Statistical NLP 2018 - Assignment 2

Due October 1 - at 11:59pm. Email code to anhad@nyu.edu and aparikh@cs.nyu.edu

Deliverables: Deliverables:

- Result file in github repository.
- Code zip emailed to instructors.
- Problem Set Questions: If you have good handwriting, writing by hand and scanning is acceptable.
- **Submission Format:** Please title your submission email **Assignment 2 submission**. You should submit a zip file (firstname_lastname_hw2.zip) containing the report in pdf format (firstname_lastname_hw2.pdf) and the zipped code directory. Write your netID and your collaborators' netID (if any) in the report.

Git and Leaderboard

On the course webpage there will be a leaderboard for tracking the prediction performance of the models that you will be building during the assignments. It will only show the (anonymized) best submission so far, not your particular one. This is simply to give you a general idea of the performance that others in the class are getting (Note that only the extra credit portion of the assignment requires you to compete with the scores of others in the class. You are eligible for full, normal credit on the assignment as long as you achieve a certain pre-determined score).

Whenever the leaderboard is updated however, it will send you an email with your test set score. Note that this is done only once every 3 hours for this assignment to reduce overfitting.

We will be using public GitHub repositories to manage the submissions. If you are not familiar with *git*, there are plenty of good tutorials online, for example: `http://www.vogella.com/articles/Git/article.html`.

If you don't already have a GitHub account yet, please create one here: `https://github.com/signup/free`. You are welcome to reuse an existing account for this course.

Now fork the master repository `https://github.com/anhad13/StatisticalNLPFall18`. We only be using the repository for submitting system output (**please do not put your code/data there**), so you will find a skeleton structure for all the assignments.

Please edit `hw0/output.txt` so it contains your name, email address, and a screen name for the course leaderboard and submit it via: `git add hw0/output.txt; git commit -m 'my first check-in'; git push;`

Finally, post your repository to Piazza, so that I can link it to your name. Your repository path should be of the form: `https://github.com/username/stat-nlp-fall2018`. There is a manual step that we need to perform after you post your repository to Piazza, so please give me a couple of hours. After that everything will be automatic.

1 Setup

You will need to download some new code and data from the directory I shared with you previously for Assignment 1. Please download *code-fall2018-a2.zip* and *data2-2018.zip* from that directory.

Make sure you can still compile the entirety of the course code without errors. The Java file to start out inspecting for this project is:

```
nlp/assignments/ProperNameTester.java
```

Try running it with:

```
java nlp.assignments.ProperNameTester
    -path DATA -model baseline -test validation
```

Here, `DATA` is wherever you unzipped the data zip to. If everything's working, you'll get some output about the performance of a baseline proper name classifier being tested.

2 Problem Statement

Proper Name Classification: Proper name classification is the task of taking proper names like *Eastwood Park* and deciding whether they are places, people, etc. In the more general task of named entity recognition (NER), which we do not consider here, one also has to detect the boundaries of the phrases. In general, we might use a variety of cues to make these kinds of decisions, including looking at the syntactic environment that the phrases occur in, whether or not the words inside the phrase occur in lists of known names of various types (gazetteers), and so on. In this assignment, however, you will write classifiers which attempt to classify proper names purely on the basis of their surface strings alone. This approach is more powerful than you might think: for example, there aren't too many people named *Xylex*. Since even the distribution of characters is very distinctive for each of these categories, we will start out trying to make classification decisions on the basis of character n-grams alone (so, for example, the suffix *-x* may indicate drugs while *-wood* may indicate places.

Start out by looking at the main method of `ProperNameTester.java`. It loads training, validation, and test sets, which are lists of `LabeledInstance` objects. These instance objects each represent a proper noun phrase string (such as *Eastwood Park*) gotten using `getInput()` along with a label gotten with `getLabel()`. The labels are one of five label strings: PLACE, MOVIE, DRUG, PERSON, or COMPANY. A classifier is then trained and tested. To start out, we have the default `MostFrequentLabelClassifier`, which always chooses MOVIE since it is the most common label in the training set. In this assignment, you will build better classifiers for this task: a discriminative classifier using a maximum entropy model and another one using the perceptron.

