Part-of-Speech Tagging with Limited Training Corpora

Robert Staubs

Thomas Jefferson High School for Science and Technology

Computer Systems Laboratory

Period 1

June 16, 2005

Abstract

The aim of this project is to analyze the effectiveness of traditional part-of-speech tagging methods with limited training corpora. The corpus used–the Susanne Corpus–is of extremely limited size thus offering less occasion to rely entirely upon tagging patterns gleamed from predigested data. Methods used focus on the comparison of tagging correctness among general and genre-specific training with limited training corpora. It is hypothesized that a transition point where genred data is less useful than general data exists for a certain lack of size, so a determination of this corpus' size in relation to that point is also a goal. Results are analyzed by comparing the system-tagged corpus with a professionally tagged one.

1 Introduction

1.1 Problem

Part-of-speech (POS) tagging is a subfield within corpus linguistics and computational linguistics. POS taggers are designed with the aim of analyzing texts of sample language use–corpora–to determine the syntactic categories of the words or phrases used in the text.

POS tagging serves as a theoretical underpinning to two fields above others: natural language processing and corpus linguistics. It is useful to natural language processing–the interpretation or generation of human language by machines–in that it provides a way of preparing processed texts to be interpreted syntactically. Rarely will it be used in such a directly applicable way (though it was thought at one point that it would be most useful for this capacity) but it serves NLP in that it can hint at effective ways of processing in general. It is also useful in the academic field of corpus linguistics in the statistical analysis of how humans use their language.

The intention of this project is to use and compare various methods of POS tagging using a small amount of statistical training and determine their relationship with said methods on larger corpora.

1.2 Scope

This project aims to try to achieve the best results with a limited training corpus. The Susanne Corpus consists of 64 texts of 2000 words or more drawn from the Brown Corpus' 4 genres: A (press reportage), G (belles lettres, biography, memoirs), J (learned–mainly scientific and technical–writing), and N (adventure and Western fiction). Tagging is always performed on the last document available in a genre. For genre-specific tagging the final document in a genre is tagged with training data from the remaining texts in the genre. The same text is tagged with all texts in the genre including itself for comparison and testing. For general tagging the final document in each genre is tagged in turn with training data from the remaining texts in the corpus. The same text is tagged with all texts in the corpus including itself for comparison and testing.

1.3 Background

Many different methods of POS tagging have been advanced in the past but no attempts give hope of "perfect" tagging at the current stage. Accuracy of around 90% on ambiguous words is typical for most methods in current use, often exceeding that. However, these more successful attempts have all been made with very large, expensive corpora and it must be recalled that the focus of this project is the examination of small corpora. POS taggers cannot at the current time mimic human methods for distinguishing part of speech in language use. Work to get taggers to approach the problem from all the expected human methods––semantic prediction, syntactic prediction, lexical frequency, and syntactical category frequency being the most prominent–have not yet reached full fruition.

1.3.1 Standard Methodology

The Hidden Markov Model (HMM) method of POS tagging is probably the most traditional. HMMs represent an underlying pattern of states with probabilities of transitions from state to state. The underlying pattern can be exposed by using the Viterbi Algorithm. In the context of POS tagging tags are viewed as the underlying pattern and the probabilities gleaned from training corpora are the probabilities used for decision making. Therefore, the method generally requires extensive training on one or more pre-tagged corpora.

Decisions on the part of speech of words or semantic units are made by analyzing by analyzing the probability that one tag would follow another $\left(P_{transition} = \frac{f(i,j)}{f(i)}\right)$ and the probability that a certain word or unit has a certain tag $\left(P_{lexical} = \frac{f(i,w)}{f(i)}\right)$ where *f(i, j)* represents the number of transitions from tag i to tag j in the corpora, *f(i, w)* represents the total number of words w with tag i, and *f(i)* represents the frequency of tag i. Transitions and tags not seen are given a small but non-zero probability. Lexical and transition probabilities are combined for comparison. The HMM method converges on its maximum accuracy, as opposed to some methods (most not usable in this situation) which converge to an accuracy level smaller than one attained earlier. HMMs have a close affinity with neural network methods.

1.3.2 Hidden Markov Model

An HMM consists of the states of the model, the probabilities of transition between the states, the probabilities of outputs for each state, and the observable outputs. Viewing the target corpus as an HMM the states of the model correspond to the tags for each word, the outputs to the words themselves, the transition probabilities to the tag transition probabilities, and the output probabilities to the lexical probabilities. Different algorithms are used for solving an HMM for each of the three basic possibilities for unknowns.

