# Applied Machine Learning HW4

Due Monday 5/31 @ 11:59pm on Canvas

Instructions:

1. This homework is about sentiment classification, using (averaged) perceptron (and optionally, other ML algorithms such as SVM). I provided a very simple naive perceptron baseline. The point of this homework is more about learning from my code and extending it, rather than implementing yours from scratch.
2. Download HW4 data from the course homepage, which includes training, dev, and semi-blind test files.
3. You should submit a single `.zip` file containing `hw4.pdf`, `test.txt.predicted`, and all your code. Again, LaTeX'ing is recommended but not required. Make sure your `test.txt.predicted` has the same format as `train.txt` and `dev.txt`.

## 0 Sentiment Classification Task and Dataset

Sentiment classification is one of the most widely used applications of machine learning. Basically, given a sentence (e.g., a movie review or a restaurant review), we want to label it as either "positive" or "negative". I have collected a movie review data set which contains 8,000 training, 1,000 dev, and 1,000 test sentences. To make it simple, I made sure that each set is exactly 50% positive and 50% negative.

Some examples from the training set (the labels were manually annotated by paid annotators):

```
+    i admired this work a lot
+    if steven soderbergh 's ` solaris ' is a failure it is a glorious failure
+    it 's refreshing to see a romance this smart
+    i could n't recommend this film more
+    manages to be original , even though it rips off many of its ideas

-    extremely bad
-    the kind of movie you see because the theater has air conditioning
-    the only thing i laughed at were the people who paid to see it
-    the entire movie is about a boring , sad man being boring and sad
-    though catch me if you can is n't badly made , the fun slowly leaks out of the movie
```

To simplify your job I have done the following preprocessing steps:

1. All words are lowercased;
2. Punctuations are split from adjoining words and become separate "words" (e.g., the commas above);
3. Verb contractions and the genitive of nouns are split into their component morphemes, and each morpheme is tagged separately, e.g.:

   ```
   children's -> children 's   parents' -> parents '   won't -> wo n't    that's -> that 's
   gonna -> gon na   i'm -> i 'm  you'll -> you 'll   we'd -> we 'd  you're -> you 're
   ```

4. Single quotes are changed to forward- and backward- quotes (e.g., `'solaris' --> ` solaris '`) and double quotes are changed to double single forward- and backward- quotes, e.g.:

   ```
   "catch me if you can" --> `` catch me if you can ''
   ```

5. There were a small amount of reviews in Spanish; I've removed them.

**Question: why is each of these steps necessary or helpful for machine learning?**

# 1 Naive Perceptron Baseline

I have written a very simple naive perceptron baseline for this task (included in the data):

```
$ python train.py train.txt dev.txt

epoch 1, update 38.8%, dev 36.6%
epoch 2, update 25.1%, dev 34.6%
epoch 3, update 20.7%, dev 33.8%
epoch 4, update 16.7%, dev 31.7%
epoch 5, update 13.8%, dev 34.0%
epoch 6, update 12.1%, dev 31.9%
epoch 7, update 10.3%, dev 30.1%
epoch 8, update 9.2%, dev 30.6%
epoch 9, update 8.4%, dev 31.8%
epoch 10, update 7.0%, dev 31.4%
best dev err 30.1%, |w|=16743, time: 2.3 secs
```

Here the update % is the update ratio on the training set, which approximates training error rate but not exactly the same (see HW1/HW3), and `|w|` is the dimensionality of the weight vector.

This implementation is extremely simple (about 40 lines of code). Basically, each (unique) word is a feature, and to make it very simple, we treat each sentence as a "bag of words", i.e., a "set" of words where the word order does not matter. This is also known as "one-hot" representation because for each word, only the corresponding dimension is on (or 1) and all other dimensions are off (or 0), and a sentence's representation is simply the sum of its word vectors, e.g. the sentence `the man bit the dog` is represented as

```
....  bit ... dog ... man ... the ...
....   1  ...  1  ...  1  ...  2  ...
```

