

Java bigram\_test\_provider *file\_name*

Here, the file\_name stands for the file that is included in the Turkish tree bank, ex:  
00084111-4.xml

After entering these xml files one by one, new sentences are appended to the end of the bigram\_test.txt file, which is later used in other programmes. The text file contains rows as the following;

Çocukluk Noun

yıllarınlzl Noun

bu Det

aptalca Adv

korku Noun

size Pron

zehir\_etti Verb

. Punc

Here each line contains a word and the tag of the word. The “.” is also included with its category.

## **input\_provider.java**

This file is used for preparing a appropriate input file for the main program which guesses the tags for the given input. We have a format for inputs and this file provides this format. It takes the bigram\_test.txt file and creates a file named "tagger\_input.txt". since the bigram\_test file contains the tags of the words, we have to get rid of the tags and have a simple file containing one word or a "." in each line. On the command line type;

Java input\_provider *file\_name*

Here, the file\_name stands for the "bigram\_test.txt". but it can be used for different purposes later, so we preferred to get the file name from the command line as an argument.

The format of the created file "tagger\_input.txt" is as the following;

```
Simdilerde
yeni
bir
korku
edindiniz
.
OGlunuzun
öleceđini
sizden
önce
ölebileceGini
düşünüyorsunuz
.
```

There is a word in each line and the sentences are separated by the "." Also "," appears if it is used in the original corpus.

## **unigram.java**

This java file is used for creating a text file including the words and their tags in each line. The corpus of Turkish tree bank is analysed and the relevant information is gathered by this file. Each xml file is taken and the words and tags are appenden at the end of the unigram.txt file. On the command line type;

Java unigram *file\_name*

*file\_name* stands for the names of the xml files in the Turkish tree bank. Enter each xml file one by one and what you get is a big text file including words and the tags they get in the corpus. This information is further used in the tagger.

The content of the unigram.txt is as follows;

YaSamInIz Noun

boyunca Postp

birçok Det

korkunuz Noun

oldu Verb

Bir\_ara Det

YaSamInIz Verb

babanIzla Noun

In each line, there is a word and the tag it takes in the corpus. This information is used for producing statistics showing which word is used in which tag how often. The unigram\_prob.txt file is created by using this unigram.txt

## **bigram.java**

This java file is used for creating a text file including the a couple of words and the tags they get . The corpus of Turkish tree bank is analysed and the relevant information is gathered by this file. "bigram\_test.txt" file is used for this purpose. The file is opened and the words are taken two by two. The aim is to identify which tag of a word is generally used after a known word and tag of it.

On the command line type;

```
Java bigram file_name
```

*file\_name* stands for the “bigram\_test.txt”.. This information is further used in the tagger.  
The content of the bigram.txt is as follows;

Dükkanclm Noun bunlarl Pron  
bunlarl Pron söyledikten Verb  
söyledikten Verb YaSamInlz Noun  
YaSamInlz Noun boyunca Postp  
boyunca Postp birçok Det

In each line, there are two words and two tags. The order is word1 tag1 word2 tag2 .This information is used for producing statistics showing word couples are used together in which tags. We know that the mostly used ones are more probable than others. The bigram\_prob.txt file is created by using this bigram.txt

### **trigram.java**

This java file is used for creating a text file including the triples of words and the tags they get . The corpus of Turkish tree bank is analysed and the relevant information is gathered by this file. “bigram\_test.txt” file is used for this purpose. The file is opened and the words are taken three by three. The aim is to identify which word triples are used together and in which tags mostly. On the command line type;

```
Java trigram file_name
```

*file\_name* stands for the “bigram\_test.txt”.. This information is further used in the tagger.  
The content of the bigram.txt is as follows;

Dükkanclm Noun bunlarl Pron söyledikten Verb  
bunlarl Pron söyledikten Verb YaSamInlz Noun  
söyledikten Verb YaSamInlz Noun boyunca Postp



YaSamInIz Noun boyunca Postp birçok Det  
boyunca Postp birçok Det korkunuz Noun  
birçok Det korkunuz Noun oldu Verb  
korkunuz Noun oldu Verb Bir\_ara Det

In each line, there are three words and three tags. The order is word1 tag1 word2 tag2 word3 tag3. This information is used for producing statistics showing word triples are used together in which tags. We know that the mostly used ones are more probable than others. The trigram\_prob.txt file is created by using this trigram.txt

### **unigram\_prob.java**

This file provides statistics about words by using the previously created “unigram.txt” file. A new file named “unigram\_prob.txt” which contains probabilities of the words taken a particular tag. Each tag is included and the probability with that tag. Unigram.txt file is taken and all words are counted. Then then the relative probabilities are calculated and written to the unigram\_prob.txt for further use in the tagger.

