Part-of-Speech Tagging using the Viterbi Algorithm

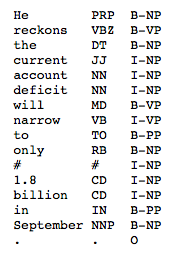
For assignment, you will write a program to extract bigram and lexical generation probabilities from a text file that contains a training corpus of tagged sentences, and then use this data to tag words in sentences in an input file with the appropriate part-of-speech tag using the Viterbi algorithm, which we have discussed in lecture. The corpus is furnished below. The input file to be tagged will be specified as a command line argument. The required output is described in detail below. The program must be written in the Java programming language.

Training Corpus

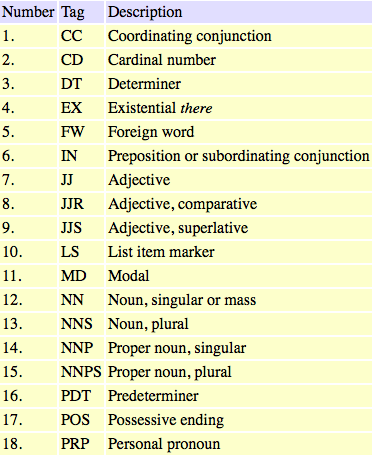
The text corpus used for this assignment will be the CoNLL 2000 text chunking corpus, which you can download from [here](https://webcourses.ucf.edu/courses/1134955/files/49148349/download?wrap=1).  You must unzip the file in order to use it.

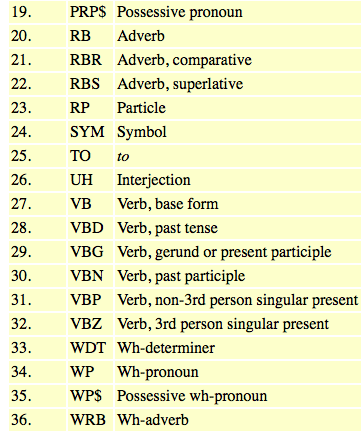
The file contains test data in three columns.  Each word has been put on a separate line and there is an empty line after each sentence. The first column contains the current word, the second its part-of-speech tag as derived by the Brill tagger and the third its chunk tag as derived from the WSJ corpus. We will not use the chunk information for this assignment. It should be ignored.

Here is an example of the file format:



The part-of-speech tags that appear in the corpus are from the Penn Treebank tagset:





There are additional tags that will appear, such as for punctuation and special characters in the corpus extract above.

Corpus Analysis

Your program must analyze the corpus to obtain the following probability data:

(a) bigram probabilities of the form:  p( tag2 | tag1 ).  For example p( NN | JJ ) is the probability that a common noun follows an adjective; and

(b) lexical generation probabilities of the form:  p( *word* | tag ).  For example, p( report | NN ) is the probability that the word "report" appears in the corpus with the tag "NN". Many words will have more than one lexical generation probability. For example, p( report | VB ) may also appear in the corpus.

