CSE 250B: Assignment #5

February 16, 2016

Apoorve Dave

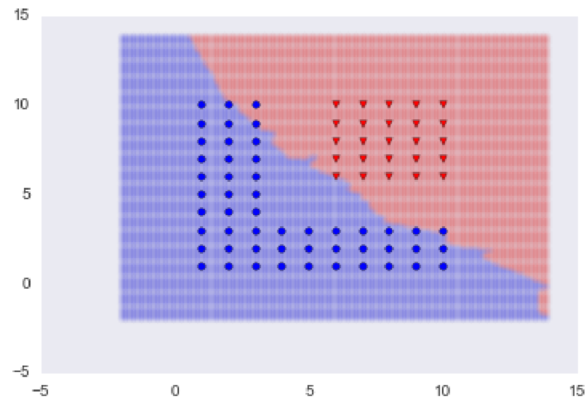
a1dave@ucsd.edu - A53103070

1 Voting Perceptrons

1.1 Voting Perceptron

The decision boundary from voting perceptron is Not Linear. Plot:

Figure 1: Voting Perceptron Decision Boundary(T=10):



1.2 Downsampled Voting Perceptron

1.2.1 Psuedocode

One way to downsample the weights as obtained by normal voting perceptron is to pick the latest L weights, discarding the previous ones. The main idea behind this approach is that in the beginning of learning, the weights of perceptrons will be premature. The earliest of learned weights therefore tend to be of lesser importance compared to the latest learned weights. Therefore, in my algorithm, I have implemented the same voting perceptron as mentioned in the HW but stored them in a FIFO queue, discarding the earliest weights as and when a new weight vector is added to the queue. Prediction is done as before, based on weighted majority vote:

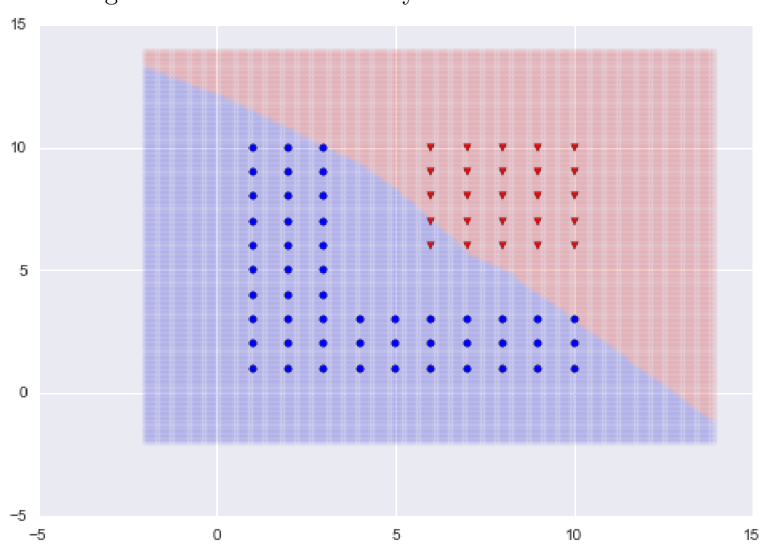
$$sign\left(\sum_{j=1}^{L} c_j sign(w_j.x)\right)$$

Algorithm 1 Psuedocode for downsampled classifiers

Create queue *weightQueue* to store the latest L weight vectors, and a corresponding queue for *survivalTimes*
Set *maxLength* of queues to L. (These are FIFO. If filled, earliest element pops out)
Set initial weight $w_1 = 0$ to 0 and $c_1 = 0$.
for i = 1 to T: **do**
    -Randomly permute the data points
    for j = 1 to n: **do**
        $w = weightQueue.head$ //latest value
        if $(x(i); y(i))$ is misclassied by w: **then**
            $w_{new} = w + y^{(i)}.x^{(i)}$
            Push w_{new} to head of *weightQueue*, 1 to *survivalTimes*. Earliest elements will be popped if queue is full.
        else
            Increment count for the head of *survivalTimes*
        end if
    end for
end for
return *weightQueue* and *survivalTimes*