On the command line type;

Java unigram\_prob *file\_name*

File\_name stands for “unigram.txt” in this case. This is taken as an argument. This file should be in the same directory as the unigram\_prob.java. A new file containing probabilities is created in the same directory. The newly created file “unigram\_prob.txt” contains the information as follows;

boyunca Noun 28  
boyunca Adj 0  
boyunca Adv 14  
boyunca Verb 42  
boyunca Pron 0  
boyunca Conj 0  
boyunca Det 0  
boyunca Postp 14

boyunca Ques 0  
boyunca Interj 0  
boyunca Num 0  
boyunca Dup 0  
boyunca Punc 0

Each line contains a word, a tag and the relative probability of that word with that tag.

### **bigram\_prob.java**

This file provides statistics about word couples by using the previously created “bigram.txt” file. A new file named “bigram\_prob.txt” which contains probabilities of the word couples taken particular tags. Each tag set is included and the probabilities with those tags. bigram.txt file is taken and all word couples are counted. Then then the relative probabilities are calculated and written to the bigram\_prob.txt for further use in the tagger.

On the command line type;

Java bigram\_prob *file\_name*

File\_name stands for “bigram.txt” in this case. This is taken as an argument. This file should be in the same directorory as the bigram\_prob.java. A new file containing probabilities is created in the same directory. The newly created file “bigram\_prob.txt” contains the information as follows;

Dükkancılm bunlarl Noun Pron 50  
bunlarl söyledikten Pron Verb 100  
söyledikten YaSamInlz Verb Noun 100  
YaSamInlz boyunca Noun Postp 100  
boyunca birçok Postp Det 100  
birçok korkunuz Det Noun 100  
Dükkancılm bunlarl Noun Verb 50  
korkunuz oldu Noun Verb 100

The structure of each line is as follows; word1 word2 tag1 tag2 probability. This order is preserved and the tag couples that have never occurred are not included.

### **trigram\_prob.java**

This file provides statistics about word triples by using the previously created “trigram.txt” file. A new file named “trigram\_prob.txt” which contains probabilities of the word triples taken particular tags. Each tag set is included and the probabilities with those tags. trigram.txt file is taken and all word triples are counted. Then then the relative probabilities are calculated and written to the trigram\_prob.txt for further use in the tagger.

On the command line type;

Java trigram\_prob *file\_name*

File\_name stands for “trigram.txt” in this case. This is taken as an argument. This file should be in the same directorory as the trigram\_prob.java. A new file containing probabilities is created in the same directory. The newly created file “trigram\_prob.txt” contains the information as follows;

Dükkanclm bunlarl söyledikten Noun Pron Verb 66  
bunlarl söyledikten YaSamInlz Pron Verb Noun 100  
söyledikten YaSamInlz boyunca Verb Noun Postp 100  
YaSamInlz boyunca birçok Noun Postp Det 100  
boyunca birçok korkunuz Postp Det Noun 100  
birçok korkunuz oldu Det Noun Verb 100  
korkunuz oldu Bir\_ara Noun Verb Det 100  
oldu Bir\_ara babanlzla Verb Det Noun 100  
Bir\_ara babanlzla annenizin Det Noun Noun 100

The structure of each line is as follows; word1 word2 word3 tag1 tag2 tag3 probability. This order is preserved and the tag triples that have never occurred are not included.

## **Statistics.java**

This file uses the “bigram\_test.txt” file and produces some statistics about the words according to their position on the sentence. The first word and the last word of the sentence is considered here. This heuristics is used in the tagger.

On the command line type;

```
Java statistics file_name;
```

File\_name stands for the bigram\_test.txt in this case. The output file is named “statistics.txt”. It includes rows as follows;

```
first_noun 36
last_noun 2
first_adj 0
last_adj 0
first_adv 17
last_adv 2
first_verb 9
last_verb 95
```

There is a word and a number with it in each line. The word says if it is in the beginning of the sentence or at the last. For example nouns at the beginning of a sentence has a probability of %36 in this case.

## **Usage summery**

```
Java pc_kimmo_parser zOrnek.out
```

```
Java bigram_test_provider 122564.xml
```

```
Java input_provider bigram_test.txt
```

```
Java unigram 423454.xml
```

Java bigram bigram\_test.txt

Java trigram bigram\_test.txt

Java unigram\_prob unigram.txt

Java bigram\_prob bigram.txt

Java trigram\_prob trigram.txt

Java statistics bigram\_test.txt

## 4. RESULTS AND CONCLUSSION

Part of speech tagging is a wide research area in computational linguistics. While we were looking for papers in order to understand the concept, we found lots of papers written about part of speech tagiing for English. But we see that there is not enough research on this topic for Turkish. During our literature research, we couldn't find so much paper which is related to part of speech tagging for Turkish.