**NOTE 1:**  Any transition probability that is not observed must be given the nonzero value of 0.0001.

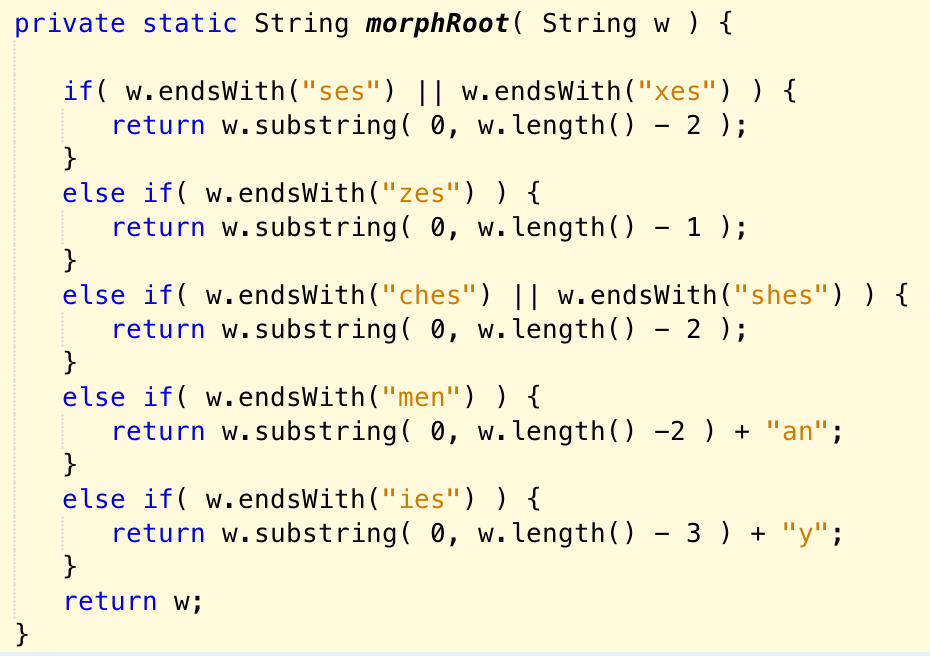
**NOTE 2:**  The first tag in the file, plus all tags appearing immediately after the blank lines that separate sentences, will produce bigrams of the form p( tag | {} ), that is, with an empty prior tag.

Required Corpus Cleansing

Although the corpus is larger than would be convenient to list, it is still small for natural language processing purposes. Therefore, in order to introduce some additional ambiguity into the data, the following steps must be taken when analyzing the corpus:

(a) all words should be converted to lower case before analysis, even proper names; and

(b) certain suffixes should be removed or simplified. It is required that the removal follow what is shown in the following Java method:



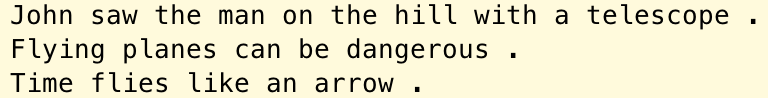
It is not required that you use the method, although you are free to do so, so long as you implement its functionality.

Test Data

The test data for this assignment will be contained in a text file. The name of the text file must be furnished as a command line argument to the program.

The file will contain some sample sentences, one to a line. The sentences will be composed of tokens, which may be words, numbers, punctuation, or special characters. Your program will not need to do any preprocessing on the input data, except for cleansing, as described in the note below.

Punctuation will be separated, as shown in this sample test file:

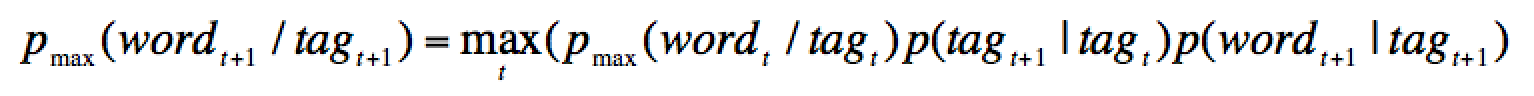


There may be extra empty lines at the bottom of the file. Your program should ignore such lines.

**IMPORTANT:**  The test data must be cleansed in the same manner as the corpus words before processing. However, in the outputs described below, the original words, with any upper case letters and/or original suffixes, must be displayed.

Viterbi Algorithm

The Viterbi algorithm proceeds by examining each token in the input sequence, and for each token it computes the maximum total probability of the path, starting from the no-input state, to each of the nonzero possible tag states for the token. Each possible state is a combination of a token and a part-of-speech tag value. The possible tag states are those that are specified by the lexical generation probabilities that were obtained from the training corpus. The value at each such tag state is the maximum value produced by any prior tag state that can validly transition to the current tag state according to the bigrams extracted from the corpus, computed as follows:



where: (i) the first term on the right-hand side is the maximum total probability of a path to a prior state that has a valid transition to the current tag state; (ii) the second term is the bigram transition probability for transition; and (iii) the third term is the lexical generation probability for the particular word/tag combination.

