CS519 Natural language processing — Implementation Assignment 2 —
Out Jan 21st Due Feb 6th 11:59PM, 2015

General instruction.

1. You are encouraged to use Python or Java. But other languages are also accepted.

2. You are encouraged to form 2 person teams on the programming assignment. Each team will only need to submit one copy of the source code and report.

3. You are allowed to make use of existing packages for some functionalities including handling file IO, tokenlization and getting the raw counts of different tokens. Anything beyond this list will need advance approval from the instructor. Also you need to clearly reference your source.

4. Your source code and report will be submitted through the TEACH site

   `https://secure.engr.oregonstate.edu:8000/teach.php?type=want_auth`

   In your report, please clearly indicate all your team members.

5. Your code should be clearly documented. Undocumented code can lead to lower score. Your report has one section that clearly describes how to run your code. The instructor may request a demo session in which you must demonstrate how your program works and how the results were produced.

6. Grading will depend on your code as well as your report. In particular, **the clarity and quality of the report will be worth 20% of the grade**. So please write your report in clear and concise manner. Keep it organized, and clearly label your figures, legends, and tables if any.

POS Tagging with HMM

For this assignment, you will build a Hidden Markov Model and use it to tag words with their parts of speech. You will use supervised learning to estimate the parameters p(tag— previous tag) and p(word—tag) from a training set of already-tagged text. Some smoothing is necessary. You will then evaluate the learned model by finding the Viterbi tagging (i.e., best tag sequence) for some test data and measuring how many tags were correct.

For speed and simplicity, you will use relatively small datasets, and a bigram model instead of a trigram model. You will also ignore the spelling of words (useful for tagging unknown words). All these simplifications influence accuracy. So overall, your percentage of correct tags will be in the low 90's instead of the high 90's that are reported for state-of-the-art POS taggers.

1 Data

Two sets of data are provided (PA2_data.zip):

- The first one (*ic*, ice cream cone sequenceS with one character tags(C, H)) is an easy dataset for testing your algorithm/implementations and verifying its correctness.
- The second one (*en*, English word sequences with 1-character tags (see figure below)) is the one that you should report your results on.

C	Coordinating conjunction **or** Cardinal number
D	Determiner
E	Existential *there*
F	Foreign word
I	Preposition or subordinating conjunction
J	Adjective
L	List item marker (*a., b., c., . . .*) (rare)
M	Modal (*could, would, must, can, might . . .*)
N	Noun
P	Pronoun **or** Possessive ending (*'s*) **or** Predeterminer
R	Adverb **or** Particle
S	Symbol, mathematical (rare)
T	The word *to*
U	Interjection (rare)
V	Verb
W	*wh*-word (question word)
###	Boundary between sentences
,	Comma
.	Period
:	Colon, semicolon, or dash
-	Parenthesis
'	Quotation mark
$	Currency symbol

Figure 1: The tags in the en dataset. These come from longer, more specific tag names stripped down to their first letters.

```
1.  (* find best μ values from left to right by dynamic programming; they are initially 0 *)
2.  μ###(0) := 1
3.  for i := 1 to n                                              (* ranges over test data *)
4.      for t_i ∈ tag_dict(w_i)                                  (* a set of possible tags for w_i *)
5.          for t_{i-1} ∈ tag_dict(w_{i-1})
6.              p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)       (* arc probability *)
7.              μ := μ_{t_{i-1}}(i − 1) · p                      (* prob of best sequence that ends in t_{i-1}, t_i *)
8.              if μ > μ_{t_i}(i)                                (* but is it the best sequence (so far) that ends in t_i at time i? *)
9.                  μ_{t_i}(i) = μ                               (* if it's the best, remember it *)
10.                 backpointer_{t_i}(i) = t_{i-1}               (* and remember t_i's predecessor in that sequence *)
11. (* follow backpointers to find the best tag sequence that ends at the final state (### at time n) *)
12. t_n := ###
13. for i := n downto 1
14.     t_{i-1} :=backpointer_{t_i}(i)
```

Not all details are shown above. In particular, be sure to initialize variables in an appropriate way.

Figure 2: Sketch of the Viterbi tagging algorithm. $\mu_t(i)$ is the probability of the best path from the start state (### at time 0) to state t at time i.

Each dataset has two files:
train : tagged data for supervised training
test : tagged data for testing, your tagger should ignore the provided tags except when measuring the tagging accuracy

2 HMM Training and Tagging

The data is formatted as follows. Each line is one word/tag pair. Sentences are separated by token ### with tag ###.

Please use the train file to train a bigram HMM tagger (i.e. estimate the transition, emission and initial probabilities). For the purposes of this assignment, you do not need to smooth the emission or transition probabilities. For the *ic* data set, you do not need to worry about unseen words. But for the *en* dataset, you still need to use an UNK token to handle unseen words. In particular, turn any words that appear less than 5 times in the training data into UNK. Note that probablities that are small should be stored as log probabilities to avoid underflow.

You will then implement the Viterbi decoding algorithm for tagging the test data. Ignore the provided tags in the test file when tagging, and consider them only when evaluating. The following figure provides the pseudo-code for the Viterbi algorithm. Note that to speed things up, you are required to use a "tag dictionary" (tag_dict in line 4), otherwise your tagger will be much too slow. The tag dictionary for each word is a list of allowed tags for that word. Derive your tag dictionary from the training data. For a known word, allow only the tags that it appeared with in the training set. For an unknown word, allow all tags except ###.

Apply your Viterbi algorithm on the test set of both datasets and compare your output with the tags provided and report the accuracies. In particular, for the *ic* data set, you only need to report the overall accuracy. For the *en* dataset, report three separate accuracies: the overall accuracy, the accuracy on known words (all words that have appeared at least once in the training data), and accuracy on novel words (all words in the test data that have never appeared in the training data).

As a reference point, here are some baseline accuracies obtained or the *en* data set: Overall 92.48%, (known 95.99%, novel: 56.07%). This baseline result came from a naive tagger that just tags every known word with its most common part of speech from training data, ignoring context, and tags all novel words with N.

For each of the two datasets, you should also create a confusion matrix that indicates how often words that are supposed to have tag t_i was tagged with tag t_j . The entries in this confusion matrix can be raw frequencies. Please describe what kind of errors you are seeing for the *en* data set and comment on what modification can be made to improve the results.