

600.465 — Intro to NLP

Assignment 6: Tagging with a Hidden Markov Model

Prof. J. Eisner — Fall 2009

Due date: Monday 7 December, 2 pm

Absolute late deadline: Sunday 13 December, 2 pm

Final exam: Thursday 17 December, 9am–noon

In this assignment, you will build a Hidden Markov Model and use it to tag words with their parts of speech. At the risk of making this handout too long, I have added lots of specific directions and hints so that you won't get stuck. Don't be intimidated—your program will be a lot shorter than this handout! Just read carefully.

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.

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 full forward-backward algorithm. This yields a **partially 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.

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.¹ 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.

Programming language: As usual, your choice, but pick a language in which you can program quickly and well, and that supports hash tables. My final program was about 250 lines in Perl. Running on `ugrad5`, it handled problem 2 in about 15 seconds and 22M memory, and problem 3 (four iterations) in about 17.5 minutes and 131M 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 would probably run far faster.

¹But 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.

How to hand in your work: The procedure will be similar to previous assignments. Again, specific instructions will be announced before the due date. You must test that your programs run on the **ugrad** machines with no problems before submitting them. You may prefer to develop them on the **ugrad** machines in the first place, since there are copies of the data already there.

Data: There are three datasets, available in `/usr/local/data/cs465/hw6/data` on the **ugrad** machines (or at <http://cs.jhu.edu/~jason/465/hw6/data>). They are as follows:

- **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 1).
- **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.)

You 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
- **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 **###**.

Notation: In the discussion and in Figures 2–3, I’ll use the following notation. You might want to use the same notation in your program.

- Whichever string is being discussed (whether it is from **train**, **test**, or **raw**) consists of $n + 1$ words, w_0, w_1, \dots, w_n .
- The corresponding tags are t_0, t_1, \dots, t_n . We have $w_i/t_i = \text{###/###}$ 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 $p_{\text{tt}}(t_i | t_{i-1})$.
- I’ll use “**tw**” to name tag-to-word *emission* probabilities, as in $p_{\text{tw}}(w_i | t_i)$.

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

Figure 1: 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. (However, it results in a couple of unnatural categories, C and P.)

Spreadsheets: 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.

That spreadsheet is at <http://www.cs.jhu.edu/~jason/465/hw6/lect24-hmm.xls>. It is appropriate for problem 3. There also exists a Viterbi version that you can use for problems 1 and 2: <http://www.cs.jhu.edu/~jason/465/hw6/lect24-hmm-viterbi.xls>. Excel 2000 and Excel 97 display these spreadsheets correctly. The experiments we did in class are described at <http://www.cs.jhu.edu/~jason/papers/eisner.tnlp02.pdf>.

This is a long handout. By way of summary, a suggested work plan:

0. (a) Read the overview material above.
(b) Briefly look at the data files.
(c) Play with the spreadsheets from class, and study Figures 2–3. Repeat until you understand the algorithms.
1. (a) Read problem 1 carefully.
(b) Implement the unsmoothed Viterbi tagger **vtag** for problem 1. Follow Figure 2 and the implementation suggestions.
(c) Run **vtag** on the **ic** (ice cream) dataset. Check your that your tagging accuracy and perplexity match the numbers provided. Check your tag sequence against the Viterbi spreadsheet as described.
2. (a) Read problem 2’s smoothing method carefully.
(b) Improve **vtag** to do smoothing.
(c) Run **vtag** with smoothing on the **en** (English) dataset. Answer the questions at the end of problem 2.
3. (a) Read problem 3 carefully.
(b) Implement **vtagem**, starting with a copy of **vtag**. Follow Figure 3
(c) Run **vtagem** on the **ic** dataset, and check its behavior against the forward-backward spreadsheet as described.
(d) Run **vtagem** on the **en** dataset. Answer the questions at the end of problem 3.
4. (a) Try out **vtag** or **vtagem** on the **cz** dataset if you are curious.
(b) Try problem 4 (extra credit) if you are so inclined.

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

```
vtag ictrain ictest
```

You may want to review the slides on Hidden Markov Model tagging, and perhaps a textbook exposition as well, such as chapter 6 of Jurafsky & Martin (2nd edition), which specifically discusses the ice cream example.

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

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

```
Tagging accuracy: 92.48% (known: 95.99% novel: 56.07%)
Perplexity per tagged test word: 1577.499
```

You are also free to print out whatever other information is useful to you, including the tags your program picks, its accuracy as it goes along, various probabilities, etc. A common trick, to give yourself something to stare at, is to print a period to standard error (**stderr** or **cerr**) every 100 words.

