LING 227/627 — Language and Computation I
Fall 2017

Problem Set 4: Hidden Markov Models
Due 11/15/17 at midnight

Part 1: Warming up with NLTK

The first part of the homework will get you used to training HMM part of speech taggers using the NLTK toolkit, and estimating some relevant probability distributions. For this part, you will be working in the file `Homework04-nltk.py` (available for download on the website). This file provides templates for the various functions you will need to write. All of your code should go there, and your (brief) answers to the non-programming questions should be put as comments in your code.

Before beginning, you may find it helpful to look at the help pages about the tools that NLTK provides relating to HMM taggers by typing `help(nltk.tag.hmm.HiddenMarkovModelTagger)` within Python.

As we have discussed, NLTK provides corpora annotated with part-of-speech (POS) information and some tools to access this information. The Penn Treebank tagset is commonly used for annotating English sentences. We can inspect this tagset in the following way:

```
>>> nltk.help.upenn_tagset()
```

Loading `Homework04-nltk.py` will show you some information about this, as well as some tagged sentences from the corpus we will be working with.

The Brown corpus provided with NLTK is also tagged with POS information, although the tagset is slightly different than the Penn Treebank tagset. Information about the Brown corpus tagset can be found at `https://en.wikipedia.org/wiki/Brown_Corpus#Part-of-speech_tags_used`. We can retrieve the tagged sentences in the Brown corpus by calling the tagged sents() function and looking at an annotated sentence:

```
>>> tagged_sentences = brown.tagged_sents(categories= 'news')
>>> print tagged_sentences[29]
```

Sometimes it is useful to use a coarser label set in order to avoid data sparsity or to allow a mapping between the POS labels for different languages. The Universal tagset was designed to be applicable for all languages, and is described at `http://universaldependencies.org/u/pos/`.

Mappings have been defined between the POS tagsets used for a variety of languages and the Universal tagset (on which see `http://universaldependencies.org/tagset-conversion/`). We can access the Universal tags for the Brown corpus sentences by changing the tagset argument:

```
>>> tagged_sentences_universal = brown.tagged_sents(categories= 'news', tagset='universal')
>>> print tagged_sentences_universal[29]
```

Problem 1: Corpora and POS Tags

We will start off by computing a Frequency Distribution over tags that appear in the Brown corpus. The template of the function that you have to implement takes two parameters: one is the category of the text and the other is the tagset name. You are given the code to retrieve the list of (word, tag) tuples from the brown corpus corresponding to the given category and tagset.

a. Convert the list of word+tag pairs to a list of tags

b. Use the list of tags to compute a frequency distribution over the tags, using `FreqDist`

c. Compute the total number of tags in the Frequency Distribution

d. Retrieve the top 10 most frequent tags. (Hint: remember that given a `FreqDist` object `fdist`, you can get the n most frequent using the command `fdist.most_common(n)`.)

Uncomment the test code. What do you observe by comparing the number of tags and most frequent tags across different genres? What happens when you change the tagset?

Problem 2: Training an HMM Tagger

Next, we will train a HMM tagger on a training set and evaluate it on a test set. The template of the function that you have to implement takes two parameters: a sentence to be tagged and the size of the training corpus in number of sentences. You are given the code that creates the training and test datasets from the tagged sentences in the Brown corpus.

a. Train a Hidden Markov Model tagger on the training dataset. You may find it helpful to refer to the `help(nltk.tag.hmm.HiddenMarkovModelTagger.train)`.

b. Use the trained model to tag the sentence

c. Use the trained model to evaluate the tagger on the test dataset

Uncomment the test code. Look at the tagged sentence and the accuracy of the tagger. How does the size of the training set affect the accuracy?

Problem 3: Estimating Emission and Transition Probabilities

In preparation for Part 2 of the homework, you will now work on estimating the parameters of the emission model. In order to estimate the Conditional Probability Distribution of $P(word|tag)$, we first compute the Conditional Frequency Distribution of a word given a tag. You will find the following functions useful to do this:

```
>>> help(nltk.probability.ConditionalFreqDist)
>>> help(nltk.probability.ConditionalProbDist)
```

The `ConditionalFreqDist` function takes as input a list of tuples, each tuple consisting of a condition and an observation, and returns a `ConditionalFreqDist` object. For the emission model, the conditions are tags and the observations are the words. The template of the function that you have to implement takes as argument the list of tagged words from the Brown corpus.

a. Build the dataset to be passed to the `ConditionalFreqDist` function. Words should be lowercased. Each item of data should be a tuple of tag (a condition) and word (an observation).

b. Compute the Conditional Frequency Distribution of words given tags.

c. Return the top 10 most frequent words given the tag NN. (Hint: `most_common` works here too)

d. Compute the Conditional Probability Distribution for the above Conditional Frequency Distribution. Use the `ConditionalProbDist` function to do this: it takes as its first argument a Conditional Frequency Distribution, and as its second argument an estimator. Use the `MLEProbDist` estimator.

e. Compute the probabilities $P(year|NN)$ and $P(year|DT)$

Now, uncomment the test code. Look at the estimated probabilities. Why is $P(year|DT) = 0$? What is `emission_FD['NN']['year']`? Contrast that with `emission_FD['DT']['year']`?

