Natural Language Processing (JHU 601.465/665)
Answers to "Structured Prediction" practice problems

1. (a) BELOW.

The gradient is observed - expected, or more precisely,
(empirical expectation) - (model expectation).
In particular, the partial derivative of the log-likelihood with respect to theta[5] is
(empirical expectation of feature 5) - (model expectation of feature 5).
so it's positive if the model expectation is BELOW the empirical expectation.
Intuitively, by increasing theta[5], we move more probability to taggings on which feature 5 fires, which raises the model expectation to more closely match the empirical expectation.

(b) ADDITION, MULTIPLICATION, DIVISION.

You need to know the expected number of positions i in the sentence where you have a P tag preceded by "walked":
w[i-1]="walked" and t[i]=P.

To do this, you can use the forward-backward algorithm to find the posterior probabilities of all tags at all positions i. Then just consider the positions i that are preceded by "walked" and look at the posterior probability p(t[i]=P).

The forward algorithm efficiently uses MULTIPLICATION to extend partial paths by new arcs, and ADDITION to add up different paths to the same state.
The same is true of the backward algorithm. At the end, you have to DIVIDE by the total probability of all paths to get the posterior probability that the paths go through a particular state: alpha_i(P) * beta_i(P) / Z.

(c) 159/52108. Actually, this isn't the "maximum possible" value because it's not possible to quite get up to 159/52108, but you can get arbitrarily close.

For example, if you drive theta[5] toward infinity, the posterior probability of P at positions following "walked" will approach 1. So you will have 159 instances of "walked" and the model thinks that they are all followed by P and the feature will fire every time.

Thus, it fires 159 times on 52108 training examples, for an average of 159/52108 times per example.

(Some of you said 1. But remember that f_5(t,w) is a count over the whole sentence, so it doesn't have to be in the range [0,1].)

(Some of you said 159. But remember that you are looking for the expected value of f_5(t,w) on a random sentence, so you need to average over the whole corpus.)

(d) 0. Again, it's not possible to quite get down to 0, but you can get arbitrarily close, for example by driving theta[5] toward negative infinity.
Then the posterior probability of P at positions following "walked" will approach 0, so the model thinks that it never fires.

(e) 159. Or in a less efficient implementation, 159*k, where k is the number of tag types. Let's go through this in detail.

For any finite theta vector, the model expectation will be somewhere in between (c) and (d), and in fact learning wil try to make it match the empirical expectation, as discussed in part (a).

As part (b) says, you can use the forward-backward algorithm to do this. Here's why:

* The log-linear model states that the probability of a path y, given a sentence x, is proportional to
     exp(sum of weights of features on that path)
  This is a product of exponentiated feature weights.

  (We count a feature multiple times in this sum or product if the feature fires multiple times on the path, i.e., if its count is > 1.)

* The forward-backward algorithm treats the posterior probability of a path y, given a sentence x, as proportional to the product of the probabilities of transitions on that path.

Therefore, to use the forward-backward algorithm, we have to assign "probabilities" to the transitions in the lattice such that the product of a path's transition probabilities will be the product of the exponentiated weights of the features of that path.

For example, to get the "probability" of a transition from X at position i-1 to tag P at position i, you will take the product of the exponentiated weights of the features that fire on that transition.

(The resulting arc weight plays the same role as p(t[i]=P | t[i-1]=X) in an HMM.)

Feature 5 fires only at positions i such that w[i-1]="walked". So you have to look it up 159 times in computing these "probabilities."

If you're reading carefully, you will say "Wait! For each such position i, there are k transitions of the form X --> P, since there are k different possibilities for X. So the answer should be 159*k."

That's true. However, a careful implementation will recognize that the feature is the same for all X, and look it up only once for each position i. It is reused for all the transitions to i.

In fact, the best implementation will make use of the distributive law. If you're going to multiply all of the transitions to a state by the same number, then it's faster to leave out that number at first when adding up the summands to alpha(state), and then FINALLY multiply alpha(state) by that number.

In fact, that is the standard presentation of the forward algorithm. It's also useful in an HMM, where the probability that state H emits 2 ice creams doesn't depend on the state for the previous day, so a careful implementation of an HMM will look it up only once. (Perhaps you discovered this trick in the HMM homework?)

In other words, the standard presentation actually takes the weight of a path to be a product of numbers associated with its transitions AND its states. You can say that feature 5 or the emission probability p(2 | H) as associated with the state at time i, rather than the transition from i-1 to i. This means its weight is used at most once per state, rather than once per transition.

(f) YES. You still use the forward-backward algorithm, which is what allows you to marginalize over exponentially many paths in polynomial time.

The new feature 5 is still asking whether P is good at position i of this sentence. It may take a little longer to compute it, because you have to count the number of copies of "walked" at all positions j < i, instead of just the position i-1. But once you have this count, you can determine the transition probability as before. Then you can run the forward-backward algorithm as before.

