CS378 Assignment 1: Sentiment Classification

Due date: Tuesday, January 31th at 11:59pm CST

Academic Honesty: While you are encouraged to discuss the assignment with other students, **all code you write and your writeup must be your own!** You should not share codes with anyone.

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 learning algorithms work, and how the feature design process goes.

Dataset and Code

Please use Python 3.5+ for this project. You may find numpy useful for storing and manipulating vectors in this project, though it is not strictly required. The easiest way to install numpy is to install anaconda,[1] which includes useful packages for scientific computing and is a handy package manager that will make it easier to install PyTorch for later assignments.

Data You'll be using the movie review dataset of Socher et al. (2013). This is a dataset of movie review snippets taken from the movie review website Rotten Tomatoes. We are tackling a simplified version of this dataset which frequently appears in the literature: positive/negative binary sentiment classification of sentences, with neutral sentences discarded from the dataset.

You first have to download the data and skeleton code from the link.[2] The data files given to you contain of newline-separated sentiment examples, consisting of a label (0 or 1) followed by a tab, followed by the sentence, which has been tokenized but not 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 labels and only the sentences are given to you. 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: 444 / 872 = 0.509174`. 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. It uses `argparse` to read in several command line arguments. It's okay to add command line arguments during development or do whatever you need, but **you cannot modify this file for your final submission**. For the existing arguments, you should not need to modify the paths if you execute within the `a1-distrib` directory. `--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.

[1] `https://docs.anaconda.com/anaconda/install/`

[2] `https://www.cs.utexas.edu/~eunsol/courses/data/a1distrib.tgz`

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

`utils.py` implements an `Indexer` class, which can be used to maintain 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_perceptron` and `train_logistic_regression` methods, which you will be implementing. `train_model` is your entry point which you may modify if needed.

Part 1: Perceptron (45 points)

In this part, you should implement a perceptron classifier with a bag-of-words unigram featurization, as discussed in lecture and the textbook. This will require modifying `train_perceptron`, `UnigramFeatureExtractor`, and `PerceptronClassifier`, all in `models.py`. `train_perceptron` should handle the processing of the training data using the feature extractor. `PerceptronClassifier` should take the results of that training procedure (model weights) 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. There is no single right way to define unigram features. For example, do you want to throw out low-count words? Do you want to lowercase? Do you want to discard stopwords? Do you want the value in the feature vector to be 0/1 for absence or presence of a word, or reflect its count in the given sentence?

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 a good strategy for managing complex feature sets.

There are two approaches you can take to extract features for training: (1) extract features "on-the-fly" during training and grow your weight vector 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).

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, as this is very inefficient. Instead, you want to maintain a list of only the nonzero features and their counts. Our starter code suggests `Counter` from the `collections` as the return type for the `extract_features` method; this class is a convenient map from objects to floats and is useful for storing sparse vectors like this.

Weight vector The most efficient way to store the weight vector is a fixed-size numpy array.

Perceptron and randomness Throughout this course, the examples in our training sets *not necessarily* randomly ordered. **You should make sure to randomly shuffle the data before iterating through it.** Even better, you could do a random shuffle every epoch.

Random seed If you do use randomness, you can either fix the random seed or leave it variable. Fixing the seed (with `random.seed`) can make the behavior you observe consistent, which can make debugging or regression testing easier (e.g., ensuring that code refactoring didn't actually change the results). However, your results are not guaranteed to be exactly the same as in the autograder environment.

Q1 (25 points; AUTOGRADED) Implement unigram perceptron. To receive full credit on this part, you must get at least **74% accuracy** on the development set, and the training and evaluation (the printed time) should take less than **20 seconds** on a CS lab machine-equivalent computer. Note that it's fine to use your learning rate schedules from Q2 to achieve this performance.

Please list the performance you observe in your writeup, but otherwise you do not need to write anything else to answer this question.

Q2 (10 points) Try at least two different "schedules" for the step size for perceptron (having one be the constant schedule is fine). One common one is to decrease the step size by some factor every epoch or few; another is to decrease it like $\frac{1}{t}$. How do the results change?

Q3 (5 points) List the 10 words that have the highest positive weight under your model and the 10 words with the lowest negative weight. What trends do you see?

Q4 (5 points) Compare the training accuracy and development accuracy of the model. What do you see? Explain in 1-3 sentences what is happening here.

Part 2: Logistic Regression (30 points)

In this part, you'll additionally implement a logistic regression classifier with the same unigram bag-of-words feature set as in the previous part. Implement logistic regression training in `train_logistic_regression` and `LogisticRegressionClassifier` in `models.py`.

Q5 (20 points; AUTOGRADED) Implement logistic regression. Report your model's performance on the dataset. You must get at least **77% accuracy** on the development set and it must run in less than **20 seconds** on a CS lab machine-equivalent computer.

Q6 (10 points) Plot (using matplotlib or another tool) the training objective (dataset log likelihood) **and** development accuracy of logistic regression vs. number of training iterations for a couple of different step sizes. What do you observe?

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 your features for this can go beyond word n-grams; for example, you could define a `FirstWord=X` to extract a feature based on what first word of a sentence is, although this one may not be useful.

Q7 (10 points) Implement and experiment with `BigramFeatureExtractor`. Bigram features should be indicators on adjacent pairs of words in the text. Report the performance of your perceptron classifier and of logistic regression with this feature set.

Q8 (8 points; AUTOGRADED) Experiment with at least one feature modification in `BetterFeatureExtractor`. Try it out with either algorithm (though it should work with both). Report the performance it gives. Things you might try: other types of n-grams, tf-idf weighting, clipping your word frequencies, discarding rare words, discarding stopwords, 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. **This feature modification should not just consist of combining unigrams and bigrams.**

Q9 (7 points) Describe your feature modification and give one reason you might be seeing the performance delta (or lack of delta) from this modification.

Deliverables and Submission

You will submit both your code and writeup to Gradescope. These are submitted as **two separate uploads** to Gradescope.

Written Submission You should upload to Gradescope a PDF or text file of your answers to the questions. This can be handwritten and scanned/photographed if that works best for you.

Note that you can submit the written assignment independently of the code. If you are unable to get the code fully working, please write up what you did and answer as many questions as possible, even partially, so we can assign you appropriate partial credit.

Code Submission Your code will be evaluated by our autograder 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 perceptron, unigram logistic regression, and "better" perceptron / logistic regression (we will take the higher number)
3. Accuracy on the blind test set: this is not explicitly reported by the autograder but we may consider it, particularly if it differs greatly from the dev performance (by more than a few percent)

The **only** file you will be submitting is `models.py`, which should be submitted as an individual file upload. If you wish to implement other classes or functions, put them in `models.py` even if this might not be the ideal coding practice. Importing additional utilities is allowed, but the autograder environment may not have every package. We therefore recommend you stick to numpy, PyTorch (for future assignments), and the provided utilities.

Note that at the course staff's discretion, you may receive a higher score than what the autograder gives you. The autograder cutoffs are set to assign reasonable scores to high-performing submissions. If your code doesn't work or achieves very low performance, we will assess what you've done manually and award partial credit appropriately.

Make sure that the following commands work before you submit:

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

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

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

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

References

[Socher et al.2013] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, A. Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*.

Credit: The assignment was original developed by Greg Durrett.