601.465/665 — Natural Language Processing
Homework 6: Structured Prediction

Prof. Jason Eisner — Fall 2025
Due date: Monday 10 November, 11 pm

In this homework and the next, you will build bigram taggers. Bigram tagging is a canonical structured prediction problem because it is simple, fast, and useful for many purposes. The ideas and methods here can also be generalized to tagging models and to harder structured prediction problems such as path prediction and parsing.

- You will first train a Hidden Markov Model (HMM) to predict the word sequences in the training corpus, along with any tags that are observed in the training corpus. You will use the Expectation-Maximization algorithm to fill in any "hidden" tags that are *not* observed during training—allowing unsupervised or semi-supervised learning.
- You will then convert it to a Conditional Random Field (CRF)—a conditional log-linear model over entire taggings—which you will train *discriminatively* to predict the tags *given* the words. This supports supervised learning only.
- In the next homework, you will make your CRF fancier by adding neural features.

Your code could be used for many kinds of tagging, but you'll evaluate it on the traditional task of tagging a word sequence with a part-of-speech sequence.

This is not the first time you'll be writing code to predict structures using dynamic programming. But last time (parsing), the grammar weights were *given* to you. This time you'll have to *train* the weights that you'll use to predict structures.

(In the next homework, you will "neuralize" your taggers by using neural networks to predict the transition and emission probabilities. At that point you'll have to use backprop to compute gradient updates.)

Homework goals: This homework ties together most of the ideas from the course: linguistic modeling, structured prediction, dynamic programming, training objectives, and (in the next homework) deep embeddings of structures.
After completing this and the next homework, you should be comfortable with modern structured prediction methods, including different training objectives (supervised and semi-supervised; generative and discriminative), different modeling techniques including neural nets, and different decoders.
As a result, you should be able to see how to apply the same methods to other kinds of structured prediction (PCFG trees and FST paths).

Collaboration: ***You may work in pairs on this homework.*** 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:

(a) You should do all the work *together*, for example by pair programming. Don't divide it up into "my part" and "your part."

(b) Your PDF submission to Gradescope should describe at the top what each of you contributed, so that we know you shared the work fairly.

(c) Your partner for homework 6 can't be the same as your partner from homework 4 (parsing).

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

Materials: We provide spreadsheets, some of which were shown in class. You should use Python for this homework—we give you Python starter code, you'll get a fast implementation using the PyTorch tensor operations, and you'll add PyTorch back-propagation to your codebase in the next homework. The materials can be found in `http://www.cs.jhu.edu/~jason/465/hw-tag/`. Data for the homework are described in reading section F.

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

1 Understanding HMMs and the Forward-Backward Algorithm

The forward-backward algorithm is given in Algorithm 4 of the reading handout, and we encourage you to play with the spreadsheet that we used in class (see `INSTRUCTIONS`).

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

(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 why it's roughly $\frac{1}{2}$, by considering the probability of paths starting with HHH versus CHH. (Days 2 and 3 are probably H.)

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

(b) We talked about stereotypes in class. Suppose you bring a very strong bias to interpreting the data: you believe that I *never* eat only 1 ice cream on a hot day. So, again reload the spreadsheet, and set $p(\mathtt{1} \mid \mathtt{H}) = 0$, $p(\mathtt{2} \mid \mathtt{H}) = 0.3$, $p(\mathtt{3} \mid \mathtt{H}) = 0.7$.

i. How does this change affect the initial reconstruction of the weather (the leftmost graph)?

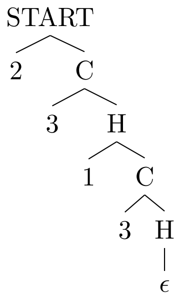
ii. What does the final graph look like after 10 iterations of reestimation?

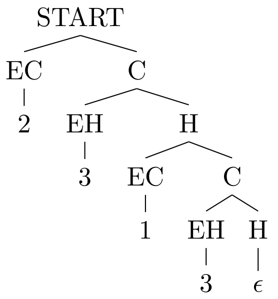
iii. What is $p(\mathtt{1} \mid \mathtt{H})$ after 10 iterations? Explain carefully why this is, discussing what happens at each reestimation step, in terms of the 2^{33} paths through the trellis.

(c) The backward algorithm (which computes all the β 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:





EC = "emit from cold"
EH = "emit from hot"

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

2 Building an HMM

Make a bigram Viterbi tagger called `tag.py` that can be trained and run on the semi-supervised English data in the `data/` subdirectory. Follow the design described in the reading handout.

Starter code and detailed `INSTRUCTIONS` are given in the `code/` subdirectory.[1] These walk you through a simple, reasonable path for building up your HMM tagger, partly using some pieces we've implemented for you, and checking your work against the ice cream spreadsheet from class.

Hand in answers to the following questions:

2 (a) Why does Algorithm 1 initialize $\alpha_{\text{BOS}}(0)$ and $\beta_{\text{EOS}}(n)$ to 1?

3 (b) If you train on a **sup** file and then evaluate on a held-out **raw** file, you'll get lower perplexity than if you evaluate on a held-out **dev** file. Why is that? Which perplexity do you think is more important and why?

4 (c) V includes the word types from **sup** and **raw** (plus OOV). Why not from **dev** as well?

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

6 (e) Explain in a few clear sentences why the semi-supervised approach might sometimes help.

[1] If you adopt the starter code for `tag.py`, then your work will be in `hmm.py`, which it imports.

How does it get additional value out of the **enraw** file?

✌$_7$ (*Extra credit:* You could investigate this empirically. For example, add some code to compare two models on a development corpus, printing out tag tokens that were incorrect under the first model but were fixed in the second model. In principle, you could even try to automatically trace what was responsible for the fix. That is, how that the good transitions/emissions are winning over the bad ones, which tokens in **enraw** contributed the fractional counts to make that happen?)

☞$_8$ (f) Suggest at least two reasons to explain why the semi-supervised approach didn't always help.

☞$_9$ (g) How does your bigram HMM tagger compare to a baseline unigram HMM tagger (see reading section G.3)? Consider both accuracy and cross-entropy. Does it matter whether you use **enraw**?

✌$_{10}$ (h) *Extra credit:* Experiment with different training strategies for using **enraw**. For example, you could train on **enraw** alone, or a combined corpus of **ensup+enraw**, or **ensup+ensup+ensup+enraw** (so that the supervised data is weighted more heavily). And you could do staged training where you do additional training on **ensup** before or after this. What seems to work? Why?

3 *Extra credit:* Open-Ended Improvements

Optionally implement an `--awesome` flag in `tag.py` that improves your cross-entropy or accuracy (on held-out data).

Your improvement should demonstrate intellectual engagement with the problem of how to improve your tagging accuracy or speed. It doesn't have to be complicated, but a one-line change is not enough by itself. More successful or interesting solutions will get more credit.

A good starting point is to study the output on the English data. As a previous question noted, our basic tagging architecture seems to get a lot of known words wrong. Think about why it might incorrectly tag a certain word as a determiner, even though that word only appeared as a noun in training data.

With this in mind, here are some things you might try:

- Posterior decoding, as discussed in `INSTRUCTIONS` and reading section D as an alternative to Viterbi tagging. (You can add this as an option for the `--loss` argument, rather than using `--awesome`.)

- Some other type of smoothing at the M-step. In choosing one, you should consider that for our particular application of HMMs—natural language part-of-speech tagging—the emission matrix B is sparse. Most words have only a few possible tags. Thus, many entries in B are zero (or at least very small). Perhaps you want to encourage a sparse transition matrix. For example, are there smoothing methods that pick up on the fact that some tags like `D` and `P` are closed-class tags that only allow a small set of words? Can you come up with one?

- Use hard constraints: a word that is known from supervised training data can only be tagged with one of its *supervised* tags.[2] You could choose to impose this restriction only when tagging the evaluation data (decoding), or during training as well.

- Use a higher-order HMM, e.g., trigram transition probabilities instead of bigrams.

- Anything else you can think of!

11 Discuss what you did and what you learned.

Your tagger might still fall short of the state-of-the-art 97% for English tagging—even though the reduced tagset in Figure 2 of the reading means that you're solving an *easier* problem than the state-of-the-art taggers. Why? Because you only have 100,000 words of training data (together with embeddings).

As a warning, too much experimentation with the dev data may lead to overfitting. Thus, you might consider splitting the dev data into two parts, holding some of it out to check that your final system actually works on a new dataset.

4 Conditional Random Field Tagger

Finally, add a simple CRF tagger to your codebase, again following the `INSTRUCTIONS` file.

You'll use only transition and emission features. As explained in reading section E.3, this is basically equivalent to a discriminatively trained HMM—although the parameterization and training algorithm happen to be different, the main difference is that you'll be maximizing the *conditional* likelihood.

12 (a) Compare the CRF to the HMM, training on **ensup** and evaluating on **endev**. Run a few experiments to compare them under different conditions (hyperparameters, unigram mode, features from etc.) and using different metrics (cross-entropy, accuracy). Look at the output if possible. What did you learn?

13 (b) What happens when you include **enraw** in the training data for your CRF? Explain.

In the next homework, you'll add more useful, contextual features that can make the CRF more powerful than the HMM, although learning to wield that power correctly may still be difficult with small training data.

5 Final Question

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

[2] In other words, set τ_j to a restricted set of tags that depends on w_j. This can be made reasonably fast by zeroing out some of the B parameters.

601.465/665 — Natural Language Processing
Reading for Homework 6: Structured Prediction

Prof. Jason Eisner — Fall 2025

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies homework 6, which refers to it.

A HMM Basics

A.1 Notation

In this reading handout, I'll use the following notation for hidden Markov models (HMMs). You might want to use the same notation in your program.

- A sentence is a word sequence $w_1, \ldots w_n$. We also define $w_0 = \text{BOSW}$ and $w_{n+1} = \text{EOSW}$. The W is for 'word'.
- The corresponding tags are $t_1, \ldots t_n$. We also define $t_0 = \text{BOS}$ and $t_{n+1} = \text{EOS}$.
- We sometimes write the i^{th} tagged word as w_i/t_i, for example in the data files.
- I'll use "A" to indicate tag-to-tag *transition* probabilities, as in $p_{\text{A}}(t_i \mid t_{i-1})$.
- I'll use "B" to indicate tag-to-word *emission* probabilities, as in $p_{\text{B}}(w_i \mid t_i)$.
- It is sometimes convenient to put these probabilities into matrices. Suppose we have integerized the tag types. Then let the **transition matrix** A be a square matrix defined by $A_{st} = p_{\text{A}}(t \mid s)$. Notice that row s sums to 1: it gives a probability distribution over the tags t that might follow tag s.
- Suppose we have also integerized the word types. Then let the **emission matrix** B be a wide rectangular matrix defined by $B_{tw} = p_{\text{B}}(w \mid t)$, so row t gives a probability distribution over the words w that might be emitted by tag t. Usually this $k \times V$ matrix is wider than it is tall, since the vocabulary (V words) is much larger than the tagset (k tags).

