CS6350 - Homework/Assignment-2

Arnabd Das(u1014840)

September 26, 2016

Question-1: Feature Expansion

Given Concept Class, C, consisting of functions f_r, defined as ,

$f_r = +1$; if $4x_1^4 + 16x_2^4 \leq r$
$f_r = -1; otherwise$

Let us define a transformation ϕ that transforms $\mathbf{X}$ to a newspace $\mathbf{Z}$, such that:

$z_j = \phi(x_j) = x_j^4$

Thus in the 2D space of Z, we can write:

$[z_1, z_2] = \phi([x_1, x_2]) = [x_1^4, x_2^4]$

Hence, in the newspace , f_r can be transformed transformed as:

$f_r = +1$; if $4z_1 + 16z_2 \leq r$
$f_r = -1; otherwise$

Then we can write an activation function as:

$\mathrm{Y(Z,W)} = r - 4z_1 - 16z_2$

$$\mathrm{Y(Z,W)} = r + [-4 - 16] \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

$$\mathrm{Y(Z,W)} = \mathrm{r} + W^T \mathrm{Z}$$

Where, the Bias = r , Weight Vector(W) = $[-4 - 16]$

Consider a point, in $[z_{1_t}, z_{2_t}]$ in the new space. Then,

$$Y(Z, W) = r - 4z_{1_t} - 16z_{2_t} \tag{1}$$

and using the transformation back to X domain, we can write

$$Y(Z, W) = r - 4x_{1_t}^4 - 16x_{2_t}^4 \tag{2}$$

Since (1) and (2) are equivalent under the transformation ϕ, therefore

$\mathrm{Y(Z,W)} = \mathrm{r} + W^T\mathrm{Z} \geq 0$ implies $r - 4x_{1_t}^4 - 16x_{2_t}^4 \geq 0$ implies $f_r(x_{1_t}, x_{2_t}) = +1$
$\mathrm{Y(Z,W)} = \mathrm{r} + W^T\mathrm{Z} < 0$ implies $r - 4x_{1_t}^4 - 16x_{2_t}^4 < 0$ implies $f_r(x_{1_t}, x_{2_t}) = -1$

Thus, with the above bias and weight vector in the newspace, the points are linearly separable. (Answer)

Question-2:: Mistake Bound Model of Learning

Consider the instance space of integer points on the two dimensional plane (x_1, x_2) with $-80 \leq x_1, x_2 \leq 80$. Let C be a concept class defined on this instance space. Each function, f_r, in C is defined by a radius r($1 \leq r \leq 80$) as follows:

$f_r(x_1, x_2) = +1$; $x_1^2 + x_2^2 \leq r^2$
$f_r(x_1, x_2) =$ -1; otherwise

(1) The set of target concepts in the concept class C is limited by the range of r:

$SizeofConceptClass, \left|C\right| = 80$ (answer).

(2) If f_r is our current guess for the function, and given an input point (x_1^t, x_2^t), the prediction is incorrect if:

$$f_r(x_1^t, x_2^t) \neq y_t$$

The mismatch condition has 2 cases:
1. The prediction is +1 while the output label is -1.
2. The prediction is -1 while the output label is +1.
This means the product of the prediction and the correct output label, y^t, will always be less than zero if there is a mismatch.

$$f_r(x_1^t, x_2^t).y^t < 0 \tag{3}$$

We can replace $f_r(x_1^t, x_2^t)$ directly in terms of x_1^t, x_2^t, r such that the expression evaluation directly outputs a $+1/-1$ prediction equivalent to f_r. Wrtiting the expression $sgn(r^2 - (x_1^t)^2 - (x_2^t)^2)$ results in +1 when $f_r(x_1^t, x_2^t)$ is +1 and -1 when $f_r(x_1^t, x_2^t)$ is -1.
Thus, we can write the mismatch condition checker as:

$$sgn(r^2 - (x_1^t)^2 - (x_2^t)^2).y^t < 0$$

(3) The FIRST Method:In case there is a mismatch, it entails two cases.