To represent this space I use my very simple sparse vector library `svector.py` (also included), which provides an `svector` class based on Python's built-in default dictionary (`collections.defaultdict`). Unlike numpy vectors which has a fixed dimensions, my `svector` does not assume any dimensionality, and each unique key is a dimension. This made it perfectly suitable for natural language tasks where the vocabulary size is huge, e.g., in the training file there are 16,743 unique words (including punctuatons). I made `svector` very similar to numpy vector, so that you can add, subtract, and dot-product with sparse vectors:

```
>>> from svector import svector
>>> a = svector()
>>> a
defaultdict(<type 'float'>, {})
>>> a['the'] = 1
>>> a
defaultdict(<type 'float'>, {'the': 1})
>>> a += a
>>> a
defaultdict(<type 'float'>, {'the': 2})
>>> a.dot(a)
4
>>> a * 2
defaultdict(<type 'float'>, {'the': 4})
```

```
>>> 2 * a
defaultdict(<type 'float'>, {'the': 4})
>>> - a
defaultdict(<type 'float'>, {'the': -2})
>>> a['boy'] = 1
>>> a
defaultdict(<type 'float'>, {'boy': 1, 'the': 2})
>>> a.dot({'the': -1, 'girl': 2})
-2.0
```

Now the sentence `the man bit the dog` is represented as `{'dog': 1, 'bit': 1, 'the': 2, 'man': 1}`. Questions:

1. Take a look at `svector.py`, and briefly explain why it can support addition, subtraction, scalar product, dot product, and negation.
2. Take a look at `train.py`, and briefly explain `train()` and `test()` functions.
3. There is one thing missing in my `train.py`: the bias dimension! Try add it. How did you do it? (Hint: by adding `bias` or `<bias>`?) Did it improve error on dev?
4. Wait a second, I thought the data set is already balanced (50% positive, 50% negative). I remember the bias being important in highly unbalanced data sets. Why do I still need to add the bias dimension here??

## 2 Average Perceptron and Vocabulary Pruning

Now implement averaged perceptron based on the version with the bias discussed above. Note that unlike in HW1, this time you have to use the smart implementation of averaging, otherwise it will be way too slow to run, due to the much bigger dimensionality (sparse vectors).

1. Train for 10 epochs and report the results. Did averaging improve on dev? (Hint: should be around 26%). Did it make dev error rates more smooth?
2. Did smart averaging slow down training?
3. What are the top 20 most positive and top 20 most negative features? Do they make sense?
4. Show 5 negative examples in dev where your model most strongly believes to be positive. Show 5 positive examples in dev where your model most strongly believes to be negative. What observations do you get?
5. Try improve the speed by caching sentence vectors from training and dev sets, and show timing results.

## 3 Pruning the Vocabulary

Actually many words only appear once in the training set, and many of them are unlikely to appear in the dev set. Therefore it is common to remove low-frequency words from the training set, as a regularization trick (smaller models alleviate overfitting; those rare words might just make the model memorize idiosyncrasies or noises in the training set).

1. Try neglecting one-count words in the training set during training. Did it improve on dev? (Hint: should help a little bit, error rate lower than 26%).
2. Did your model size shrink? (Hint: should almost halve).
3. Did update % change? Does the change make sense?
4. Did the training speed change?
5. What about further pruning two-count words (words that appear twice in the training set), etc?

## 4 Try some other learning algorithms with sklearn

Now you can try **at least one** other machine learning algorithms of your choice (either covered or uncovered in our course) using `sklearn` (so that you don't need to implement them), such as $k$-NN, SVM, logistic regression, neural networks, decision trees, XGBoost, etc. To use `sklearn`, like in HW3, you will need to convert an `svector` object to a numpy vector. Note that this training might take a very long time, so you should prune one-count and two-count words first. After pruning, the training might still take a long time. If you find it too slow, you can further prune more low-count words, and/or use a subset (e.g., 5,000) of the training set.

Q: What did you learn in terms of the comparison between averaged perceptron and these other (presumably more popular and well-known) learning algorithms?

## 5 Deployment

You can also try other tricks such as bigrams (two consecutive words) and removing STOP words (such as "the", "an", "in", "and", just Google it). Collect your best model and predict on `test.txt` to `test.txt.predicted`.

Q: what's your best error rate on dev, and which algorithm and setting achieved it?

## 6 Debriefing (required):

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone, or mostly with other people?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?