Language & Speech Processing

Mid-Term Project

Hidden Markov Model POS Tagger

Roberto Valenti, 0493198
Rómulo Gonçalves, 0536601

Introduction and problem description

In this project we were required to implement a Hidden Markov Model POS tagger. To achieve this gradually, the project was shattered in four main sub-projects, or project steps.
The first step was to implement Markov language model over word sequences.
The requirements for this sub-project were to load a file with sentences (each word is delimited with a space) extracting from it a table of unigrams and bigrams with their own frequencies. After that, we had to build a procedure that calculated the probability of any word given the previous, for every bigram encountered. The next requirement was to build a procedure that takes a sentence and calculates its probability. Finally, we had to implement another procedure that takes a sequence of words as input and, by enumerating all the possible sequences that can be created with those words, outputs the sentence with the highest probability.
As an optional extra task, we managed to efficiently find the sentence with the highest probability implementing the *Viterbi algorithm*, feeding it with a lattice created by the sequence of words given as input.
The second step of the project was to use the structures we created in the first step to create a Markov language model over POS tags sequences and to extract a table of word-POS pairs with their probability, in order to have a lexical model over the word-tag pairs. Furthermore, we were required to build a procedure that takes a sentence as input and builds for that sentence a lattice assigned by a 1st-order Markov model.
The third step required us to implement the *Viterbi algorithm* for POS tagging and the forward algorithm to easily calculate the sentence probability. In this step it was required to evaluate the performance of the produced POS tagger.
The fourth and final step required to smooth the lexical model in order to deal with unseen <POS, word> pairs, evaluating and comparing it with the one done for the third step (without smoothing).
An additional evaluation of accuracy was required for known words alone and for unknown words alone.
The next sections of the report are structured as follows: we will first formalize the solutions for the given tasks, talking a bit about the theory behind each project steps; we will then discuss some of the implementation choices and details together with the description of the graphical interface we built for this project. Furthermore, we will illustrate some empirical experiments we performed with the final tagger, comparing them with some other experiments of the tagger in some of earlier developing stages. Finally, before come to a conclusion, we will start a discussion about the results obtained, the effects and the reasons of the decisions we took during developing phases.

Formalization of the solution

In this section we will describe the theory behind each step required for this assignment. Extracting unigrams and bigrams from a text is a trivial task, but we should spend some words about why we need them and describe the language model that was built with them.

Let's define the formal language as set **Ω** of sequences of words from a vocabulary **V**, where all x ϵ **Ω** are called sentences. The language model is a probability distribution over the formal language

$$P: \Omega \rightarrow [0,1] \quad \sum_{x \in \Omega} P(x) = 1$$

If we have a vocabulary **V** and a probability model over word sequences $P: \Omega \rightarrow [0,1]$ we can predict the next word w_n given the preceding $w_1,\dots,w_{n-1}$ word with the conditional probability

$$P(w_n \mid w_1,\dots,w_{n-1}) = \frac{P(w_1,\dots,w_n)}{P(w_1,\dots,w_{n-1})}$$

where we can call $P(w_1,\dots,w_{n-1})$ the "history". Using the chain rule, we can define

$$P(w_1,\dots,w_n) = P(w_1)\prod_{i=2}^{n} P(w_i \mid w_1,\dots,w_{i-1})$$

In our case, in order to significantly reduce the number of probability estimations, we were requested to build a first order Markov model, which is based on the assumption that

$$P(w_1,\dots,w_n) \approx P(w_1)\prod_{i=1}^{n-1} P(w_{i+1} \mid w_i)$$

This leads to a very simple probability estimation for word sequences:

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_i)}$$

where *count(x)* represents the frequency of *x* in the corpus.
These frequencies are stored on the unigrams and bigrams tables.
In the first step of the project, we were required to build a compact representation of all the possible word combination and calculate the most probable sentence starting from the set *S* = *{proposed, The, investors, changes, to, the, federal, funds, .}*
Using the implemented functions, the results were the following:

> *1)The proposed changes proposed to the federal funds .*
> *2)to the federal funds to the federal funds .*

Note that the first result and the second result were obtained using respectively a case sensitive and a case insensitive version of our program, and those results match with the results obtained through the *Viterbi algorithm*. We will discuss the *Viterbi algorithm* later in this section, while we will discuss the case sensibility of the program in the next section (Implementation details).

