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. 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.

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

Programming language: As usual, your choice, as long as we can make support for your language available on the autograder. Pick a language in which you can program quickly and well, and which provides fast dictionaries (e.g., hash maps).

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

1. The forward-backward algorithm was given in reading section C, which encouraged you to play with the spreadsheet that we used in class.

Play as much as you like, but here are some questions to answer in your README:

(a) Reload the spreadsheet to go back to the default settings. Now, change the first day to have just 1 ice cream.

i. What is the new probability (in the initial reconstruction) that day 1 is hot? Explain, by considering the probability of HHH versus CHH as explanations of the first three days of data, 133.

ii. How much does this change affect the probability that day 2 is hot? That is, what is that probability before vs. after the change to the day 1 data? What cell in the spreadsheet holds this probability?

iii. How does this change affect the final graph after 10 iterations of reestimation? In particular, what is \( p(H) \) on days 1 and 2? (Hint: Look at where the other 1-ice-cream days fall during the summer.)

(b) We talked about stereotypes in class. Suppose you bring a very strong bias to interpreting the data: you believe that I never eat only 1 ice cream on a hot day. So, again reload the spreadsheet, and set \( p(1 \mid H) = 0 \), \( p(2 \mid H) = 0.3 \), \( p(3 \mid H) = 0.7 \).
i. How does this change affect the initial reconstruction of the weather (the leftmost graph)?

ii. What does the final graph look like after 10 iterations of reestimation?

iii. What is \( p(1 \mid \mathbf{H}) \) after 10 iterations? Explain carefully why this is, discussing what happens at each reestimation step, in terms of the \( 2^{33} \) paths through the trellis.

(c) The backward algorithm (which computes all the \( \beta \) probabilities) is exactly analogous to the inside algorithm. Recall that the inside algorithm finds the probability of a sentence by summing over all possible parses. The backward algorithm finds the probability of a sentence by summing over all possible taggings that could have generated that sentence.

i. Let’s make that precise. Each state (node) in the trellis has a \( \beta \) probability. Which state’s \( \beta \) probability equals the total probability of the sentence?

ii. It is actually possible to regard the backward algorithm as a special case of the inside algorithm! In other words, there is a particular grammar whose parses correspond to taggings of the sentence. One parse of the sequence 2313 would look like the tree on the left:

```
START
  2 C
  3 H
  1 C
  3 H
  ε
```

```
START
  EC
  1 EH H
  2 EH H
  3 EC C
  1 EH H
  3 ε
```

In the tree on the left, what is the meaning of an H constituent? What is the probability of the rule \( H \rightarrow 1 \ C \)? How about the probability of \( H \rightarrow ε \)? An equivalent approach uses a grammar that instead produces the slightly more complicated parse on the right; why might one prefer that approach?

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

```
vtag ictrain ictest
```

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

Your program must summarize the model’s performance on the test set, in the following format (ignore the particular numbers in this example for now). These performance metrics were defined in reading section G.1.

**Tagging accuracy (Viterbi decoding): 92.12%  (known: 95.60%  novel: 56.07%)**

You are also free to print comment lines starting with #. These can report other information that you may want to see, including the tags your program picks, its accuracy as it goes along, various probabilities, etc.

3. Now, you will improve your tagger so that you can run it on real data (reading section F):
This means using a proper tag dictionary (for speed) and smoothed probabilities (for accuracy). Ideally, your tagger should beat the following “baseline” result:

**Model perplexity per tagged test word:** 2649.489  
**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 G.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 stupid *unigram* Viterbi tagger in which the *bigram* transition probabilities are replaced by *unigram* transition probabilities. The bigram case is called a 1st-order HMM, and a fancier trigram version would be called a 2nd-order HMM; so the unigram baseline can be called a 0th-order HMM.

The baseline tagger can be really fast. If you think about what a 0th-order HMM does, it just tags every known word with its most common part of speech from training data, which can be looked up in a hash table. It tags every novel word with the most common part of speech overall, namely N. Since context is not considered, no dynamic programming is needed; the overall runtime is $O(n)$. 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*!

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.

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.)*

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, that sequence will still have positive smoothed probability, which you should evaluate for the perplexity number.

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-$\lambda$ 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!

---

1On 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*.
2Ties are broken arbitrarily.
3Comparison to a previous baseline is generally required when reporting NLP results.
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 so that it tries posterior decoding (reading section E) once it’s done with Viterbi decoding. At the end of vtag, 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 vtag 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 vtag ictrain ictest should now look like this:

Model perplexity per tagged test word: 4.111
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 $p(H) > 0.5$.