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

- The overall accuracy (e.g., 92.48%) considers all word tokens, other than the sentence boundary markers **###**.²
- The known-word accuracy (e.g., 95.99%) considers only tokens of words (other than **###**) that also appeared in **train**.
- The novel-word accuracy (e.g., 56.07%) considers only tokens of words that did *not* also appear in **train**. (These are very hard to tag, since context 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.)

²No one in NLP tries to take credit for tagging **###** correctly with **###**!

The perplexity per tagged test word (e.g., 1577.499) is defined as³

$$\exp\left(-\frac{\log p(w_1, t_1, \dots, w_n, t_n \mid w_0, t_0)}{n}\right)$$

where $t_0, t_1, t_2, \dots, t_n$ is the winning tag sequence that your tagger assigns to **test** data (with $t_0 = t_n = w_0 = w_n = \###$). The perplexity is high because it considers the model's uncertainty about predicting both the word *and* its tag. (We are not computing perplexity per word, or per tag, but rather per tagged word.)

Some suggestions that will make your life easier (read carefully!):

- Make sure you really understand the algorithm before you start coding! Write pseudocode before you write the details. Work out an example on paper if that helps. Play with the spreadsheet. Review the reading or the slides. Read this handout more than once, and ask questions. Coding should be a few straightforward hours of work if you really understand everything and can avoid careless bugs.
- Your program should go through the following steps:
 - (a) 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 2, you will modify those functions to do smoothing.)
 - (b) Read the **test** data \vec{w} into memory.
 - (c) Follow the Viterbi algorithm pseudocode in Figure 2 to find the tag sequence \vec{t} that maximizes $p(\vec{t}, \vec{w})$.
 - (d) 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.)
- 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.

³The w_i, t_i notation was discussed above and refers here to test data. Why are we computing the perplexity with \exp and \log base e instead of base 2? It doesn't matter, as the two bases cancel each other out: $e^{-(\log x)/n} = 2^{-(\log_2 x)/n}$, so this really is perplexity as we've defined it. 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 2, line 2, explicitly hard-codes your prior knowledge that $t_0 = \###$.)

```

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  $t_i \in \text{tag\_dict}(w_i)$                          (* a set of possible tags for  $w_i$  *)
5.          for  $t_{i-1} \in \text{tag\_dict}(w_{i-1})$ 
6.               $p := p_{\text{tt}}(t_i | t_{i-1}) \cdot p_{\text{tw}}(w_i | t_i)$           (* 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.                  $\text{backpointer}_{t_i}(i) = t_{i-1}$           (* 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.       $t_{i-1} := \text{backpointer}_{t_i}(i)$ 

```

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

Figure 2: Sketch of the Viterbi tagging algorithm. $\mu_t(i)$ is the probability of the best path from the start state (### at time 0) to state t at time i . In other words, it maximizes $p(t_1, w_1, t_2, w_2, \dots, t_i, w_i | t_0, w_0)$ over all possible choices of t_1, \dots, t_i such that $t_i = t$.

- Figure 2 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 $\text{tag_dict}(w) = \{\text{C}, \text{H}\}$ for any other word w . In the next problem, you’ll generalize this 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 $c(\dots)$. 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 $\mu_{\text{D}}(5)$ in Figure 2.) 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.)
- *Warning:* You should **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.
- Small probabilities should be stored in memory as log-probabilities. This is actually crucial to prevent underflow.⁴
 - This assignment will talk 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 bit:** 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.⁵
 - *Suggestion:* I recommend that you use natural logarithms (\log_e) because they are simpler than \log_2 , slightly faster, and less prone to programming mistakes. (Although it is conventional to *report* log-probability using \log_2 , you can use whatever representation you like internally, and convert it later with the formula $\log_2 x = \log_e x / \log_e 2$. Anyway, you are not required to report any log-probabilities for this assignment. See footnote 3 on calculating perplexity from a natural logarithm.)

Check your work as follows. `vtag ictrain ictest` should yield a tagging accuracy of

⁴At 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.

⁵The IEEE floating-point standard does have a way of representing $-\infty$, so you could genuinely set $lp = -\text{Inf}$, which will work correctly with $+$, $>$, 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).

87.88% or 90.91%,⁶ with no novel words and a perplexity per tagged word of 3.620.⁷ You can use the Viterbi version of the spreadsheet to check your μ probabilities and your tagging:⁸

- `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 0:⁹
 - μ probabilities for each day

⁶Why are there two possibilities? Because the code in Figure 2 breaks ties arbitrarily. In this example, there are two tagging paths that disagree on day 27 but have *exactly* the same probability. So `backpointerH(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 HW2. 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!

