Comp9417 Final Exam

Question 1:

a)

At first, we assume A, B, C and D are mutually independent.

According to the Bayes Assumption, we have:

$$p(C|\boldsymbol{X}) = \frac{p(C) * p(\boldsymbol{X}|C)}{p(\boldsymbol{X})}$$
$$= \frac{p(C) * \prod_{i=1}^{4} p(X_i|C)}{p(\boldsymbol{X})}.$$

From the dataset we know that

$$p(C = +) = 0.5.$$

And the count vector for positive is(5,3,9,6) and that for negative is(11,3,0,2).

Then the estimated parameter vectors are:

$$\hat{\theta}^{+} = \left(\frac{5}{23}, \frac{3}{23}, \frac{9}{23}, \frac{6}{23}\right)$$
$$\hat{\theta}^{-} = \left(\frac{11}{16}, \frac{3}{16}, 0, \frac{2}{16}\right).$$

According to the formula given:

$$P(\boldsymbol{X} = (1,0,0,2)| +) = \frac{(3)!}{1! * 0! * 0! * 2!} * \frac{5}{23}^{1} * \frac{3}{23}^{0} * \frac{9}{23}^{0} * \frac{6}{23}^{2} = 0.04438$$
$$\approx 0.0444(4\ decimal)$$

$$P\big(\boldsymbol{X} = (1,0,0,2)\big) = \frac{(3)!}{1! * 0! * 0! * 2!} * \frac{16}{39}^{1} * \frac{6}{39}^{0} * \frac{9}{39}^{0} * \frac{8}{39}^{2} = 0.05179$$
$$\approx 0.0518(4\ decimal).$$

Thus,

$$P(+|\boldsymbol{X}) = \frac{0.5 * 0.0444}{0.05179} = 0.42865 \approx 0.04287.$$

b)

With smooth, the estimated parameter vectors are:

$$\hat{\theta}^{+} = \left(\frac{6}{27}, \frac{4}{27}, \frac{10}{27}, \frac{7}{27}\right)$$
$$\hat{\theta}^{-} = \left(\frac{12}{20}, \frac{4}{20}, \frac{1}{20}, \frac{3}{20}\right).$$

Hence,

$$P(\boldsymbol{X}=(1,0,0,2)|+)=\frac{(3)!}{1!*0!*0!*2!}*\frac{6}{27}^{1}*\frac{4}{27}^{0}*\frac{10}{27}^{0}*\frac{7}{27}^{2}=0.04481\approx 0.0448$$

$$P(\boldsymbol{X}=(1,0,0,2)|-)=\frac{(3)!}{1!*0!*0!*2!}*\frac{6}{27}^{1}*\frac{4}{27}^{0}*\frac{10}{27}^{0}*\frac{7}{27}^{2}=0.0405$$

$$P\left(-|\boldsymbol{X}=(1,0,0,2)\right)=\frac{0.0405}{0.0405+0.0448}=0.4747.$$

c)

Under the Bernoulli distribution, the estimated parameter vectors are:

$$\hat{\theta}^{+}=\left(\frac{2}{8},\frac{1}{8},\frac{3}{8},\frac{2}{8}\right)$$

$$\hat{\theta}^{-}=\left(\frac{3}{6},\frac{1}{6},0,\frac{2}{6}\right).$$

And we have the bit vector $\boldsymbol{X}=(1,0,0,1)$

Thus,

$$P(\boldsymbol{X}=(1,0,0,1)|+)=\frac{2}{8}*\left(1-\frac{1}{8}\right)*\left(1-\frac{3}{8}\right)*\frac{2}{8}=\frac{2}{8}*\frac{7}{8}*\frac{5}{8}*\frac{2}{8}=0.03418\approx 0.0342$$

$$P(\boldsymbol{X}=(1,0,0,1)|-)=\frac{3}{6}*\left(1-\frac{1}{6}\right)*(1-0)*\frac{2}{6}=\frac{3}{6}*\frac{5}{6}*1*\frac{2}{6}=0.034188\approx 0.1389.$$

Therefore,

$$P(+|\boldsymbol{X})=\frac{0.0342}{0.0342+0.1389}=0.19757\approx 0.1976$$

d)

With smooth, the estimated parameter vectors are:

$$\hat{\theta}^{+}=\left(\frac{3}{8},\frac{2}{8},\frac{4}{8},\frac{3}{8}\right)$$

$$\hat{\theta}^{-}=\left(\frac{4}{6},\frac{2}{6},\frac{1}{6},\frac{3}{6}\right).$$

And we have the bit vector $\boldsymbol{X}=(1,0,0,1)$

Thus,

$$P(\boldsymbol{X}=(1,0,0,1)|+)=\frac{3}{8}*\left(1-\frac{2}{8}\right)*\left(1-\frac{4}{8}\right)*\frac{3}{8}=\frac{3}{8}*\frac{6}{8}*\frac{4}{8}*\frac{3}{8}=0.05273\approx 0.0527$$

$$P(\boldsymbol{X}=(1,0,0,1)|-)=\frac{4}{6}*\left(1-\frac{2}{6}\right)*\left(1-\frac{1}{6}\right)*\frac{3}{6}=\frac{4}{6}*\frac{4}{6}*\frac{5}{6}*\frac{3}{6}=0.18518\approx 0.1852.$$

Therefore,

$$P(-|\boldsymbol{X})=\frac{0.1852}{0.0527+0.1852}=0.77847\approx 0.7785.$$

Question 2

a)

According to the question, $w^T X_i = w_0 + w_1 x_i$, thus

$$\mathrm{L_c}(y_i, \widehat{y_i}) = \sqrt{\frac{1}{c^2}(y - \hat{y})^2 + 1} - 1 = \sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1} - 1,$$

Then we have:

$$\frac{\partial L_c(y_i, \widehat{y_i})}{\partial w_0} = \frac{\partial}{\partial w_0}\left[\sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1} - 1\right]$$
$$= \frac{\partial}{\partial w_0}\sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1}$$
$$= \frac{1}{2}\left(\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1\right)^{-\frac{1}{2}} \cdot \frac{2}{c^2}(y_i - w_0 - w_1 x_i) \cdot -1$$
$$= -\frac{(y_i - w_0 - w_1 x_i)}{c^2\sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1}}$$
$$= -\frac{(y_i - w_0 - w_1 x_i)}{c\sqrt{(y_i - w_0 - w_1 x_i)^2 + c^2}}$$

and

$$\frac{\partial L_c(y_i, \widehat{y_i})}{\partial w_1} = \frac{\partial}{\partial w_1}\left[\sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1} - 1\right]$$
$$= \frac{\partial}{\partial w_0}\sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1}$$
$$= \frac{1}{2}\left(\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1\right)^{-\frac{1}{2}} \cdot \frac{2}{c^2}(y_i - w_0 - w_1 x_i) \cdot -x_i$$
$$= -\frac{x_i(y_i - w_0 - w_1 x_i)}{c^2\sqrt{\frac{1}{c^2}(y_i - w_0 - w_1 x_i)^2 + 1}}$$
$$= -\frac{x_i(y_i - w_0 - w_1 x_i)}{c\sqrt{(y_i - w_0 - w_1 x_i)^2 + c^2}}.$$

b)

While condition satisfied (iteration < 100 in this question):