A Maximum Entropy Classifier: You will build a feature-driven maximum entropy classifier for the same task. We'll start with building the classifier itself, which is in

```
assignments/MaximumEntropyClassifier.java
```

Forget about proper name identification for the moment, and look at the main method of this class. It runs a miniature problem which you will use to debug your work as you flesh out the classifier, which currently has some major gaps in its code. In the toy problem, we create several training instances (and one test instance), which are either cats or bears, and which have several features each. These training instances are passed to a `MaximumEntropyClassifier.Factory` which uses them to learn a `MaximumEntropyClassifier`. This classifier is then applied to the test set, and a distribution over labels is printed out.

Part a: (14 points) To start out, the whole classification pipeline runs, but there's no maximum entropy classification involved. You'll have to fill in two chunks of code (marked by `TODO` lines) in order to turn the placeholder code into a maximum entropy classifier. First, look at

```
MaximumEntropyClassifier.getLogProbabilities()
```

This method takes an instance (as an `EncodedDatum`), and produces the (log) distribution, according to the model, over the various possible labels. There will be some interface shock here, because you're looking at a method buried deeply in my implementation of the rest of the classifier. This situation won't be typical for this course, but for here it's necessary to ensure your classifiers are efficient enough for future assignments. In the present method, you are given several arguments, whose classes are defined in this same Java file:

```
EncodedDatum datum
Encoding<F,L> encoding
IndexLinearizer indexLinearizer
double[] weights
```

The `EncodedDatum` represents the input instance represented as a feature vector. It is a sparse encoding which tells you which features were present for that that instance, and with what counts. When you ask an `EncodedDatum` what features are present, it will return feature indexes instead of feature objects - for example it might tell you that feature 121 is present with count 1.0 and feature 3317 is present with count 2.0. If you want to recover the original (`String`) representation of those features, you'll have to go through the `Encoding`, which maps between features and feature indexes. `Encoding`s also manage maps between labels and label indexes. So while your test labels are *cat* and *bear*, the `Encoding` will map these to indexes 0 and 1, and your returned log distribution should be a double array indexed by 0 and 1, rather than a hash on *cat* and *bear*.

Once you've gotten a handle on the encodings, you should flesh out `getLogProbabilities()`. It should return an array of doubles, where the indexes are the label indexes, and the entries are the log probabilities of that label, given the current instance and weights. To do this, you will need to properly combine the model weights (w from lecture). The double vector weights contains these weights linearized into a one-dimensional array. Remember that there is a weight for each predicate, that is, for each pair of feature and label. To find the weight for the feature "fuzzy" on the label *cat*, you'll therefore need to take their individual indexes (2 and 0) and use the `IndexLinearizer` to find out what joint predicate index in `weights` to use for that pair.

Try to do this calculation of log scores as efficiently as possible - this is the inner loop of the classifier training. Indeed, the reason for all this primitive-type array machinery is to minimize the amount of time it'll take to train large maxent classifiers, which would be very time consuming with a friendlier collection-based implementation.

Part b (14 points): Once you've gotten the code set to calculate the log scores, run the mini test again. Now that it's actually voting properly according to the model weights, you won't get a 0.0/1.0 distribution anymore - you'll get 0.5/0.5, because, while it is voting now, the weights are all zero. The next step is to fill in the weight estimation code. Look at

```
Pair<Double, double[]> calculate(double[] w)
```

buried all the way in

```
MaximumEntropyClassifier.ObjectiveFunction
```

This method takes a vector `w`, which is some proposed weight vector, and calculates the (negative) log conditional likelihood of the training labels (y) given the weight vector (w) and examples (x):

$$L(w) = \sum_i logP(y_i|x_i, w)$$

where

$$P(y|x, w) = \frac{e^{w_y \cdot f(x)}}{\sum_{y'} e^{w_{y'} \cdot f(x)}}$$

Your code will have to compute both L and its derivatives:

$$\frac{\partial L}{\partial w_y} = \sum_i I(y_i = y) f(x_i) - \sum_i P(y|x_i, w) f(x_i)$$

Verify these equations for yourself. Recall that the left sum is the total feature count vector over examples with true class y in the training, while the right sum is the expectation of the same quantity, over all examples, using the label distributions the model predicts. To sanity check this expression, you should convince yourself that if the model predictions put mass 1.0 on the true labels, the two quantities will be equal.

