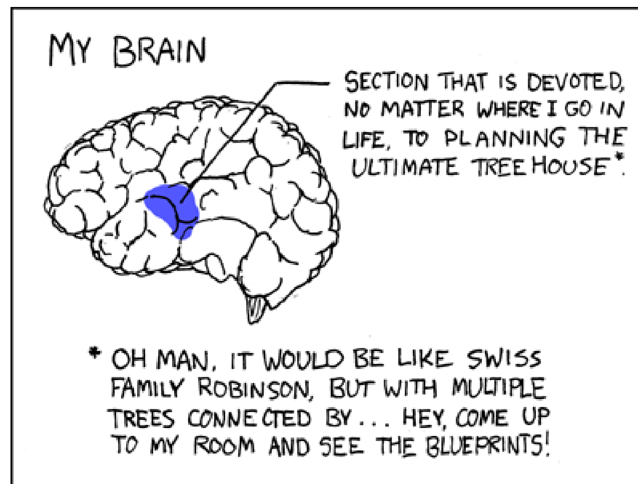
CS451 - Assignment 3
Perceptron Learning Algorithm

Due: Sunday, September 18 by 11:59pm



For this assignment we will be implementing some of the perceptron learning algorithm variations and comparing both their performance and run-times. As always, make sure to read through the entire handout before starting. You may (and I would strongly encourage you to) work with a partner on this assignment. If you do, you must both be there whenever you are working on the project. If you'd like a partner for the lab, e-mail me asap and I can try and pair people up.

1 Requirements

Implement two preceptron learning algorithms we talked about in class:

1. Implement a class called `PerceptronClassifier` that implements the `Classifier` interface. Your implementation must:
   - Follow the basic algorithm outlined in class (and in the book), updating the weights when a mistake is made. You do NOT need to check for convergence, we will only stop when we have reached the iteration limit.
   - Include an empty constructor.
   - To figure which weights you'll need in your `train` method you can call the `getAllFeatureIndices()` method from the data set that's passed in. The features in the set returned will define the features that your classifier will be learning over, i.e. the weights.

- Initially all of the weights and the b-value should start out at 0.
- Include a method called `setIterations` that takes an integer and sets the number of iterations to iterate over on the data. By default, the number of iterations should be set to 10.
- Include a `toString` method that returns the weights and the b value of the classifier as a string in the following format:

  `0:weight_0 1:weight_1 2:weight2 ... n:weight_n b-value`

  that is, `<feature_number>:<weight>`, space separated with the features in increasing order.

2. Implement a class called `AveragePerceptronClassifier` that implements the `Classifier` interface. This class should follow the same specification as the basic perceptron algorithm (including constructor, `setIterations` and `toString`), however, in addition to recalculating the weight vector each time a mistake is make, we will also keep track of an average weight vector that is the weighted average of all of the weight vectors seen so far (called the "average perceptron" algorithm in the book and slides).

The pseudocode in the book for this algorithm is NOT correct so I have included a correct version here:

```
w = [0,0,...,0] // normal weights, one for each feature
b = 0  // normal b
u = [0,0,...,0] // aggregate weights, one for each feature
b2 = 0
updated = 0
total = 0

for iter = 1 ... MaxIter{
  // randomly shuffle the dataset
  for all (x,label) in dataset{
    if ((w x + b) * label <= 0 ){  // if we misclassify example x using weights w
      // update our final, weighted weights
      for each u_i in u:
        u_i = u_i + updated * w_i

      b2 = b2 + updated * b

      // update all the perceptron weights
      for each w_i in w:
        w_i = w_i + label * x_i

      b = b + label

      updated = 0
    }// end of misclassify

    updated = updated+1
    total = total + 1
  } // end of dataset for loop
} // end of outer loop

// do one last weighted update here of the u and b2 weights based on the final weights

// divide all of the aggregate weights by the total number of examples
for each u_i in u:
  u_i = u_i/total

b2 = b2/total

return (u, b2)
```

This version works by keeping a running total in `updated` of the number of examples the current weights have gotten correct. When a mistake is made by the weights, we add to our aggregate weights (u and b2) a weighted copy of the current weight vector. In the end, we then normalize the weights by the total number of examples we examined during training. This approach can suffer from overflow on the weights in some rare circumstances, but it is

easier to implement.