Our study is an introduction to the topic and there can be further improvements on it. What we aim in this project is first of all understand the part of speech tagging and try to create our own techniques on it. We wanted to add something new. The process of the project was so exciting and we learned from it very much. After all we believe that we created a part of speech tager which contains our own ideas and approximations to the problem. The steps that we went through was literature survey, proposal preperation, and coding. By combining different techniques and other programs as a supporter for the project, we reached to a satisfactory correctness level.

We have made 3 different tests by using the different parts of the curpus as an input text. The test are made on both only using PCKimmo and using stochastic data and Pckimmo both. The results are given below.

Number Of Sentences	300	220	576
Number Of Words	942	1253	1662
Tags Assigned Correctly Using PC Kimmo	653	787	1132
Success Rate	69.3 %	62.8%	68.1%
Tags Assigned Correctly Using PC Kimmo and Stochastic Data	798	1002	1367
Success Rate	84.7 %	79.9%	82.2%

The drawbacks of the program :

- Initial tag probabilities assigned to words at the beginning of the tagging process depend on PCKimmo, but PCKimmo do not always assign proper tags to words. For instance “terliklerini ADJ ADJ ADJ ADJ”, “neyi N N ADJ ADJ” are outputs of PCKimmo. As a result the probability of assigning true tag to word is highly decreases.
- The tag set of PCKimmo and the tagset of Treebank we have used is not one to one correspondent. For instance where “det” is a tag in Treebank’s tagset, there is no “det” output in PCKimmo.
- Most of the bigrams and trigrams found in the corpus have the probability 100 percent, because the corpus is not big enough. As a result if a couple or triple is found in the corpus, the score of their tag gets the most probable tag.
- Complexity of the program is very high, at each word for instance, it searches all unigram text file which is about 6 MB. As a result running program for big input is very time consuming.

Futurework:

- The program will give better performance with bigger training corpus. If a large enough corpus can be constructed, the programs output will be more satisfactory.
- The code can be fixed due to problems occur because of alignment problems in the input text. Such as punctuations.
- Since searching operations takes very long time in the program. Time complexity of the program can be decreased by using a Database Management System instead of text files.
- More heuristics can be added to process in order to come up with a better result.

## 5. REFERENCES

1. Brill, Eric. 1995. Unsupervised learning of disambiguation rules for part of speech tagging.
2. Schütze, Hinrich. 1993. Part-of-speech induction from scratch. *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, 251-258.
3. Brill, Eric. 1992. A simple rule-based part of speech tagger. *Proceedings of the Third Annual Conference on Applied Natural Language Processing, ACL*.
4. Allen, James. 1995. *Natural Language Understanding*. Redwood City, CA: Benjamin Cummings.
5. Brill, Eric & Marcus, M. 1993. Tagging an unfamiliar text with minimal human supervision. ARPA Technical Report.
6. Baum, L. 1972. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. *Inequalities* 3:1-8.
7. Kupiec, J. 1992. Robust Part-of-speech tagging using a hidden Markov model. *Computer Speech and Language* 6.



## 6. APPENDIX : LISTING OF THE PROGRAM (important parts)

### Assign\_Tags.java

```
import java.io.*;
import java.util.regex.*;
import javax.swing.*;
import java.beans.*;
import java.util.*;

public class Assign_Tags
{
    public static double [] tagProbabilityArray;

    public static void main(String[] argv)
    {
        String foundProbabilityFileName = argv[0];
        initialize_tag_probabilities();

        assign(foundProbabilityFileName);
    }

    public static void initialize_tag_probabilities()
    {
        tagProbabilityArray = new double[13];

        for (int i=0; i<12;i++)
        {
            tagProbabilityArray[i]= 0.0;
        }
    }

    public static void assign (String probabilityFile)
    {
        File inputFile = new File(probabilityFile);

        File outputFile = new File("assignedTags.txt");

        String maxProbabilityTag = null;
        double maxProbability = 0.0;

        String inputLine = null;
        String [] inputLineString = null;
        inputLineString = new String[10000];

        int inCounter = 0;

        try
        {
            BufferedReader inputReader = new BufferedReader(new
FileReader(inputFile));
            BufferedWriter outputWriter = new BufferedWriter(new
FileWriter(outputFile,true),100);

            while ((inputLine = inputReader.readLine()) != null)
            {
                inCounter = 0;

                System.out.println (inputLine);
```

```

        StringTokenizer inputTokens = new StringTokenizer(inputLine);

        while (inputTokens.hasMoreTokens())
        {
            inputLineString[inCounter] = inputTokens.nextToken();
            inCounter++;
        }

        tagProbabilityArray[0] =
Double.parseDouble(inputLineString[2]);

        maxProbabilityTag = inputLineString[1];
        maxProbability = tagProbabilityArray[0];

        for (int i = 1; i<13; i++)
        {
            inCounter = 0;
            try
            {
                inputLine = inputReader.readLine();

                StringTokenizer inputTokens2 = new
StringTokenizer(inputLine);

                while (inputTokens2.hasMoreTokens())
                {
                    inputLineString[inCounter] =
inputTokens2.nextToken();
                    inCounter++;
                }

                tagProbabilityArray[i] =
Double.parseDouble(inputLineString[2]);

                if (tagProbabilityArray[i] > maxProbability)
                {
                    maxProbability = tagProbabilityArray[i];
                    maxProbabilityTag = inputLineString[1];
                }

            }
            catch (NullPointerException e)
            {}

        }

        outputWriter.write(inputLineString[0] + " " +
maxProbabilityTag + " " +maxProbability);
        outputWriter.newLine();
    }

    inputReader.close();
    outputWriter.close();
}
catch (IOException e)
{
    System.err.println(e);
    System.out.println("Cannot read the input file");
} //end of catch
}