The current code just says that the objective is 42 and the derivatives are flat. Note that you don't have to write code that guesses at w - that's the job of the optimization code, which is provided. All you have to do is evaluate proposed weight vectors. In scope are the data, the string-to-index `encoding`, and the `linearizer` from before:

```
EncodedDatum[] data;
Encoding encoding;
IndexLinearizer indexLinearizer;
```

You should now write new code to calculate the objective and its derivatives, and return the Pair of those two quantities. Important point: because you wish to maximize L and your optimizer is a minimizer, you will have to negate the computed values and derivatives in the code.

Run the mini test again. This time, after a few iterations, the optimization should find a good solution, one that puts all or nearly all of the mass onto the correct answer, "cat."

Part c (2 points): Almost done! Remember that predicting probability one on cat is probably the wrong behavior here. To smooth, or regularize, our model, we're going to modify the objective function to penalize large weights. In `calculate()`, you should now add code which adds a penalty to get a new objective:

$$L'(w) = L(w) - \frac{1}{2\sigma^2}||w||_2^2$$

The derivatives then change by the corresponding amounts, as well.

Run the mini test one last time. You should now get less than 1.0 on "cat" (0.73 with the default `sigma`).

Part d (25 points): Now that your classifier works, goodbye mini test! Return to the proper name classification code and invoke it with

```
java nlp.assignments.ProperNameTester
    -path DATA -model maxent -test validation
```

It now trains a maxent classifier using your code. The `ProbabilisticClassifier` interface takes a string name and returns a string label. To actually use a feature-based classifier like our maxent classifier, however, we need to write code which turns the name into a vector of feature values. This code is a `FeatureExtractor`, which is defined in

```
ProperNameTester.ProperNameFeatureExtractor
```

This extractor currently converts each name string into a Counter of String features, one for each character unigram in the name. So "Xylex" will become

```
["X" : 1.0, "y" : 1.0, "l" : 1.0, "e" : 1.0, "x" : 1.0]
```

The classifier should train relatively quickly using these unigram features (should be no more than a few minutes, possibly tens of seconds, and you can reduce the number of iterations for quick tests). It won't work very well,

though perhaps better than you might have thought. You should get an accuracy of 63.7% using the default amount of smoothing (sigma of 1.0) and 40 iterations. This classifier has the same information available as a class-conditional unigram model.

Your final task is to flesh out the feature extraction code by improving the feature extractor. You can take that input name and create any String features you want, such as `BIGRAM-Xy` indicating the presence of the bigram *Xy*. Or `Length<10`, `Length=5`, `WORD=Xylex`, or `FIRST-LETTER-SAME-AS-LAST-LETTER` – whatever you can think of. (If you want bigrams or longer n-grams, you might find `util.BoundedList` useful - it lets you ask for list items without worrying about a list's range.) Any descriptor of an aspect of the input that seems relevant is fair game (though add feature classes gradually so you can judge how much you're slowing down your training). Better indicators should raise the accuracy of the classifier. You should be able to get your classification accuracy over 70% (very easy), and possibly close to or even over 90% (harder).

To be eligible for full credit for this part if you obtain a score of 85% on the test set (You will receive partial credit up to this number).

Part 2 (15 points): With all the code that you had to write so far, it should take just a few extra lines to implement a perceptron trainer. Remember, instead of having to compute the derivative over the entire training set, the perceptron simply picks up each example in sequence, and tries to classify it given the current weight vector. If it gets it right, it simply moves on to the next example, otherwise it updates the weight vector with the difference of the feature counts in the correct example and in the prediction (no need to compute expected counts!).