1.3.3 Viterbi Algorithm

The Viterbi Algorithm is the HMM-solving algorithm used for the case where only the internal states are unknown. This represents the tagging problem because what we are trying to find are the tags which form the states of the model. The Viterbi Algorithm finds a path through a the state space which maximizes the probability of each state. The algorithm works in a chaining fashion, building each newly established state off of the previously determined ones.

2 Procedure

2.1 Training

Training data consists of: tags represented in the corpus, words represented in the corpus, transitions represented in the corpus, and the frequency of each. Words and tags are stored in word-based and tag-based sorted arrays of structures. This data forms the basis for the statistical information extracted by taggers for making decisions on a unit's tag.

2.1.1 Training Implementation

Corpus data is stored in text files with each subsequent word on a separate line. Word, base word form, tag, etc. are stored on each line tab-delineated. A data-extractor was created using the C++ programming language. The extractor stores each encountered word in an ordered array of structs. If a struct for that word

already exists, an internal variable representing word-frequency is incremented. The tag associated with that word is added to an internal array or, if the tag is already stored there, its frequency is incremented. A similar process is followed for encountered tags. Each tag is added to an associated struct in an ordered array. Its frequency is incremented if it is encountered more than once. The tag that occurred *before* the added one is added to an array of preceding tags or, if that tag is already present, its frequency is incremented.

2.2 Tagging

Tagging of individual texts was hard-implemented.

2.2.1 Tagging Implementation

After adding all training data to the data set in an ordered manner the program proceeds through the target corpus. The lexical and transitional probabilities are calculated as detailed above, and the tag with the maximum total probability is found. The program in its current form could very easily then output the computer-tagged text but it was considered more useful for it to merely maintain a count of successful taggings, assessing its success as it goes along. This sort of output lent itself better to direct analysis of effectiveness after the completion of testing. The transitional probability used for unencountered transitions was 0.0001. This number was arrived at by a study of the distribution of accuracy for a shortened corpus file based on various probabilities. Several different minima and maxima were observed so the potential for better transitional assignments is acknowledged.

3 Results

Test runs are represented using the shorthand TαSβ where α is the genre of the target text and β is the genre of the training text–x if all other texts, capitalized if a primed test.

3.1 Unprimed, Genred Tagging

The results for POS tagging for individual genres without training on the target are shown in Table 1. The range of accuracy found with these samples is from 63.4-65.0%. The order of result accuracy, from most to least accurate, is: G, A, N, J.

TestID	*Correct*	*Total*	*Correct/Total*	*TrainingDataEntries*
TASa	1603	2466	0.65004055	32208
TGSg	1623	2445	0.66380368	37041
TJSj	1486	2343	0.63422962	35921
TNSn	1540	2377	0.64787547	38189

Table 1: *Results for unprimed, genred tagging.*

3.2 Primed, Genred Tagging

The results for primed tagging based on individual genres are not of particular interest except for the purposes of comparison. The accuracy range for these tests was from 77.9-84.8%. The order of accuracy was: A, G, J, N. These results are summarized in Table 2.

TestID	*Correct*	*Total*	*Correct/Total*	*TrainingDataEntries*
TASA	2091	2466	0.84793187	34696
TGSG	2058	2445	0.84171779	39529
TJSJ	1870	2343	0.79812207	38311
TNSN	1849	2377	0.77787127	40583

Table 2: *Results for primed, genred tagging.*

3.3 General Tagging

Summarized in Table 3, the results for training on all texts of the corpus save the target ranged in accuracy from 61.0-67.2%. The order of accuracy was: N, G, A, J.

TestID	*Correct*	*Total*	*Correct/Total*	*TrainingDataEntries*
TASx	1555	2466	0.63057583	150631
TGSx	1618	2445	0.66175869	150631
TJSx	1429	2343	0.60990184	150729
TNSx	1598	2377	0.67227598	150725

Table 3: *Results for unprimed, general tagging.*

4 Analysis and Conclusions

4.1 Observation on Training Length

No clear pattern can be ascertained for the relationship between amount of training data and tagging accuracy for either genred (Figure 1) or general (Figure 2). Clearly the differences in training text sizes are too small to observe differences. However, it is sure that reduced accuracy comes with reduced amounts of training data from the position of all tests relative to larger-sized corpus tests.