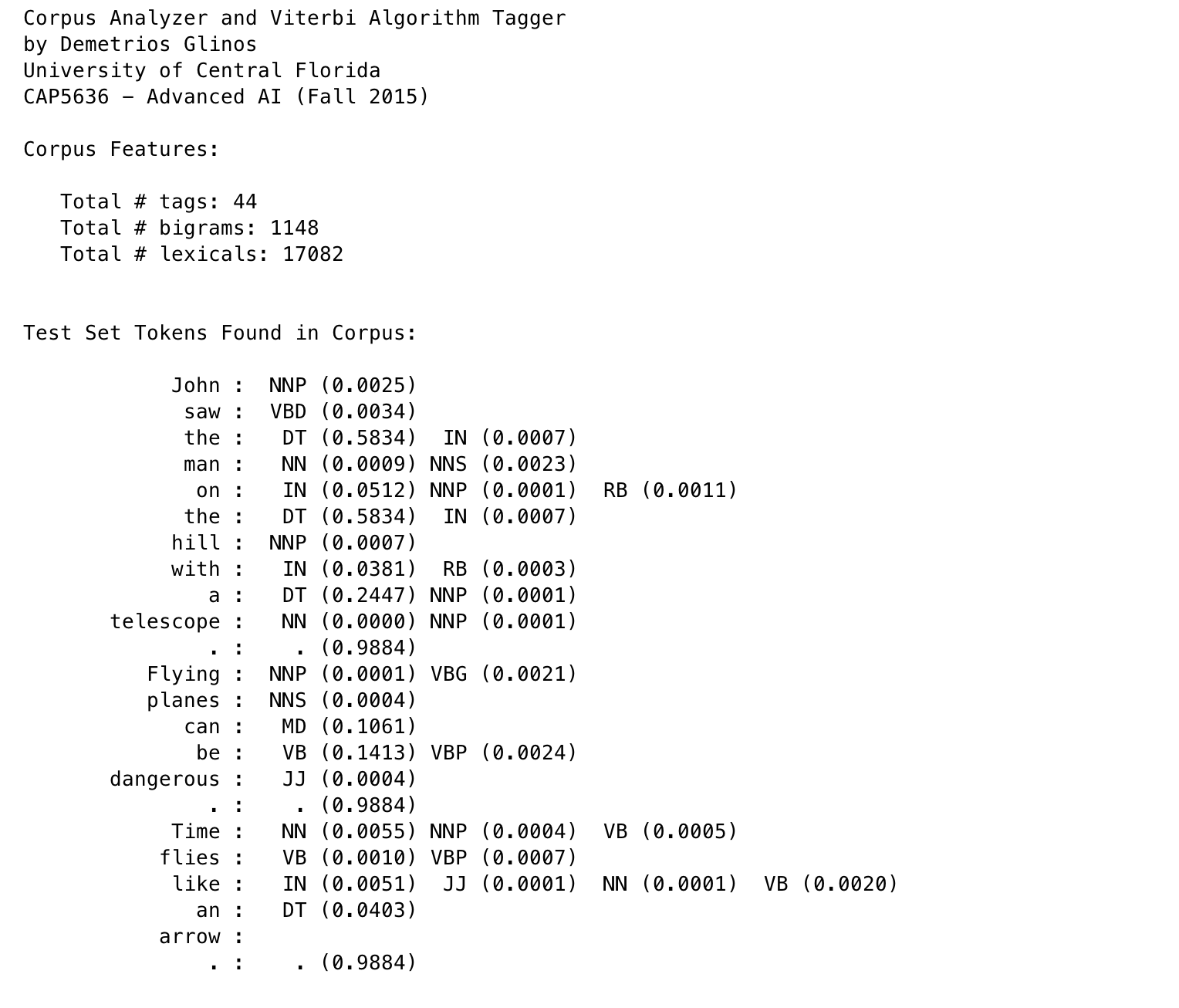
The algorithm begins with an empty set of prior tags, so the first bigrams used by the algorithm will contain no prior tag values. When the last input word is processed, the most likely tag sequence will be the path ending in the tag state with the greatest value for the last token.