BEGIN

Compute $\nabla L_c(\boldsymbol{w}^t)$ by

$$\nabla L_c(w_0^t) = \frac{\partial \mathrm{L_c}(y,\hat{y})}{\partial \mathrm{w}_0^t} = \Sigma_{i=1}^{n}\left[-\frac{(y_i - w_0^t - w_1^t x_i)}{c\sqrt{(y_i - w_0^t - w_1^t x_i)^2 + c^2}}\right]$$

and

$$\nabla L_c(w_1^t) = \frac{\partial \mathrm{L_c}(y,\hat{y})}{\partial \mathrm{w}_1^t} = \Sigma_{i=1}^{n}\left[-\frac{x_i(y_i - w_1^t - w_1^t x_i)}{c\sqrt{(y_i - w_1^t - w_1^t x_i)^2 + c^2}}\right].$$

$$\nabla L_c(\boldsymbol{w}^t) = \nabla L_c([w_0^t, w_1^t])$$

Then calculate $\Delta \boldsymbol{w}$

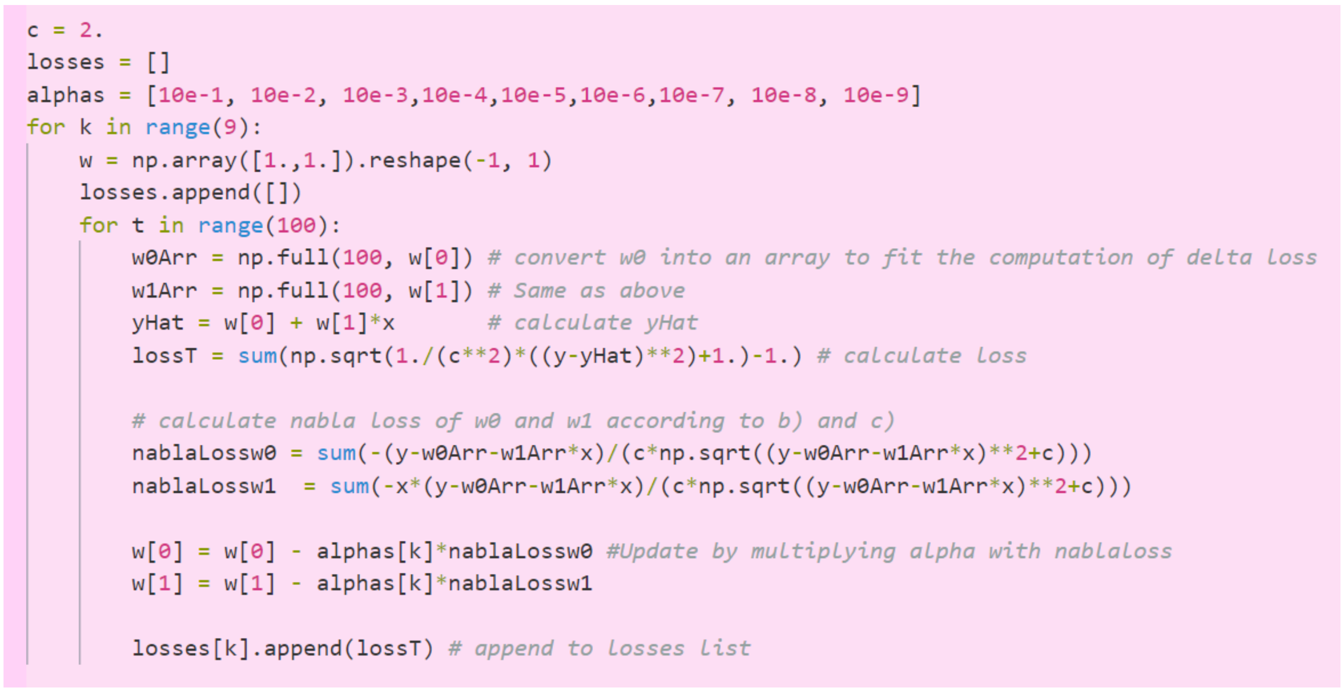
$$\Delta \boldsymbol{w} = \alpha \nabla L_c(\boldsymbol{w}^t)$$

Finally update $\boldsymbol{w}^{t+1}$

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t + \Delta \boldsymbol{w}$$

END

c)



```python
c = 2.
losses = []
alphas = [10e-1, 10e-2, 10e-3,10e-4,10e-5,10e-6,10e-7, 10e-8, 10e-9]
for k in range(9):
    w = np.array([1.,1.]).reshape(-1, 1)
    losses.append([])
    for t in range(100):
        w0Arr = np.full(100, w[0]) # convert w0 into an array to fit the computation of delta loss
        w1Arr = np.full(100, w[1]) # Same as above
        yHat = w[0] + w[1]*x        # calculate yHat
        lossT = sum(np.sqrt(1./(c**2)*((y-yHat)**2)+1.)-1.) # calculate loss

        # calculate nabla loss of w0 and w1 according to b) and c)
        nablaLossw0 = sum(-(y-w0Arr-w1Arr*x)/(c*np.sqrt((y-w0Arr-w1Arr*x)**2+c)))
        nablaLossw1  = sum(-x*(y-w0Arr-w1Arr*x)/(c*np.sqrt((y-w0Arr-w1Arr*x)**2+c)))

        w[0] = w[0] - alphas[k]*nablaLossw0 #Update by multiplying alpha with nablaloss
        w[1] = w[1] - alphas[k]*nablaLossw1

        losses[k].append(lossT) # append to losses list
```

Figure 2.1 Python code for gradient descent

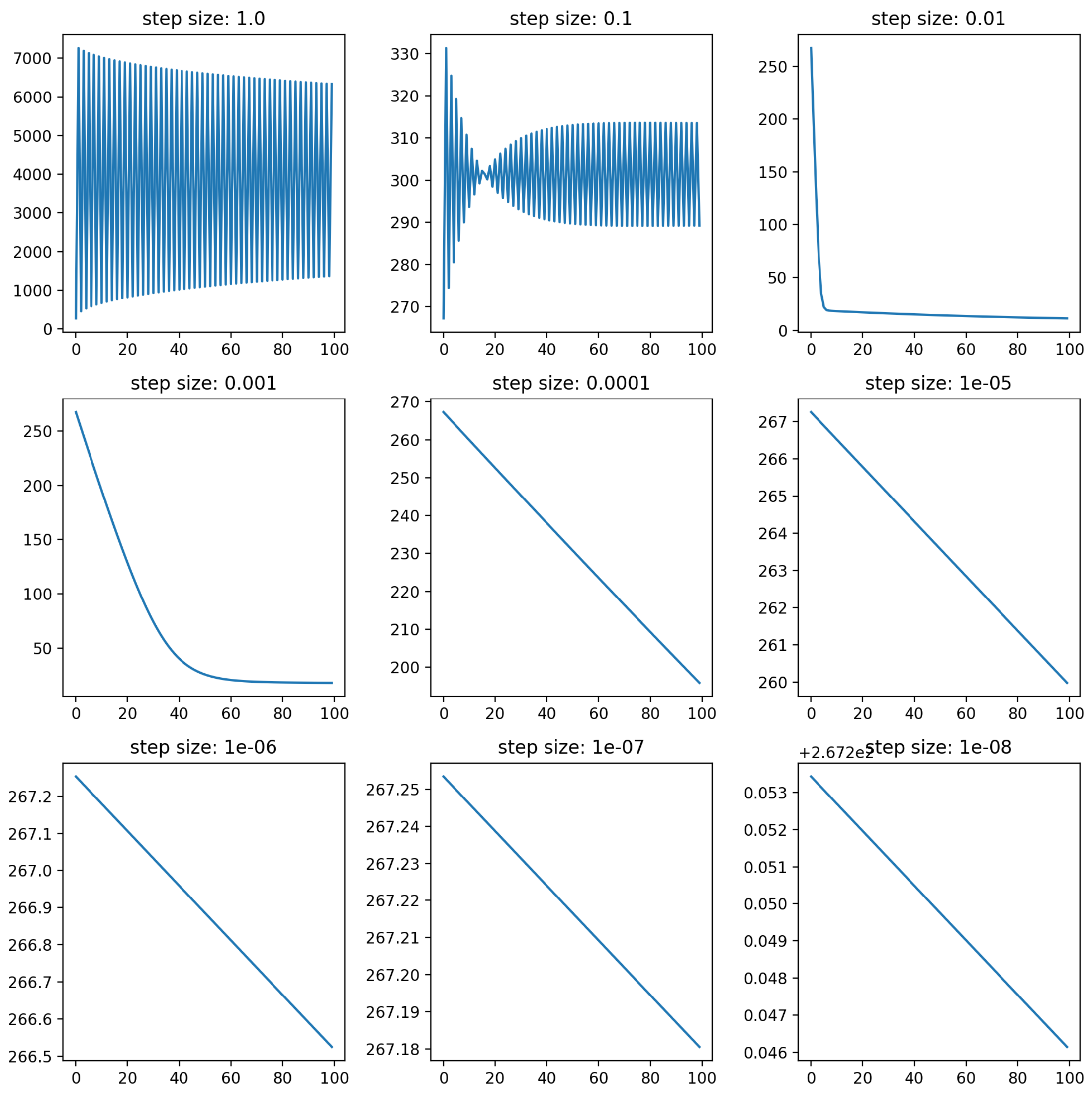


Figure 2.2 Gradient for nine different alphas

d)