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 vtag 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.

Turn in the source code for your current version of vtag. (Remember, for full credit, this should include a tag dictionary, and both Viterbi and posterior decoders; for extra credit, it can include one-count smoothing.) In README give your observations and results, including the output from running vtag entrain entest (or at least the required lines from that output).

5. Now try to improve your HMM tagger to do well on the leaderboard. Beating the accuracy of the unigram baseline is required. Better solutions will get more credit.

For example, you could improve the smoothing method. One relatively simple method to consider is “one-count smoothing” (reading section I). This uses backoff and should improve your performance considerably.

Or you could try using a higher-order HMM, e.g., trigram transition probabilities instead of bigrams.
Or you could use a more interesting emission model that considers word spellings (e.g., word endings) to help with novel words. Maybe the word "actuary" is a noun because it ends in "ry," or because "actuaries" appears in training data as a noun. Feel free to talk to the course staff about how to structure such a model.

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 3)? Answer in your README.

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 actual grading, however, we will evaluate your code on “final-test” data that you have never seen (as in Assignment 3). The autograder will run your code to both train and test your model. It will compare the actual test-output generated by your posterior decoder to the gold-standard tags on the final-test data.

6. Now let’s try the EM algorithm. Copy vtag to a new program, vtagem, and modify it to reestimate the HMM parameters on raw (untagged) data. You should be able to run it as

    vtagem 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. (Don’t bother running it on the test data anymore—unlike vtag, vtagem does not have to report posterior decoding on test data.)

The forward-backward algorithm was given in reading section C and some implementation hints for EM were given in reading section J.

The program should run at least 3 iterations of EM. Its output format should be as shown in Figure 1.

Note that vtagem'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.

Or if you prefer, you can submit a single program that behaves like vtag when given two files (train, test) and like vtagem when given three files (train, test, raw).
[read train]
[read test]
[read raw]
[decode the test data]

Model perplexity per tagged test word: ...
Tagging accuracy (Viterbi decoding): ...% (known: ...% seen: ...% novel: ...%)
Tagging accuracy (posterior 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: ...%)
Tagging accuracy (posterior 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: ...%)
Tagging accuracy (posterior 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: ...%)
Tagging accuracy (posterior 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 vtagem. Your program should include the lines shown in this font, and any comment lines starting with # that you find helpful. The material in [brackets] is not necessarily part of the output; it just indicates what your program would be doing at each stage.

Submit the source code for vtagem. In README, include the output from running vtagem 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. 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?
As mentioned in class, 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 at http://aclweb.org/anthology/J94-2001 has been cited 700 times, often by people who are attempting to build better unsupervised learners!

7. Extra credit: vtagem will be quite slow on the cz dataset. Why? Czech is a morphologically complex language: each word contains several morphemes. Since its words are more complicated, more of them are unknown (50% instead of 9%) and we need more tags (66 instead of 25).\(^5\) So there are \((66/25)^2 \approx 7\) times as many tag bigrams ... and the worst case of two unknown words in a row (which forces us to consider all those tag bigrams) occurs far more often.

Speed up vtagem by implementing some kind of tag pruning during the computations of \(\mu\), \(\alpha\), and \(\beta\). (Feel free to talk to the course staff about your ideas.) Submit your improved source code. Answer the following questions in your README:

(a) Using your sped-up program, what accuracy and perplexity do you obtain for the cz dataset?
(b) Estimate your speedup on the en and cz datasets.
(c) But how seriously does pruning hurt your accuracy and perplexity? Estimate this by testing on the en dataset with and without pruning.
(d) How else could you cope with tagging a morphologically complex language like Czech? You can assume that you have a morphological analyzer for the language.

For comparison, my Perl tagger had the following runtimes:

<table>
<thead>
<tr>
<th></th>
<th>English Viterbi (only)</th>
<th>Czech Viterbi (only)</th>
<th>English EM</th>
</tr>
</thead>
<tbody>
<tr>
<td>no pruning</td>
<td>15 sec.</td>
<td>14 min.</td>
<td>17.5 min</td>
</tr>
<tr>
<td>light pruning</td>
<td>10 sec.</td>
<td>8.5 min.</td>
<td>6 min.</td>
</tr>
<tr>
<td>strong pruning</td>
<td>8 sec.</td>
<td>6.5 min.</td>
<td>5.5 min.</td>
</tr>
<tr>
<td>aggressive pruning</td>
<td>5 sec.</td>
<td>2 min.</td>
<td>4 min.</td>
</tr>
</tbody>
</table>

Using light to strong pruning didn’t change the accuracy and perplexity much.

