Homework 2: HMM, Viterbi, CRF/Perceptron

CS 585, UMass Amherst, Fall 2015

Version: Oct5

Overview

Due **Tuesday, Oct 13 at midnight**.

Get starter code from the course website's schedule page. You should submit a zipped directory named hw2_YOUR-USERNAME that contains:

- code
- writeup
- a plain text file called "feedback.txt": an estimate of the number of hours this assignment took you to complete, and any additional feedback.

Please don't include the sentiment documents data in the zip file (just because it's large).

Our course's collaboration policy is specified on the website.

1 HMM

[15 total points]

Question 1.1. *[3 points]* Assume we have an HMM called $\mathcal{M}$ and a sequence of n observations called **O** that were generated from $\mathcal{M}$. Does the sequence of observations **O** or $\mathbf{O} + o_{n+1}$ (i.e., the same sequence with one additional observation o_{n+1}) have a higher probability under $\mathcal{M}$? If not enough information is given, explain what extra information is required.

Question 1.2. *[12 points]* Answer the following questions using the transition matrix T and emission probabilities E below. Below, Δ and $\Box$ are two output variables, A and B are two hidden states; s_n refers to the n^{th} hidden state in the sequence and o_n refers to the n^{th} observation.

$T =$

	A	B	END
START	0.5	0.5	0.0
A	0.2	0.3	0.5
B	0.4	0.4	0.2

$E =$

	Δ	$\Box$
A	0.5	0.5
B	0.3	0.7

1. *[1 points]* Is $P(o_2 = \Delta | s_1 = B) = P(o_2 = \Delta | o_1 = \Box)$?

2. *[1 points]* Is $P(s_2 = B | s_1 = A) = P(s_2 = B | s_1 = A, o_1 = \Delta)$?

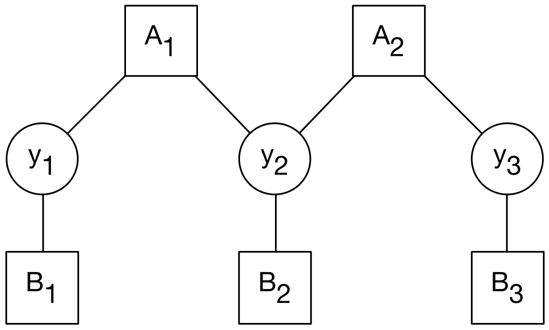
3. *[2 points]* Is $P(o_2 = \Delta | s_1 = A) = P(o_2 = \Box | s_1 = A, s_3 = A)$?

4. *[3 points]* Compute the probability of observing $\Box$ as the first emission of a sequence generated by an HMM with transition matrix T and emission probabilities E.

5. *[5 points]* Compute the probability of the first state being A given that the last token in an observed sequence of length 2 was the token Δ.

2 Viterbi (of the additive loggy variety)

[20 total points]



We will use factor graph notation, for the factor-variable bipartite graph above. A_1 is the "transition" factor that has preferences for the two neighboring variables y_1 and y_2; and A_2 is the preferences over the pair (y_2, y_3), etc.. We write the goodness factor for position 1 to be $A_1(y_1, y_2)$, for position 2 to be $A_2(y_2, y_3)$, as depicted in the diagram. so there's no A_T. But the transition preference function will be the same at all position so we can just write $A(y_1, y_2)$, $A(y_2, y_3)$, etc. B_t is the "emission" factor that has preferences for the variable y_t. As a goodness function it is e.g. $B_1(y_1)$, $B_2(y_2)$, etc.

Let $\vec{y} = (y_1, y_2, ..., y_T)$, the tag sequence for a T length sentence. The total goodness function for a solution $\vec{y}$ is

$$G(\vec{y}) = \sum_{t=1}^{T} B_t(y_t) + \sum_{t=2}^{T} A(y_{t-1}, y_t)$$

Question 2.1. *[2 points]* Define A and B_t in terms of the HMM model, such that G is the same thing as $\log p(\vec{y}, \vec{w})$ under the HMM.

Note: please work from the definition of the HMM in Jurafsky and Martin 3rd ed (chapter 8). They use a START state before the first y_1 ... unlike our lecture slides, which glossed over that part. You will NOT use start or stop within the $(y_1 ... y_T)$ sequence; instead, conditional probabilities involving them will have to be built in to the emission factors. (JM are inconsistent about whether a STOP state has to be generated. You can do it or ignore it, your choice.)

Question 2.2. *[18 points]* Implement additive log-space Viterbi in "vit.py", by completing the *viterbi()* function. It takes in tables that represent the A and B functions as input. We give you an implementation of $G()$ you can check to make sure you understand the data structures, and also the exhaustive decoding algorithm too. Feel free to add debugging print statements as needed. The main code runs the exercise example by default.