(1). **If the prediction is positive and the actual label is negative** : If the prediction is positive, it means the point lies completely inside the circle corresponding to our current **r**. The correction will require an update to **r** such that the point lies on or outside the circle. Thus we need to reduce **r** as:

$$\mathbf{r} = \mathbf{floor}(\sqrt{x_1^2 + x_2^2})$$

(2). **If the prediction is negative and the actual label is positive** : If the prediction is negative, it means the point lies on or outside the circle corresponding to our current **r**. The correction will require an update to **r** such that the point lies completely inside the circle. Thus we need to increase **r** as:

$$\mathbf{r} = ceil(\sqrt{x_1^2 + x_2^2})$$

The SECOND Method: In this method, everytime for the prediction following a mismatch, we choose a random r from an adjusted upper and lower bound of **r**.
Suppose $\mathbf{r}_{ub}$ denotes the current upper bound
Suppose $\mathbf{r}_{lb}$ denotes the current lower bound
Let us revisit the 2 conditions of mismatch:

(1). **If the prediction is positive and the actual label is negative** : This necessarily means the correct classifier in this concept class lies definitely within the circle of radius dictated by the point itself. Thus we can reduce the upperbound to be :

$$r_{ub} = floor(\sqrt{x_1^2 + x_2^2})$$

Then for the next prediction choose a random **r** as:

$$\mathbf{r} = \mathbf{randomInRange}(r_{lb}, r_{ub})$$

(2). **If the prediction is negative and the actual label is positive** : This necessarily means the correct classifier in this concept class lies definitely outside the circle of radius dictated bu the point itself. Thus we can increase the lowerbound to be

$$r_{lb} = ceil(\sqrt{x_1^2 + x_2^2})$$

Then for the next prediction we choose a random **r** as:

$$\mathbf{r} = \mathbf{randomInRange}(r_{lb}, r_{ub})$$

(4) Using the FIRST Method: Since, in this case the concept class and the hypothesis space are the same, we are guaranteed to find the true target function using our method. The algorithm pseudoCode is described below:

```
findTargetFunction()
    set r = randomInRange(1,80)
    mistakeCounter = 0
    while True:
        if (mistakeCounter == |C| − 1) return r
        X = [x1, x2] = getInputVector()
        if(X == NULL) return r //Input Vector exhausted
            mistakeCounter++
```

if($y < 0$)
r = floor($\sqrt{x_1^2 + x_2^2}$)
else
r = ceil($\sqrt{x_1^2 + x_2^2}$)
}

For every mistake made, it is guaranteed to prune atleast 1 **r** from the space. If the available concepts class size in i'th step is $|C_i|$, and a mistaker is made in the i'th step , then $|C_{i+1}| < |C_i|$. Thus it can make atmost $|C| - 1 = 79$ mistakes.(Answer). Hence, in this algorithm, we exit the while loop under two circumstances, if the mistake bound is reached or if the training list has been exhausted. Since the current **r** satisfies all vectors in the training set, it has to be the correct classifier for that training set. Or, if the mistake bound is reached, only one function is left in the space and has to be the correct classifier.

Using the SECOND Method: Since in this case the concept class and the hypothesis space are the same, we are guaranteed to find the true target function using our method. The algorithm pseudocode is described below:

findTargetFunction(r_{lb}, r_{ub}**)**
set **r** = randomInRange(r_{lb}, r_{ub})
while true:
X = $[x_1, x_2]$ = getInputVector()
if (**X** == NULL) **return r** // Input vector exhausted
if($sgn(r^2 - x_1^2 - x_2^2).y < 0$) {
if($y < 0$)
r_{ub} = floor($\sqrt{x_1^2 + x_2^2}$)
else
r_{lb} = ceil($\sqrt{x_1^2 + x_2^2}$)
if ($r_{ub} == r_{lb}$) **return** r_{lb}
else **r** = randomInRange(r_{lb}, r_{ub})
}

In this case, we adjustr the upper and lower bound depending on the direction of mismatch. The idea is we keep reducing the band of **r** between r_{lb} and r_{ub} such that it is guaranteed to converge to a single **r** when it finds the true classifier. However, before converging if we exhaust the training set, then the current **r** is guaranteed to be true for this training set and is the correct classifier.
Note, we do not track the number of mistakes. Although, the atmost number of mistakes still remains $|C| - 1 = 79$, we do not need to check on that since it is guaranteed to converge when $r_{ub} = r_{lb}$.