1.2.2 Decision Boundary with Downsampled weights

Figure 2: Decision Boundary with T=100 and L=20



1.3 Averaged Perceptron

1.3.1 Psuedocode:

The averaged perceptron is a simpler implementation of maintaining survival times as a measure of good weight vectors. In this case, we can keep a running average of all the weight vectors seen till now. If a weight correctly classifies a point, we add this weight vector to a cumulative weight vector. If it fails to classify the point correctly, we update the weight vector as before (i.e. $w_{l+1} = w_l + y^{(i)} * x^{(i)}$) before adding it to cumulative weight vector. We can then either average over the cumulative weight vector or just use the sum directly to classify unseen data based on

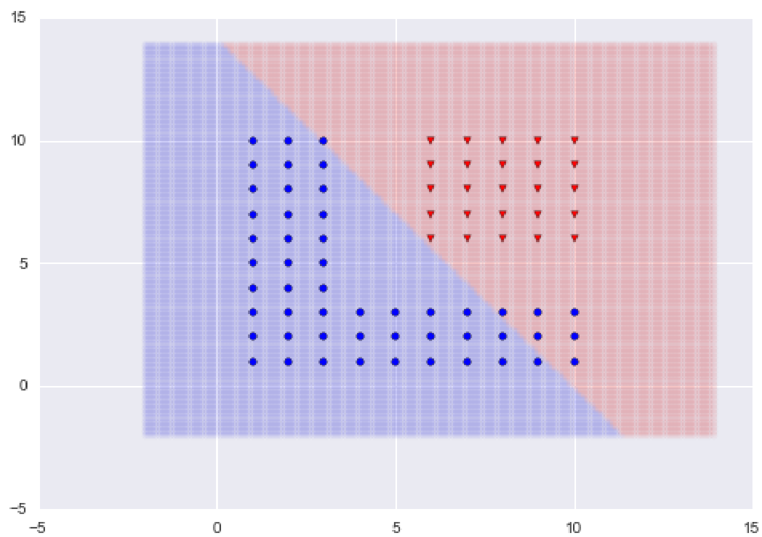
$$sign(w_{cumulative}.x) \qquad \text{(where } w_{cumulative} = \sum c_j w_j\text{)}$$

Algorithm 2 Pseudocode for the Averaged Perceptron

```
Create w_current = 0, w_cumulative = w_current
for i = 1 to T: do
    -Randomly permute the data points
    for j = 1 to n: do
        if (x(i); y(i)) is misclassied: then
            w_current = w_current + y^(i).x^(i)
        end if
        w_cumulative = w_cumulative + w_current
    end for
end for
return w_cumulative
P.S. Note here score is maintained as every correct classification updates w_cumulative by w_current, otherwise updates w_current before adding to w_cumulative.
```

