601.465/665 — Natural Language Processing
Assignment 5: Tagging with a Hidden Markov Model

Prof. Kevin Duh and Jason Eisner — Fall 2019
Due date: Friday 15 November, 11:00

Last compiled: 2019-11-06 at 02:20:16 UTC

In this assignment, you will build a Hidden Markov Model and use it to tag words with their parts of speech.

Collaboration: ***You may work in pairs on this assignment,*** as it is programming-intensive and requires some real problem-solving. That is, if you choose, you may collaborate with one partner from the class, handing in a single homework with both your names on it. However:

1. You should do all the work *together*, for example by pair programming. Don't divide it up into "my part" and "your part."
2. Your `README` file should **describe at the top what each of you contributed**, so that we know you shared the work fairly.
3. Your partner **must not be the same partner** as you had for HW4. Make new friends! :-)

In any case, observe academic integrity and never claim any work by third parties as your own.

Reading: First read the handout attached to the end of this assignment!

---

1. In the first part of the assignment, you will do **supervised** learning, estimating the parameters $p(\text{tag} \mid \text{previous tag})$ and $p(\text{word} \mid \text{tag})$ 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.

   Then in the second part of the assignment, you will improve your supervised parameters by reestimating them on additional "raw" (untagged) data, using the Expectation-Maximization (EM) algorithm. This yields a **semi-supervised model**, which you will again evaluate by finding the Viterbi tagging on the test data. Note that you'll use the Viterbi approximation for testing but *not* for training—you'll do real EM training using the full **forward-backward algorithm**.

   This kind of procedure is common. Will it work in this case? 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).[1] The forward-backward algorithm is given in reading section B, which is based on the ice-cream spreadsheet from Jason Eisner, available here.

2. Write a bigram Viterbi tagger that can be run as follows on the ice cream data (reading section E):

   ```
   python3 vtag.py ictrain ictest
   ```

[1] 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. But another factor helps your accuracy measurement: you will also use a smaller-than-usual set of tags. The motivation is speed, but it has the side effect that your tagger won't have to make fine distinctions.

For now, you should use naive unsmoothed estimates (i.e., maximum-likelihood estimates). The Viterbi algorithm was given in reading section C and some implementation hints were given in reading section G.

Your program must summarize the model's performance on the **test** set, in the following format. These performance metrics were defined in reading section F.1.

```
Tagging accuracy (Viterbi decoding): 87.88%   (known: 87.88%  novel: 0.00%)
```

3. Now, you will improve your tagger so that you can run it on real data (reading section E):

```
python3 vtag.py entrain entest
```

This means using a proper tag dictionary (for speed) and smoothed probabilities (for accuracy).[2] Ideally, your tagger should beat the following "baseline" result:

```
Model perplexity per tagged test word: 2802.383
Tagging accuracy (Viterbi decoding): 92.12%   (known: 95.60%  novel: 56.07%)
```

Now that you are using probabilities that are smoothed away from 0, you will have finite perplexity. So make your program also print the perplexity, in the format shown above (see reading section F.2). This measures how surprised the model would be by the test observations—both words and tags—before you try to reconstruct the tags from the words.

The baseline result shown above came from a naive *unigram* Viterbi tagger where the *bigram* transition probabilities $p_{\text{tt}}(t_i \mid t_{i-1})$ are replaced by *unigram* transition probabilities $p_{\text{t}}(t_i)$.[3]

The baseline tagger is really fast because it is a degenerate case—it throws away the tag-to-tag dependencies. If you think about what it does, it just tags every known word separately with its most probable part of speech from training data.[4] Smoothing these simple probabilities will break ties in favor of tags that were more probable in training data; in particular, every novel word will be tagged with the most common part of speech overall, namely `N`. Since context is not considered, no dynamic programming is needed. It can just look up each word token's most probable part of speech in a hash table—with overall runtime $O(n)$. This baseline tagger also is pretty accurate (92.12%) because most words are easy to tag. To justify the added complexity of a bigram tagger, you must show it can do *better*![5]

Your implementation is required to use a "tag dictionary" as shown in Figure 3 of the reading—otherwise your tagger will be much too 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.

[2] 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**.

[3] The bigram case is the plain-vanilla definition of an HMM. It is sometimes called a 1st-order HMM, meaning that each tag depends on 1 previous tag—just enough to make things interesting. A fancier trigram version using $p_{\text{ttt}}(t_i \mid t_{i-2}, t_{i-1})$ would be called a 2nd-order HMM. So by analogy, the unigram baseline can be called a 0th-order HMM.

[4] That is, it chooses t_i to maximize $p(t_i \mid w_i)$. Why? Well, modifying the HMM algorithm to use unigram probabilities says that t_{i-1} and t_{i+1} have no effect on t_i. so (as you can check for yourself) the best path is just maximizing $p(t_i) \cdot p(w_i \mid t_i)$ at each position i separately. But that is simply the Bayes' Theorem method for maximizing $p(t_i \mid w_i)$.

[5] Comparison to a previous baseline is generally required when reporting NLP results.

Derive your tag dictionary from the training data. For a word that appears in **train**, allow only the tags that it appeared with in **train**. (*Remark:* This implies that the word ### will only allow the tag ###.) If a word doesn't appear in **train**, it won't be in the tag dictionary; in this case, you should allow all tags except ### (see reading section E). (*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.)