A.2 Probability Model

Now, given a tagged sentence

$$w_1/t_1 \ w_2/t_2 \ldots w_n/t_n,$$

the probability of generating it (according to the hidden Markov model) is

$$p(\boldsymbol{t}, \boldsymbol{w}) = \left(\prod_{i=1}^{n+1} p_{\text{A}}(t_i \mid t_{i-1}) \right) \cdot \left(\prod_{i=1}^{n} p_{\text{B}}(w_i \mid t_i) \right) \tag{1}$$

Algorithm 1 The forward algorithm. $\boldsymbol{w}$ is the input sentence, and $\boldsymbol{\tau}$ is a sequence of sets of tags. For each $j \in [1, n]$, τ_j denotes the set of possible tags for w_j. The algorithm returns $\sum_{\boldsymbol{t}:(\forall j) t_j \in \tau_j} p(\boldsymbol{t}, \boldsymbol{w})$. In the standard case, each τ_j is the set of all k tags, and then the algorithm returns $p(\boldsymbol{w})$ (equation (3)). But it can also be made to return $Z = p(\boldsymbol{t}, \boldsymbol{w})$ (equation (1)) by taking each $\tau_j = \{t_j\}$. In general, it returns the likelihood when we observe the words $\boldsymbol{w}$ and we also observe the fact that each t_j falls in τ_j; this sums over all tagged sentences that are consistent with those observations.

```
function FORWARD($\boldsymbol{w}$, $\boldsymbol{\tau}$)
    $w_0 \leftarrow$ BOSW; $\tau_0 \leftarrow \{$BOS$\}$    ▷ precede word/tag sequence with BOSW/BOS
    $n = |\boldsymbol{w}|$; $w_{n+1} \leftarrow$ EOSW; $\tau_{n+1} \leftarrow \{$EOS$\}$    ▷ terminate word/tag sequence with EOSW/EOS
    ▷ all $\alpha$ values are initially 0
    $\alpha_{\text{BOS}}(0) \leftarrow 1$    ▷ "start" node of the graph
    for $j \leftarrow 1$ to $n+1$ :
        ▷ note: the loops below could be replaced by more efficient tensor operations (see equation (12))
        for $t_j \in \tau_j$ :
            for $t_{j-1} \in \tau_{j-1}$ :    ▷ computes equation (7)
                ▷ find total prob of prefix paths ending in $t_{j-1}t_j$; add it to total prob of prefix paths ending in $t_j$
                $p \leftarrow p_{\text{A}}(t_j \mid t_{j-1}) \cdot p_{\text{B}}(w_j \mid t_j)$    ▷ probability of the $t_{j-1} \rightarrow t_j$ arc
                $\alpha_{t_j}(j) \mathrel{+}= \alpha_{t_{j-1}}(j-1) \cdot p$    ▷ extend prefix paths starting with $t_{j-1}$; keep a running total
    $Z \leftarrow \alpha_{\text{EOS}}(n+1)$    ▷ total prob of all complete paths (from BOS,0 to EOS,$n+1$)
    return $Z$
```

We can also write this as a product of $2n+1$ matrix entries,

$$p(\boldsymbol{t}, \boldsymbol{w}) = \left(\prod_{i=1}^{n+1} A_{t_{i-1}, t_i} \right) \cdot \left(\prod_{i=1}^{n} B_{t_i, w_i} \right) \tag{2}$$

where the subscripts are the integers that represent these tags and words.

The first product is the prior probability of generating the tag sequence under a bigram model, and the second product is the probability of generating the observed words once we have chosen the tags. We don't have to generate $t_0 =$ BOS because it is given, just as ROOT was given in homework 1 and BOSW was given in homework 3. However, just as in homework 3, we do have to generate $t_{n+1} =$ EOS, in order to know when the sentence ends: the sentence length $n \geq 0$ is the first natural number such that $t_{n+1} =$ EOS.

In the above equations, we omitted the trivial factors $p_{\text{B}}(w_0 \mid t_0) = p_{\text{B}}(\text{BOSW} \mid \text{BOS}) = 1$ and $p_{\text{B}}(w_{n+1} \mid t_{n+1}) = p_{\text{B}}(\text{EOSW} \mid \text{EOS}) = 1$. However, our pseudocode will include the latter factor, for convenience.

A.3 Marginal Probabilities

The probability of generating the words $\boldsymbol{w}$—marginalizing out the tag sequence that produced those words—is

$$Z = p(\boldsymbol{w}) = \sum_{\boldsymbol{t}} p(\boldsymbol{t}, \boldsymbol{w}) = \sum_{t_1} \cdots \sum_{t_n} \left(\prod_{i=1}^{n+1} p_{\text{A}}(t_i \mid t_{i-1}) \right) \cdot \left(\prod_{i=1}^{n} p_{\text{B}}(w_i \mid t_i) \right) \tag{3}$$

where we have observed $t_0 = \text{BOS}$ and $t_{n+1} = \text{EOS}$ but have to sum over all possible taggings for the n positions in between. With k possible tags at each position, there are k^n possible taggings to sum over! Yet as we saw in lecture, this grand summation can be computed in $O(nk^2)$ time by the **forward algorithm** (Algorithm 1). Basically, the trick is that we can express (3) as

$$Z = p(\boldsymbol{w}) = \alpha_{\text{EOS}}(n+1) \tag{4}$$

if we first define $\alpha_{t_j}(j)$ for any time step $j \in [1, n+1]$ and any possible tag t_j just preceding that time step:

$$\alpha_{t_j}(j) \overset{\text{def}}{=} \sum_{t_1} \cdots \sum_{t_{j-1}} \left(\prod_{i=1}^{j} p_{\text{A}}(t_i \mid t_{i-1}) \right) \cdot \left(\prod_{i=1}^{j} p_{\text{B}}(w_i \mid t_i) \right) \tag{5}$$

This represents the total probability of all prefix tag sequences for $w_1, \ldots, w_j$ that end with t_j. For $j \geq 2$ (so there is at least one $\sum$ symbol), it can be rearranged and written as an efficient recurrence in terms of $j-1$:

$$\begin{aligned}
\alpha_{t_j}(j) &= \sum_{t_{j-1}} \left(\sum_{t_1} \cdots \sum_{t_{j-2}} \left(\prod_{i=1}^{j-1} p_{\text{A}}(t_i \mid t_{i-1}) \right) \cdot \left(\prod_{i=1}^{j-1} p_{\text{B}}(w_i \mid t_i) \right) \right) \cdot p_{\text{A}}(t_j \mid t_{j-1}) \cdot p_{\text{B}}(w_j \mid t_j) && (6) \\
&= \sum_{t_{j-1}} \alpha_{t_{j-1}}(j-1) \cdot p_{\text{A}}(t_j \mid t_{j-1}) \cdot p_{\text{B}}(w_j \mid t_j) && (7)
\end{aligned}$$

and that formula also works for the case $j = 1$ if we take as the base case

$$\alpha_{\text{BOS}}(0) = 1 \qquad\qquad \alpha_{t_0}(0) = 0 \text{ if } t_0 \neq \text{BOS} \tag{8}$$

A.4 Matrix Formulas

For efficiency, let's rewrite equation (7) to use vector and matrix operations. First, we switch to using the parameter matrices A and B. (We also rename t_{j-1} and t_j to s and t for simplicity.)

$$\alpha_t(j) = \sum_s \alpha_s(j-1) \cdot A_{st} \cdot B_{tw_j} \tag{9}$$

Since the B term doesn't depend on s, we can write this as

$$\alpha_t(j) = \left(\sum_s \alpha_s(j-1) \cdot A_{st} \right) B_{tw_j} \tag{10}$$

If we regard $\vec{\alpha}(j)$ and $\vec{\alpha}(j-1)$ as row vectors indexed by tags, this is

$$\vec{\alpha}_t(j) = \big(\vec{\alpha}(j-1) \cdot A\big)_t \cdot B_{tw_j} \tag{11}$$

or, if we want to express and compute this for all t at once,

$$\vec{\alpha}(j) = (\vec{\alpha}(j-1) \cdot A) \odot B_{\cdot w_j} \tag{12}$$

where $B_{\cdot w_j}$ denotes column w_j of B, and the $\odot$ operator performs elementwise multiplication of two vectors (that is, if $\vec{c} = \vec{a} \odot \vec{b}$, then $c_t = a_t \cdot b_t$ for each t).[1]

[1] Can't we somehow just express equation (9) as a matrix multiplication? Yes, if we first define a new matrix. For any word type w, define the **observable operator** $P(w)$ to be the square matrix whose entries are given by

Algorithm 2 The Viterbi algorithm, which finds $\operatorname{argmax}_{\boldsymbol{t}:(\forall j) t_j \in \tau_j} p(\boldsymbol{t}, \boldsymbol{w})$. Compare Algorithm 1.

```
function VITERBI($\boldsymbol{w}$, $\boldsymbol{\tau}$)
    $w_0 \leftarrow$ BOSW; $\tau_0 \leftarrow \{$BOS$\}$                ▷ precede word/tag sequence with BOSW/BOS
    $n = |\boldsymbol{w}|$; $w_{n+1} \leftarrow$ EOSW; $\tau_{n+1} \leftarrow \{$EOS$\}$   ▷ terminate word/tag sequence with EOSW/EOS
    ▷ all $\hat{\alpha}$ values are initially $0$ (or $-\infty$), and all backpointers are initially None
    $\hat{\alpha}_{\text{BOS}}(0) \leftarrow 1$                                  ▷ "start" node of the graph
    for $j \leftarrow 1$ to $n+1$ :
        for $t_j \in \tau_j$ :
            for $t_{j-1} \in \tau_{j-1}$ :
                $p \leftarrow p_{\text{A}}(t_j \mid t_{j-1}) \cdot p_{\text{B}}(w_j \mid t_j)$        ▷ probability of the $t_{j-1} \to t_j$ arc
                if $\hat{\alpha}_{t_j}(j) < \hat{\alpha}_{t_{j-1}}(j-1) \cdot p$ :          ▷ keep a running maximum
                    $\hat{\alpha}_{t_j}(j) \leftarrow \hat{\alpha}_{t_{j-1}}(j-1) \cdot p$              ▷ increase the maximum
                    $\text{backpointer}_{t_j}(j) \leftarrow t_{j-1}$      ▷ and remember who provided the new maximum
    ▷ follow backpointers to find the best tag sequence that ends at the final state (EOS at time $n+1$)
    $t_{n+1} \leftarrow$ EOS
    for $j \leftarrow n+1$ downto $1$ :
        $t_{j-1} \leftarrow \text{backpointer}_{t_j}(j)$
    return $\boldsymbol{t}$
```

A.5 The Viterbi Algorithm

Algorithm 2 famously finds the most probable tagging of $\boldsymbol{w}$. Where the forward algorithm uses the semiring $(\oplus, \otimes) = (+, \times)$ to find the *total* probability of all taggings, the Viterbi algorithm uses $(\oplus, \otimes) = (\max, \times)$ to find the *maximum* probability of all taggings. To avoid confusion, we refer to this as $\hat{\alpha}$ rather than α. It then follows backpointers to extract the argmax—that is, the tagging that achieves this maximum.

