Homework 7

The goal of this homework is to implement an HMM tagger.

HMM module

The file `hmm.py` is provided to give you a leg up. Simply copy it into the local directory so that you can import from it.

Model

The `hmm` module contains the class `Model` and an example model, which you can view like this:

```
>>> from hmm import example_model
>>> example_model.display()
```

One can access the transition matrix as follows. The value of `tprob(x,y)` is the probability of y given x, and `tcost(x,y)` is the corresponding cost. The value `None` is used for beginning or end of sentence. For example, the probability of beginning a sentence with `NNS` is:

```
>>> example_model.tprob(None, 'NNS')
0.75
>>> example_model.tcost(None, 'NNS')
0.12493873660829995
```

The emission matrix is accessed similarly. The value of `eprob(t,w)` is the probability of word w given tag t, and `ecost(t,w)` is the corresponding cost.

```
>>> example_model.eprob('NNS', 'dogs')
0.4166666666666667
>>> example_model.ecost('NNS', 'dogs')
0.38021124171160603
```

Finally, the method `parts` returns the list of parts of speech for a given word, sorted alphabetically.

```
>>> example_model.parts('dogs')
['NNS', 'VB']
```

Node

The `hmm` module also contains the class `Node`, and the function `print_graph`. The first step in tagging is to build the graph, and the graph is represented simply as a list of `Node` instances. A `Node` has the following members:

- `index` is the position of this node in the list of nodes. The first node created has index 0, the next index 1, and so on.
- `i` is the position of this node's word in the sentence. The left boundary node has `i = -1`, and `i` for the right boundary node is the length of the sentence.
- `word` is the word (string) associated with this node. It has value `None` for the boundary nodes.
- `pos` is the part of speech that this node assigns to its word. Recall that every node represents the pairing of a particular word token with a particular part of speech. Boundary nodes have `pos = None`.
- `prev_nodes` is a list of nodes, containing the nodes whose value for `i` is one less than this node's value for `i`.
- `score` is to be filled in by the tagger. It represents the cost of the best partial path leading up to this node.
- `best_prev` is to be filled in by the tagger. It contains the previous node along the best partial path leading up to this node. Its value is a `Node` instance.

The `Node` constructor takes five arguments, representing the values of the first five members: `index`, `i`, `word`, `pos`, and `prev_nodes`. For example:

```
>>> from hmm import Node, print_graph
>>> n0 = Node(0, 0, ’the’, ’DT’, [])
>>> n1 = Node(1, 1, ’dog’, ’NN’, [n0])
>>> n2 = Node(2, 1, ’dog’, ’VB’, [n0])
>>> print_graph([n0, n1, n2])
Graph:
     ind   i word       pos   prevs    bp   score
    ---------------------------------------------
    [  0]  0 the        DT
    ---------------------------------------------
    [  1]  1 dog        NN    0
    [  2]  1 dog        VB    0
```

Assignment

In the file `hw7.py`, you should define a class called `Tagger`. The following questions specify its methods.

1. The `__init__` method should take an HMM model as input, and simply save it in the member `model`. Also, define a `reset` method that takes a list of words, stores them in the member `words`, and sets the member `nodes` to the empty list.

```
>>> tagger = Tagger(example_model)
>>> tagger.model.tprob(None, 'NNS')
0.75
>>> tagger.reset(['dogs', 'bark', 'often'])
>>> tagger.words
['dogs', 'bark', 'often']
```

2. The first step in tagging is to build the graph, represented as a list of `Node` instances. We break the process of building the graph into two pieces.

a. First, define a helper method called `new_node`. It takes four arguments: `i`, `word`, `pos`, and `prev_nodes`. It should create a new `Node` instance and append it to the tagger's list of `nodes`. Note: in order to create the `Node` instance, you will need to figure out what its index will be. The following should always be true:

```
tagger.nodes[k].index == k
```

The return value is the new node.

b. Then define a method called `build_graph`. It takes a sentence (a list of word tokens) as input, and does the following. First, it creates the left boundary node, which should always have index 0. Then it iterates through the words of the sentence. For each word, it uses the model to get the list of possible parts of speech. For each part of speech, it creates a separate node. Finally, after creating nodes for all words in the sentence, it creates the right boundary node, which should always be the last node in `tagger.nodes`.

Note 1: you will need to keep track of the list of previous nodes. For the left boundary, it is the empty list. For the first word in the sentence, it is the list containing just the left boundary node. You will need to update it appropriately for each word in the sentence: the "previous nodes" for the next iteration are the nodes you create in this iteration.

