Homework 4, Part B: Structured perceptron

CS 585, UMass Amherst, Fall 2016

Overview

Due **Friday, Oct 28**.

Get starter code/data from the course website's schedule page. You should submit a zipped directory named hw4b_YOUR-USERNAME that contains:

- your vit.py and structperc.py files
- writeup

Please don't include data in the zip file. Our course's collaboration policy is specified on the website.

1 Perceptron and Averaged Training

[10 total points]

We will be using the following definition of the perceptron (which applies to 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). The training algorithm is as follows:

- For T 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 1.1. *[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 1.2. *[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 1.3. *[4 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 1.4. *[4 points]* Show that Equation 2 correctly computes $\bar{\theta}_3$ and $\bar{\theta}_4$.

Question 1.5. *[OPTIONAL: 2 Extra Credit points]* Use proof by induction to show that this algorithm correctly computes $\bar{\theta}_t$ for any t.

2 Classifier Perceptron

[15 total points]

The Twitter account, @realDonaldTrump, often shows two types of tweets: those written by Donald Trump, and those written by Trump's political staff.

A programmer decides to use the perceptron algorithm (without averaging) to automatically place Donald Trump's tweets into one of two categories: those written by the candidate and those written by his political staff. (They want to build a Twitter bot that automatically classifies tweets in the future).

The programmer creates a dataset of labeled tweets based on human guesses about authorship. They try to use that dataset to train a perceptron classifier using the code below. But there is a bug, so the classifier does not seem to be working properly. Find the bug, specify what is wrong and explain what would happen if the programmer runs the code without fixing the bug. (Sidenote: see `http://varianceexplained.org/r/trump-tweets/` for more on this.)

Code on next page:

```
import random

def train_perceptron(labeled_tweets):
    '''
    train a perceptron to distinguish between candidate tweets and staff tweets

    labeled_tweets is a list of tuples, each of the form:
    ("candidate", "The media is going crazy. They totally distort
                so many things on purpose. Crimea, nuclear, "the baby
                and so much more. Very dishonest!")
    ("staff", "Thank you Windham, New Hampshire! #TrumpPence16 #MAGA")
    '''

    weights = defaultdict(float)
    t = 0
    stepsize = .5

    iters = 10000
    for pass_iteration in range(iters):
        print "Training iteration %d" % pass_iteration
        random.shuffle(labeled_tweets)
        for goldlabel, tweet in labeled_tweets:
            t += 1
            # get_features is not shown.
            # It returns a dict representing the nonzero entries of a feature vector.
            candidate_feat_vec = get_features(tweet, "candidate")
            staff_feat_vec = get_features(tweet, "staff")

            # predict if a tweet is a "staff" or "candidate" tweet
            predlabel = predict(tweet, weights, candidate_feat_vec, staff_feat_vec)

            if predlabel == "candidate"
                predfeats = candidate_feat_vec
            else:
                predfeats = staff_feat_vec

            if goldlabel == "candidate"
                goldfeats = candidate_feat_vec
            else:
                goldfeats = staff_feat_vec

            diffs = defaultdict(float)
            diffs.update(goldfeats)
            for k in predfeats:
                diffs[k] += predfeats[k]

            for feat_name, feat_value in diffs.iteritems():
                weights[feat_name] += stepsize * feat_value
```

3 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 from HW 4(A) 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 3.1. *[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 (from HW 4(A)) 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 3.2. *[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. In your write up, 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 3.3. *[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

[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*.

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 3.4. *[4 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 from 4(A), 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 3.5. *[6 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 3.6. *[4 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 from HW 4(A) 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()*).

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

Question 3.7. *[14 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 3.8. *[6 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 3.9. *[OPTIONAL: 4 Extra Credit 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..

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".