If you want to use tensor operations instead of nested loops to compute the $\hat{\alpha}$ probabilities at position j, try to express equation (12) in PyTorch where you express matrix multiplication without using the `@` operator, instead using `*` and `sum`. Then for Viterbi, replace `sum` with `max` (and figure out how to use `argmax` for the backpointers). *Hint:* The `*` operation is broadcastable.

$P(w)_{st} = A_{st} B_{tw}$. This is basically a copy of A, but where each column t has been reweighted by tag t's probability of emitting word w. In the "trellis graph" whose paths correspond to possible taggings, the $k \times k$ matrix $P(w_j)$ gives the weights of the k^2 arcs from time $j-1$ to time j. Now we can rewrite equation (9) as $\vec{\alpha}(j) = \vec{\alpha}(j-1) P(w_j)$. In other words, when we observe the word token w_j, we update the $\vec{\alpha}$ vector by multiplying it by w_j's observable operator—that is, we update it by a linear transformation.

As the forward algorithm is basically just a sequence of $n+1$ such updates, we get we get $p(\boldsymbol{w}) = \big(\vec{\alpha}(0) P(w_1) \cdot P(w_2) \cdot \cdots \cdot P(w_n) P(w_{n+1})\big)_{\text{EOS}}$, or less efficiently, $p(\boldsymbol{w}) = \big(P(w_1) \cdot P(w_2) \cdot \cdots \cdot P(w_n) P(w_{n+1}\big)_{\text{BOS},\text{EOS}}$, which represents the total weight of all paths of length $n+1$ from $(\text{BOS}, 0)$ to $(\text{EOS}, n+1)$ in the trellis graph.

However, that's probably not the best way to implement it unless you have space to precompute and store the matrices $P(w)$ for all w. The reason is that each time you construct a copy of $P(w_j)$ from A and B, you have to scale each column of A. Fewer multiplications are used by equation (12), which computes the row vector $\vec{\alpha}(j-1) \cdot A$ first and then scales each element of that row vector (faster than scaling a whole column).

B Training HMMs[2]

B.1 Maximum Likelihood Estimation

Let's model sentences as being generated independently of one another—all by the same HMM with parameters A, B. Then the probability of generating an observed sequence of sentences $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots$ is given by $\prod_m p(\boldsymbol{w}_m)$. This is known as the **likelihood** of the parameters A, B (given this dataset).

To estimate the parameters of a model,[3] a pretty good estimation procedure (with a lot of theoretical justification) is to maximize their likelihood, or equivalently their log-likelihood:

$$\mathcal{L}(A, B) \overset{\text{def}}{=} \sum_m \log \underbrace{p(\boldsymbol{w}_m)}_{\text{given by the sum (3)}} \tag{13}$$

But it's wise to add a regularization term to this objective, to resist overfitting to the training data. The scale of this regularization term is, as usual, a hyperparameter C that you can tune on dev data.

$$\underset{A,B}{\operatorname{argmax}} \, \mathcal{L}(A, B) - C \cdot R(A, B) \tag{14}$$

Above, I assumed the unsupervised setting, where you observed a sequence of untagged sentences $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots$. But what if you observed something else? In general, the likelihood of the parameters is defined as the probability of generating *all the observed data* from those parameters.[4] So when you *observe* more (or less) data, you'll have to revise the definition (13) of $\mathcal{L}(A, B)$. Examples:

- If you have more sentences in your training corpus, then equation (13) will have more summands.

- If for a particular observed sentence $\boldsymbol{w}_m$ you also observe the tags $\boldsymbol{t}_m$, then you should replace $p(\boldsymbol{w}_m)$ with $p(\boldsymbol{t}_m, \boldsymbol{w}_m)$ in equation (13). That is, use equation (1) instead of equation (3). The probability $p(\boldsymbol{t}_m, \boldsymbol{w}_m)$ can be regarded as still summing over all the tag sequences that are *consistent with the observations*: but now we have enough observations that there is only one such tag sequence, so $p(\boldsymbol{t}_m, \boldsymbol{w}_m)$ is the only summand.

- As an advanced case, if you observe $\boldsymbol{w}_m$ and *part* of $\boldsymbol{t}_m$, then you should modify equation (3) to marginalize over only the unobserved tags. For example, if you observe that $t_3 = \textsc{noun}$, then just take $t_3 = \textsc{noun}$ in the formula and remove the summation $\sum_{t_3}$ over all possible values of t_3. Our Algorithm 1 actually handles this case.

B.2 Discussion: Supervised, Unsupervised, and Semi-Supervised Learning

For the **tagging task**, we are interested in learning the mapping $\boldsymbol{w} \mapsto \boldsymbol{t}$ from a sentence to its tagging. (This is just an example of the common machine learning problem of learning an $x \mapsto y$ mapping.) But what kind of observed data do we learn from?

[2] The training methods discussed in this section are much more general than just HMMs. Very similar methods apply to PCFGs, for example. They also apply to lots of other models. One nice thing about HMMs (and PCFGs) is that the probability Z of generating the observed data can be computed efficiently, by the forward (or inside) algorithm. But the methods can be used even for models where we only have slow or approximate methods for computing Z.

[3] An alternative is to "go full Bayesian" and get a posterior distribution over the parameters, rather than just picking one value for the parameters.

[4] Here's a quick explanation of what "likelihood" means in general.

Supervised learning. If we observe the tags for all sentences, then we have a *supervised* learning problem. In that case, you already know how to estimate the probabilities $p_{\mathrm{A}}(t \mid t_{\mathrm{prev}})$ and $p_{\mathrm{B}}(w \mid t)$.

Specifically, if you have a small number of probabilities to estimate, then you can just use the relative frequencies of the different transitions and emissions (see equation (17) below). This turns out to be equivalent to maximizing the log-likelihood.

If you have a larger vocabulary, then you should probably do something to avoid overfitting, for example:

- smooth the counts (see reading section B.3 below)
- maximize the *regularized* log-likelihood (see equation (14) above)
- use a model of $p_{\mathrm{B}}(w \mid t)$ that has fewer parameters (see reading section H.1 on the next homework)

Unsupervised learning. The name "hidden Markov model" is properly used for the case where we don't observe the tags. Although the tags are governed by a Markov model (bigram model), they are *hidden* (or *unobserved* or *missing*). The log-likelihood in equation (13) is sometimes called the *incomplete-data log-likelihood* for this reason.

There are different reasons we might want to estimate an HMM. If you're using the HMM for language modeling, then the tags $\boldsymbol{t}$ are *latent* random variables—unseen forces that we posit to help explain the observed data $\boldsymbol{w}$. In that case, we don't actually care about which tags are used, only that they are helpful in modeling $p(\boldsymbol{w})$.

But if we are actually going to evaluate on the tagging task, then this is an *unsupervised* learning problem. And it's hard! We want to predict the correct tags despite never having seen any tags in training data.

Semi-supervised learning. Seeing even *some* of the tags makes the unsupervised problem easier. It is then called semi-supervised. For one thing, we might observe in the tagged sentences that `the` usually gets tagged as `Det` and `caviar` usually gets tagged as `Noun`, rather than vice-versa. (The purely unsupervised learning problem has no information to break this tie—swapping the names `Det` and `Noun` through the model would not change its likelihood function.)

How does the unsupervised data help? Even if a semi-supervised learner never observes `caviar/Noun` in the tagged sentences, it might be able to guess this tag in an untagged sentence that contains `the caviar with`. It knows from the tagged sentences that `the/Det` and `with/Prep` are good guesses for those frequent words. And then it will want to guess `caviar/Noun` in between, because it knows from other tagged sentences that `Det Noun` and `Noun Prep` are common bigrams.

In other words, the highest-probability paths that explain the untagged sentence will tag this token of `caviar` as `Noun`. As a result, we can best increase the likelihood by adjusting the parameters to increase $p_{\mathrm{B}}(\texttt{caviar} \mid \texttt{Noun})$. That will help tag `caviar` correctly in test data.[5]

Unsupervised and semi-supervised learning are finicky. As mentioned in class, Merialdo (1994) famously published this troubling finding:[6]

[5] In the EM algorithm (reading sections B.4–B.5 below), the E step guesses the `Noun` tag, so the M step gets a fractional observation of `Noun` emitting `caviar`, which will increase $p_{\mathrm{B}}(\texttt{caviar} \mid \texttt{Noun})$. However, any other way of increasing the likelihood in reading section B.1, such as gradient ascent (reading section B.3 below), will also have this effect.

[6] He used unsmoothed EM to train a trigram HMM tagger, without regularization, using about a million words of newspaper text with varying amounts of supervision. The tagset had 76 different tags, corresponding to the

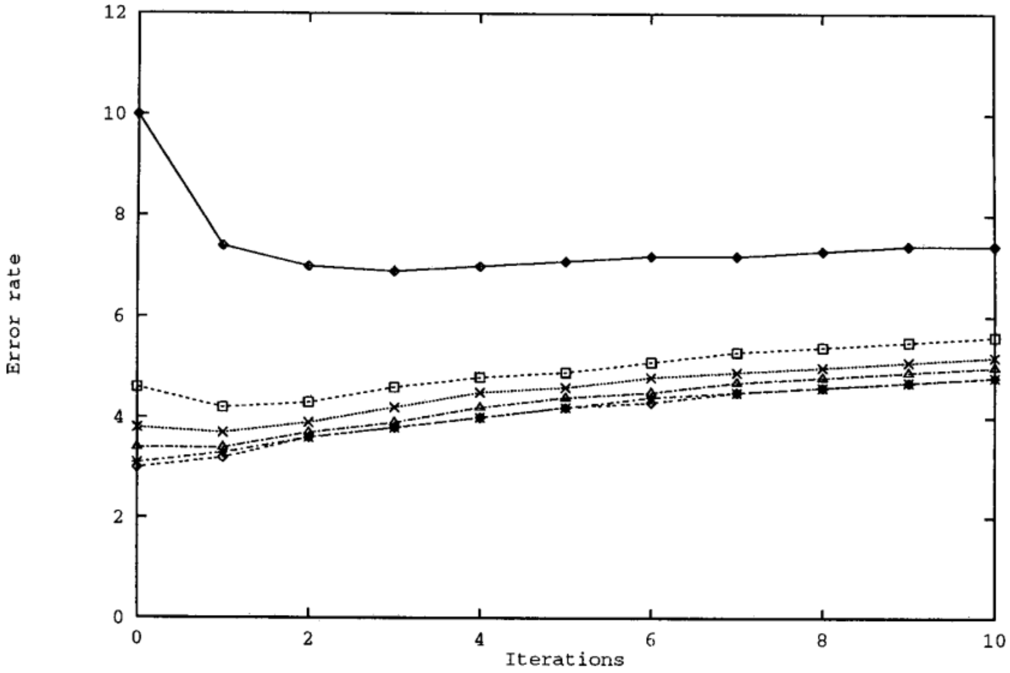


Figure 1: Reproduced from Merialdo (1994). Maximum-likelihood training on the unsupervised corpus from various initial points. Top line initializes with 100 supervised examples, bottom line with all 40,000 supervised examples.

