Language and Speech Processing
Midterm project
University of Amsterdam, 2005/2006

Gideon Borensztajn, Jiří Iša

November 10th, 2005

Abstract

This project aims at building a Hidden Markov Model POS tagger and testing it on actual data. It is developed as a *Midterm Project* for *Language and Speech Processing* at University of Amsterdam, 2005/2006. In the article we discuss the topic and describe our solution.

1 Introduction and problem description

In the current state of the technology there is a big emphasis to automatic text understanding, information retrieval and communication with the user. The aim of the *Language and Speech Processing* lectures is to explain the statistical language modelling. To allow the students the better and deeper understanding of the topic, they are supposed to develop a *Midterm Project*.

Midterm project contains several steps. Each of them contains several techniques that make us closer to the final goal - Part Of Speech Tagger (POSTagger).

The difficulty of the language processing arises mainly from *ambiguity* - the fact that the word may have several meanings and part of speech tags. For instance *list* might be a noun or a verb. If more ambiguities are combined in the single sentence, it may became extremely difficult to automatically understand the text.

There are several techniques to deal with the text processing. One of them is to make the grammar of the language manually. This proved to be very difficult, sensitive to slight changes and time consuming. The other way is to use supervised learning methods. The ultimate goal is to develop a learner, that, provided by the training text tagged by a human, could extract the language knowledge by itself. One such opportunity are *Hidden Markov models (HMM)*, based on the statistically discovered features in the text.

2 Formalization of the solution

2.1 Probabilistic Language models

Let us define the probabilistic approach first. Let V be the set of all words in the language and Ω a set of sequences of words from V. $x \in \Omega$ is called a *sentence*. Then we can assign a probability to every sentence:

$$P:\ \Omega \rightarrow [0,1]$$

$$\sum_{x \in \Omega} P(x) = 1$$

2.2 Markov models

In the Markov models there is an assumption, that a probability of a word appearing in the sentence is dependent only on the previous words. Then the probability of the sentence may be written as:

$$P(w_1, w_2, ..., w_n) = P(w_1) \prod_{i=2}^{n} P(w_i | w_1, ..., w_{i-1})$$

Then we assume, that the word is dependent only on few, let's say k, previous words. We may rewrite the expression above as:

$$P(w_1, w_2, ..., w_n) = P(w_1) \prod_{i=2}^{n} P(w_i | w_{i-k}, ..., w_{i-1})$$

In the following text we work with first order Markov models ($k = 2$):

$$P(w_1, w_2, ..., w_n) = P(w_1) \prod_{i=2}^{n} P(w_i | w_{i-1})$$

To obtain the maximum likelihood estimate of the conditional probability $P(w_i|w_{i-1})$ the text is parsed and counts of *bigrams* $c(< w_{i-1}, w_i >)$ are calculated. The similar, $c(< w_i >)$, is done for *unigrams*. From the definition of the conditional probability we obtain:

$$P(w_i|w_{i-1}) = \frac{P(< w_{i-1}, w_i >)}{P(< w_i >)} = \frac{\frac{c(< w_{i-1}, w_i >)}{N}}{\frac{c(< w_i >)}{N}} = \frac{c(< w_{i-1}, w_i >)}{c(< w_{i-1} >)}$$

where N is the number of words in the training text.

2.3 Hidden Markov models

So far we have introduced basic *Markov models*. Now it is time to introduce *hidden variables*. For our work hidden variables are *part of speech tags*, that we do not observe in a normal sentence. They are hidden. They were relevant when the sentence was constructed and we would like to reconstruct them again to understand the sentence better.

The model is now viewed as a noisy channel: on the side of the sender a message of POS tags (t_1^n) is entered into the channel, during transfer they spit out words, and the receiver has to reconstruct from the sequence of observed words (w_1^n) which POS tags were meant by the sender.

We are looking for the sequence of tags t_1^n that gives the highest probability of the sentence $P(w_1^n|t_1^n)$:

$$argmax_{t_1^n} P(t_1^n|w_1^n) = argmax_{t_1^n} \frac{P(t_1^n, w_1^n)}{P(w_1^n)} = argmax_{t_1^n} P(t_1^n, w_1^n)$$

$$P(t_1^n, w_1^n) = P(w_1^n|t_1^n) * P(t_1^n)$$

$P(t_1^n)$ is called a *language model*, defines the probability of sequence of tags and describes the underlying structure of the language, the POS tags, as a *Markov model*. $P(w_1^n|t_1^n)$ is a *lexical model*, defines the probability of a word being *spit (emitted)* on the place of the sentence with given (hidden) tag and captures the relation between POS tags and words. Both models can be constructed independently.

