600.465 — Intro to NLP
Assignment 6: Tagging with a Hidden Markov Model

Prof. J. Eisner — Fall 2016
Due date: Friday 15 April, 2 pm

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.

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.

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 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*.

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.

[1] 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.

Data: 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). 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.

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

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.)

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, $w_0, w_1, \ldots w_n$.
- 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 $p_{\text{tt}}(t_i \mid t_{i-1})$.
- I'll use "tw" to name tag-to-word *emission* probabilities, as in $p_{\text{tw}}(w_i \mid t_i)$.

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/hw-hmm/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/hw-hmm/lect24-hmm-viterbi.xls`. Excel displays these spreadsheets correctly, and LibreOffice or OpenOffice does a decent job as well. All 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:

1. (a) Do problem 1, which asks you to play with the spreadsheet from class until you understand the algorithms. You may want to look at Figure 3.
   (b) Now read the overview material above.
   (c) Briefly look at the data files.
2. (a) Read problem 2 carefully. Play with the Viterbi version of the spreadsheet. Make sure you understand the Viterbi algorithm in Figure 2.
   (b) Implement the unsmoothed Viterbi tagger `vtag` for problem 2. 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.
3. (a) Read problem 3 carefully.
   (b) Improve `vtag` to use a tag dictionary and add-1 smoothing.
   (c) Check your perplexity on `entest` against the answer given.
4. (a) For problem 4 (extra credit), carefully read the smoothing method in the Appendix.
   (b) Implement this method, and test it as explained in problem 4.
   (c) Try your implementation on `entest` and answer the questions at the end of problem 4.

5. For problem 5, study the forward-backward algorithm in Figure 3. Implement it in your `vtag`. Then use it to try out posterior decoding, and hand in your results and code.

6. (a) Implement `vtagem` as per problem 6, starting with a copy of `vtag`.
   (b) Run `vtagem` on the **ic** dataset, and check its behavior against the forward-backward spreadsheet as described.
   (c) Run `vtagem` on the **en** dataset. Answer the questions at the end of problem 6.

7. (a) Try out `vtag` or `vtagem` on the **cz** dataset if you are curious.
   (b) Try problem 5 (extra credit) if you are so inclined.

---

1. Play around with the spreadsheet at http://www.cs.jhu.edu/~jason/465/hw-hmm/lect24-hmm.xls. Try changing the red numbers: the initial transition/emission probabilities or the ice cream data. Explore what happens to the various graphs—reconstructed weather on each day and each pair of days, as well as perplexity. Look at what happens to the parameters from iteration to iteration. Study the intermediate computations to see *how* the graphs and parameters are computed. If you click or double-click on any cell, you'll see its formula. You may want to look at Figure 3.

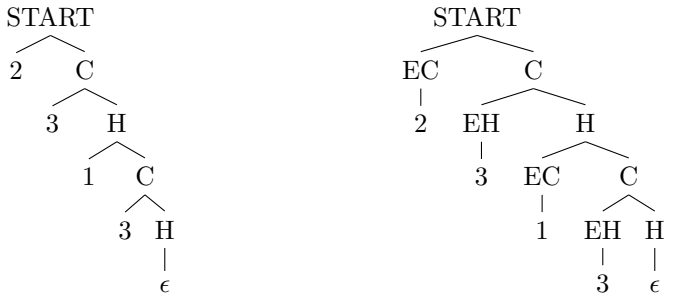
   Play as much as you like, but here are some questions to answer in your `README`:

   (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 $p(\texttt{H})$ on days 1 and 2? (*Hint:* Look at where the other 1-ice-cream days fall during the summer.)

   (b) We talked about sterotypes 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 $p(\texttt{1} \mid \texttt{H}) = 0$, $p(\texttt{2} \mid \texttt{H}) = 0.3$, $p(\texttt{3} \mid \texttt{H}) = 0.7$.
      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 $p(\texttt{1} \mid \texttt{H})$ 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 α 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 an β probability. *Which* state's β 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:



In the tree on the left, what is the meaning of an `H` constituent? What is the probability of the rule `H` → `1 C`? How about the probability of `H` → 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. A Viterbi tagger finds the single best path through an HMM—the single most likely weather sequence given the ice cream, or the single most tag.

   This 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.

   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 (Viterbi decoding): 92.48%  (known: 95.99%  novel: 56.07%)
   Perplexity per Viterbi-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 1000 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]

[2]No one in NLP tries to take credit for tagging `###` correctly with `###`!

- The known-word accuracy (e.g., 95.99%) considers only tokens of words (other than **###**) that also appeared in **train**.
- The novel-word accuracy (e.g., 56.07%) considers only tokens of words that did *not* 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.)