The tag dictionary is only a form of pruning during the dynamic programming. The tags that are *not* considered will still have positive smoothed probability. Thus, even if the pruned dynamic programming algorithm doesn't manage to consider the true ("gold") tag sequence during its search, the true sequence will still have positive smoothed probability, which you should evaluate for the perplexity number. How is that possible? Remember that in general, the true path may not match the Viterbi path (because the tagger may be wrong). If you're pruning, the true path may not even be in the trellis at all! Yet you can compute its $n + 1$ arc weights anyway (by calling the same methods you called to compute the arc weights in the trellis).

Smoothing is necessary not only to compute the perplexity of the model, but also to do bigram tagging at all. You won't be able to find *any* tagging of the `entest` data without using some novel transitions, so you need to smooth so they have positive probability.

To get the program working on this dataset, use some very simple form of smoothing for now. For example, add-λ smoothing without backoff (on both p_{tw} and p_{tt}). However, at least with $\lambda = 1$, you'll find that this smoothing method gives lower accuracy than the baseline tagger!

```
Model perplexity per tagged test word: 1690.606
Tagging accuracy (Viterbi decoding): 91.84%   (known: 96.60%  novel: 42.55%)
```

Certainly the above result is much better than baseline on perplexity, and it is also a little more accurate on *known* words. However, the baseline is much more accurate on *novel* words, merely by the simple heuristic of assuming that they are nouns. Your tagger doesn't have enough training data to *figure out* that unknown words are most likely to be nouns (in any context), *because it doesn't back off from contexts.*

4. Extend your `vtag.py` so that it tries **posterior decoding** (reading section D) once it's done with Viterbi decoding. At the end of `vtag.py`, you should run forward-backward over the *test* word sequence: see pseudocode in Figure 2. The decoder will have to use the forward-backward results to find the tag at each position that is most likely *a posteriori* (hint: insert some code at line 13).

   If you *turn off smoothing* (just set $\lambda = 0$) and run `python3 vtag.py ictrain ictest`, you can check the unigram and bigram posterior probabilities (lines 13 and 17 of Figure 2) against the ice cream spreadsheet. They are shown directly on the spreadsheet on the two graphs for iteration 0.

   With *no smoothing*, the output of `python3 vtag.py ictrain ictest` should now look like this:

```
Model perplexity per tagged test word: 3.833
Tagging accuracy (Viterbi decoding): ...%  (known: ...%  novel: 0.00%)
Tagging accuracy (posterior decoding): 87.88%   (known: 87.88%  novel: 0.00%)
```

   This corresponds to looking at the reconstructed weather graph and picking tag `H` or `C` for each day, according to whether $p(\texttt{H}) > 0.5$.

Posterior decoding tries to maximize tagging accuracy (the number of tags you get right), rather than the probability of getting the whole sequence right. On the other hand, it may be a bit slower. How does it do on `python3 vtag.py entrain entest`?

Finally, your program should create a file called `test-output` in the current working directory, which contains the tagging of the test data produced by posterior decoding. The file format should be a tagged sequence in the same format as `entrain` and `entest`. You can compare `test-output` to `entest` at this output to see where your tagger is making mistakes, and the autograder will score this output.

Tuning We have not provided separate dev data: so if you want to tune any hyperparameters such as smoothing parameters, you will have to split the training data into train and dev portions for yourself (e.g., using k-fold cross-validation). Of course, don't use the test data to train your system.

Your tagger might still fall short of the state-of-the-art 97%, even though the reduced tagset in Figure 4 of the reading ought to make the problem easier. Why? Because you only have 100,000 words of training data.

How much did your tagger improve on the accuracy and perplexity of the baseline tagger (see page 2)? Answer in your `README` with your observations and results, including the required output from running `python3 vtag.py entrain entest`. Submit the source code for this version of `vtag.py`. Remember, for full credit, your tagger should include a tag dictionary, and both Viterbi and posterior decoders.

In this assignment, the leaderboard will show the various performance numbers that your code prints out. The autograder will probably run your code on the same `entest` that you have in your possession. This is really "dev-test" data because it is being used to help develop everyone's systems.

For grading, however, we will evaluate your code on "final-test" data that you have never seen (as in HW3). The autograder will run your code to both train and test your model. It will compare the `test-output` generated by your posterior decoder to the gold-standard tags on the final-test data.

5. Now let's try the EM algorithm. Copy `vtag.py` to a new program, `vtag_em.py`, and modify it to reestimate the HMM parameters on **raw** (untagged) data. You should be able to run it as

```
python3 vtag_em.py entrain25k entest enraw
```

Here `entrain25k` is a shorter version of `entrain`. In other words, let's suppose that you don't have much supervised data, so your tagger does badly and you need to use the unsupervised data in `enraw` to improve it.

Your EM program will alternately tag the **test** data (using your Viterbi decoder) and modify the training counts. So you will be able to see how successive steps of EM help or hurt the performance on **test** data.

Again, you'll use the forward-backward algorithm, but it should now be run on the **raw** data. The forward-backward algorithm was given in reading section B and some implementation hints for EM were given in reading section H.

The program should run 10 iterations of EM. Its output format should be as shown in Figure 1. Note that `vtag_em.py`'s output distinguishes three kinds of accuracy rather than two, and includes the perplexity per *untagged* **raw** word as well as the perplexity per *tagged* **test** word.

After printing that output, `vtag_em.py` should again write a `test-output` file for the autograder. Just as in question 4, `test-output` should give the result of posterior decoding on the **test** data, but now using the model that resulted from 10 iterations of EM training.