```

```
}
```

## Find\_Tags.java

```
import java.io.*;
import java.util.regex.*;
import javax.swing.*;
import java.beans.*;
import java.util.*;

public class Find_Tags
{
    public static double [] tagProbabilityArray;
    private static int [] tagCounterArray;
    private static double [] firstWordArray;
    private static double [] lastWordArray;

    public static void main(String[] argv)
    {
        String inputFileName = "input.txt";
        String tagsFileName = "tags.txt";
        String unigramFileName ="unigram_prob.txt";
        String bigramFileName ="bigram_prob.txt";
        String trigramFileName = "trigram_prob.txt";
        String statisticsFileName = "statistics.txt";

        initialize_tag_probabilities();
        initialize_tag_counter();
        initialize_first_last_word_probabilities(statisticsFileName);

        find_tag_probabilities(inputFileName,tagsFileName);
        unigram_effect(inputFileName,unigramFileName);
        bigram_effect(inputFileName,bigramFileName);
        trigram_effect(inputFileName,trigramFileName);
    }

    public static void initialize_tag_probabilities()
    {
        tagProbabilityArray = new double[13];

        for (int i=0; i<12;i++)
        {
            tagProbabilityArray[i]= 0.0;
        }
    }

    public static void initialize_first_last_word_probabilities(String
statistics)
    {
        firstWordArray = new double[13];
        lastWordArray = new double[13];

        String statisticsLine = null;
        String [] statisticsLineString;
        statisticsLineString = new String[5];

        File statisticsFile  = new File(statistics);
    }
}
```

```

        try
        {
            BufferedReader statisticsReader = new BufferedReader(new
FileReader(statisticsFile));
            int i = 0;
            int stCounter = 0;

            while ((statisticsLine = statisticsReader.readLine()) != null)
            {
                StringTokenizer statisticsTokens = new
StringTokenizer(statisticsLine);

                while (statisticsTokens.hasMoreTokens())
                {
                    statisticsLineString[stCounter] =
statisticsTokens.nextToken();
                    stCounter++;
                }

                firstWordArray[i]= Double.parseDouble(statisticsLineString[1]);

                stCounter = 0;
                statisticsLine = statisticsReader.readLine();
                StringTokenizer statisticsTokens2 = new
StringTokenizer(statisticsLine);

                while (statisticsTokens2.hasMoreTokens())
                {
                    statisticsLineString[stCounter] =
statisticsTokens2.nextToken();
                    stCounter++;
                }

                lastWordArray[i]= Double.parseDouble(statisticsLineString[1]);

                i++;
                stCounter = 0;
            }
        }

        catch(IOException e)
        {
            System.err.println(e);
            System.out.println("Cannot read the input file");
        }

    }

    public static void initialize_tag_counter()
    {
        tagCounterArray = new int[13];

        for (int i=0; i<12;i++)
        {
            tagCounterArray[i] = 0;
        }
    }

    public static void unigram_effect(String input,String unigram)
    {
        File inputFile = new File(input);
        File unigramFile = new File(unigram);
        File tagProbabilityFile = new File("tagProbabilities.txt");
    }

```

```

File outputFile = new File("kimmoUnigramProbabilities.txt");

try {
    BufferedReader inputReader = new BufferedReader(new
FileReader(inputFile));
    BufferedReader unigramReader = new BufferedReader(new
FileReader(unigramFile));
    BufferedReader tagProbabilityReader = new BufferedReader(new
FileReader(tagProbabilityFile));

    unigramReader.mark(100000);
    tagProbabilityReader.mark(100000);

    BufferedWriter outputWriter = new BufferedWriter(new
FileWriter(outputFile,true),100);

    String inputLine = null;
    String tagProbabilityLine = null;

    String [] inputLineString = null;
    inputLineString = new String[100000];

    String currentInputWord = null;

    String unigramLine = null;

    String [] unigramLineString = null;
    unigramLineString = new String[3];

    String [] tagProbabilityLineString = null;
    tagProbabilityLineString = new String[14];

    String currentUnigramWord = null;
    String currentUnigramTag = null;
    int currentUnigramProbability = 0;

    String outputString=null;

    int inputTokenCounter = 0;

    int inCounter = 0;
    int tagProbabilityCounter = 0;

    while ((inputLine = inputReader.readLine()) != null)
    {
        StringTokenizer inputTokens = new StringTokenizer(inputLine);
        inputTokenCounter = inputTokens.countTokens();

        while (inputTokens.hasMoreTokens())
        {
            inputLineString[inCounter] = inputTokens.nextToken();
            inCounter++;
        }
    }

    boolean inputFound ;
    int curIndex = 0;
    int uniCounter = 0;
    int inFoundCount = 0;

    while ((unigramLine = unigramReader.readLine()) != null)