3 Implementation details

3.1 General details

For our implementation we chose the Java programming language. It offers rich programming environment and allows rapid development. We use Java 1.5 as a Java runtime environment. It is available for free download as Java 5.0 from http://java.sun.com. The code and resulting Java binaries are not backward compatible with previous versions of Java Runtime.

3.2 Step 1

The goal of the first step of the project is to build a 1st-order Markov model over word sequences. It would be a first-order Markov model (as described above). We were provided with the training file.

The reading of the datafile and counting of *unigrams* and *bigrams* is done in a class called *WordCounter*. The datafile is read sentence by sentence, and START and END symbols are appended before and after every sentence. *WordCounter* keeps track of the unigrams and their frequency in a *HashMap* called `unigrams`, and it keeps track of the frequency of *bigrams* in a *HashMap* called `bigrams`. It has a method `getConditionalProbability` that, given two words, w_x and w_y, returns the conditional probability $P(w_x|w_y)$ as defined in 2.

The method `calculateSentenceProbability` takes as input a sentence and returns the sentence probability.

We tested conditional probabilities for the following word pairs:

- Probability of 'Motor' following 'Rolls-Royce'= 0.50
- Probability of 'the' following 'the'= 0.000188
- Probability of 'END_SYMBOL' following 'changes'= 0.124

We also tested the long sentence from training file. Probability of the sentence *Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990* is $3.66 * 10^{-38}$.

Finally, we calculated the sentence with highest probability, given a set of words. We constructed a timeline for the length of the sentence, with a columns containing all the words from the set at every timepoint on the timeline. If the timeline has a single starting point and ending point, it is called a *lattice*, the nodes are called *states*. A sentence is then any path through the *lattice*. The content of the columns was prepared in the *POSLattice* constructor.

3.2.1 Viterbi algorithm

It is inefficient to find the sentence with highest probability by enumerating every possible path through the lattice. Therefore, usually a dynamic programming algorithm is used. In our case the Viterbi algorithm.

In the Viterbi algorithm, the maximum probability of the sentence is calculated incrementally, by going through the timeline from left to right, and remembering, at every state on the lattice, the most probable sentence until this point, together with its probability. For this purpose we introduced the class *StateVector*, which has members `word`, which is the current word at the state, `bestHistory`, which is the path leading up till the most probable sentence, and `maxProbability`. For *Step 3*, the *StateVector* also has members `summedProbability` and `emissionProbability`. In the first step we simply set `emissionProbability` to 1.

The `calculateViterbi` method then finds for every state the most probable partial sentence, depending only on the most probable sentence of the previous states and the conditional probability of the current word conditioned on the previous words. When it reaches the end of the *lattice*, the most probable sentence is the one with the highest probability stored in the last state.

3.2.2 Experiments

We tested the set of words: $S =$ *{proposed, The, investors, changes, to, the, federal, funds, .}*. The most likely sentence (using the Viterbi-algorithm) turned out to be *"The proposed changes proposed changes to the federal funds"* with probability $4.69 * 10^{-17}$.

For the set *{record, n't, has, a, set, date, been, . }* the most probable sentence is: *"A record date has been . A record"* with probability $1.82 * 10^{-20}$.

And for the set *{the, investor, changes, were, an, suggested, by, . }* (which does not form a sentence available in the training corpus) it is: *"The changes were by changes were an investor"* with probability $2.51 * 10^{-20}$.

The reason, we believe, for the repetition of words, is that it is only a first order Markov model. So the model does not incorporate long distance dependencies between words.

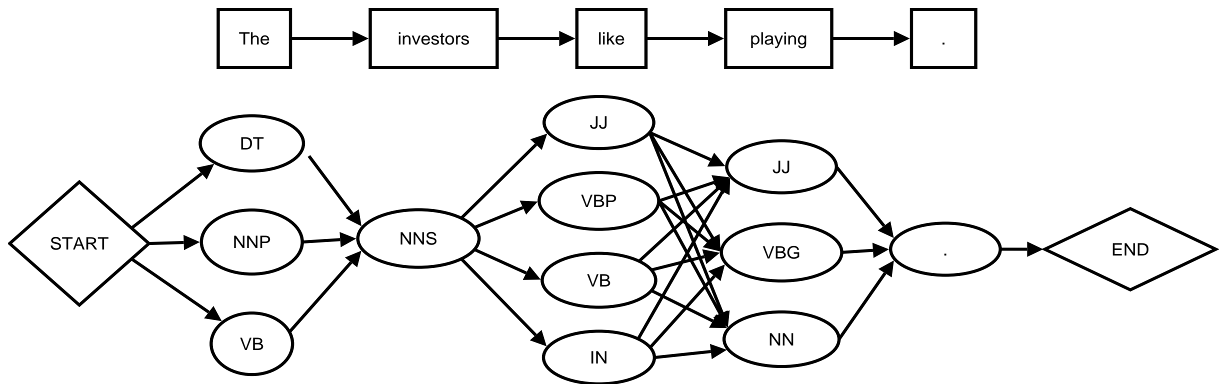

Figure 1: Lattice for the Viterbi algorithm for the sentence *"The investors like playing . "*

3.3 Step 2

In *Step 2*, the goal is to build the components of a POS tagger: a 1st-order Markov language model over POS tags and a lexical model over word-tag pairs.

