COMP 182: Algorithmic Thinking

Homework 6: Markov Models and Part-of-Speech Tagging

Luay Nakhleh

The goal of this homework is for you to practice discrete probability as well as to see its use in a real-world application from the domain of natural language processing.

Part-of-speech tagging, or just tagging for short, is the process of assigning a part of speech or other syntactic class marker to each word in a corpus. Examples of tags include 'adjective,' 'noun,' 'adverb,' etc. The input to a tagging algorithm is a string of words and a specified tagset. The output is a single best tag for each word. For example, given the input

The grand jury commented on a number of other topics.

the tagging process would produce

The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS ./.

where a tag follows each word in the sentence (the word and tag are separated by a / symbol). Here, the tags are as follows: DT=determiner, JJ=adjective, NN=noun (singular), VBD=verb (past tense), IN=preposition, NNS=noun (plural), and .=sentence-final punctuation. Notice that tagging the punctuations is also part of the tagging process.

Some tagging distinctions are quite hard for both humans and machines to make. For example, prepositions (IN), particles (PR), and adverbs (RB) can have a large overlap. Words like *around* can be all three:

- Mrs./NNP Shaefer/NNP never/RB got/VBD **around/RP** to/TO joining/VBG
- All/DT we/PRP gotta/VBN do/VB is/VBZ go/VB **around/IN** the/DT corner/NN
- Chateau/NNP Petrus/NNP costs/VBZ **around/RB** 250/CD

In addition to the tags we've already seen, here we have: NNP=proper noun (singular), TO="to", VBG=verg (gerund form), PRP=personal pronoun, VBN=verb (past participle), VBZ=verb (3sg pres), VB=verb (base form), CD=cardinal number.

The problem of POS-tagging is to resolve such ambiguities, choosing the proper tag for the context. Part-of-speech tagging is thus a disambiguation task. How hard is the tagging problem? We saw examples above of some difficulty tagging decisions; how common is tag ambiguity? It turns out that most words in English are unambiguous; that is, they have only a single tag. But many of the most common words of English are ambiguous. For example, *can* can be an auxiliary ('to be able'), a noun ('a metal container'), or a verb ('to put something in such a metal container').

One class of tagging algorithms is stochastic taggers. Such taggers generally resolve tagging ambiguities by using a training corpus to compute the probabilities that constitute the parameters of a statistical model, such as the HMM POS tagger that we will see below. The trained tagger is then applied to a test corpus that is different from the training corpus. The accuracy of the POS tagger is quantified by computing the percentage of all tags in the test set where the tagger and the "true" tags agree. The question, of course, is: How do we know the true tags? One way is to use a human-tagged gold standard test set, where experts do manual tagging of the corpus. Various sources of such data exist (alas, not freely available), such as the Penn Treebank and WSJ Corpus. In this homework assignment, we will develop a stochastic POS tagger based on hidden Markov models (HMMs), which we describe next.

1 An HMM Approach

Use of HMMs to do POS tagging, as we define it here, is a special case of Bayesian inference. In a classification task, we are given some observation(s) and our job is to determine which of a set of classes it belongs to. Part-of-speech tagging is generally treated as a sequence classification task. So, here the observation is a sequence of words (e.g., a sentence), and it is our job to assign them a sequence of part-of-speech tags.

Given a sentence that consists of n words, denoted by w_1^n, we want a sequence of n tags, denoted by t_1^n, such that $P(t_1^n|w_1^n)$ is highest (among all possible sequences of n tags). In other words, we are interested in the estimate

$$\hat{t}_1^n \leftarrow \text{argmax}_{t_1^n} P(t_1^n|w_1^n). \tag{1}$$

Using Bayes' Theorem, this is equivalent to

$$\hat{t}_1^n \leftarrow \text{argmax}_{t_1^n} \frac{P(w_1^n|t_1^n)P(t_1^n)}{P(w_1^n)}. \tag{2}$$

Eq. (2) is equivalent to

$$\hat{t}_1^n \leftarrow \text{argmax}_{t_1^n} P(w_1^n|t_1^n)P(t_1^n). \tag{3}$$

That is, we can drop the denominator from the equation. The two terms in Eq. (3) are the likelihood of the tag string $P(w_1^n|t_1^n)$ and the prior on tag strings $P(t_1^n)$. To conduct POS tagging based on Eq. (3), HMMs make two simplifying assumptions:

Simplifying Assumption 1. The probability of a word appearing depends only on its own part-of-speech tag; that is, it is independent of other words around it and of the other tags around it:

$$P(w_1^n|t_1^n) \approx \prod_{i=1}^{n} P(w_i|t_i). \tag{4}$$

The $P(w_i|t_i)$ terms comprise the emission probabilities of the HMM. Notice that we index the letters of strings and sequences from 1 to n, rather than from 0 to $n-1$. This applies only to the mathematical description. Feel free to index from 0 to $n-1$ in your Python implementation (of course, as long as you maintain correctness!).