⁷A 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 3.620 is an improvement.

⁸The Viterbi version of the spreadsheet is almost identical to the forward-backward version. However, it substitutes “max” for “+”, so instead of computing the forward probability α , it computes the Viterbi approximation μ .

⁹To 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.

- weather tag for each day (shown on the graph)¹⁰
- perplexity per tagged word: see upper right corner of spreadsheet

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

2. Now, you will improve your tagger so that you can run it on real data:

```
vtag entrain entest
```

This means using a proper tag dictionary (for speed) and smoothed probabilities (for accuracy).¹¹ Your tagger should beat the following “baseline” result:

```
Tagging accuracy: 92.48% (known: 95.99% novel: 56.07%)
Perplexity per tagged test word: 1577.499
```

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

You are required to use a “tag dictionary”—otherwise your tagger will be 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, even if they have positive smoothed probability. See the pseudocode in Figure 2.

Derive your tag dictionary from the training data. For a known word, allow only the tags that it appeared with in the training set. For an unknown word, allow all tags except ###. (*Hint*: During training, before you add an observed tag t to $\text{tag_dict}(w)$ (and before incrementing $c(t, w)$), check whether $c(t, w) > 0$ already. This lets you avoid adding duplicates.)

How about smoothing? Just to get the program working on the **en** dataset, you could use some very simple form of smoothing at first. For example, add-one smoothing on p_{tw} will take care of the novel words in **entest**, and you could get away with no smoothing at all on p_{tt} .

¹⁰You won't be able to check your backpointers directly. Backpointers would be clumsy to implement in Excel, so to find the best path, the Viterbi spreadsheet instead uses μ and ν probabilities, which are the Viterbi approximations to the forward and backward probabilities α and β . This trick makes it resemble the original spreadsheet more. But backpointers are conceptually simpler, and in a conventional programming language they are both faster and easier for you to implement.

¹¹On 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**.

However, the version of the program that you submit should use the following type of smoothing. It is basically just add- λ smoothing with backoff, but λ 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.¹²

First let us define our backoff estimates:

- Let

$$p_{\text{tt-backoff}}(t_i | t_{i-1}) = p_{\text{t-unsmoothed}}(t_i) = \frac{c(t_i)}{n}$$

Do you see why it’s okay to back off to this totally unsmoothed, maximum likelihood estimate?¹³ I’ll explain below why the denominator is n rather than $n + 1$, even though there are $n + 1$ tokens t_0, t_1, \dots, t_n .

- Let

$$p_{\text{tw-backoff}}(w_i | t_i) = p_{\text{w-addone}}(w_i) = \frac{c(w_i) + 1}{n + V}$$

This backoff estimate uses add-one-smoothing. n and V denote the number of word *tokens* and *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, \dots, t_n .)

Notice that according to this formula, any novel word has count 0 and backoff probability $p_{\text{w-addone}} = \frac{1}{n+V}$. 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:

¹²Many 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 HW2. The “one-count” method is due to Chen and Goodman, who actually give it in a more general form where λ is a linear function of the number of singletons. This allows some smoothing to occur ($\lambda > 0$) even if there are no singletons ($\text{sing} = 0$). Chen and Goodman recommend using held-out data to choose the slope and intercept of the linear function.

¹³It’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 \vec{t} of the test corpus \vec{w} . So we will have to smooth $p_{\text{tw-backoff}}(w_i | t_i)$ below; it is only $p_{\text{tt-backoff}}(t_i | t_{i-1})$ that can safely rule out novel events.

- Define a function *sing* that counts singletons. Let

$$\begin{aligned} \text{sing}_{\text{tt}}(\cdot | t_{i-1}) &= \text{number of tag types } t \text{ such that } c(t_{i-1}, t) = 1 \\ \text{sing}_{\text{tw}}(\cdot | t_i) &= \text{number of word types } w \text{ such that } c(t_i, w) = 1 \end{aligned}$$

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 $\text{sing}_{\text{tw}}(\cdot | \text{N})$ 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, $\text{sing}_{\text{tw}}(\cdot | \text{D})$ 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

$$\begin{aligned} p_{\text{tt}}(t_i | t_{i-1}) &= \frac{c(t_{i-1}, t_i) + \lambda \cdot p_{\text{tt-backoff}}(t_i | t_{i-1})}{c(t_{i-1}) + \lambda} \text{ where } \lambda = \text{sing}_{\text{tt}}(\cdot | t_{i-1}) \\ p_{\text{tw}}(w_i | t_i) &= \frac{c(t_i, w_i) + \lambda \cdot p_{\text{tw-backoff}}(w_i | t_i)}{c(t_i) + \lambda} \text{ where } \lambda = \text{sing}_{\text{tw}}(\cdot | t_i) \end{aligned}$$

Note that λ will be higher for $p_{\text{tw}}(\cdot | \text{N})$ than for $p_{\text{tw}}(\cdot | \text{D})$. Hence $p_{\text{tw}}(\cdot | \text{N})$ 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)!

