CSE 5525 Homework 3: CNNs and Tagging

Wei Xu

1 Structured Perceptron Tagger (15 points)

In the second part of the assignment you will implement the structured perceptron and Viterbi algorithms for part-of-speech tagging. Then you will experiment with your trained models on a small dataset of tweets annotated with parts-of-speech and named entities. These experiments will explore the question of how well part-of-speech taggers perform when applied to domains other than for which they were trained. For example, how well does a Wall Street Journal trained part-of-speech tagger perform when applied on Twitter?

We provide you with starter Python code to help read in the data and evaluate the results of your model's predictions. You are *required* to make use of the provided code. If you really prefer to implement everything from scratch for some reason, please talk to the instructor first. Your submitted code should run on the command line in a unix-like environment (e.g. Linux, OSX, Cygwin).

The experiments required to complete the assignment will take some time to run, so it is highly recommended to start early. We recommend you read through this entire document and run the sample code before getting started.

We have provided an evaluation script (`eval.sh`) to train your models and generate predictions on the test data as follows:

```
> #Trains a part-of-speech tagger on the provided Twitter training set
> bash eval.sh twitter
> #Trains a part-of-speech tagger on the provided penn treebank data
> bash eval.sh ptb
#Trains a part-of-speech tagger on the provided IRC chat data
> bash eval.sh nps
```

Your model's predictions on the test data will be output into the directory, `eval/`. You can check the accuracy of your model on the test data by using the provided scripts `accuracy.py` for POS-tagging and the Perl script `conlleval.pl` for named entity recognition.

When you first run the starter code, the tagger will always predict every word is a noun. You will need to implement the Viterbi algorithm for decoding in addition to parameter updates analogous to the perceptron algorithm in Homework #2.

Word's Most Frequent Tag Baseline (2 points)

Before getting started with implementing the perceptron tagger, write a simple program to implement the following baseline. First count the number of times each word occurs with each tag in the training dataset (`data/twitter_train_universal.txt`). Now generate an output file (using the same format as the training data) that predicts the tag of each word using the following heuristic: simply tag each word in the test dataset (`data/twitter_test_universal.txt`) with the tag that appears most frequently in the training dataset (breaking ties arbitrarily). Tag all the unseen words in the test set as nouns. Report your accuracy on the test data using the provided script like so:

```
> python accuracy.py mft_baseline.out data/twitter_test_universal.txt
```

Viterbi Algorithm (6 points)

Implement the Viterbi Algorithm for a bigram perceptron tagger. The provided code in `Data.py` will read in the provided training data. You should make use of log-scores (unnormalized log probabilities) - each multiplication in Viterbi should be replaced with addition, and unnormalized probabilities are simply dot products of feature vectors and weights. Note: you will need to complete the next part of the assignment before you can test if your implementation is properly working.

The method you will need to implement is `ViterbiTagger.Viterbi`. Before you do this, the classifier always predicts words as nouns.

Structured Perceptron (5 points)

Next, implement the structured perceptron algorithm. For this you will need to modify `Tagger.Train`. Include parameter averaging as in Homework #2.

Report your performance (accuracy) training and testing on the Twitter data.

Cross-Domain Experiments (2 points)

Next, try training your POS tagger in each of the following scenarios and report accuracy:

- Train on the provided penn-treebank data and test on Twitter.
- Train on the provided IRC-chat data and test on Twitter.
- Train on all the data (irc + ptb + twitter) and test on Twitter.

What can you say about the performance of part-of-speech taggers when they are applied on text outside their training domain?

Extra Credit: Named Entity Recognition (2 points)

Train your tagger on the provided named entity recognition dataset `twitter_ner_train.txt` and report precision, recall and F_1 on `twitter_ner_test.txt` using the provided script `conlleval.pl`. Next, add additional features to the tagger specifically for the named entity recognition task, and report performance. Commonly used features for named entity recognition include lists of first and last names.[1] Lists of companies, products, etc... can be scraped from various places on the web, such as Wikipedia.[2]

FAQ

Q: What is theta?
A: Basically, you have a separate weight vector for each pair of tags ("theta" in the starter code), then at each time step, take a dot product between the feature vector and the weight vector associated with the current and previous tag. Then, if you were to sum up all these dot products (i.e., local scores), that would give you a score for the entire tag sequence.These local scores can be used to fill out the Viterbi table, in the same way, we discussed in class,

[1] `http://www.census.gov/topics/population/genealogy/data/1990_census/1990_census_namefiles.html`

[2] `https://en.wikipedia.org/wiki/List_of_companies_of_the_United_States`

but with 2 differences: (1) the emission and transitions factors are combined into a single scoring function and (2) you are adding instead of multiplying (effectively this is in log-space). This way, using features for pairs of tags, also allows you to use more types of features than having separate emission and transition factors.

2 Convolutional Neural Networks for Text Classification (5 points)

For the first part of this assignment, you will implement a convolutional neural network for sentiment classification using the IMDB movie review dataset provided in homework #2. First, copy the starter code from the file `FFNN.py` to a new file `CNN.py`, and use this as a starting point. Create a network containing an embedding layer, a single convolutional layer, followed by a pooling layer that does max-pooling over time, a nonlinearity (e.g, ReLU or Tanh), a linear layer and a softmax. You should make use of unigram, bigram and trigram filters. To do this, you will make use of the following classes in Pytorch (see documentaiton on the Pytorch Website):

- `nn.Conv1d`
- `nn.MaxPool1d`

Your report should clearly explain hyperparameters you used (embedding dimensions, number of filters, etc). You should report performance on the development and test sets (`aclImdb_small`) for both your CNN and feedforward neural network implementations.

Hints: You will most likely need to use the functions `torch.unsqueeze` and `torch.view`. You probably also need to use at least 100-200 convolutional filters of each size in order to get good performance.