The data was the same text file as in *Step 1*, but now annotated with POS tags for every word.

As usually we start by estimating the parameters of the model. For the *language model*, this is analogous to *Step 1*, but instead of words we use POS tags. We counted occurrences of single POS tags (*unigrams*), and of pairs of POS tags (*bigrams*) and from this the conditional probability of transition between POS tags. All this is done in the class *TagCounter*.

Parameter estimation for the *lexical model* is done in the class *LexicalCounter*. Here, the bigrams are pairs of a word with its corresponding tag, and the probability of emitting a word is conditioned on the tag. This gives the emission probability of the word: $P(word|POS)$.

At the same time, a *lexicon* is updated for later use. The *lexicon* lists for every word in the datafile the possible tags that cooccur with the word. This is done in the method `updateLexicon`.

3.3.1 Experiments

For our experiments we have chosen a sentence *"The investors like playing."*.

- Possible tags for the word *The* are: VB, DT, NNP
- Possible tags for the word *investors'* are: NNS
- Possible tags for the word like are: JJ, VBP, VB, IN
- Possible tags for the word *playing* are: JJ, VBG, NN
- Possible tags for the dot are:

As we can see on the possible tags of the word *The*, our learning file is not flawless.

The best tag sequence for the sentence *"The investors like playing ."* is: DT NNS IN NN. As we see one ambiguity was solved wrongly.

On the figure 1 we can see the constructed lattice for our selected sentence.

3.4 Step 3

The goals of this step were to calculate the most probable POS tag sequence for a given sentence, and to calculate sentence probability.

To calculate the most probable POS tag sequence, given a sentence, we can use again the Viterbi algorithm, over the *language model*. For every word of the sentence the possible POS tags are retrieved

from the *lexicon*, and then passed as arguments to the *POSLattice* class. In addition, together with the POS tag, this time also the emission probabilities for the current word of the sentence conditioned on the POS tag are passed as arguments, and stored in every node of the lattice as `emissionProbability`.

The only change to the Viterbi algorithm as compared to *Step 1* is that we now have to multiply the probability of the previous state by both the transition probability between tags $P(t_i|t_{i-1})$ and the emission probability $P(w_i|t_i)$ as described in chapter 2.This is implemented in the method `calculateViterbi`.

3.4.1 Experiments

Most probable tag sequences The algorithm finds the following most probable tag sequences for the sentences:

- *The investors are rich .*
  DT NNS VBP JJ .
  correct

- *Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990 .*
  NNP NNP NNPS NNP VBD PRP VBZ PRP$ NNP NNS TO VB JJ IN IN CD NNS IN CD .
  correct

- *A record date has n't been set .*
  DT NN NN VBZ RB VBN VBN .
  correct

- *The changes were suggested by an investor .*
  DT NNS VBD VBN IN DT NN .
  correct

Sentence probability The sentence probability is calculated with the *Forward algorithm*, which is similar to the Viterbi algorithm. The Forward algorithm remembers for every state at every timestep the summed probability of all tag sequences which yield the given sentence. The sentence probability is the `summedProbability` of the last column, which contains the `END` symbol. It is done in the method `calculateForwardAlgorithm` of the *Lattice* class.

We calculated the probabilities for the following sentences:

- P(*The investors are rich .*) $= 3.46 * 10^{-13}$
- P(*Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990 .*) $= 9.84 * 10^{-63}$
- P(*proposed the investors changes to The federal funds .*) $= 4.24 * 10^{-26}$
- P(*A record date has n't been set .*) $= 3.980 * 10^{-24}$
- P(*The changes were suggested by an investor .*) $= 7.77 * 10^{-23}$

Accuracy After we trained the parser on the training corpus, we tested it on a test corpus, from which the POS tags were used for error checking. The accuracy of the parser is measured as the percentage of correctly tagged POS tags over the entire test corpus (all words). It is calculated in the `getAccuracy` method of the *AccuracyMeter* class.

We obtained an accuracy (before smoothing) of 86.28%.

