COMS W4705, Spring 2015: Problem Set 4

Total points: 145

Analytic Problems (due April 27th at 5pm)

Question 1 (20 points)

Clarissa Linguistica decides to build a log-linear model for language modeling. She has a training sample (x_i, y_i) for $i = 1 \ldots n$, where each x_i is a prefix of a document (e.g., $x_i =$ "Yesterday, George Bush said") and y_i is the next word seen after this prefix (e.g., $y_i =$ "that"). As usual in log-linear models, she defines a function $\mathbf{f}(x, y)$ that maps any x, y pair to a vector in $\mathbb{R}^d$. Given parameter values $\mathbf{v} \in \mathbb{R}^d$, the model defines

$$P(y|x, \mathbf{v}) = \frac{e^{\mathbf{v} \cdot \mathbf{f}(x,y)}}{\sum_{y' \in \mathcal{V}} e^{\mathbf{v} \cdot \mathbf{f}(x,y')}}$$

where $\mathcal{V}$ is the *vocabulary*, i.e., the set of possible words; and $\mathbf{v} \cdot \mathbf{f}(x, y)$ is the inner product between the vectors $\mathbf{v}$ and $\mathbf{f}(x, y)$.

Given the training set, the training procedure returns parameters $\mathbf{v}^* = \arg\max_{\mathbf{v}} L(\mathbf{v})$, where

$$L(\mathbf{v}) = \sum_i \log P(y_i|x_i, \mathbf{v}) - C \sum_k v_k^2$$

and $C > 0$ is some constant.

Clarissa makes the following choice of her first two features in the model:

$$f_1(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and previous word in } x \text{ is } \texttt{the} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and previous word in } x \text{ is } \texttt{the} \\ 0 & \text{otherwise} \end{cases}$$

So $f_1(x, y)$ and $f_2(x, y)$ are *identical features*.

Question (10 points): Show that for any training set, with f_1 and f_2 defined as above, the optimal parameters $\mathbf{v}^*$ satisfy the property that $v_1^* = v_2^*$.

Question (10 points): Now say we define the optimal parameters to be $\mathbf{v}^* = \arg\max_{\mathbf{v}} L(\mathbf{v})$, where

$$L(\mathbf{v}) = \sum_i \log P(y_i|x_i, \mathbf{v}) - C \sum_k |v_k|$$

and $C > 0$ is some constant. (Here $|v_k|$ is the absolute value of the k'th feature.) In this case, does the property $v_1^* = v_2^*$ necessarily hold? If not, what constraints do hold for the values v_1^* and v_2^*?

Question 2 (15 points)

Nathan L. Pedant now decides to build a bigram language model using log-linear models. He gathers a training sample (x_i, y_i) for $i = 1 \ldots n$. Given a vocabulary of words $\mathcal{V}$, each x_i and each y_i is a member of

$\mathcal{V}$. Each (x_i, y_i) pair is a *bigram* extracted from the corpus, where the word y_i is seen following x_i in the corpus.

Nathan's model is similar to Clarissa's, except he chooses the optimal parameters $\mathbf{v}^*$ to be $\arg\max L(\mathbf{v})$ where

$$L(\mathbf{v}) = \sum_i \log P(y_i|x_i, \mathbf{v})$$

The features in his model are of the following form:

$$f_i(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and } x = \texttt{the} \\ 0 & \text{otherwise} \end{cases}$$

i.e., the features track pairs of words. To be more specific, he creates one feature of the form

$$f_i(x, y) = \begin{cases} 1 & \text{if } y = w_2 \text{ and } x = w_1 \\ 0 & \text{otherwise} \end{cases}$$

for every (w_1, w_2) in $\mathcal{V} \times \mathcal{V}$.

Question (15 points): Assume that the training corpus contains all possible bigrams: i.e., for all $w_1, w_2 \in \mathcal{V}$ there is some i such that $x_i = w_1$ and $y_i = w_2$. The optimal parameter estimates $\mathbf{v}^*$ define a probability $P(y = w_2|x = w_1, \mathbf{v}^*)$ for any bigram w_1, w_2. Show that for any w_1, w_2 pair, we have

$$P(y = w_2|x = w_1, \mathbf{v}^*) = \frac{Count(w_1, w_2)}{Count(w_1)}$$

where $Count(w_1, w_2)$ = number of times $(x_i, y_i) = (w_1, w_2)$, and $Count(w_1)$ = number of times $x_i = w_1$.

