CMSC 723: Computational Linguistics I

Assignment 3: Let's play tag!

Jimmy Lin (Instructor) and Melissa Egan (TA)

Due: **October 14, 2009**

Introduction

This assignment is about exploring part-of-speech (POS) tagging using n-gram taggers, tagger combination, and hidden Markov models (HMMs). There are a total of three problems; the first requires no programming. This assignment requires the Python modules below:

1. `Matplotlib`: This provides advanced plotting and visualization capabilities that we will need to use for Problem 2.
2. `Numpy`: This provides the efficient multi-dimensional array structure that we will need to use for Problem 3.

A link to installation instructions can be found on the course website under "Software".

Background

In this section, we provide some background on building POS taggers. NLTK ships with a factory of POS taggers that can be easily trained on the included pre-tagged corpora.

There are two main POS taggers that we will use:

1. `DefaultTagger`: This tagger tags *every* word with a default tag. For example, a very good baseline for English POS-tagging is to just tag every word as a noun. Listing 1 shows how to build such a tagger.
2. `NgramTagger`: N-grams over any given sequence can be informally defined as overlapping subsequences each of length N. We will formally define n-grams later in the course. For the purposes of this assignment, the informal definition should suffice. As an example, the sentence *My name is Nitin Madnani* will yield the following n-grams for various values of N:

Listing 1: Building and using a `DefaultTagger`

```
>>> import nltk
>>> t = nltk.DefaultTagger('NN')
>>> sentence = 'This is a sentence'
>>> words = sentence.split()
>>> print t.tag(words)
[('This', 'NN'), ('is', 'NN'), ('a', 'NN'), ('sentence', 'NN')]
```

- N = 1 (**1-grams** or **Unigrams**): *My*, *name*, *is*, *Nitin*, *Madnani*
- N = 2 (**2-grams** or **Bigrams**): *My name*, *name is*, *is Nitin*, *Nitin Madnani*
- N = 3 (**3-grams** or **Trigrams**): *My name is*, *name is Nitin*, *is Nitin Madnani*
- N = 4 (**4-grams**): *My name is Nitin*, *name is Nitin Madnani*
- N = 5 (**5-grams**): *My name is Nitin Madnani*

So, how do we use n-grams for POS-tagging? Figure 1 shows the basic idea behind this strategy. Instead of just looking at the word being tagged, we also look at the POS tags of the previous n words. Therefore, using n-grams allows us to be able to take context into consideration when performing POS-tagging. In the figure, we are using the text of the word itself plus two previous tags, so N=3.

Looking at the figure, it should be easy to see how a `UnigramTagger` (N=1) would work. It would use just the text of the word itself as the *only* context for predicting its POS tag. For example, it might learn that the word *promise* is more likely to be tagged as a verb (*I promise you ...*) than a noun (*It is a promise ...*). Therefore, it would always tag promise as a verb even though that's not always correct! However, if we were to use the previous tag as additional context, our tagger might also learn that if *promise* were preceded by an article (*a*), it should be tagged as a noun instead. Therefore, using larger context is usually a good strategy when building n-gram based POS taggers.

The important thing to realize is that when using an `NgramTagger`, you need to *train* it on some sentences for which you already know the POS tags. This is needed because an `NgramTagger` needs to count and build tables of how many times a particular word is tagged as a verb (when N=1) or how many

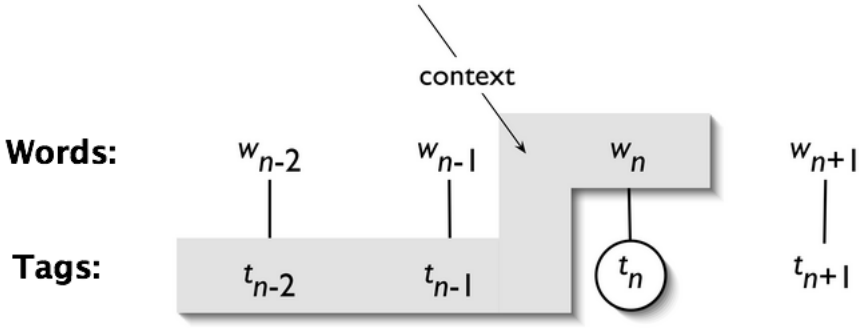

Figure 1: How does an `NgramTagger` work? In this figure, N = 3 (original image from NLTK documentation).

