CS 10: Problem Solving via Object Oriented Programming
Spring 2022
PS-5

The goal of this problem set is to build a personal digital assistant named "Sudi". Okay, we'll stop short of that, but we will tackle part of the speech understanding problem that a digital assistant such as Sudi would have to handle.

Background

A part of speech (POS) tagger labels each word in a sentence with its part of speech (noun, verb, etc.). For example:

```
The Fulton County Grand Jury said Friday an investigation of Atlanta's
recent primary election produced `` no evidence '' that any
irregularities took place .

=>
DET NP N ADJ N VD N DET N P NP
ADJ N N VD `` DET N '' CNJ DET
N VD N .

or
The/DET Fulton/NP County/N Grand/ADJ Jury/N said/VD Friday/N an/DET investigation/N of/P Atlanta's/NP
recent/ADJ primary/N election/N produced/VD ``/`` no/DET evidence/N ''/'' that/CNJ any/DET
irregularities/N took/VD place/N ./.
```

So we see that "The" is a "DET" (determiner), "Fulton" an "NP" (proper noun), "County" a "N" (noun), etc. The punctuation is its own tag.

The tags that we'll use for this problem set are taken from the book Natural Language Processing with Python:

Tag	Meaning	Examples
`ADJ`	adjective	new, good, high, special, big, local
`ADV`	adverb	really, already, still, early, now
`CNJ`	conjunction	and, or, but, if, while, although
`DET`	determiner	the, a, some, most, every, no
`EX`	existential	there, there's
`FW`	foreign word	dolce, ersatz, esprit, quo, maitre
`MOD`	modal verb	will, can, would, may, must, should
`N`	noun	year, home, costs, time, education
`NP`	proper noun	Alison, Africa, April, Washington
`NUM`	number	twenty-four, fourth, 1991, 14:24
`PRO`	pronoun	he, their, her, its, my, I, us
`P`	preposition	on, of, at, with, by, into, under
`TO`	the word to	to
`UH`	interjection	ah, bang, ha, whee, hmpf, oops
`V`	verb	is, has, get, do, make, see, run
`VD`	past tense	said, took, told, made, asked
`VG`	present participle	making, going, playing, working
`VN`	past participle	given, taken, begun, sung
`WH`	wh determiner	who, which, when, what, where, how

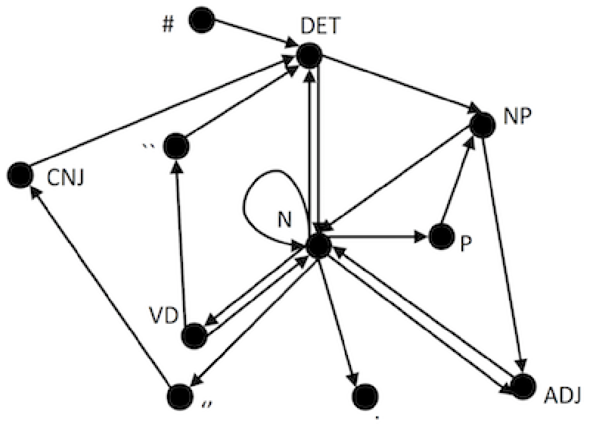
A single word can have different tags in different contexts; e.g., "present" could be a verb ("I'd like to present you with this opportunity to write some great code"); an adjective ("it'll be easier if you're present at section meeting"); or a noun ("no time like the present to get started on it"). A good tagger makes use of the context to disambiguate the possibilities and determine the correct label. The tagging is in turn a core part of having a "Sudi" understand what a sentence means and how to respond.

We'll be taking the statistical approach to tagging, which means we need data. Fortunately the problem has been studied a great deal, and there exist good datasets. One prominent one is the Brown corpus (extra credit: outdo it with a Dartmouth corpus), covering a range of works from the news, fictional stories, science writing, etc. It was annotated with a more complex set of tags, which have subsequently been distilled down to the ones above.

The goal of POS tagging is to take a sequence of words and produce the corresponding sequence of tags.

POS tagging via HMM

We will use a hidden Markov model (HMM) approach, applying the principles covered in class. Recall that in an HMM, the states are the things we don't see (hidden) and are trying to infer, and the observations are what we do see. So the **observations are words** in a sentence and the **states are tags** because the text we'll observe is not annoted with its part of speech tag (that it our program's job). We proceed through a model by moving from state to state, producing one observation per state. In this "bigram" model, each tag depends on the previous tag. Then each word depends on the tag. (For extra credit, you can go to a "trigram" model, where each tag depends on the previous two tags.) Let "#" be the tag "before" the start of the sentence. Then a model encapsulating just our example sentence is:



The sentence follows the path: # (start) — DET ("the") — NP ("Fulton") — N ("County") — ADJ ("Grand") — N ("jury") — VD ("said") — N ("Friday") — ... — N ("place") — . (".").