Question 3

Nathan L. Pedant generates (x, y) pairs as follows. Take $\mathcal{V}$ to be set of possible words (vocabulary), e.g., $\mathcal{V}$ = {`the, cat, dog, happy, ...`}. Take $\mathcal{V}'$ to be the set of all words in $\mathcal{V}$, **plus** the reversed string of each word, e.g., $\mathcal{V}'$ = {`the, eht, cat, tac, dog, god, happy, yppah, ...`}.

For each x, Nathan chooses a word from some vocabulary $\mathcal{V}$. He then does the following:

- With 0.4 probability, he chooses y to be identical to x.
- With 0.3 probability, he chooses y to be the reversed string of x.
- With 0.3 probability, he chooses y to be some string that is neither x nor the reverse of x. In this case he chooses y from the uniform distribution over words in $\mathcal{V}'$ that are neither x nor the reverse of x.

Question (10 points)

Define a log-linear model that can model this distribution $P(y|x)$ perfectly (Note: you may assume that there are no palindromes in the vocabulary, i.e., no words like `eye` which stay the same when reversed.) Your model should make use of as few parameters as possible (we will give you 10 points for a correct model with 2 parameters, 8 points for a correct model with 3 parameters, 5 points for a correct model with more than 3 parameters.)

Question (10 points)

Write an expression for each of the probabilities

$$P(\text{the}|\text{the})$$
$$P(\text{eht}|\text{the})$$
$$P(\text{dog}|\text{the})$$

as a function of the parameters in your model.

Question (10 points)

What value do the parameters in your model take to give the distribution described above?

Programming Problems (due May 4th at 5pm)

Please read the submission instructions, policy and hints on the course webpage.

In this programming assignment, you will train a perceptron model for part-of-speech tagging. For the decoding problem for a global linear model, we are given an input instance $x \in \mathcal{X}$ and a function that generates possible output structures $\text{GEN}(x)$, e.g. x is a sentence $w_1, \ldots, w_n$ and $y \in \text{GEN}(x)$ is a possible tagging $t_1, \ldots, t_n$. Additionally we assume that we have a feature function $f(x, y)$ and a weight vector $v \in R^m$. The decoding problem is defined as

$$y^* = \arg \max_{y \in \text{GEN}(x)} f(x, y) \cdot v$$

Crucially, the feature function f must factor to make the problem tractable. For part-of-speech tagging, we partition a tagging into a sequence of histories, $h_1, \ldots, h_n$ and decompose f into a sum of feature functions over each history and the corresponding output, i.e. $\sum_{i=1}^{n} g(h_i, t_i)$. Our focus will be on bigram tagging, so we define a history as

$$h_i = \langle t_{i-1}, x, i \rangle$$

In this assignment you will experiment with different feature functions and implement the perceptron training algorithm for learning the vector v.

Histories and Feature Vectors

Consider a typical tagged training sentence from the Wall Street Journal corpus

```
There/DET is/VERB no/DET asbestos/NOUN in/ADP our/PRON products/NOUN now/ADV ./.
```

In order to score this sentence it is necessary to generate all the (h_i, t_i) pairs for the sentence. We provide a script to generate the histories seen for the gold tagging.

```
python tagger_history_generator.py GOLD < example.sent
  1 * DET
  2 DET VERB
  3 VERB DET
  4 DET ADV
  ...
```

Each line has three columns, i, t_{i-1},and t_i, which, along with the sentence itself, give all the information needed to compute the feature vector for the gold tagging.