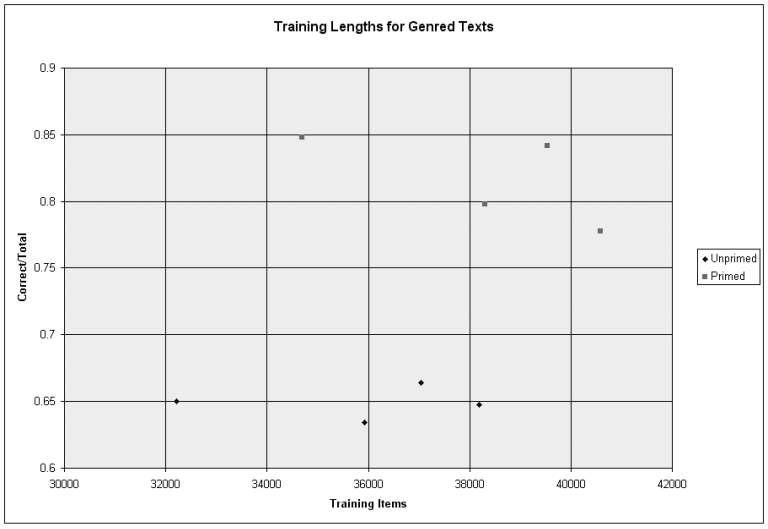

Figure 1: *Accuracy compared with number of training items for genred training.*

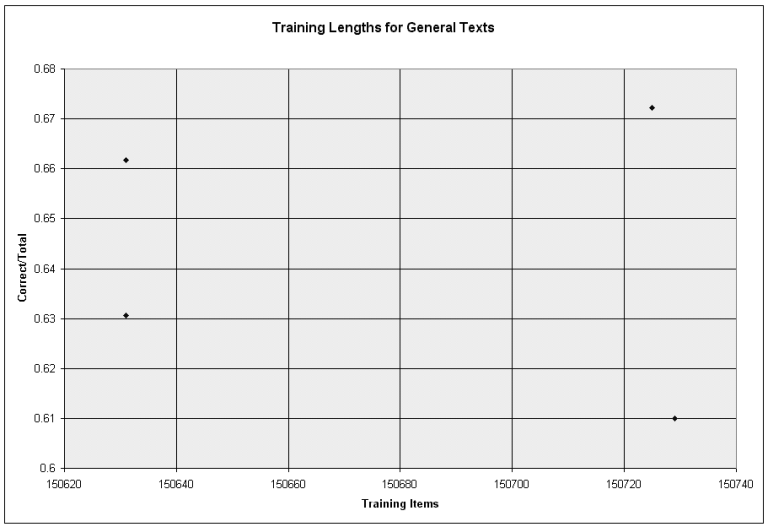

Figure 2: *Accuracy compared with number of training items for general training.*

4.2 Significance of Primed Tests

For all target texts the primed, genred method of tagging was the most effective. This was quite as expected and does not contribute further to our understanding of the subject. However, the large discrepancy between primed and unprimed runs *is* of note. The greatest difference in accuracy between a text tagged with genred training and one tagged with general training amounts to only a few percent while the difference between the two and primed, genred training can be as much as over 20 percent. This indicates that at this small size the corpus does not offer enough generality to fully apply to novel data.

4.3 Comparison of Genred and General Training

From the data no definitive conclusion on the relationship between genred and general training can be drawn. Genred tagging outstrips general by several percentage points for genres A and J and very slightly for genre G. On the other hand, general tagging surpasses genred tagging for genre H by several points. A comparison of all tagging methods can be seen in Figure 3.

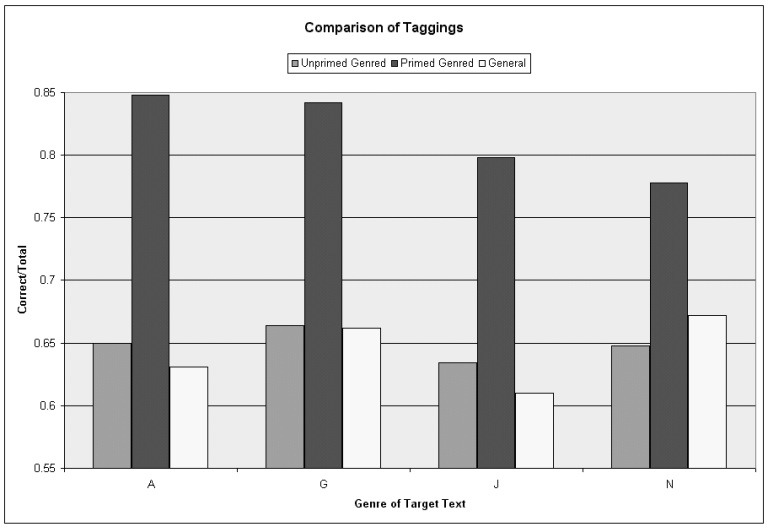

Figure 3: *Comparison of tagging accuracy among different target genres and techniques.*

