Computational Linguistics Project Report

Leo Mehr
St. Catherine's College

1 Introduction

This report overviews the theory and work behind creating a Part of Speech (POS) tagger, a practical project for the Oxford Computational Linguistics course (lectured by Stephen Pulman). POS tagging is a useful preprocessing step for further computational linguistic analysis, and provides non-trivial information.

Consider "Time flies". While this sentence has a well-known obvious meaning of time passing quickly, it also may refer to a more bizarre command to use a watch and record the time for athletic insects. Many sentences, even quite simple ones, often have many interpretations, an ambiguity that must be resolved when trying to solve more complicated problems, for example many in Natural Langauge Processing.

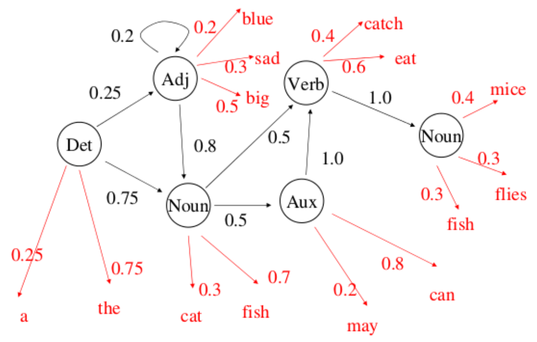
The project consists of two parts. The first covers the construction of a bigram POS tagger.

2 Bigram Hidden Markov Model Tagger

2.1 The Sentence Model

In order to create a Part of Speech tagger that is accurate, we first abstract the problem by creating a model for sentences. Thus is the purpose of the Hidden Markov Model (HMM): a sentence can be represented as the output of a Markov Model (essentially a finite state machine with probabilities for state transitions) that is unknown to us. Our sentence model has states such as Noun, Verb, Adjective, etc., each of which emits a certain word with some probability.

We start with some sample data: a collection of sentences (a corpus) that have been POS tagged by humans. We model that these senteces are the output of a HMM and our task is this: given a sentence find the tag sequence that maximizes the probability that such a sequence in the HMM generated the sentence.



A possible sentence with the above model is "The fish eat[s] flies", emitted with probability $0.75 * 0.75 * 0.7 * 0.5 * 0.6 * 1 * 0.3 \approx 0.035$. Or "A sad blue cat may eat fish", with probability $3.24 * 10^{-6}$. Generally, because words' POS can be ambiguous, the HMM provides the means to determine whether certain taggings are more or less likely than others. In the above example, every word is uniquely associated with a POS. However, it is easy to imagine an HMM that generates the sentence "Time flies" will have a probability distribution on transitions and emissions that makes the labelling "Time/noun flies/verb" much more likely than "Time/verb flies/noun".

More rigorously, given a sequence of words $W = w_1, w_2, \ldots, w_n$, we pick a sequence of tags $T = t_1, t_2, \ldots, t_n$ such that we maximize $P(T|W)$.

Using Bayes' Theorem, we know $P(T|W) = \frac{P(T)P(W|T)}{P(W)}$. $P(T) = P(t_1 \wedge t_2 \wedge \ldots t_n)$ and $P(W|T)$ come from a very large probability space (there are more than 30 tags, so a sentence with 10 words already has at least 30^{10} possiblities). Since $P(x_1 \wedge x_2 \wedge \ldots x_n) = P(x_1|x_2 \wedge \ldots x_3) * P(x_2|x_3 \wedge \ldots x_n) * \ldots P(x_n)$, we can approximate $P(t_1 \wedge t_2 \wedge \ldots t_n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1})$, and similarly $P(w_1 \wedge \ldots w_n|t_1 \wedge \ldots t_n) \approx \prod_{i=1}^{n} P(w_i|t_i)$. Because we approximate by considering only two tags at a time, we call this *bigram* POS tagging.

Taking away the above manipulation, we can see that in order to approximate our HMM, we need only find two groups of probabilities: those of a tag preceded by another, and those of a word given a tag. We obtain these by parsing the hand-tagged corpus.

2.2 Parsing the corpus

The corpus used in this project is provided by the Linguistic Data Consortium (Treebank 3) and requires a [costly] license to obtain. For this reason, it is not publicly accessible with the project. The sentences in the corpus come from Wall Street Journal publications. A rough example of the file format is written out in `test_file.POS`.

`CorpusParser.py` is designed to parse the word-tag pairs in the LDC corpus, which consists of sentences from the Wall Street Journal, and provide two functions we require: $P(word|tag)$ and $P(tag|previous\_tag)$.

A `Word` is an object that represents a word in a stricter linguistic sense: a word that has a certain part of speech. The parsing involves decomposing the format of the .POS file provided in the corpus, and storing the (word,tag) and (tag, previous tag) pairs that occur. A `TagCounter` is an object that makes these counts when called to `parse_corpus()`. It stores hashmaps for (tag,tag) counts and for (word,tag) counts, as well as the individual counts of words and tags. Using the counts, it calculates the two probabilties we need with the functions `p_word()` and `p_tag()`.

2.3 Problems Encountered

When I began parsing the corpus, creating `InvalidWordFormatError` for unusual cases proved quite useful to catch unexpected formatting. Aside from the end tag qualification, the rest of the errors were discovered with this (then appropriately dealt with, making the error obsolete).

When can a Word count as the end of a sentence? The '.' tag clearly marks '.', '!', and '?' as the end of a sentence. However, ')' must also count as an end, as parentheticals are sentences on their own in the corpus.

The parsing must handle any words that contain a '/'. The character is used as syntax in the corpus to distinguish the word, tag pair. Thus, '/'s actually part of a word are indicated with an escape character '\'.

Ocassionally a single word has multiple tags, i.e. in "the/DT data/NN—NNS" 'data' is a considered both a signular or a plural noun. The Wall Street Journal published an article about its acceptance of this treatment.

There are 4 strangely formed word/tag pairs in the entire corpus: 3 instances of "S*/NNP&P/NN" and 1 instance of "AT*/NNP&T/NN". The corpus has over half a million words, I decided instead of making the code uglier with these four edge cases, to simply ignore them.

2.4 Tagging Words

There are 36 word tags used by the Penn Treebank tagset. A sentence with n words has 36^n possible taggings, which is infeasible. The Viterbi Algorithm is a dynamic programming approach that reduces the work to $O(n)$.