When debugging, you should make new A and B examples that are very simple. This will test different code paths. Also you can try the *randomized_test()* from the starter code (posted to website's starter code on Oct 4).

Look out for negative indexes as a bug. In python, if you use an index that's too high to be in the list, it throws an error. But it will silently accept a negative index ... it interprets that as indexing from the right.

3 Averaged Perceptron

[5 total points]

We will be using the following definition of the perceptron, which is the multiclass or structured version of the perceptron. The training set is a bunch of input-output pairs (x_i, y_i). (For classification, y_i is a label, but for tagging, y_i is a sequence).

- For 10 or so iterations, iterate through each (x_i, y_i) pair in the dataset, and for each,
  - Predict $y^* := \arg\max_{y'} \theta^\mathsf{T} f(x_i, y')$
  - If $y_i \neq y^*$: then update $\theta := \theta^{(old)} + rg$

where r is a fixed step size (e.g. $r = 1$) and g is the "gradient" vector, meaning a vector that will get added into θ for the update, specifically

$$g = \underbrace{f(x_i, y_i)}_{\text{feats of true output}} - \underbrace{f(x_i, y^*)}_{\text{feats of predicted output}}$$

Both in theory and in practice, the predictive accuracy of a model trained by the structured perceptron will be better if we use the average value of θ over the course of training, rather than the final value of θ. This is because θ wanders around and doesnt converge (typically), because it overfits to whatever data it saw most recently. After seeing t training examples, define the *averaged parameter vector* as

$$\bar{\theta}_t = \frac{1}{t} \sum_{t'=1}^{t} \theta_{t'} \tag{1}$$

where $\theta_{t'}$ is the weight vector after t' updates. (We are counting t by the number of training examples, not passes through the data. So if you had 1000 examples and made 10 passes through the data in order, the final time you see the final example is $t = 10000$.) For training, you still use the current θ parameter for predictions. But at the very end, you return the $\bar{\theta}$, not θ, as your final model parameters to use on test data.

Directly implementing Equation 1 would be really slow. So here's a better algorithm. This is the same as in Hal Daume's CIML chapter on perceptrons, but adapted for the structured case (as opposed to Daume's algorithm, which assumes binary output). Define g_t to be the update vector g as described earlier. The perceptron update can be written

$$\theta_t = \theta_{t-1} + r g_t$$

Thus the averaged perceptron algorithm is, using a new "weightsums" vector S,

- Initialize $t = 1, \theta_0 = \vec{0}, S_0 = \vec{0}$

- For each example i (iterating multiples times through dataset),
  - Predict $y^* = \arg\max_{y'} \theta^\mathsf{T} f(x_i, y')$
  - Let $g_t = f(x_i, y_i) - f(x_i, y^*)$
  - Update $\theta_t = \theta_{t-1} + r g_t$
  - Update $S_t = S_{t-1} + (t-1) r g_t$
  - $t := t + 1$
- Calculate $\bar{\theta}$ based on S

In an actual implementation, you don't keep old versions of S or θ around ... above we're using the t subscripts above just to make the mathematical analysis clearer.

Our proposed algorithm computes $\bar{\theta}_t$ as

$$\bar{\theta}_t = \theta_t - \frac{1}{t} S_t \tag{2}$$

For the following problems, feel free to set $r = 1$ just to simplify them.

Question 3.1. *[1 points]* What is the computational advantage of computing $\bar{\theta}$ using Equation 2 instead of directly implementing Equation 1?

Now we'll show this works, at least for early iterations.

Question 3.2. *[1 points]* What are $\bar{\theta}_1$, $\bar{\theta}_2$, $\bar{\theta}_3$, and $\bar{\theta}_4$? Please derive them from the Equation 1 definition, and state them in terms of g_1, g_2, g_3, and/or g_4.

Question 3.3. *[1 points]* What are S_1, S_2, S_3, and S_4? Please state them in terms of g_1, g_2, g_3, and/or g_4.

Question 3.4. *[2 points]* Show that Equation 2 correctly computes $\bar{\theta}_3$ and $\bar{\theta}_4$.

Question 3.5. *[2 Extra Credit points]* Use proof by induction to show that this algorithm correctly computes $\bar{\theta}_t$ for any t.

4 Classifier Perceptron

[20 total points]

Implement the averaged perceptron for document classification, using the same sentiment analysis dataset as you used for HW1. On the first two questions, we're asking you to develop using only a subset of the data, since that makes debugging easier. On the third question, you'll run on the full dataset, and you should be able to achieve a higher accuracy compared to your previous Naive Bayes implementation. Starter code is provided in *classperc.py*.

Question 4.1. *[8 points]* Implement the simple, non-averaged perceptron. Run your code on **the first 1000 training instances** for 10 passes through the training data. For each pass, report **the training and test set accuracies**.