For the second step, we were required to create a Markov model over POS (Parts of Speech) tags. POS tags are classes associated to words (such as verbs, nouns, adverbs, adjectives, prepositions, determiners etc…). Following the same model described above for words, we could implement the same model using those classes. The problem is that there isn't a one-to-one mapping between POS tags and words. This creates ambiguity (each word can be associated to multiple POS tags) that can be solved through syntactic analysis.

To explain better how a POS tagger can disambiguate, let's introduce the noisy channel model. In this model (represented in figure 1), we assume that **A** has a concept and wants **B** to receive this concept. In our case this concept is the correct POS tagging of the sentence and **B** has to reconstruct the correct POS tagging which **A** had in mind starting from the observations he get through the noisy channel (in figure 1, this is represented by the sentence "list the list").

Figure 1 - The noisy channel model

So, the POS tagger (B) should start from the observation $w_1,\dots,w_n$ and try to reconstruct the original message $t_1,\dots,t_n$.
In order to disambiguate, the tagger should select a sequence of $t_1,\dots,t_n$ that maximizes $P(t_1,\dots,t_n \mid w_1,\dots,w_n)$. This is:

$$\arg\max_{t_1,\dots,t_n} P((t_1,\dots,t_n) \mid (w_1,\dots,w_n)) = \arg\max_{t_1,\dots,t_n} \frac{P((t_1,\dots,t_n),(w_1,\dots,w_n))}{P(w_1,\dots,w_n)} =$$

$$= \arg\max_{t_1,\dots,t_n} P((t_1,\dots,t_n),(w_1,\dots,w_n))$$

Where

$$P((t_1,\dots,t_n),(w_1,\dots,w_n)) = P(t_1,\dots,t_n) \times P((w_1,\dots,w_n) \mid (t_1,\dots,t_n))$$

Here we can distinguish two main components: the Language model $P(t_1,\dots,t_n)$ and the Lexical model $P((w_1,\dots,w_n) \mid (t_1,\dots,t_n))$. The tagger approach to disambiguate the observation coming from the noisy channel is to create all the possible interpretation of $t_1,\dots,t_n$ then use his knowledge of the lexical model and the language model to give a probabilistic value to those hypothesis, in order to finally select the hypothesis with the maximum probability.
This pipeline is displayed in figure 2.

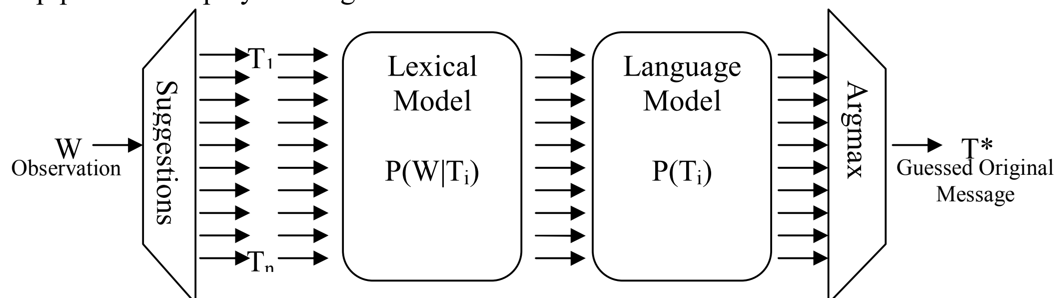

Figure 2 - Pipeline of a typical POS tagger

The language model over POS tags is straight forward from the language model over words (see above), while the lexical model is given by

$$P(w_1,\dots,w_n \mid t_1,\dots,t_n) = P(w_1 \mid t_1,\dots,t_n) \prod_{i=2}^{n} P(w_i \mid (t_1,\dots,t_n),(w_1,\dots,w_{i-1}))$$

In order to simplify this formula, we can use the following independence assumptions:

- Every word is independent of all other words

$$P(w_1,...,w_n \mid t_1,...,t_n) = P(w_1 \mid t_1,...,t_n)\prod_{i=2}^{n} P(w_i \mid (t_1,...,t_n))$$