\(^5\)Although both tagsets have been simplified for this homework. The Czech tags were originally 6 letters long, and were stripped down to 2 letters. The simplification of the English tags was already described in the caption to Figure 4.
A Semi-Supervised Learning

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 try to 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). 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 that I mentioned in class.

How about speed? My final program was about 250 lines in Perl. Running on ugrad5, it handled the final “vtag” task in about 30 seconds and 35M memory, and the final “vtagem” task in about 13 minutes and 140M memory. (A simple pruning step cuts the latter time to 6 minutes with virtually no effect on results, but you are not required to implement this.) Note that a compiled language should run far faster.

B Notation

In this reading handout, I’ll use the following notation. You might want to use the same notation in your program.

- The input string consists of $n + 1$ words, $w_0, w_1, \ldots w_n$.
- The corresponding tags are $t_0, t_1, \ldots t_n$. We have $w_i/t_i = \#\#\#/####$ for $i = 0$, for $i = n$, and probably also for some other values of $i$.
- I’ll use “tt” to name tag-to-tag transition probabilities, as in $p_{tt}(t_i \mid t_{i-1})$.
- I’ll use “tw” to name tag-to-word emission probabilities, as in $p_{tw}(w_i \mid t_i)$.

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.
1. (* build $\alpha$ values from left to right by dynamic programming; they are initially $0$ *)
2. $\alpha_{###}(0) := 1$
3. for $i := 1$ to $n$ (* ranges over raw data *)
4. for $t_i \in \text{tag_dict}(w_i)$
5. for $t_{i-1} \in \text{tag_dict}(w_{i-1})$
6. $p := p_{tt}(t_i \mid t_{i-1}) \cdot p_{tw}(w_i \mid t_i)$ (* arc probability *)
7. $\alpha_{t_i}(i) := \alpha_{t_i}(i) + \alpha_{t_{i-1}}(i-1) \cdot p$ (* add prob of all paths ending in $t_{i-1}, t_i$ *)
8. $Z := \alpha_{###}(n)$ (* total prob of all complete paths (from ###,0 to ###,n *)
9. (* build $\beta$ values from right to left by dynamic programming; they are initially $0$ *)
10. $\beta_{###}(n) := 1$
11. for $i := n$ downto $1$
12. for $t_i \in \text{tag_dict}(w_i)$
13. (* now we can compute $p(T_i = t_i \mid \vec{w})$: it is $\alpha_{t_i}(i) \cdot \beta_{t_i}(i)/Z$ *)
14. for $t_{i-1} \in \text{tag_dict}(w_{i-1})$
15. $p := p_{tt}(t_i \mid t_{i-1}) \cdot p_{tw}(w_i \mid t_i)$ (* arc probability *)
16. $\beta_{t_{i-1}}(i-1) := \beta_{t_{i-1}}(i-1) + p \cdot \beta_{t_i}(i)$ (* add prob of all paths starting with $t_{i-1}, t_i$ *)
17. (* now we can compute $p(T_{i-1} = t_{i-1}, T_i = t_i \mid \vec{w})$: it is $\alpha_{t_{i-1}}(i-1) \cdot p \cdot \beta_{t_i}(i)/Z$ *)

Figure 2: Sketch of the forward-backward algorithm. $\alpha_{t_i}(i)$ is the total probability of all paths from the start state (### at time 0) to state $t$ at time $i$. $\beta_{t_i}(i)$ is the total probability of all paths from state $t$ at time $i$ to the final state (### at time $n$).

C 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. If you want a deeper mathematical understanding of what’s going on, try “Inside-outside and forward-backward algorithms are just backprop.”

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 E 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/papers/eisner.tnlp02.pdf.

D 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

2Excel displays these spreadsheets correctly, and LibreOffice or OpenOffice does a decent job as well.
1. (* find best $\mu$ values from left to right by dynamic programming; they are initially 0 *)
2. $\mu^{###}(0) := 1$
3. for $i := 1$ to $n$ (* ranges over test data *)
4. for $t_i \in$ tag.dict($w_i$) (* a set of possible tags for $w_i$ *)
5. for $t_{i-1} \in$ tag.dict($w_{i-1}$)
6. $p := p_{tt}(t_i | t_{i-1}) \cdot p_{tw}(w_i | t_i)$ (* arc probability *)
7. $\mu := \mu_{t_{i-1}}(i - 1) \cdot p$ (* prob of best sequence that ends in $t_{i-1}, t_i$ *)
8. if $\mu > \mu_{t_i}(i)$ (* but is it the best sequence (so far) that ends in $t_i$ at time $i$? *)
9. $\mu_{t_i}(i) = \mu$ (* 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 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 $p(t_1, w_1, t_2, w_2, \ldots t_i, w_i | t_0, w_0)$ over all possible choices of $t_1, \ldots t_i$ such that $t_i = t$.