- If $\lambda = 0$ because there are no singletons, some probabilities can still work out to 0/0. A trick to avoid this is to add a very small number, like $1\text{e-}100$, to λ before using it. Then in the 0/0 case you will get the backoff probability.¹⁴
- The **ic** dataset happens to have no singletons at all, so you will always back off to the unsmoothed estimate on this dataset. Therefore, your results on the **ic** dataset should not change from problem 1.

¹⁴If you want to be anal-retentive, the p_{tw} routine should specially force $\lambda = 0$ for the tag **###** in order to prevent any smoothing of $p_{\text{tw}}(\cdot | \text{###})$: it is a fact of nature that $p_{\text{tw}}(\text{###} | \text{###}) = 1$ exactly.

We've also provided a version of the ice cream data that has been slightly modified to contain singletons, so that you can use it to check your one-count smoother. Here's what I got with `vtag ic2train ic2test`:

```
Tagging Accuracy: 90.91% (known: 90.32% seen: 0.00% novel: 100.00%)
Perplexity per tagged test word: 8.152
```

(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?)

- If you want to be careful and obtain precisely the sample results provided in this assignment, you should ignore the training file's very first or very last `###/###` when you accumulate counts during training. So there are only n word/tag tokens, not $n + 1$. 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.¹⁵

When reestimating counts using **raw** data in problem 3, you should similarly ignore the initial or final `###/###` in the **raw** data.

Turn in the source code for your smoothing version of `vtag`. In `README` give your observations and results, including the output from running `vtag entrain entest` (or at least the required lines from that output). How much did your tagger improve on the accuracy and perplexity of the baseline tagger (see page 10)?

3. Write an improved version of `vtag`, called `vtagem`, that extends `vtag` to reestimate the HMM parameters on **raw** (untagged) data. You should be able to run it as

```
vtagem entrain25k entest enraw
```

¹⁵The 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 $\sum_t c(t)$, $\sum_w c(w)$, 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, consider the unsmoothed estimate $p_{tt}(t_i | t_{i-1}) = c(t_{i-1}, t_i) / c(t_{i-1})$: i can only range from 1 to n in the numerator ($i \neq 0$), so for consistency it should only be allowed to range from 1 to n in the denominator $c(t_{i-1})$ and in the numerator of the backoff estimate $c(t_i) / n$. Hence the stored count $c(t)$ should only count n of the $n + 1$ tags, omitting one of the boundary tags `###` (it doesn't matter which boundary since they are identical). Now notice that $c(t)$ is also used in the denominator of $p_{tw}(w_i | t_i)$; since we have just decided that this count will only consider i from 1 to n in the denominator, we have to count the numerator $c(t_i, w_i)$ and the numerator of the backoff estimate $c(w_i) / n$ in the same way, for consistency.

```

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.        $p := p_{tt}(t_i | t_{i-1}) \cdot p_{tw}(w_i | t_i)$  (* 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.    $S := \alpha_{###}(n)$  (* 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.      $c_{\text{new}}(t_i, w_i) := c_{\text{new}}(t_i, w_i) + (\alpha_{t_i}(i) \cdot \beta_{t_i}(i) / S)$  (* can now compute prob of  $t_i$  at time  $i$  *)
14.     for  $t_{i-1} \in \text{tag\_dict}(w_{i-1})$ 
15.        $p := p_{tt}(t_i | t_{i-1}) \cdot p_{tw}(w_i | t_i)$  (* 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.        $c_{\text{new}}(t_{i-1}, t_i) := c_{\text{new}}(t_{i-1}, t_i) + (\alpha_{t_{i-1}}(i-1) \cdot p \cdot \beta_{t_i}(i) / S)$  (* prob of arc at time  $i$  *)

```

Only some of the necessary c_{new} updates are shown above! If you have other count tables, make sure to update those too; your distributions must sum to 1. Also remember to initialize variables appropriately.

Figure 3: Sketch of the forward-backward (EM) algorithm (one iteration). $\alpha_t(i)$ 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_{new} accumulates expected counts given the entire observed sequence w_0, \dots, w_n .