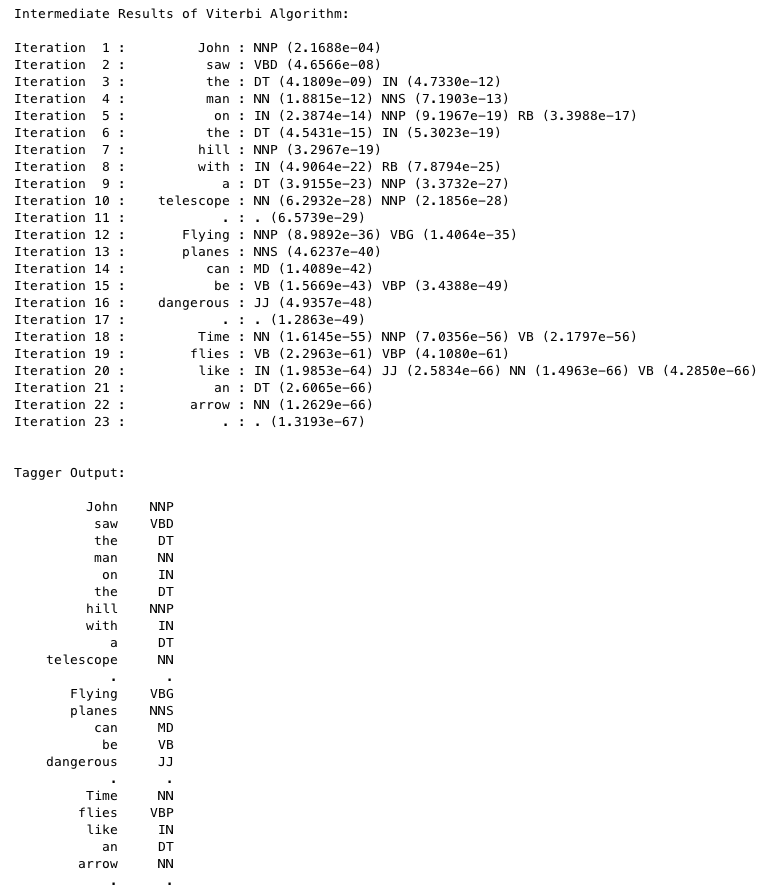
To report the actual path, the algorithm must keep track, for each set of tag states, of the prior tag states from whose transitions the maximal values were obtained. By so doing, the most likely tag sequence can be recovered by going backwards recursively from the maximal tag state for the final word, using the saved transitions.

**NOTE:**  For any word or token that was not encountered in the training corpus, the program must assume a tag of "NN" with a probability of 1.0.

Required Output

The program must produce output to System.out (the console) in the format shown below. The different sections of the output are discussed separately below.





**Header**

The must contain the title shown, the author's name, and identification of UCF and this course and semester

**Corpus Features**

The corpus features section must report the total number of unique tags observed in the training data. This section must also report on the number of bigrams that were extracted and the number of lexical generation probabilities that were computed.

**Test Set Tokens Found in Corpus**

This section shows the lexical generation probabilities that match the input tokens. For each token, this section reports the original token (before cleansing), and each part of speech tag with which it was observed in the training corpus, along with the probability for each such tag for the training corpus. If a word is not found in the corpus, the word should be listed.

**Intermediate Results of Viterbi Algorithm**

This section reports the maximal path probabilities for each tag state as each test input word is processed. Thus, iteration 1 will report the tag state values after the first test input word is processed, iteration 2 will do the same for the second word, and so on.

**Tagger Output**

This section reports the sequence of input tokens and their corresponding part-of-speech tags for the tag sequence that produces the maximal tag state value for the final test input token. Please note that the input tokens must be shown in their original form, that is, without cleansing and with all capitalization and/or suffixes as in the original input file.

**What to Submit**

You should submit a single Java source file (.java, not .class). This file should contain your Viterbi algorithm class and any supporting classes as well. Do not put support classes in separate files. Simply do not declare them to be public and just add them to the bottom of your Viterbi class file.

Your source file should include a header identifying UCF, this course and semester, and identifying the program author.

The program must be submitted on Webcourses. Email submissions will not be graded.