1.3.2 Averaged Perceptron Decision Boundary:

Figure 3: Averaged Perceptron Decision Boundary with T=10

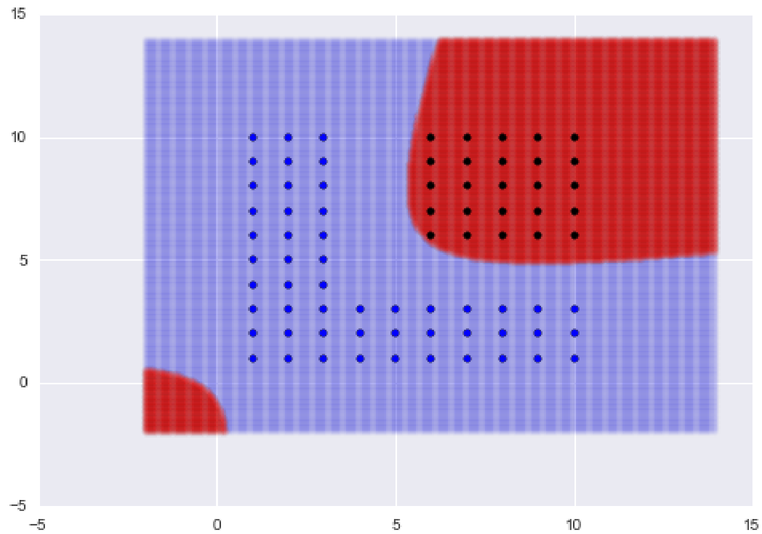


2 Kernelized Perceptrons

2.1 Quadratic Kernel

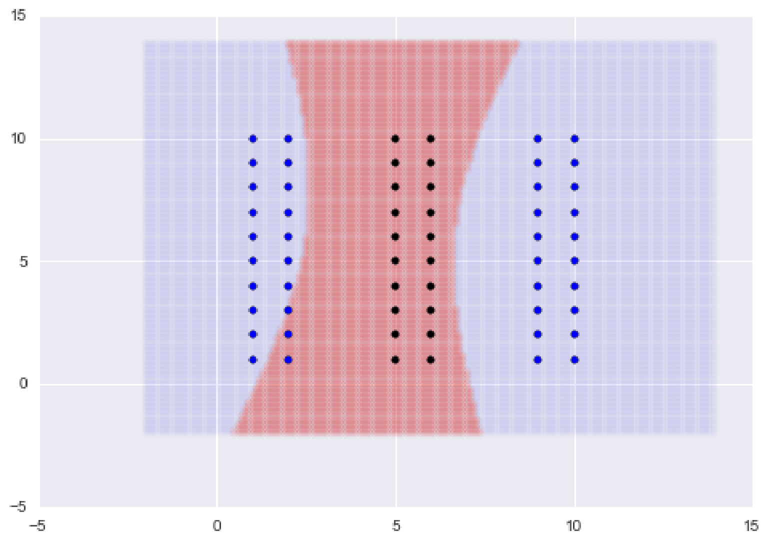
2.1.1 Data Set 1 Decision Boundary

Figure 4: Data Set 1 Decision Boundary with Quadratic Kernel



2.1.2 Data Set 2 Decision Boundary

Figure 5: Data Set 2 Decision Boundary with Quadratic Kernel



2.2 RBF Kernel

2.2.1 Data Set 1 Decision Boundaries

Figure 6: RBF, Data Set 1, $\sigma = 0.05$

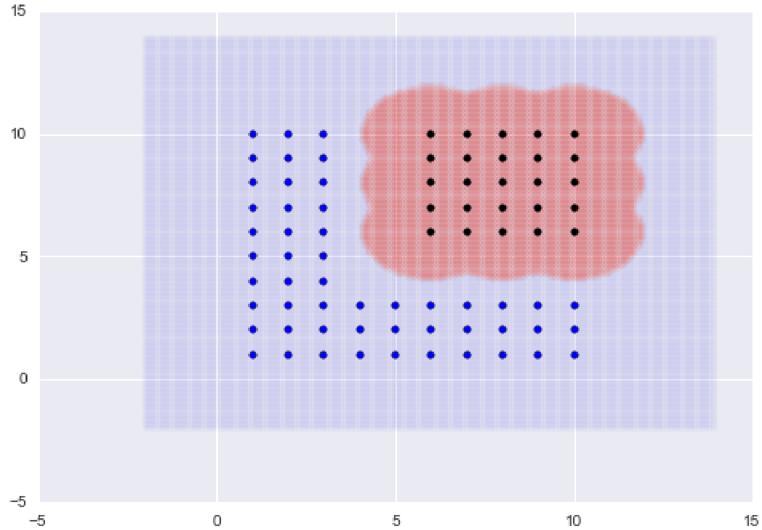


Figure 7: RBF, Data Set 1, $\sigma = 0.08$

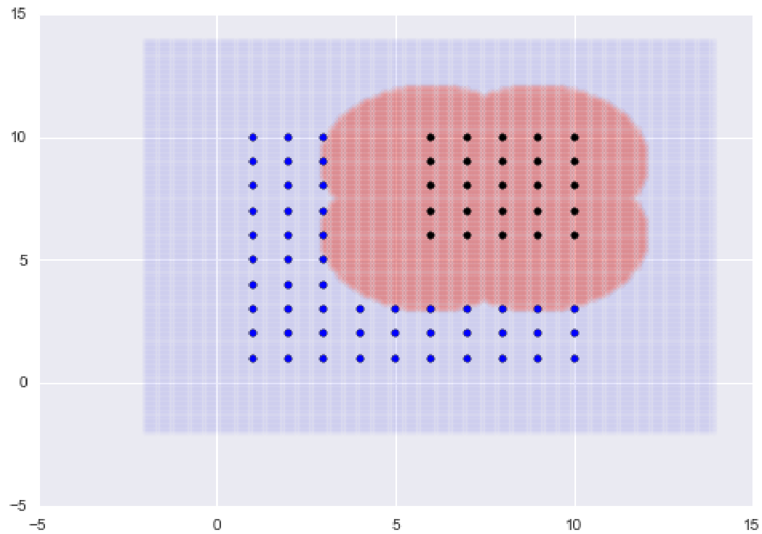
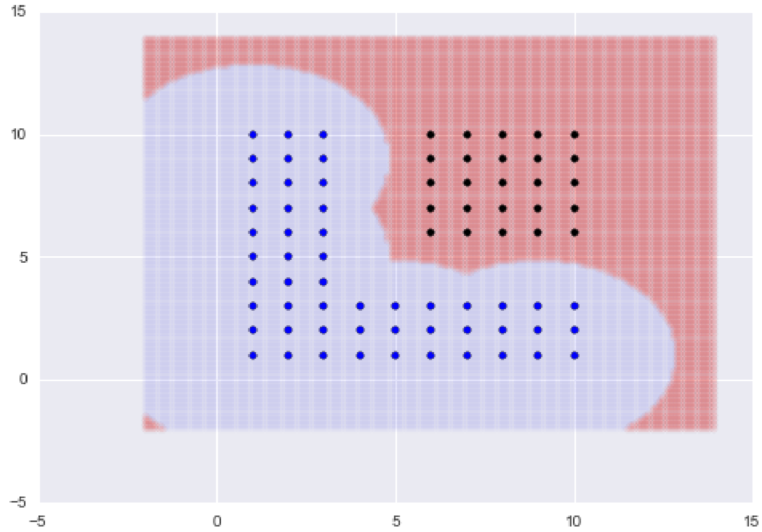


