DEVELOPING METHODS FOR PART OF SPEECH TAGGING IN TURKISH

by

Berna Arslan

&

Özlem Patan

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 2009

ABSTRACT

Project Name: Developing Methods for Part of Speech Tagging in Turkish

Project Team: Berna Arslan and Özlem Patan

Term: 2008/09, II. Semester

Keywords: Part of Speech Tagger for Turkish, Support Vector Machines, Part of Speech Tagging

Summary: The aim of the project is to develop methods for part of speech tagging in Turkish. This study is carried out by using support vector machine approach, which has not been used for Turkish language before. Main focus is to find appropriate features that will distinguish different grammatical classes and to test these features with a tool, that has been developed for support vector machines approach, namely SVMLight. Results indicate that a success level of 75 per cent has been reached with a relatively low feature number. Success rate can be improved by using a larger number of training words and features.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 What is Part-Of-Speech Tagging?

Part-of-speech tagging is labeling the words in a text as corresponding to a particular part of speech. POS tagging, which is also called word-category disambiguation or grammatical tagging, is based on both the definition and the context i.e. the relationship of the word with adjacent and related words in the phrase, sentence or paragraph that it exists in. A simplified version of POS tagging, is the identification of words as nouns, verbs, adjectives etc. in a sentence, taught to school-age children. [1]

POS Tagging can be regarded as a simplified form of morphological analysis. However, it only deals with assigning an appropriate POS tag to the word, whereas morphological analysis deals with finding the internal structure of the word.

1.2 Why do we need Part-Of-Speech Tagging?

Automatic text tagging is an important concept in natural language processing. There are several different stages of natural language processing, where the preceeding stage feeds the next one. Morphological analysis, based on phonemes and morphemes, is the first stage generally.The next stage is the analysis of semantics. Part-of-speech tagging is one of the earliest steps within this sequence.

1.3 A Brief Overview of Part-Of-Speech Tagging

1.3.1 Parts-of–Speech Used Commonly

The linguists claim that there are three major parts-of-speech: noun, verb and adjective. In the case of lexical categorization, additional parts-of-speech are proposed such as adposition and determiner, which have secondary importance. These additional categories may have subcategories, which reflect either morphosyntactic properties such as tense and number, or semantic properties such as for nouns, count and mass noun.

While constructing the tag set, the major consideration is providing a distinct part-of-speech for each class of word that have distinct grammatical behaviour, though the size and contents of the tag set are oriented linguistically.

1.3.2 Problems of Part-Of-Speech Tagging

There are two major problems of POS tagging: Ambiguous words and unknown words. The first problem is the existence of words for which more than one tag is possible. The solution of the problem is taking the context into consideration rather than single words.This is a trivial task for humans but not so easy for automatic text taggers. Below is an example sentence, which have different possible tags for one word:

We can can the can. [2]

In the above example, 'can' corresponds to auxiliary verb, verb and noun respectively.

The latter problem is the occurrence of unknown words, i.e. words that do not exist in the corpus. Thus, it is a significant design issue to provide mechanisms for handling unknown words.

1.3.3 Part-Of-Speech Tagging Approaches

1.3.3.1 Rule Based Approaches

The earliest part-of-speech tagging systems use the rule based approach, which is based on a manually constructed set of rules.These rules are then, applied to the given text. The necessity of a linguistic background and manually constructing the rules are the main drawbacks of the rule based systems.

1.3.3.2 Transformation Based Learning (TBL)

A system which learns a set of correction rules is described by Brill, in order to avoid linguistic rules manually.[3] The idea which the system is based on is to assign an initial tag to each word in the corpus. After the initialization process, by using a predetermined rule template, a set of rules is obtained by instantiating each rule template with data from the corpus. Each rule is applied temporarily to words that are tagged incorrectly and the best rule which reduces the most number of errors, is identified.

The rule is added to the learned rules and the process iterates on the new corpus formed by applying the newly added rule, until reducing the error rate less than a predetermined threshold is not possible by applying the remaining rules.

1.3.3.3 Markov Model Approaches

As a consequence of having rich variety of data sources such as lexicons which may include frequency data, large corpora and bilingual parallel corpora, we can benefit from statistical methods, learn the patterns of tag sequences to use in tagging new sentences. Hidden Markov Model (HMM) is a popular statistical tagging method which makes use of Markov Model and a simple smoothing technique.

1.3.3.4 Maximum Entropy Approaches

Maximum entropy approach provides us more flexibility with the context , which is used poorly in HMM framework.The usage of context is similar to TBL approach in the way that a set of feature templates is gathered in analogy to the set of rule templates in TBL.These feature templates are predefined and by instantiating the feature templates with the data from the corpus, differentiating features are learned by the system.The flexibility is the result of adding any feature template that is regarded as useful.

1.3.3.5 Support Vector Machines

Support vector machines have two advantages over other models: They can easily handle high dimensional spaces i.e. large number of features and they are usually more resistant to overfitting. The reason, why we work with Support Vector Machines will be clear in later sections.

1.4 Previous Work on Part of Speech Tagging in Turkish

Morphological disambiguation is very crucial in aggluginative languages like Turkish. A POS tagger is developed for Turkish based on a two level specification of Turkish morphology, which makes use of a lexicon of 24,000 words.The tagger is enhanced with the usage of a multiword and idiomatic construct recognizer and morphological disambiguator, which benefits from the neighbourhood constraints, heuristics and statistical data.The tagger reaches to an accuracy level of 98 - 99 per cent.[4]

Another tagger developed for tagging Turkish text is based on a composite approach which combines the rule based and statistical approaches and makes use of some characteristics of the language regarding heuristics. Both word frequencies and n-gram (unigram, bigram and trigram) probabilities are used. In order to enhance the accuracy of the system, the output of a morphological analyzer with stochastic methods are combined. The tagger reached an accuracy level of 80 per cent. [5]

1.5 What is A Support Vector Machine?

Support vector machines (SVMs) are a set of related supervised learning methods used for classification and regression. Input data is viewed as two sets of vectors in an n-dimensional space as shown as black and white dots in the Figure 1. An SVM constructs a separating hyperplane in that space, one which maximizes the margin between the two data sets. To calculate the margin, two parallel hyperplanes are constructed, one on each side of the separating

