CSCI 404 Assignment #5

Out: Feb 16, 2017

Due: Feb 25 midnight (11:59pm), online submission

Submission

1. All files should be submitted to Canvas. Only one of your team members needs to submit on behalf of the team. Indicate clearly in your submission the team members. You can submit multiple times, but please have the same team member resubmit all required files each time.
2. Implementation in Python.
3. Submit the assn5.zip file. For each question, submit your programs (if any) and a written report (report. txt, .doc, or .pdf) that presents the results you obtained, summarizes the methods you implemented if any, any additional resources you used, and discuss the results if necessary.
4. You must test that your programs on the CS lab machines with no problem before submitting them.
5. **Don't include the train/test data files in your submission!**

Part-of-Speech Tagging with a Hidden Markov Model (60 pts)

In this assignment, you will build a Hidden Markov Model and use it to tag words with their part of speech. This handout seems long because lots of specific directions are given so that you won't get stuck. Your program will be much shorter than this handout. Just read carefully.

The basic setting is a supervised learning. You need to estimate P(current tag | previous tag) and P(current word | current tag) from the training data set of pre-tagged text. Some smoothing is necessary to handle zero probability issue. You will then evaluate the learned model by finding the Viterbi tagging (i.e., the best tag sequence) for the test data and measuring how many tags were correct. Note that you'll use the Viterbi algorithm for testing but not for training.

For speed and simplicity, you will use relatively small datasets, a smaller tag set, and a bi-gram model instead of a trigram model. You will also ignore the spelling of words (useful for tagging unknown (or "unseen") words). All these simplifications hurt accuracy. So overall, your percentage of correct tags will be in the low 90's instead of the high 90's as reported on the slides.

Part 1: Data sets:

- ***ic***: Ice cream cone sequences with 1-character tags (C, H). Start with this easy dataset.
- ***en***: English words sequences with 1-character tags.

You only need to hand in results on the *en* dataset. The other is just for your convenience in testing your code.

The following figure shows the tags in the ***en*** data set. These are the simplifications of the Penn Treebank tag set. For example, all kinds of nouns (formerly NN, NNS, NNP, NNPS) are simply tagged as N here. Using only the first letters reduces the number of tags, speeding things up. (However, it results in a couple of unnatural categories, C and P.

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

Each dataset consists of

- **train**: tagged data for supervised training (en provides 4,000-100,000 words)
- **test**: tagged data for testing (25,000 words for en); your tagger should ignore the tags in this file except when measuring the accuracy of its tagging
- **raw**: untagged data (100,000 words for en). – see the file "one-count-smoothing.pdf" for the use of this file.

The file format is quite simple. Each line has a single word/tag pair separated by the / character. Punctuation marks count as words. The special word **###** is used for sentence boundaries, and is always tagged with **###**.

Suggested notation for discussion and for your program:

- Whichever string is being discussed (whether it is from train or test) consists of n+1 words, w_0, w_1, .., w_n
- The corresponding tags are t_0, t_1, …, t_n. We have w_i/t_i = ###/### for i = 0, and for i = n, and probably also for some other values of i.
- "tt" to name tag-to-tag *transition* probabilities, as in $p_{tt}(t_i \mid t_{i-1})$.
- "tw" to name tag-to-word *emission* probabilities, as in $p_{tw}(w_i \mid t_i)$.

Part 2: Viterbi decoder on *ic* data:

A Viterbi tagger find the single best path through an HMM – the single most likely weather sequence given the ice cream, or the single most likely part-of-speech tag sequence.

Write a bigram Viterbi tagger that can be run as follows:

```
vtag ictrain ictest
```

You may want to review the slides on Hidden Markov Model tagging, and any other reference that you find useful.

For now, you should use naïve unsmoothed estimates (i.e., maximum-likelihood estimates).

Your program must print two lines summarizing its performance on the test data, in the following format (ignore the particular numbers in this example for now):

```
Tagging accuracy (Viterbi decoding): 92.48% (known: 95.99% novel: 56.07%)
```

You are also free to print out whatever other information is useful to you, including the tags your program picks, it's accuracy it goes along, various counts/probabilities, etc. A common trick, to yourself something to stare at, is to print a period (or some special mark) to standard error every n words (n = 1000).

In the required output illustrated above, each accuracy number considers some subset of the test tokens and asks what percentage of them received the correct tag:

- The overall accuracy (e.g., 92.48%) considers all word tokens, other than the sentence boundary markers ### (No one in NLP tries to take credit for tagging ### correctly with ###!).
- The known-word accuracy (e.g., 95.99%) considers only tokens of words (other than ###) that also appeared in **train**.
- The novel-word accuracy (e.g., 56.07%) considers only tokens of words that did NOT appear in train. (These are very hard to tag, since context is the only clue to correct tag. But they constitute about 9% of all tokens in **entest**, so it is important to tag them as accurately as possible.)