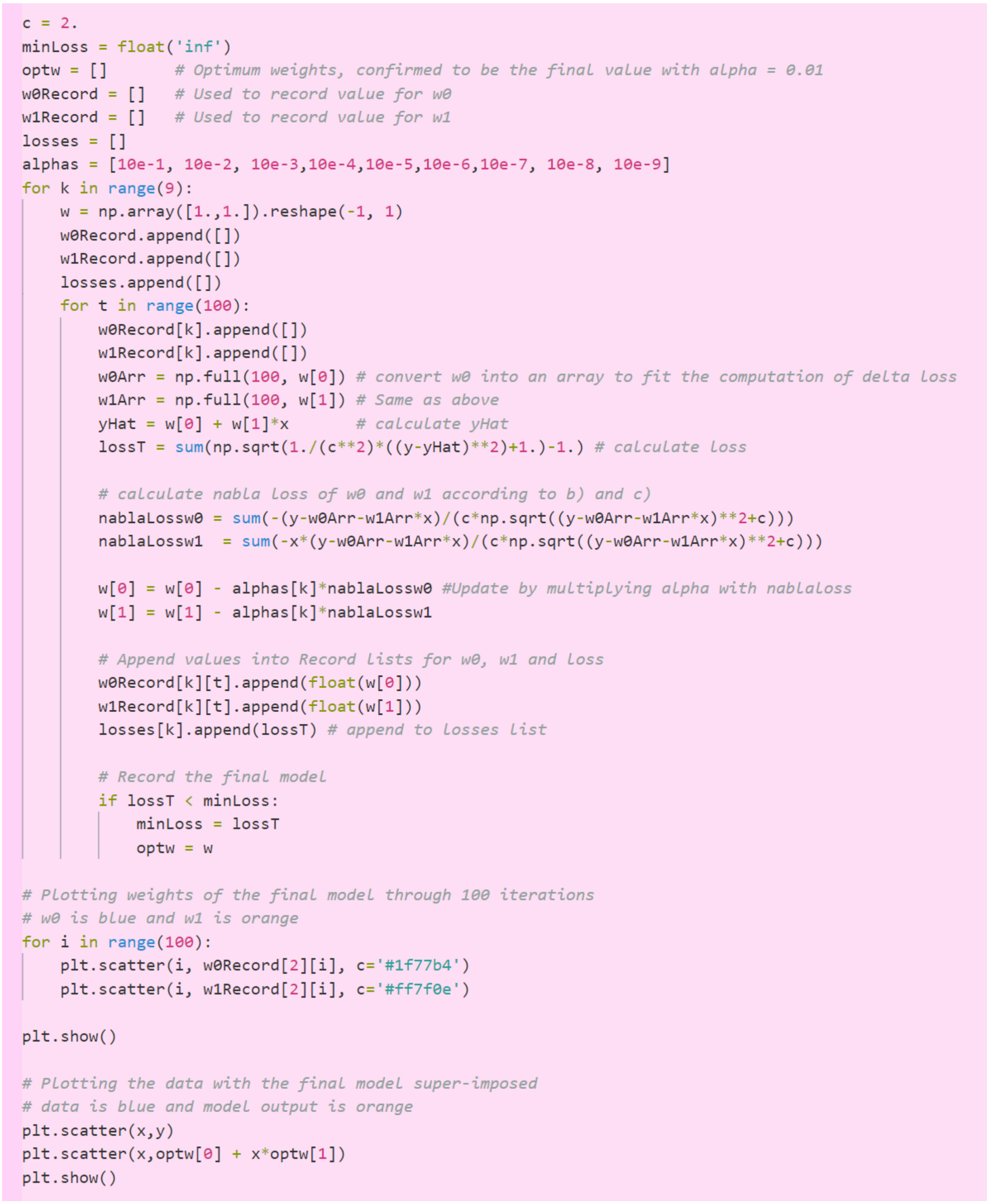
Oscillations appeared for step size of 1.0 and 0.1. And the loss decreased significantly when $a = 0.01\ and\ 0.001$ while the model performs best with $a = 0.01$. However, when step size became smaller, no obvious change happened to the loss.

The reason for the oscillation is that the learning rate is too high so that the parameters are over adjusted and when it becomes low no significant adjustment could be made. If we change iteration size to a large number (for example 10000) we can observe a significant decrease in losses of a smaller learning rate.

e)

The optimal step-size is 0.01, and the final model is

$$y = 0.7791 + 5.9713 * x$$



```python
c = 2.
minLoss = float('inf')
optw = []       # Optimum weights, confirmed to be the final value with alpha = 0.01
w0Record = []   # Used to record value for w0
w1Record = []   # Used to record value for w1
losses = []
alphas = [10e-1, 10e-2, 10e-3,10e-4,10e-5,10e-6,10e-7, 10e-8, 10e-9]
for k in range(9):
    w = np.array([1.,1.]).reshape(-1, 1)
    w0Record.append([])
    w1Record.append([])
    losses.append([])
    for t in range(100):
        w0Record[k].append([])
        w1Record[k].append([])
        w0Arr = np.full(100, w[0]) # convert w0 into an array to fit the computation of delta loss
        w1Arr = np.full(100, w[1]) # Same as above
        yHat = w[0] + w[1]*x        # calculate yHat
        lossT = sum(np.sqrt(1./(c**2)*((y-yHat)**2)+1.)-1.) # calculate loss

        # calculate nabla loss of w0 and w1 according to b) and c)
        nablaLossw0 = sum(-(y-w0Arr-w1Arr*x)/(c*np.sqrt((y-w0Arr-w1Arr*x)**2+c)))
        nablaLossw1  = sum(-x*(y-w0Arr-w1Arr*x)/(c*np.sqrt((y-w0Arr-w1Arr*x)**2+c)))

        w[0] = w[0] - alphas[k]*nablaLossw0 #Update by multiplying alpha with nablaloss
        w[1] = w[1] - alphas[k]*nablaLossw1

        # Append values into Record lists for w0, w1 and loss
        w0Record[k][t].append(float(w[0]))
        w1Record[k][t].append(float(w[1]))
        losses[k].append(lossT) # append to losses list

        # Record the final model
        if lossT < minLoss:
            minLoss = lossT
            optw = w

# Plotting weights of the final model through 100 iterations
# w0 is blue and w1 is orange
for i in range(100):
    plt.scatter(i, w0Record[2][i], c='#1f77b4')
    plt.scatter(i, w1Record[2][i], c='#ff7f0e')

plt.show()

# Plotting the data with the final model super-imposed
# data is blue and model output is orange
plt.scatter(x,y)
plt.scatter(x,optw[0] + x*optw[1])
plt.show()
```