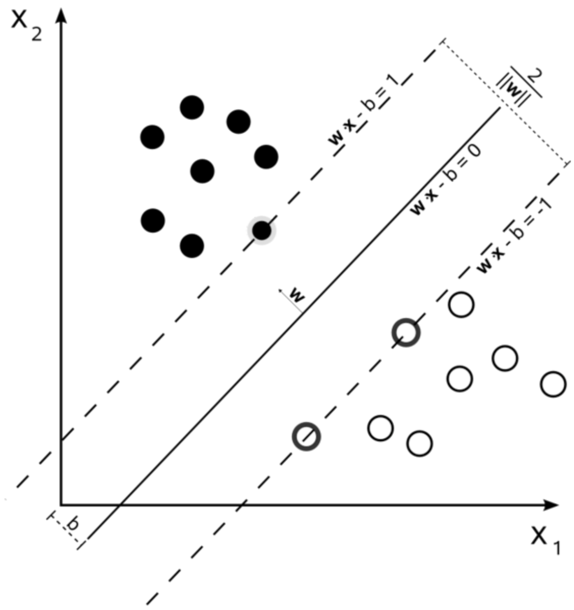

Figure 1. Maximum margin hyperplane

hyperplane, which are "pushed up against" the two data sets. A hyperplane is desired, which has the largest distance to the neighboring datapoints of both classes, since in general the larger the margin, the lower the generalization error of the classifier. [6]

1.6 Why did we prefer SVM Approach?

As we mentioned above, several studies are made on part-of-speech Tagging in Turkish but none of them were based on Support Vector Machines. In addition, support vector machine approach is very flexible in terms of providing a large feature set which includes both rule-based and statistical data. Since, morphemes in agglutinative languages have the capability to change the class of a word, they should be taken into account. Thus, we preferred to use some rules for morphemes and also some other rules, which deal with properties of Turkish words. Another advantage of SVMs is that no problem regarding unknown words can occur. All words are unknown in a sense, since it is not significant how many times a word occurs with a specific tag in the corpus. Both manually written rules and statistical data contributed to building our feature set.

2. A New Approach to POS Tagging in Turkish

2.1 Overview

2.1.1 Description

The project's aim is to develop methods for Part of Speech Tagging in Turkish by making use of both statistical data and manually written rules.

2.1.2 Input

The corpus contains 753.248 words. 90 per cent is used for training and 10 per cent is used for testing purposes. The format of the corpus is explained in Resources section.

2.1.2.1 Restrictions

In other studies about POS tagging that we have examined, a larger corpus is used, such as Penn Treebank corpus for English which consists over 4.5 million words. Larger corpus is better for training.

In addition to the corpus size, another restriction is the dependence to an external tool, SVMlight Multiclass.Hence the accuracy of the study is directly affected by the performance of SVM tool.

2.1.3 Output

A text file named output.txt is created which shows tag predictions and values for each class for each word. Format of this file is shown below:

```
1 0.046902 -0.052224 0.004939 0.000384….
2 -0.029545 0.027581 0.000934 0.001031….
1 0.092526 -0.095487 0.002418 0.000543….
4 0.005377 -0.044700 0.001667 0.037656….
```

Figure 2: Format of the output file

Here the first number denotes the predicted tag and the rest of the numbers denote the probabilities for each tag.The tag with the highest probability value is assigned to the word.

In the command window of SVMLight, we can also see success rate and the number of correctly and uncorrectly tagged words.

2.1.4 Resources

A pretagged corpus is used, whose format is as shown in the Figure 3.

<S>+BSTag

İki iki[Adj] iki[Noun]+[A3sg]+[Pnon]+[Nom]
senaryonun senaryo[Noun]+[A3sg]+[Pnon]+NHn[Gen] senaryo[Noun]+[A3sg]+Hn[P2sg]+NHn[Gen]
da da[Conj]
, ,[Punc]
beyin beyin[Noun]+[A3sg]+[Pnon]+[Nom] Bey[Noun]+[Prop]+[A3sg]+Hn[P2sg]+[Nom] bey[Noun]+[A3sg]+[Pnon]+NHn[Gen] bey[Noun]+[A3sg]+Hn[P2sg]+[Nom]
cimnastiği cimnastiği[Unknown]
uğruna uğur(II)[Noun]+[A3sg]+SH[P3sg]+NA[Dat] uğur(II)[Noun]+[A3sg]+Hn[P2sg]+NA[Dat] uğru[Noun]+[A3sg]+Hn[P2sg]+NA[Dat]
diğer diğer[Adj]
şartlar şart[Noun]+lAr[A3pl]+[Pnon]+[Nom] şartla[Verb]+[Pos]+Hr[Aor]+[A3sg]
eşit eşit[Adj] Eşit[Noun]+[Prop]+[A3sg]+[Pnon]+[Nom]
varsayımı varsayım[Noun]+[A3sg]+SH[P3sg]+[Nom] varsayım[Noun]+[A3sg]+[Pnon]+YH[Acc]
altında alt[Noun]+[A3sg]+SH[P3sg]+NDA[Loc] alt[Noun]+[A3sg]+Hn[P2sg]+NDA[Loc] alt[Adj]-[Noun]+[A3sg]+SH[P3sg]+NDA[Loc] alt[Adj]-[Noun]+[A3sg]+Hn[P2sg]+NDA[Loc] altı[Noun]+[A3sg]+Hn[P2sg]+NDA[Loc] altın[Adj]-[Noun]+[A3sg]+[Pnon]+DA[Loc] altı[Adj]-[Noun]+[A3sg]+Hn[P2sg]+NDA[Loc] altın[Noun]+[A3sg]+[Pnon]+DA[Loc]
çizilen çiz[Verb]-Hl[Verb+Pass]+[Pos]-YAn[Adj+PresPart] çizi[Noun]+[A3sg]+[Pnon]+[Nom]-lAn[Verb+Acquire]+[Pos]+[Imp]+[A2sg]
akışı ak[Verb]+[Pos]-YHş[Noun+Inf3]+[A3sg]+[Pnon]+YH[Acc] ak[Verb]+[Pos]-YHş[Noun+Inf3]+[A3sg]+SH[P3sg]+[Nom] akış[Noun]+[A3sg]+[Pnon]+YH[Acc]
…

</S> </S>+ESTag

Figure 3: A sample sentence from the pretagged corpus

BSTag and ESTag denote beginning and ending of a sentence respectively. Bold words shown in the figure are the words in the sentence. There may be more than one parses for one word, which are written to the right of the word. Tag of a word can be obtained from the first tag or from a tag of an inflectional suffix. Tags are explained in the section 2.2.1.1.