Simplifying Assumption 2. The probability of a tag appearing is dependent only on a very small number of tags, rather than the entire tag sequence. In this homework, we will consider two models of such dependence:

- *The bigram model:* The probability of a tag appearing is dependent only on the previous tag in the sequence:

$$P(t_1^n) \approx P(t_1)\prod_{i=2}^{n} P(t_i|t_{i-1}). \tag{5}$$

- *The trigram model:* The probability of a tag appearing is dependent only on the previous two tags in the sequence:

$$P(t_1^n) \approx P(t_1, t_2)\prod_{i=3}^{n} P(t_i|t_{i-2}, t_{i-1}). \tag{6}$$

The $P(t_i|t_{i-1})$ and $P(t_i|t_{i-2}, t_{i-1})$ terms comprise the transition probabilities of the bigram and trigram HMMs, respectively. The terms $P(t_1)$ and $P(t_1, t_2)$ are given by initial probability distributions:

- $\pi^1 = (\pi_1^1, \pi_2^1, \ldots, \pi_N^1)$, where $\sum_{i=1}^{N} \pi_i^1 = 1$, and π_i^1 is the probability that tag t_i appears at the beginning of a sentence. Therefore, $\pi_i = 0$ for some states.
- $\pi^2 = (\pi_{1,1}^2, \pi_{1,2}^2, \ldots, \pi_{N,N}^2)$, where $\sum_{i,j=1}^{i,j=N} \pi_{i,j}^2 = 1$, and $\pi_{i,j}^2$ is the probability that tag bigram t_i, t_j appears at the beginning of a sentence.

1.1 Training the HMM

In this homework assignment, we will consider only training HMMs from labeled data. In this task, we are given a training corpus in which parts-of-speech are labeled, and maximum likelihood estimates of the emission and transmission probabilities are obtained by simple counting. Define the following four counts obtained from a given training corpus:

- $C(t_i, w_j)$: The number of times word w_j is tagged with t_i.
- $C(t_i)$: The number of times tag t_i appears.
- $C(t_{i-1}, t_i)$: The number of times the tag sequence t_{i-1}, t_i appears.
- $C(t_{i-2}, t_{i-1}, t_i)$: The number of times the tag sequence t_{i-2}, t_{i-1}, t_i appears.

Then, the emission probabilities can be estimated using

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}. \tag{7}$$

For bigram transition probabilities, we have

$$P(t_i|t_{i-1}) = \lambda_1 \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} + \lambda_0 \frac{C(t_i)}{NumTokens} \tag{8}$$

and for trigram transition probabilities, we have

$$P(t_i|t_{i-2}, t_{i-1}) = \lambda_2 \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})} + \lambda_1 \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} + \lambda_0 \frac{C(t_i)}{NumTokens} \tag{9}$$

where the coefficients $\lambda_0, \lambda_1, \lambda_2$ satisfy $\lambda_0 + \lambda_1 = 1$ in Eq. (8) and $\lambda_0 + \lambda_1 + \lambda_2 = 1$ in Eq. (9). Algorithm **Compute**λ gives the pseudo-code of a cross-validation based technique for computing the λs for Eq. (9). If in some iteration, the denominator in the calculation of α_i equals 0, then that α_i is set to 0 in that iteration. $NumTokens$ is the total number of tokens in the training corpus. A simple modification to the pseudo-code allows for computing the λs for Eq. (8) (α_i and λ_i are considered only for $i = 0, 1$ and the loop iterates over bigrams t_{i-1}, t_i).

Algorithm 1: Computeλ.

```
Input: A training corpus $D$.
Output: Three coefficients $\lambda_0, \lambda_1, \lambda_2$ for the linear interpolation in Eq. (9).

$(\lambda_0, \lambda_1, \lambda_2) \leftarrow (0, 0, 0)$;
foreach trigram $t_{i-2}, t_{i-1}, t_i$ with $C(t_{i-2}, t_{i-1}, t_i) > 0$ do
    $\alpha_0 \leftarrow (C(t_i) - 1)/NumTokens$;
    $\alpha_1 \leftarrow (C(t_{i-1}, t_i) - 1)/(C(t_{i-1}) - 1)$;
    $\alpha_2 \leftarrow (C(t_{i-2}, t_{i-1}, t_i) - 1)/(C(t_{i-2}, t_{i-1}) - 1)$;
    $i \leftarrow \mathrm{argmax}_i\{\alpha_i : 0 \le i \le 2\}$;
    $\lambda_i \leftarrow \lambda_i + C(t_{i-2}, t_{i-1}, t_i)$;
$(\lambda_0, \lambda_1, \lambda_2) \leftarrow (\lambda_0, \lambda_1, \lambda_2)/(\lambda_0 + \lambda_1 + \lambda_2)$;
return $(\lambda_0, \lambda_1, \lambda_2)$;
```

For the initial probability distributions, π^1 and π^2, we will only use simple counting for tags at the beginning of sentences in the training corpus. You must use the '.' (period) at the end of a sentence as the indication of the beginning of a new sentence. Do not do smoothing of initial distributions.

1.2 Decoding and POS Tagging

Algorithm **Viterbi** that we covered in class (and shown below) can be used to estimate $\hat{t}_1^n$ based on Eq. (3) when the likelihood and prior are approximated using Eq. (4) and Eq. (5), respectively. A modification is needed for the trigram model.