> Experiments show that the best training is obtained by using as much tagged text as possible. They also show that [unsupervised] maximum likelihood training, the procedure that is routinely used to estimate HMM parameters from training data, will not necessarily improve the tagging accuracy. In fact, it will generally degrade this accuracy, except when only a limited amount of hand-tagged text is available.

Increasing the log-likelihood (13) does get a better model in some sense. Certainly it improves the log-probability of the *training* data—by definition. Hopefully, it will also improve the log-probability of the *dev* and *test* data (the cross-entropy); if not, you're overfitting to the training data and you should add a regularizer to the objective. But, as mentioned in class, there is no guarantee that it will increase the *accuracy* of any tagger (reading section A.5 or reading section D). As we saw in an earlier homework, cross-entropy and accuracy are different measures.

Figure 1 is copied from Merialdo's paper. He first did supervised training on a subset of the data, then fine-tuned this with fully unsupervised training on all of the data. As you can see, when he started with a bad tagger trained on only 100 supervised sentences, its accuracy did improve for the first 3 iterations of semi-supervised training (presumably as it learned new words like `caviar` from context), but then started to get worse. And if he started with any more accurate tagger, only the first 0 or 1 iterations helped! By 2 or 3 iterations, he was worse off than where he started.

The problem is that the accuracy measure demands particular "correct" tags. But the log-likelihood objective (13) doesn't actually care what tags are used on an unsupervised sentence, as long as they help to explain its words. The tagging scheme that does that best might not match

preterminals of a syntactic treebank.

the tagging scheme in the supervised data. The supervised tags were specifically chosen to be coarse-grained preterminals. Yet to predict the words of the unsupervised sentences, it might be *more* useful to keep track of whether there was a recent transitive verb, or whether the sentence is about politics! So unsupervised learning might repurpose a tag like `F` (foreign word) to mean something else. We already saw this in class with the ice cream spreadsheet, where EM training sometimes used the `H` tag to mean something other than the intended "hot day."

To stick to the intended use of the tags, it might help to take supervision into account in a more serious way than Merialdo did. His objective function completely ignored the supervised data—it was the wholly unsupervised log-likelihood objective (13). This ignores the supervised data, which would have been *wholly* irrelevant if he'd used a magical optimizer that found the global maximum of (13). The supervised data only influenced his tagger because his optimization algorithm (EM) got stuck in a local maximum near his supervised initialization point. So we could change the objective function to consider the supervised data explicitly. Some possibilities:

- If $\vec{\theta}_{\text{sup}}$ is the parameters of the initial supervised model, you could modify the unsupervised objective equation (13) by adding a regularizer $||\vec{\theta} - \vec{\theta}_{\text{sup}}||^2$ to your unsupervised objective. This is basically doing L_2 regularization on the *change* in the parameters, to discourage them from forgetting the initializer.

- You could use a semi-supervised log-likelihood objective: the sum of $\log p(\boldsymbol{t}, \boldsymbol{w})$ on the supervised sentences and $\log p(\boldsymbol{w})$ on the unsupervised sentences. Then you still get some guidance from the supervised data. **You'll try this version in the homework.**

- Unfortunately, if you have 100 supervised sentences and 39,900 unsupervised sentences, like Merialdo, then the objective function will be dominated by the untrustworthy guessed tags. The good tags will have very little influence. So you could try to fix this by placing more weight on the supervised sentences.[7] This weight is a hyperparameter.

- Rather than set a weight, you could try to maximize the unsupervised objective *subject to the constraint* that the tagging of the 100 supervised sentences stays *as good* as it was in the supervised model.[8] However, this requires constrained optimization techniques.

- Use a semi-supervised *discriminative* objective: the sum of $\log p(\boldsymbol{t} \mid \boldsymbol{w})$ on the supervised sentences (see equation (24)) and $\log p(\boldsymbol{w})$ on the unsupervised sentences.

  Again, you can place less weight on the unsupervised term. In effect, the supervised term $\sum_m \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m)$ term more or less captures your real objective, which is to do a good job of tagging. The unsupervised term $\log p(\boldsymbol{w})$ acts as a "multi-task regularizer" that encourages the parameters to *also* be good at predicting word sequences. The weight of the regularizer is a hyperparameter, as usual.

You are welcome to try any of the ideas above for extra credit.

[7] One quick method is to include many copies of them in the semi-supervised training corpus.

[8] This could mean that the Viterbi tagging doesn't get worse than it was in the supervised model, or that the posterior tagging doesn't get worse, or that $\log p(\boldsymbol{t} \mid \boldsymbol{w})$ doesn't decrease (where $\boldsymbol{t}$ is the supervised tagging).

This last one would actually be pretty easy to try, using one Lagrange multiplier per sentence. In fact, it's similar to the method below that puts $\log p(\boldsymbol{t} \mid \boldsymbol{w})$ directly into the objective. The difference is that now the weight of that term will vary by sentence: it's a Lagrange multiplier that will *automatically* be increased as much as necessary in order to get $\log p(\boldsymbol{t} \mid \boldsymbol{w})$ as high as it was in the supervised model. No hyperparameter is needed.

B.3 Procedures for Optimizing A, B

Relative frequency estimation. If you have fully supervised data, then the log-likelihood is

$$\mathcal{L}(A,B) \overset{\text{def}}{=} \sum_m \log \underbrace{p(\boldsymbol{t}_m, \boldsymbol{w}_m)}_{\text{given by (1)}} \tag{15}$$

$$= \left(\sum_{i=1}^{n+1} \log p_{\text{A}}(t_i \mid t_{i-1})\right) + \left(\sum_{i=1}^{n} \log p_{\text{B}}(w_i \mid t_i)\right) \tag{16}$$

and as mentioned above, this is trivial to optimize! The log-likelihood function is concave, and the globally optimal values are just count ratios, as on the language modeling homework:

$$p_{\text{A}}(t_i \mid t_{i-1}) = \frac{c(t_{i-1}, t_i)}{c(t_{i-1})} \qquad\qquad p_{\text{B}}(w_i \mid t_i) = \frac{c(t_i, w_i)}{c(t_i)} \tag{17}$$

To resist overfitting to the training data, you can use add-λ smoothing or another smoothing technique, as noted above.

Gradient ascent. When some or all tags are unobserved, the log-likelihood or regularized log-likelihood function from reading section B.1 can have multiple local optima. The global optimum is now very hard to find. But we can at least try to improve the function up to a local optimum.

How should we do that? Gradient ascent is an obvious direct approach. You already know how to do SGD from the language modeling homework. We'll switch to SGD when we reach CRFs (question 4 and reading section E), and we'll continue using SGD on the next homework.

The EM algorithm. However, first we'll explore maximizing log-likelihood by Expectation-Maximization (EM), as illustrated in class with the ice cream spreadsheet. EM is an attractive method that is often faster. The M step often makes a big change to the parameters in place of taking many little steps as in gradient ascent. It's not even necessary to choose a stepsize! The EM method is developed in the following sections, without proofs.

B.4 Monte Carlo Expectation-Maximization (MCEM)

The incomplete-data likelihood (13) *sums* over all possible taggings for each sentence. A variant approach is to *impute* each sentence's tagging, i.e., guess what it might be. This gives us a "supervised" version of the corpus, on which we can train our parameters to maximize the *complete*-data likelihood. In other words, we use imputation to convert the unsupervised learning problem into a supervised learning problem that we can solve with equation (17)!

How does this work? For each sentence $\boldsymbol{w}_m$ in the training corpus, we randomly sample a *guess* $\hat{\boldsymbol{t}}_m$ of its tag sequence. The guess $\hat{\boldsymbol{t}}_m$ should be sampled from the posterior distribution $p(\boldsymbol{t} \mid \boldsymbol{w}_m)$ defined by the *current* model parameters. In other words, we will probably sample a tagging that the model thinks has a high chance of being correct.

If we pretend that $\hat{\boldsymbol{t}}_m$ really is correct, then $(\hat{\boldsymbol{t}}_m, \boldsymbol{w}_m)$ is a *supervised* version of the sentence. However, of course it's just a guess—the true tags $\boldsymbol{t}_m$ are missing (or "hidden") and we have no way to know for sure what they were. $\hat{\boldsymbol{t}}_m$ may not even be a *good* guess, because the current model parameters aren't correct. But we can iterate the procedure, which will work in some sense.

We use the current parameters to make the guesses (the **Monte Carlo E step**). Then we do supervised training via (17) to re-estimate the parameters (the **M step**). So our next E step will have better guesses, and so on. This iterative procedure is known as the **Monte Carlo EM algorithm (MCEM)**.[9]

Correctness. Does this algorithm maximize the likelihood? A first problem is that it might not even converge. Even if we were to reach the maximum-likelihood parameters, we wouldn't stay there, because on the next iteration we'd make different random guesses, so equation (17) would change. In general, our estimated probabilities (17) will keep bouncing around due to the randomness.

However, the law of large numbers implies that if the total number of sentences M is large, the count ratios in (17) will usually be pretty close to their theoretical expectation. If M is small, you can get the same effect by sampling S different taggings per sentence, increasing the number of randomly tagged sentences to SM. (This is known as "multiple imputation" and is widely used in statistics to deal with missing data.) As $S \to \infty$, the counts $c(\cdots)$ get large and the variance in the estimated probabilities (17) shtrinks to 0. Thus, one way to gradually remove the randomness is to increase S on each iteration. MCEM then indeed converges to a (local) maximum of log-likelihood.[10]

Efficiency. MCEM is easy to understand: it simply converts semi-supervised learning to supervised learning by multiple imputation. It is practically useful in advanced machine learning models where the summation in equation (13) is computationally expensive, but where imputation is cheaper. That is, it's useful when there's an efficient procedure[11] for sampling the latent variable $\boldsymbol{t}$ from the posterior $p(\boldsymbol{t} \mid \boldsymbol{w}_m)$, either exactly or approximately.