- Every word depends only on its own tag

$$P(w_1,...,w_n \mid t_1,...,t_n) = \prod_{i=1}^{n} P(w_i \mid t_i)$$

The two models (Language model and Lexical model) can be combined together in

$$P(t_1,...,t_n \mid w_1,...,w_n) = P(t_1)\prod_{i=1}^{n-1} P(t_{i+1} \mid t_i) \times \prod_{i=1}^{n} P(w_i \mid t_i)$$

As stated before, the tagger should generate or suggest a list of possible POS sequences in order to select the most probable one. We can represent this process as a lattice of states, where each state consists of a POS tag. The lattice is represented enclosed in the grey part of figure 3, where all the arrows indicate a possibility of transaction from one state to another. As shown in this lattice, there are 4x2x2x2= 32 possible paths (indicated by the arrows) from the beginning of a sentence consisting of only four words. Considering that most of the sentences are way larger than four words, we can expect the complexity to grow exponentially on the sentence length.
Enumerating all the possible POS sequences for a sentence, and calculating for each of them the one with the highest probability is computational expansive.
To solve this problem we used the *Viterbi algorithm* (as requested by step 3 of the project).
The main idea behind this algorithm we can use the time invariance of the probabilities to reduce the complexity of the problem in such a way that it is possible to reconstruct the optimal path in the lattice starting from the end symbol, avoiding the necessity of examining every route through the lattice. In order to do that, the algorithm keeps a backward pointer for each state and stores a probability (γ_t) with each state at time t.

This probability is the probability of having reached the state following the path indicated by the back pointers. When the algorithm reaches the states at time t = T, the γ_t's for the final states are the probabilities of following the optimal (most probable) route to that state. Thus selecting the largest, and using the implied route, provides the best answer to the problem.
To calculate the probability $\gamma_t(state_i)$ for time t the following formula is used:

$$\gamma_t(state_i) = \max_{\substack{state_j \\ w}} \gamma_{t-1}(state_j) \times P(state_i \mid state_j) \times P(w \mid state_i)$$

The formula includes both the language and the lexical model described before. For instance, if we consider the state named NNS on time 4 of figure 3, its probability γ_4 is equal to the maximum of the products given by the γ_3 of each of its previous states, multiplied with the transition probability and the lexical model probability for each of the previous connected states at t=3 (NNP, CC). One can prove that $\gamma_n(<end>)$ is the probability of the best path.

An important point to note about the *Viterbi algorithm* is that it does not greedily accept the most likely state for a given time instant, but takes a decision based on the whole sequence. Thus, if there is a particularly 'unlikely' event midway through the sequence, this will not matter provided the whole context of what is seen is reasonable.
Furthermore, once the algorithm was implemented, we could use the same structure to calculate "on-the-fly" the sentence probability through the *Forward Algorithm*, which closely resembles the *Viterbi algorithm* (except for taking the sum instead of the max).

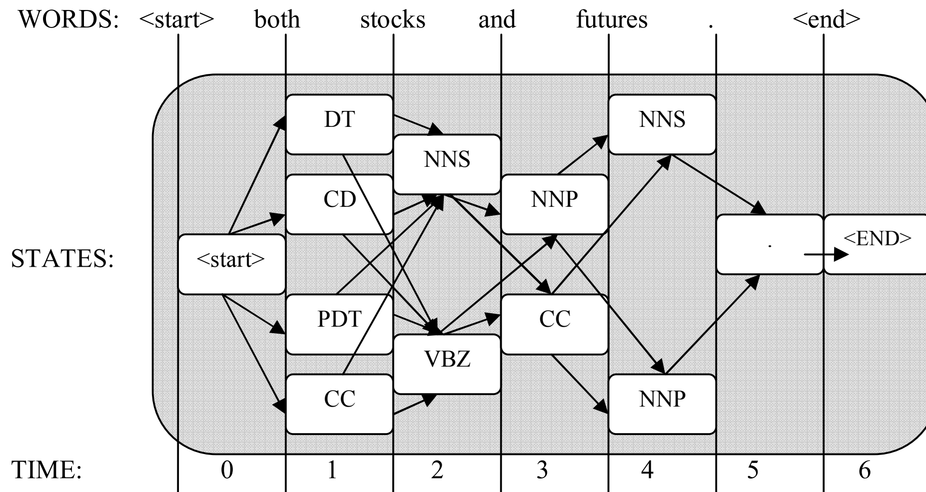

Figure 3 - Pipeline of a typical POS tagger, the grey area represent the lattice created with the sentence “ both stock and futures .”