Figure 2.3 Code with two plots

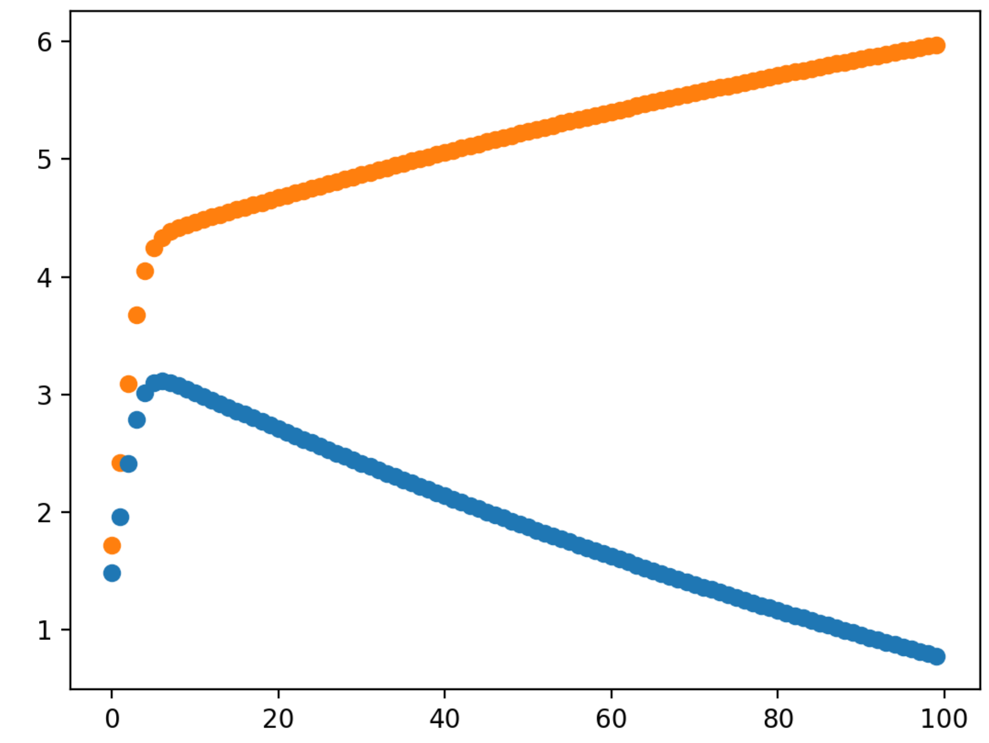


Figure 2.4 Plot of final model for w0 and w1 over 100 iterations (w0: blue, w1: orange).

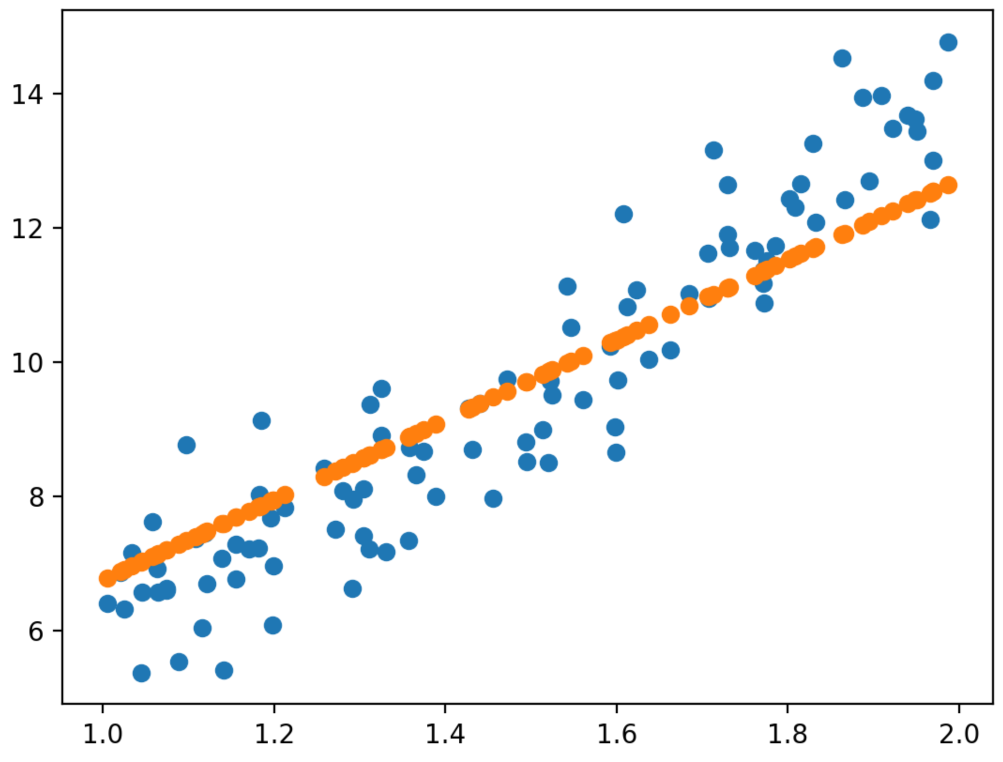


Figure 2.5 Plot of the data with the final model super-imposed (data: blue, predict: orange).

f)

The value for optimal alpha value increases as c increases, vice versa.

A larger value of c is chosen when we want to reduce the influence of the squared error and reduce the numerical value of SE, vice versa.

The final c chosen should generate a more accurate model and should also produce a more intuitive loss value for human brain.

Question 3

a)

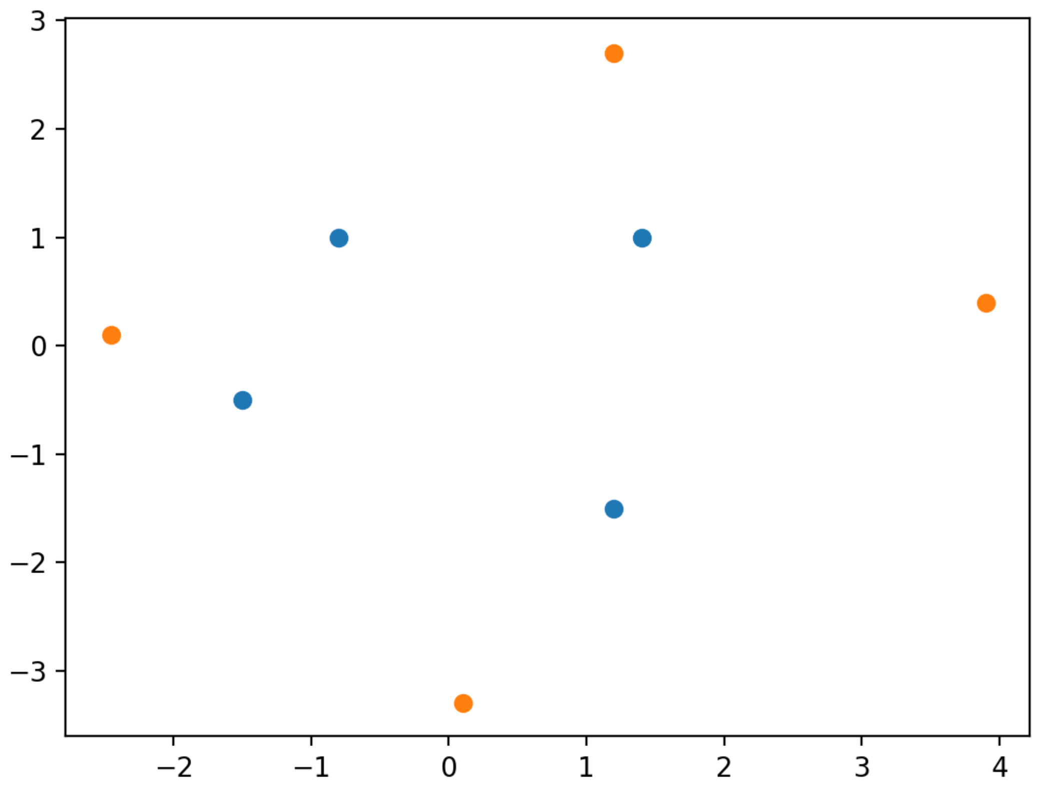


Figure 3.1 plot of data (blue: data for y=1, orange: data for y=-1)

The simplest polynomial kernel is

$$K(x_i, x_j) = (x_1 \cdot x_2)^2,$$

in which $m = 0$ and $d = 2$.