Listing 2: Building and using an `NgramTagger`

```
>>> import nltk
>>> from nltk.corpus import brown
>>> traindata = brown.tagged_sents(categories='reviews')
>>> t = nltk.NgramTagger(n=2, train=traindata)
>>> sentence = 'This is a sentence'
>>> words = sentence.split()
>>> print t.tag(words)
[('This', 'DT'), ('is', 'BEZ'), ('a', 'AT'), ('sentence', None)]
```

times a particular word preceded by a noun is tagged as a verb (when N=2) and so on. In order to build these tables, it requires sentences with the correct tags already assigned to each word.

It's usually a little complicated to build and train an `NgramTagger`. However, NLTK makes it extremely easy. Listing 2 shows how to build and train a bigram tagger (N=2) on the `reviews` category of the Brown corpus.[1] If the tagger cannot make a prediction about a particular word, it assigns that word a null tag indicated by `None`.

[1] The Brown corpus tagset is shown in Figure 5.7 on page 134 of your textbook.

Listing 3: Using the `cutoff` parameter during training

```
>>> import nltk
>>> from nltk.corpus import brown
>>> traindata = brown.tagged_sents(categories='reviews')
# Treat everything as evidence (very noisy)
>>> t = nltk.NgramTagger(n=2, train=traindata, cutoff=0)
```

Restricting Training Evidence

As explained above, training an `NgramTagger` basically entails keeping track of the tag that was assigned to each word for every context that it was seen in and then using that as evidence for making predictions on test data. Now, it's reasonable to think that not *all* evidence should be considered reliable. For example, if a particular piece of evidence occurs only once in the training data, we may not want to rely on it lest it was just an artifact of noise. NLTK allows us to achieve this with the `cutoff` parameter as shown in Listing 3. By default, the value of the `cutoff` parameter is 1, i.e., during training, NLTK will ignore any evidence unless it occurs in the training data at least twice (one higher than the cutoff value). Note that the default cutoff of 1 should be sufficient for this assignment. The point of this section is just to provide information that may be worth having.

Measuring tagger accuracy

Assuming that you have the correct POS tags for the sentences that you wish to test your tagger on, NLTK also provides a simple way to compute how accurate your tagger is in its predictions. Of course, these test sentences should be completely separate from the sentences that are used to train the tagger. Listing 4 shows how to compute the accuracy of a `DefaultTagger` on the `editorial` category of the Brown corpus. On this particular test set, tagging everything as a noun is successful only about 12.5% of the time.

Combining taggers

It's possible to combine two taggers such that if the primary tagger was unable to assign the tag to a particular word, it backs off to the second tagger for the prediction. This is known as **Backoff**. Listing 5 shows how to do this in NLTK.

Listing 4: Measuring the accuracy of a `DefaultTagger`

```
>>> import nltk
>>> from nltk.corpus import brown
>>> testdata = brown.tagged_sents(categories='editorial')
>>> t = nltk.DefaultTagger('NN')
>>> print t.evaluate(testdata)
0.12458606583988052
```

Listing 5: Combining taggers in NLTK

```
>>> import nltk
>>> from nltk.corpus import brown
>>> traindata = brown.tagged_sents(categories='reviews')
>>> t1 = nltk.NgramTagger(n=1, train=traindata)
>>> t2 = nltk.NgramTagger(n=2, train= traindata, backoff=t1)
>>> sentence = 'This is a sentence'
>>> words = sentence.split()
>>> print t2.tag(words)
[('This', 'DT'), ('is', 'BEZ'), ('a', 'AT'), ('sentence', 'NN')]
```

Plotting using Matplotlib

As we have seen in class, the plotting capabilities of NLTK are quite primitive. The Python package `Matplotlib` provides more advanced plotting functions that generate nicer-looking plots. For this assignment, we will only need to know how to make line plots and save them as image files. Listing 6 shows how to create and save a plot. Figure 2 shows the file `plot.png`.