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 $\alpha$, it computes the Viterbi approximation $\mu$.

However, this means that the spreadsheet version does not actually use backpointers. Instead, it uses $\mu$ and $\nu$ probabilities, which are the Viterbi approximations to the forward and backward probabilities $\alpha$ and $\beta$. 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 (at least if the best path is unique).

Backpointers as in Figure 3 are conceptually simpler. They would be clumsy to implement in Excel. However, and in a conventional programming language they are both faster and easier for you to implement than the $\mu \cdot \nu$ approach.

### E Posterior Decoding

Recall that 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 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 (repeated from the HMM slides in class).
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:

<table>
<thead>
<tr>
<th>prob</th>
<th>actual sequence</th>
<th>score if predicted sequence is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.45</td>
<td>N V</td>
<td>2 0 0 1 ...</td>
</tr>
<tr>
<td>0.35</td>
<td>Det Adj</td>
<td>0 2 1 1 ...</td>
</tr>
<tr>
<td>0.2</td>
<td>Det N</td>
<td>0 1 2 1 ...</td>
</tr>
</tbody>
</table>

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 \( t_1 = \text{Det} \) because that has a 0.55 chance of being right, so it adds 0.55 to the expected score. And we predict \( t_2 = \text{V} \) 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.

F Data Resources for the Assignment

There are three datasets, available in `/usr/local/data/cs465/hw-hmm/data` on the ugrad machines (or at [http://cs.jhu.edu/~jason/465/hw-hmm/data](http://cs.jhu.edu/~jason/465/hw-hmm/data)). You may like to develop your programs on the ugrad machines rather than copying the data elsewhere.

The datasets are

- **ic**: Ice cream cone sequences with 1-character tags (C, H). Start with this easy dataset.
- **en**: English word sequences with 1-character tags (documented in Figure 4).
- **cz**: Czech word sequences with 2-character tags. (If you want to see the accented characters more-or-less correctly, look at the files in Emacs.)

In the assignment, you will only need to hand in results on the **en** dataset. The others are just for your convenience in testing your code, and for the extra credit problem.

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.
Figure 4: Tags in the en dataset. These are the preterminals from wallstreet.gr in assignment 3, but stripped down to their first letters. For example, all kinds of nouns (formerly NN, NNS, NNP, NNPS) are simply tagged as N in this assignment. Using only the first letters reduces the number of tags, speeding things up and increasing accuracy. (However, it results in a couple of unnatural categories, C and P.)

- **raw**: untagged data for reestimating parameters (100,000 words for en)

The file format is quite simple. 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 ###.

You are strongly encouraged to test 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.

**G Measuring Tagging Performance**

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

**G.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 very hard to tag, since context at test time is the only clue to the correct tag. But they constitute about 9% of all tokens in entest, so it is important to tag them as accurately as possible.

vtagem’s output must also include the perplexity per untagged raw word. This is defined on raw data $\tilde{w}$ as

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

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 $Z = p(w_1,\ldots,w_n | w_0)$ 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.

### G.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

$$\text{perplexity per tagged word} = 2^{\text{cross-entropy per tagged word}}$$

where

$$\text{cross-entropy per tagged word} = \frac{-\log_2 p(w_1,t_1,\ldots,w_n,t_n | w_0,t_0)}{n}$$

Since the power of 2 and the log base 2 cancel each other out, you can equivalently write this using a power of $e$ and log base $e$:

$$\text{perplexity per tagged word} = \exp\left(-\frac{\log p(w_1,t_1,\ldots,w_n,t_n | w_0,t_0)}{n}\right)$$

This is equivalent because $e^{-(\log x)/n} = (e^{\log x})^{-1/n} = x^{-1/n} = (2^{\log_2 x})^{-1/n} = 2^{-\log_2 x}/n$.

Why is the corpus probability in the formula conditioned on $w_0,t_0$? Because you knew in advance that the tagged test corpus would start with ###/###—your model is only predicting the rest of that corpus. (The model has no parameter that would even tell you $p(w_0,t_0)$. Instead, Figure 3, line 2, explicitly hard-codes your prior knowledge that $t_0 =$$##$.)