[read **train**]
[read **test**]
[read **raw**]
[decode the **test** data]
`Model perplexity per tagged test word: ...`
`Tagging accuracy (Viterbi decoding): ...% (known: ...% seen: ...% novel: ...%)`
[compute new counts via forward-backward algorithm on **raw**]
`Iteration 0: Model perplexity per untagged raw word: ...`
[switch to using the new counts]
[re-decode the **test** data]
`Model perplexity per tagged test word: ...`
`Tagging accuracy (Viterbi decoding): ...% (known: ...% seen: ...% novel: ...%)`
[compute new counts via forward-backward algorithm on **raw**]
`Iteration 1: Model perplexity per untagged raw word: ...`
[switch to using the new counts]
[re-decode the **test** data]
`Model perplexity per tagged test word: ...`
`Tagging accuracy (Viterbi decoding): ...% (known: ...% seen: ...% novel: ...%)`
[compute new counts via forward-backward algorithm on **raw**]
`Iteration 2: Model perplexity per untagged raw word: ...`
[switch to using the new counts]
[re-decode the **test** data]
`Model perplexity per tagged test word: ...`
`Tagging accuracy (Viterbi decoding): ...% (known: ...% seen: ...% novel: ...%)`
[compute new counts via forward-backward algorithm on **raw**]
`Iteration 3: Model perplexity per untagged raw word: ...`
[switch to using the new counts]

Figure 1: Output format for `vtag_em.py`. Your program should include the lines shown in `this font`. The material in [brackets] is not part of the output; it indicates what your program would be doing at each stage.

☜3 ☜4 Submit the source code for `vtag_em.py`. In `README`, include the output from running `python3 vtag_em.py entrain25k entest enraw` (or at least the required lines from that output). Your `README` should also answer the following questions:

(a) Why does Figure 2 initialize $\alpha_{\#\#\#}(0)$ and $\beta_{\#\#\#}(n)$ to 1?

(b) Why is the perplexity per tagged test word so much higher than the perplexity per untagged raw word? Which perplexity do you think is more important and why?

(c) V counts the word types from **train** and **raw** (plus 1 for OOV). Why not from **test** as well?

(d) Did the iterations of EM help or hurt overall tagging accuracy? How about tagging accuracy on known, seen, and novel words (respectively)?

(e) Explain in a few clear sentences why you think the EM reestimation procedure helped where it did. How did it get additional value out of the `enraw` file?

(f) Suggest at least two reasons to explain why EM didn't always help.

(g) What is the maximum amount of ice cream you have ever eaten in one day? Why? Did you get sick?

Merialdo (1994) found that although the EM algorithm improves likelihood at every iteration, the tags start looking less like parts of speech after the first few iterations, so the tagging accuracy will get worse even though the perplexity improves. His paper has been cited 700 times, often by people who are attempting to build better unsupervised learners!

601.465/665 — Natural Language Processing
Reading for Assignment 5: Tagging with a Hidden Markov Model

Profs. Kevin Duh and Jason Eisner — Fall 2019

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies assignment 5, which refers to it.

A Notation

In this reading handout, I'll use the following notation. But this is 2019. Please use complete words, not these shorthands, to name variables in your code.

- The input string consists of n words, $w_1, \ldots w_n$.
- The corresponding tags are $t_1, \ldots t_n$.
- We sometimes write a tagged word as w_i/t_i, for example in the data files.
- I'll use "tt" to name tag-to-tag *transition* probabilities, as in $p_{\text{tt}}(t_i \mid t_{i-1})$.
- I'll use "tw" to name tag-to-word *emission* probabilities, as in $p_{\text{tw}}(w_i \mid t_i)$.

A.1 Sentence Boundaries

To provide a left context for t_1, we define $t_0 =$ ###, representing BOS (the "beginning of sentence" context).

If we generate $t_i =$ ### for $i > 0$, this represents EOS (the "end of sentence" decision):

- Certainly $t_i =$ ### when $i = n$, since our input string always ends at the end of a sentence.
- Possibly $t_i =$ ### for some positions $0 < i < n$, if we allow an input string to consist of multiple sentences. See discussion of this setup in reading section G.2.

For notational convenience, we also have words associated with the boundary tags. We will ensure that $w_i =$ ### if and only if $t_i =$ ###.

B The Forward-Backward Algorithm

The forward-backward algorithm from class is sketched in Figure 2. You may want to review the slides on Hidden Markov Model tagging, and perhaps a textbook exposition as well, such as chapter 6 of Jurafsky & Martin (2nd edition), which specifically discusses our ice cream example.

Figure 2 notes that the posterior probabilities of the possible tag unigrams and bigrams can be computed at lines 13 and 17. When implementing the algorithm, you would ordinarily insert some code at those points to compute and *use* those posterior probabilities. They are used both for the Expectation step (E step) of the Expectation Maximization (EM) algorithm, and for posterior decoding (reading section D below).

