OREGON STATE UNIVERSITY

CS534: Machine Learning
Homework 3

Shangjia Dong

December 20, 2017

0. VITERBI DECODER AND MAXIMUM-LIKELIHOOD ESTIMATION FOR HMMS

To simply your work, I have implemented for you both the Viterbi decoder (dynamic programming) and the maximum likelihood estimation for HMMs. You need to understand every single line of my tagger.py. Run python tagger.py train.txt.lower.unk dev.txt.lower.unk, and you should get:
train_err 2.73%
dev_err 5.26%
Note: in this HW, for each word w, only consider the tags that associate with it in the training set.
Response: Using the code provided by Dr. Huang, I obtained following results:
train_err 2.73%
dev_err 5.26%

1 STRUCTURED PERCEPTRON

You should implement the structured perceptron using an HMM Viterbi decoder (see slides).
1. First just use two feature templates: ht, t0 iand ht, wi.
Training unaveraged perceptron for 5 epochs. Which epoch gives the best error rates on dev?
Hint: at the end of each epoch, your code should output the weights only if it achieves a new highest accuracy on dev (so that at the end of training this file has the best weights).
You should also output logging info which would output something like this:

epoch 1, updates 102, features 291, train_err 3.90%, dev_err 9.36%
Note: trainer.py should import tagger and reuse the Viterbi decoder and the dictionary.
Answer:
epoch 1, updates 102, |w| = 291, train_err 4%, dev_err 9.36%
epoch 2, updates 91, |w| = 334, train_err 3%, dev_err 8.19%
epoch 3, updates 78, |w| = 347, train_err 3%, dev_err **5.85%**
epoch 4, updates 81, |w| = 368, train_err 3%, dev_err 6.73%
epoch 5, updates 78, |w| = 378, train_err 3%, dev_err 6.14%

For 5 epochs, the best dev err I achieved on perceptron is 5.85%. For 10 epochs, the best dev err I achieved is 5.85% as well.

2. Now implement the averaged perceptron. What is the new best error rate on dev?
Answer:
epoch 1, updates 102, |w| = 291, train_err 4%, dev_err 6.14%
epoch 2, updates 91, |w| = 334, train_err 3%, dev_err **4.97%**
epoch 3, updates 78, |w| = 347, train_err 3%, dev_err 4.97%
epoch 4, updates 81, |w| = 368, train_err 3%, dev_err 5.85%
epoch 5, updates 78, |w| = 378, train_err 3%, dev_err 5.56%

The best dev error achieve on average perceptron is 4.97%.
3. How much slower is averaged perceptron? Compare the time between unaveraged and averaged version.
Answer:
The perceptron took 0.408 seconds, while the average perceptron took 0.334 seconds.
4. Plot a single plot that includes four curves: unaveraged, averaged x train err, dev err. What did you find from this plot?
Answer:
The train error of both algorithm is very similar. However, the dev error are very different. In addition, the average perceptron achieves better results.

5. Explain why we replaced all one-count and zero-count words by <unk>.
Answer:
Because there are cases that a lot of words will only be appearing in the dataset once, counting them as a normal word-tag pair would significantly increase the data sparsity. To reduce it, we replace the one-count and zero-count words as <unk>.

2 Feature Engineering (based on averaged perceptron)

Now you can play with your (averaged) perceptron to include as many feature templates as you want.