---

$^4$Using the notation from reading section B.
When you have untagged data, you can also compute the model’s perplexity on that:

\[
\text{perplexity per untagged word} = \exp\left(\frac{-\log p(w_1, \ldots, w_n, t_1, \ldots, t_n | w_0, t_0)}{n}\right)
\]

where the forward or backward algorithm can compute

\[
p(w_1, \ldots, w_n, t_1, \ldots, t_n | w_0, t_0) = \sum_{t_1, \ldots, t_n} p(w_1, t_1, \ldots, w_n, t_n | w_0, t_0)
\]

Notice that

\[
p(w_1, t_1, \ldots, w_n, t_n | w_0, t_0) = p(w_1, \ldots, w_n | w_0, t_0) \cdot p(t_1, \ldots, t_n | \vec{w}, t_0)
\]

so the tagged perplexity (2) can be regarded as the product of two perplexities—namely, how perplexed is the model by the words (in (4)), 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 (2), this is equivalent to maximizing the model’s likelihood (or log-likelihood) on training data. Maximizing the tagged likelihood \(p(w_1, t_1, \ldots, w_n, t_n | w_0, t_0)\) corresponds to unsmoothed training on a tagged corpus—as in question 2 on the assignment. Maximizing the untagged likelihood (5) corresponds to unsmoothed training on an untagged corpus, and is what EM attempts to do.

Thus, question 6 on the assignment will ask you to report (4) 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 (2) on test data.

**H 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.

**H.1 Steps of the Tagger**

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

1. Read the train data and store the counts in global tables. (Your functions for computing probabilities on demand, such as \(p_{tw}\), should access these tables. 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.)
H.2 One Long String

Don’t bother to train on each sentence separately, or to tag each sentence separately. Just treat the train file as one long string that happens to contain some ### words. Similarly for the test file.

Tagging sentences separately would save you memory, since then you could throw away each sentence (and its tag probabilities and backpointers) when you were done with it. But why bother if you seem to have enough memory? Just pretend it’s one long sentence. Worked for me.

H.3 Fixing the Vocabulary

As in homework 3, you should use the same vocabulary size $V$ for all your computations, so that your perplexity results will be comparable to one another. So you need to compute it before you Viterbi-tag test the first time (even though you have not used raw yet in any other way).

Take the vocabulary to be all the words that appeared at least once in train, plus an OOV type. (Once you do EM, your vocabulary should also include the words that appeared at least once in raw.)

H.4 Data Structures

Some suggestions:

- 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: \( \text{tag\_dict}(###) = \{###\} \), and \( \text{tag\_dict}(w) = \{C,H\} \) 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.

  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 \( \text{count\_tt} \) whose elements have names like \( \text{count\_tt}("D","N") \). 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., \( \text{count\_tt}("D","N") \) above, and \( \mu_D(5) \) in Figure 3.) There are various easy ways to implement these:

  - a hash table indexed by a single string that happens to have two parts, such as "D/N" or "5/D". This works well, and is especially memory-efficient since no space is wasted on nonexistent entries.

  - a hash table of arrays. This wastes a little more space.

  - an ordinary multidimensional array (or array of arrays). 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. It is slow and wasteful of memory to have many small hash tables. Better to combine them into one big hash table as described in the first bullet point above.
H.5 Avoiding Underflow

Probabilities that might be small (such as \( \alpha \) and \( \beta \) in Figure 2) should be stored in memory as log-probabilities. Doing this is actually crucial to prevent underflow.

- 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 \( \log p \).

- **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 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: \( -\log_2 x = -(\log_e x)/\log_2 e \) (As it happens, you are not required to print any log-probabilities for this assignment, only perplexities: see equation (3)).

- 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!

Instead you need to write \( lp := \logsumexp(lp, lq) \), where

\[
\logsumexp(x, y) \overset{\text{def}}{=} \begin{cases} 
  x + \log(1 + \exp(y - x)) & \text{if } y \leq x \\
  y + \log(1 + \exp(x - y)) & \text{otherwise}
\end{cases}
\]

You can check for yourself that this equals \( \log(\exp x + \exp y) \); that the exp can’t overflow (because its argument is always \( \leq 0 \)); and that you get an appropriate answer even if the exp underflows.

The sub-expression \( \log(1+z) \) can be computed more quickly and accurately by the specialized function \( \log1p(z) = z - z^2/2 + z^3/3 - \cdots \) (Taylor series), which is usually available in the math library of your programming language (or see [http://www.johndcook.com/cpp_log_](http://www.johndcook.com/cpp_log_)).

---

4 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 \( \mu \) values (or \( \alpha \) or \( \beta \) 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.