The perplexity per Viterbi-tagged test word (e.g., 1577.499) is defined as[3]

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

where $t_0, t_1, t_2, \ldots t_n$ 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 at `http://www.cs.jhu.edu/~jason/465/hw-hmm/lect24-hmm-viterbi.xls`. 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 3, 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.

[3] The 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 $t_0 =$**###**.)

```
(* find best μ values from left to right by dynamic programming; they are initially 0 *)
μ_###(0) := 1
for i := 1 to n                                          (* ranges over test data *)
    for t_i ∈ tag_dict(w_i)                              (* a set of possible tags for w_i *)
        for t_{i-1} ∈ tag_dict(w_{i-1})
            p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)   (* arc probability *)
            μ := μ_{t_{i-1}}(i − 1) · p                  (* prob of best sequence that ends in t_{i-1}, t_i *)
            if μ > μ_{t_i}(i)                            (* but is it the best sequence (so far) that ends in t_i at time i? *)
                μ_{t_i}(i) = μ                           (* if it's the best, remember it *)
                backpointer_{t_i}(i) = t_{i-1}           (* and remember t_i's predecessor in that sequence *)
(* follow backpointers to find the best tag sequence that ends at the final state (### at time n) *)
t_n := ###
for i := n downto 1
    t_{i-1} :=backpointer_{t_i}(i)
```

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

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

- 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: tag_dict(`###`) = {`###`}, and tag_dict(w) = {`C`, `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 $c(\cdots)$. So write down, for example, that you will store the count $c(t_{i-1}, t_i)$ in a table `count_tt` whose elements have names like `count_tt("D","N")`. When you read the training data you will increment these elements.
- You will need some multidimensional tables, indexed by strings and/or integers, to store the training counts and the path probabilities. (E.g., `count_tt("D","N")` above, and $\mu_{\texttt{D}}(5)$ 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.

- Probabilities that might be small (such as α and β) should be stored in memory as log-probabilities. Doing 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 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.[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 87.88% or 90.91%,[6] with no novel words and a perplexity per Viterbi-taggged 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).

[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* μ values (or α or β values) at each time step—i.e., up to a multiplicative constant. So to avoid underflow, you can rescale them by an arbitrary constant at every time step, or every several time steps when they get too small.

[5] The IEEE floating-point standard does have a way of representing $-\infty$, so you could genuinely set $lp =$ `-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., $lp =$ `undef` or $lp =$ `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).

[6] Why 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 backpointer$_{\texttt{H}}(28)$ will be set to `H` or `C` according to how the tie is broken, which depends on whether $t_{27} =$ `H` or $t_{27} =$ `C` is considered first in the loop at line 5. (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!

[7] 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 3.620 is an improvement.

[8] The 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.

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

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

3. 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 (Viterbi decoding): 92.48%  (known: 95.99%  novel: 56.07%)
Perplexity per Viterbi-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 tag_dict(w) (and before incrementing $c(t, w)$), check whether $c(t, w) > 0$ already. This lets you avoid adding duplicates.)

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

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

[10] You 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.

[11] On the **ic** dataset, you were able to get away without smoothing because you didn't have sparse data. You had actually observed all possible "words" and "tags" in **ictrain**.

To get the program working on this dataset, use some very simple form of smoothing for now. For example, add-one smoothing without backoff (on both p_{tw} and p_{tt}).[12] However, you'll find that this smoothing method gives lower accuracy than the baseline tagger![13]

```
Tagging accuracy (Viterbi decoding): 91.84%  (known: 96.60%  novel: 42.55%)
Perplexity per Viterbi-tagged test word: 1463.932
```

Note: If you want 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.[14] When reestimating counts using **raw** data in problem 4, you should similarly ignore the initial or final `###/###` in the **raw** data.

Don't hand in your code yet.

4. For *extra credit*, now improve the smoothing. Use the backoff smoothing method that is described in the appendix, a relatively simple method called "one-count smoothing." This should improve your performance considerably.[15]

[12] But don't smooth $p_{\text{tw}}(w_i = \texttt{###} \mid t_i = \texttt{###})$. This probability should *always* be 1, because you *know*—without any data—that the `###` tag always emits the `###` word. 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."

[13] You'll notice that it actually does better than baseline on perplexity and is a little more accurate on known words. However, the baseline does far better 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.