To get a better feel for the forward-backward algorithm and its use in EM reestimation, play around with the spreadsheet at `http://www.cs.jhu.edu/~jason/465/hw-hmm/lect24-hmm.xls`. Try changing the red numbers: the initial transition/emission probabilities or the ice cream data. Explore what happens to the various graphs—reconstructed weather on each day and each pair of days, as well as perplexity. Look at what happens to the parameters from iteration to iteration. Study the intermediate computations to see *how* the graphs and parameters are computed. If you click or double-click on any cell, you'll see its formula. All the experiments we did in class are described in this workshop paper.

```
(* build α values from left to right by dynamic programming; they are initially 0 *)
α_###(0) := 1
for i := 1 to n                                                    (* ranges over raw data *)
    for t_i ∈ tag_dict(w_i)
        for t_{i-1} ∈ tag_dict(w_{i-1})
            p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)             (* arc probability *)
            α_{t_i}(i) := α_{t_i}(i) + α_{t_{i-1}}(i-1) · p        (* add prob of all paths ending in t_{i-1}, t_i *)
Z := α_###(n)                                                      (* total prob of all complete paths (from ###,0 to ###,n) *)
(* build β values from right to left by dynamic programming; they are initially 0 *)
β_###(n) := 1
for i := n downto 1
    for t_i ∈ tag_dict(w_i)
        (* now we can compute p(T_i = t_i | w⃗): it is α_{t_i}(i) · β_{t_i}(i)/Z *)
        for t_{i-1} ∈ tag_dict(w_{i-1})
            p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)             (* arc probability *)
            β_{t_{i-1}}(i-1) := β_{t_{i-1}}(i-1) + p · β_{t_i}(i)  (* add prob of all paths starting with t_{i-1}, t_i *)
            (* now we can compute p(T_{i-1} = t_{i-1}, T_i = t_i | w⃗): it is α_{t_{i-1}}(i-1) · p · β_{t_i}(i)/Z *)
```

Figure 2: Sketch of the forward-backward algorithm. We define $w_0 = t_0 =$ ### as a left context for the first tagged word. $\alpha_t(i)$ is the total probability of all paths from the start state (### at time 0) to state t at time i. $\beta_t(i)$ is the total probability of all paths from state t at time i to the final state (### at time n).

C Viterbi decoding

The Viterbi decoding algorithm is sketched in Figure 3. It finds the *single best path through an HMM*—the single most likely weather sequence given the ice cream sequence, or the single most likely tag sequence given the word sequence.

Viterbi tagging is like a parsing problem where you find the single best parse. It is like the forward algorithm, but it uses a different semiring—you max over paths rather than summing over them. This gives you the probability of the best path, instead of the total probability of all paths. You can then follow backpointers (as in parsing) to extract the actual best path.

If you are curious or want to check your implementation, a spreadsheet implementation of the Viterbi algorithm is available at http://www.cs.jhu.edu/~jason/465/hw-hmm/lect24-hmm-viterbi.xls. It's basically the same as the previous spreadsheet, but with a change of semiring. That is, it substitutes "max" for "+", so instead of computing the forward probability α, it computes the Viterbi approximation μ. (This is exactly the same as the distinction between `BEST-PARSE` and `TOTAL-WEIGHT` from HW4!)

However, this means that the spreadsheet version does not actually use backpointers. Instead, it uses μ and ν probabilities, which are the Viterbi approximations to the forward and backward probabilities α and β. Just as $\alpha \cdot \beta$ gives the total probability of *all* paths through a state, $\mu \cdot \nu$ gives the probability of the *best* path through a state. So if at every time step you print out the state with the highest $\mu \cdot \nu$ value, you will have printed out exactly the states on the best path (if the best path is unique).

Backpointers as in Figure 3 are conceptually simpler. The Excel implementation doesn't use them because they would be clumsy to implement in Excel. In a conventional programming language, however, they are both faster and easier for you to implement than the $\mu \cdot \nu$ approach.