(5) (a) It is possible to define the set of hypothesis using just 2 integers. Basically we define the hypothesis space in terms of a band of circle whose radius lies between an upper bound and a lower bound that limits the value that **r** can take. Everytime a new hypothesis is required for prediction, we define the predicting function in terms of **r** such that:
if $r_{lb} -> lowerBound$
if $r_{ub} -> upperBound$, Then,
$\mathbf{r} = \frac{r_{lb} + r_{ub}}{2}$
And for every prediction that goes wrong, instead of correcting **r**, we prune the r_{ub} and r_{lb}, such that the band of valid **r**'s reduces by atleast half the previous concept class size.

(b) If f_r is our current guess for the function, and given an input point (x_1^t, x_2^t), the prediction is incorrect if:

$$f_r(x_1^t, x_2^t) \neq y_t$$

The mismatch condition has 2 cases:

1. The prediction is +1 while the output label is -1.
2. The prediction is -1 while the output label is +1.

This means the product of the prediction and the correct output label, y^t, will always be less than zero if there is a mismatch.

$$f_r(x_1^t, x_2^t).y^t < 0 \tag{4}$$

We can replace $f_r(x_1^t, x_2^t)$ directly in terms of x_1^t, x_2^t, r such that the expression evaluation directly outputs a $+1/-1$ prediction equivalent to f_r. Since in this case, $\mathbf{r} = \frac{r_{lb} + r_{ub}}{2}$, we can write the expression $sgn(\frac{r_{lb} + r_{ub}}{2}^2 - (x_1^t)^2 - (x_2^t)^2)$ results in +1 when $f\left(\frac{r_{lb} + r_{ub}}{2}\right)(x_1^t, x_2^t)$ is +1 and -1 when $f\left(\frac{r_{lb} + r_{ub}}{2}\right)(x_1^t, x_2^t)$ is -1.

Thus, we can write the mismatch condition checker as:

$$sgn\left(\left(\frac{r_{lb} + r_{ub}}{2}\right)^2 - (x_1^t)^2 - (x_2^t)^2\right).y^t < 0$$

(c) The Halving Algorithm pseudocode for this specific concept class is described below:

findTargetFunction()

r_{lb} = lowerBound of $\mathbf{r} = 1$
r_{ub} = upperBound of $\mathbf{r} = 80$
set $\mathbf{r} = \frac{r_{lb} + r_{ub}}{2}$
while True:
  $\mathbf{X} = [x_1, x_2]$ = getInputVector()
  if($\mathbf{X}$ == NULL) **return r** // Input Vector exhausted
  if($sgn(r^2 - x_1^2 - x_2^2).y < 0$) {
    if($y < 0$)
      r_{ub} = floor($\sqrt{x_1^2 + x_2^2}$)
    else
      r_{lb} = ceil($\sqrt{x_1^2 + x_2^2}$)
    if ($r_{ub} == r_{lb}$) **return** r_{lb}
    else $\mathbf{r} = \frac{r_{lb} + r_{ub}}{2}$
  }

Since, every mistake leads to pruning of atleast $\frac{1}{2}$ of the space because the hypothesizing $\mathbf{r}$ was the mid point , then if a mistake is amde in i'th step with concept class C_i, then at the (i+1)'th step the following must hold:

$$|C_{i+1}| < \frac{1}{2}|C_i| \tag{5}$$

Since, at the n'th step, which is the final step, we will have the convergence of $r_{lb} == r_{ub}$, only one concept left in the concept class that appropriately fits the data. Hence the following relation must hold at the end:

$$1 = |C_n| < \frac{1}{2}|C_{n-1}| < \frac{1}{2^2}|C_{n-2}| \cdots < \frac{1}{2^n}|C_0| = \frac{1}{2^n}|C0| \tag{6}$$

Thus , the above halving algorithm makes atmost $\mathbf{log}|\mathbf{C}|$. (Answer)
Since $|C|$=80, the atmost mistakes possible suing this halving algorithm is $\mathbf{log}|80|$. (Answer).