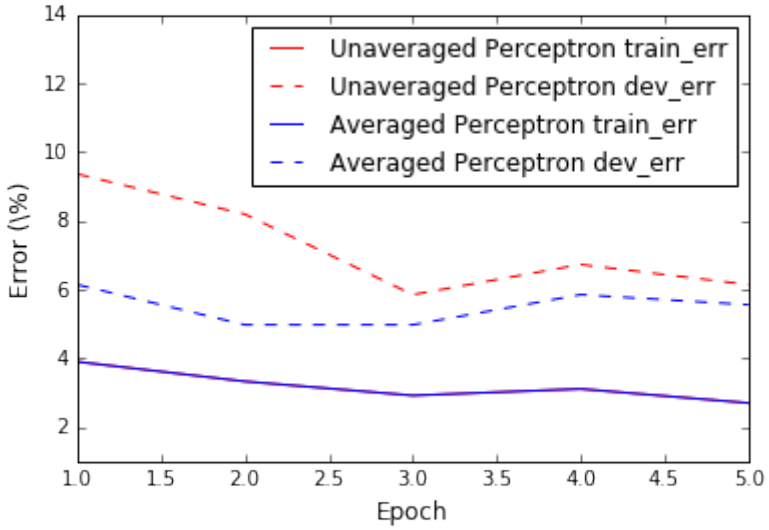

Figure 1.1: Error comparison

Report what templates you tried, which helped and which did not; your best set of templates, and the best accuracies on dev and test you get.
Hint: to simplify your experiments, you might want to define the feature templates in a file and let your code automatically figure out those templates on the fly, i.e., you can define t_i(i = -2, -1, 0) to be a tag in the local window with t0 being the tag for the current word, and similarly w_i(i = -2, -1, 0, 1, 2) to be a word with w0 being the current word. So for example you can define the following two templates:
t0_t-1
t0_w0
which corresponds to the model in Section 1, and you can add
t0_t-1_t-2
t0_w-1_w+1
...
and so on, so that your trainer.py does not need to be changed (first make sure your Viterbi is extended to handle trigrams).
Make sure each template is smoothed (i.e., all back-off versions are included: e.g., if you include t0_t-1_w0, then also include its three backoff versions: t0, t0_t-1, and t0_w0) and don't use overcomplicated templates since they will be too sparse. Also note that using more t_i's slows down dynamic programming, while using more w_i's doesn't (since they are input and static; recall that discriminative models do not model input and do not impose independence assumptions on it). However, the sparsity of the latter increases the dimensionality of feature vector very quickly, which also slows down the whole thing. In general, you should avoid word trigrams (like w0_w1_w2). Also, every template should include t0 (why?).
Include your the best accuracy you achieved on the dev set, and label the test file using that model. Include both dev.lower.unk.best and test.lower.unk.best (the resulting word/tag se-

quences).

Answer:

I have tried four different feature map. Unfortunately I cannot make the "smart" way to update the template every time, so I did each feature map individually.

First, I tried feature map **(t-1 t0 w0)**, the results is as follows:

```
epoch 1, updates 191, |w| = 732, train_err  8%, dev_err 10.53%
epoch 2, updates 216, |w| = 954, train_err  8%, dev_err 6.14%
epoch 3, updates 106, |w| = 988, train_err  4%, dev_err 6.14%
epoch 4, updates 100, |w| = 1017, train_err  4%, dev_err 5.85%
epoch 5, updates 96, |w| = 1030, train_err  3%, dev_err 6.43%
epoch 6, updates 83, |w| = 1055, train_err  3%, dev_err 6.43%
epoch 7, updates 77, |w| = 1090, train_err  3%, dev_err 6.43%
epoch 8, updates 70, |w| = 1098, train_err  3%, dev_err 5.26%
epoch 9, updates 59, |w| = 1107, train_err  2%, dev_err 5.56%
epoch 10, updates 57, |w| = 1122, train_err  2%, dev_err 5.85%
epoch 11, updates 53, |w| = 1135, train_err  2%, dev_err 5.56%
epoch 12, updates 50, |w| = 1142, train_err  2%, dev_err 5.56%
epoch 13, updates 48, |w| = 1146, train_err  2%, dev_err 5.26%
epoch 14, updates 48, |w| = 1152, train_err  2%, dev_err 5.26%
epoch 15, updates 45, |w| = 1158, train_err  2%, dev_err 4.97%
epoch 16, updates 41, |w| = 1162, train_err  2%, dev_err 5.56%
epoch 17, updates 34, |w| = 1166, train_err  1%, dev_err 5.85%
epoch 18, updates 34, |w| = 1167, train_err  1%, dev_err 5.85%
epoch 19, updates 32, |w| = 1170, train_err  1%, dev_err 6.14%
epoch 20, updates 32, |w| = 1171, train_err  1%, dev_err 5.56%
running time 0.916999816895 sec
```

The best dev err I got is 4.97%, the running time is 0.917 sec.

Second, I tried feature map **(t0 w0 w1)**, the results is as follows:

```
epoch 1, updates 191, |w| = 85, train_err  8%, dev_err 11.70%
epoch 2, updates 232, |w| = 85, train_err  9%, dev_err 9.36%
epoch 3, updates 171, |w| = 85, train_err  7%, dev_err 7.89%
epoch 4, updates 143, |w| = 85, train_err  5%, dev_err 8.48%
epoch 5, updates 151, |w| = 85, train_err  6%, dev_err 8.77%
epoch 6, updates 150, |w| = 85, train_err  6%, dev_err 11.11%
epoch 7, updates 202, |w| = 85, train_err  7%, dev_err 11.11%
epoch 8, updates 205, |w| = 85, train_err  7%, dev_err 11.70%
epoch 9, updates 213, |w| = 85, train_err  8%, dev_err 9.36%
epoch 10, updates 169, |w| = 85, train_err  7%, dev_err 9.94%
epoch 11, updates 172, |w| = 85, train_err  7%, dev_err 8.19%
epoch 12, updates 167, |w| = 85, train_err  6%, dev_err 7.89%
epoch 13, updates 160, |w| = 85, train_err  6%, dev_err 7.02%
epoch 14, updates 150, |w| = 85, train_err  5%, dev_err 9.06%
epoch 15, updates 169, |w| = 85, train_err  6%, dev_err 8.77%
epoch 16, updates 169, |w| = 85, train_err  6%, dev_err 9.65%
epoch 17, updates 161, |w| = 85, train_err  6%, dev_err 9.06%
epoch 18, updates 145, |w| = 85, train_err  5%, dev_err 9.06%
epoch 19, updates 145, |w| = 85, train_err  5%, dev_err 9.94%
epoch 20, updates 165, |w| = 85, train_err  7%, dev_err 9.65%
running time 0.934000015259  sec
```

