# CS378 Assignment 1: Sentiment Classification

**Due date: Thursday, February 6 at 11:59pm 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 directly discuss details of the 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 standard 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 a handy package manager that will make it easier to install PyTorch for Assignment 2.

**Data** You'll be using the movie review dataset of Socher et al. (2013). This is a dataset of movie review snippets taken from Rotten Tomatoes. The labeled data actually consists of full parse trees with each syntactic phrase of a sentence labeled with sentiment (including the whole sentence). The labels are "fine-grained" sentiment labels ranging from 0 to 4: highly negative, negative, neutral, positive, and highly positive.

We are tackling a simplified version of this task which frequently appears in the literature: positive/negative binary sentiment classification of sentences, with neutral sentences discarded from the dataset. 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. **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

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

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 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 one 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 to clip counts to 1 or count all occurrences of each word?

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.

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).

**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. You might find `Counter` from the `collections` package 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 algorithm** Note that the Stanford Sentiment Treebank examples are *not* 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.

**Q1 (25 points)** Implement unigram perceptron. Report your model's performance on this data. 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.

**Q2 (10 points)** Try at least two different "schedules" for the step size for logistic regression (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)** 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.

**Q8 (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.

**Q8 (10 points)** Implement and experiment with `BigramFeatureExtractor`. Bigram features should be indicators on adjacent pairs of words in the text. What is the performance of your perceptron classifier with this feature set? What is the performance of logistic regression?

**Q9 (15 points)** Experiment with at least one feature modification in `BetterFeatureExtractor`. Try it out with either algorithm. 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, 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.

## 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 perceptron, unigram logistic regression, and "better" perceptron / logistic regression (we will take the higher number)
3. Accuracy on the blind test set (you should run prediction with your best model and include this output)

**Submission** You should submit the following files to Canvas **as three separate file uploads** (not a zip file):

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`, which should be submitted as an individual file upload. **Do not modify or upload `sentiment_classifier.py`, `sentiment_data.py`, or `utils.py`**. Please put all of your code in `models.py`.

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
```

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

## References

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.