4.4 Conclusion on Hypothesis

No definitive conclusion on the hypothesized transition point from the supremacy of genred training to that of general training can be drawn. The results for genres G and H may indicate the beginning of a transition to general training supremacy or they may arise simply from abnormalities in those texts. No consistent pattern in differences between accuracy for unprimed, genred training and general training can be seen (Figure 4). At the point surveyed, genred tagging can reservedly still be described as supreme. Transition points could still be possible, but they would most likely exist at a lower information level than that surveyed by this project. Future investigation could thus endeavor to pin down the exact points of transition–if they do indeed exist–and how they vary between different types of tagging and different types of corpus text.

4.5 Limitations

The chief limitations of the algorithm were the paucity of training data provided to it, its speed, and its lack of predefined rules. The data paucity clearly was the object of exploration and deserves no further mention. The algorithm consumed more memory than could be hoped. A faster, more efficient version would be much better for building the algorithm up into a larger and more effective system. Methods for accomplishing

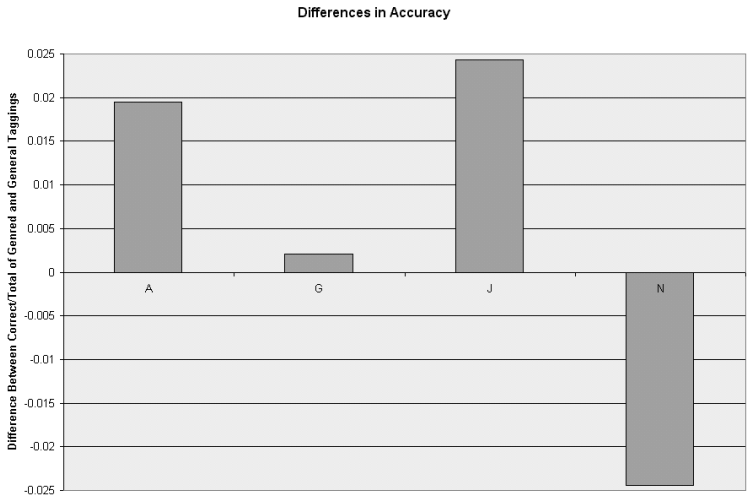

Figure 4: *Differences between accuracy for unprimed, genred training and general training for the four target genres.*

this would probably focus on using storage documents and databases for storing acquired information rather than structures and memory of the algorithm. The algorithm only needs each piece of data occasionally so there is no reason for it to be resident in memory constantly.

The algorithm was not provided with outside knowledge to assist it in making tagging decisions. This was useful for theoretical testing but would realistically never be the case. There are a number of very common, highly ambiguous words that could be made less so by human-designed disambiguation schemes. Extralinguistic clues such as punctuation and capitalization could be more thoroughly exploited with hand-crafted methods. These techniques, already common in many of large-corpus taggers, would be invaluable to a limited-corpus tagger.

The probability assigned to unobserved transition was not determined fully analytically. A survey of available documents was not particularly helpful for corpora of this size so the number was arrived at with limited experimentation with the available models. The numbers pointed to an overall maximum at the number used for the small sample texts but this was by no means an exhaustive series of tests. Future exploration could focus more on the determination of the effects of changing base transition probabilities using thorough computational methods rather than estimation.

4.6 Evaluation of Potential

Limited corpora training clearly cannot be used for most purposes on its own using the standard tagging methods. However, this tagger does not exhibit the full potential of such an algorithm. Because this tagger was designed to work without linguistic foreknowledge it performs rather poorly. Performance could potentially be improved through a number of general descriptions of the more productive morphological rules in English for deriving a word of one POS category from another, as well as with the addition of a number of hyper-frequent words and their POS. An initial limited tagging, if made more efficient, could be used as an initial step in a more extensive tagging algorithm with a larger training corpus. Re-estimation techniques such as Baum-Welsh could potentially be aided by it. Additionally, if a transition point were to be found, tagging below that would form a more effective element in a composite tagger if genred data were not readily available.