The best dev err I got is 7.02%, the running time is 0.934 sec.

Third I tried feature map **(t-2 t-1 t0 w0)**, the results is as follows:

```
epoch 1, updates 191, |w| = 1927, train_err  7%, dev_err 8.48%
epoch 2, updates 146, |w| = 2704, train_err  5%, dev_err 6.43%
epoch 3, updates 107, |w| = 2856, train_err  3%, dev_err 5.85%
epoch 4, updates 74, |w| = 3009, train_err  2%, dev_err 6.43%
epoch 5, updates 77, |w| = 3157, train_err  2%, dev_err 6.14%
epoch 6, updates 79, |w| = 3263, train_err  2%, dev_err 6.73%
epoch 7, updates 73, |w| = 3333, train_err  2%, dev_err 6.14%
epoch 8, updates 62, |w| = 3370, train_err  2%, dev_err 5.56%
epoch 9, updates 58, |w| = 3390, train_err  2%, dev_err 4.97%
epoch 10, updates 50, |w| = 3440, train_err  1%, dev_err 4.97%
epoch 11, updates 50, |w| = 3463, train_err  1%, dev_err 4.97%
epoch 12, updates 48, |w| = 3473, train_err  1%, dev_err 5.85%
epoch 13, updates 42, |w| = 3511, train_err  1%, dev_err 5.85%
epoch 14, updates 44, |w| = 3565, train_err  1%, dev_err 5.85%
epoch 15, updates 45, |w| = 3596, train_err  1%, dev_err 5.26%
epoch 16, updates 38, |w| = 3602, train_err  1%, dev_err 5.26%
epoch 17, updates 30, |w| = 3626, train_err  1%, dev_err 5.26%
epoch 18, updates 30, |w| = 3634, train_err  1%, dev_err 5.26%
epoch 19, updates 27, |w| = 3645, train_err  1%, dev_err 5.26%
epoch 20, updates 30, |w| = 3660, train_err  1%, dev_err 5.56%
running time 5.38199996948  sec
```

The best dev err I got is 4.97%, the running time is 5.382 sec.
In the last, I tried feature map **(t-3 t-2 t-1 t0 w0)**, the kernel crashed:

```
IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.

Current values:
NotebookApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
NotebookApp.rate_limit_window=3.0 (secs)
```

Looking at the results above, when we adding the tag feature, the running time is dramatically increasing. Also, instead of using t-1 t0 w0, using t0 w0 w-1 will lead to the drop of performance, which makes sense. There are so many different combination of words, so have pair of words feature will increase the sparsity of the feature, but not help much on the training because the probability that two words show up as pair again is less than the probability that a tag pair shows up. Therefore, if we add tag feature, this will help training because there are less tag pairs than word pairs. However, adding tag feature will lead to the increase of the running time. So a trade off needs to be made between running efficiency and training error. In summary, feature map (t-1 t0 w0) and (t-2 t-1 t0 w0) achieve the best dev err, but (t-2 t-1 t0 w0) has lowest training error. So I decided to use (t-2 t-1 t0 w0) to create the submitted file.
Acknowledgement: I would like to thank TA for the explanation on the homework at office hour.

3 DEBRIEFING

Please answer these questions in debrief.txt and submit it along with your work. Note: You get 5 points off for not responding to this part. 1. How many hours did you spend on this assignment?
Response: I roughly spend more than 25 hours on this homework.
2. Would you rate it as easy, moderate, or difficult?
Response: I would consider this homework as difficult.
3. Are the lectures too fast, too slow, or just in the right pace?
Response: The lecture was too slow, we could have controlled the time that recaps the old knowledge better.
4. Any other comments (to the course or to the instructors)?
Response: In the next offering, I would change homework 3 as group homework, and homework 1 or 2 as individual homework. Also, if there are more time left for the homework 3, that would be better.
Also, the quiz is a big portion of the grades, it should be better designed.
All in all, I had a great experience taking this class.