Extra credit: There are two ways to get extra credit on this assignment. For both parts you are limited to linear models, but are free to experiment with any features, regularization, or learning strategy you wish.

- (5 pts): Be the highest scoring submission in the class.
- (5 pts): Have the most creative idea in the class as determined by the instructors. (If no ideas are particularly creative then no one will get extra credit.)

3 Part 4: Write-Up / Mini-Problem Set (30 points)

(Small) Writeup (5 points):

1. Record the performance of both your top scoring submission as well as the performance of your perceptron (on the development set).
2. Describe the features you added in Part 1(d) to boost the performance.

More Language Modeling (10 points):

Let n be the order of the language model i.e.

$$P(w_i|w_1, ..., w_{i-1}) \approx P(w_i|w_{i-n+1}, ..., w_{i-1}) = P(w_i|w_{i-n+1}^{i-1})$$

For this question, the term *inference time* refers to one call to the language model i.e. computing $P(w_i|w_{i-n+1}^{i-1})$ under the language model for one choice of w_{i-n+1}^{i-1}, w_i.

- Consider the particular neural language model (Mnih and Hinton 2007) below (where $\mathbf{e}_w \in \mathbb{R}^d$ is a trainable vector for the word w, $C_j \in \mathbb{R}^{d \times d}$ is a trainable transformation matrix, and $b_w \in \mathbb{R}$ is a scalar bias.

$$\mathbf{p}_i = \sum_{j=1}^{n-1} \mathbf{e}_{w_{i-j}} C_j$$

$$\nu_i(w) = \mathbf{p}_i^T \mathbf{e}_w + b_w$$

$$P(w_i|w_{i-n+1}^{i-1}) = \frac{\exp(\nu_i(w_i))}{\sum_{v \in \mathcal{V}} \exp(\nu_i(v))}$$

What is the memory and inference complexity of the above neural language model in big-O notation?

Repeated Features in Naive Bayes / Logistic Regression (15 points): Consider a classification problem with 3 binary features x_1, x_2, x_3 where $x_2 = x_3$ (i.e. they are identical features) and a binary class label y. Ideally, the addition of repeated (identical) features should not affect the classification decision. We will study the effect of these repeated features on Naive Bayes and Logistic Regression.

Assume all models below are trained with Maximum Likelihood to convergence.

- Mathematically characterize the effect (or non-effect) of repeated features on Naive Bayes e.g. how A Naive Bayes model trained only on (x_1, x_2) compares to one trained on (x_1, x_2, x_3).
- Mathematically characterize the effect (or non-effect) of repeated features on Logistic Regression (without any regularization) e.g. how a Logistic Regression model trained only on (x_1, x_2) compares to one trained on (x_1, x_2, x_3).
- Does addition of ℓ_2 regularization to logistic regression affect its sensitivity to repeated features? (If so, how?)

Random Advice: When you invoke java, you may want to (a) increase the maximum heap size with the `-mx` option (e.g. `-mx512m`) and (b) run the JVM in server model with the `-server` option. You may (or may not) see substantial speed-ups from both.

Submission Setup: So far, you should have been running your experiments on the validation dataset. For the final submission you should switch to the test set by setting the `-test` command-line flag. I have erased the labels from the test set, so the code will print 0.0 as your accuracy. Don't worry, test set performance is close to dev set performance in this case. You should set the `-verbose` flag and grep all lines starting with "Example" and pipe them to `hw2 output.txt`. When you upload that file to git, the leaderboard will show you the final test set performance. Please limit yourself to a handful of submissions and try to not overtune.

If you are unsure about the format of the output file, you can take a look at this file:

```
https://github.com/slavpetrov/stat-nlp-nyu/blob/master/hw2/output.txt
```

To automatically fetch the baseline output you will first need to configure a remote:

```
git remote add upstream https://github.com/slavpetrov/stat-nlp-nyu.git
```

Then you should be able to fetch the baseline:

```
git fetch upstream; git checkout master; git merge upstream/master
```

Please also email the instructors the code zip and write up with a short description of how you boosted performance beyond the vanilla implementation in Part 1 as well as answering the questions in Part 2 and Part 3.