5 The IEEE floating-point standard does have a way of representing \( -\infty \), so you could genuinely set \( lp = -\text{Inf} \), which will work correctly with \(+, >, \text{ and } \geq\). Or you could just use an extremely negative value. Or you could use some other convention to represent the fact that \( p = 0 \), such as setting a boolean variable \( p\_is\_zero \) or setting \( lp \) to some special value (e.g., \( lp = \text{undef} \) or \( lp = \text{null} \) in a language that supports this, or even \( lp = +9999 \), since a positive value like this will never be used to represent any other log-probability).
This avoids ever computing 1 + z, which would lose most of z’s significant digits for small z.

Make sure to handle the special case where p = 0 or q = 0 (see above).

Tip: logsumexp is available in Python as numpy.logaddexp.

- If you want to be slick, you might consider implementing a Probability class for all of this. It should support binary operations *, +, and max. Also, it should have a constructor 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.

You’re not required to write a class (or even to use an object-oriented language). You may prefer just to inline these simple methods. But even so, the above is a good way of thinking about what you’re doing.

H.6 Counting Carefully

To be careful and obtain precisely the sample results provided in this assignment, your unigram counts should skip the training file's very first (or very last) ###/###.

So even though the training file appears to have n + 1 word/tag unigrams, you should only count n of these unigrams. This matches the fact that there are n bigrams.

This counting procedure also slightly affects c(t), c(w), and c(t, w).

Why count this way? Because doing so makes the smoothed (or unsmoothed) probabilities sum to 1 as required.

The root of the problem is that there are n + 1 tagged words but only n tag-tag pairs. Omitting one of the boundaries arranges that \( \sum c(t), \sum w c(w), \) and \( \sum t, w c(t, w) \) all equal \( n \), just as \( \sum t, t' c(t, t') = n \).

To see how this works out in practice, suppose you have the very short corpus

```
#/# H/3 C/2 #/# C/1 #/#
```

(I’m abbreviating ### as # in this section)

I’m just saying that you should set \( c(\#) = 2 \), not \( c(\#) = 3 \), for both the tag # and the word #. Remember that probabilities have the form \( p(event | context) \). Let’s go through the settings where \( c(\#) \) is used:

- **In an expression** \( p_{tt}(\cdot | \#) \), where we are transitioning from #. You want unsmoothed \( p_{tt}(H | \#) = \frac{c(H\#)}{c(\#)} = \frac{1}{2} \) (not \( \frac{1}{3} \)), because # only appears twice as a context (representing BOS at positions 0 and 3—the EOS at position 5 is not the context for any event).

- **In an expression** \( p_{tt}(\# | \cdot) \), where we are transitioning to #. Now, unsmoothed \( p_{tt}(\# | C) = \frac{c(C\#)}{c(C)} = \frac{2}{2} \) doesn’t use \( c(\#) \). But it backs off to \( p_{tt}(\#) \), and you want unsmoothed \( p_{tt}(\#) = \frac{c(\#)}{n} = \frac{2}{5} \) (not \( \frac{3}{5} \)), because # only appears twice as an event (representing EOS at positions 3 and 5—the BOS at position 0 is not an event but rather is given “for free” as the context of the first event, like the ROOT symbol in a PCFG).

- **In an expression** \( p_{tw}(\# | \#) \), where we are emitting a word from #. Reading section H.7 notes that there’s no point in smoothing this probability or even estimating it from data: you know it’s 1! But if you were to estimate it, you would want to count the unsmoothed value
\( \frac{c(\#\#)}{c(\#\#)} \) should be counted as \( \frac{2}{2} \) (not \( \frac{3}{3} \), and certainly not \( \frac{3}{2} \)). This is best interpreted as saying that no word was ever emitted at position 0. The only thing that exists at position 0 is a \text{bos} state that transitions to the \( H \) state.

When reestimating counts using raw data in problem 6, you should similarly ignore the initial or final \#\#/\#\# in the raw data.

H.7 Don’t Guess When You Know

Don’t smooth \( p_{tw}(w_i = \#\# \ | \ t_i = \#\#) \). This probability doesn’t have to be estimated from data. It should always be 1, because you know—without any data—that the \#\# tag always emits the \#\# word.

Indeed, the only reason we even bother to specify \#\# words in the file, not just \#\# tags, is to make your code simpler and more uniform: “treat the file as one long string that happens to contain some \#\# words.”

H.8 Checking Your Implementation

Check your work as follows. vtag ictrain ictest should yield a tagging accuracy of 87.88% or 90.91%, with no novel words and a perplexity per tagged test word of 4.326. You can use the Viterbi version of the spreadsheet (reading section D) to check your \( \mu \) probabilities and your tagging.