2.1.5 Process

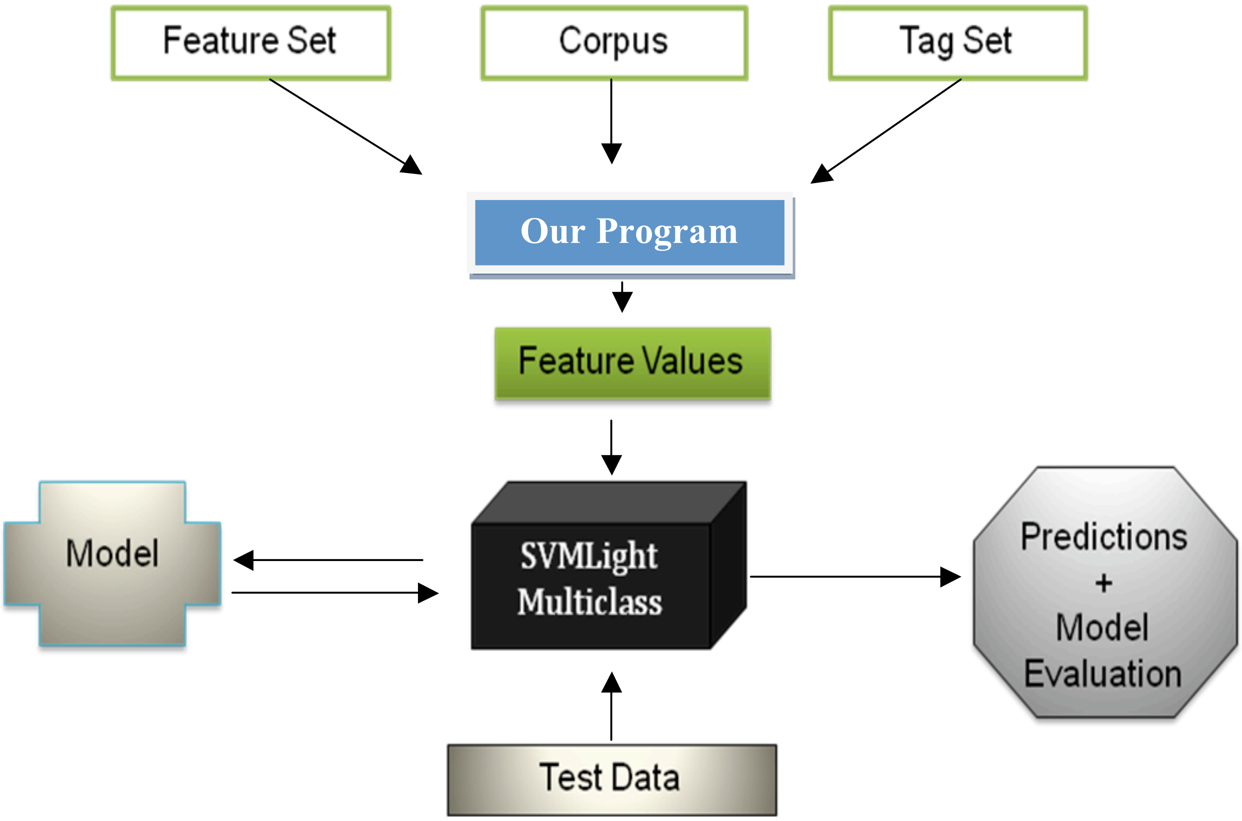

Figure 4: Overall picture of the process

2.2 Methodology

2.2.1 High Level Description

POS tagger consists of different modules each serving different purposes. Below, details of these modules can be found. But first resources that they modules make use of will be explained.

2.2.1.1 Tag Set

From the corpus a tag set is gathered, which contains the following 14 tags: Noun, Conj, Postp, Verb, Adj, Punc, Det, Adv,Pron, Num, Unknown, Interj, Dup,Ques. These abbreviations stand for Noun, Conjunction, Postposition, Verb, Adjective, Punctuation Mark, Determiner, Adverb, Pronoun, Numeral, Unknown, Interjection, Duplicate and Question respectively.

To gather the tags, a program is written to take each tag in the corpus. Then we have eliminated those tags to 13. Later we have noticed that there is a tag named “Unknown” in the corpus. We have left this tag as it is and added a new tag with the same name to the tag set. Later in the evaluation part, we have not taken words having ‘unknown’ tag into account.

2.2.1.2 Feature Set

A feature set is created by examining the corpus and the properties of different lexical categories carefully.

Word is one of the maximum words	w
Word bigrams	(w-1, w)
Word trigrams	(w-2, w-1, w)
Next word is one of the maximum words	w+1
Next word marks end of sentence	Being last word
Previous word marks end of sentence	Being first word
Previous word is equal to the current word	For duplicates
POS tag of the previous word	p-1
Binary word features	Initial capital Contains apostrophe Contains hyphen Contains number Length is greater than 15 Length is equal to one and word is not a number Word equal to a punctuation mark Word equal to conjunction ("da") Word equal to question ("starts with mu, mü etc.")
Suffixes	Defined in the Appendix A.

Table 1: Feature Set Used in the Study

2.2.1.3 Modules

Statistical Data Gathering

The role of this part is collecting statistical data out of the corpus. First, corpus is transformed into a file with words and their tags, so that processing will be easier. Splitting of this file into two files is done to create input files for training and testing programs. Later, statistics are collected from the newly created file. Among these statistics we can name finding maximum words, pairs, triples; maximum occurring last two or three letters of some grammatical classes and other useful information that will be helpful in successful tagging of words.

Creating input file for SVMLight

The corpus is splitted into two parts , 90 % for training data and the rest 10 % for the test data. Both training and testing programs create input files for SVMLight. Codes for these files are very similar.

Firstly, the feature values are calculated for each word in the corpus,by examining if the current feature is hold for the current word.If the feature holds for the word, the value is written to the output file. If the feature does not hold for the word hence the feature value is zero, it is not added to the output file in order not to make the file size grow excessively.

Next, the same process is repeated for the test data , feature values are calculated and written into an output file.