```

```

{
    uniCounter = 0;
    inputFound=false;

    StringTokenizer unigramTokens = new
StringTokenizer(unigramLine);

    while (unigramTokens.hasMoreTokens())
    {
        unigramLineString[uniCounter] = unigramTokens.nextToken();
        uniCounter++;
    }

    currentInputWord = inputLineString[curIndex];
    currentUnigramWord = unigramLineString[0];
    currentUnigramTag = unigramLineString[1];
    try
    {
        currentUnigramProbability =
Integer.parseInt(unigramLineString[2]);
    }
    catch(NumberFormatException e)
    {
        currentUnigramProbability = 0;
    }

    try
    {
        if (currentInputWord.equals(currentUnigramWord))
        {
            boolean tagProbabilityNotFound = true;
            while ((tagProbabilityLine =
tagProbabilityReader.readLine()) != null && tagProbabilityNotFound )
            {
                StringTokenizer tagProbabilityTokens = new
StringTokenizer(tagProbabilityLine);

                tagProbabilityCounter = 0;

                while (tagProbabilityTokens.hasMoreTokens())
                {
                    tagProbabilityLineString[tagProbabilityCounter]
= tagProbabilityTokens.nextToken();
                    tagProbabilityCounter++;
                }
                if
(currentInputWord.equals(tagProbabilityLineString[0]))
                {
                    for (int k = 0; k < 13; k++)
                    {

tagProbabilityArray[k]=Double.parseDouble(tagProbabilityLineString[k+1]);
                    }
                    tagProbabilityNotFound = false;
                }
            }
            tagProbabilityReader.reset();
            if (currentUnigramTag.equals("Noun"))
            {

```

```

tagProbabilityArray[0]
=(0.5*tagProbabilityArray[0]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " + tagProbabilityArray[0]);
outputWriter.newLine();
}
else if (currentUnigramTag.equals("Adj") )
{
tagProbabilityArray[1]
=(0.5*tagProbabilityArray[1]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[1]);
outputWriter.newLine();
}
else if (currentUnigramTag.equals("Adv"))
{
tagProbabilityArray[2]
=(0.5*tagProbabilityArray[2]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[2]);
outputWriter.newLine();
}
else if (currentUnigramTag.equals("Verb"))
{
tagProbabilityArray[3]
=(0.5*tagProbabilityArray[3]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[3]);
outputWriter.newLine();
}
else if (currentUnigramTag.equals("Pron"))
{
tagProbabilityArray[4]
=(0.5*tagProbabilityArray[4]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[4]);
outputWriter.newLine();
}
else if (currentUnigramTag.equals("Conj"))
{
tagProbabilityArray[5]
=(0.5*tagProbabilityArray[5]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[5]);
outputWriter.newLine();
}
else if (currentUnigramTag.equals("Det"))
{
tagProbabilityArray[6]
=(0.5*tagProbabilityArray[6]) +(0.5*currentUnigramProbability);
inFoundCount ++;
outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[6]);
outputWriter.newLine();
}
}

```

```

        else if (currentUnigramTag.equals("Postp"))
        {
            tagProbabilityArray[7]
            =(0.5*tagProbabilityArray[7]) +(0.5*currentUnigramProbability);
            inFoundCount ++;
            outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[7]);
            outputWriter.newLine();
        }
        else if (currentUnigramTag.equals("Ques"))
        {
            tagProbabilityArray[8]
            =(0.5*tagProbabilityArray[8]) +(0.5*currentUnigramProbability);
            inFoundCount ++;
            outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[8]);
            outputWriter.newLine();
        }
        else if (currentUnigramTag.equals("Interj"))
        {
            tagProbabilityArray[9]
            =(0.5*tagProbabilityArray[9]) +(0.5*currentUnigramProbability);
            inFoundCount ++;
            outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[9]);
            outputWriter.newLine();
        }
        else if (currentUnigramTag.equals("Num"))
        {
            tagProbabilityArray[10]
            =(0.5*tagProbabilityArray[10]) +(0.5*currentUnigramProbability);
            inFoundCount ++;
            outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[10]);
            outputWriter.newLine();
        }
        else if (currentUnigramTag.equals("Dup"))
        {
            tagProbabilityArray[11]
            =(0.5*tagProbabilityArray[11]) +(0.5*currentUnigramProbability);
            inFoundCount ++;
            outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[11]);
            outputWriter.newLine();
        }
        else if (currentUnigramTag.equals("Punc"))
        {
            tagProbabilityArray[12] = tagProbabilityArray[12] +
currentUnigramProbability;
            inFoundCount ++;
            outputWriter.write(currentInputWord+" "+
currentUnigramTag + " " +tagProbabilityArray[12]);
            outputWriter.newLine();
        }
    }

    if (inFoundCount%13==0)
    {
        inputFound = true;
        unigramReader.reset() ;
    }