2 Starter Code

I have again included code to get you started at:

`http://www.cs.pomona.edu/~dkauchak/classes/cs158/assignments/assign3/assign3-starter.tar.gz`

The starter code includes all of the same code as last time with the following changes:

- I have changed the `DataSet` class to also include a constructor that just takes a CSV file and does NOT require the index of the label. This constructor assumes that the label is the last column.
- I have included a class called `ClassifierTimer` in the `ml.classifier` package. This class implements functionality for timing how long it takes for a classifier to train and test.

3 Data

At: `http://www.cs.pomona.edu/~dkauchak/classes/cs158/assignments/assign3/data/`

I have included three .csv files for use on this assignment:

- `titanic-train.perc.csv`: The same training data as our last assignment, but now the labels are 1 for positive and -1 for negative.
- `simple1.csv`: The first example perceptron learning algorithm walkthrough from the notes.
- `simple2.csv`: The second example perceptron learning algorithm walkthrough from the notes.

4 Evaluation

Once you have your two perceptron classifier algorithms working, let's see how well they do!

Answer the following questions and put them in a file called `experiments` (pick some reasonable file type). Explicitly label each answer with the question number and make sure to put your name at the top of the file.

1. What is the accuracy of the two perceptron classifiers on the Titanic data set. To calculate this, generate a random 80/20 split (using `dataset.split(0.8)`) train the model on the 80% fraction and then evaluate the accuracy on the 20% fraction. Repeat this 100 times and average the results (hint: do the repetition in code :).

2. What is the best iteration limit to use for this data? To answer this, do the same calculations as above (average 100 experiments), but do it for increasing iteration limits.
3. Do we see overfitting with this data set? Repeat the experiment from question 2 and calculate the accuracy this time on both the testing data (like before) *and* the training data. How does this compare to your results for decision trees? Does this make sense? Include a 2-3 sentence explanation.
4. Use the `ClassifierTimer` class to compare the run-time performance of your classifiers. Do the results make sense? Include a 1-2 sentence explanation.

5 Hints/Advice

- I have included two very basic CSV files where you know what the answer at each step of the algorithm should be. I *strongly* encourage you to check your results on these first. Because of the simplicity of the perceptron learning algorithm, it can be challenging to make sure your code is working correctly. To compare against the results in the slides, you'll need to fix `b` to zero. To do this, just comment out the line in your code that updates b. In addition, you'll need to start out the weight vector at 1.0 (instead of 0.0). Once you've done this, your answers should be identical to those on the slides.
- As always, test/debug incrementally!!! Do NOT simply try and code the whole thing up and then start debugging from there. Break the problem into smaller subproblems/submethods, implement them one at a time and test each one before moving on. I know this seems like it takes more time, but the time you put in early on will more than make up for it down the road both in time and grief.
- Don't forget that you need to randomize the order which you traverse your examples before each iteration.

6 Extra Credit

For those who would like to experiment (and push themselves) a bit more (and of course, get a bit of extra credit) you can try out some of these extra credit options. If you try out these options, include an extra file called `extra.txt` that describes what extra credit you did.

- Implement a class called `VotedPerceptronClassifier` that implements the `Classifier` interface. Your implementation:
  - Should have the same methods as the basic perceptron algorithm (i.e. same constructor and `setIterations` method), however, you don't have to implement the `toString` method.
  - Should keep track of all of the weight vectors found during training along with their votes (i.e. how many examples were correctly classified by that weight vector).

  – The `classify` method should then calculate a voted majority.

  If you go this route, also include classification performance in your writeup along with train/test timing.

- The right way to implement the `AveragePerceptronClassifier` is to `extend` the basic `PerceptronClassifier` class and only alter the `train` method. For extra credit, implement it this way. To do this, you must minimize the amount of repeated code and maximize the amount of shared code.
- Find another dataset with at least 5 features and 500 examples and provide a comparison of your perceptron algorithms on it.

7 When You're Done

Make sure that your code compiles, that your files are named as specified and that you have followed the specifications exactly (i.e. method names, number of parameters, etc.).

Create a directory with your last name, followed by the assignment number, for example, for this assignment mine would be `kauchak3`. If you worked with a partner, put both last names.

Inside this directory, create a `code` directory and copy all of your code into this directory, maintaining the package structure.

Finally, also include your `experiments` file with the answers from Section 4.

`zip` this folder and submit that file on the submission page on the course web page.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file
- Each class and method should have an appropriate doctsring
- If anything is complicated, it should include some comments.

There are many possible ways to approach this problem, which makes code style and comments very important here so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.