3.5 Step 4

It is a problem for parsers to deal with unseen data in the test corpus. Words or bigrams which do not occur in the training corpus will get assigned a zero probability. When unknown word or bigram is encountered in the test corpus, then, according to our model, the probability of the sentence is equal to zero. This is called the *sparse data problem*, and it always occurs no matter how big the training corpus because of *Zipf's law*.

The solution to this problem is to apply *smoothing*, which means generally to redistribute the probabilities such that zero and very low probabilities get some probability mass at the expense of the higher probabilities.

There are many different smoothing methods, of which we will use the *Good-Turing* method to smooth the lexical model [1].

3.5.1 Good-Turing smoothing

We use the following notation:
r ... frequency of event e (for example a word or a bigram)
n_r ... number of events e with frequency r (i.e. number of different types that occur exactly r times)
N_r ... total frequency of events occurring exactly r times
$N_r = r * n_r$

For event e with frequency r, the Good-Turing estimate r^* is given by:

$$r^* = (r+1)\frac{n_{r+1}}{n_r}$$

The Good-Turing probability estimate is given by

$$P_{GT}(e) = \frac{r^*}{N}$$

So the total frequencies are redistributed, such that the total frequency for unknown words, which are events with frequency $r = 0$, becomes equal to the number of events with a frequency of 1.

$$N_0 = n_0 * 0^* = n_0 * \frac{1 * n_1}{n_0} = n_1$$

It can be shown that the total number of counts is preserved by Good-Turing.

3.5.2 Bucketing

The *Good-Turing estimate* reserves some mass for the unseen events, but it does not tell us how to distribute the mass among the unseen events. For this, we apply a bucketing procedure. We try to divide the unknown words in equivalence classes based on their suffixes, prefixes, whether they are capitalized or any other feature.

The words that occurred only once in the training file are considered to be *rare*. All these words are grouped into *buckets* based on their features, removed from the *lexicon* and replaced by the special `UNKNOWN_XXX` words designed for specific buckets. Now every `UNKNOWN_XXX` word has a higher probability corresponding to the number of *rare words* assigned to the bucket. If the new, unknown, word is encountered during the tagging, it is assigned to its bucket and $P(t|UNKNOWN_XXX)$ is used instead of $P(t|w)$.

3.5.3 Experiments

We distinguished the following features for bucketing:

- words that end with a dot
- words containing only capitals

- words with capitalized first letter
- words containing slash
- words containing dash
- words that start with a digit
- words ending on ""ing" |"ish" | "ity" | "ate" | "ed" |"ly" |"ry" |"ic" | "or" | "al" | "s" |"y" |"e" | "t"

The overall accuracy increased by smoothing to 93.88%. Accuracy of tagging of known words is 94.45%, while the accuracy of unknown words tagging is 78.53%.

4 Disscussion and conclusions

As reported in *Step 1*, the sentences with the highest probability according to the 1st-order Markov model show a repetition of words. This is probably due to the fact that we used a first order Markov model, that does not incorporate long distance dependencies between words.

An inherent property of this model is that longer sentences always end up with lower probabilities, because of the multiplication of probabilities. Therefore, a sentence like *Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990* gets assigned a probability of $1.88 * 10^{-39}$, which is disproportional compared to the probability of $1.07 * 10^{-12}$ for the "sentence" *the the* (*Step 1*).

Results of the *Step 2* indicate, we are not dealing with perfect training corpus. The word *The* is tagged as a verb in one sentence. It was also observed, that tags are used in a little bit different meaning in the training file and in the test file, which ends up in some testing error.

A problem that we encountered in *Step 3* is, that as a consequence of unknown words, all the words that follow the unknown word in the sentence are tagged wrongly. This is explained by the fact that the language model conditions the probability of every tag on the previous tag, and therefore the error is propagated till the end of the sentence.

Smoothing solves this problem and improves the accuracy from 86.28% to 93.01%. Bucketing increases the performance almost by another percent to 93.88%.

We suggest that the results might still be improved by employing a higher order lexical model and higher order language model. Another way might be searching for more features for *bucketing*. For instance we don't use prefixes at the moment. However, there seem to be only few words where the prefix would be significant. And, of course, more sofisticated smoothing algorithms, like Katz smoothing, should give slightly better results. We have also tested smoothing of language model (*Good-Turing*, no bucketing) as it might also improve the performance. However, in our case performance slightly degraded.

As the whole our algorithm reached lower center of the expected accuracy level between 93% and 96%.

References

[1] Joshua Goodman, Stanley F. Chen: *An Empirical Study of Smoothing Techniques for Language Modeling*, (1998)

[2] Khalil Simaan: *Language and Speech processing lectures*, University of Amsterdam, 2005/2006