5 References

1. D. Elworthy (10/1994). Automatic Error Detection in Part of Speech Tagging.
2. D. Elworthy (10/1994). Does Baum-Welch Re-estimation Help Taggers?
3. M. Maragoudakis, *et al.* Towards a Bayesian Stochastic Part-of-Speech and Case Tagger of Natural Language Corpora
4. M. Marcus, *et al.* Building a large annotated corpus of English: the Penn Treebank
5. G. Marton and B. Katz. Exploring the Role of Part of Speech in the Lexicon
6. T. Nakagawa, *et al.* (2001). Unknown Word Guessing and Part-of-Speech Tagging Using Support Vector Machines
7. V. Savova and L. Pashkin. Part-of-Speech Tagging with Minimal Lexicalization
8. K. Toutanova, *et al.* Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network

6 Appendix

6.1 Program Explanation

As mentioned above, individual tests were hard-coded. There was no facility for user-determined switching. For this reason the provided code executes only one type of test run. However, all test run configurations are included in the code in a commented form, sans replacement of lengths and target filenames. Filenames used assume the default SUSANNE Corpus filenames installed in the default directory. The program as shown is configured for a TASx test.

6.2 Program Code

```
//Robert Staubs
//Period 1

//Part-of-Speech Tagging Program

#include <iostream>
#include <fstream>
#include <vector>
#include <stdlib.h>

using namespace std;

//Struct and constructor for it for storing tags and counts within other structures
struct subtag
{
  subtag();
  string tag; //Stores tag
  int count; //Stores frequency of tag
};

subtag::subtag() :
  tag(""), count(0)
{
}

//Struct and constructor for it for storing tags, counts, and a list of preceding tags
struct tagstorage
{
  tagstorage();
  string tag; //Stores the tag in question.
  int count; //Stores frequency of tag in various occurrences.
```

```
  vector<subtag> previous; //Stores frequencies of tags occurring before this one
};

tagstorage::tagstorage() :
  tag(""), count(0), previous(1)
{
}

//Struct and constructor for it for storing words, counts, and a list of associated tags
struct lexstorage
{
  lexstorage();
  string word; //Stores the word in question.
  int count; //Stores the number of occurrences of the word in the corpora.
  vector<subtag> tags; //Stores frequencies of tags for this word.
};

lexstorage::lexstorage() :
  word(""), count(0), tags(1)
{
}

void insword(vector<lexstorage >& vect, int loc, string val);
void instag(vector<tagstorage >& vect, int loc, string val);
void inssubtag(vector<subtag >& vect, int loc, string val);

int main()
{
  string junk, newid, tag, word, oldid="0", prevtag, temp, answer;
  vector<lexstorage > words(1); //Holds words in training corpus
  vector<tagstorage > tags(1); //Holds tags in training corpus
  ifstream infile[63]; //Array of ifstreams for reading from multiple files
  ifstream target("fc/A19"); //target text
  int len, wherethetagat, wherethewordat;
//what follows are the arrays for training texts,
//with all but the general training operation on A19 commented out.
//Other changes must be made to provide for different runs
char * files [] =
{"fc/A01", "fc/A05", "fc/A09", "fc/A13", "fc/G01", "fc/G05", "fc/G09",
 "fc/G13", "fc/J01", "fc/J05", "fc/J09", "fc/J21", "fc/N01", "fc/N05",
 "fc/N09", "fc/N13", "fc/A02", "fc/A06", "fc/A10", "fc/A14", "fc/G02",
 "fc/G06", "fc/G10", "fc/G17", "fc/J02", "fc/J06", "fc/J10", "fc/J22",
 "fc/N02", "fc/N06", "fc/N10", "fc/N14", "fc/A03", "fc/A07", "fc/A11",
 "fc/G03", "fc/G07", "fc/G11", "fc/G18", "fc/J03", "fc/J07", "fc/J12",
 "fc/J23", "fc/N03", "fc/N07", "fc/N11", "fc/N15", "fc/A04", "fc/A08",
 "fc/A12", "fc/A20", "fc/G04", "fc/G08", "fc/G12", "fc/G22", "fc/J04",
 "fc/J08", "fc/J17", "fc/J24", "fc/N04", "fc/N08", "fc/N12", "fc/N18"};

//char * files [] =
{"fc/A01", "fc/A02", "fc/A03", "fc/A04", "fc/A05", "fc/A06", "fc/A07",
 "fc/A08", "fc/A09", "fc/A10", "fc/A11", "fc/A12", "fc/A13", "fc/A14"};
```

```
//char * files [] =