Listing 6: Create and save a simple line plot

```
from pylab import xlabel, ylabel, plot, savefig
x = range(1, 11)
y = [i**3+3 for i in x]
xlabel('x')
ylabel('x^3 + 3')
plot(x, y)
savefig('plot.png')
```

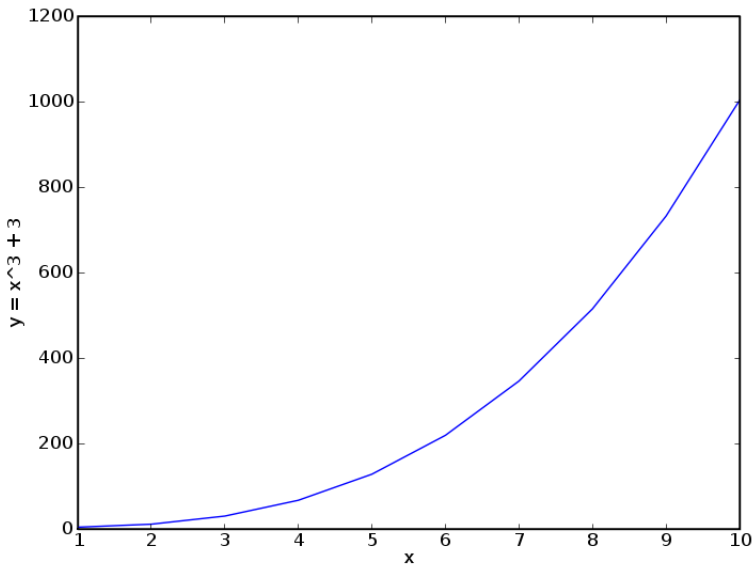

Figure 2: The file `plot.png` as produced by Listing 6.

Problem 1 (10 points)

Recall the constraints used in the EngCG rule-based tagger that we looked at in class. The system is described in more detail in Section 5.4 of your textbook. Say we have the following constraint in our tagger grammar:

if
(-1 has only DT tag)
then remove verb tags

Can you think of two different counter-examples where applying this constraint could lead to a possibly incorrect tagging?

Problem 2 (40 points)

(a) In Listing 2 above, why do you think the bigram tagger could not assign a tag to the word *sentence*? However, in Listing 5, a bigram tagger combined with a unigram tagger was able to correctly predict the tag for the same word. Why do you think that strategy worked?

(b) Create different combinations using a `DefaultTagger` and 3 different n-gram taggers (N = 1, 2 and 3). Use the first 500 sentences of the `news` category of the Brown corpus as the training data. Test each combination on the `religion` category of the Brown corpus. Which combination yields the highest accuracy? Plot the accuracy of the winning combination as the number of sentences used for training increases (by 500 sentences at each step). You need only go up to 4500 sentences.

(c) Let the *coverage* of a test set be defined as the percentage of words that are not assigned a null tag (`None`) by a tagger. Train 6 different n-gram taggers (N= 1...6) on the `news` category of the Brown corpus. Compute the coverage and accuracy of each individual tagger (no combinations) on the `religion` category of the Brown corpus. Explain what happens to the two numbers as N increases.

(d) Note that the contextual information used both by a bigram tagger and a first-order HMM tagger pertains only to the previous word. Does that mean that a trigram tagger will *always* prove to be a better tagger than a first-order HMM? Put another way, does a first-order HMM have any advantages over an n-gram tagger with a much larger N (>= 3)? If so, what are they?

Notes:

- A combination should have at least two taggers. Listing 5 shows you how to combine two taggers. You have to figure out how you would use this method to create a combination of 3 or more taggers.
- Even though there are a large number of possible combinations, you should be able to rule out many of them by thinking about how the individual taggers work and how they can complement each other.
- Your code *should not* enumerate all possible combinations to find the best one. The point of the problem is to ensure that you understand the pros and cons of each tagger enough to come up with combinations that are reasonably good.
- Since POS tagging is sentence oriented, we need to make sure that an `NgramTagger` does not consider context that goes beyond sentence boundaries. The NLTK implementation takes care of this for you.