So, the feature function could be defined as

$$\phi(x_1, x_2) = \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix}.$$

b)

According to a), the feature representation is in $\mathbb{R}^3$.

And the subset chosen is $\begin{pmatrix} x_1^2 \\ x_2^2 \end{pmatrix}$. The plot is shown in the following figure, and the transformed data is linear separable

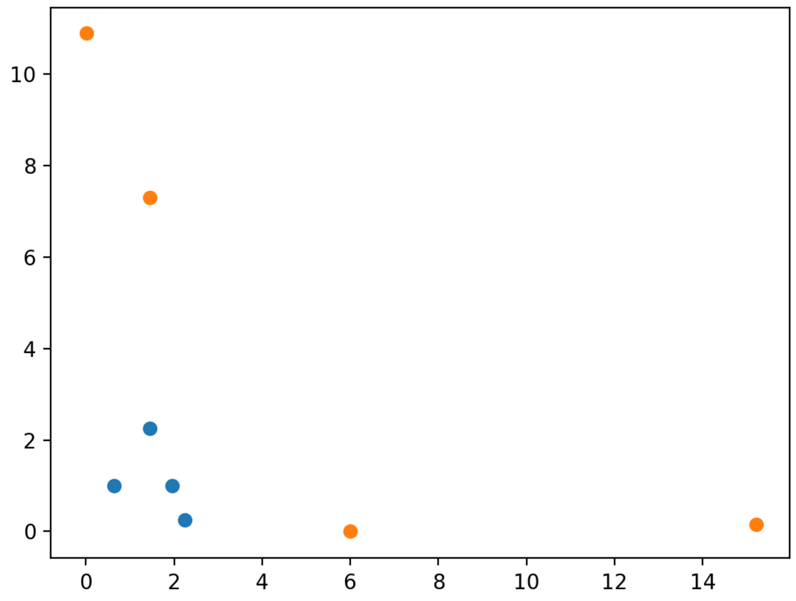


Figure 3.2 plot of the subset of the 3-dimensional vector (blue: data for y=1, orange: data for y=-1)

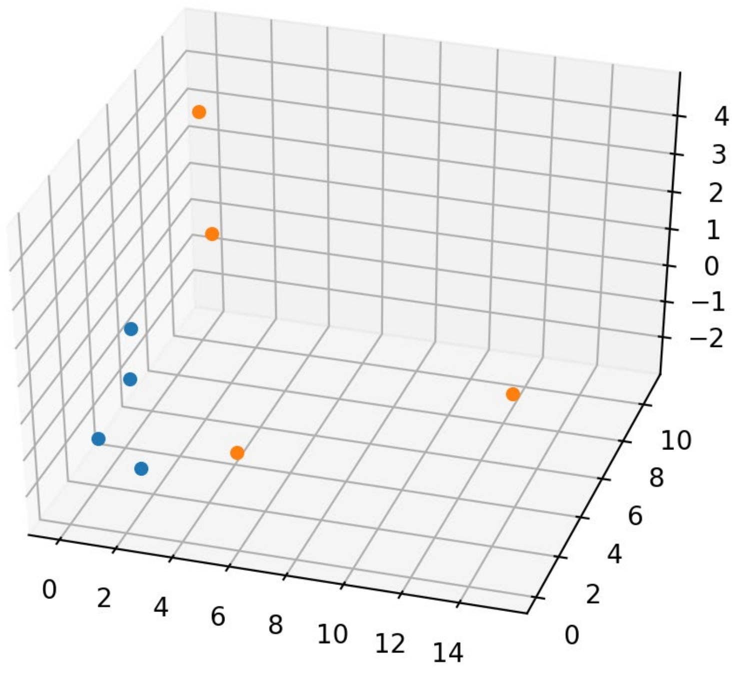


Figure 3.3 plot of transformed data (blue: data for y=1, orange: data for y=-1)

c)

The table below is the table outlining all updates of the weight vector with the iteration. Notice that the training process terminates after 72 iterations and the weight vector is

$$\boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} \begin{pmatrix} 3 \\ -0.62 \\ -0.654 \\ 0.07793 \end{pmatrix}$$

Iteration No.	w_0	w_1	w_2	w_3
0	1	1	1	1
1	1	1	1	1
2	0.8	-2.042	0.968	0.558765
3	1	-1.65	1.168	0.954745
4	0.8	-1.652	-1.01	1.048083
5	0.8	-1.652	-1.01	1.048083
6	0.8	-1.652	-1.01	1.048083
7	1	-1.202	-0.96	1.260215
8	1.2	-0.914	-0.51	0.751098
9	1.4	-0.786	-0.31	0.524824
10	1.4	-0.786	-0.31	0.524824
11	1.4	-0.786	-0.31	0.524824
12	1.4	-0.786	-0.31	0.524824
13	1.2	-1.074	-1.768	-0.39159
14	1.2	-1.074	-1.768	-0.39159
15	1.4	-0.624	-1.718	-0.17945
16	1.6	-0.336	-1.268	-0.68857
17	1.6	-0.336	-1.268	-0.68857
18	1.6	-0.336	-1.268	-0.68857
19	1.8	0.056	-1.068	-0.29259
20	1.8	0.056	-1.068	-0.29259
21	1.8	0.056	-1.068	-0.29259
22	1.6	-1.1445	-1.07	-0.22329
23	1.8	-0.6945	-1.02	-0.01116
24	2	-0.4065	-0.57	-0.52028
25	2	-0.4065	-0.57	-0.52028
26	2	-0.4065	-0.57	-0.52028
27	2.2	-0.0145	-0.37	-0.1243
28	2.2	-0.0145	-0.37	-0.1243
29	2.2	-0.0145	-0.37	-0.1243
30	2	-1.215	-0.372	-0.055
31	2.2	-0.765	-0.322	0.157129
32	2.4	-0.477	0.128	-0.35199
33	2.4	-0.477	0.128	-0.35199
34	2.4	-0.477	0.128	-0.35199
35	2.4	-0.477	0.128	-0.35199

36	2.2	-0.479	-2.05	-0.25865
37	2.2	-0.479	-2.05	-0.25865
38	2.2	-0.479	-2.05	-0.25865
39	2.2	-0.479	-2.05	-0.25865
40	2.4	-0.191	-1.6	-0.76777
41	2.4	-0.191	-1.6	-0.76777
42	2.4	-0.191	-1.6	-0.76777
43	2.6	0.201	-1.4	-0.37179
44	2.6	0.201	-1.4	-0.37179
45	2.6	0.201	-1.4	-0.37179
46	2.4	-0.9995	-1.402	-0.30249
47	2.6	-0.5495	-1.352	-0.09036
48	2.8	-0.2615	-0.902	-0.59948
49	2.8	-0.2615	-0.902	-0.59948
50	2.8	-0.2615	-0.902	-0.59948
51	2.8	-0.2615	-0.902	-0.59948
52	2.8	-0.2615	-0.902	-0.59948
53	2.8	-0.2615	-0.902	-0.59948
54	2.6	-1.462	-0.904	-0.53018
55	2.8	-1.012	-0.854	-0.31805
56	2.8	-1.012	-0.854	-0.31805
57	2.8	-1.012	-0.854	-0.31805
58	2.8	-1.012	-0.854	-0.31805
59	3	-0.62	-0.654	0.077933
60	3	-0.62	-0.654	0.077933
61	3	-0.62	-0.654	0.077933
62	3	-0.62	-0.654	0.077933
63	3	-0.62	-0.654	0.077933
64	3	-0.62	-0.654	0.077933
65	3	-0.62	-0.654	0.077933
66	3	-0.62	-0.654	0.077933
67	3	-0.62	-0.654	0.077933
68	3	-0.62	-0.654	0.077933
69	3	-0.62	-0.654	0.077933
70	3	-0.62	-0.654	0.077933
71	3	-0.62	-0.654	0.077933
72	3	-0.62	-0.654	0.077933