Algorithm 2: Viterbi.

```
Input: A first-order HMM $M$ with states $\mathcal{S} = \{1, 2, \ldots, K\}$ given by its transition matrix $A$, emission
       probability matrix $E$ (alphabet $\Sigma$), and probability distribution $\pi$ on the (initial) states; a
       sequence $X$ of length $L$ (indexed from 0 to $L-1$).
Output: A sequence $Z$, with $|Z| = |X|$, that maximizes $p(Z, X)$.

$v[\ell, 0] \leftarrow (\pi_\ell \cdot E_\ell(X_0))$ for every $\ell \in \mathcal{S}$;
for $i \leftarrow 1$ to $L-1$ do
    foreach $\ell \in \mathcal{S}$ do
        $v[\ell, i] \leftarrow E_\ell(X_i) \cdot \max_{\ell' \in \mathcal{S}}(v[\ell', i-1] \cdot A_{\ell',\ell})$;
        $bp[\ell, i] \leftarrow \text{argmax}_{\ell' \in \mathcal{S}}(v[\ell', i-1] \cdot A_{\ell',\ell})$;
$Z_{L-1} \leftarrow \text{argmax}_{\ell' \in \mathcal{S}} v[\ell', L-1]$;
for $i \leftarrow L-2$ downto $0$ do
    $Z_i \leftarrow bp[Z_{i+1}, i+1]$;
return $Z$
```

1.2.1 Handling Unknown Words

No matter how extensive the training data set is, there is always a good chance that the test data on which annotation (in our case, POS tagging) will be conducted includes words that were not present in the training corpus. To handle this case, and after the training data has been read, you need to do the following (this is function `update_hmm` that you'll be asked to implement):

- For every word w in the test data that is not in the training data, assign a very small emission probability of that word from each state (we find that a probability of $\epsilon = 0.00001$ works). However, to make sure that this new word doesn't have a probability higher than words that were already seen in the test data set, adjust the non-zero emission probabilities of other words from all states by adding the same ϵ to them. After these adjustments, normalize the emission values for each state so that they add up to 1.

While you are not asked explicitly below to write a function to handle unknown words, you must write such a function and call it in the appropriate place in your code.

1.3 Implementation-related issues

You are provided with function `read_pos_file` which reads an input tagged text file `filename`, and returns a tuple `(file_representation, unique_words, unique_tags)`, where `file_representation` is a list of (word, POS-tag) pairs in the input file, `unique_words` is a set of the unique words that appear in the input file, and `unique_tags` is a set of the unique tags that appear in the input file.

It is very important that your code is modular and broken into individual, auxiliary functions (you should look at the other problems in this section before you start the implementation). Further, in your implementation of the Viterbi algorithms below, use log probabilities instead of probabilities (which turns the multiplication of probabilities into addition of log probabilities. This is important for avoiding underflow (multiplication of many small numbers) and speed (addition is faster than multiplication). You must use the `HMM` class, described below, for HMMs.

To represent a Hidden Markov Model in Python, you must define a simple class that contains all of the information about the HMM that you may want to keep track of, such as the order of the model, the initial tag distribution, the end tag distribution, the emission matrix, and the transition matrix. Define an HMM class in your solution file as follows:

```
class HMM:

"""

Simple class to represent a Hidden Markov Model.

"""

def __init__(self, order, initial_distribution,
                 emission_matrix, transition_matrix):

        self.order = order

        self.initial_distribution = initial_distribution

        self.emission_matrix = emission_matrix

        self.transition_matrix = transition_matrix
```

The order of the HMM will be 2 when representing a bigram model, and 3 when representing a trigram model. For a bigram, the initial distribution will be a one-dimensional dictionary where the keys are all possible tags and the value corresponding to a key is the probability that the tag labels the first word of a sentence in the training corpus. For a trigram, this will be a two-dimensional dictionary.

In both the bigram and trigram models, the emission matrix is a two-dimensional dictionary where the first key is the tag and the second key is the word. The values is the emission probability of the word given the tag.

In the bigram model, the transition matrix is a two-dimensional dictionary where the first key is the "previous ta"g and the second key is the "current tag." The representation of the transition matrix for the trigram model is a logical three-dimensional extension of the representation for the bigram model.

Since the transition and emission matrices are sparse (think about why this is the case), it is recommended that you use the `defaultdict` structure to represent your dictionaries. Unlike standard dictionaries, `defaultdict` allows you to assign and access values that have not yet been assigned. You can use a `defaultdict` as follows:

```
from collections import defaultdict

// Initialize a 2D dictionary.
d2 = defaultdict(lambda: defaultdict(int))

// Initialize a 3D dictionary.
d3 = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))

d2[key1][key2] = int_val
d3[key1][key2][key3] = int_val
```

After you read in the training/test data from a file, the text should be represented in Python as a list of tuples, where each tuple is of the form `(word, POS-tag)`.

2 References

Description of the POS tagging problem (including text taken verbatim) is based on

D. Jurafasky and J.H. Martin, 2009. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Second Edition. Pearson, Prentice-Hall.