For the fourth task we were requested to smooth the lexical model in order to deal with unseen <POS,word> sequences.
Generally, different smoothing algorithms are available to deal with unseen words, some works by reserving and redistributing a mass in the probability distribution (discounting), some works by interpolating the maximum likelihood counts of bigrams with unigrams (interpolation), some others works by adding a small value to all the probabilities in order to be able to assign a probability slightly bigger than 0 to the unseen words (Adding λ).
For “morphologically-poor” language as English, it is possible to deal with unseen words by bucketing them in classes that based on the suffixes, prefixes or parts of the unknown word. The main question is how to generate those classes, and which rules to use. The solution we found is to select the <POS, word> sequences that occur only once in the training set and use those to create new classes based on the following rules:

Class Name	Description	Possible matching cases
"nx"	generic number	99.9, 12-15-2005, 1/4
"anx"	alphanumeric sequence	Butterfly84, 7up
"xzx"	word-word	simple-minded, co-worker
"xen"	word-en	Fallen, swollen
"xed"	word-ed	Programmed, joined
"xify"	word-en	Simplify, amplify
"xful"	word-ful	Useful ,wonderful
"xion"	word-ion	Intention, possession
"xable"	word-able	Enable, affordable
"xing"	word-ing	Justifying, applying
"xly"	word-ly	Finally, suddenly
"xs"	plurals, word-s	Accounts, houses
"X"	Uppercase words	NASA, Microsoft
"cox"	co-word	cooperate,construct
"dex"	de-word	destruct, debug
"disx"	dis-word	disconnect , disengage
"xx"	generic unknown word	accident, jijijij

Table 1 – Rules for bucketing

In our idea, this new classes are treated as if they were words present in the training corpus. The corresponding tags, with their own frequencies are automatically learned from the one-frequency <POS, word> sequences we selected from the corpus. To deal with the mass created by this procedure, it is also possible to consider the classes as words present in the corpus, instead of using a smoothing technique. We will leave this discussion about smoothing and bucketing to the apposite section (*Discussions*). In the rest of the report, we will refer to this procedure as "smoothing"

Implementation details

For its availability and platform independency, we decided to use java as our developing environment.
This sped up the process of creating a graphical interface to the program and introduced a nice object oriented approach to the problems, but created additional problems that we could avoid if we decided to use scripting languages such as PHP or Python.
We implemented every part of the project keeping in mind the similarities of the Markov models over words and over POS tags, in such a way that the user could easily switch between the two modalities and still using the exact same functions.
The effect of this choice can be easily seen on the interface, where the user can easily switch between the two modalities.

The Interface

When we started implementing the tagger we decided to create a graphic interface to allow end-users to experiment with the POS tagger without modifying and recompiling the code. The interface is really easy to use and offers a good way to test the more important functions. It is possible to find compiled version of the tagger, together with the source files, at the address http://staff.science.uva.nl/~rvalenti/uva/lsp/tagger.rar (refer to the readme file to obtain instruction on how to run the tagger)
Figure 4 illustrates the interface that appears when the tagger is executed.