Question-3.3: The Sanity Checker

(1) The Perceptron algorithm has been implemented as a combination of simplePerceptron and MarginPerceptron. Basically simple perceptron is a special case of margin perceptron with the margin value set to 0. It has been run on the learning rates(0,0.1,0.01) with weight vector and bias initialized to zero and run for just one pass. Table-1 result describe the findings for the different learning rates:

Table 1: Sanity Experiment with Table:2

LeraningRate	Bias	WeightVector	Mistakes-During-Learning	LearningAccuracy
1	0	0,1,0,-1,2	4	83.33%
0.1	0.0	0.0,0.1,0.0,-0.1,0.2	4	83.33%
0.01	0.0	0.0,0.01,0.0,-0.01, 0.02	4	83.33%

(2) In this experiment both the simple perceptron and the margin perceptron were run on random initialization of the weight vectors and the bias. The random weight vectors and the bias were chosen from a random normal distribution. **Due to the random initialization, every run shows a different result(not extremely different) and hence reruns do not exactly match with previous runs.** Hence, the logs used to report the data will also be submitted along with the submission. For the simple perceptron learning rates of **1,0.1,0.01** were used and for the margin perceptron additional margins used were **1,2,3,4,5**. In Table 2, we report the results for the simple perceptron and in Table 3, the margin perceptron across all the learning rates used for one pass.

Table 2: Simple Perceptron Experiment-one pass

LeraningRate	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Acuracy
1	-2.995266	1373	78.67%	77.38%
0.1	-0.300220	1373	81.353%	81.0915%
0.01	-0.029309	1390	82.91%	82.353%

Table 3: Margin Perceptron Experiment-one pass

LeraningRate	Margin	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
1	1	-3.001774	1493	76.59%	76.14%
1	2	-3.004479	1575	73.82%	73.14%
1	3	-4.017713	1630	73.80%	72.66%
1	4	-4.001484	1692	70.23%	69.52%
1	5	-5.001172	1781	72.91%	72.18%
0.1	1	-0.716866	1947	70.25%	70.02%
0.1	2	-0.994660	2176	67.96%	67.82%
0.1	3	-1.104182	2279	65.51%	65.43%
0.1	4	-1.295657	2365	64.34%	63.77%
0.1	5	-1.595738	2436	64.25%	64.35%
0.01	1	-0.230197	2653	61.63%	61.53%
0.01	2	-0.416474	2910	59.29%	59.21%
0.01	3	-0.593608	3058	58.13%	58.20%
0.01	4	-0.767157	3149	55.83%	55.75%
0.01	5	-0.929041	3215	55.44%	55.14%

For a single pass, margin perceptron shows more accuracy with learning rates of 1 and lower margin values.

(3) In this section, we run batch experiments over the training data, that is, instead of a single pass, we

loop over the training data multiple times. There are two sets of experiments done here, first we loop over the training data with no-shuffling and Second we loop over the training data with shuffling. Initialization is random in the same manner as previous question. Table 4 and Table 5 Show the results for simple and margin perceptron with no-shuffling respectively.

Table 4: Simple Perceptron Experiment-Batch-noShuffle

LeraningRate	Epochs	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
1	3	-4.002588	4054	79.62%	79.39%
0.1	3	-0.289600	4045	79.59%	78.42%
0.01	3	-0.043126	4054	79.62%	79.39%
1	4	-3.994685	5394	80.35%	79.61%
0.1	4	-0.318648	5427	74.19%	73.44%
0.01	4	-0.050691	5412	78.45%	77.88%
1	5	-4.004014	6752	83.28%	82.78%
0.1	5	-0.402212	6752	83.28%	82.78%
0.01	5	-0.036570	6732	81.01%	80.53%