Question 4.2. *[8 points]* Implement the averaged perceptron. Run your code on **the first 1000 training instances** for 10 passes through the training data. For each pass, compute the $\bar{\theta}$ so far, and report its **test set accuracy**.

Question 4.3. *[4 points]* Graph four curves on the same plot, using the **full dataset**:

- accuracy of the vanilla perceptron on the training set
- accuracy of the vanilla perceptron on the test set
- accuracy of the averaged perceptron on the test set
- accuracy of your Naive Bayes classifier from HW1 (you don't need to re-run it; just take the best accuracy from your previous results).

The x-axis of the plot should show the pass number through the training set and the y-axis should show the accuracy of the classifier. For this part of the HW run your code on **the entire dataset (all instances)**. Since Naive Bayes doesn't require multiple passes through the data just produce a single horizontal line showing its overall accuracy. Make sure your plot has a title, a label on the x-axis, a label on the y-axis and a legend showing which line is which. Explain verbally what's happening in this plot.

Question 4.4. *[2 Extra Credit points]* You may notice that the training code includes a line that randomizes the order of the training examples (i.e., `random.shuffle(examples)`). The reason we need this line is that our loading code loads in all of the positive examples followed by all of our negative examples. Comment it out and see what happens to the accuracy of your classifiers. What is going on? Explain why.

5 Structured Perceptron with Viterbi

[40 total points]

In this problem, you will implement a part-of-speech tagger for Twitter, using the structured perceptron algorithm. Your system will be not too far off from state of the art performance, coding it all up yourself from scratch!

The dataset comes from `http://www.ark.cs.cmu.edu/TweetNLP/` and is described in the papers listed there (Gimpel et al. 2011 and Owoputi et al. 2013). The Gimpel article describes the tagset; the annotation guidelines on that webpage describe it futher.

Your structured perceptron will use your Viterbi implementation as a subroutine. If that's buggy, this will cause many problems here—your perceptron will have really weird behavior. (This happened to us when designing your assignment!) If you have problems, try using the greedy decoding algorithm, which we provide in the starter code. Make sure to note which decoding algorithm you're using in your writeup.

The starter code is *structperc.py* and it assumes the two data files *oct27.train* and *oct27.dev* are in the same directory. (For simplicity we're just going to use this "dev" set as our test set.)

Question 5.1. *[2 points]* First let's do a little data analysis to establish the "most common tag" baseline accuracy. Using a small script or ipython notebook, load up the dev dataset (oct27.dev) using the function *structperc.read_tagging_file* (from *import structperc*). (Make sure to include a copy of this code or notebook in your submisssion.) Calculate the following: What is the most common tag, and what would your accuracy be if you predicted it for all tags?

The structured perceptron algorithm works very similarly as the classification version you did in the previous question, except the prediction function uses Viterbi as a subroutine, which has to call feature extraction functions for local emissions and transition factors. There also has to be a large overall feature extraction function for an entire structure at once.[1] The following parts will build up these pieces. First, we will focus on inference, not learning.

Question 5.2. *[2 points]* We provide a barebones version of *local_emission_features*, which calculates the local features for a particular tag at a token position. You can run this function all by itself. Make up an example sentence, and call this function with it, giving it a particular index and candidate tag. Show the code for the function call you made and the function's return value, and explain what the features mean (just a sentence or two).

Question 5.3. *[2 points]* Implement *features_for_seq()*, which extracts the full feature vector $f(x, y)$, where x is a sentence and y is an entire tagging sequence for that sentence. This will add up the feature vectors from each local emissions features for every position, as well as transition features for every position (there are $N - 1$ of them, of course). Show the output on a very short example sentence and example proposed tagging, that's only 2 or 3 words long.

To define $f(x, y)$ a little more precisely: If $f^{(B)}(t, x, y)$ means the local emissions feature vector at position t (i.e. the *local_emission_features* function), and $f^{(A)}(y_{t-1}, y_t, y)$ is the transition feature function for positions $(t-1, t)$ (which just returns a feature vector where everything is zero, except a single element is 1), then the full sequence feature vector will be the vector-sum of all those feature vectors:

$$f(x, y) = \sum_{t}^{T} f^{(B)}(t, x, y) + \sum_{t=2}^{T} f^{(A)}(y_{t-1}, y_t)$$

You implemented $f^{(B)}$ above. You probably don't need to bother implementing $f^{(A)}$ as a standalone function. You will have to decide on a particular convention to encode the name of a transition feature. For example, one way to do it is with string concatenation like this, `"trans_%s_%s" % (prevtag, curtag)`, where prevtag and curtag are strings. Or you could use a python tuple of strings, which works since tuples have the ability to be keys in a python dictionary.