D Posterior Decoding

Viterbi decoding prints the single most likely overall sequence. By contrast, a posterior decoder will separately choose the best tag *at each position*—the tag with highest posterior marginal probability—even if this

```
(* find best μ values from left to right by dynamic programming; they are initially 0 *)
μ_###(0) := 1
for i := 1 to n                                          (* ranges over test data *)
    for t_i ∈ tag_dict(w_i)                              (* a set of possible tags for w_i *)
        for t_{i-1} ∈ tag_dict(w_{i-1})
            p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)   (* arc probability *)
            μ := μ_{t_{i-1}}(i − 1) · p                  (* prob of best sequence that ends in t_{i-1}, t_i *)
            if μ > μ_{t_i}(i)                            (* but is it the best sequence (so far) that ends in t_i at time i? *)
                μ_{t_i}(i) = μ                           (* if it's the best, remember it *)
                backpointer_{t_i}(i) = t_{i-1}           (* and remember t_i 's predecessor in that sequence *)
(* follow backpointers to find the best tag sequence that ends at the final state (### at time n) *)
t_n := ###
for i := n downto 1
    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 3: 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 $t_i = t$.

gives an unlikely overall sequence. The posterior marginal probabilities can be found efficiently with the forward-backward algorithm.

Here's an example of how posterior decoding works. Suppose you have a 2-word string, and the HMM assigns positive probability to three different tag sequences, as shown at the left of this table:

			score if predicted sequence is …				
prob	actual sequence		`N V`	`Det Adj`	`Det N`	`Det V`	…
0.45	`N`	`V`	2	0	0	1	…
0.35	`Det`	`Adj`	0	2	1	1	…
0.2	`Det`	`N`	0	1	2	1	…
expected score			0.9	0.9	0.75	1.0	…

The Viterbi decoder will return `N V` because that's the most probable tag sequence. However, the HMM itself says that this has only a 45% chance of being correct. There are two other possible answers, as shown by the rows of the table, so `N V` might be totally wrong.

So is `N V` a good output for our system? Suppose we will be evaluated by the number of correct tags in the output. The `N V` column shows how many tags we might get right if we output `N V`: we have a 45% chance of getting 2 tags right, but a 55% chance of getting 0 tags right, so *on average* we expect to get only 0.9 tags right. The `Det Adj` or `Det N` columns show how many tags we'd expect to get right if we predicted those sequences instead.

It's not hard to see that with this evaluation setup, the best way to maximize our score is to separately predict the most likely tag at every position. We predict $t_1 =$ `Det` because that has a 0.55 chance of being right, so it adds 0.55 to the expected score. And we predict $t_2 =$ `V` because that has an 0.45 chance of being right, so it adds 0.45—more than if we had chosen `Adj` or `N`.

Thus, our best output is `Det V`, where *on average* we expect to get 1.0 tags right. This is not the highest-probability output—in fact it has probability 0 of being correct, according to the HMM! (That's why there's no `Det V` row in the table.) It's just a *good compromise* that is likely to get a pretty good score. It can never achieve the maximum score of 2 (only the three rows in the table can do that), but it also is never completely wrong with a score of 0.

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
‘	Open quotation mark
’	Close quotation mark
$	Currency symbol

Figure 4: Tags in the **en** dataset.

E Data Resources for the Assignment

There are two datasets, available in `/home/arya/hw-hmm` (updated 10/25 because of @481) on the ugrad machines.

ic: Ice cream cone sequences with 1-character tags (C, H). Start with this easy dataset. This is just for your convenience in testing your code.
en: English word sequences with 1-character tags (documented in Figure 4).

Each dataset consists of three files:

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. No peeking!
raw: untagged data for reestimating parameters (100,000 words for **en**)

File Format Each line has a single word/tag pair separated by the / character. (In the **raw** file, only the word appears.) Punctuation marks count as words. The special word ### is used for sentence boundaries, and is always tagged with ###.

(When you generate ###, that represents a decision to end a sentence (so ### = EOS). When you then generate the next tag, conditioning on ### as the previous tag means that you're at the beginning of a new sentence (so ### = BOS).)

Debug your code using the artificial **ic** dataset. This dataset is small and should run fast. More important, it is designed so you can check your work: when you run the forward-backward algorithm, the initial parameters, intermediate results, and perplexities should all agree *exactly* with the results on the spreadsheet we used in class. (But please realize that "in the real world," no one is going to hand you the correct results like this, nor offer any other easy way of detecting bugs in your statistical code.)

F Measuring Tagging Performance

There are various metrics that you could report to measure the quality of a part-of-speech tagger.

F.1 Accuracy

In these task-specific metrics, you look at some subset of the **test** tokens and ask what percentage of them received the correct tag.

accuracy looks at all test tokens, except for the sentence boundary markers. (No one in NLP tries to take credit for tagging ### correctly with ###!)

known-word accuracy considers only tokens of words (other than ###) that also appeared in **train**. So we have observed some possible parts of speech.

seen-word accuracy considers tokens of words that did not appear in **train**, but did appear in **raw** untagged data. Thus, we have observed the words in context and have used EM to try to infer their parts of speech.

novel-word accuracy considers only tokens of words that did *not* appear in **train** or **raw**. These are hard to tag, since context at test time is the only clue to the correct tag. But they are about 9% of all tokens in `entest`, so it is important to tag them as accurately as possible.

`vtag_em.py` must also give the perplexity per *untagged* raw word. This is defined on **raw** data $\vec{w}$ as

$$\exp\left(-\frac{\log p(w_1, \ldots w_n \mid w_0)}{n}\right)$$

Note that this does not mention the tags for raw data, which we don't even know. It is easy to compute, since you found $Z = p(w_1, \ldots w_n \mid w_0)$ while running the forward-backward algorithm (Figure 2, line 8). It is the total probability of *all* paths (tag sequences compatible with the dictionary) that generate the raw word sequence.

F.2 Perplexity

As usual, perplexity is a useful task-independent metric that may correlate with accuracy.

Given a tagged corpus, the model's perplexity per tagged word is given by