Next, estimate the transition model, i.e., the Conditional Probability Distribution of $P(tag_{i+1}|tag_i)$. To do this, we first compute the Conditional Frequency Distribution of a tag at position $i+1$ given the previous tag. This time the pairs that are given to the `ConditionalFreqDist` function consist of a condition, which is a tag at position i, and an observation, a tag at position $i+1$. The template of the function that you have to implement takes as argument the list of tagged sentences from the Brown corpus.

a. Build the dataset to be passed to `ConditionalFreqDist()`. Each item in your data should be a pair of condition and observation: (tag_i, tag_{i+1})

b. Compute the Conditional Frequency Distribution of a tag at position $i+1$ given the previous tag at position i.

c. Compute the Conditional Probability Distribution for the above Conditional Frequency Distribution. Use the `MLEProbDist` estimator when calling the `ConditionalProbDist` (as its second argument).

d. Compute the probabilities $P(NN|VBD)$ and $P(NN|DT)$

Uncomment the test code. Are the results what you would expect? The sequence 'DT NN' seems very probable. How will this affect the tagging of real, longer, sequences?

Part 2: Building your own HMM

In this part of the assignment, you will build your own function to train an HMM and decode it it using the Viterbi algorithm. We are providing you with a template file `Homework04-hmm.py` that provides access to training and test data for the part of speech tagging task. This file uses Python's support for object-oriented programming to define an HMM class and a number of methods that can be applied to objects of this class. If you are unfamiliar with object-oriented programming, do not worry: you will be able to use the template functions that are provided to code up functions essentially as you have in the past.

If you run this file, by typing `python Homework04-hmm.py` at the command line, the main function (defined at the bottom of the file) will be executed and will create an HMM Object to which the training and test data are passed as arguments. The data consists of sentences that are tagged with the Universal POS tagset. As you complete portions of your code, you will need to move down the `exit()` line that appears inside of the `main()` function, so that the functions you have implemented get executed.

Problem 1: Estimate the Emission model

In this problem you will estimate the emission mode, by modifying the relevant portion of the code inside of the function `emission_model(self,train_data)`. This function takes the training data (a list of lists of tagged sentences) as its argument. The `self` argument that is also listed (but is not used when the function is used as a method for an HMM object) allows you to reference the HMM object itself from inside the function. An HMM object (as defined in the `__init__` function) has a number of properties, including:

- `self.train_data`: the training data used to train this HMM
- `self.test_data`: the test data used to test this HMM
- `self.states`: the states of the HMM represented as a list. For POS tagging, this will be the set of POS tags.

- `self.viterbi`: the trellis data structure that includes the Viterbi probabilities.
- `self.backpointer`: the backpointer data structure that allows for the decoding of the trellis.

You can access or set any of these values inside of the definition of the methods for the HMM object.

Be sure to lowercase all the observations (words) while you are preparing the data, and then use `ConditionalProbDist` with the estimator `LidstoneProbDist` and 0.01 added to each sample count each bin (these can be given as the second and third arguments to `ConditionalProbDist`). Store the emission model in the variable self.emission_PD. Finally, save the states (types) that were seen in training in the variable `self.states`.

Problem 2: Estimate the Transition model

Now, fill in the function `transition_model(self,train_data)` from the HMM class to estimate the transition model. In preparing the training data for the parameter estimation, be sure to add a start state `<s>` and an end state `</s>` to the beginning and the end of each sentence. Again use `ConditionalProbDist` with the estimator `LidstoneProbDist` with 0.01 added to the sample count for each bin. Finally, store the transition model in the variable `self.transition_PD`.

Interlude: Testing the model and Next Steps

The file we provide provides two functions to test the last 2 problems: `test_emission(self)` and `test transition(self)`. These tests are called from the main function (once the `exit()` line has been moved down). The expected output for these functions is written as a comment in the template. Please check your solutions with the test functions. Consider this a sanity check for your code.

The remaining portions of this assignment will ask you to implement the Viterbi algorithm that we discussed in class. The pseudo-code for the algorithm can be found in figure 10.8 of chapter 10 of the online 3rd edition draft of *Speech and Language Processing* or in the 2nd edition book in chapter 5, Figure 5.17. Follow the pseudo-code to guide your implementation.

In the pseudo-code, the `b` probabilities correspond to the emission model and the `a` probabilities correspond to the transition model. Throughout your implementation, you should use costs (i.e., negative log base 2 probabilities) rather than the probabilities themselves. Therefore, rather multiplying probabilities (as is shown in the pseudo code), you should add costs. Furthermore, where the pseudo-code uses `max`, you should use `min`.

Problem 3: Implementing the Viterbi Algorithm: Initialization

Implement the initialization step of the algorithm, by filling in the function `initiatlize(self,observation)`. The argument `observation` will be the first word of the sentence to be tagged (this is called o_1 in the pseudo-code). The algorithm uses two data structures that have to be initialized for each sentence that is being tagged: the `viterbi` data structure and the `backpointer` data structure. Think carefully about how you should implement these. (One possibility is a `defauldict`, which you may have used in the previous assignment.) Describe your implementation of these data structures in your comments. Finally, remember to use costs rather probabilities when initializing the `viterbi` data structure.

Problem 4: Implementing the Viterbi Algorithm: Recursion and Termination

Implement the recursion and termination steps of the algorithm, by filling in the function `tag(self,observations)`. The argument `observations` is a list of words representing the sentence to be tagged. Remember to use costs. The two loops of the recursion step have been provided in the template. Fill in the code inside the loops for the recursion step. Fill in the code for the termination step outside the loops. Describe your implementation with comments. Reconstruct the tag sequence corresponding to the best path using the backpointer structure.

The template provides a test for this part of the assignment: accuracy over the test set is computed using your implementation of the `tag` function. The test data is tagged with the Universal tagset. These test is called from the `main` function. The expected output for the functions is written as a comment in the template. Please use this test as a sanity check for your code.

That's it!

You submission should be in the form of a single archive containing your updated versions of the two files `Homework04-nltk.py` and `Homework04-hmm.py`). Your prose answers to the questions in the problem set should be given in comments in the code.