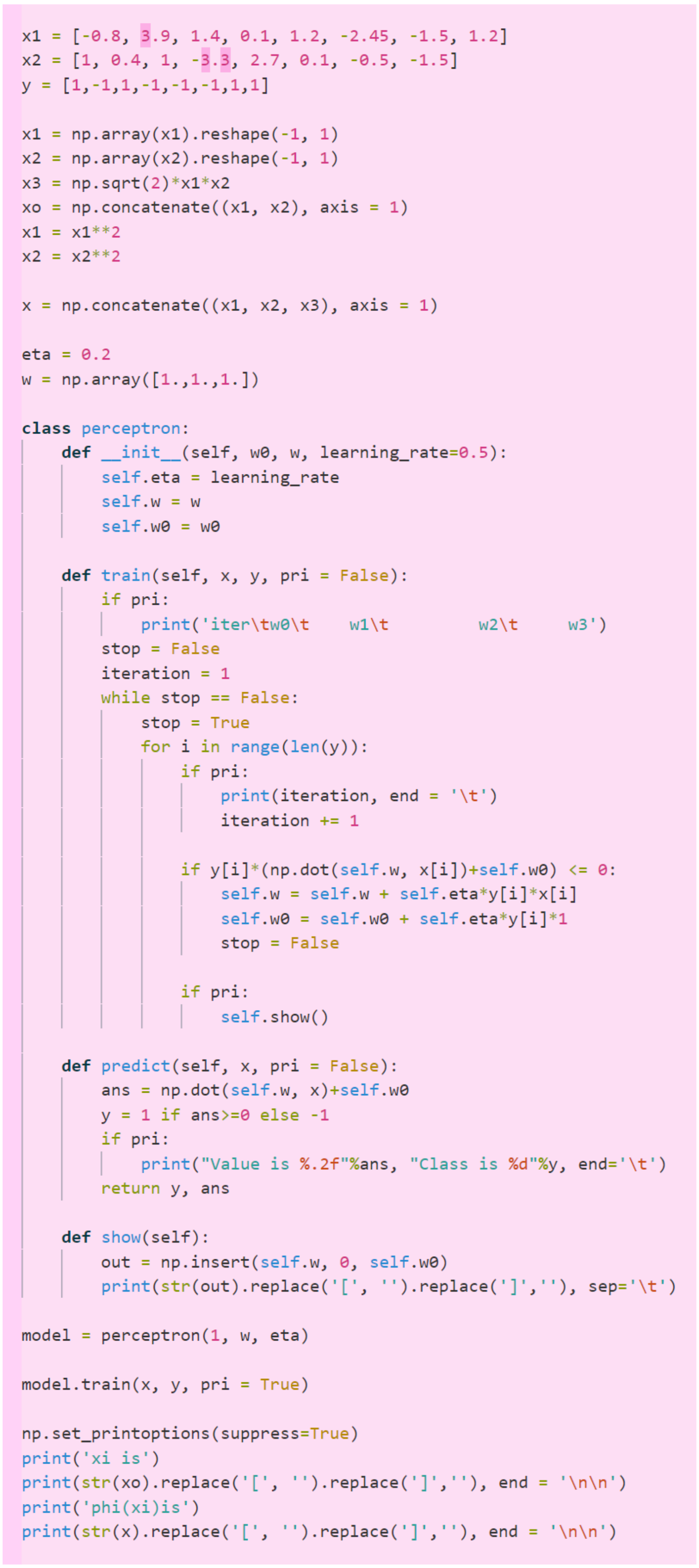
Figure 3.4 table for all updates of weight vector

The following table demonstrates the correcteness:

x_i^T	$\phi(x_i)^T$	$y_i\phi^T(x_i)w^*$	positiveness
(-0.8 1.)	(0.64 1. -1.13137085)	1.86102915	+
(3.9 0.4)	(15.21 0.16 2.20617316)	6.362906843	+
(1.4 1.)	(1.96 1. 1.97989899)	1.285098987	+
(0.1 -3.3)	(0.01 10.89 -0.46669048)	4.164630476	+
(1.2 2.7)	(1.44 7.29 4.58205194)	2.303368058	+
(-2.45 0.1)	(6.0025 0.01 -0.34648232)	0.755092323	+
(-1.5 -0.5)	(2.25 0.25 1.06066017)	1.524160172	+
(1.2 -1.5)	(1.44 2.25 -2.54558441)	0.437315588	+

Figure 3.5 table for correctness

Screen shot for code is in next page.



```
x1 = [-0.8, 3.9, 1.4, 0.1, 1.2, -2.45, -1.5, 1.2]
x2 = [1, 0.4, 1, -3.3, 2.7, 0.1, -0.5, -1.5]
y = [1,-1,1,-1,-1,-1,1,1]

x1 = np.array(x1).reshape(-1, 1)
x2 = np.array(x2).reshape(-1, 1)
x3 = np.sqrt(2)*x1*x2
xo = np.concatenate((x1, x2), axis = 1)
x1 = x1**2
x2 = x2**2

x = np.concatenate((x1, x2, x3), axis = 1)

eta = 0.2
w = np.array([1.,1.,1.])

class perceptron:
    def __init__(self, w0, w, learning_rate=0.5):
        self.eta = learning_rate
        self.w = w
        self.w0 = w0

    def train(self, x, y, pri = False):
        if pri:
            print('iter\tw0\t    w1\t         w2\t     w3')
        stop = False
        iteration = 1
        while stop == False:
            stop = True
            for i in range(len(y)):
                if pri:
                    print(iteration, end = '\t')
                    iteration += 1

                if y[i]*(np.dot(self.w, x[i])+self.w0) <= 0:
                    self.w = self.w + self.eta*y[i]*x[i]
                    self.w0 = self.w0 + self.eta*y[i]*1
                    stop = False

                if pri:
                    self.show()

    def predict(self, x, pri = False):
        ans = np.dot(self.w, x)+self.w0
        y = 1 if ans>=0 else -1
        if pri:
            print("Value is %.2f"%ans, "Class is %d"%y, end='\t')
        return y, ans

    def show(self):
        out = np.insert(self.w, 0, self.w0)
        print(str(out).replace('[', '').replace(']',''), sep='\t')

model = perceptron(1, w, eta)

model.train(x, y, pri = True)

np.set_printoptions(suppress=True)
print('xi is')
print(str(xo).replace('[', '').replace(']',''), end = '\n\n')
print('phi(xi)is')
print(str(x).replace('[', '').replace(']',''), end = '\n\n')
```

Figure3.6 Code for this question

d)

From the question, we have:

$$\begin{aligned} k(x,y) = (2x^Ty+3)^3 &= (2x_1y_1+2x_2y_2+3)^3 \\ &= 8x_1^3y_1^3 + 36x_1^2y_1^2 + 24x_1^2y_1^2x_2y_2 + 54x_1y_1 + 24x_1y_1x_2^2y_2^2 \\ &+ 72x_1y_1x_2y_2 + 8x_2^3y_2^3 + 36x_2^2y_2^2 + 54x_2y_2 + 27 = \phi(x)^T\phi(y). \end{aligned}$$

Therefore,

$$\phi(\boldsymbol{x}) = \begin{pmatrix} 2\sqrt{2}x_1^3 \\ 6x_1^2 \\ 2\sqrt{6}x_1^2x_2 \\ 3\sqrt{6}x_1 \\ 2\sqrt{6}x_1x_2^2 \\ 6\sqrt{2}x_1x_2 \\ 2\sqrt{2}x^3 \\ 6x_2^2 \\ 3\sqrt{6}x_2 \\ 3\sqrt{3} \end{pmatrix}$$

e)

The claim is true.

