CS378 Assignment 1: Sentiment Classification

Due date: Thursday, February 7 at 5:00pm CST

Academic Honesty: Reminder that assignments should be completed independently by each student. See the syllabus for more detailed discussion of academic honesty. Limit any discussion of assignments with other students to clarification of the requirements or definitions of the problems, or to understanding the existing code or general course material. Never discuss issues directly relevant to problem solutions. Finally, you may not publish solutions to these assignments or consult solutions that exist in the wild.

Goals The main goal of this assignment is for you to get experience extracting features and training classifiers on text. You'll get a sense of what the standard machine learning workflow looks like (reading in data, training, and testing), how some standard algorithms work, and how the feature design process goes.

Dataset and Code

Please use Python 3.5+ for this project.

Data The dataset you'll be using here is the movie review dataset of Pang et al. (2002). This is a dataset of short movie review snippets (typically sentences) taken from IMDB. This data consists of a label (0 or 1) followed by a tab followed by the sentence, which has already been lowercased. The data has been split into a train, development (dev), and blind test set. On the blind test set, you do not see the correct labels and all sentences are simply labeled as 0. The framework code reads these in for you.

Getting started Download the code and data. Expand the file and change into the directory. To confirm everything is working properly, run:

```
python sentiment_classifier.py --model TRIVIAL --no_run_on_test
```

This loads the data, instantiates a `TrivialSentimentClassifier` that always returns 1 (positive), and evaluates it on the training and dev sets. The reported dev accuracy should be `Accuracy: 538 / 1066 = 0.504690`. Always predicting positive isn't so good!

Framework code The framework code you are given consists of several files. `sentiment_classifier.py` is the main class. **Do not modify this file for your final submission**, though it's okay to add command line arguments during development or do whatever you need. It uses `argparse` to read in several command line arguments. You should generally not need to modify the paths. `--model` and `--feats` control the model specification. This file also contains evaluation code. The main method loads in the data, initializes the feature extractor, trains the model, and evaluates it on train, dev, and blind test, and writes the blind test results to a file.

Data reading is handled in `sentiment_data.py`. This also defines a `SentimentExample` object, which wraps a sequence of words with an integer label (0/1).

`utils.py` implements an `Indexer` class, which maintains a bijective mapping between indices and features (strings).

`models.py` is the primary file you'll be modifying. It defines base classes for the `FeatureExtractor` and the classifiers, and defines `train_nb` and `train_perceptron` methods, which you will be implementing. `train_model` is your entry point which you may or may not need to modify.

Part 1: Unigram Naive Bayes (40 points)

In this part, you should implement a multinomial Naive Bayes classifier over unigrams (i.e., bag-of-words), as discussed in lecture and the textbook. This will require modifying `train_nb`, `UnigramFeatureExtractor`, and `NaiveBayesClassifier`, all in `models.py`. `train_nb` should handle the processing of the training data using the feature extractor. `NaiveBayesClassifier` should take the results of that training procedure (counts/probabilities that you need) and use them to do inference.

Feature extraction First, you will need a way of mapping from sentences (lists of strings) to feature vectors, a process called feature extraction or featurization. A unigram feature vector will be a sparse vector with length equal to the vocabulary size. It is sparse because a sentence generally contains only 20-40 unique words, so only this many positions in the vector will be nonzero. There is no one right way to define unigram features. For example, do you want to throw out low-count words? Do you want to clip counts to 1 or let words repeat?

You can use the provided `Indexer` class in `utils.py` to map from string-valued feature names to indices. Note that later in this assignment when you have other types of features in the mix (e.g., bigrams in Part 3), you can still get away with just using a single `Indexer`: you can encode your features with "magic words" like `Unigram=great` and `Bigram=great|movie`. This is definitely the best strategy for managing complex feature sets.

In terms of implementation, there are two approaches you can take: (1) extract features "on-the-fly" during training and grow your weights/counts as you add features; (2) iterate through all training points and pre-extract features so you know how many there are in advance (optionally: build a feature cache to speed things up for the next pass). Your decisions may be different for Naive Bayes and perceptron.