$$\begin{aligned} \text{perplexity per tagged word} &= \exp\left(-\text{log-likelihood per tagged word}\right) && (1)\\ &= \exp\left(-\frac{\log p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0)}{n}\right) && (2) \end{aligned}$$

The logarithm base doesn't really matter as long as it's consistent with the exponent: $e^{-(\log x)/n} = (e^{\log x})^{-1/n} = x^{-1/n} = (2^{\log_2 x})^{-1/n} = 2^{-(\log_2 x)/n}$.

Why is the corpus probability in the formula conditioned on w_0/t_0? Because the model only generates $w_1/t_1, \ldots, w_n/t_n$. You knew in advance that w_0/t_0 = ###/### would be the left context for generating

those tagged words. The model has no distribution $p(w_0, t_0)$. Instead, Figure 3, line 2, explicitly hard-codes your prior knowledge that t_0 =###. This is equivalent to in HW1 and HW4 when everything was conditioned on starting with ROOT.
When you have untagged data, you can also compute the model's perplexity on that:

$$\begin{aligned} \text{perplexity per untagged word} &= \exp\left(-\text{log-likelihood per untagged word}\right) \\ &= \exp\left(-\frac{\log p(w_1, \ldots w_n, \mid w_0 t_0)}{n}\right) \end{aligned} \tag{3}$$

where the forward or backward algorithm can compute

$$p(w_1, \ldots w_n, \mid w_0, t_0) = \sum_{t_1, \ldots, t_n} p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0) \tag{4}$$

Notice that

$$p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0) = p(w_1, \ldots w_n \mid w_0, t_0) \cdot p(t_1, \ldots t_n \mid \vec{w}, t_0) \tag{5}$$

so the tagged perplexity (1) can be regarded as the product of two perplexities—namely, how perplexed is the model by the words (in (3)), and how perplexed is it by the tags given the words?

To *evaluate* a trained model, you should ordinarily consider its perplexity on *test* data. Lower perplexity is better.

On the other hand, models can be trained in the first place to minimize their perplexity on *training* data. As equation (1), this is equivalent to maximizing the model's likelihood (or log-likelihood) on training data. Maximizing the tagged likelihood $p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0)$ corresponds to unsmoothed training on a tagged corpus—as in question 2. Maximizing the untagged likelihood (4) corresponds to unsmoothed training on an *untagged* corpus, and is what EM attempts to do.

Thus, question 5 will ask you to report (3) on untagged training data, simply to track how well EM is improving *its own training objective*. This does not evaluate how well the resulting model will generalize to *test data*, which is why we will also ask you to report (1) on test data.

F.3 Speed

How about speed? My final program was about 300 lines in Perl. Running on `ugrad12`, it handled the final "vtag" task in about 3 seconds, and the final "vtag_em.py " task from question 5 in under 2 minutes. Note that a compiled language would run *far faster*.

G Implementation Hints for Viterbi Tagging

Make sure you really understand the algorithm before you start coding! Perhaps write pseudocode or work out an example on paper. Review the reading or the slides. Coding should be a few straightforward hours of work if you really understand everything and can avoid careless bugs. Pepper `assert` statements throughout your code to avoid those bugs, and write unit tests!

G.1 Steps of the Tagger

Your `vtag.py` program in the assignment should go through the following steps:

1. Read the **train** data and store the counts in a class instance. (Your functions for computing probabilities on demand, such as p_{tw}, should access these tables. Perhaps they should be member functions? In problem 3, you will modify those functions to do smoothing.)

2. Read the **test** data $\vec{w}$ into memory.
3. Follow the Viterbi algorithm pseudocode in Figure 3 to find the tag sequence $\vec{t}$ that maximizes $p(\vec{t}, \vec{w})$.
4. Compute and print the accuracy and perplexity of the tagging. (You can compute the accuracy at the same time as you extract the tag sequence while following backpointers.)

G.2 One Long String

Our HMM describes the probability of a single sentence. We condition on $t_0 =$ `###`, and generate tags and their associated words up until we generate $t_n =$ `###` for some $n > 0$, which ends the sentence. (See reading section A.1).

This implies that we should train on each sentence separately, and tag each sentence separately. A file with 10 sentences then provides 10 independent examples for training or testing.

But as a shortcut, it is convenient to consider the entire **train** file as one long multi-sentence sequence that happens to contain some `###` symbols. Similarly for the **test** file. (See reading section A.1.)

In this case, we may have $t_i =$ `###` for some positions $0 < i < n$. Then the model will choose t_{i+1} to be a good tag for starting the next sentence. So the `###/###` at position i serves not only as the EOS that terminates the preceding sentence, but *also effectively does double duty* as the BOS context for the start of the next sentence.

As a result, the probability of the single long string is the product of the HMM probabilities of the individual sentences. We should get the same results from training and decoding on the single long string as if we had treated the sentences individually.

G.3 Data Structures

- Figure 3 refers to a "tag dictionary" that stores all the 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. But for natural-language problems, you'll generalize this as described in the assignment to derive the tag dictionary from training data.

- Before you start coding, make a list of the data structures you will need to maintain, and choose names for those data structures as well as their access methods. Object-orientation is your friend!

  For example, you will have to look up certain values of $c(\cdots)$. So write down, for example, that you will store the count $c(t_{i-1}, t_i)$ in a table `count_tt` whose elements have names like `count_tt("D","N")`. When you read the training data you will increment these elements.

- You will need some multidimensional tables, indexed by strings and/or integers, to store the training counts and the path probabilities. (E.g., `count_tt("D","N")` above, and $\mu_{\mathtt{D}}(5)$ in Figure 3.) There are various easy ways to implement these:

  - a hash table indexed by a single tuple, such as `("D", "N")` or `("5", "D")`. This works well, and is especially memory-efficient since no space is wasted on nonexistent entries.
  - an ordinary 2-D array. This means you have to convert strings (words or tags) to integers and use those integers as array indices. But this conversion is a simple matter of lookup in a hash table. (High-speed NLP packages do all their internal processing using integers, converting to and from strings only during I/O.)

  It's best to **avoid** an array of hash tables or a hash table of hash tables.