[14] 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 $\sum_t c(t)$, $\sum_w c(w)$, 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, suppose you have the very short corpus

`#/# H/3 C/2 #/# C/1 #/#` *(I'm abbreviating `###` as `#` in this footnote)*

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

- **In an expression** $p_{tt}(\cdot \mid \texttt{\#})$, where we are transitioning *from* `#`. You want unsmoothed $p_{tt}(H \mid \texttt{\#}) = \frac{c(\texttt{\#}H)}{c(\texttt{\#})} = \frac{1}{2}$ (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** $p_{tt}(\texttt{\#} \mid \cdot)$, where we are transitioning *to* `#`. Now, unsmoothed $p_{tt}(\texttt{\#} \mid C) = \frac{c(C\texttt{\#})}{c(C)} = \frac{2}{2}$ doesn't use $c(\texttt{\#})$. But it backs off to $p_{tt}(\texttt{\#})$, and you want unsmoothed $p_{tt}(\texttt{\#}) = \frac{c(\texttt{\#})}{n} = \frac{2}{5}$ (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** $p_{tw}(\texttt{\#} \mid \texttt{\#})$, where we are emitting a word from `#`. Footnote 12 already says 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 $\frac{c(\texttt{\#\#})}{c(\texttt{\#})}$ 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 BOS state that transitions to the H state.

[15] You'll still be rather short of the state-of-the-art 97%, even though the reduced tagset in Figure 1 ought to make

```
1.  (* build α values from left to right by dynamic programming; they are initially 0 *)
2.  α_###(0) := 1
3.  for i := 1 to n                                              (* ranges over raw data *)
4.      for t_i ∈tag_dict(w_i)
5.          for t_{i-1} ∈tag_dict(w_{i-1})
6.              p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)       (* arc probability *)
7.              α_{t_i}(i) := α_{t_i}(i) + α_{t_{i-1}}(i − 1) · p   (* add prob of all paths ending in t_{i-1}, t_i *)
8.  S := α_###(n)                                   (* total prob of all complete paths (from ###,0 to ###,n) *)
9.  (* build β values from right to left by dynamic programming; they are initially 0 *)
10. β_###(n) := 1
11. for i := n downto 1
12.     for t_i ∈tag_dict(w_i)
13.         (* now we can compute p(T_i = t_i | w⃗): it is α_{t_i}(i) · β_{t_i}(i)/S *)
14.         for t_{i-1} ∈tag_dict(w_{i-1})
15.             p := p_tt(t_i | t_{i-1}) · p_tw(w_i | t_i)       (* arc probability *)
16.             β_{t_{i-1}}(i − 1) := β_{t_{i-1}}(i − 1) + p · β_{t_i}(i)   (* add prob of all paths starting with t_{i-1}, t_i *)
17.             (* now we can compute p(T_{i-1} = t_{i-1}, T_i = t_i | w⃗): it is α_{t_{i-1}}(i − 1) · p · β_{t_i}(i)/S *)
```

Figure 3: Sketch of the forward-backward algorithm. $\alpha_t(i)$ 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). Note that the posterior probabilities of the possible tag unigrams and bigrams can be computed at lines 13 and 17. You will probably want to insert some code at those points to compute and *use* those posterior probabilities.

How much did your tagger improve on the accuracy and perplexity of the baseline tagger (see page 9)? Answer in your `README`.

Don't hand in your code yet.

5. Now it's time to implement the forward-backward algorithm. Our first use of it is for "posterior decoding" on test data. Recall that Viterbi decoding prints the single most likely overall sequence. By contrast, a posterior decoder will choose the single most likely tag at each position, even if this gives an unlikely overall sequence.

   Extend your `vtag` so that it tries posterior decoding once it's done with Viterbi decoding. At the end of `vtag`, you should run forward-backward over the *test* data: see pseudocode in Figure 3. The decoder will have to use the forward-backward results to find the best tag at each position (*hint*: insert some code at 13).

   If you *turn off smoothing* (just set $\lambda = 0$) and run `vtag ictrain ictest`, you can check the posterior probabilities (lines 13 and 17) against the ice cream spreadsheet. They are shown on the spreadsheet on the first-order and second-order graphs for iteration 0.

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

   ```
   Tagging accuracy (Viterbi decoding): ...%  (known: ...%  novel: 0.00%)
   Perplexity per Viterbi-tagged test word: 3.620
   Tagging accuracy (posterior decoding): 87.88%   (known: 87.88%  novel: 0.00%)
   ```

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

---

the problem easier for us. Why? Because you only have 100,000 words of training data; you are only using tag bigrams, not trigrams; and you are not using any morphology (e.g., word endings) to help with novel words.

Implementation note: 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 $lp :=$ `logadd`(lp, lq), where

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

You can check for yourself that this equals $\log(\exp x + \exp y)$; that the exp can't overflow (because its argument is always negative); 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 available in most languages (or see http://www.johndcook.com/cpp_log_one_plus_x.html). 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 page 8).[16]

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`?