- **ictrain** has been designed so that your initial 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 on these data should produce the same values as the spreadsheet’s iteration 0.\(^6\)

---

\(^6\)Why are there two possibilities? Because the code in Figure 3 breaks ties arbitrarily. In this example, there are two tagging paths that disagree on day 27 but have exactly the same probability. So backpointer\textsubscript{a}(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. (Since line 8 happens to be written with a strict inequality \( > \), the tie will arbitrarily be broken in favor of the first one we try; the second one will not be strictly better and so will not be able to displace it. Using \( \geq \) at line 8 would instead break ties in favor of the last one we tried.)

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.

Ideally, though, a Viterbi tagger would output both taggings in this unusual case, and give an average score of 29.5 correct tags. This is how you handled ties on HW3. However, keeping track of multiple answers is harder in the Viterbi algorithm, when the answer is a whole sequence of tags. You would have to keep multiple backpointers at every point where you had a tie. Then the backpointers wouldn’t define a single best tag string, but rather, a skinny FSA that weaves together all the tag strings that are tied for best. The output of the Viterbi algorithm would then actually be this skinny FSA. (Or rather its reversal, so that the strings go left-to-right rather than right-to-left.) When I say it’s ”skinny,” I mean it is pretty close to a straight-line FSA, since it usually will only contain one or a few paths. To score this skinny FSA and give partial credit, you’d have to compute, for each tag, the fraction of its paths that got the right answer on that tag. How would you do this efficiently? By running the forward-backward algorithm on the skinny FSA!

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 4.326 is an improvement.

\(^8\)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
- \( \mu \) probabilities for each day
- weather tag for each day (shown on the graph)\(^9\)

## I One-Count Smoothing (not required)

One-count smoothing is basically just add-\( \lambda \) smoothing with backoff, but \( \lambda \) is set higher in contexts with a lot of “singletons”—words that have only occurred once—because such contexts are likely to have novel words in test data. This is called “one-count” smoothing.\(^{10}\)

First let us define our backoff estimates:

- Let
  \[
  p_{tt\text{-backoff}}(t_i \mid t_{i-1}) = p_{t\text{-unsmoothed}}(t_i) = \frac{c(t_i)}{n}
  \]
  Do you see why it’s okay to back off to this totally unsmoothed, maximum likelihood estimate?\(^{11}\) I’ll explain below why the denominator is \( n \) rather than \( n + 1 \), even though there are \( n + 1 \) tokens \( t_0, t_1, \ldots t_n \).

- Let
  \[
  p_{tw\text{-backoff}}(w_i \mid t_i) = p_{w\text{-addone}}(w_i) = \frac{c(w_i) + 1}{n + V}
  \]
  This backoff estimate uses add-one-smoothing. \( n \) and \( V \) denote the number of word tokens \( tokens \) and types \( types \), respectively, that were observed in training data. (In addition, \( V \) includes an OOV type. Again, I’ll explain below why the token count is taken to be \( n \) even though there are \( n + 1 \) tokens \( t_0, t_1, \ldots t_n \).)

Notice that according to this formula, any novel word has count 0 and backoff probability \( p_{w\text{-addone}} = \frac{1}{n+V} \). In effect, we are following assignment 2 and 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.

Now for the smoothed estimates:

- Define a function \( sing \) that counts singletons. Let
  \[
  \begin{align*}
  sing_{tt}(\cdot \mid t_{i-1}) &= \text{number of tag types } t \text{ such that } c(t_{i-1}, t) = 1 \\
  sing_{tw}(\cdot \mid t_i) &= \text{number of word types } w \text{ such that } c(t_i, w) = 1
  \end{align*}
  \]
  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.

\(^9\)You won’t be able to check your backpointers directly.
\(^{10}\)Many smoothing methods use the probability of singletons to estimate the probability of novel words, as in Good-Turing smoothing and in one of the extra-credit problems on HW3. The “one-count” method is due to Chen and Goodman, who actually give it in a more general form where \( \lambda \) is a linear function of the number of singletons. This allows some smoothing to occur \( (\lambda > 0) \) even if there are no singletons \( (sing = 0) \). Chen and Goodman recommend using held-out data to choose the slope and intercept of the linear function.
\(^{11}\)It’s because tags are not observed in the test data, so we can safely treat novel tag unigrams as impossible (probability 0). This just means that we will never guess a tag that we didn’t see in training data—which is reasonable. By contrast, it would not be safe to assign 0 probability to novel words, because words are actually observed in the test data: if any novel words showed up there, we’d end up computing \( p(\vec{t}, \vec{w}) = 0 \) probability for every tagging \( f \) of the test corpus \( \vec{w} \). So we will have to smooth \( p_{tw\text{-backoff}}(w_i \mid t_i) \) below; it is only \( p_{tt\text{-backoff}}(t_i \mid t_{i-1}) \) that can safely rule out novel events.
There is an easy way to accumulate these singleton counts during training. Whenever you increment $c(t, w)$ or $c(t, t)$, check whether it is now 1 or 2. If it is now 1, you have just found a new singleton and you should increment the appropriate singleton count. If it is now 2, you have just lost a singleton and you should decrement the appropriate singleton count.