Because the input vectors for positive class and negative class are symmetric along a straight line $y = -x$ and there are four points for each class.

From the symmetrical characteristic, we can state that when the model correctly classify ith data point for the positive class, it means that it can correctly classify ith data for the negative model. For example, $\boldsymbol{x}$ is classified as positive which means that

$$y = 1$$

$$1 * \boldsymbol{w} \cdot \boldsymbol{x} > 0$$

Then we have

$$-1 * \boldsymbol{w} \cdot -\boldsymbol{x} > 0$$

which means that when $\boldsymbol{x}' = -\boldsymbol{x}$, we have $y = -1$.

Thus, if $(2,0)$ is classified as positive, $(-2,0)$ must be classified as negative by the same model.

From the algorithm we can know that after a mistake, the weight would be adjusted and the model would be correctly classified then. And, when a data point is correctly classified, no change would be made.

Therefore, after 4 mistakes, the remaining data would not make mistakes anymore.

Question 4

a)

The data is linearly separable

b)

According to the question, we have:

$$\boldsymbol{x} = \begin{pmatrix} -3 & 9 \\ -2 & 4 \\ -1 & 1 \\ 0 & -3 \\ 1 & 1 \\ 2 & 4 \\ 3 & 9 \end{pmatrix}$$

$$y = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \end{pmatrix}$$

And we have

$$(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7) = argmax\left(\Sigma_{i=1}^{n}\alpha_i - \frac{1}{2}\Sigma_{i=1}^{n}\Sigma_{i=1}^{n}\alpha_i\alpha_j y_i y_j x_i \cdot x_j\right)$$

Matrix multiplication for a 7-dimensional matrix is impossible for hand writing, so we must ellipses some of the points.

By observation, the line separating x_3, x_5, x_6 could separate the whole dataset appropriately. So, we calculate α by those samples.

Thus, the new x and y are:

$$X = \begin{pmatrix} -1 & 1 \\ 1 & 1 \\ 2 & 4 \end{pmatrix}$$

$$y = \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}$$

Then calculate the helper matrices:

$$X' = \begin{pmatrix} 1 & -1 \\ 1 & 1 \\ -2 & -4 \end{pmatrix}$$

$$X'^T = \begin{pmatrix} 1 & 1 & -2 \\ -1 & 1 & -4 \end{pmatrix}$$

Hence,

$$X'X'^T = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 2 & -6 \\ 2 & -6 & 20 \end{pmatrix}$$

Then, we have

$$-\frac{1}{2}\left(2a_3^2+2a_5^2+20a_6^2+4a_3a_6-12a_5a_6\right)+a_3+a_5+a_6$$

From$\Sigma_{i=1}^{n} a_i y_i = 0$, we can derive that

$$-a_3+a_5+-a_6=0$$

$$a_5=a_3+a_6$$

Substitute a_5 by a_3+a_6 we get:

$$-\frac{1}{2}(4a_3^2+10a_6^2-4a_3a_6)+2a_3+2a_6=-2a_3^2-5a_6^2+2a_3a_6+2a_3+2a_6$$

The partial derivatives are:

$$\frac{\partial}{\partial a_3}=-4a_3+2a_6+2=0$$

$$\frac{\partial}{\partial a_6}=-10a_6+2a_3+2=0$$

Solve to get

$$a_3=\frac{2}{3}$$

$$a_6=\frac{1}{3}$$

$$a_5=1$$

while $a_1=a_2=a_4=a_7=0$

Thus

$$(\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5,\alpha_6,\alpha_7)=(0,0,\frac{2}{3},0,1,\frac{1}{3},0)$$

c)

To calculate **w**:

$$\boldsymbol{w}=\Sigma_{x\in(supportvectors)}\mathrm{a_i}y_i x_i$$

According to b), a_3, a_5, a_6 are support vectors.

$$\boldsymbol{w}=\frac{2}{3}*-1*\binom{-1}{1}+1*1*\binom{1}{1}+\frac{1}{3}*-1*\binom{2}{4}=\binom{1}{-1}$$

Solve t:

$$y_i(w\cdot x_i-t)=1$$

$$1\left((1\quad -1)\binom{1}{1}-t\right)=1$$

$$t=-1$$

The margin is

$$margin = \frac{1}{||\boldsymbol{w}||} = \frac{1}{\sqrt{1^2 + (-1)^2}} = \frac{1}{\sqrt{2}}$$

Use w and -t to predict by

$$y = wx - t$$

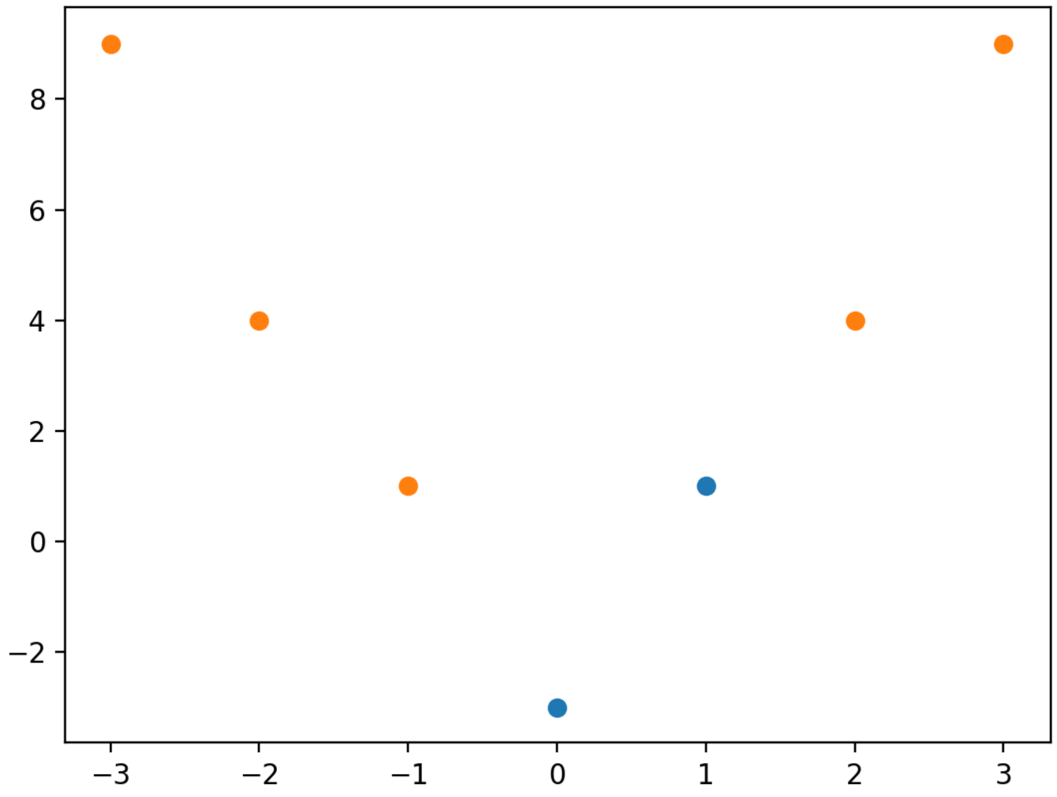


Figure 4.1 superimposed graph (blue for class=1, orange for class=-1)

From the figure above, we can conclude that the model perfectly fit the data.

d)

Without kernel, a linear classifier could never represent non-linear functions. But if appropriate kernel applied, the classifier then becomes linear function of polynomials of input data, then the functions may be represented.

e)

Kernel trick applies the concept of kernel without calculating feature vectors using $\phi(x)$. In contrast, it calculates the similarity of input data and output data by substitution. The key idea is to use linear kernel calculation to replace high dimension calculations.