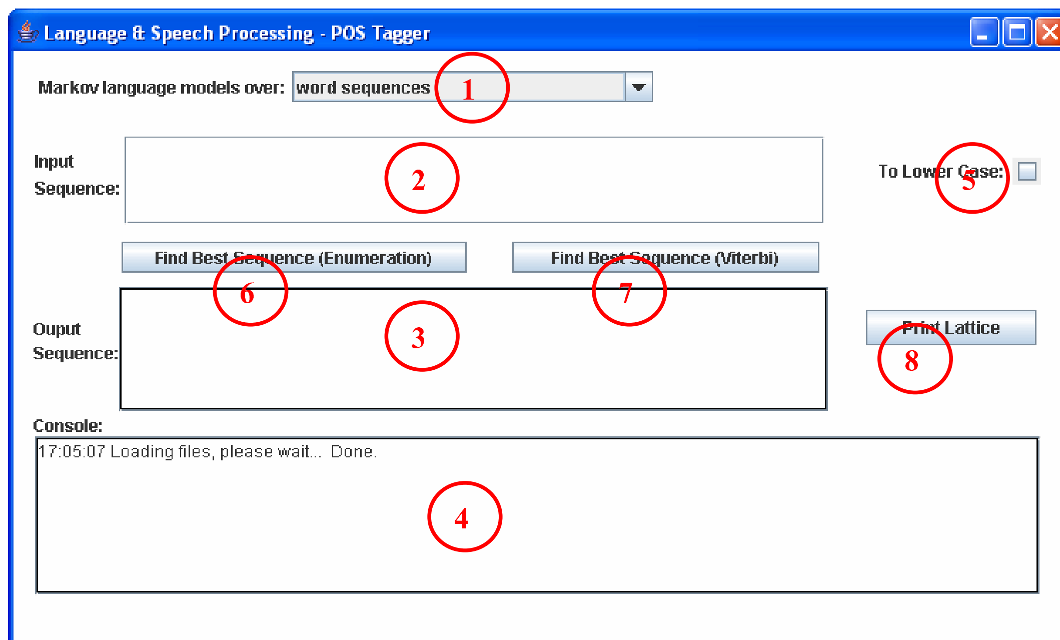

Figure 4 - The graphical interface in "Word sequences" modality.

Using (1), the user can decide the main configuration of the tagger (Markov language models over words or over POS tags).
In (2), a user should type a sequence of words. Button (5) defines whether or not both the input sentence and the training corpus have to be case sensitive. By pressing on (6), the application will enumerate all the possible combinations of the input words, and will print in (3) the most probable sentence that could be generated, together with its probability. The same result is obtained by pressing on button (7), but the search is done "efficiently" using the *Viterbi algorithm.* Internally in the code, the function under the button (6) creates a three and explores it in order to enumerate all the possible combinations. This function accepts different three depths, but for the interface we decided to enforce the three depths to be equal to the length of the input sentences.
With (8), it is possible to print the lattice associated to the sentence, which is going to be used in the *Viterbi algorithm*. This is more interesting if done over POS tags instead of words.
A valuable addition in the user interface is the console (4), which gives information about the internals of the program, for instance when it is loading a file. The user should wait until the end of the operations on the console, or he could get some strange behaviours. Note that the application declares to be loading files at start up. Those are the files containing the sentences and the pos tagged sentences, which are loaded in memory to obtain a slightly faster access to it when they need to be parsed (this happens all the times a setting is changed).
Each time a switch between "word sentences" and "POS tag sequences and lexical models" using (1) as in figure 5, the interface adds, preserves or removes some of the functionalities, in order to reduce to a minimum the interface complexity.

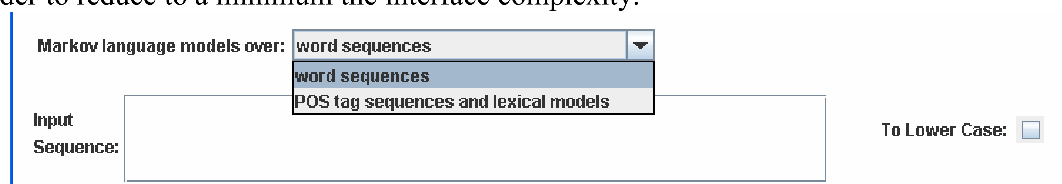

Figure 5 - Switching modalities in the graphical interface

When the user selects to use the Markov language model over "POS tag sequences and lexical models", the interface will mutate as in figure 6