Feature vectors Since there are a large number of possible features, it is always preferable to represent feature vectors sparsely. That is, if you are using unigram features with a 10,000 word vocabulary, you should not be instantiating a 10,000-dimensional vector for each example. You might find `Counter` from the `collections` package useful for storing sparse vectors like this.

Smoothing As discussed in lecture, you probably want to use some sort of smoothing. Laplace (add-α) smoothing is a good choice to start with.

Analysis

Q1 (25 points) Implement unigram Naive Bayes. Report your model's performance on this data. You must get at least 73% accuracy on the development set and the training and evaluation (the printed time) in less than 10 seconds on a CS lab machine-equivalent computer.

Q2 (5 points) List the 10 words that, under your model, have the highest ratio of $\frac{P(w|+)}{P(w|-)}$ (the most distinctly positive words). List the 10 words with the lowest ratio. What trends do you see?

Q3 (5 points) Compare the training accuracy and development accuracy of the model. What do you see? Explain in 1-3 sentences how this might be happening.

Q4 (5 points) Describe what smoothing you use and give a brief comparison of your method with and without smoothing.

Part 2: Perceptron (35 points)

In this part, you'll additionally implement a perceptron classifier with the same unigram bag-of-words feature set as in the previous part. Implement the perceptron algorithm in `train_perceptron` and `PerceptronClassifier` in `models.py`.

Features You should feel free to implement different methods on the featurizers to handle perceptron feats differently from Naive Bayes features – you don't necessarily need to stay with the same interface.

Weight vector For perceptron training, you'll need to store a weight vector. You can represent this in a few ways. The most efficient way is a fixed-size numpy array.

Q5 (20 points) Implement perceptron. Report your model's performance on the dataset. You must get at least 73% accuracy on the development set and it must run in less than 20 seconds on a CS lab machine-equivalent computer.

Q6 (5 points) List the ten unigram features with the highest positive and highest negative weights in your model. What trends do you see? How is this different from the Naive Bayes case?

Q7 (10 points) Try some different schedules for the step size for perceptron. How do the results change?

Part 3: Features (25 points)

In this part, you'll be implementing a more sophisticated set of features. You should implement two additional feature extractors `BigramFeatureExtractor` and `BetterFeatureExtractor`. Note that features for this can go beyond word n-grams; for example, `FirstWord=X` gives features that tell you what the first word of a sentence is, although this one may not be useful.

Q8 (10 points) Implement and experiment with `BigramFeatureExtractor`. How well does Naive Bayes do with this? How about perceptron?

Q9 (15 points) Experiment with at least one feature modification in `BetterFeatureExtractor`. Try it out with either Naive Bayes or perceptron, whichever makes sense depending on what you are doing. Briefly describe what you did and what performance it gives. Things you might try: other types of n-grams, tf-idf weighting, clipping your word frequencies, discarding rare words, etc. **Your final code here should be whatever works best (even if that's one of your other feature extractors)** This model should train and evaluate in at most 60 seconds.

Deliverables and Submission

Beyond your writeup, your submission will be evaluated on several axes:

1. Execution: your code should train and evaluate within the time limits without crashing
2. Accuracy on the development set of your Unigram Naive Bayes, Unigram Perceptron, and Better Naive Bayes / Perceptron (we will take the higher number)

3. Accuracy on the blind test set (you should evaluate your best model and include this output)

Submission You should submit the following files to Canvas:

1. A PDF or text file of your answers to the questions
2. Blind test set output in a file named `test-blind.output.txt`. The code produces this by default, but make sure you include the right version!
3. `models.py` plus any additional python files that you included. Please submit these as individual file uploads. **Do not modify or upload `sentiment_classifier.py`.** `sentiment_data.py` and `utils.py` are okay to modify but you shouldn't need to.

Make sure that the following commands work before you submit:

```
python sentiment_classifier.py --model NB --feats UNIGRAM

python sentiment_classifier.py --model PERCEPTRON --feats UNIGRAM

python sentiment_classifier.py --model NB --feats BETTER

python sentiment_classifier.py --model PERCEPTRON --feats BETTER
```

These commands should all print dev results and write blind test output to the file by default.

References

Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. 2002. Thumbs up? Sentiment Classification using Machine Learning Techniques. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.