{"fc/A01", "fc/A02", "fc/A03", "fc/A04", "fc/A05", "fc/A06", "fc/A07",
 "fc/A08", "fc/A09", "fc/A10", "fc/A11", "fc/A12", "fc/A13", "fc/A14", "fc/A19"};
//char * files [] =
{"fc/G01", "fc/G02", "fc/G03", "fc/G04", "fc/G05",  "fc/G06", "fc/G07",
 "fc/G08", "fc/G09", "fc/G10", "fc/G11", "fc/G12", "fc/G13", "fc/G17", "fc/G18"};
//char * files [] =
{"fc/G01", "fc/G02", "fc/G03", "fc/G04", "fc/G05",  "fc/G06", "fc/G07",
 "fc/G08", "fc/G09", "fc/G10", "fc/G11", "fc/G12", "fc/G13", "fc/G17",
 "fc/G18" , "fc/G22"};
//char * files [] =
{"fc/J01", "fc/J02", "fc/J03", "fc/J04", "fc/J05", "fc/J06", "fc/J07",
 "fc/J08", "fc/J09", "fc/J10", "fc/J12", "fc/J17", "fc/J21", "fc/J22",
 "fc/J23"};
//char * files [] =
{"fc/J01", "fc/J02", "fc/J03", "fc/J04", "fc/J05", "fc/J06", "fc/J07",
 "fc/J08", "fc/J09", "fc/J10", "fc/J12", "fc/J17", "fc/J21", "fc/J22",
 "fc/J23", "fc/J24"};
//char * files [] =
{"fc/N01", "fc/N02", "fc/N03", "fc/N04", "fc/N05", "fc/N06", "fc/N07",
 "fc/N08", "fc/N09", "fc/N10", "fc/N11", "fc/N12", "fc/N13", "fc/N14",
 "fc/N15"};
//char * files [] =
{"fc/N01", "fc/N02", "fc/N03", "fc/N04", "fc/N05", "fc/N06", "fc/N07",
 "fc/N08", "fc/N09", "fc/N10", "fc/N11", "fc/N12", "fc/N13", "fc/N14",
 "fc/N15", "fc/N18"};

//Array of filenames for reading from multiple files

  for(int w=0; w<63; w++)
  {
    cout<<w<<endl;
    prevtag="Start";
    infile[w].open(files[w]); //open corpus file
    //---This section gets information from training texts---
    while(!infile[w].eof())
    {
      infile[w] >> newid >> junk >> tag >> word >> junk >> junk;
      //reads in the third and fourth items in a tab-seperated list,
      //others go into unused variable
      if(newid != oldid) //removes problem at end of file
                         //by checking this line has not been read before
      {
        oldid=newid;
//---This section stores words and frequencies thereof---
        len=words.size();
        if(len==1 && words[0].count==0)
        {
          //creates the first element, because otherwise there can be sizing problem
          words[0].word=word;
          words[0].count++;
```

```
          wherethewordat = 0;
        }
        else
        {
          for(int x=0; x<len; x++)
          {
    if(word==words[x].word)
    {
      words[x].count++;
              wherethewordat = x;
      break;
    }
    else
    {
      if((word < words[x].word && x==0) ||
                 (word < words[x].word && word > words[x-1].word))
      {
        insword(words, x, word);
        words[x].count++;
                wherethewordat = x;
        break;
      }
      else
             {
      if((word > words[x].word && x==len-1) ||
                  (word > words[x].word && word < words[x+1].word))
      {
        insword(words, x+1, word);
        words[x+1].count++;
                wherethewordat = x+1;
        break;
      }
      }
           }
        }
        }
//---This section stores tags and frequencies thereof---
        len=tags.size();
        if(len==1 && tags[0].count==0)
        {
          //creates the first element, because otherwise there can be sizing problem
          tags[0].tag=tag;
          tags[0].count++;
          wherethetagat = 0;
        }
        else
        {
          for(int x=0; x<len; x++)
          {
       if(tag==tags[x].tag)
    {
```

```
      tags[x].count++;
              wherethetagat = x;
      break;
    }
    else
    {
      if((tag < tags[x].tag && x==0) || (tag < tags[x].tag && tag > tags[x-1].tag))
      {
        instag(tags, x, tag);
        tags[x].count++;
                wherethetagat = x;
        break;
         }
      else
              {
        if((tag > tags[x].tag && x==len-1) || (tag > tags[x].tag && tag < tags[x+1].tag))
        {
          instag(tags, x+1, tag);
          tags[x+1].count++;
                  wherethetagat = x+1;
          break;
        }
      }
            }
          }
        }
//---This section stores the preceding tags---
        len=tags[wherethetagat].previous.size();