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).

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.)

[16] 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.

At lines 13 and 17 of the forward-backward algorithm (Figure 3), you will probably want to accumulate some posterior counts of the form $c_{\text{new}}(t, w)$ and $c_{\text{new}}(t, t)$. Make sure to update all necessary count tables. Also remember to initialize variables appropriately. The updated counts can be used to get new smoothed probabilities for the next iteration of EM.

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 (Viterbi decoding):  ...% (known:  ...% seen:  ...% novel:  ...%)
Perplexity per Viterbi-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 (Viterbi decoding):  ...% (known:  ...% seen:  ...% novel:  ...%)
Perplexity per Viterbi-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 (Viterbi decoding):  ...% (known:  ...% seen:  ...% novel:  ...%)
Perplexity per Viterbi-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 (Viterbi decoding):  ...% (known:  ...% seen:  ...% novel:  ...%)
Perplexity per Viterbi-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 distinguishes 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

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

Note that this does not mention the tags for raw data, which we don't even know. It is easy to compute, since you found $p(w_1, \ldots w_n \mid w_0)$ 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* (see Appendix) 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 3 (see discussion near the start of the assignment), 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).
- Suppose `accounts/N` appeared 2 times in **train** and the forward-backward algorithm thinks it also appeared 7.8 times in **raw**. Then you should update $c(\texttt{N}, \texttt{accounts})$ from 2 to 9.8, since you believe you have seen it a *total* of 9.8 times. (Why ignore the 2 supervised counts that you're sure of?)

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

  To make this work, you will need to have *three versions* of the $c(t, w)$ table. Indeed, every count table $c(\cdots)$ in `vtag`, 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:

[17] 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** $\cup$ **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).

- The functions that compute smoothed probabilities on demand, like $p_{\text{tw}}()$, 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**.

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.

  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!

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 $\alpha_{\texttt{###}}(0)$ and $\beta_{\texttt{###}}(n)$ to 1?

(b) Why is the perplexity per Viterbi-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?

Background: 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. His paper at http://aclweb.org/anthology/J94-2001 has been cited over 500 times (http://snipurl.com/29fkead), 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).[18] 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:

	English Viterbi (only)	**Czech Viterbi (only)**	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.

Appendix: One-count smoothing

For extra credit, use the following nice 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.[19]

[18] 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 1.

[19] 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 λ 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.

First let us define our backoff estimates:

- Let
$$p_{\text{tt-backoff}}(t_i \mid t_{i-1}) = p_{\text{t-unsmoothed}}(t_i) = \frac{c(t_i)}{n}$$
Do you see why it's okay to back off to this totally unsmoothed, maximum likelihood estimate?[20] 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
$$p_{\text{tw-backoff}}(w_i \mid t_i) = p_{\text{w-addone}}(w_i) = \frac{c(w_i)+1}{n+V}$$
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 $t_0, t_1, \ldots t_n$.)

  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:

- Define a function *sing* that counts singletons. Let
$$\begin{aligned} sing_{\text{tt}}(\cdot \mid t_{i-1}) &= \text{number of tag types } t \text{ such that } c(t_{i-1}, t) = 1 \\ sing_{\text{tw}}(\cdot \mid t_i) &= \text{number of word types } w \text{ such that } c(t_i, w) = 1 \end{aligned}$$
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 $sing_{\text{tw}}(\cdot \mid \texttt{N})$ 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, $sing_{\text{tw}}(\cdot \mid \texttt{D})$ 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
$$\begin{aligned} p_{\text{tt}}(t_i \mid t_{i-1}) &= \frac{c(t_{i-1}, t_i) + \lambda \cdot p_{\text{tt-backoff}}(t_i \mid t_{i-1})}{c(t_{i-1}) + \lambda} \text{ where } \lambda = 1 + sing_{\text{tt}}(\cdot \mid t_{i-1}) \\ p_{\text{tw}}(w_i \mid t_i) &= \frac{c(t_i, w_i) + \lambda \cdot p_{\text{tw-backoff}}(w_i \mid t_i)}{c(t_i) + \lambda} \text{ where } \lambda = 1 + sing_{\text{tw}}(\cdot \mid t_i) \end{aligned}$$

[20] 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 $\vec{t}$ of the test corpus $\vec{w}$. So we will have to smooth $p_{\text{tw-backoff}}(w_i \mid t_i)$ below; it is only $p_{\text{tt-backoff}}(t_i \mid t_{i-1})$ that can safely rule out novel events.

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

If one doesn't pay respect to the difference betwen 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 λ is a linear function of the number of singletons (as in footnote 19). 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%)
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%)
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?)