- Conversely, it's a good idea to make use of things in the Python standard library that make sanity-checking your code easier, like the `typing.NamedTuple` class. (It's a lot easier to read `pair.tag` than `pair[0]`, so it'll make finding bugs faster. `0` is a magic number and ought to be avoided.)

G.4 Avoiding Underflow

Probabilities that might be small (such as α and β in Figure 2) should be stored in memory as log-probabilities. Doing this is actually crucial to prevent underflow.[1]

- This handout has been talking in terms of probabilities, but when you see something like $p := p \cdot q$ you should implement it as something like $lp = lp + \log q$, where lp is a variable storing $\log p$. (PLEASE don't name it lp, though.)
- ***Tricky question:*** If p is 0, what should you store in lp? How can you represent that value in your program? You are welcome to use any trick or hack that works.
- *Suggestion:* To simplify your code and avoid bugs, I recommend that you use log-probabilities rather than negative log-probabilities. Then you won't have to remember to negate the output to log or the input to exp. (The convention of negating log-probabilities is designed to keep minus signs out of the *printed numbers*; but when you're coding, it's safer to keep minus signs out of the *formulas and code* instead.)

  Similarly, I recommend that you use natural logarithms ($\log_e$) because they are simpler than $\log_2$, slightly faster, and less prone to programming mistakes.

  Yes, it's conventional to *report* $-\log_2$ probabilities, (the unit here is "bits"). But you can store $\log_e x$ internally, and convert to bits only when and if you print it out: $-\log_2 x = -(\log_e x)/\log_e 2$.
- The forward-backward algorithm requires you to add probabilities, as in $p := p + q$. But you are probably storing these probabilities p and q as their logs, lp and lq.

  You might try to write $lp := \log(\exp lp + \exp lq)$, but the exp operation will probably underflow and return 0—that is why you are using logs in the first place! We previously discussed a workaround. Make sure to handle the special case where $p = 0$ or $q = 0$ (see above).

  Tip: `logsumexp` is available in Python as `scipy.misc.logsumexp`.
- If you want to be slick, you might consider implementing a Probability class for all of this. It should support binary operations *, +, and max. (In Python, you get the operators for free by defining methods `__mult__`, `__add__`, and some of the (in)equality functions.) Also, it should have an `__init__` that turns a real into a Probability, and a method for getting the real value of a Probability.

  Internally, the Probability class stores p as $\log p$, which enables it to represent very small probabilities. It has some other, special way of storing $p = 0$. The implementations of *, +, max need to pay attention to this special case.

[1] At least, if you are tagging the **test** set as one long sentence (see above). Conceivably you might be able to get away without logs if you are tagging one sentence at a time. That's how the ice cream spreadsheet got away without using logs: its corpus was only 33 "words." There is also an alternative way to avoid logs, which you are welcome to use if you care to work out the details. It turns out that for most purposes you only care about the *relative* μ values (or α or β values) at each time step—i.e., up to a multiplicative constant. So to avoid underflow, you can rescale them by an arbitrary constant at every time step, or every several time steps when they get too small.

G.5 Counting Carefully

Your *unigram* counts should not count $w_0/t_0 =$ `###/###`. You should count only the n word/tag unigrams in the training file, namely w_1/t_1 through w_n/t_n. These are generated by the model, whereas w_0/t_0 is given as the starting context (like the ROOT symbol of a PCFG).

For the bigram counts, you will count n tag bigrams, namely the same n tag unigrams in their left contexts. This means that t_0t_1 is a bigram—it is counted in the denominator of $p(t_1 \mid t_0)$.

This setup ensures that the probabilities will sum to 1 as required. Consider that when we estimate $p_{\text{tt}}(t'|t)$ as $\frac{c(t,t')}{c(t)}$ (or some smoothed version), we need $c(t) = \sum_{t'} c(t, t')$. This implies that $c(t)$ should count the occurrences of t in positions $0, \ldots, n-1$. The recipe above for $c(t)$ instead counts the occurrences of t in positions $1, \ldots, n$. Fortunately, that happens to give the same answer, since $t_0 =$ `###` $= t_n$. (This is handy because when we estimate $p_{\text{tw}}(w|t)$ as $\frac{c(t,w)}{c(t)}$, we need $c(t) = \sum_w c(t, w)$, which implies that we should indeed use positions $1, \ldots, n$. So it's nice that one way of computing $c(t)$ works for both purposes, at least under our specific setup here.)

G.6 Tag and Word Vocabularies

The tag vocabulary is all the tag types that appeared at least once in **train**. For simplicity, we will not include an OOV tag. Thus, any novel tag will simply have probability 0 (even with smoothing).[2]

Take the word vocabulary to be all the word types that appeared at least once in **train** ∪ **raw** (or just in **train** if no **raw** file is provided, as in the case of `vtag.py`), plus an OOV type in case any out-of-vocabulary word types show up in **test**.[3]

As in homework 3, you should use the same vocabulary size V throughout a run of your program, that your perplexity results will be comparable to one another. So you need to construct the vocabulary before you Viterbi-tag **test** the first time (even though you have not used **raw** yet in any other way).

G.7 Checking Your Implementation of Tagging

Check your implementation as follows. If you use add-1 smoothing without backoff, then `vtag ictrain ictest` should yield a tagging accuracy of 87.88% or 90.91%,[4] with no novel words and a perplexity per tagged test word of `4.401`.[5] If you turn smoothing off, then the correct results are shown in question 4 of the assignment, and you can use the Viterbi version of the spreadsheet (reading section C)—which doesn't smooth—to check your μ probabilities and your tagging:

[2] Is this simplification okay? How bad is it to assign probability 0 to novel tags?

- Effect on perplexity (reading section F.2): You might occasionally assign probability 0 to the *correct* tagging of a **test** sentence, because it includes novel tag types of probability 0. This yields perplexity of ∞.
- Effect on accuracy (reading section F.1): The effect on accuracy will be minimal, however. The decoder will simply never guess any novel tags. But few **test** tokens require a novel tag, anyway.

[3] It would not be safe to assign 0 probability to novel *words*, because words are actually observed in **test**. If any novel words showed up in **test**, we'd end up computing $p(\vec{t}, \vec{w}) = 0$ for *every* tagging $\vec{t}$ of the test corpus $\vec{w}$, and we couldn't identify the *best* tagging. So we need to hold out some smoothed probability for the OOV word.