Note 2: to create a node, you need to know its word's position in the sentence. When iterating through the sentence, you will actually want to iterate through the word *indices.*

Continuing our example:

```
>>> tagger.build_graph()
>>> print_graph(tagger.nodes)
Graph:
     ind   i word       pos   prevs    bp   score
    ---------------------------------------------
    [  0] -1
    ---------------------------------------------
    [  1]  0 dogs       NNS   0
    [  2]  0 dogs       VB    0
    ---------------------------------------------
```

```
    [  3]  1 bark       NNS   1,2
    [  4]  1 bark       VB    1,2
    ---------------------------------------------
    [  5]  2 often      RB    3,4
    ---------------------------------------------
    [  6]  3                  5
```

3. The heart of the algorithm is `score_node`, which computes the score for a given node. It should consider each preceding node in turn, and choose the one that gives the best score. According, we first define the helper method `edge_score`, which is given the node and one of its preceding nodes.

a. Define the method `edge_score`. It is given a pair of nodes as input, `prev` and `next`, such that `next.i == prev.i + 1`. You may assume that `prev.score` is known, but `next.score` is not yet known. Recall that `prev.score` represents the cost of the best partial path leading up to `prev`. Compute and return the cost of the best partial path that *goes through* `prev` and leads up to `next`.

For example, let us manually set up the predecessors for node 3:

```
>>> (n1, n2, n3) = tagger.nodes[1:3]
>>> n1.score = .9
>>> n2.score = .1
>>> tagger.edge_score(n1, n3)
2.0583624920952497
>>> tagger.edge_score(n2, n3)
1.01204482964487
```

b. Define the method `score_node`. It is given a `node` as input. You may assume that the score has already been computed for all nodes in `node.prev_nodes`. Set `node.score` to the score of the best path leading up to `node`, and set `node.best_prev` to the predecessor that the best path passes through.

Continuing our example:

```
>>> tagger.score_node(n3)
>>> n3.score
1.01204482964487
>>> n3.best_prev
<Node 2>
```

c. Define a method `score_graph`. It takes no input and returns no value, but it should compute the score for each node in the graph. To start things off, it will need to set the score for the left boundary node appropriately.

```
>>> tagger.score_graph()
>>> print_graph(tagger.nodes)
Graph:
```

```
 ind   i word       pos   prevs    bp   score
---------------------------------------------
[  0] -1                              0.0000
---------------------------------------------
[  1]  0 dogs       NNS   0         0  0.1249
[  2]  0 dogs       VB    0         0  0.9031
---------------------------------------------
[  3]  1 bark       NNS   1,2       1  1.2833
[  4]  1 bark       VB    1,2       1  0.8854
---------------------------------------------
[  5]  2 often      RB    3,4       4  1.8766
---------------------------------------------
[  6]  3                  5         5  2.1776
```

4. The last step is to read the tags off of the graph. Define a method called `unwind`. It takes no input, but it returns the list of tags in the best path through the graph. Start at the right boundary node, and follow `best_prev` links backward through the graph. Return the resulting list of tags. Make sure that the order is correct, so that the tags line up with the words of the sentence. For our running example:

```
>>> tagger.unwind()
['NNS', 'VB', 'RB']
```

5. Now package everything up. We will make the tagger be callable, like a function. We can do that by using the special name `__call__` for the method that puts the pieces together. It should take a sentence (list of word tokens) as input and return a tagged sentence as output. Represent a tagged sentence as a list of pairs (w, t) where w is a word of the original sentence and t is the part of speech assigned by the tagger. For example:

```
>>> tagger.__call__(['dogs', 'bark', 'often'])
[('dogs', 'NNS'), ('bark', 'VB'), ('often', 'RB')]
```

Python will automatically use the `__call__` method if you simply use the tagger as a function:

```
>>> tagger(['dogs', 'bark', 'often'])
[('dogs', 'NNS'), ('bark', 'VB'), ('often', 'RB')]
```

Note: the function `zip` will take two lists of the same length and generate pairs of corresponding elements. For example:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))
>>> [('a', 1), ('b', 2), ('c', 3)]
```

6. *Further exploration.* (This question is optional and will not be included in the grading.) To train a new model, import `Model` from `hmm` and do

```
model = Model(train)
```

where `train` is a list of tagged sentences: for example, the `train` of Handout 11 #5e.

a. Train a model on the training data of Handout 11 #5e and try tagging some more interesting sentences.

b. Compare performance to the taggers that we built in the first half of Handout 11. To do this, you will need to do two things. First, you will need to apply the `FreqFilter` from HW 6 to the training and testing data. Next, you will need to write an `evaluate` function. It should take two sets of tagged sentences: the *system* output (that is, the output of the tagger) and the *gold* annotation (the manually labeled data). It should compare the tags, and count up the number of times that the system and gold tags match. (It should ignore the words: if you use `FreqFilter`, the words will not match.) The return value is the proportion of words that are correctly tagged.