Question 5

a)

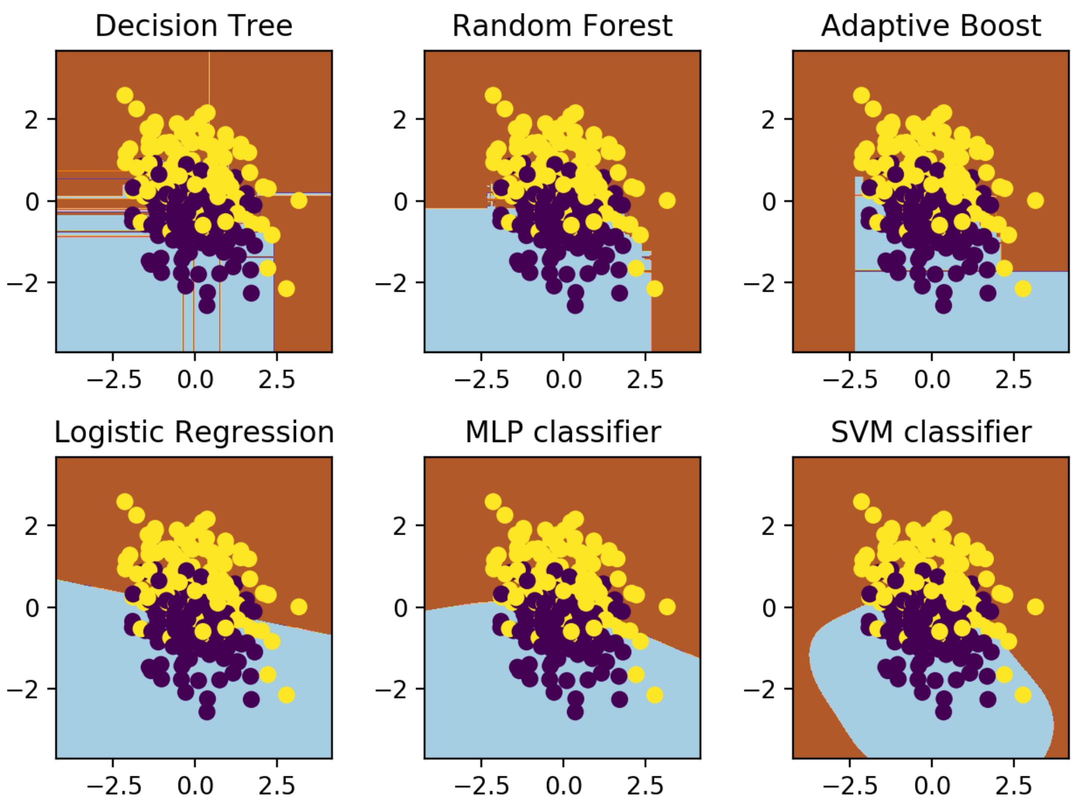


Figure 5.1 Decision boundary for different classifiers

b)

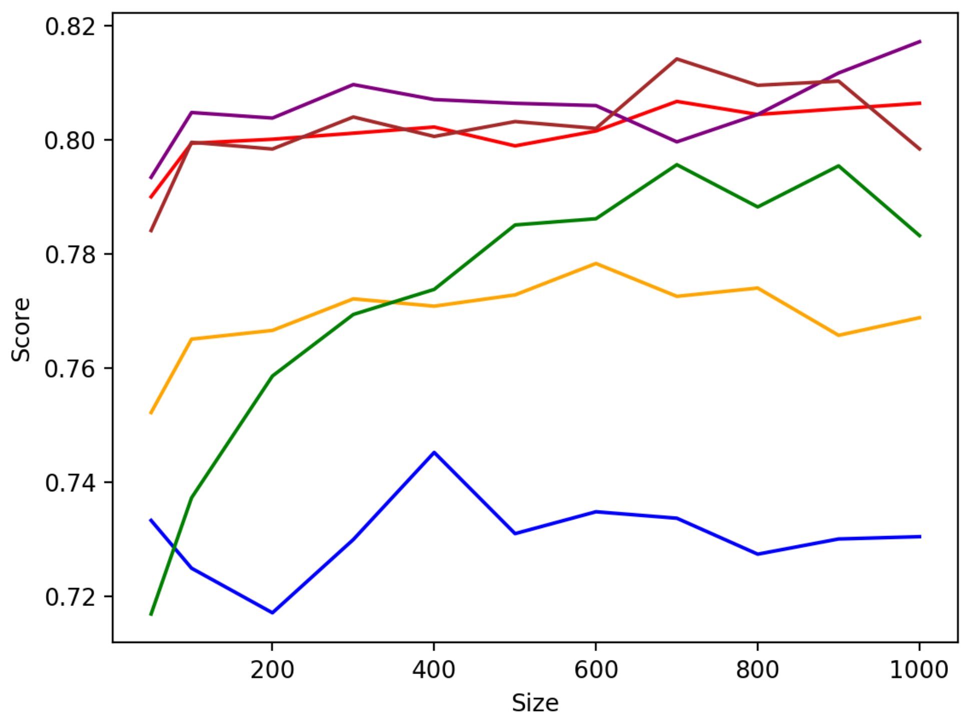


Figure 5.2 Score comparation of different models with different train size

It is obvious that the decision tree has worse performance while Random Forest and AdaBoost models begin with low accuracy but gradually fit the dataset as the train size grows. The top performers are Logic Regression, Neural Network and SVM, and the Neural Network usually performs better than the others. But the difference is not significant among those three models.

C)

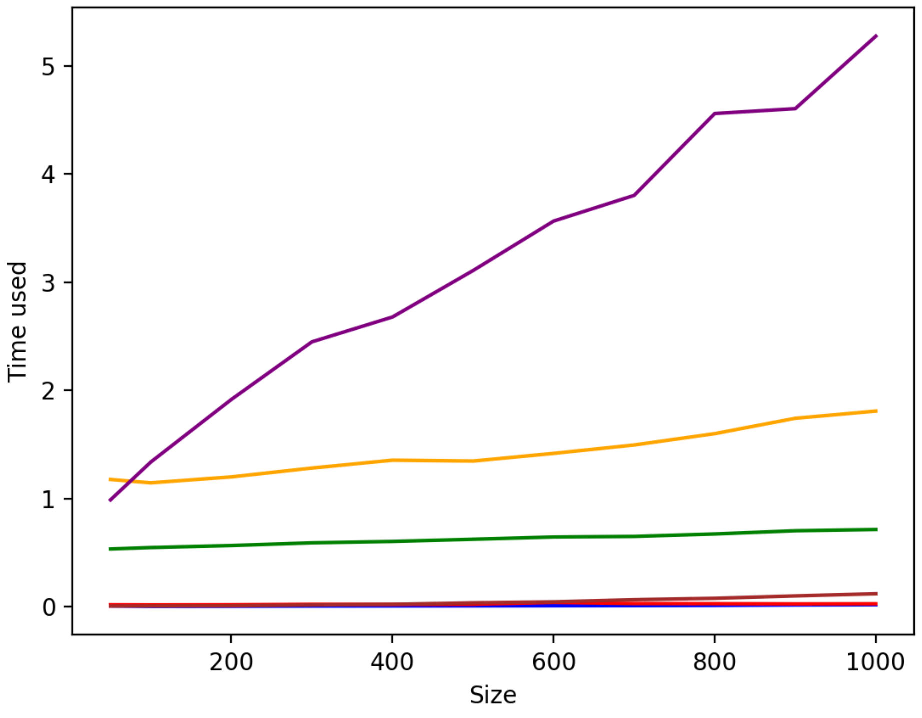


Figure 5.3 Time cost of each model for different train size

It is observed that the time costs of Decision Tree, SVM and Logistic regression are constantly low (around 0) and independent from train size. While time costs of AdaBoost and Random Forest are relative higher with 0.5 and 1.0 respectively. Notice that the cost of AdaBoost is uniform while that of Random Forest increases slightly with size accordingly. However, time used by Neural network grows linearly as the size increases.