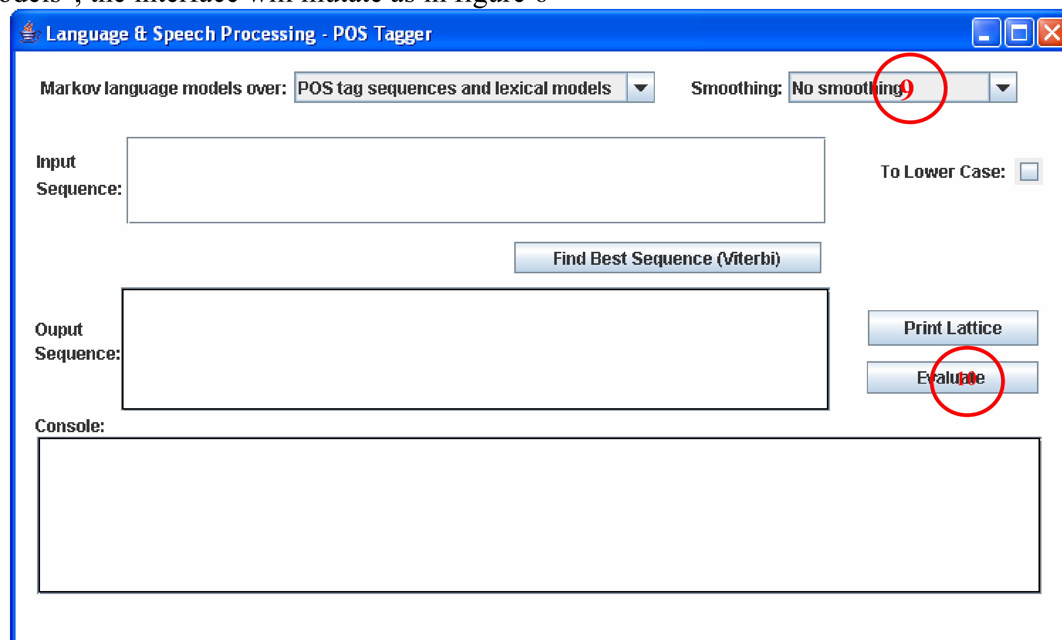

Figure 6 - The graphical interface in "POS tag sequences" modality

The button (6) disappeared, since it is possible to use it only when dealing with word sequences, while new functionality, (9) and (10), are now available. With (9), the user can choose the smoothing method used to deal with unseen words in the training data (none or bucketing), and with (10) the user can open an evaluation window used to evaluate the POS tagger. Pressing on (10) the window in figure 7 is added to the interface.
In this window, the “sentence file” field requires the name of a file with untagged sentences (has for default the file “test23.sentence), while the “Test File” requires the name of the file with the correctly tagged sentences (has for default the file “test23.gold”). These files are the files available on web-site of the course.
After click on the button “Evaluate” (11), a result like the one in figure 7 will appear, containing statistics about the performance of the POS tagger over the given files. The evaluation and the results are explained and analyzed in the next section.

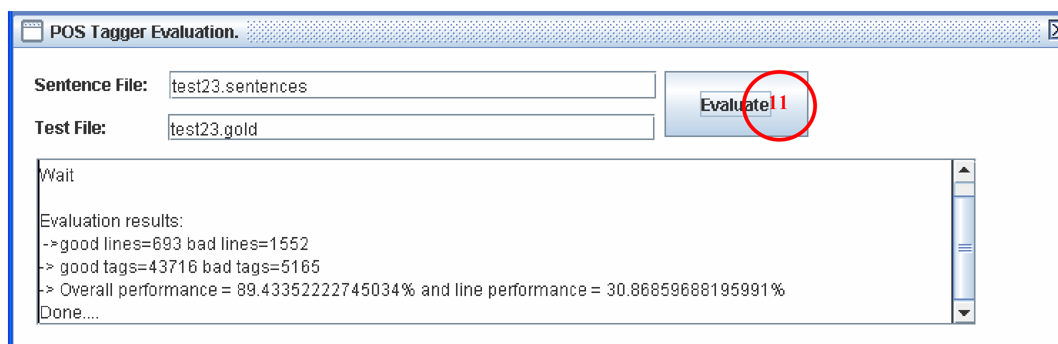

Figure 7 - The evaluation window

Implementation choices

During the development of the tagger we had to make some decisions, which we think is important to discuss in this section since they may create incongruence with other implementations.
A big issue was the case sensitivity of the tagger: in the assignment it was requested to don’t consider differences between uppercase and lowercase words. We believe that by considering differences between upper and lower cases, we can improve the language model, so we considered implementing an easy way to switch between the two cases, in such a way that we could easily compare the results. This comparison is done in the experiment section of this report.
In the earlier version of our tagger, we used “^” as start symbol, and “.” as end symbol, as was suggested in the assignment. While experimenting with the tagger, we realized that some of the sentences had a dot in the middle. For instance, the quote *“a quote . ”* . contains a dot and can easily be found in the middle of a sentence. We decided to treat each dots as a normal word and store the probability model for it, replacing the previous end symbol with the tag <end>. Of course the probability of a dot being followed by the <end> tag is very high, while the probability of each word being followed by <end> should be 0 (unless one of the sentences finished without a dot).
Together with <end> tag, we changed the “^” symbol to the <start> tag. We decided to don’t use the combination “<s>, </s>” respectively for start and end tag in order to avoid strange behaviors of the application while reading pos tags (in fact, </s> could be easily interpreted as the word “s>” being tagged as POS “<”. This implementation choice forces a dot to be present at the end of every sentence, otherwise the probability of the sentence will be null. To help the end-user in not committing this error and think the tagger is buggy, we decided to