Table 5: Margin Perceptron Experiment-Batch-noShuffle

LeraningRate	Epochs	Margin	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
1	3	1	-4.996818	4345	80.66%	80.70%
1	3	2	-4.998888	4583	76.56%	75.51%
1	3	3	-5.005082	4787	74.41%	73.38%
1	3	4	-4.995312	4959	69.87%	68.68%
1	3	5	-6.002095	5144	73.47%	72.29%
0.1	3	1	-0.796138	5634	69.36%	68.83%
0.1	3	2	-1.190008	6176	67.35%	66.85%
0.1	3	3	-1.516534	6475	65.07%	64.27%
0.1	3	4	-1.903123	6667	65.18%	64.54%
0.1	3	5	-2.112248	6807	64.60%	64.06%
0.01	3	1	-0.310731	7233	63.03%	62.47%
0.01	3	2	-0.492699	7725	62.65%	62.37%
0.01	3	3	-0.706165	8073	62.25%	61.98%
0.01	3	4	-0.856294	8365	61.17%	61.03%
0.01	3	5	-1.030997	8601	60.27%	60.22%
1	5	1	-4.004053	7218	73.90%	72.45%
1	5	2	-4.995607	7595	73.49%	71.83%
1	5	3	-6.998321	7937	76.70%	75.34%
1	5	4	-5.995706	8184	70.59%	69.24%
1	5	5	-6.001567	8486	70.95%	70.03%
0.1	5	1	-0.8951	9287	70.34%	69.76%
0.1	5	2	-1.29919	10137	66.83%	66.14%
0.1	5	3	-1.615769	10594	64.93%	64.04%
0.1	5	4	-1.988379	10900	65.37%	64.57%
0.1	5	5	-2.405718	11100	64.93%	64.30%
0.01	5	1	-0.377825	11705	63.95%	63.29%
0.01	5	2	-0.596443	12360	63.61%	63.29%
0.01	5	3	-0.800907	12797	63.76%	63.45%
0.01	5	4	-0.966331	13184	62.89%	62.62%
0.01	5	5	-1.112367	13507	62.33%	61.99%

Observation: For the simple perceptron best accuracies are seen in both theb training data and the test data for learning rates 1,0.1 in epochs of 5.
For the margin perceptron best accuracies are seen in learning rates of 1,0.1 in epochs of 3.

Next, we perform these batch experiments with shuffling of the training data in every epoch. Table 6 and Table 7 shows the experimental results for the batch experiments with data shuffling for simple perceptron and margin perceptron respectively.

Table 6: Simple Perceptron Experiment-Batch- With Shuffle

LeraningRate	Epochs	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
1	3	-1.980228	4104	69.03%	68.54%
0.1	3	-0.389109	4102	82.78%	82.49%
0.01	3	-0.042181	4103	77.75%	78.18%
1	4	-3.024821	5471	82.85%	81.95%
0.1	4	-0.319061	5483	80.27%	80.08%
0.01	4	-0.025665	5399	80.85%	80.80%
1	5	-5.002354	6713	81.63%	81.56%
0.1	5	-0.292741	6751	77.76%	76.82%
0.01	5	-0.030401	6727	81.88%	82.65%

Table 7: Margin Perceptron Experiment-Batch- With Shuffle

LeraningRate	Epochs	Margin	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
1	3	1	-4.005854	4384	78.42%	77.38%
1	3	2	-4.009058	4590	69.97%	68.99%
1	3	3	-5.998597	4792	78.03%	77.26%
1	3	4	-5.010376	4945	70.47%	69.22%
1	3	5	-6.012511	5144	73.47%	72.29%
0.1	3	1	-0.903450	5659	71.17%	70.46%
0.1	3	2	-1.106591	6165	66.27%	65.58%
0.1	3	3	-1.522468	6475	65.07%	64.27%
0.1	3	4	-1.900346	6667	65.18%	64.54%
0.1	3	5	-2.101776	6807	64.60%	64.06%
0.01	3	1	-0.318264	7233	63.03%	62.47%
0.01	3	2	-0.496225	7725	62.65%	62.37%
0.01	3	3	-0.695790	8077	62.20%	61.96%
0.01	3	4	-0.854892	8365	61.17%	61.03%
0.01	3	5	-1.037856	8604	60.67%	60.64%
1	5	1	-4.997682	7203	78.35%	78.00%
1	5	2	-4.997048	7595	73.49%	71.83%
1	5	3	-6.001237	7898	75.64%	74.58%
1	5	4	-6.004164	8198	74.07%	73.07%
1	5	5	-7.996589	8438	73.46%	72.18%
0.1	5	1	-0.887182	9287	70.34%	69.76%
0.1	5	2	-1.211206	10130	65.37%	64.30%
0.1	5	3	-1.602849	10594	64.93%	64.04%
0.1	5	4	-1.992592	10900	65.37%	64.57%
0.1	5	5	-2.391255	11110	65.21%	64.64%
0.01	5	1	-0.371951	11705	63.95%	63.29%
0.01	5	2	-0.570455	12354	63.37%	63.09%
0.01	5	3	-0.788071	12798	63.53%	63.20%