In other words: the emissions and transition features will all be in the same vector, just as keys in the dictionary that represents the feature vector. The transition features are going to be the count of how many times a particular transition (tag bigram) happened. The emissions features are going to be the vector-sum of all the local emission features, as calculated from *local_emission_features*.

Question 5.4. *[2 points]* Look at the starter code for *calc_factor_scores*, which calculates the A and B score functions that are going to be passed in to your Viterbi implementation, in order to do a prediction. The only function it will need to call is *local_emission_features*. It should NOT call *features_for_seq*. Why not?

Question 5.5. *[4 points]* Implement *calc_factor_scores*. Make up a simple example (2 or 3 words long), with a simple model with at least some nonzero features (you might want to use a *defaultdict(float)*, so you don't have to fill up a dict with dummy values for all possible transitions), and show your call to this function and the output.

Question 5.6. *[2 points]* Implement *predict_seq()*, which predicts the tags for an input sentence, given a model. It will have to calculate the factor scores, then call Viterbi as a subroutine, then return the best sequence prediction. If your Viterbi implementation does not seem to be working, use the implementation of the greedy decoding algorithm that we provide (it uses the same inputs as *vit.viterbi()*).

[1] If we were clever with function or OO abstractions it's actually possible to share code for this... but in practice that's too hard, so please just make a new implementation in *structperc.py*.

OK, you're done with the inference part. Time to put it all together into the parameter learning algorithm and see it go.

Question 5.7. *[10 points]* Implement *train()*, which does structured perceptron training with the averaged perceptron algorithm. You should train on oct27.train, and evaluate on oct27.dev. You will want to first get it working without averaging, then add averaging to it. Run it for 10 iterations, and print the devset accuracy at each training iteration. Note that we provide evaluation code, which assumes *predict_seq()* and everything it depends on is working properly.

For us, here's the performance we get at the first and last iterations, using the features in the starter code (just the bias term and the current word feature, without case normalization).

```
Training iteration 0
DEV RAW EVAL: 2556/4823 = 0.5300 accuracy
DEV AVG EVAL: 2986/4823 = 0.6191 accuracy
...
Training iteration 9
DEV RAW EVAL: 3232/4823 = 0.6701 accuracy
DEV AVG EVAL: 3341/4823 = 0.6927 accuracy
Learned weights for 24361 features from 1000 examples
```

Question 5.8. *[4 points]* Print out a report of the accuracy rate for each tag in the development set. We provided a function to do this (*fancy_eval*). Look at the two sentences in the dev data, and in your writeup show and compare the gold-standard tags versus your model's predictions for them. Consult the tagset description to understand what's going on. What types of things does your tagger get right and wrong?

To look at the examples, you may find it convenient to use *show_predictions* (or write up the equivalent manually). For example, after 1 iteration of training, we get this output from the first sentence in the devset. (After investigating TV shows that were popular in 2011 when the tweet was authored, we actually think some of the gold-standard tags in this example might be wrong.)

```
word                gold pred
----                ---- ----
@ciaranyree         @    @
it                  O    O
was                 V    V
on                  P    P
football            N    ^      *** Error
wives               N    N
,                   ,    ,
one                 $    $
of                  P    P
the                 D    D
players             N    N
and                 &    &
his                 D    D
wife                N    N
own                 V    V
smash               ^    D      *** Error
burger              ^    N      *** Error
```

To do this part, you may find it useful to either use ipython notebook, or else to save your model's weights with pickle.dumps (or json.dumps) and have a short analysis script that loads the model and devdata to do the reports. If you have to re-train each time you tweak your analysis code, it can be annoying.

Question 5.9. *[10 points]* Improve the features of your tagger to improve accuracy on the development set. This will only require changes to *local_emission_features*. Implement at least 4 new types of features. Report your tagger's accuracy with these improvements. Please make a table that reports accuracy from adding different features. The first row should be the basic system, and the last row should be the fanciest system. Rows in between should report different combinations of features. One simple way to do this is, if you have 4 different feature types, to run 4 experiments where in each one, you add only one feature type to the basic system. For example:

System	Acc
basic	0.6927
basic plus first new feature	..number..
basic plus second new feature	..number..
basic plus third new feature	..number..
basic plus fourth new feature	..number..
basic plus all 1st+2nd+3rd+4th features	..number..

Hint: if you make features about the first character of a word, that helps a lot for the "#" (hashtag) and "@" (at-mention) tags. The URL tag is easy to get too with a similar form of character affix analysis. Character affixes help lots of other tags too. Also, if you have a feature that looks at the word at position t, you can make new versions of it that look to the left or right of the t position in question: for example, "word_to_left=the".

Question 5.10. *[2 points]* For future work, what types of features would you like to try in order to make the tagger better? Please suggest one type of feature that you didn't try, and explain why you think it would help.