An HMM is defined by its states (here parts of speech tags), transitions (here tag to tag, with weights), and observations (here tag to word, with weights). Let's consider training the pictured model from just that one sentence (i.e., knowing the corresponding words and tags, or the path). VD was used three times, tagging "said", "took", and "produced" one time each (and thus with probability 1/3). Two of its transitions were to N (once corresponding to "said Friday" and once to "took place") and one was a transition to `` (for "produced ``"). Thus the transition probabilities from VD would be 2/3 to N and 1/3 to ``. To reemphasize: **probabilities are computed for the tags (i.e., the states)**: for each tag, the frequencies of the transitions out are divided by the total number of transitions out, and the frequencies of the words it tags are divided by the total number of words it tags. Some probabilities / log-based edge weights you can compute from this training data by counting:

tag	# trans	transitions	# obs	observations
#	1	DET = log(1/1) = 0	n/a	n/a

DET	4	N = log(3/4) = -0.12
		NP = log(1/4) = -0.60
			4	the = log(1/4) = -0.60
				an = log(1/4) = -0.60
				no = log(1/4) = -0.60
				any = log(1/4) = -0.60

N	9	ADJ = log(1/9) = -0.95
		VD = log(3/9) = -0.48
		DET = log(1/9) = -0.95
		...
			9	investigation = log(1/9) = -0.95
				evidence = log(1/9) = -0.95
				...

VD	3	N = log(2/3) = -0.18 ``= log(1/3) = -0.48	3	produced = log(1/3) = -0.48 ...

NP	2	N = log(1/2) = -0.30
		ADJ = log(1/2) = -0.30
			2	Fulton = log(1/2) = 0.30
				Atlanta's = log(1/2) = -0.30

Now, suppose we had learned the model and wanted to tag a new sentence: "The investigation produced evidence" (that's about the most sensible sentence I could come up with using that vocabulary, though of course the sentence need not be sensible — mad libs anyone?). Note that we're not going to force a "stop" state (ending with a period, question mark, or exclamation point) since the Brown corpus includes headlines that break that rule. The Viterbi algorithm starts at the # (start) state, with a score of 0, before any observation. Then to handle observation *i*, it propagates from each reached state at observation *i*-1, following each transition. The score for the next state through observation *i* is the sum of the score at the current state through *i*-1 plus the transition score from current to next plus the score of observation *i* in next. So:

#	observation	next state	curr state	scores: curr + transit + obs	next score
start	n/a	#	0	n/a	0

0	The	DET	#	0 + 0 + -0.60	-0.60
1	investigation	N	DET	-0.60 + -0.12 + -0.95	-1.67
		NP	DET	-0.60 + -0.60 + U	-1.20 + U
2	produced	VD	N	-1.67 + -0.48 + -0.48	-2.63
		DET	N	-1.68 + -0.95 + U	-2.63 + U
		ADJ	N	-1.68 + -0.95 + U	-2.63 + U
			NP	(-1.20 + U) + -0.30 + U	~~-1.50 + 2U~~
		N	NP	(-1.20 + U) + -0.30 + U	-1.50 + 2U
		...			
3	evidence	N	VD	-2.63 + -0.18 + -0.95	-3.76
			DET	(-2.63 + U) + -0.12 + U	~~-2.75 + 2U~~
			...		
		``	VD	-2.63 + -0.48 + U	-3.11 + U
		NP	DET	(-2.63 + U) + -0.60 + U	-3.23 + 2U
		...			

When we're in NP, we've never actually seen the word "investigation", so what do we use as the observation probability? log(0)=-infinity, which would make it impossible, but maybe we don't want to completely rule out something that we've never seen. So let's just give it a low log probability, call it *U* for "unseen". It should be a negative number, perhaps worse than the observed ones but not totally out of the realm.

Notice there are some cases where we arrive at a particular next state from different previous states. E.g., we could be in N for "evidence" with the previous word tagged as wither VD or DET. In that case, we give the tag the best score over the possibilities. I showed some of the possibilities in these cases, labeling the best as "winner" and the others as "losers". In practice, the code will just keep track of the winning score and previous state.

After working through the sentence, we look at the possible states for the last observation, to see where we could end up with the best possible score. Here it's N, at (assuming U is sufficiently bad). So we know the tag for "evidence" is N. We then backtrace to where this state came from: VD. So now we know the tag for "produced" is VD. Backtracing again goes to N, so "investigation" is N. Then back to DET, so "The" is DET. Then #, so we're done.

Testing

To assess how good a model is, we can compute how many tags it gets right and how many it gets wrong on some test sentences. (Even tougher: how many sentences does it get entirely right vs. at least one tag wrong.) It wouldn't be fair to test it on the sentences that we used to train it (though if it did poorly on those, we'd be worried).

Provided in texts.zip are sets of files, one pair with the sentences and a corresponding one with the tags to be used for training, and another pair with the sentences and tags for testing. Each line is a single sentence (or a headline), cleanly separated by whitespace into words/tokens, with punctuation also thus separated out. The example at the beginning ("The Fulton County Grand Jury...") shows the first line of the "brown-train-sentences.txt" and "brown-train-tags.txt" file (they are supposed to be on a single line each, but were split for clarity).

