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.

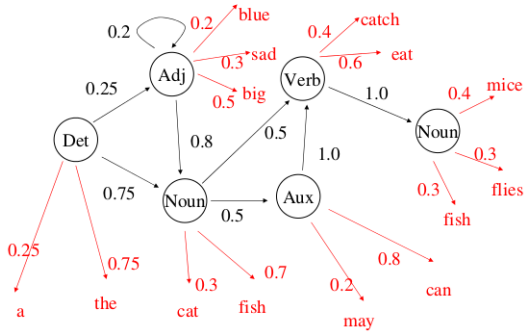
The project consists of two parts. The first covers the construction of a bigram POS tagger. The second describes extending the tagger to guess the POS tags of unknown words using a Naive Bayes classifier.

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 $P(w_1 \wedge \ldots w_n|t_1 \wedge \ldots t_n) \approx \prod_{i=1}^{n} P(w_i|t_i)$. This is the same naive conditional dependence argument that is used in Naive Bayes classification. 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()`. One problem is what probability to assign to a word that has not been seen in the training set. A solution to this is to smooth counts (see Chen and Goodman 1998). The `TagCounter` uses a constant `_delta = 1e-6` added to any observed count.

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

Consider a tagset with k tags. A sentence with n words has k^n possible taggings, which is infeasibly large to exhaust. The Viterbi Algorithm is a dynamic programming approach that reduces the work to $O(nk^2)$. There are $k = 36$ tags used by the Penn Treebank tagset, meaning our work really becomes $O(n)$, which is excellent.

2.5 Speed

The first implementation was rather slow. Running cross validation on 500,000 words took 10 minutes. The first major speedup was to consider only encountered tags for known words. (This reduces the work from $O(nk^2)$ to $O(nk)$ when a sentence consists of entirely known words).

Caching where possible, and making small minor optimizations to the code brought about the second major speed up. After the improvements, the plain tagger is cross-validated for about 12 seconds per fold (totaling just over 2 minutes). The training takes 4-5 seconds per fold (it is quite inefficient because every fold all the counts files are reparsed.)

The next biggest speedup would be to cache the sentences parsed, and to cache the counts for every fold. This is not technically difficult, and would decrease speed, but I shall leave this improvement for some rainy day.

2.6 Results

To test the accuracy of the tagger, I do a 10 fold cross-validation over the data set in two runs: one in which the tagger is trained on all 10 of the combinations of 9/10ths of the data and tested on the remaining 1/10th, and one that is the same, except that the tagger counts the occurances of word/tag pairs from the entirety of the data, and then is tested on a 1/10th chunk. Thus, We have cross validation with some unknown words (10%) and with no unknown words.

- **91.84%** accuracy for some unknown words
- **95.94%** for no unknown words

3 POS Guessing

The "vanilla" tagger uses the Viterbi algorithm, word and tag counting, and constant smoothing, as described in the above sections.

However, we would like to improve the accuracy when tagging a sentence that contains unkown words. This will often be the case in reality, because the amount of tagged data is rather limited, and there will always be new words and - especially - proper nouns that cannot be trained on.

Thus, we create a `WordClassifier` that is a Naive Bayes Classifier, as cited in section 2.1. When training, it extracts features from the observed words (which we must hand-craft) in order to predict the likeliness of an unknown word with certain features to have a particular tag.

3.1 Features

Here are some features that improved performance and have some intuitive explanation. Naive Bayes assumes that features are conditionally independent of one another, and so the ideal features would be such, and would perfectly capture the POS-meaning of a word. Generally, the greater the number of (different and diverse) features the better the classifier will perform. Here are some of the features considered in the word classifier:

1. 1st character uppercase or lowercase: binary feature
2. stem of the word (using a 3rd party stemmer, namely one from NLTK).
3. whether the word ends with an 's': binary feature
4. the final character of the word
5. the first character of the word
6. the length of the word

Some of the features have obvious conditional dependence (1 and 5; 3 and 4; 1,5, and 6 since proper nouns are often longer; etc.). However, every added feature resulted in increased accuracy, except for feature 2, which decreased the tagger accuracy by about 0.5%. Of course, many more features could be constructed and used in the model, but unfortunately increasing the number of features led to much greater runtime.

Let the "standard features" be 1,3,4,5,6. Note that the standard feature space is not so small despite its simplicity, roughly 2x2x26x52x15 = 81000 combinations (since words end in lowercase letters usually, start with uppercase or lowercase, and are not usually longer than 15 characters).

3.2 Results

As defined, the POS guessing does not improve accuracy when testing with no unknown words. Using the standard features from above the cross validation accuracy improved from **91.84%** to **94.24%**.