- Notice that $\text{sing}_{tw}(\cdot \mid N)$ will be high because many nouns only appeared once. This suggests that the class of nouns is open to accepting new members and it is reasonable to tag new words with $N$ too. By contrast, $\text{sing}_{tw}(\cdot \mid D)$ will be 0 or very small because the class of determiners is pretty much closed—suggesting that novel words should not be tagged with $D$. We will now take advantage of these suggestions.

- Let
  
  \begin{align*}
  p_{tt}(t_i \mid t_{i-1}) &= \frac{c(t_{i-1}, t_i) + \lambda \cdot p_{tt}^{-\text{backoff}}(t_i \mid t_{i-1})}{c(t_{i-1}) + \lambda} \quad \text{where } \lambda = 1 + \text{sing}_{tt}(\cdot \mid t_{i-1}) \\
  p_{tw}(w_i \mid t_i) &= \frac{c(t_i, w_i) + \lambda \cdot p_{tw}^{-\text{backoff}}(w_i \mid t_i)}{c(t_i) + \lambda} \quad \text{where } \lambda = 1 + \text{sing}_{tw}(\cdot \mid t_i)
  \end{align*}

  Note that $\lambda$ will be higher for $p_{tw}(\cdot \mid N)$ than for $p_{tw}(\cdot \mid D)$. Hence $p_{tw}(\cdot \mid N)$ allows more backoff, other things equal, and so assigns a higher probability to novel words.

  If one doesn’t pay respect to the difference between open and closed classes, then novel words will often get tagged as $D$ (for example) in order to make neighboring words happy. Such a tagger does worse than the baseline tagger (which simply tags all novel words with the most common singleton tag, $N$)!

  Note that $\lambda$ is a linear function of the number of singletons (as in footnote 10). Since this function ensures that $\lambda > 0$ (even if the number of singletons is 0), our estimated probability will never be 0 or 0/0.

You can test your code on the ic dataset. Here’s what I got with vtag ictrain ictest:

Tagging accuracy (Viterbi decoding): 90.91%  (known: 90.91%  novel: 0.00%)
Model perplexity per tagged test word: 3.689

Since the ic dataset happens to have no singletons at all, you’ll always have $\lambda = 1$ (equivalent to add-one smoothing with backoff). To allow a more detailed test of whether you counted singletons correctly in your one-count smoother, we’ve also provided a version of the ice cream data that has been modified to contain singletons. Here’s what I got with vtag ic2train ic2test:

Tagging accuracy (Viterbi decoding): 90.91%  (known: 90.32%  novel: 100.00%)
Model perplexity per tagged test word: 8.072

(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. I’m sure that quite a few bogus results have been unwittingly published in the research literature because of undetected bugs. How would you check the validity of your code?)
J Implementation Hints for Expectation Maximization

J.1 Interaction Between EM and Smoothing

If you are doing one-count smoothing, try not to reestimate the singleton counts *sing* (see reading section I) during the forward-backward algorithm. (It wouldn’t make sense: forward-backward yields counts *c* that aren’t even integers!) Just continue using the singleton counts that you derived from *train* in the first place. They are a sufficiently good indication of which tags are open-class vs. closed-class.

Your smoothing method may refer to the number of word types, *V* (including oov). (For example, see the formula for *p.tw-backoff* in I.) Your definition of *V* should now include all types that were observed in *train ∪ raw*.

J.2 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*new(*t*, *w*) and *c*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.

J.3 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*(N, *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*(· · · ) in vtage, as well as the token count *n*, 12 will have to be replaced by three versions in vtagem!

**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.tw()*, use only the counts in **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**.

- 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.

---

12 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 you read, and keep 3 copies).
• Once an iteration of the forward-backward algorithm has completed, it is finally safe to set `current := new`.

J.4 Checking Your Work

As noted before, you can run `vtagen 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:
  – $\alpha$ and $\beta$ 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.
  
  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!