Problem 3 (50 points)

You are provided with the file `hmm.py` that defines a class called `hmm`. As soon as you instantiate this class, the various parameters of the HMM (transition probabilities, emission probabilities etc.) are automatically computed by using the first 1000 sentences of the `news` category of the Brown corpus as the training data (using functions defined in the supporting file `hmmtrainer.py`). The following five parameters are available to each instance of the `hmm` class:

- `transitions`: The probabilities of transitioning from one state to another. To get the probability of going to state `s2` from state `s1`, use `self.transitions[s1].prob(s2)`.
- `emissions`: The probability of emitting a particular output symbol from a particular state. To get the probability of emitting output symbol `sym` in state `s`, use `self.emissions[s].prob(sym)`.
- `priors`: The probability of starting in a particular state. To get the probability that the HMM starts in state `s`, use `self.priors.prob(s)`.
- `states`: The states (tags) in the trained HMM.
- `symbols`: The output symbols (words) in the trained HMM.

Listing 7: Using multi-dimensional arrays

```
>>> from numpy import zeros, random, max, argmax, float32

# Create a 10x10 two-dimensional array initialized to zeros
# Must use float32 to indicate 32-bit floating point precision
>>> a = zeros((10, 10), float32)

# add 0.5 to all elements
>>> a += 0.5

# element at row 1 and column 1 (zero-indexed)
>>> a[0,0]
0.5

# add 1.0 to each element in the 6th column
>>> a[:,5] += 1.0

# create a 5x5 two-dimensional array with each element x
# randomly generated such that 0 <= x < 10
>>> b = random.randint(0, 10, (5, 5))

# find the largest element in the 5th column
>>> max(a[:,4])
9

# find the row number in which this maximum occurred
>>> argmax(a[:,4])
3
```

For this problem, implement the following:

(a) Add a `decode()` method to the class that performs Viterbi decoding to find the most likely tag sequence for a given word sequence.

(b) Add a `tag()` method that takes a sentence string as input and tags the words in that sentence using Viterbi decoding. It should have an output of the form: *This/DT is/BEZ a/AT sentence/NN.*

(c) Tag each of the six sentences in the provided file `given.sentences`. Do you see any errors in the tags assigned to each sentence? If so, mention them.

Turn in the file `hmm.py` that implements the items above. Your program should accept sentences from stdin and print the tagged results to stdout. We will test your program with the following command-line invocation:

```
python hmm.py < given.sentences
```

Make sure your solution behaves exactly in this manner.

Notes:

1. The Viterbi decoding algorithm requires a two-dimensional trellis or chart. It is extremely tedious to use Python lists to implement such a chart. This is where the efficient and versatile `array` datatype provided by `Numpy` comes in. Listing 7 should tell you how to create, initialize and use a two-dimensional array. You should use such an array to implement the chart you need for decoding.

2. The probability values that are calculated by the trainer are going to be extremely small in scale. Multiplying two very small numbers can lead to loss of precision. Therefore, we strongly recommend that you use the log of the probabilities (logprobs) instead. To compute the log of the transition probability of going from `s1` to `s2`, use `self.transitions[s1].logprob(s2)` instead, and so on.

3. You do *not* need to lowercase the training data. Use the words as they occur in the data.

4. You do *not* need to worry about any words that are not seen in the training data. The probability distributions that the `hmmtrainer` module computes are all *smoothed*, which means that it assigns some non-zero probability mass to *every* event whether or not it was observed

in the data. In general, assigning a zero probability to any event is not a good idea when building statistical models. This has an intuitive reason: just because you don't observe an event in your limited view of the world (as represented by the training data) doesn't mean that it never happens in the real world (which is what assigning it zero probability says). We will delve deeper into the technical details of smoothing later in the semester. For this problem, just know that you don't have to do anything special about unseen words.