0.01	5	4	-0.965035	13184	62.89%	62.62%
0.01	5	5	-1.132438	13511	62.51%	62.21%

The simple perceptron still gives better results , that is argin 0, seems most suited for this set of training and test data. For higher margins and lower learning rates the accuracy performance drops off towards the lower 60's.

(4-The Grads) In this experiment we model the aggressive perceptron, where the learning rate is adjusted everytime a istake is made, the goal of this modification being to find the smallest change in weights so that the current example is on the correct side of the weight vector. Table 8 and 9 shows the experimental findings for the aggressive perceptron on batch settings of epoch 3,5 with and without shuffling of data respectively.

Table 8: Aggressive Perceptron Experiment-Batch- no Shuffle

Epochs	Margin	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
3	0	-0.009682	4424	62.98%	62.44%
3	1	-0.537700	7913	81.74%	82.00%
3	2	-1.080245	7913	81.74%	82.00%
3	3	-1.161445	7913	81.74%	82.00%
3	4	-2.163234	7913	81.74%	82.00%
3	5	-2.707325	7913	81.74%	82.00%
5	0	-0.001494	7471	79.24%	78.22%
5	1	-0.556418	13064	81.86%	82.02%
5	2	-1.133336	13064	81.86%	82.02%
5	3	-1.687009	13064	81.86%	82.02%
5	4	-2.257658	13064	81.86%	82.02%
5	5	-2.800719	13064	81.86%	82.02%

Table 9: Aggressive Perceptron Experiment-Batch- With Shuffle

Epochs	Margin	Bias	Mistakes-During-Learning	LearningAccuracy	Test-Accuracy
3	0	-0.001732	4483	77.31%	75.89%
3	1	-0.518594	7758	80.57%	81.10%
3	2	-0.363459	7839	83.45%	82.82%
3	3	-2.147180	7850	76.90%	77.57%
3	4	-1.963089	7821	77.08%	77.36%
3	5	-2.568195	7842	82.64%	82.71%
5	0	-0.007758	7303	80.04%	79.06%
5	1	-0.498756	12810	80.52%	81.28%
5	2	-1.092983	12967	81.32%	81.62%
5	3	-0.989864	12954	82.56%	81.22%
5	4	-2.466388	12894	82.11%	82.23%
5	5	-2.605096	12936	82.33%	82.19%

Observation: The aggressive perceptron gives more consistent results since we are dynamically tuning the learning rate for every mistake it makes in the prediction. As table-8 and table-9 suggests, the accuracy on the test data is roughly stable around $\approx 82\%$ both with and without shuffle.

Thats all for now, please check out the code !!

Notes about the coding: Feature extraction is done completely from the training data, and there is no assumption about how many features the data contains. Any feature not present in the training data, will have zero weights in the final weight vector and not random weights. As the training data parsing is done, list extension is done based on the max feature found till that point and the number of unseen features for that size is also tracked. In this manner, feature extraction is taken care of.

Following is the template for perceptron:

Perceptron(lr, initEn, xdata, ydata, wsize, epochs, margin, enShuffle, seenwt)
lr = learning rate, initEn = Enable Initialization, xdata = accumulated vectors of input,
ydata = output labels corresponding to xdata, epochs = number of epochs(for single pass this is fixed to 1)
margin = margin values(for simple perceptron this is fixed to 0), enShuffle = to enable data shuffling,
seenwt = list of seen features(used to set the unseen features to 0 weights)

The aggressive perceptron has the same templeate except the learning rate, since its learning rate is derived dynamically.