`etrain25k` is a shorter version of `etrain`. 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 program will alternately tag the `test` data (using Viterbi) and modify the training counts (using EM). So you will be able to see how successive steps of EM help or hurt the performance on `test` data. See pseudocode in Figure 3.

The program should run at least 3 iterations of EM. Its output format should be as shown in Figure 4.

```

[read train]
[read test]
[read raw]
[Viterbi tagging on test]
Tagging accuracy: ... (known: ...% seen: ...% novel: ...%)
Perplexity per tagged test word: ...
[compute new counts via forward-backward algorithm on raw]
Iteration 0: Perplexity per untagged raw word: ...
[switch to using the new counts]
[new Viterbi tagging on test]
Tagging accuracy: ... (known: ...% seen: ...% novel: ...%)
Perplexity per tagged test word: ...
[compute new counts via forward-backward algorithm on raw]
Iteration 1: Perplexity per untagged raw word: ...
[switch to using the new counts]
[new Viterbi tagging on test]
Tagging accuracy: ... (known: ...% seen: ...% novel: ...%)
Perplexity per tagged test word: ...
[compute new counts via forward-backward algorithm on raw]
Iteration 2: Perplexity per untagged raw word: ...
[switch to using the new counts]
[new Viterbi tagging on test]
Tagging accuracy: ... (known: ...% seen: ...% novel: ...%)
Perplexity per tagged test word: ...
[compute new counts via forward-backward algorithm on raw]
Iteration 3: Perplexity per untagged raw word: ...
[switch to using the new counts]

```

Figure 4: Output format for `vtagem`. Your program should include the lines shown in **this font** and any other output 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.

Note that **vtagem**'s output must distinguish three kinds of accuracy rather than two:

known: accuracy on **test** tokens that also appeared in **train** (so we know their possible parts of speech)

seen: accuracy on **test** tokens that did not appear in **train**, but did appear in **raw** (so we've tried to infer their parts of speech from context)

novel: accuracy on **test** tokens that appeared in neither **train** nor **raw**

vtagem's output must also include the perplexity per *untagged* raw word. This is defined on **raw** data \vec{w} as

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

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 $p(w_1, \dots, w_n | w_0)$ while running the forward-backward algorithm. It is the total probability of *all* paths (tag sequences compatible with the dictionary) that generate the raw word sequence.

Some things you *must do*:

- Do not try to reestimate the singleton counts *sing* 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.
- Remember that $p_{\text{tw-backoff}}$ is defined in terms of the number of word types, V (including oov). Your definition of V should now include all types that were observed in **train** \cup **raw**. As in homework 2 (see discussion before questions 5–6), 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).
- 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—that is why you are using logs in the first place!

Instead you need to write $lp := \text{logadd}(lp, lq)$, where

$$\text{logadd}(x, y) \stackrel{\text{def}}{=} \begin{cases} x + \log(1 + \exp(y - x)) & \text{if } y \leq x \\ y + \log(1 + \exp(x - y)) & \text{otherwise} \end{cases}$$

You can check for yourself that this equals $\log(\exp x + \exp y)$. If you are using C/C++, note that $\log(1 + z)$ can be computed more quickly and accurately by the specialized function $\log1p(z)$.

Make sure to handle the special case where $p = 0$ or $q = 0$ (see page 8).¹⁶

- 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 $c(N, \text{accounts})$ 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 $c(\dots)$ in **vtag**, as well as the token count n ,¹⁷ 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.
- Once an iteration of the forward-backward algorithm has completed, it is finally safe to set **current** := **new**.

¹⁶If 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.

¹⁷Will 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** \cup **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).

As noted before, you can run `vtagem 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.

With this change, 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.¹⁸ 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!

Turn in 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 3 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 `rawem` file?
- (f) Suggest at least two reasons to explain why EM didn't always help.

¹⁸The spreadsheet does not do any smoothing when it computes these probabilities, so in principle you need to match this by turning off smoothing in your code. But in practice, `ictrain` happens to contain no singletons, as already noted on page 12, so that your code just happens not to smooth it.

- (g) What is the maximum amount of ice cream you have ever eaten in one day? Why? Did you get sick?
4. *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).¹⁹ 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 μ , α , and β . (Feel free to talk to me about your ideas.) Submit your improved source code, and 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 (on a different machine):

	English Viterbi	Czech Viterbi	English EM
no pruning	15 sec.	14 min.	17.5 min
light pruning	10 sec.	8.5 min.	6 min.
strong pruning	8 sec.	6.5 min.	5.5 min.
aggressive pruning	5 sec.	2 min.	4 min.

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

¹⁹Although 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 1.