2.2.2 Middle Level Description

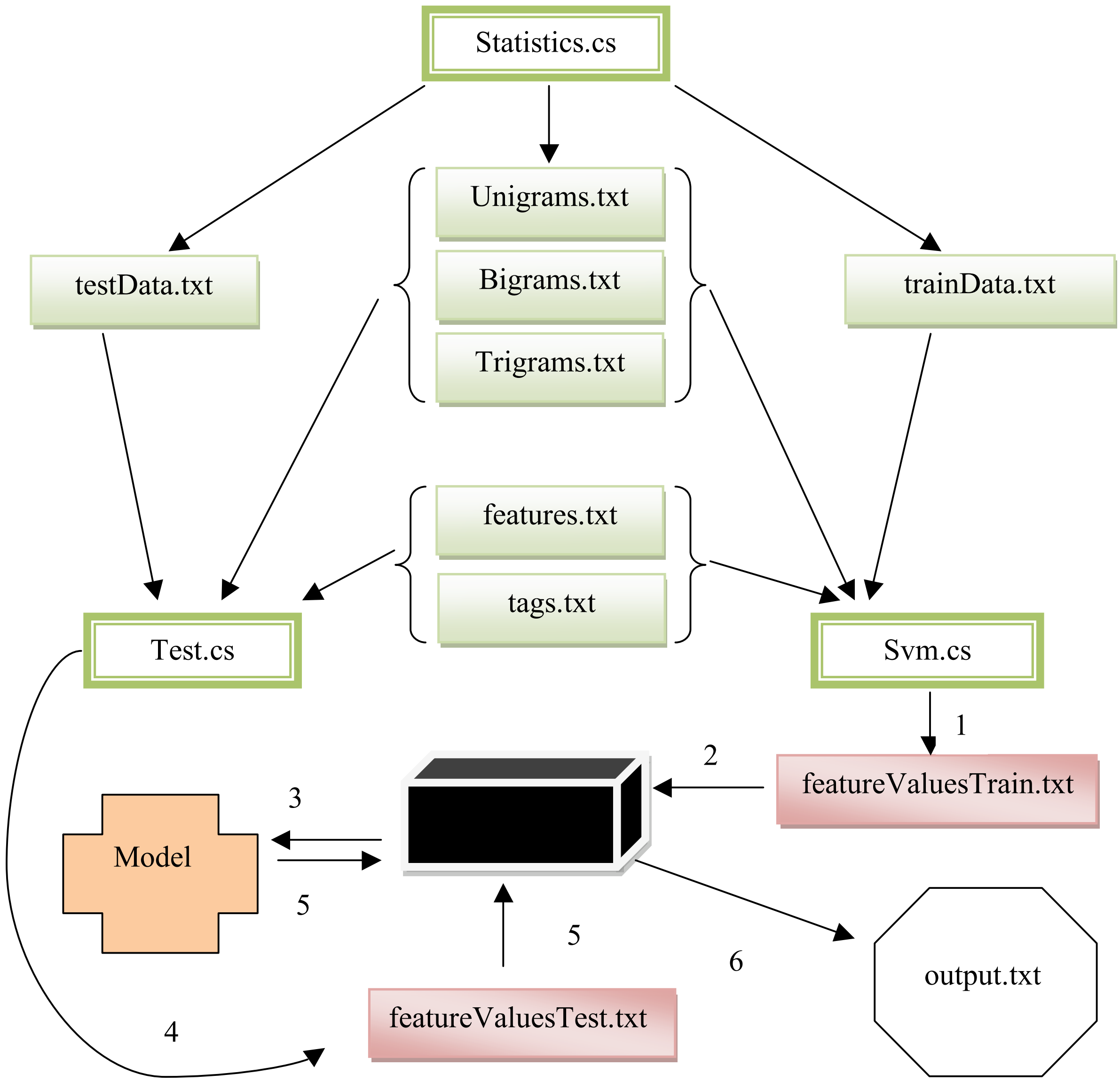

Figure 5: Detailed picture of implementation

1: Feature values for train data are calculated by running SVM.cs and featureValuesTrain.txt file is obtained as output.

2: featureValuesTrain.txt file is given to SVMlight Multiclass as input and ‘learn’ command which lets the tool to generate a model is entered from the command line.

3: A model is created by SVM tool which does not need to be readable and will be given to the tool as input when making predictions for the test data.

4:featureValuesTest.txt file is obtained by running the Test.cs file and feature values for test data are calculated.At this step, Model is given as input to the Test.cs in order to use the POS tag predicted for the previous word as a feature for the current word.

5: featureValuesTest.txt and Model are given as input to the SVM tool and ‘classify’ command is entered from the command line in order to predict the tag classes for the test data .

6: the tag classes of test data are predicted and written to output.txt by the SVM tool using the previously created model.The given classes for the words and the predicted ones are compared, the success rate is returned as a result.

2.2.3 Low Level Description

The code for the POS tagger is written in C# programming language using Microsoft .NET environment. Details of the classes are explained below. As mentioned before, the main focus is creating an input file for SVMLight. Let us first examine the format of this input file.

2.2.3.1 Input file for SVM^{light} Multiclass

Format of the input file is as follows:

```
<class id> <feature id>:<feature value> <feature id>:<feature value>…
<class id> <feature id>:<feature value> <feature id>:<feature value>…
```

Figure 6: Format of the input file for SVMLight Multiclass tool

Here class id corresponds to one of the tags in the training data. (To learn about the tags, please refer to Section 2.2.1.1.)

Below is a sample sentence and the feature values of the sentence written in the input file.

Figure 7: A sample sentence with feature values calculated

This sample from the corpus we have used, shows two blocks, i.e feature value evaluations for six words. In this example,there are only limited number of features for each word for demonstration purposes.In our case there are about 56000 features that is dependent to the size of the corpus due to the n-gram properties.

2.2.3.2 Code classes

<u>Tag.cs</u>

A tag structure that stores tag numbers(ids) and tag names is created. Tags are read from the file and stored in this structure for further use. Methods are called by Svm.cs.

<u>Statistics.cs</u>

In this code file, the main job is to gather statistical data from the corpus. Before gathering these data, corpus is read and written into a new file called wordsWithTags.txt, where all words are written with their tags, one word-tag pair per line. This file is as follows:

```
Refah Noun
da Conj
Türkçe Noun
için Postp
görüş Noun
istedi Verb
Öztürkçe Noun
çalışmalarıyla Noun
…
```

Figure 8: An example part of the wordsWithTags.txt file

Later this file is splitted into two parts, one for training and one for testing purposes,trainData.txt and testData.txt respectively. For creation of this wordsWithTags.txt file, words and their tags are taken from the corpus. Whenever there is a tag in the corpus, that marks the ending of a sentence, a point is put to this file for marking the end.