        if(len==1 && tags[wherethetagat].previous[0].count==0)
        {
          //creates the first element, because otherwise there can be sizing problem
          tags[wherethetagat].previous[0].tag=prevtag;
          tags[wherethetagat].previous[0].count++;
        }
        else
        {
          for(int x=0; x<len; x++)
          {
    if(prevtag==tags[wherethetagat].previous[x].tag)
    {
      tags[wherethetagat].previous[x].count++;
      break;
    }
    else
    {
      if((prevtag < tags[wherethetagat].previous[x].tag && x==0) ||
                 (prevtag < tags[wherethetagat].previous[x].tag && prevtag
                  > tags[wherethetagat].previous[x-1].tag))
      {
        inssubtag(tags[wherethetagat].previous, x, prevtag);
        tags[wherethetagat].previous[x].count++;
```

```
        break;
      }
      else
             {
        if((prevtag > tags[wherethetagat].previous[x].tag && x==len-1) ||
                   (prevtag > tags[wherethetagat].previous[x].tag && prevtag <
                   tags[wherethetagat].previous[x+1].tag))
        {
          inssubtag(tags[wherethetagat].previous, x+1, prevtag);
          tags[wherethetagat].previous[x+1].count++;
          break;
        }
      }
            }
          }
        }
//---This section stores word-associated tags---
        prevtag=tag;
        len=words[wherethewordat].tags.size();
        if(len==1 && words[wherethewordat].tags[0].count==0)
        {
          //creates the first element, because otherwise there can be sizing problem
          words[wherethewordat].tags[0].tag=tag;
          words[wherethewordat].tags[0].count++;
        }
        else
        {
          for(int x=0; x<len; x++)
          {
    if(tag==words[wherethewordat].tags[x].tag)
    {
      words[wherethewordat].tags[x].count++;
      break;
    }
    else
    {
      if((tag < words[wherethewordat].tags[x].tag && x==0) ||
                 (tag < words[wherethewordat].tags[x].tag && tag >
                  words[wherethewordat].tags[x-1].tag))
      {
        inssubtag(words[wherethewordat].tags, x, tag);
        words[wherethewordat].tags[x].count++;
        break;
      }
      else
             {
        if((tag > words[wherethewordat].tags[x].tag && x==len-1) ||
                   (tag > words[wherethewordat].tags[x].tag && tag <
                    words[wherethewordat].tags[x+1].tag))
        {
          inssubtag(words[wherethewordat].tags, x+1, tag);
```

```
          words[wherethewordat].tags[x+1].count++;
          break;
        }
      }
            }
          }
        }
      }
    }
    infile[w].close(); //close file
  }
  //---This section tags the target text---
  int tar, pre, rez, correct=0, total=0, hey;
  double plex, ptrans, ptot=0.0;
  prevtag="Start";
  while(!target.eof())
  {
    target >> newid >> junk >> tag >> word >> junk >> junk;
    //reads in the third and fourth items in a tab-seperated list,
    //others go into unused variable
    tar=-1;
    pre=-1;
    ptot=0.0;
    plex=0.0;
    ptrans=0.0001;
    for(int x=0; x<words.size(); x++)
    {
      if(words[x].word==word)
      {
        tar=x;
        break;
      }
    }
    if(tar != -1)
    {
      for(int z=0; z<tags.size(); z++)
      {
        if(prevtag==tags[z].tag)
{
  hey=z;
  break;
}
        if(prevtag<tags[z].tag) { break; }
      }
      for(int q=0; q<words[tar].tags.size(); q++)
      {
        for(int y=0; y<tags.size(); y++)
        {
  if(tags[y].tag == words[tar].tags[q].tag)
  {
    wherethetagat=y;
```

```
    break;
  }
        }
for(int z=0; z<tags[wherethetagat].previous.size(); z++)
{
  if(tags[wherethetagat].previous[z].tag==prevtag)
  {
    pre=z;
    break;
  }
}
        plex=((double)(words[tar].tags[q].count))/((double)(tags[wherethetagat].count));
    if(pre!=-1) { ptrans=((double)(tags[wherethetagat].previous[pre].count))/
                    ((double)(tags[hey].count));}
        if((plex+ptrans)>ptot)
{
  ptot=plex+ptrans;
  rez=wherethetagat;
}
      }
    }