But what is the sampling procedure for HMMs? That is, how do we draw a random tagging from the posterior distribution $p(\boldsymbol{t} \mid \boldsymbol{w}_m)$? A "backward sampling" algorithm is shown in Algorithm 3. It is sort of a randomized version of the Viterbi algorithm—instead of following the *best* backpointers to find the *most probable* tag sequence, it follows *random* backpointers to find a *random* tag sequence. Of course, the tag sequences aren't all equally probable! If most of the paths to `Noun` at token 8 come through `Det` at token 7, then most of $\alpha_{\texttt{Noun}}(8)$ will be contributed by $\alpha_{\texttt{Det}}(7)$, and the random backpointer$_{\texttt{Noun}}(8)$ will probably be `Det`.

This sampling procedure is efficient, but it's not *more* efficient than summation. To figure out which paths contribute most, we first have to do all the work of the forward algorithm. So in the case of HMMs, sampling one tagging is not in fact more efficient than summing over all of the taggings.

[9] "Monte Carlo" algorithms are algorithms that use randomness. The name was originally a military code name, referring to a casino in Monaco—casinos also use randomness.

[10] An alternative way to get MCEM to converge is to replace the M steps with SGD steps. At each step, sample a mini-batch of sentences $\boldsymbol{w}_m$, from the training corpus, then impute tags $\hat{\boldsymbol{t}}_m$ for each sentence in the mini-batch, and then take a gradient step on these supervised sentences to try to improve the mini-batch log-likelihood $\sum_m \log(\hat{\boldsymbol{t}}_m, \boldsymbol{w}_m)$. There is already randomness in the choice of mini-batch, so the additional randomness in the imputation is not a problem—SGD can still converge due to the decreasing stepsize over time. Recall that the stochastic gradient in SGD only has to be correct in expectation. In our setting, it's not actually true that the expectation of $\nabla \log p(\hat{\boldsymbol{t}}_m, \boldsymbol{w}_m)$ is $\nabla \log p(\boldsymbol{w}_m)$. But even so, this procedure also turns out to be correct; it again converges to a (local) maximum of log-likelihood.

[11] Often a Markov chain Monte Carlo method such as Gibbs sampling.

Algorithm 3 Forward algorithm with backward sampling. This is a variant of the forward algorithm (Algorithm 1) that keeps *randomized* backpointers, and then follows them just as in the Viterbi algorithm (Algorithm 2).

```
function RANDOM-TAGGING($\boldsymbol{w}$, $\boldsymbol{\tau}$)
    $w_0 \leftarrow$ BOSW; $\tau_0 \leftarrow \{$BOS$\}$                ▷ precede word/tag sequence with BOSW/BOS
    $n = |\boldsymbol{w}|$; $w_{n+1} \leftarrow$ EOSW; $\tau_{n+1} \leftarrow \{$EOS$\}$   ▷ terminate word/tag sequence with EOSW/EOS
    ▷ all $\alpha$ values are initially 0
    $\alpha_{\text{BOS}}(0) \leftarrow 1$                              ▷ "start" node of the graph
    for $j \leftarrow 1$ to $n+1$ :
        for $t_j \in \tau_j$ :
            for $t_{j-1} \in \tau_{j-1}$ :
                $p \leftarrow p_A(t_j \mid t_{j-1}) \cdot p_B(w_j \mid t_j)$   ▷ probability of the $t_{j-1} \rightarrow t_j$ arc
                $\alpha_{t_j}(j) \mathrel{+}= \alpha_{t_{j-1}}(j-1) \cdot p$
                with probability of $(\alpha_{t_{j-1}}(j-1) \cdot p)/\alpha_{t_j}(j)$   ▷ keep a running sample
                    $\text{backpointer}_{t_j}(j) \leftarrow t_{j-1}$

    ▷ follow backpointers to extract a random tag sequence that ends at the final state (EOS at time $n+1$)
    $t_{n+1} \leftarrow$ EOS
    for $j \leftarrow n+1$ downto $1$ :
        $t_{j-1} \leftarrow \text{backpointer}_{t_j}(j)$

    return $\boldsymbol{t}$
```

Therefore, we won't ask you to use Monte Carlo EM. Instead, we'll use true EM, which uses the forward-backward algorithm. True EM is equivalent to Monte Carlo EM with $S = \infty$. This also avoids the noise from random sampling.[12]

B.5 Expectation-Maximization (EM)

The procedure shown on the spreadsheet in class was the classical EM procedure, which can be understood as "MCEM without randomness." EM averages over all ways of guessing the tag sequence, instead of randomly sampling just one (or a finite number $S > 1$) as in MCEM.

So instead of increasing $\log p(\hat{\boldsymbol{t}}_m, \boldsymbol{w}_m)$ for a *particular* (random) tag sequence $\hat{\boldsymbol{t}}_m$, EM increases the expected value of that quantity, namely $\sum_{\boldsymbol{t}} p(\boldsymbol{t} \mid \boldsymbol{w}_m) \cdot \log p(\boldsymbol{t}, \boldsymbol{w}_m)$. This is the behavior that multiple imputation approaches in the limit $S \rightarrow \infty$.

Traditional EM doesn't even have SGD's randomness of choosing different examples $\boldsymbol{w}_m$ at different steps.[13] Rather, the **M step** is a "batch" procedure that tries to maximize the expected log-likelihood of the whole training corpus, i.e.,

$$\sum_m \sum_{\boldsymbol{t}} \underbrace{p(\boldsymbol{t} \mid \boldsymbol{w}_m)}_{\text{hold constant}} \cdot \log p(\boldsymbol{t}, \boldsymbol{w}_m) \tag{18}$$

It's important to understand exactly what's being maximized here. The coefficients $p(\boldsymbol{t} \mid \boldsymbol{w}_m)$ are computed by the **E step** and are then held fixed during the M step. That is, the first p in this

[12] Again, the other option would be to simply follow the gradient of the incomplete-data log-likelihood (13), which involves a summation that can be computed by the forward algorithm.

[13] Although that is a possible variant: Liang & Klein (2009) found that "stepwise EM" using minibatches is quite effective. Having some randomness in the objective can help jolt the optimizer out of local optima.

formula is a constant copy of the "current" p. We adjust only the second p during maximization. This looks for new parameters A, B that give high log-probabilities $\log p(\boldsymbol{t}, \boldsymbol{w}_m)$ to the "currently" probable taggings $\boldsymbol{t}$. Equation (18) is convex in A, B, so we really can globally maximize it.

Once we've maximized equation (18), we use these new parameters to make new guesses—another E step—and then do another M step, and so on. Again, this turns out to be a valid procedure for (locally) maximizing equation (13).[14]

Crucially, for an HMM, we can maximize equation (18) efficiently.[15] We don't have to iterate over all k^n possible taggings for a sentence of length n. It turns out that to maximize equation (18) at the M step, *we only need the total fractional counts of the transitions and emissions*, which we feed into equation (17).[16] So that is all that the E step computes. For a given sentence, the forward-backward algorithm of Algorithm 4 computes the fractional counts in $O(nk^2)$ time via dynamic programming. We compute the total counts by running Algorithm 4 on all the sentences.[17]

The M step is just like training on a supervised corpus. Each untagged training sentence $\boldsymbol{w}_m$ has given rise to lots of tagged training sentences $(\boldsymbol{t}, \boldsymbol{w}_m)$, which are all "fractionally" in the corpus. Our old supervised training methods work. Specifically, you can maximize equation (18) by setting the transition and emission probabilities as ratios (17) of *fractional* counts.

B.6 Don't Guess When You Know

"Don't guess when you know" is a good general principle. If there are things that *can't* or *must* happen, then you may as well ensure that their probabilities in your model are 0 or 1, respectively.

The relative frequency estimates (17) will actually get this right—e.g., they will assign probability 0 to something that never happened in your reconstructed supervised taggings. But what if you smooth those estimates? Smoothing tries to avoid estimating probabilities as 0. A naive implementation of add-λ smoothing at the M step will produce estimates in $(0, 1)$ even for probabilities that you *know* must be 0 or 1.

It's not the end of the world if you estimate 0.00001 for something that you know is 0. But it's good software design to nail down that known 0, unless handling that special case makes the code slow or ugly. This is known as a "structural zero" in the model—it's part of the model's structure, not something you estimate.

In particular, we know about p_{B}:

- $p_{\text{B}}(\textsc{bosw} \mid \textsc{bos}) = 1$

[14] Why? It can be shown (via something called Jensen's inequality) that equation (18) is always $\leq$ equation (13). So by finding parameters that make the lower bound (18) high, we can ensure that the log-likelihood (13) is also high. Furthermore, it can be be shown that when EM converges—meaning that two successive E steps return the same distribution $p(\boldsymbol{t} \mid \boldsymbol{w}_m)$—the two quantities are equal and both are at a local maximum. Thus, EM locally maximizes equation (13). Monte Carlo EM (reading section B.4) is essentially the stochastic version of this procedure.

[15] For fancier models where this is not true, a useful generalization is **variational EM**. Namely, when there there is no efficient way to work with $p(\boldsymbol{t} \mid \boldsymbol{w}_m)$ in equation (18), we can use a more tractable distribution $q_m(\boldsymbol{t})$ over the taggings. No matter what distribution q_m we pick, we'll still get a lower bound on the log-likelihood (still by Jensen's inequality). So the variational E step tries to increase the lower bound $\sum_m \sum_{\boldsymbol{t}} q_m(\boldsymbol{t}) \cdot \log p(\boldsymbol{t}, \boldsymbol{w}_m)$ by reestimating the distributions q_m (while keeping them tractable), and the M step tries to increase it by reestimating the model p.

[16] Can you prove this? Hint: start by using equation (1) to expand the $\log p$ term in equation (18).

[17] How about a matrix version? Footnote 1 noted that the forward algorithm amounted to computing a sequence of row vectors via $\vec{\alpha}(j) = \vec{\alpha}(j-1)P(w_j)$, where the base case $\vec{\alpha}(0)$ is the row vector that is one-hot at BOS. In the same way, the backward algorithm amounts to computing a sequence of column vectors via $\vec{\beta}(j-1) = P(w_j)\vec{\beta}(j)$, where the base case $\vec{\beta}(n+1)$ is the column vector that is one-hot at EOS. We obtain Z as $\alpha_{\text{EOS}}(n+1)$ or alternatively as $Z = \beta_{\text{BOS}}(0)$. The posterior distribution over the tag T_j is given by the vector $\vec{\alpha}(j) \odot \vec{\beta}(j) \,/\, Z$.