Figure 8: RBF, Data Set 1, $\sigma = 0.1$



2.2.2 Data Set 2 Decision Boundaries

Figure 9: RBF, Data Set 2, $\sigma = 0.05$

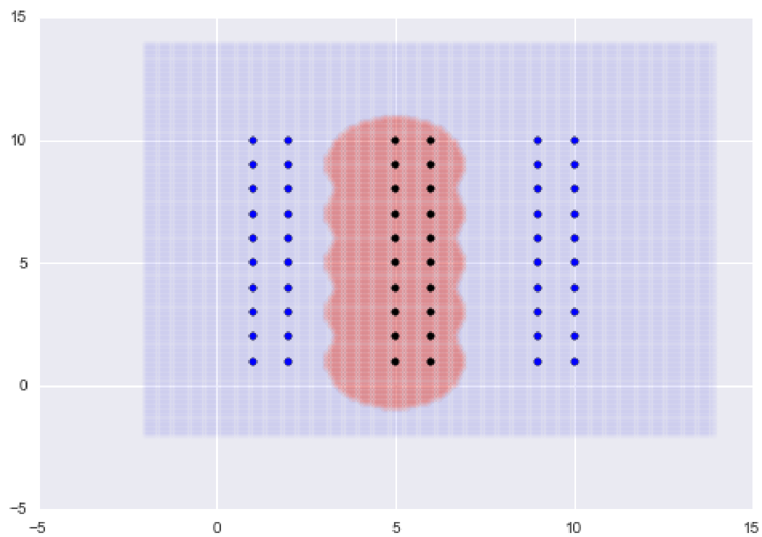


Figure 10: RBF, Data Set 2, $\sigma = 0.08$

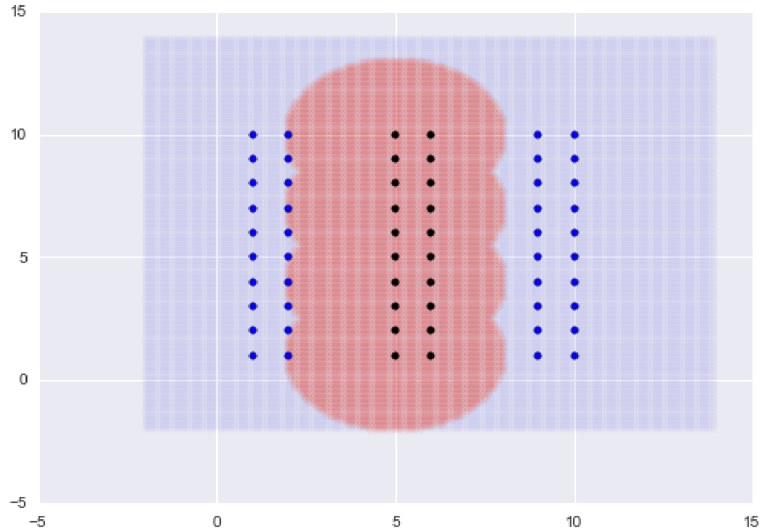


Figure 11: RBF, Data Set 2, $\sigma = 0.1$