Write a Bigram Part of Speech tagger.

Note: there is a bigram tagger package in NLTK. This assignment asks you to write this from scratch. You may look at the text book for guidance about how to write this code, but don't simply reproduce NLTK or other Tagging code for this assignment.

1. [0] Train using the Brown corpus as found in NLTK. (cf: http://www.nltk.org/book/ch05.html about how to read tagged corpura. this will make your life easier.) You may use the 'universal' tagset or the 'original' tagset, but report which you are using.

2. [5] Construct a training and testing set from the Brown Corpus (section 5.2 on the above page will demonstrate a reasonable way to do this.) The training set should be 90%, and the test 10%. If you'd like to use a tuning set (for smoothing, or other experimentation), this should be 80/10/10 training/tuning/testing.

3. [20] Train the transition and observation probabilities of the HMM tagger directly on tagged data.
   
   Store these probabilities in some way that can be used by the program used in Question 4.

4. [40] Write a program postag that reads a space delimited sentence from the command line and outputs a sequence of tags, separated by spaces. No other information should be printed beyond the tag sequence. This will require you to implement the Viterbi algorithm from J&M2 Chapter 5 and Chapter 6 so that you can decode an arbitrary test sentence.

   Note that because the sentence is delimited, you don't need to worry about tokenizing, abbreviations, or punctuation. It is your decision about how to handle capitalization — the input data may include both upper and lowercase words.

Sample input:

colorless ideas sleep furiously .

Sample output:

ADJ N V ADJ .

You may want or need to smooth the transition or observation probabilities for your tagger. This is a design choice that you can make at your discretion.
The grading breakdown of this section is as follows:

* (10) Correctness - does the program appropriately respond to user input? This is a difficult problem. It is not expected that your program will work perfectly on all input. The grading of "correctness" will be based on whether reasonable output it produced rather than the accuracy of your approach. Make sure that the README file describes your code and any unexpected techniques used — this will facilitate fair grading of this component.

* (10) Functionality - does the program work as described? Does the program compile correctly and without error?

* (5) README - Does the attached README file include all appropriate documentation about how to run and possibly compile your work. Any other elements about interesting or important elements of the code should be highlighted here.

* (5) Within code documentation - Are all functions commented to describe what they do and appropriate inputs and outputs? Are non-trivial sections of code appropriately documented?

* (5) Style - is the code well written? Do functions and variable names have meaningful names? are spacing and other stylistic aspects consistent?

* (5) Grader’s Discretion - Does the assignment go "above and beyond" in some way, either by using a more sophisticated smoothing or tagging style, especially well written and designed code, or some other feature not required by the assignment.

6. [5] Compare this error rate to the 'most frequent tag' baseline. This is the baseline where each word is tagged by its most frequent tag in the training set. There may be words in the test set that do not appear in the training data. Make a decision about how to tag these words.

7. [20] Open ended question: Build a confusion matrix and investigate the most frequent errors produced by your tagger. Describe these specific confusion rates (i.e. which tag is frequently misrecognized as which other tag) and the contexts in which some of these mistakes occur. Propose some features or techniques for improving the performance of your tagger on these errors.