A word in the corpus has the following structure:

```
word root[tag ] + suffix[tag1] + suffix[tag2+tag3] + ..
```

Figure 9: Format of a word and its parsed structure in the corpus

Since there are usually more than one tags for a word, the correct one should be taken. This is done as follows: If a word has an inflectional suffix, i.e. a suffix that changes the grammatical category of the word, then the resulting tag is taken. If it does not possess such a suffix, the first tag is taken.

The file for testing purposes is named as testData.txt and given as input to Test.cs. Its format is the same as the input file for Svm.cs.

Other than creating input files, this code file has an important job in collecting statistical data from the corpus. These statistics are listed below:

1. **Finding most occurring words, pair of words and triples of words**

   These data are used for word unigram, bigram and trigram purposes. Threshold values are varied and tested experimentally. These values are 300, 100 and 2 respectively.

2. **Finding most occurring last 2 or 3 letters of words based on their classes (verb, adjective etc.)**

   The aim in gathering these letters is determining new suffixes that will be used for tagging some classes. Based on the occurence of these suffixes, they are taken into account.

3. **Getting percentages of word classes in the corpus**

   This is done to get an idea about the corpus. The distribution can be observed in the chart below.

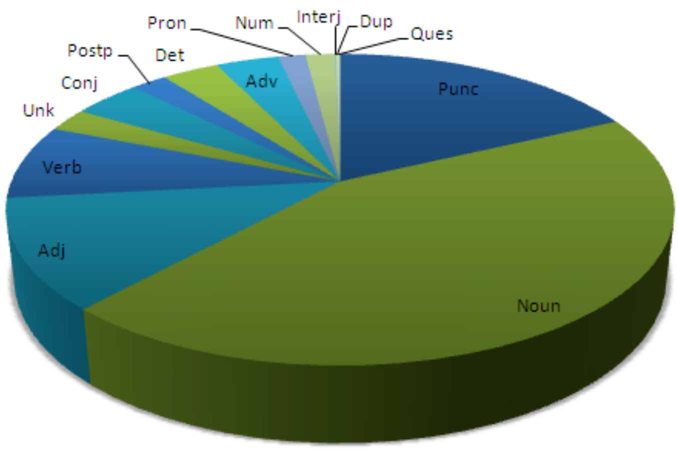

Figure 10: Distribution of grammatical classes in the corpus

4. **Gathering data about some suffixes, i.e. how successful they are in tagging some specific grammatical class**

   This is extensively used to measure performance of the suffixes.

5. **Writing words of some classes to files**

   To observe the similarity of words of a class in terms of suffixes and unary word features.

<u>Svm.cs</u>

This is the main part for creating input file for SVMLight software. In this file, trainData.txt is used, which was created by Statistics.cs.

In each step of the execution, a word is read from this file and feature operations are executed on it. Some operations make use of files that contain statistical data, whereas others are operations on one word for suffixes and other unary features.While analyzing each word for containing the given suffixes, for shorter suffixes containing less than 3 letters, only the end of the word is taken into account whereas for longer suffixes the 2 / 3 rd of the word is examined for containing the current suffix.

During operation, number of word that is currently being processed is outputted so that it can be guessed approximately when execution will end.
As a result, a file named featureValuesTrain.txt is created, which is input file for model training of SVMLight.

Test.cs

This code file is for creating an input file for SVMLight Multiclass for testing purposes. It is very similar to Svm.cs, but differs from it in terms of file names and some feature operations. An important difference is POS tag of the previous word.

In Svm.cs this operation is done easily by taking the POS tag of the previous word. But for testing, this tag should be found by SVMLight first. So, at first svm_multiclass_learn command is executed with the input file created by Svm.cs. Then, by using model file, svm_multiclass_classify is called from within the code of Test.cs to assign a tag for the word. This tag is written onto a file by SVMlight software. Now this tag can be used as the previous POS tag for the next word.

Another important part of the code is evaluation of the success. Although success is evaluated by SVMLight, unknown tagged words are also taken into account. In addition, it is good to have the success rate for each grammatical class, so that we could know which sides to improve.

3. RESULTS AND CONCLUSION

The project aimed at assigning correct part of speech tags to words in a Turkish text by using support vector machines approach making use of SVMLight tool.
Success level reached at the end of different experiments is 75 per cent.

Since Turkish is an agglutinative language, success levels of POS taggers of Turkish and not agglutinative languages such as English should not be compared. But we will also give examples of success levels of such languages. Several studies on part-of-speech tagging using support vector machine approach that are carried out on English have an accuracy level above 90 per cent. As an example, the study by Jesus Gimenez, and Lluis Marquez has pointed out an accuracy of 97.16 per cent. [7]

Another study carried out by Tetsuji Nakagawa, Taku Kudoh and Yuji Matsumoto has an accuracy level of 97.1 per cent.[8]

There are some studies on part of speech tagging in Turkish, completed previously. However, neither of them are based on using support vector machines. A study which has been completed by Levent Altınyurt, Zihni Orhan and Tunga Göngör has an accuracy level about 80 per cent. [9]

By looking at the Table 2, we can interpret the success rate of tagging each grammatical class:

Class	**Tagging Success**
Noun	96,91
Conj	74,13
Postp	1,37
Verb	66,37
Adj	5,65
Punc	100
Det	71,57
Adv	12,11
Pron	4,3
Num	67,74
Unknown	-
Interj	0
Dup	0
Ques	21,95

Table 2: Success rates of tagging different classes

From this table we can see that some classes are tagged successfully whereas some are not. Lack of tagging success shows the lack of successful feature selection for that class. For example, pronouns usually do not possess distinctive suffixes. It is difficult to find features that will distinguish pronouns. Suffixes desired for using for adjectives are common in other classes so that these features are not selected, since they will cause erroneus results.

We believe that success can be improved by taking following issues into account:

- Larger corpus size will be better in training and model building,
- Including new distinguishing features,
- Analyzing the corpus with the features by reading from right to left in order to use the pos tag of next words.