Algorithm 4 Forward-backward algorithm for computing emission counts $c(t, w)$ and expected transition counts $c(s, t)$. This algorithm increments the existing counts $c(\dots)$ by the counts from this sentence.

```
function FORWARD-BACKWARD($\boldsymbol{w}$, $\boldsymbol{\tau}$)
    run Algorithm 1 to compute the $\alpha$ values as well as $n, w_0, \tau_0, w_{n+1}, \tau_{n+1}$, and $Z$
    ▷ now for the backward pass; all $\beta$ values are initially 0
    $\beta_{\text{EOS}}(n+1) \leftarrow 1$
    for $j \leftarrow n+1$ downto 1 :
        for $t_j \in \tau_j$ :
            $c(t_j, w_j) \mathrel{+}= \alpha_{t_j}(j) \cdot \beta_{t_j}(j) \,/\, Z$        ▷ increment count by $p(T_j = t_j \mid \boldsymbol{w}, \boldsymbol{\tau})$
            for $t_{j-1} \in \tau_{j-1}$ :
                ▷ find total prob of all suffix paths that start with $t_{j-1}t_j$,
                  and add it to total prob of suffix paths starting with $t_{j-1}$
                $p \leftarrow p_{\text{A}}(t_j \mid t_{j-1}) \cdot p_{\text{B}}(w_j \mid t_j)$    ▷ probability of the $t_{j-1} \to t_j$ arc (same as in forward pass)
                $c(t_{j-1}, t_j) \mathrel{+}= \alpha_{t_{j-1}}(j-1) \cdot p \cdot \beta_{t_j}(j) \,/\, Z$    ▷ increment by $p(T_{j-1} = t_{j-1}, T_j = t_j \mid \boldsymbol{w}, \boldsymbol{\tau})$
                $\beta_{t_{j-1}}(j-1) \mathrel{+}= p \cdot \beta_{t_j}(j)$    ▷ left-extend suffix path starting with $t_j$; add to running total
    assert $\beta_{\text{BOS}}(0) = Z$        ▷ backward algorithm computes same sum as forward
```

- $p_{\text{B}}(\text{BOSW} \mid t) = 0$ for $t \neq \text{BOS}$
- $p_{\text{B}}(\text{EOSW} \mid \text{EOS}) = 1$
- $p_{\text{B}}(\text{EOSW} \mid t) = 0$ for $t \neq \text{EOS}$

To avoid all of these special cases, let's just say that B only provides the emission probabilities $p_{\text{B}}(w_j \mid t_j)$ for $j \in [1, n]$, so it never conditions on $t_0 = \text{BOS}$ or $t_{n+1} = \text{EOS}$. In fact, equation (1) only considers those emission probabilities. (It's true that Algorithm 1 line 11 does look up $p(\text{EOSW} \mid \text{EOS})$, but that factor can be skipped or forced to 1 by a simple "if" test. There's no need to actually look it up in B.)

Then we don't need rows in the B matrix for BOS and EOS. And we don't need columns for BOSW and EOSW—we can actually leave them out of the vocabulary. Then each row of B is a probability distribution over the vocabulary and cannot give any probability to BOSW or EOSW.

You can physically omit the rows in B for BOS and EOS if you add them last to the tag integerizer; then a matrix with $k-2$ rows will have rows indexed by the other tags. Alternatively, it's okay if those tags have rows in the matrix, but they should never be accessed. Set them to 0, or to `NaN` so that you get an error if they are accidentally used.

We also know about p_{A}:

- $p_{\text{A}}(\text{BOS} \mid s) = 0$ for any preceding tag s
- $p_{\text{A}}(t \mid \text{EOS})$ doesn't matter (will not be used)

So it would be nice to omit the BOS column and the EOS row from the A matrix.

The problem is that $\vec{\alpha}(j)$ should have k elements—one for each tag including BOS (nonzero for $j = 0$) and EOS (nonzero for $j = n+1$). So if we're going to multiply by A to transform that vector (equation (12)), we need A to be $k \times k$.

The best option is probably to force the BOS column to be 0. For example, if you set the BOS column of W^A to $-\infty$, then exponentiating it will give a column of zeroes in A. Then you can set the EOS row to be `NaN`.

(In Python, use `math.nan` and `-math.inf`, or `float("nan")` and `float("-inf")`.)

C Numerical Issues

For long sentences or large vocabularies, the α and Z probabilities for a sentence will be quite small. You'll need a method for preventing underflow. Here are a few options.[18]

C.1 The Scaling Trick

A simple, efficient, and effective option is to just scale the probabilities up when they get small. Recall our update rule (equation (12)):

$$\vec{\alpha}(j) \leftarrow (\vec{\alpha}(j-1) \cdot A) \odot B_{\cdot w_j} \tag{19}$$

Suppose after we compute this, we divide it by a small constant κ_j:

$$\vec{\alpha}(j) \leftarrow \vec{\alpha}(j) \,/\, \kappa_j \tag{20}$$

Now Z in Algorithm 1 will be off by a factor of $\prod_j \kappa_j$. But you can easily multiply that back in at the end of the algorithm (line 13).

$$Z \leftarrow \alpha_{\text{EOS}}(n+1) \cdot \prod_j \kappa_j \tag{21}$$

except that to avoid underflow, you should switch to logspace:

$$\log Z \leftarrow \log \alpha_{\text{EOS}}(n+1) + \sum_j \log \kappa_j \tag{22}$$

So you just have to keep a running total $\sum_j \log \kappa_j$ as you go. The only reason to keep this running total is to report $\log Z$ accurately so that you can watch this log-likelihood improve over time.

You could choose κ_j to be any positive constant. It's just a bookkeeping convenience to avoid underflow. An elegant choice is $\kappa_j = \sum_t \alpha_t(j)$ because then the rescaled $\vec{\alpha}(j)$ then gives the relative probabilities of the tags (i.e., they sum to 1). Another reasonable choice would be $\kappa_j = \max_t \alpha_t(j)$, in which case the rescaled $\vec{\alpha}(j)$ will have a max value of 1.

A couple of fancy improvements you could consider:

- At steps j where this sum or max is not very small, you could optionally just skip the rescaling step (equivalent to taking $\kappa_j = 1$). This means you only have to rescale every so often, when the probabilities are in danger of underflowing.

- You could ensure that κ_j is a power of 2, which means that there are especially fast ways to multiply $\vec{\alpha}(j)$ by κ_j. (This is probably not worth it, unless you're writing low-level code, but it connects nicely to the discussion in reading section C.3 below.)

Note that this trick works for tagging specifically. How would you make it work for parsing? (The most obvious approach is wrong. I can see a solution, but it took a little thought!) The next solution is more general, but also more annoying.

[18] These methods can also help prevent *overflow*. Overflow could arise during exponentiation in equation (54) or equations (31)–(45) if you have allowed the weights to get too large. However, if the weights are really large enough to cause overflow, then it probably means you've taken too large a gradient step or you haven't regularized enough.

C.2 Log Probabilities

A common general trick for avoiding underflow is to store your probabilities "in logspace" (or equivalently, "in the log domain"). That is, when you see something like $p = q \cdot r$, implement it as something like $lp = lq + lr$, where lp, lq, lr are variables storing the logs of the probabilities. Note that $\log 0 = -\infty$. (Which is a number, not an exception. `numpy.log(0)` will indeed return the floating-point representation of $-\infty$, which can also be obtained directly by `-math.inf` or `-numpy.inf`. PyTorch behaves similarly.)

Addition in logspace. The forward algorithm requires you to add probabilities, as in $p \leftarrow p+q$. But you are now storing these probabilities p and q as their logs, lp and lq.

You might try to write $lp \leftarrow$ `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, the function `logaddexp(`lp`,` lq`)` is available in numpy and PyTorch. It safely does the computation above. That is, it adds two values that are represented in logspace. (If you want to sum a whole sequence of values, use `logsumexp`.) How does it work?

$$\texttt{logaddexp}(x,y) \overset{\text{def}}{=} \begin{cases} x + \log(1+\exp(y-x)) & \text{if } y \leq x \\ y + \log(1+\exp(x-y)) & \text{otherwise} \end{cases} \tag{23}$$

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

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

Tensor operations. After you update A and B at the M step (by equation (17) or a smoothed version), you will have to take their logs to get versions lA and lB, whose entries are log-probabilities.

Recall that equation (12) involves a product of the row vector $\vec{\alpha}(j-1)$ with the square matrix A. But if you are representing these using logspace tensors, you can't use the ordinary matrix product operator `@`. You need a version that uses `+` and `logsumexp` (in place of the usual `*` and `sum`). You'll have to figure out how to do this efficiently with PyTorch tensor operations. Just as in reading section A.5, a good first step is to try to implement `@` using only `*` and `sum`.

What kind of logs? To simplify your code and avoid bugs, this section has recommended that you use log-probabilities rather than negative log-probabilities. Then you won't have to remember to negate the output to log or the input to exp. (The convention of negating log-probabilities helps to keep minus signs out of the *printed numbers*; but when you're coding, it's safer to keep minus signs out of the *formulas and code* instead.)

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

Yes, it's conventional to *report* $-\log_2$ probabilities, (the unit here is "bits"). But you can store $\log_e x$ internally, and convert to bits only when and if you print it out: $-\log_2 x = -(\log_e x)/\log_e 2$.

(As it happens, you won't be required to print any log-probabilities for this homework, only perplexities: see equation (38).)

C.3 High Precision Arithmetic

A final possibility is to use a type of number that won't underflow. 32-bit floating point numbers can get as small as 10^{-38}, and 64-bit floating point numbers can get as small as 10^{-308}. But there are Python libraries for high-precision arithmetic, such as `mpmath`, `bigfloat`, and `mxNumber`. These use more than 64 bits to represent a number—indeed, they can expand the size of the representation as necessary.

Recall that a floating point number stores a fixed-point mantissa together with an integer exponent which is basically the $\log_2$ of the number (rounded towards zero). So to avoid underflow, really what we want is to have a lot of bits in the exponent.

The previous alternatives can be actually thought of specialized floating-point representations[19] that focus on the exponent:

- In the scaling trick of reading section C.1, the running total $\sum_{i=1}^{j} \log \kappa_i$ can be regarded as a possibly large, possibly non-integer exponent that is shared by all of the values in $\vec{\alpha}(j)$.

- The logspace trick of reading section C.2 can be thought of as a numeric representation where the mantissa is 1 and the exponent is itself a floating point number (allowing it to be very large). Indeed, `logaddexp` is actually rather similar to floating-point addition.[20]

Using the general high-precision arithmetic libraries is probably too slow and unnecessarily precise, and won't work with PyTorch.

D Posterior Decoding

Decoding refers to extracting a prediction from the model. Recall that a Viterbi decoder (reading section A.5) finds the single most likely overall sequence. By contrast, a posterior decoder will separately choose the best tag at each position—the tag with highest posterior marginal probability—even if this gives an unlikely overall sequence.