```



```

        if ( curIndex < inCounter-1 )
        {
            curIndex++;
        }
    }

    else if( currentUnigramWord.equals("#"))
    {
        unigramReader.reset();
        curIndex++;

        boolean tagProbabilityNotFound = true;

        while ((tagProbabilityLine =
tagProbabilityReader.readLine()) != null && tagProbabilityNotFound )
        {
            StringTokenizer tagProbabilityTokens = new
StringTokenizer(tagProbabilityLine);

            tagProbabilityCounter = 0;

            while (tagProbabilityTokens.hasMoreTokens())
            {
                tagProbabilityLineString[tagProbabilityCounter]
= tagProbabilityTokens.nextToken();
                tagProbabilityCounter++;
            }

            if
(currentInputWord.equals(tagProbabilityLineString[0]))
            {
                for (int k = 0; k < 13; k++)
                {
                    try
                    {
.....
.....

                }

                tagProbabilityReader.reset();
            }
        }

        catch(NullPointerException e)
        {}

    }

    outputWriter.close();
    inputReader.close();

}

catch(IOException e)
{
    System.err.println(e);
    System.out.println("Cannot read the input file input.txt");
} //end of catch

```

```

        return;
    }
    public static void bigram_effect(String input,String bigram)
    {
        File inputFile = new File(input);
        File bigramFile = new File(bigram);
        File tagPreviousProbabilityFile = new
File("kimmoUnigramProbabilities.txt");
        File outputFile = new File("kimmoUnigramBigramProbabilities.txt");

        try
        {
            BufferedReader inputReader = new BufferedReader(new
FileReader(inputFile));
            BufferedReader bigramReader = new BufferedReader(new
FileReader(bigramFile));
            BufferedReader tagPreviousProbabilityReader = new
BufferedReader(new FileReader(tagPreviousProbabilityFile));

            bigramReader.mark(100000);
            tagPreviousProbabilityReader.mark(100000);

            BufferedWriter outputWriter = new BufferedWriter(new
FileWriter(outputFile,true),100);

            String inputLine = null;
            String inputNextLine = null;
            String tagPreviousProbabilityLine=null;
            String tagPPPreviousrobabilityLine = null;

            String [] inputLineString = null;
            inputLineString = new String[100000];

            String currentInputWord = null;
            String nextInputWord = null;

            String bigramLine = null;

            String [] bigramLineString = null;
            bigramLineString = new String[10];

            String [] tagPreviousProbabilityLineString = null;
            tagPreviousProbabilityLineString = new String[3];

            String currentBigramWord = null;
            String currentBigramTag = null;

            String nextBigramWord = null;
            String nextBigramTag = null;

            String nextInputTag = null;

            double currentBigramProbability = 0.0;
            double previousProbability = 0.0;
            String previousTag = null;

            String outputString=null;

            int inCounter = 0;
            int tagProbabilityCounter = 0;

```

```

int curIndex = 0;
int inFoundCount = 0;

boolean markNext = false;

while ((inputLine = inputReader.readLine()) != null)
{
    StringTokenizer inputTokens = new StringTokenizer(inputLine);

    while (inputTokens.hasMoreTokens())
    {
        inputLineString[inCounter] = inputTokens.nextToken();
        inCounter++;
    }
}
int biCounter = 0;

int previousProbCount = 0;

while ((bigramLine = bigramReader.readLine()) != null &&
curIndex<inCounter-1)
{
    biCounter = 0;

    StringTokenizer bigramTokens = new StringTokenizer(bigramLine);

    while (bigramTokens.hasMoreTokens())
    {
        bigramLineString[biCounter] = bigramTokens.nextToken();
        biCounter++;
    }

    if (curIndex<inCounter-1)
    {
        try
        {
            currentInputWord = inputLineString[curIndex];
            nextInputWord = inputLineString[curIndex+1];
        }
        catch (NullPointerException e)
        {}
    }

    currentBigramWord = bigramLineString[0];
    currentBigramTag = bigramLineString[2];
    nextBigramWord = bigramLineString[1];
    nextBigramTag = bigramLineString[3];

    try
    {
        currentBigramProbability =
Double.parseDouble(bigramLineString[10]);
    }
    catch (NumberFormatException e)
    {
        currentBigramProbability = 0.0;
    }

    try
    {