detect if the dot is present in the end of the user input and, in case it isn't, add it automatically. Please refer to the source code to have additional implementation details.

Empirical experiments

In order to test the accuracy of the tagger, we implemented a function which takes as input a file with sentences and a file with the correct tags. The function reads each sentence and generates a lattice from it. This lattice is then used by the *Viterbi algorithm* to tag each word. After tagged, the sentence is compared to the correct tagged sentence, read from the correct file.
During the experiments, we evaluated the following performances of the POS tagger: the percentage of the correct tagged word over the full test set, the percentage of fully correctly tagged sentences, the improvements given by bucketing unseen <POS, word> sequences, the percentage of correctly assigned tags using smoothing and the percentage of correctly tagged sentence over the sentence length.
We will show four different experiment settings, as result of the combination of two parameters:

1. Smoothing (none / bucketing)
2. Case (sensitive / insensitive)

The files used for these 4 experiments were:
test23.sentences: This file contains 2245 sentences, one sentence on each line.
test23.gold: This file contains the correctly tagged sentences of test23.sentences

The following are the outcomes of the four experiments:

Experiment 1:
Settings: Smoothing= none; Case = insensitive
Results:
Good Tags: 42735 Bad Tags: 6146
Good Lines: 612 Bad Lines: 1663
Overall Performance: 87.43% Line Performance: 27.26%

Experiment 2:
Settings: Smoothing = none; Case = sensitive
Results:
Good Tags: 43716 Bad Tags: 5165
Good Lines: 693 Bad Lines: 1552
Overall Performance: 89.43% Line Performance: 30.87%

Experiment 3:
Settings: Smoothing = bucketing; Case = insensitive
Results:
Good Tags: 45787 Bad Tags: 2296
Good Lines: 842 Bad Lines: 1403
Overall Performance: 95.23% Line Performance: 37.50%
Correctly Smoothed Tags: 823 Smoothing performance: 76.84%

Experiment 4:

Settings: Smoothing = bucketing; Case = sensitive

Results:

Good Tags: 46222 Bad Tags: 1734

Good Lines: 998 Bad Lines: 1247

Overall Performance: 96.38% Line Performance: 44.45%

Correctly Smoothed Tags: 950 Smoothing performance: 79.03%

Analyzing these four experiments we can conclude that the best performance is obtained by using Bucketing and a case sensitive setting.

An important observation is that the performance increases if we used the case sensitive setting instead of the suggested case insensitive setting. This can be seen in the first and third experiment, where the overall performance and line performance are smaller than the other experiments. If we compare the experiments that used bucketing with the others, it is obvious that bucketing is very important for the accuracy of our system: the overall performance improved around 8% in both cases, while the number of correctly tagged sentences improved around 7% in the case insensitive and 14% in the case sensitive setting.

Together with the overall performance, we also analyzed the performances per sentence length. The graph in figure 8 represents the results of this evaluation. It is very easy to notice drop in performance by increasing the sentence length, indicated by the red line. Note that the highest peak of performance is close to 8% of accuracy with sentences of length 8. Since the performances for sentences longer than 41 words was 0%, we decided to don't display them.