    else
    {
      for(int x=0; x<tags.size(); x++)
      {
        for(int y=0; y<tags[x].previous.size(); y++)
        {
  if(tags[x].previous[y].tag==prevtag)
         {
    wherethetagat=y;
    break;
  }
}
for(int z=0; z<tags.size(); z++)
        {
  if(prevtag==tags[z].tag)
  {
    hey=z;
    break;
  }
}
ptrans=(double)(tags[x].previous[wherethetagat].count)/(double)(tags[hey].count);
if(ptrans>ptot)
        {
  ptot=ptrans;
  rez=x;
}
      }
    }
    prevtag=tags[rez].tag;
    total++;
```

```
    if(tags[rez].tag==tag) {correct++;}
  }

  cout<<"Correct: "<<correct<<"  Total: "<<total<<endl;
  //---Debug and testing of training---
/*
  cout<<"ALL WORDS:"<<endl;
  for(int i=0; i<words.size(); i++)
  {
    cout<<"  word: "<<words[i].word<<"  "<<words[i].count<<endl;
    cout<<"  WORDS TAGS: "<<endl;
    for(int j=0; j<words[i].tags.size(); j++)
    {
      cout<<"    tag: "<<words[i].tags[j].tag<<"  "<<words[i].tags[j].count<<endl;
    }
  }
  cout<<"ALL TAGS:"<<endl;
  for(int i=0; i<tags.size(); i++)
  {
    cout<<"  tag: "<<tags[i].tag<<"  "<<tags[i].count<<endl;
    cout<<"  PRECEDING TAGS:"<<endl;
    for(int j=0; j<tags[i].previous.size(); j++)
    {
      cout<<"    previous: "<<tags[i].previous[j].tag<<"  "
          <<tags[i].previous[j].count<<endl;
    }
  }*/

  return 0;
}

//insertion function for lexstorage vectors
void insword(vector<lexstorage >& vect, int loc, string val)
{
  int lenny = vect.size();
  vector<lexstorage > nyuu(lenny+1);
  if(loc==0)
  {
    nyuu[0].word=val;
    nyuu[0].count=0;
    for(int y=0; y<lenny; y++)
      nyuu[y+1]=vect[y];
  }
  else if(loc==lenny)
  {
    for(int z=0; z<lenny; z++)
      nyuu[z]=vect[z];
    nyuu[lenny].word=val;
    nyuu[lenny].count=0;
  }
  else
```

```
  {
    for(int x=0; x<loc; x++)
      nyuu[x]=vect[x];
    nyuu[loc].word=val;
    nyuu[loc].count=0;
    for(int i=loc; i<lenny; i++)
      nyuu[i+1]=vect[i];
  }
  vect=nyuu;
}

//insertion function for tagstorage vectors
void instag(vector<tagstorage >& vect, int loc, string val)
{
  int lenny = vect.size();
  vector<tagstorage> nyuu(lenny+1);
  if(loc==0)
  {
    nyuu[0].tag=val;
    nyuu[0].count=0;
    for(int y=0; y<lenny; y++)
      nyuu[y+1]=vect[y];
  }
  else if(loc==lenny)
  {
    for(int z=0; z<lenny; z++)
      nyuu[z]=vect[z];
    nyuu[lenny].tag=val;
    nyuu[lenny].count=0;
  }
  else
  {
    for(int x=0; x<loc; x++)
      nyuu[x]=vect[x];
    nyuu[loc].tag=val;
    nyuu[loc].count=0;
    for(int i=loc; i<lenny; i++)
      nyuu[i+1]=vect[i];
  }
  vect=nyuu;
}

//insertion function for subtag vectors
void inssubtag(vector<subtag >& vect, int loc, string val)
{
  int lenny = vect.size();
  vector<subtag > nyuu(lenny+1);
  if(loc==0)
  {
    nyuu[0].tag=val;
    nyuu[0].count=0;
```

```
    for(int y=0; y<lenny; y++)
      nyuu[y+1]=vect[y];
  }
  else if(loc==lenny)
  {
    for(int z=0; z<lenny; z++)
      nyuu[z]=vect[z];
    nyuu[lenny].tag=val;
    nyuu[lenny].count=0;
  }
  else
  {
    for(int x=0; x<loc; x++)
      nyuu[x]=vect[x];
    nyuu[loc].tag=val;
    nyuu[loc].count=0;
    for(int i=loc; i<lenny; i++)
      nyuu[i+1]=vect[i];
  }
  vect=nyuu;
}
```

6.3 Program Output and Explanation

```
TASa Correct: 1603  Total: 2466
TASA Correct: 2091  Total: 2466
TGSg Correct: 1623  Total: 2445
TGSG Correct: 2058  Total: 2445
TJSj Correct: 1486  Total: 2343
TJSJ Correct: 1870  Total: 2343
TNSn Correct: 1540  Total: 2377
TNSN Correct: 1849  Total: 2377
TASx Correct: 1555  Total: 2466
TGSx Correct: 1618  Total: 2445
TJSx Correct: 1429  Total: 2343
TNSx Correct: 1598  Total: 2377
```

Shown above are the various test run designations and the output given by them. The designation is not part of the output, only from Correct right is. Correct shows the number of taggings performed that are the same as the hand-tagged version. Total shows the total number of words. The TASx test for which the code is given is shown. More explanation of results can be found in the Results and Analysis sections.