```

```

        if ((currentInputWord.equals(currentBigramWord) &&
nextInputWord.equals(nextBigramWord)) || (markNext && previousProbCount%13==0)
)
    {
        if (markNext)
        {
            currentBigramWord = bigramLineString[1];
            currentBigramTag = bigramLineString[3];

            //      System.out.println( currentBigramWord+"-
"+currentBigramTag);
        }
        boolean tagProbabilityNotFound = true;
        //      System.out.println( "asil" + currentBigramWord+"-
"+currentBigramTag);
        .....
        .....
public static void trigram_effect(String input,String trigram)
    {
        File inputFile  = new File(input);
        File trigramFile = new File(trigram);
        File tagPreviousProbabilityFile = new
File("kimmoUnigramBigramProbabilities.txt");
        File outputFile = new
File("kimmoUnigramBigramTrigramProbabilities.txt");

        try
        {
            BufferedReader inputReader = new BufferedReader(new
FileReader(inputFile));
            BufferedReader trigramReader = new BufferedReader(new
FileReader(trigramFile));
            BufferedReader tagPreviousProbabilityReader = new
BufferedReader(new FileReader(tagPreviousProbabilityFile));

            trigramReader.mark(100000);
            tagPreviousProbabilityReader.mark(100000);

            BufferedWriter outputWriter = new BufferedWriter(new
FileWriter(outputFile,true),100);

            String inputLine = null;
            String tagPreviousProbabilityLine = null;

            String [] inputLineString = null;
            inputLineString = new String[100000];

            String currentInputWord = null;
            String nextInputWord = null;
            String nextNextInputWord = null;

            String trigramLine = null;

            String [] trigramLineString = null;
            trigramLineString = new String[14];

            String [] tagPreviousProbabilityLineString = null;
            tagPreviousProbabilityLineString = new String[3];

            String currentTrigramWord = null;

```

```

String currentTrigramTag = null;

String nextTrigramWord = null;
String nextTrigramTag = null;

String nextNextTrigramWord = null;
String nextNextTrigramTag = null;

String nextInputTag = null;
String nextNextInputTag = null;

double currentTrigramProbability = 0.0;
double previousProbability = 0.0;
String previousTag = null;

String outputString=null;

int inCounter = 0;
int tagProbabilityCounter = 0;

int curIndex = 0;
int inFoundCount = 0;

while ((inputLine = inputReader.readLine()) != null)
{
    StringTokenizer inputTokens = new StringTokenizer(inputLine);

    while (inputTokens.hasMoreTokens())
    {
        inputLineString[inCounter] = inputTokens.nextToken();
        inCounter++;
    }
}

int triCounter =0;

int previousProbCount = 0;

while ((trigramLine = trigramReader.readLine()) != null &&
curIndex<inCounter-1)
{
    triCounter = 0;

    StringTokenizer trigramTokens = new
StringTokenizer(trigramLine);

    while (trigramTokens.hasMoreTokens())
    {
        trigramLineString[triCounter] = trigramTokens.nextToken();
        triCounter++;
    }

    if (curIndex<inCounter-1)
    {
        try
        {
            currentInputWord = inputLineString[curIndex];
            nextInputWord = inputLineString[curIndex+1];
            nextNextInputWord = inputLineString[curIndex+2];
        }
        catch (NullPointerException e)
        {}
    }
}

```

```

    }

    currentTrigramWord = trigramLineString[0];
    currentTrigramTag = trigramLineString[3];
    nextTrigramWord = trigramLineString[1];
    nextTrigramTag = trigramLineString[4];
    nextNextTrigramWord = trigramLineString[2];
    nextNextTrigramTag = trigramLineString[5];

    try
    {
        currentTrigramProbability =
Double.parseDouble(trigramLineString[6]);
    }
    catch (NumberFormatException e)
    {
        currentTrigramProbability = 0.0;
    }

    try
    {
        if (currentInputWord.equals(currentTrigramWord) &&
nextInputWord.equals(nextTrigramWord) &&
nextNextInputWord.equals(nextNextTrigramWord))
        {

            boolean tagProbabilityNotFound = true;

            while ((tagPreviousProbabilityLine =
tagPreviousProbabilityReader.readLine()) != null && tagProbabilityNotFound )
            {
                StringTokenizer tagPreviousProbabilityTokens = new
StringTokenizer(tagPreviousProbabilityLine);

                tagProbabilityCounter = 0;

                while
(tagPreviousProbabilityTokens.hasMoreTokens())
                {

tagPreviousProbabilityLineString[tagProbabilityCounter] =
tagPreviousProbabilityTokens.nextToken();
                    tagProbabilityCounter++;
                }
                if
(currentInputWord.equals(tagPreviousProbabilityLineString[0]) &&
currentTrigramTag.equals(tagPreviousProbabilityLineString[1]) )
                {
                    previousProbability =
Double.parseDouble(tagPreviousProbabilityLineString[2]);

                    if (currentTrigramTag.equals("Noun"))
                    {
                        tagProbabilityArray[0]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
                        previousProbCount++;
                        outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " + tagProbabilityArray[0]);
                        outputWriter.newLine();
                    }
                    else if (currentTrigramTag.equals("Adj") )
                    {