[4] Why are there two possibilities? Because the code in Figure 3 breaks ties arbitrarily. Here, there are two tagging paths that disagree on day 27 but have *exactly* the same probability. So backpointer$_{\texttt{H}}(28)$ will be set to `H` or `C` according to how the tie is broken, which depends on whether $t_{27} =$ `H` or $t_{27} =$ `C` is considered first in the loop at line 5.

As a result, you might get an output that agrees with either 29 or 30 of the "correct" tags given by `ictest`. Breaking ties arbitrarily is common practice. It's so rare in real data for two floating-point numbers to be exactly == that the extra overhead of handling ties carefully probably isn't worth it.

[5] A uniform probability distribution over the 7 possible tagged words (`###/###`, `1/C`, `1/H`, `2/C`, `2/H`, `3/C`, `3/H`) would give a perplexity of 7, so 4.401 is an improvement.

- `ictrain` has been designed so that your initial unsmoothed supervised training on it will yield the initial parameters from the spreadsheet (transition and emission probabilities).
- `ictest` has exactly the data from the spreadsheet. Running your Viterbi tagger with the above parameters on `ictest` should produce the same values as the spreadsheet's iteration 0:[6]
  - μ probabilities for each day
  - weather tag for each day (shown on the graph)[7]

H Implementation Hints for Expectation Maximization

H.1 Performing Updates

At lines 13 and 17 of the forward-backward algorithm (Figure 2), you will probably want to accumulate some posterior counts of the form $c_{\text{new}}(t, w)$ and $c_{\text{new}}(t, t)$. Make sure to update all necessary count tables. Also remember to initialize variables appropriately. The updated counts can be used to get new smoothed probabilities for the next iteration of EM.

H.2 Interaction of train and raw Counts

Suppose `accounts/N` appeared 2 times in **train** and the forward-backward algorithm thinks it also appeared 7.8 times in **raw**. Then you should update $c(\texttt{N}, \texttt{accounts})$ from 2 to 9.8, since you believe you have seen it a *total* of 9.8 times. (Why ignore the 2 supervised counts that you're sure of?)

If on the next iteration the forward-backward algorithm thinks it appears 7.9 times in **raw**, then you will need to remember the 2 and update the count to 9.9.

To make this work, you will need to have *three versions* of the $c(t, w)$ table. Indeed, every count table $c(\cdots)$ in `vtag.py`, as well as the token count n,[8] will have to be replaced by three versions in `vtag_em.py`!

original: counts derived from **train** only (e.g., 2)

current: counts being used on the current iteration (e.g., 9.8)

new: counts we are accumulating for the next iteration (e.g., 9.9)

Here's how to use them:

- The functions that compute smoothed probabilities on demand, like $p_{\text{tw}}()$, use **current**.
- As you read the training data at the start of your program, you should accumulate its counts into **current**. When you are done reading the training data, save a copy for later: **original** := **current**.

[6] To check your work, you only have to look at iteration 0, at the left of the spreadsheet. But for your interest, the spreadsheet does do reestimation. It is just like the forward-backward spreadsheet, but uses the Viterbi approximation. Interestingly, this approximation *prevents* it from really learning the pattern in the ice cream data, especially when you start it off with bad parameters. Instead of making gradual adjustments that converge to a good model, it jumps right to a model based on the Viterbi tag sequence. This sequence tends never to change again, so we have convergence to a mediocre model after one iteration. This is not surprising. The forward-backward algorithm is biased toward interpreting the world in terms of its stereotypes and then uses those interpretations to update its stereotypes. But the Viterbi approximation turns it into a blinkered fanatic that is absolutely positive that its stereotypes are correct, and therefore can't learn much from experience.

[7] You won't be able to check your backpointers directly.

[8] Will n really change? Yes: it will differ depending on whether you are using probabilities estimated from just **train** (as on the first iteration) or from **train** ∪ **raw**. This should happen naturally if you maintain n just like the other counts (i.e., do `n++` for every new word token you read, and keep 3 copies).

- Each time you run an iteration of the forward-backward algorithm, you should first set **new** := **original**. The forward-backward algorithm should then add expected **raw** counts into **new**, which therefore ends up holding **train** + **raw** counts.
- Once an iteration of forward-backward has completed, it is finally safe to set **current** := **new**.

H.3 Checking Your Implementation of EM

As a first sanity check on forward-backward, you can confirm that $\sum_t \alpha_t(i) \cdot \beta_t(i)$ is constant over all time steps $i = 0, 1, \ldots, n$. For *any* i, this formula gives $p(\vec{w}) = \sum_{\vec{t}} p(\vec{t}, \vec{w})$. (Why? Because $\alpha_t(i) \cdot \beta_t(i)$ is the sum over just those taggings $\vec{t}$ that have $t_i = t$, so summing that over all possible choices of t gives the sum over all taggings.)

Then, as noted before, you can run `python3 vtag_em.py ictrain ictest icraw` (the ice cream example) to check whether your program is working correctly. Details (there is a catch!):

- `icraw` (like `ictest`) has exactly the data from the spreadsheet. Running the forward-backward algorithm on `icraw` should compute exactly the same values as the spreadsheet does:
  - α and β probabilities
  - perplexity per untagged raw word (i.e., perplexity per observation: see upper right corner of spreadsheet)
- The spreadsheet does not use any supervised training data. To make your code match the spreadsheet, you should temporarily modify it to initialize **original** := 0 instead of **original** := **current**. Then the training set will only be used to find the initial parameters (iteration 0). On subsequent iterations it will be ignored.

  You should also turn off smoothing (just set $\lambda = 0$), since the spreadsheet does not do any smoothing.

  With these changes, your code should compute the same **new** transition and emission counts on every iteration as the spreadsheet does. The new parameters (transition and emission probabilities) will match as well. As a result, the perplexity per untagged raw word should match the values in the upper right corner of the spreadsheet: `3.393`, `2.947`, `2.879`, `2.854`, `2.840`, `2.833`, `2.830`, `2.828`, `2.828`, `2.827`, `2.827`.

  After a few iterations, you should get 100% tagging accuracy on the test set.

  Don't forget to change the code back so you can run it on the the **en** dataset and hand it in!