So use the train sentences and train tags files to generate the HMM. Then apply the HMM to each line in the test sentences file, and compare the results to the corresponding test tags line. Count the numbers of correct vs. incorrect tags.

Implementation Notes

Training

- I preserved capitalization in the examples, but lowercasing all the words makes sense.
- While we think of the model as a graph, you need not use the Graph class, and you might in fact find it easier just to keep your own Maps of transition and observation probabilities as we did with finite automata. Think first about what the mapping structure should be (from what type to what type). Recall that you can nest maps (the value for one key is itself a map). The finite automata code might be inspiring, but remember the differences (e.g., HMM observations are on states rather than edges; everything has a log-probability associated with it).
- Make a pass through the training data just to count the number of times you see each transition and observation. Then go over all the states, normalizing each state's counts to probabilities (divide by the total *for the state*). Remember to convert to log probabilities so you can sum the scores. (FWIW, the sample solution uses natural log, not log10.)

Viterbi tagging

- While the table shows all the scores, we really need only to keep the current ones and the next ones, as shown in the pseudocode in the lecture notes. So that might simplify the representation you use.
- The backtrace, on the other hand, needs to go all the way back: for observation *i*, for each state, what was the previous state at observation *i*-1 that produced the best score upon transition and observation. Note that if we index the observations by numbers, as suggested here, the representation is essentially a list of maps.
- After handling the special case of the start state, start for real with observation 0 and work forward. Either consider all possible states and look back at where they could have come from, or consider all states from which to come and look forward to where they could go. In either case, be sure to find the max score (and keep track of which state gave it).
- Use a constant variable for the observation of an unobserved word, and play with its value.
- The backtrace starts from the state with the best score for the last observation and works back to the start state.

Exercises

For this assignment, you may work either alone, or with one other partner. Same discussion as always about partners.

1. Write a method to perform Viterbi decoding to find the best sequence of tags for a line (sequence of words).
2. Test the method on simple hard-coded graphs and input strings (e.g., from programming drill, along with others you make up). In your report, discuss the tests and how they convinced you of your code's correctness.
3. Write a method to train a model (observation and transition probabilities) on corresponding lines (sentence and tags) from a pair of training files.
4. Write a console-based test method to give the tags from an input line.
5. Write a file-based test method to evaluate the performance on a pair of test files (corresponding lines with sentences and tags).
6. Train and test with the two example cases provided in texts.zip: "simple", just some made up sentences and tags, and "brown", the full corpus of sentences and tags. The sample solution got 32 tags right and 5 wrong for simple, and 35109 right vs. 1285 wrong for brown, with an unseen-word penalty of -100. (Note: these are using natural log; if you use log10 there might be some differences.) In a short report, provide some example new sentences that are tagged as expected and some that aren't, discussing why. Also discuss your overall testing performance, and how it depends on the unseen-word penalty (and any other parameters you use).

Some ideas for extra credit.

- Move from bigrams to trigrams (i.e., a tag depends on the previous two tags). Compare performance.
- Be more careful with things that haven't been observed; this is even more necessary with trigrams. One approach is "interpolation": take a weighted sum of the unigram, bigram, and trigram probabilities, so that the less-informative but more-common lower-order terms can still contribute.
- Perform cross-validation. Randomly split the data into different partitions (say 5 different groups, or "folds"). Set aside part 0, train on parts 1-4, and then test on part 0. Then set aside part 1, train on parts 0 & 2-4, and test on 1. Etc. So each part is used as a test set, with the other parts used in training, to construct the model. Note that the Brown corpus is organized by genre (news, fiction, etc.), so deal out the rows among the partitions instead of assigning the first n/5 to one group, the second n/5 to the next, etc.
- Use your model generatively. This can be done randomly (essentially following a path through the HMM, making choices according to the probabilities), or predictively (identify what the best next word would be to continue a sentence).
- Use a different corpus; e.g., from phonemes to morphemes, or from one language to another.

Submission Instructions

Submit all your code and your short report (testing on hard-coded graphs, training/testing on provided datasets).

Acknowledgement

Thanks to Sravana Reddy for consulting in the development of the problem set, providing helpful guidance and discussion as well as the processed dataset.

Grading rubric

Total of 100 points

Correctness (70 points)

5	Loading data, splitting into lower-case words
15	Training: counting transitions and observations
5	Training: log probabilities

15 Viterbi: computing scores and maintaining back pointers
5 Viterbi: start; best end
10 Viterbi: backtrace
7 Console-driven testing
8 File-driven testing

Structure (10 points)

4 Good decomposition into objects and methods
3 Proper use of instance and local variables
3 Proper use of parameters

Style (10 points)

3 Comments for classes and methods
4 Good names for methods, variables, parameters
3 Layout (blank lines, indentation, no line wraps, etc.)

Testing (10 points)

5 Example sentences and discussion
5 Testing performance and discussion