In Algorithm 4, the probability that tag T_j has value t_j is computed at line 7 (which adds it to $c(t_j, w_j)$). The posterior marginal distribution over all possible tags at position j can be found efficiently as $\vec{\alpha}(j) \odot \vec{\beta}(j) \,/\, Z$.[21]

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

[19] Indeed, the previous alternatives could be packaged up as numeric classes. For example, you could make a `PosNum` class that stores any positive number p as $\log p$ (and perhaps stores 0 as $-\infty$), which enables it to represent very small probabilities. You'd then implement arithmetic operations such as `*`, `+`, and `max`. You'd also need a constructor that turns a nonnegative real into a `PosNum` (using log), and a method for extracting the real value of a `PosNum` (using exp).

[20] Before adding two numbers, floating-point addition has to multiply the mantissa of the smaller number by 2^{y-x}, where $y - x \leq 0$ is the difference between the integer exponents. This is similar to equation (23). But in the floating-point case, multiplying the mantissa by 2^{y-x}—which is an integer power of 2—is easily accomplished by right-shifting the bits of its binary representation.

[21] In the next homework, reading section K will lay out a way to compute this without implementing Algorithm 4 (the backward pass). Instead, it uses back-propagation.

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

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

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

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

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

More generally, Minimum Bayes Risk decoding is an important idea: you should pick the output that minimizes your **risk** (expected loss) given the evidence. Posterior decoding is just the special case where the loss is defined to be the number of tags you get wrong.

E Simple Conditional Random Fields (CRFs)

E.1 Discriminative Training of a Generative Model

An HMM defines $p(\boldsymbol{t}, \boldsymbol{w})$. It also defines the conditional distribution

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = p(\boldsymbol{t}, \boldsymbol{w}) \,/\, p(\boldsymbol{w}) \tag{24}$$

This is the ratio of equations (1) and (3); the denominator $p(\boldsymbol{w})$ can be computed by the forward algorithm. Posterior decoding and Viterbi decoding depend only on $p(\boldsymbol{t} \mid \boldsymbol{w})$—they are trying to find the individual tags or tag sequences that are most probable given $\boldsymbol{w}$.

If we have supervised data, we can decide to train our HMM *discriminatively*, meaning that we are only trying to maximize the *conditional* log-likelihood:

$$\mathcal{L}(A, B) \stackrel{\text{def}}{=} \sum_m \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m) \tag{25}$$

In constrast, our previous procedure was to train it *generatively*, meaning that we are trying to maximize the *joint* log-likelihood (15):

$$\mathcal{L}(A,B) \overset{\text{def}}{=} \sum_m \log p(\boldsymbol{t}_m, \boldsymbol{w}_m) = \sum_m \; \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m) \;+\; \underbrace{\log p(\boldsymbol{w}_m)}_{\text{excluded from (25)}} \tag{26}$$

The difference is that the discriminative objective (25) is wholly focused on the tagging task. The parameters are optimized only to predict the tags given the words (at least on training data). They are no longer trying to predict the observed training words as well, as the generative objective (26) does. This may result in parameters that work better for the actual tagging task.

E.2 Discriminative Modeling

But wait! If our goal is only to maximize equation (25), then we don't even need to start with a generative model like an HMM as in equation (24). We can write a formula to define $p(\boldsymbol{t} \mid \boldsymbol{w})$ directly rather than by the ratio (24).

Such a *discriminative* model has no opinion about $p(\boldsymbol{w})$, since $\boldsymbol{w}$ is only ever used as a conditioning variable. It is capable of choosing among taggings $\boldsymbol{t}$ for a given sentence $\boldsymbol{w}$. But unlike the generative model, we can't use it to generate sentences $\boldsymbol{w}$.

Thus, we can't train it to maximize the incomplete-data log-likelihood $\sum_m p(\boldsymbol{w}_m)$ (equation (13)), nor the complete-data log-likelihood $\sum_m p(\boldsymbol{t}_m, \boldsymbol{w}_m)$ (equations (15) and (26)).

We can only train it to maximize equation (25) (or if not all of the tags are observed, a version of (25) that marginalizes over the hidden tags).

E.3 A Simple Linear-Chain CRF

In this homework and the next homework, we'll work with a specific family of discriminative models: **linear-chain conditional random fields (CRFs)**.

In this homework, we'll use a simple restricted family of CRFs. They can model exactly the same conditional distributions $p(\boldsymbol{t} \mid \boldsymbol{w})$ as we could get by conditionalizing HMMs (equation (24)). However, we will parameterize them a bit differently, and since we are now maximizing the discriminative objective (25), we will switch from EM to SGD as our training method.

(This is warmup for the next homework, where we'll unlock the full power of CRFs by incorporating many more features to help discriminate among taggings $\boldsymbol{t}$. The resulting CRFs will be richer than HMMs, but will remain efficient thanks to their linear-chain structure.)

The formula and algorithms are very similar to the conditional HMM case. Instead of equations (1), (3) and (24), we have

$$\tilde{p}(\boldsymbol{t}, \boldsymbol{w}) = \left(\prod_{i=1}^{n+1} \phi_{\mathrm{A}}(t_{i-1}, t_i)\right) \cdot \left(\prod_{i=1}^{n} \phi_{\mathrm{B}}(t_i, w_i)\right) \tag{27}$$

$$Z(\boldsymbol{w}) = \sum_{\boldsymbol{t}} \tilde{p}(\boldsymbol{t}, \boldsymbol{w}) \tag{28}$$

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = \tilde{p}(\boldsymbol{t}, \boldsymbol{w}) \,/\, Z(\boldsymbol{w}) \tag{29}$$

This can also be expressed more compactly, using the "proportional to" symbol $\propto$,

$$p(\boldsymbol{t} \mid \boldsymbol{w}) \propto \left(\prod_{i=1}^{n+1} \phi_{\mathrm{A}}(t_{i-1}, t_i)\right) \cdot \left(\prod_{i=1}^{n} \phi_{\mathrm{B}}(t_i, w_i)\right) \tag{30}$$

where the constant of proportionality is understood to be "whatever ensures that $\sum_{\boldsymbol{t}} p(\boldsymbol{t} \mid \boldsymbol{w}) = 1$," namely $\frac{1}{Z(\boldsymbol{w})}$.

The factors in equation (27) or (30), which must be ≥ 0, are returned by the **potential functions** ϕ_{A} and ϕ_{B}. The conditional HMM was just a special case where these factors are conditional probabilities, with the result that $\tilde{p}$ is actually a probability distribution over tagged sentences: $\sum_{\boldsymbol{t},\boldsymbol{w}} \tilde{p}(\boldsymbol{t}, \boldsymbol{w}) = 1$.

The above notation is inherited from our description of conditional log-linear models. That is, $\tilde{p}$ in equation (27) denotes an *unnormalized* probability distribution—just like a probability distribution, but it might not sum to 1. We use p to denote a normalized version, which in this case is the conditional distribution (29).

$Z(\boldsymbol{w})$ can be computed using the same matrix formulas (equation (2) and reading section A.4) and the same forward algorithm code as before. The only difference is that your matrices A, B will now store the potentials $\phi_{\mathrm{A}}, \phi_{\mathrm{B}}$ instead of the conditional probabilities p_A, p_B as in Algorithm 1.

Viterbi decoding and posterior decoding are also unchanged.

E.4 Training CRFs

It's no longer the case that each row of A and B must sum to 1. However, the matrices A and B still have to be non-negative. To ensure that this remains true during SGD training, let's obtain the potentials by exponentiating some real-valued weights, which will be the underlying parameters of the models:

$$\phi_{\mathrm{A}}(s,t) = \exp W^A_{st} > 0 \qquad\qquad \phi_{\mathrm{B}}(t,w) = \exp W^B_{tw} > 0 \tag{31}$$

Thus the potential matrices are obtained as $A = \exp W^A$, $B = \exp W^B$ (where the exp is applied separately to each matrix element, as in PyTorch). The parameters in W^A, W^B are allowed to take on any real values, so there is no constraint that could be violated by an SGD step.

Let's write out the conditional log-likelihood training objective (25) in terms of these parameters:

$$\mathcal{L}(W^A, W^B) \overset{\text{def}}{=} \sum_{m=1}^{M} \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m) \tag{32}$$

where

$$\begin{aligned}\log p(\boldsymbol{t} \mid \boldsymbol{w}) &= \log \tilde{p}(\boldsymbol{t}, \boldsymbol{w}) - \log Z(\boldsymbol{w}) &(33)\\ &= \left(\sum_{i=1}^{n+1} W^A_{t_{i-1},t_i} + \sum_{i=1}^{n} W^B_{t_i,w_i}\right) - \log Z(\boldsymbol{w}) &(34)\end{aligned}$$

where $n = |\boldsymbol{t}| = |\boldsymbol{w}|$. This reveals that the CRF really is just a conditional log-linear model over taggings $\boldsymbol{t}$:

- Since W^A_{st} is added in once for every token of the tag bigram st, we can regard it as the weight of a count-valued feature that counts such tokens.

- Since W^B_{tw} is added in once for every token of the word-tag pair tw, we can regard it as the weight of a count-valued feature that counts such tokens.

You are already familiar with log-linear models. Here we happen to have a large, structured space of possible outputs. So $Z(\boldsymbol{w})$ has to sum over k^n different possible taggings of $\boldsymbol{w}$ that compete with the supervised tagging $\boldsymbol{t}$. But that's okay! Algorithm 1 computes it by dynamic programming.

You are also familiar with how to *train* log-linear models using SGD. Following the language modeling homework, we again define the training objective to be

$$F(W^A, W^B) \overset{\text{def}}{=} \tfrac{1}{M}\left(\mathcal{L}(W^A, W^B) - C \cdot R(W^A, W^B)\right) \tag{35}$$

which is the regularized conditional log-likelihood per training sentence. (There are M training sentences.) Let $R(W^A, W^B)$ be the L_2 regularizer, which adds up all the squared weights in W^A and W^B.[22] As always for log-linear models, this objective is convex as a function of the parameters W^A and W^B.

Each step of SGD draws a random supervised example $(\boldsymbol{t}_m, \boldsymbol{w}_m)$, and takes a small step in the direction of the gradient $\nabla \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m) - \nabla \frac{C}{M} R(W^A, W^B)$. Or you can add together the gradients of several random examples in a minibatch.

- The gradient term $\nabla \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m)$ is just the observed feature counts in $(\boldsymbol{t}_m, \boldsymbol{w}_m)$, minus the expected counts given the current parameters.
  - The observed counts are easy to collect by looping over the positions in $(\boldsymbol{t}_m, \boldsymbol{w}_m)$.
  - You've already written code to compute the expected counts—that's exactly the forward-backward algorithm (Algorithm 4)!
- The regularizer gradient $-\nabla \frac{C}{M} R(W^A, W^B)$ is just $-\frac{2C}{M}[A, B]$, which pulls the weights back toward zero as usual.