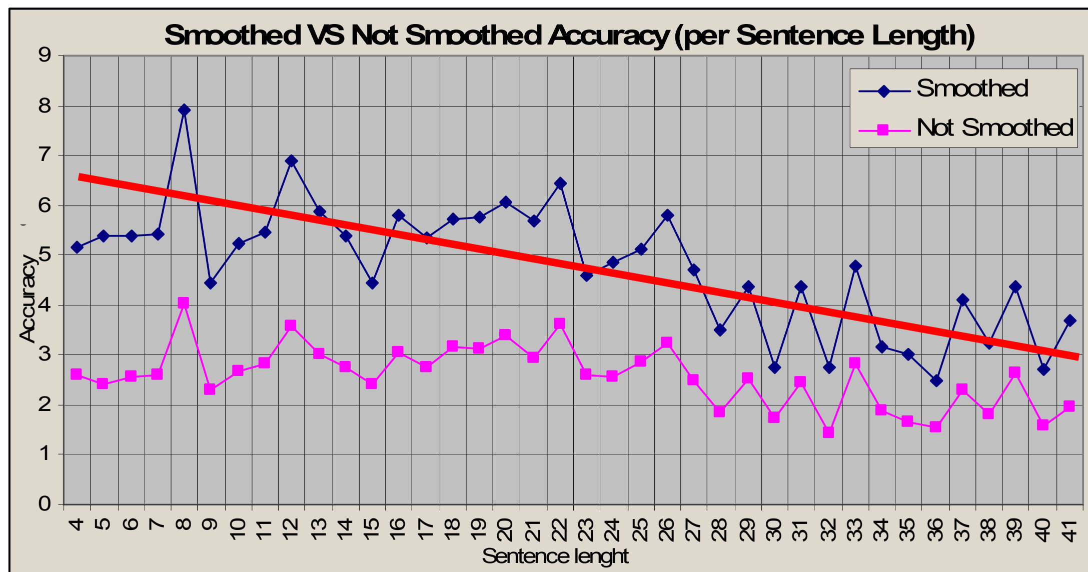

Figure 8 - Smoothed vs. not smoothed accuracy in relation with the sentence length

Discussions

A good discussion can be done analyzing the results.
Looking at the graph in figure 8, we might question whether or not the tagger works correctly. With an overall tag accuracy of 96.38% one might expect that the 44.45% of correctly tagged sentences should be heavily distributed over short sentences and a lightly over long ones. This can be seen in the graph, but the difference between the max and minimum performance on the displayed partition of the data is only 5.5%. This is not the huge difference in performance we were expecting. This is even worse for the not smoothed version of the tagger indicated by the purple line, where the difference is around 2.5%. We think that one of the reasons behind this is that for short sentences it is not possible to disambiguate correctly, while for long sentences the probability to disambiguate is overwhelmed by the probability of making a mistake given the big number of words (i.e. it is more likely to find an unknown word in the sequence). Another factor could be the distribution of the length of the sentence, which resembles a bell shape. We believe that the combination of this facts are contributing somehow to the small slope of the performance lines.

Another important discussion is about the weird way of combining smoothing and bucketing.
As stated before, we used an original method to smooth the lexical model and deal with unseen words: In our method, the new word classes generated by the rules are treated as if they were words present in the training corpus. By this, we mean that instead of smoothing the rest of the data in order to create some mass for the unseen words, and instead of removing the low-frequency word to actually free the space for the new generated word classes, we actually left the latter untouched, and update the language model over POS adding the additional generated frequencies.
After noticing the good performance achieved using this method (at least on the given dataset), we decided to drop the good-turing smoothing and use this variant.
Of course all the possibilities were explored during the development phase, and it is still possible to locate them commented inside the source code.

During this report we discussed some tricks to improve the accuracy of our tagger (such as use the case sensitivity to build better models).
Another trick we used to improve the performance is coming from the knowledge of human mistakes in the test set. In fact, the test set was tagged by humans, and “human taggers” may not be able to correctly disambiguate on all the cases and/or may use a tag incorrectly. In this case, we noticed that there was an incorrect use of the tag NNPS, so we decided to rename the encountered NNPS tagged words to NNP. As expected, incorporating human knowledge in machine learned models leaded to a better overall performance of the tagger.

Conclusions

In this assignment we built a first order Markov model POS tagger from scratches.
Beside the required tasks, we additionally implemented a usable interface for the program, which allows users to interact end experiment with the tagger over both POS and word models without changing and recompiling the code. Furthermore we tried to boost the performance of the tagger introducing bucketing for unseen words, adding some other tricks based on the human knowledge of errors on the test set, together with exploring the best settings for the given test set.
Overall, we feel that implementing the tagger helped us to understand and get good hands-on experience with language and lexical modeling.