In addition to the gold histories, we also need to compute features for all other possible histories. Computing these features is necessary for tagging new sentences and for training the model. We provide a command to enumerate all the possible histories for an example sentence.

```
python tagger_history_generator.py ENUM < example.sent
```

```
1 * DET
1 * VERB
1 * NOUN
1 * ADJ
...
```

Once we have the histories for an example we can compute features for each history. Much of the art of global linear models is selecting good features to describe the data. Feature functions in NLP typically look like

$$g_{\text{BIGRAM:DET:NOUN}}(\langle t_{i-1}, x, i\rangle, t_i) = \begin{cases} 1 & \text{if } t_{i-1} = \text{DET}, t_i = \text{NOUN} \\ 0 & \text{otherwise} \end{cases}$$

which is a binary feature indicating that the previous tag is DET and the current tag is NOUN. We would typically have a feature like this for all pairs of part-of-speech tags in addition to DET/NOUN. For examples of more features, see questions 4 and 5.

Since most problems in NLP use binary features, a common implementation technique is to store a feature vector (the output of $g(h_i, t_i)$) as a map from strings to $\{0, 1\}$. For instance if the bigram DET/NOUN is seen in the sentence, the set would contain $\{$"BIGRAM:DET:NOUN" $\rightarrow 1\}$. Similarly the weight vector v can be can be implemented as a sparse map. For instance, in Python the weight vector v would be a dictionary mapping features to weights, e.g.

```
v["BIGRAM:DET:NOUN"] = 1.23
```

The dot product in the decoding problem can then implemented as a sparse product between these two maps. For more details about this sparse representation, see the notes on log-linear models.

Decoding

If we have a weight vector v and a set of features, we can use the global linear model on new sentence. This requires solving the following decoding problem:

$$y^* = \arg \max_{y \in \text{GEN}(x)} f(x, y) \cdot v = \arg \max_{t_1, \ldots, t_n \in \text{GEN}(x)} \sum_{i=1}^{n} g(\langle t_{i-1}, i, x\rangle, t_i) \cdot v$$

To simplify your task, we have implemented a decoder for bigram tagging. That is, given the score for each history pair, $g(h_i, t_i) \cdot v$, we will return y^*. However we do not provide any of the scoring code. To use the decoder, you must score each possible history of the sentence.

The interface for scoring histories is to provide a string with columns i, t_{i-1}, t_i and the fourth column being the score $g(h_i, t_i) \cdot v$. As an example consider storing this following information in a file `history.scores`

```
cat history.scores
```

```
1 * DET 23.2
1 * VERB 0.25
1 * NOUN 0.51
1 * ADJ 0.10
...
```

Calling the tagger decoder with the sentence and the scores will return $y*$ represented as a sequence of histories.

```
python tagger_decoder.py HISTORY < history.scores
```

```
1 * DET
2 DET VERB
3 VERB DET
4 DET ADV
...
```

Note: To use the decoder, it is necessary to call this script from within your code. For efficiency reasons, you should spawn each script once and then communicate by piping sentences and history weights to the scripts. You should not need to write out to files or spawn more than a handful of processes per run. For Python, see the `subprocess` module and the example distributed with this problem for a simple interface. If you have trouble running shell commands in your language of choice, please consult the TAs.

Perceptron Training

So far, we have discussed generating histories and decoding. The next step is to use these to train your perceptron tagger. Training the tagger requires a corpus of tagged examples $(x^{(i)}, y^{(i)})$ for $i \in \{1, \ldots, M\}$. The file `tag_train.dat` contains the training sentences, and the files `tag_dev.dat` and `tag_dev.key` contain development sentences and correct results. The sentences are taken from the Wall Street Journal corpus and consists of sentences of length less than 15 words. The tag format is identical to that used in Homework 1.

The perceptron algorithm is defined in detail in the class slides. The algorithm is slightly complicated, but you will find that much of the implementation consists of gluing together the provided scripts. The following gives high-level view of the pieces required for each update

1. Get the gold tag histories using `tagger_history_generator.py GOLD`.
2. Enumerate all possible histories using `tagger_history_generator.py ENUM`.
3. Use your model to assign a weight to each history.
4. Find the histories of the highest-scoring tagging using `tagger_decoder.py`.
5. Update your weights based on the features of the gold and highest-scoring histories.

See the note above for a description of how to call these scripts from your code.

Question 4 (20 points)

In this problem you will experiment with using the bigram tagging decoder with a pre-trained model.

