Machine Learning *COMP 6630, Spring 2020*

HW03: Perceptron

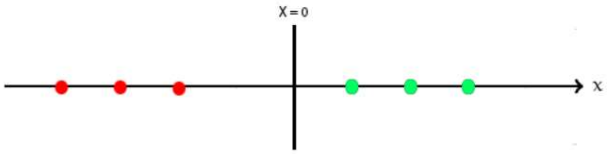
Remember that only PDF submissions are accepted.

1. A common way to get rid of having to deal with the threshold separately on a perceptron is to add anew feature. This feature always has value one, and you learn a weight for it. Thus, if you have a 100 dimensional problem with a threshold, we solve it as a 101 dimensional problem with threshold at zero. Draw a picture for one dimensional data and a linear separator with a (non-zero) threshold and draw the corresponding picture for the same data, "lifted" into two dimensions, with the corresponding linear separator with threshold zero. (Please make sure that the two separators are actually equivalent!)

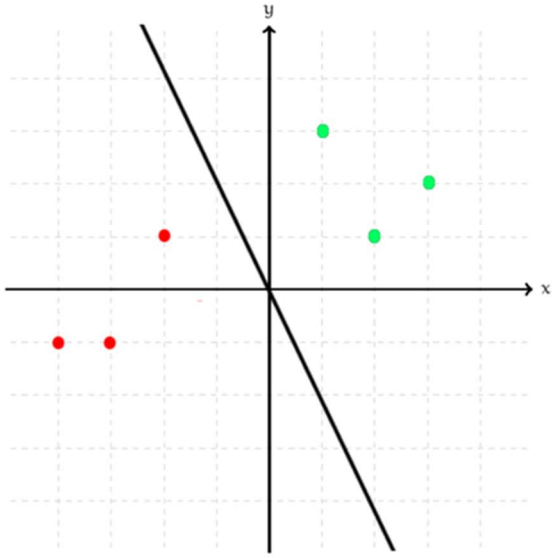
For my 1D example I will be using the vectors: Ω = (0, 1) and x = (1, x1)

In my 2D example, I will be using the vectors: Ω = (0, 1, 0.5) and x = (1, x1, x2)

Below we see an image of my 1D example:



Below we see a function of the same points lifted into a 2D graph and the resulting function of doing a dot product for it's two vectors.



By doing the dot product of our x and omega vector we can figure out what the separating line for our data should be.

1D: 0*1 + 1*x1 = x1 When we transform this, we conclude that the separating point is X>0

2D: 0*1 + 1*x1 + 0.5*x2 After some transformation we know that x2 > -2x

2. The perceptron will only converge if the data is linearly separable. For linearly separable data with margin δ, if $||\mathbf{x}|| \leq R$ for all data points, then it will converge in at most $\frac{R^2}{\delta^2}$ iterations (In class we assumed that $||\mathbf{x}|| \leq 1$). It is possible to force your data to be linearly separable as follows. If you have N data points in D dimensions, map data point $\mathbf{x}_n$ to the (D + N)-dimensional point $(\mathbf{x}_n, \mathbf{e}_n)$, where $\mathbf{e}_n$ is a N-dimensional vector of zeros, except for the n^{th} position, which is 1. (Eg., $\mathbf{e}_4 = (0,0,0,1,0,...)$.)

   (a) Show that if you apply this mapping the data becomes linearly separable (you may wish to do so by providing a weight vector $\mathbf{w}$ in (D + N)-dimensional space that successfully separates the data).

   To explain this, I would like to show an example problem. Let us say that our e2 vector is (0,0,1) and our weight vector is Ω = (0, 0.5, 1). Now we take into account our vector for our point x = (1, x1, x2). Then we can see that when we apply the method described above, it will turn out that we would only be looking at one attribute in our data. That would be the attribute that gets multiplied by the 1 in our e2 vector. Therefor we can make this linearly separable by looking at only that 1 attribute. In this case that separating line would be x2 > 0.

   (b) How long will it take the perceptron algorithm to converge on this augmented data?

   For each miss classified point, there will be one extra calculation to adjust the separating line. So how long it takes depends on the size of your data pool and on how much of that data is classified correctly to begin with.

   (c) How does this mapping affect generalization (i.e., test performance)?

   When we map our data this way, we essentially ignore all the attributes of our data except for 1 attribute. This means that when we do this, we will lose a lot of accuracy. I would say that instead transforming or forcing our data this way to use the perceptron algorithm, it would be better to use a different machine learning method.

3. (Programming) You need to implement the perceptron algorithm as in the slides. Just hand-in your code, and you don't have to play with data for now. (But you are encouraged to play with some toy data you find on your own to validate if your implementation is correct or not). You can reuse the code to this question in your mini-project, and play with data then.

```python
import numpy as np

class Perceptron:

    def __init__(self, learning_rate = 0.01, n_iters = 1000):
        self.lr = learning_rate
        self.__ne__ = n_iters
        self.activation_func =  self._unit_step_func
        self.weights = None
        self.bias = None


    def fit(self, X, y):
        n_samples, n_features = X.shape

        #initializing weights
        self.weight = np.zeros(n_features)
        self.bias = 0

        y_ = np.array([1 if i > 0 else 0 for i in y])

        for _ in range(self.n_iters):
            for index, x_i in enumerate(X):
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_predicted = self.activation_func(linear_output)


                update = self.lr * (y_[index] - y_predicted)
                self.weights += update * x_i
                self.bias += update


    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        y_predicted = self.activation_func(linear_output)
        return y_predicted


    #using np.where so it works for multiple samples
    def _unit_step_func(self, x):
        return np.where(x>=0, 1, 0)

```