Some suggestions that will make your life easier:

- Make sure you really understand the algorithm before you start coding! Review the reading and the pseudo-code on slides, and work out the example on paper if that helps.
- Your program is suggested to go through the following steps:
    - Read the **train** data and store the counts in global tables. (Your function for computing probabilities on demand, such as p_{tw},, should access these tables. When working with en dataset, you will modify those functions to do smoothing.)
    - Read the **test** data into memory.
    - Follow the Viterbi algorithm pseudo-code to find the most likely tag sequence for the given observation sequence.
    - Compute and print the accuracy of the tagging. (You can compute the accuracy at the same time as you extract the tag sequence while following backpointers.)
- If you have enough memory, you can treat the **train** file as one long string that happens to contain **###** words. Similarly for the **test** file.

- Figure 2 refers to a "tag dictionary" that stores all possible tags for each word. As long as you only use the **ic** dataset, the tag dictionary is so simple that you can specify it directly in the code: tag_dict(###) = {###}, and tag_dict(w) = {C, H} for any other word w. When working with **en** data set, you'll generalize this to derive the tag dictionary from training data.

```
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. In other words, it maximizes $p(t_1, w_1, t_2, w_2, \ldots t_i, w_i \mid t_0, w_0)$ over all possible choices of $t_1, \ldots t_i$ such that $t_i = t$.

- Before you start coding, make a list of the data structures you will need to maintain.
- Probabilities that might be small (transition probabilities α and emission probabilities β) should be stored in memory as log-probabilities. Doing this is actually crucial to prevent underflow. The ice cream data set may get away without using logs given it's corpus was only 33 "words". It turns out for most purposes you only care about the relative Viterbi (μ) values at each time step. So to avoid overflow, you can rescale them by an arbitrary constant at every time step, or eveyr several time steps when they get too small.
- You can use either natural logarithms or $\log_2$. Natural log is slightly faster though. Anyway, you are not required to report any log-probabilities for this work.

Check your work as follows. Run

```
vtag ictrain ictest
```

It should yield a tagging accuracy of 87.77% or 90.91% (depends on your choice when there is a tie), with no novel words.

You don't have to hand anything in for this dataset.

Part 3: Viterbi decoder on en data: improvement by using a tag dictionary and add-1 smoothing

Now you will improve your tagger so that you can run it on real data

```
vtag entrain entest
```

This means using a proper tag dictionary (for speed) and a smoothed probabilities (for accuracy). On the ic dataset, you were able to get away without smoothing because you didn't have sparse data. You had actually observed all possible "words" and "tags" in ictrain.

Your tagger should beat the following "baseline" result:

```
Tagging accuracy (Viterbi decoding): 92.48% (known: 95.99% novel: 56.07%)
```

The baseline result came from a naïve unigram tagger (which just tagged every known word with its most common part of speech from training data, ignoring context, and tagged all novel words with N). This baseline tagger does pretty well because most words are easy to tag. To justify using a bigram tagger, you must show it can do better.

You are required to use a "tag dictionary" – otherwise your tagger will be too much slow. Each word has a list of allowed tags, and you should consider only those tags. That is, don't consider tag sequences that are incompatible with the dictionary, even if they have positive smoothed probability. See the pseudocode in Figure 2.

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 ###. (*Hint*: During training, before you add an observed tag t to tag_dict(w) (and before incrementing c(t,w)), check whether c(t,w) > 0 already. This lets you avoid adding duplicates.)

How about smoothing? You do need some kind of smoothing, since you won't to find any tagging of the **entest** data without using some novel transitions and emissions.

To get the program working on this dataset, use some very simple form of smoothing for now. Fox example, add-one smoothing without backoff (on both p_{tw} and p_{tt}). Please don't smooth $p_{tw}(w_i = \#\#\# \mid t_i = \#\#\#)$. This probability should always be 1, even without any data. However, you'll find that this smoothing method gives lower accuracy than the baseline tagger!

```
Tagging accuracy (Viterbi decoding): 91.84% (known: 96.60% novel: 42.55%)
```

Note:

- If you want to obtain the sample results provided, you should ignore the training file's very first OR very last ###/### when accumulate counts during training to ensure the smoothed (or unsmoothed) probabilities sum to sum to 1.
- In effect, we are treating all novel words as if they had been replaced in the input by a single special word OOV. That way we can pretend that the vocabulary is limited to exactly V types, one of which is the unobserved OOV. Your definition of V should new include all types that were observed in **train U raw (Union of these two sets)**. This vocabulary set should be used for all your computations.

Part 4: Viterbi decoder on en data: improvement on smoothing

For full credit, now improve the smoothing. Use the backoff smoothing method that is described in one-count-smoothing.pdf, a relative simple method called "one-count smoothing". This should improve your performance considerably.

How much did your tagger improve on the accuracy of the baseline tagger? Answer in your report. Include in the report screenshots of your execution, and output of your program.