E.5 A Data Structures Perspective on HMMs vs. CRFs

HMMs are usually trained generatively, while CRFs are usually trained discriminatively. But you *could* train an HMM discriminatively (see reading section E.1).

The more fundamental difference is that **the CRF has a more restricted API**. Everything else flows from that.

As an analogy, a red-black tree and a heap have different APIs. A red-black tree lets you find and remove *any* element quickly. A heap only lets you find and remove the *minimum* element quickly. But sometimes that's all you need. You can't ask a heap "do you contain 1234?" or "what is your 100th largest element?" A heap doesn't have those methods. But a heap does have an advantage as a result: it is simpler and faster than a red-black tree.

An HMM is a joint model that defines $p(\boldsymbol{t}, \boldsymbol{w})$. Of course this means that it also defines $p(\boldsymbol{w}) = \sum_{\boldsymbol{t}} p(\boldsymbol{t}, \boldsymbol{w})$, so you can use it as a language model and train it on untagged sentences $\boldsymbol{w}$. And as a result, it defines $p(\boldsymbol{t} \mid \boldsymbol{w}) = p(\boldsymbol{t}, \boldsymbol{w})/p(\boldsymbol{w})$, so you can use it as a tagger.

[22] This is sometimes written as $||W^A||_F^2 + ||W^B||_F^2$, where $|| \cdot ||_F$ denotes the **Frobenius norm** of a matrix.

A CRF is a conditional model that defines only $p(\boldsymbol{t} \mid \boldsymbol{w})$. As reading section E.2 said, you can't ask it for $p(\boldsymbol{t}, \boldsymbol{w})$: it doesn't have that method. You can't use it as a language model or train it on untagged sentences $\boldsymbol{w}$. You can *only* use it as a tagger.

Since the CRF doesn't define $p(\boldsymbol{t}, \boldsymbol{w})$, of course you can't train it to maximize $\sum_m \log p(\boldsymbol{t}_m, \boldsymbol{w}_m)$ either. You have to train it to maximize something else, namely $\sum_m \log p(\boldsymbol{t}_m \mid \boldsymbol{w}_m)$.

But that discriminative objective might be what you want anyway, as it is usually more predictive on the discriminative task—see reading sections E.1–E.2.

Discriminative training is admittedly a little harder because (1) you need supervised data and (2) you have to take gradient steps. The HMM also allows generative training. Generative training is easy if you have supervised data: you can just count and divide. And you can still do it even if you have semi-supervised data: first guess the missing parts (E step), and then you can count and divide (M step).

To repeat, the CRF has a more restricted API. What's the advantage? It doesn't have to ensure that $\sum_{\boldsymbol{t},\boldsymbol{w}} p(\boldsymbol{t}, \boldsymbol{w}) = 1$.[23] That quantity is not even defined because $p(\boldsymbol{t}, \boldsymbol{w})$ isn't defined. As a result, it is free to use more complicated features. As we'll see later, it can use non-stationary transition and emission potentials that depend on both left and right context in $\boldsymbol{w}$ (see reading section H.3 in the next homework's reading handout).

F Data Resources for the Homework

There are three datasets, available at `http://cs.jhu.edu/~jason/465/hw-tag/data`.

The datasets are

- **ic**: Ice cream cone sequences with 1-character tags (`C`, `H`). Start with this easy dataset.
- **en**: English word sequences with 1-character tags (documented in Figure 2).
- **cz**: Czech word sequences with 2-character tags.

This homework will focus on the **en** dataset. The **ic** dataset is to help you test your code.

The **cz** dataset is provided just so that you can see what a harder tagging task looks like. The tagset is larger due to the morphological complexity of Czech (in fact, the original tags had 6 characters), and spelling features are more important in Czech than in English. You are welcome to try your code on it!

Each dataset consists of three files:

- **sup**: tagged data for supervised training (**ensup** provides 100,000 words)
- **dev**: tagged data for held-out evaluation (25,000 words for **endev**); your tagger should ignore the tags in this file except when you're measuring its performance
- **raw**: untagged data for reestimating parameters (100,000 words for **enraw**—but in the real world, you'd have far more raw data than supervised data)

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.

[23] It still has to ensure that $\sum_{\boldsymbol{t}} p(\boldsymbol{t} \mid \boldsymbol{w}) = 1$ for each $\boldsymbol{w}$, but it does this by explicitly dividing by the normalizing constant $Z(\boldsymbol{w})$, which it is able to compute efficiently by the forward algorithm.

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)
BOS	Context for the start of a sentence
EOS	Marks the end of a sentence
,	Comma
.	Period
:	Colon, semicolon, or dash
-	Parenthesis
‘	Open quotation mark
’	Close quotation mark
$	Currency symbol

Figure 2: Tags in the **en** dataset. These are the preterminals from `wallstreet.gr` in homework 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 homework. Using only the first letters reduces the number of tags, speeding things up and increasing accuracy. (However, it results in a couple of unnatural categories, C and P.)

F.1 Tag and Word Vocabularies

Take the tag vocabulary to be all the tag types that appeared at least once in **sup**. For simplicity, we will not include an OOV tag. (OOT?) Thus, any novel tag will simply have probability 0 (even with smoothing).[24]

Take the word vocabulary to be all the word types that appeared at least once in **sup** ∪ **raw** (or just in **sup** if no **raw** file is provided, as in the case of `vtag`), plus an OOV type in case any

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

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

out-of-vocabulary word types show up in **dev**.[25]

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

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: if 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. If you are training with SGD, then you should still converge to the same place as EM converges to.

- `icsup` has been designed so that your initial unsmoothed supervised training on it will yield the initial parameters from the spreadsheet (transition and emission probabilities).
- `icdev` has exactly the data from the spreadsheet. Running your Viterbi tagger with the above parameters on `icdev` should produce the same values as the spreadsheet's iteration 0:[26]
    - $\hat{\alpha}$ probabilities for each day
    - weather tag for each day (shown on the graph)[27]

G Measuring Tagging Performance

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

G.1 Accuracy

In these task-specific metrics, you look at some subset of the tokens in your evaluation sentences (**dev** or **test**) and ask what percentage of them received the correct tag.

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

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

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

[26] 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 interprets the interpreting the world in terms of its stereotypes but then uses those interpretations to update its stereotypes. The Viterbi approximation turns it into a blinkered fanatic that is absolutely positive that its interpretation of each sequence is correct, and therefore it learns less, particularly when it has only one sequence to learn from.

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

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

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

Your 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 $Z = p(w_1, \ldots w_n \mid w_0)$ while running the forward-backward algorithm (Algorithm 4). It is the total probability of *all* paths that generate the raw word sequence.

G.2 Perplexity

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

Given a tagged corpus, the model's perplexity per tagged word is given by[28]

$$\text{perplexity per tagged word} = 2^{\text{cross-entropy per tagged word}} \tag{36}$$

where

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

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

$$\begin{align} \text{perplexity per tagged word} &= \exp\left(-\text{log-likelihood per tagged word}\right) \tag{37} \\ &= \exp\left(-\frac{\log p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0)}{n}\right) \tag{38} \end{align}$$

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

Why is the corpus probability in the formula conditioned on w_0/t_0? Because the model only generates $w_1/t_1, \ldots, w_n/t_n$. You knew in advance that $w_0/t_0 = \text{BOSW}/\text{BOS}$ would be the left context for generating those tagged words. The model has no distribution $p(w_0, t_0)$. Instead, Algorithm 2 explicitly hard-codes your prior knowledge that $t_0 = \text{BOS}$.

When you have untagged data, you can also compute the model's perplexity on that:

$$\begin{align} \text{perplexity per untagged word} &= \exp\left(-\text{log-likelihood per untagged word}\right) \tag{39} \\ &= \exp\left(-\frac{\log p(w_1, \ldots w_n, \mid w_0 t_0)}{n}\right) \end{align}$$

[28] Using the notation from reading section A.1.

where the forward or backward algorithm can compute

$$p(w_1, \ldots w_n, \mid w_0, t_0) = \sum_{t_1,\ldots,t_n} p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0) \tag{40}$$

Notice that

$$p(w_1, t_1, \ldots w_n, t_n \mid w_0, t_0) = p(w_1, \ldots w_n \mid w_0, t_0) \cdot p(t_1, \ldots t_n \mid \vec{w}, t_0) \tag{41}$$

so the tagged perplexity (37) can be regarded as the product of two perplexities—namely, how perplexed is the model by the words (in (39)), and how perplexed is it by the tags given the words?

To *evaluate* a trained model, you should ordinarily consider its perplexity on *held-out* data. Lower perplexity is better. Of course, likelihood-based training is equivalent to seeking low perplexity on *training* data.

G.3 Baselines and Ablations

When you build a system, you should compare it with simpler methods, to make sure that all of the work you put into it is worthwhile.

In the case of an HMM tagger, a very simple **baseline method** is to tag each word with its most frequent tag from supervised training data. If the word didn't appear in supervised training, just back off and tag it with the most frequent tag overall.

If you can't improve over this baseline, where the improvement is evaluated on held-out data, then there was no point to all of the HMM stuff!

In addition, if you build a fancy system, you should try turning off parts of it to see whether they helped. This is called an **ablation experiment**. For example, you could see whether pretrained word embeddings really improved your accuracy, compared to simple one-hot embeddings.

One good ablation for an HMM is to replace the *bigram* transition probabilities $p_{\mathrm{A}}(t_i \mid t_{i-1})$ with *unigram* transition probabilities $p_{\mathrm{A}}\mathrm{t}(t_i)$. The bigram case is the plain-vanilla definition of an HMM. It is sometimes called a 1st-order HMM, meaning that each tag depends on 1 previous tag—just enough to make things interesting. A fancier trigram version using $p_{\mathrm{A}}(t_i \mid t_{i-2}, t_{i-1})$ would be called a 2nd-order HMM. So by analogy, the unigram case can be called a 0th-order HMM. The order refers to how far we look back when determining the current tag.

Since the unigram HMM throws away the tag-to-tag dependencies, it is really just tagging each word in isolation. To be precise, the best path maximizes $p_{\mathrm{A}}(t_i) \cdot p_{\mathrm{B}}(w_i \mid t_i)$ at each position i separately. But that is simply the Bayes' Theorem method for maximizing $p(t_i \mid w_i)$. So this is really just the baseline method again—picking the most probable tag for each word! The only difference is that it uses smoothed probabilities (which might be better, for example if they are based on pretrained word embeddings, so that raising the probability of an observed word in a context also tends to raise the probability of unseen words with similar embeddings).

Note that a unigram HMM can actually be implemented to be very fast. The most probable tag for each word type can be precomputed. then the tagger can just look up each word token's most probable part of speech in a hash table—with overall runtime $O(n)$. No dynamic programming is needed.