4.REFERENCES

1. Wikipedia – http://en.wikipedia.org/wiki/Part-of-speech_tagging

2. Tunga Güngör, "Part-of-Speech Tagging", 2009

3. Eric Brill "Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part of Speech Tagging.", Computational Linguistics,1995

4. Kemal Oflazer ,İlker Kuruöz, "Tagging and Morphological Disambiguation of Turkish Text"

5. Levent Altınyurt, Zihni Orhan,Tunga Güngör, " Towards combining rule-based and statistical part of speech tagging in agglutinative languages",2007

6. Support Vector Machine - http://en.wikipedia.org/wiki/Support_vector_machine

7. Gimenez, D. and L. Marquez, "SVMTool: A general POS tagger generator based on Support Vector Machines"

8. Tetsuji Nakagawa, Taku Kudoh and Yuji Matsumoto , "Unknown Word Guessing and Part-of-Speech Tagging Using Support Vector Machines"

9. Levent Altınyurt, Zihni Orhan,Tunga Güngör, "A Composite Approach For Part Of Speech Tagging In Turkish", 2007

REFERENCES NOT CITED

Gimenez J, Marquez L. Fast and Accurate part-of-Speech Tagging: The SVM Approach Revisited , 2003

Poel M, Stegeman L, Akker R. A Support Vector Machine Approach to Dutch Part-of-Speech Tagging ,2007

Pla, Molina. Part-of-Speech Tagging with Lexicalized HMM, 2001

Brill, A Simple Rule-Based Part of Speech Tagger ,1992

James Mayfield, Paul McNamee, Christine Piatko, Claudia Pearce , Lattice-based Tagging using Support Vector Machines, Conference on Information and Knowledge Management, 2003

Jason Weston, Chris Watkins, Multi-class Support Vector Machines, 1998

Ethem Alpaydın, Support Vector Machines, Introduction To Machine Learning, 2004

Taku Kudo, Yuji Matsumoto , Chunking with support vector machines, North American Chapter Of The Association For Computational Linguistics, 2001

Vapnik, V., "The Nature of Statistical Learning Theory", Springer, 1995.

SVMLight Support Vector Machine - http://svmlight.joachims.org/

SVMLight Multi-Class Support Vector Machine - http://svmlight.joachims.org/svm_multiclass.html

Kemal Oflazer ,İlker Kuruöz, "Tagging and Morphological Disambiguation of Turkish Text"

5. APPENDIX : Important parts of the code and suffixes

<u>Suffixes:</u> (Suffixes in the same row are substitutes)

ağı eği

am em

ama eme

amak emek

arak,erek,yip,yıp

ası esi

av ev

bilir

deki daki teki

dam dem tem

dız diz düz

dik dık duk

dir dır tir dur tur tır

dı du dü ti tü di tı tu

memiş mamış acaktı ecekti

gı gi gü kı ku

gıç giç

ir

ince,ınca,

inci,üncü, ıncı

eni

ıp

gun gın

ki

li lı

lak lek

lan len

landı lendi

lama leme

lar ler

lararası

lık luk lük

mış miş müş muş mıştı mişti müştü muştu

mişti muştu müştü mıştı

medi madı yordu yorlar

düler dular diler dılar

mişler mışlar müşler muşlar

sek

se

sız siz suz süz

cü çı cu çi cı ci çu

daş

inci üncü ncü

cil cıl

şın

sal sel

ıt

kır

acak ecek

ge

ıcı ücü

ın ün in

ünç inç

inti ıntı üntü

im üm ım um sin sün sın sun

siniz sünüz sunuz sınız dirler dırlar durlar

yor melidir malıdır

mek mak

ik uk

syon siyon

iyet

kar

dan

tar

madan meden

maksızın meksizin

mala mele

malı

mar mer

mazlık mezlik

mız miz muz

nak nek

nın nin nun nün

ntı nti ntu ntü

rak rek

rga rge

rsa rse zsa zse

rlar

sa se

sil sul

sın sun sün

şar

tay

ntı nti ntu ntü

van ven

gan gen kan

ken

gün kın

kün

ıcı ici ucu ücü

vi

men man

duğu

le la

ğini

lik luk

lanma lenme

ns

yan yen

nan

nel zel

ruz

ıca

ren

rek rak

'lı 'li 'lu 'lü

Şan

Ük

Code (Only the important parts)