- Consider a very simple model with two types of feature functions for all tags u and v and words r

$$g_{\text{BIGRAM:u:v}}(\langle t_{i-1}, x, i\rangle, t_i) = \begin{cases} 1 & \text{if } t_{i-1} = u \text{ and } t_i = v \\ 0 & \text{otherwise} \end{cases}$$

$$g_{\text{TAG:u:r}}(\langle t_{i-1}, (w_1 \dots w_n), i\rangle, t_i) = \begin{cases} 1 & \text{if } w_i = r \text{ and } t_i = u \\ 0 & \text{otherwise} \end{cases}$$

Consider the example sentence above ("There is no ...") as x, we extract the following feature vector from an example (wrong) history

$$g(\langle \text{DET}, x, 2\rangle, \text{NOUN}) = \{\text{BIGRAM:DET:NOUN} \rightarrow 1, \text{TAG:NOUN:is} \rightarrow 1\}$$

For this problem we provide a pre-trained weight vector v for these features (initialized $v = 0$ and trained for $k = 5$ iterations). The file `tag.model` contains the vector. All lines in the file consist of two columns `FEATURE`, `WEIGHT` indicating $v(\text{FEATURE}) = \text{WEIGHT}$. More specifically.

- Lines of the form `TAG:asbestos:NOUN 0.23` mean the feature `TAG:asbestos:NOUN` representing asbestos tagged with NOUN has weight 0.23.
- Lines of the form `BIGRAM:DET:NOUN 0.45` mean the feature `BIGRAM:DET:NOUN` representing the bigram DET, NOUN has weight 0.45.

- The goal of this problem is to decode with this model. First read `tag.model` into a map from feature strings to weights. Next for each sentence in development data run the following steps

  1. Enumerate all possible histories using `tagger_history_generator.py ENUM`.
  2. Compute the features for each history and use `tag.model` to assign a weight to each history.
  3. Call `tagger_decoder.py HISTORY` and pipe in the weighted histories to compute the highest scoring tagging.

  You can check the performance of your tagger by calling `python eval_tagger.py tag_dev.key tag_dev.out`. Report the performance of your tagger and submit your code.

Question 5 (40 points)

One reason that this model performs poorly is that we do not handle rare words. We could introduce _RARE_ tokens or word classes; however, a major benefit of discriminative models is that we can instead introduce features that look directly at properties of the word.

A good example of this type of feature is word suffixes. Suffixes are very a useful indicator of part-of-speech in English (e.g. "ed", "ing", "er") and so we expect features of this type to help disambiguate unknown words. The new features are, for all suffixes u, tag v, and lengths $j \in \{1, 2, 3\}$

$$g_{\text{SUFF:u:j:v}}(\langle t_{i-1}, (w_1 \dots w_n), i\rangle, t_i) = \begin{cases} 1 & \text{if } u = \text{suffix}(w_i, j) \text{ and } t_i = v \\ 0 & \text{otherwise} \end{cases}$$

where $\text{suffix}(w, j)$ returns the last j letters of w.

- For this question, you should first implement the perceptron algorithm to estimate a weight vector v for the new set of features. Start with $v = 0$ and run $K = 5$ iterations. Follow the steps in the Perceptron Training section. When your code is finished write the final model out to `suffix_tagger.model`.
- After training is complete, add the new suffix featurs to your code from question 4 and then run on the development set. Report the performance of your tagger and the tagged output as well as your code. In addition, write up any observations. Did the new features help? How did they compare to the previous model?

Question 6 (20 Points)

As mentioned above, the benefit of this formulation is the ability to add arbitrary features conditioned on the input sentence x. In the last question we experimented with adding suffix features from i-th word of the sentence. Many other features have been shown to be useful for part-of-speech tagging.

- For this question you should run tagging experiments with custom features. There are many ways to come up with new features. One strategy is to look at the errors your tagger made in the question 5 and try out features that target specific issues. We also encourage you to look up features in the NLP literature or in open-source taggers. Don't be discouraged if adding seemingly useful features sometimes decreases the accuracy of your tagger; feature selection can be a difficult problem.

  When you come up with interesting features to use, run experiments with **three** different feature combinations. Compare the results you saw and write up the issues that you encountered.