Remark: Note that feature 5 is probably the result of instantiating a template (w[i-1], t[i]).

```
    Feature 5 counts the number of positions i in the sentence
        such that (w[i-1], t[i])=(walked,P).
    Perhaps feature 6 counts
                  (w[i-1], t[i])=(walked,D)
    and feature 7 counts
                  (w[i-1], t[i])=(jumped,P).
    These are all instantiations of the same feature template.
```

2. (a) B = beginning of a place name,
I = inside a place name but not the beginning,
O = outside a place name.

```
      O   B      I      B   I    O    O    O  O   O
   From San Francisco New York can seem like a museum
```

Note that the second "B" signals the start of a second place name immediately following the first one. If that "B" were an "I", then the tagging would be incorrect because it would indicate a single place name, "San Francisco New York."

(b) 11 transition values and 10 emission values.

(Alternatively: In Homework 6, we would have said 11 transition values and 12 emission values, because we treated the BOS and EOS tags as emitting special BOS and EOS words with probability 1. By contrast, in this solution, I will assume that the BOS and EOS tags don't emit anything. They only participate in transitions, namely the tag bigrams BOS O at the start and O EOS at the end. Which approach you choose is a matter of taste.)

(c) Remark: The beta notation used here is similar to the notation used in the Viterbi inside algorithm. In other words, beta_t(j,n) is the score of the best tagging that starts with tag t for the substring from position j to position n (that is, the final n-j words: w[j+1], ... w[n]). In the backward algorithm, we usually write beta_t(j,n) as just beta_t(j), because the second argument is always n.

```
j. for j = n to 1  (by step -1)

   So we initialize beta(n,n+1) and then the loop computes
      j = n:     compute beta_t(n-1, n+1) using word w[n]
      j = n-1:   compute beta_t(n-2,n+1) using word w[n-1]
        ...
      j = 1:   compute beta_t(0,n) using word w[1]
   in that order.
```

ii. max_{t'} transition_score(BOS,t') + beta_{t'}(0,n)

This considers the transition from BOS to the initial tag t'.

(Remark: A good name for this quantity would be
beta_BOS(-1,n), but the loops above do not compute that
quantity. They only compute beta_t where t is in {I,O,B},
and they include an emission score.)

iii. log p("O" | "I")
(log because we're adding edge weights, not multiplying them)
(not negated because we're using max, not min)

iv. true. The weights are no longer log-probabilities, though.

v. true. The weights are no longer log-probabilities, though.

(d) Change the second equation to

```
beta_t(i,n+1) = max_j max_{t'} emission_score(t, w[i+1] ... w[j])
                                + transition_score(t,t')
                                + beta_{t'}(j,n+1)
```

For example, to tag "San Francisco" as a place name,
we would take
    i=1 (the start position of "San Francisco")
    j=3 (the end position of "San Francisco")
    t = PLACE
and we consider emission_score(PLACE, San Francisco).

The maximization now maximizes over j as well as t'.
As a result, the runtime is now O(n^2) rather than O(n).
In a straightforward implementation, the outer loop will
now range over i rather than j, with the maximization
over j handled in an inner loop.

This is technically close to something called a "hidden
semi-Markov model."

(e) * example features indicating place name:

all words are capitalized
probably the object of the preposition "from", according to
   our parse forest or our 1-best parse
preceded by "from"
not preceded by a capitalized word
   (where we don't count sentence-initial words as capitalized)

* example features indicating non-named-entity:

starts with "a"
probably starts with a determiner, according to our parse forest
    or our 1-best parse
contains no proper nouns, according to our parse forest
    or our 1-best parse
the negation of any feature in previous list :-)

(f) O(n), O(n^2), O(n^3).

Explanation:

In the HMM program, the slowest rule is the one that
must be instantiated at all O(n) positions J.

In the phrasal tagging program, the slowest rule is the
one that must be instantiated at all O(n^2) position pairs

I,J.

In the parsing program, the slowest rule is the one that must be instantiated at all O(n^3) position triples I,Mid,J.

(g) The former is larger by a factor of e^2 (about 2.71828^2 or 7.39).

(h) C, B, A.

Remember that a strong L1 regularizer prefers all weights to be small, and furthermore tends to produce sparse weight vectors, where most weights are 0.

Notice that under scheme A, such weight vectors will pick out a few particular lengths as having small nonzero weights. Under scheme B, such weight vectors will divide the lengths into a few ranges, with equal weight within each range, and small differences in weight from one range to the next. And under scheme C, sparse weight vectors will give a piecewise linear graph, with small differences in slope from one range to the next.

These graphs therefore correspond to schemes C, B, A respectively. In fact, each of them corresponds to the same sparse weight vector (0,0,1.5,0,-2,0,0,0) under a different scheme.