Statistics.cs

```
public struct bigram
   {
      public string word;
      public string prevWord  }
   public struct trigram
   {
      public string word;
      public string prevWord;
      public string twoPrevWord;}
   public struct Word
   {
      public string word;
      public string tag;
   };
```

```
public void writeWordsToFile()
{
    dictUnigram = new Dictionary<string, int>();
    dictUnigramThreshold = new Dictionary<string, int>();
    dictBigram = new Dictionary<bigram, int>();
    dictBigramThreshold = new Dictionary<bigram, int>();
    dictTrigram = new Dictionary<trigram, int>();
    dictTrigramThreshold = new Dictionary<trigram, int>();

    bigram newBigram = new bigram();
    trigram newTrigram = new trigram();

    StreamReader sr = File.OpenText(corpus);
    StreamWriter sw = File.CreateText("wordsWithTags10.txt");
    string line;
    string[] tags = new string[20];
    string word = String.Empty;
    string tag = String.Empty;

    string[] WordsInLine = new string[20];
    string[] splitMatch = new string[2];
    string[] parse = new string[13];
    string[] splitted = new string[2];

    for (int i = 1; i <= 3; i++)
    {
    sr = File.OpenText(corpus);
    line = sr.ReadLine();
    wordCount = 1;
    while (!String.IsNullOrEmpty(line))
    {

        if (line.StartsWith("<") && line.Contains("ESTag"))
```

```
{
    sw.WriteLine("." + " " + "Punc");
}
else if (!line.StartsWith("<"))
{
  WordsInLine = line.Split(' ');
  word = WordsInLine[0];

  if (!WordsInLine[1].StartsWith("+"))
  {
    string input = WordsInLine[1];
    string pattern = "]";
    string[] substrings = Regex.Split(input, pattern);

    int index = 0;

    for (int m = 0; m < substrings.Length; m++)
    {
      if (!substrings[m].Equals(String.Empty))
      {
        string[] splittedNew = substrings[m].Split('[');
        tags[index] = splittedNew[1];
        index++;
      }
    }

    int control = -1;
    int j = 0;
    int tagIndex = 0;

    for (j = 0; j < index; j++)
    {
      if (!tags[j].Equals(String.Empty))
```

```
    {
        if (tags[j].Contains("+"))
        {
            tagIndex = j;
            control = tagIndex;
        }
    }
}
if (control == -1)
    tag = tags[0];
else
    tag = tags[tagIndex].Split('+')[0];

   sw.WriteLine(word + " " + tag);
if (i == 1)
{
    if (dictUnigram.ContainsKey(word.ToLower()))
    {
        dictUnigram[word.ToLower()]++;

        if (dictUnigram[word.ToLower()] == wordThreshold)
        {
            dictUnigramThreshold.Add(word.ToLower(), wordThreshold);
            dictUnigram.Remove(word.ToLower());
        }

    }
    else if (dictUnigramThreshold.ContainsKey(word.ToLower()))
    {
        dictUnigramThreshold[word.ToLower()]++;
    }
    else
    {
```

```
                dictUnigram.Add(word.ToLower(), 1);
            } } } }
if (i == 2)
{
    newBigram.prevWord = newBigram.word;
    newBigram.word = word.ToLower();

    if (dictBigram.ContainsKey(newBigram))
    {
        dictBigram[newBigram]++;
        if (dictBigram[newBigram] == pairThreshold)
        {
            dictBigramThreshold.Add(newBigram, pairThreshold);
            dictBigram.Remove(newBigram);
        }

    }
    else if (dictBigramThreshold.ContainsKey(newBigram))
    {
        dictBigramThreshold[newBigram]++;
    }
    else if (!String.IsNullOrEmpty(newBigram.prevWord))
    {
        dictBigram.Add(newBigram, 1);
    }
}

line = sr.ReadLine();
wordCount++;
if (!String.IsNullOrEmpty(line))
{
    while (!String.IsNullOrEmpty(line) && line.StartsWith("<"))
    {
```

```
            if (line.Contains("ESTag"))
            {
               if (i == 1)
                sw.WriteLine("." + " " + "Punc");
            }
            line = sr.ReadLine();
         }
         if (i == 3 && !String.IsNullOrEmpty(line))
         {

            WordsInLine = line.Split(' ');
            word = WordsInLine[0];
            newTrigram.twoPrevWord = newTrigram.prevWord;
            newTrigram.prevWord = newTrigram.word;
            newTrigram.word = word.ToLower();

            if (dictTrigram.ContainsKey(newTrigram))
            {
               dictTrigram[newTrigram]++;
               if (dictTrigram[newTrigram] == tripleThreshold)
               {
                  dictTrigramThreshold.Add(newTrigram, tripleThreshold);
               }


            }
            else if (dictTrigramThreshold.ContainsKey(newTrigram))
            {
               dictTrigramThreshold[newTrigram]++;
            }
            else if (!String.IsNullOrEmpty(newTrigram.prevWord) &&
!String.IsNullOrEmpty(newTrigram.twoPrevWord))
            {
               dictTrigram.Add(newTrigram, 1);
```

```
                }
            }
        }
        Console.WriteLine(wordCount);
    }

    if (i == 1)
    {
        getUnigrams();
        dictUnigram.Clear();
        dictUnigramThreshold.Clear();
    }

    else if (i == 2)
    {
        getBigrams();
        dictBigram.Clear();
        dictBigramThreshold.Clear();
    }
    else if (i == 3)
    {
        getTrigrams();
        dictTrigram.Clear();
        dictTrigramThreshold.Clear();
    }
    sr.Close();
    }
    sw.Close();

}

public void splitCorpus()
  {
```

```
StreamReader sr = File.OpenText("corpus.txt");
StreamReader sr2 = File.OpenText("wordsWithTags.txt");
string line = sr.ReadLine();
string line2 = sr2.ReadLine();
int count = 1;
StreamWriter sw = File.CreateText("corpusSplitted.txt");
StreamWriter sw2 = File.CreateText("corpusRemaining.txt");
StreamWriter sw3 = File.CreateText("trainData.txt");
StreamWriter swTest = File.CreateText("testData.txt");

while ((line != null) && (line.Length > 0) && (line2 != null) && (line2.Length > 0)
&& (count <= 753248))
{
    if (count <= 75000)
    {
        if (!line.StartsWith("<") || (!line2.StartsWith("<")))
        {
            swTest.WriteLine(line2);
            sw.WriteLine(line);
        }
    }
    else
    {
        sw2.WriteLine(line);
        sw3.WriteLine(line2);
    }

    line = sr.ReadLine();
    line2 = sr2.ReadLine();
    count++;
}
sw.Close();
sw2.Close();
```

```
        sr.Close();
        sw3.Close();
        sr2.Close();
    }

    public void getUnigrams()
    {

        StreamWriter sw = File.CreateText("Unigrams.txt");
        foreach (KeyValuePair<string, int> kvp in dictUnigramThreshold)
        {
            sw.WriteLine(kvp.Key + " " + kvp.Value);
        }
        sw.Close();
    }
```