```

```

tagProbabilityArray[1]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[1]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Adv"))
{
tagProbabilityArray[2]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[2]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Verb"))
{
tagProbabilityArray[3]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[3]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Pron"))
{
tagProbabilityArray[4]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[4]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Conj"))
{
tagProbabilityArray[5]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[5]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Det"))
{
tagProbabilityArray[6]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[6]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Postp"))
{
tagProbabilityArray[7]
=(0.5*previousProbability) +(0.5*currentTrigramProbability);
previousProbCount++;
outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[7]);
outputWriter.newLine();
}
else if (currentTrigramTag.equals("Ques"))

```

```

        {
            tagProbabilityArray[8]
            =(0.5*previousProbability) +(0.5*currentTrigramProbability);
            previousProbCount++;
            outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[8]);
            outputWriter.newLine();
        }
        else if (currentTrigramTag.equals("Interj"))
        {
            tagProbabilityArray[9]
            =(0.5*previousProbability) +(0.5*currentTrigramProbability);
            previousProbCount++;
            outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[9]);
            outputWriter.newLine();
        }
        else if (currentTrigramTag.equals("Num"))
        {
            tagProbabilityArray[10]
            =(0.5*previousProbability) +(0.5*currentTrigramProbability);
            previousProbCount++;
            outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[10]);
            outputWriter.newLine();
        }
        else if (currentTrigramTag.equals("Dup"))
        {
            tagProbabilityArray[11]
            =(0.5*previousProbability) +(0.5*currentTrigramProbability);
            previousProbCount++;
            outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[11]);
            outputWriter.newLine();
        }
        else if (currentTrigramTag.equals("Punc"))
        {
            tagProbabilityArray[12] =
            (0.5*previousProbability) +(0.5*currentTrigramProbability);
            previousProbCount++;
            outputWriter.write(currentInputWord+" "+
currentTrigramTag + " " +tagProbabilityArray[12]);
            outputWriter.newLine();
        }
    }
    else if
    (currentInputWord.equals(tagPreviousProbabilityLineString[0]))
    {
        previousTag =
tagPreviousProbabilityLineString[1];
        previousProbability =
Double.parseDouble(tagPreviousProbabilityLineString[2]);

        outputWriter.write(currentInputWord+" "+
previousTag + " " + previousProbability);
        outputWriter.newLine();

        previousProbCount++;
        if(previousProbCount%13 == 0)
        {
            tagProbabilityNotFound = false;

```



```

        curIndex++;
    }

    }
    tagPreviousProbabilityReader.reset();

    trigramReader.reset();
}

else if( currentTrigramWord.equals("#"))
{
    trigramReader.reset();

    boolean tagProbabilityNotFound = true;

    String tagPreviousProbabilityNextLine = null;

    while ((tagPreviousProbabilityLine =
tagPreviousProbabilityReader.readLine()) != null && tagProbabilityNotFound )
    {
        StringTokenizer tagPreviousProbabilityTokens = new
StringTokenizer(tagPreviousProbabilityLine);

        tagProbabilityCounter = 0;

        while
(tagPreviousProbabilityTokens.hasMoreTokens())
        {

tagPreviousProbabilityLineString[tagProbabilityCounter] =
tagPreviousProbabilityTokens.nextToken();
            tagProbabilityCounter++;
        }

        if
(currentInputWord.equals(tagPreviousProbabilityLineString[0]))
        {
            previousProbability =
Double.parseDouble(tagPreviousProbabilityLineString[2]);
            previousTag =
tagPreviousProbabilityLineString[1];

            outputWriter.write(currentInputWord+" "+
previousTag + " " + previousProbability);
            outputWriter.newLine();

            previousProbCount++;
            if(previousProbCount%13 == 0)
            {
                tagProbabilityNotFound = false;
                curIndex++;
            }
        }
    }
    tagPreviousProbabilityReader.reset();
}

}

catch(IOException e)

```

```

        {
            System.err.println(e);
            System.out.println("Cannot read the input file");
        } //end of catch
    }

    outputWriter.close();
    inputReader.close();
}
catch(IOException e)
{
    System.err.println(e);
    System.out.println("Cannot read the input file");
} //end of catch
}

```