Svm.cs

```
public void containsSuffix(string word, string tag)
  {
      StreamWriter sww=null;
      if (tag.Equals("Punc"))
          return;
      StreamReader SR;
      StreamWriter SW;
      string line;
      string[] suffixes = new string[20];
      int count = 1;

      SW = File.CreateText("suffixOut.txt");
      SR = File.OpenText(fileName);
      line = SR.ReadLine();
      while ((line != null) && (line.Length > 0))
```

```
{ int value = 0;
  suffixes = line.Split(' ');
  int i = 0;
  //int temp = 0;
  for (i = 0; i < suffixes.Length; i++)
  {
    if (!suffixes[i].Equals(""))
    {
      if (word.Contains(suffixes[i]))
      {
        sww = File.AppendText("suffixesAndTags.txt");
        sww.WriteLine(suffixes[i] + " " + tag);
        sww.Close();
        value = 1;
       if (suffix.ContainsKey(suffixes[i]))
          suffix[suffixes[i]]++;
        else
          suffix.Add(suffixes[i],1);
      }
      else
      {
        value = 0;
        if (!suffix.ContainsKey(suffixes[i]))
        {
          suffix.Add(suffixes[i], 0);
        } }  } }
  count++;
  SW.WriteLine(value);
  line = SR.ReadLine();
}
SR.Close();
SW.Close();
}
```

```
public bool startsCapital(string word)
{
    if (char.IsUpper(word[0]))
                        return true;
    return false;
}

public bool containsApostrophe(string word)
{
    if (word.Contains("\'") && !word.Equals("\'"))
        return true;
    return false;
}
static void Main(string[] args)
{

    #region STATISTICS
    Statistics stat = new Statistics();
    stat.splitCorpus();
    stat.writeWordsToFile();

    #endregion



    SVMMain pr = new SVMMain("");
    pr.setSuffixes("features.txt");
    int countWords = 0;

    string filename = "tags.txt";
    Tag.setTags(filename);

    //   feature set:
```

```
string[] featureSet = new string[8] { "startsCapital",
"containsApostrophe", "containsNumber", "longerThan15",
"lengthOneAndNotNumber", "equalToPunc", "equalToConj", "equalToQues" };
//to call methods by name
SVMMain p = new SVMMain("");
object[] parameters = new object[1];
bool returnValue = false;
//for writing input file
StreamWriter swFeatureValues = File.CreateText("featureValuesTrain.txt");
//for reading input file
StreamReader srTrain = File.OpenText("trainData.txt");
string line = srTrain.ReadLine();
string[] wordTag = new string[2];
Word twoPrevWord;
Word prevWord;
Word nextWord;
twoPrevWord.word = String.Empty;
twoPrevWord.tag = String.Empty;
prevWord.word = String.Empty;
prevWord.tag = String.Empty;
nextWord.word = String.Empty;
nextWord.tag = String.Empty;
int numFeatures = 0;
Word newWord = new Word();
countWords++;
while (!String.IsNullOrEmpty(line))
{
  twoPrevWord.word = prevWord.word;
  prevWord.word = newWord.word;
  twoPrevWord.tag = prevWord.tag;
  prevWord.tag = newWord.tag;
  wordTag = line.Split(' ');
  newWord.word = wordTag[0];
```

```
newWord.tag = wordTag[1];

//operations:
for (int j = 0; j < 8; j++)
{
    // Get the desired method by name:
    MethodInfo methodInfo = typeof(SVMMain).GetMethod(featureSet[j]);
    // Use the instance to call the method without arguments
    parameters[0] = newWord.word;
    try
    {
        //call a feature method
        returnValue = Convert.ToBoolean(methodInfo.Invoke(p, parameters));
        //write to feature values file:
       //file format:
        //tagID featureID:featureValue featureID:featureValue...
        string value = "0";
       if (returnValue)
            value = "1";
       int tagID = 0;
        if (j == 0)
        {
            for (int k = 0; k < Tag.tagArr.Length; k++)
                if (Tag.tagArr[k].tagName.Equals(newWord.tag))
                {
                    tagID = Tag.tagArr[k].tagID;
                    break;
                }
            if (value == "1")
                swFeatureValues.Write(tagID + " " + (j + 1) + ":" + value + " ");
            else
                swFeatureValues.Write(tagID + " ");
        }
```

```
        else
        {
            if (value == "1")
                swFeatureValues.Write((j + 1) + ":" + value + " ");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}//end for
try
{
 //  check if word contains suffixes
    pr.containsSuffix(newWord.word);
    StreamReader srNew = File.OpenText("suffixOut.txt");
    string newLine = srNew.ReadLine();
    while (newLine != null)
    {
        int valueF = Convert.ToInt32(newLine);
        if (valueF == 1)
            swFeatureValues.Write(numFeatures + ":" + valueF + " ");
        numFeatures++;
        newLine = srNew.ReadLine();
    }
    srNew.Close();
    numFeatures = 4;
}
catch { }
//word unigram
StreamReader srUnigram = File.OpenText("Unigrams.txt");
string line2 = srUnigram.ReadLine();
while (!String.IsNullOrEmpty(line2))
```

```
{
    if (line2.Split(' ')[0].ToLower().Equals(newWord.word.ToLower()))
    {
        swFeatureValues.Write(numFeatures + ":" + 1 + " ");
        numFeatures++;
    }
    line2 = srUnigram.ReadLine();
}
srUnigram.Close();


if (!String.IsNullOrEmpty(prevWord.word))
//word Bigram:
{
    {
        StreamReader srBigram = File.OpenText("Bigrams.txt");
        line2 = srBigram.ReadLine();
        string[] pair = new string[2];

        while (!String.IsNullOrEmpty(line2))
        {
            pair = line2.Split(' ');

            if ((pair[0].Equals(prevWord.word.ToLower())) &&
(pair[1].Equals(newWord.word.ToLower())))
            {
                swFeatureValues.Write(numFeatures + ":" + 1 + " ");
                numFeatures++;
            }

            line2 = srBigram.ReadLine();
        }
        srBigram.Close();
```

```
        }
      }
      if (!String.IsNullOrEmpty(prevWord.word) &&
!String.IsNullOrEmpty(twoPrevWord.word))
      //word Trigram:
      {
        {
          StreamReader srTrigram = File.OpenText("Trigrams.txt");
          line2 = srTrigram.ReadLine();
          string[] triple = new string[3];
          while (!String.IsNullOrEmpty(line2))
          {
            triple = line2.Split(' ');
            if ((triple[0].Equals(twoPrevWord.word.ToLower())) &&
(triple[1].Equals(prevWord.word.ToLower())) &&
(triple[2].Equals(newWord.word.ToLower())))
            {
              swFeatureValues.Write(numFeatures + ":" + 1 + " ");
              numFeatures++;
            }
            line2 = srTrigram.ReadLine();
          }
          srTrigram.Close();
        }
      }
      //duplicate
      if (!String.IsNullOrEmpty(prevWord.word))
      {
        if (newWord.word.ToLower().Equals(prevWord.word.ToLower()))
        {
          swFeatureValues.Write(numFeatures + ":" + 1 + " ");
        }
        numFeatures++;
```

```
            }
            //first word
            if (Convert.ToBoolean(p.firstWord(prevWord.word)))
                swFeatureValues.Write(numFeatures + ":" + 1 + " ");
            numFeatures++;
            // first word
            string first = (Convert.ToInt32(p.firstWord(prevWord.word))).ToString();
            swFeatureValues.Write(numFeatures + ":" + first + " ");
            numFeatures++;
            line = srTrain.ReadLine();
            if (!String.IsNullOrEmpty(line))
            {
                //next word
                nextWord.word = line.Split(' ')[0];
                if (Convert.ToBoolean(p.nextWord(nextWord.word)))
                    swFeatureValues.Write(numFeatures + ":" + 1 + " ");
                numFeatures++;
                if (Convert.ToBoolean(p.lastWord(nextWord.word)))
                    swFeatureValues.Write(numFeatures + ":" + 1 + " ");
                numFeatures++;
            }
            countWords++;
            Console.WriteLine(countWords);
            swFeatureValues.Write(Environment.NewLine);
        }
        swFeatureValues.Close();
        srTrain.Close();
    }
  }
}
```