CS 6140: Assignment 2

Luke Davidson

October 21 2022

Exercise 1

In this problem, we consider the signed distance between a point **x** and a hyperplane. This distance, d is given as:

$$d = \frac{w^T \mathbf{x} + b}{\parallel w \parallel} \tag{1.1}$$

where

$$\parallel w \parallel = \sqrt{\sum_{j=1}^{d} w_j^2} \tag{1.2}$$

a.) For an under classified **x**, or the class is given as negative when it should be positive, the weight update will perform as below:

$$\mathbf{w}^{t+1} = w^t + \mathbf{x}$$

If over classified (given a positive class when it should be negative), the weight update process will be:

$$\mathbf{w}^{t+1} = w^t - \mathbf{x}$$

I used the following code to simulate weight vectors and x data points and calculate the distance to a hyperplane before and after under and over classification:

```
num_tests = 1000
results_add = []
results_sub = []
for pn in range(2):
    """
    0: add
    1: sub
    """
    num_dec_underclassified = 0
    num_inc_overclassified = 0
    for i in range(num_tests):
        w = np.random.rand(1 ,3)*0.1
        val = 1000
        x = np.array([1, np.random.random()*val ,
            np.random.random()* val]).reshape(3, 1)
        dist_1 = dist(w, x)
        if pn == 0:
            w[0, np.random.randint(1,3)] += x[np.random.randint(1,3), 0]
             dist_2 = dist(w, x)
             if dist_1 > dist_2:
                 # decreased when underclassified
                 num_dec_underclassified += 1
        else:
            w[0, np.random.randint(1,3)] -= x[np.random.randint(1,3), 0]
             dist_2 = dist(w, x)
             if dist_1 < dist_2:
                 # increased when overclassified
                 num_inc_overclassified += 1
    if pn == 0:
        print(num_dec_underclassified)
    else:
        print(num_inc_overclassified)
```

Through my simulations and playing with hyperparameters, I found that this statement is false. I found that for most large values of x, the statement that the signed distance will decrease if x is overclassified held true for the most part (>99%, 1 or 2 out of thousands of runs), and vice versa for the first statement with large negative numbers as values of x. Although there was still a small percent chance that the statement did not hold true based on my simulations. Therefore, I believe these statements are completely dependent on x even if $\mathbf{x} \in R^d$.

b.) This statement is false. For an incorrectly classified data point **x**, the weight vector is updated in the following way:

$$\mathbf{w}^{t+1} = w^t + y\mathbf{x}$$

where y is the class label of data point **x**. To disprove the statement, we can use the simple case of a data point **x** that was incorrectly classified as positive, but is really negative. In this case, $y = -1$, so the weight vector will get the **x** value subtracted from it, causing it to be smaller. The weight update step will look like:

$$\mathbf{w}^{t+1} = w^t - \mathbf{x}$$

When we go to recalculate the denominator of equation 1.1, our new w_j corresponding to our incorrectly satisfied data point **x** will be smaller, causing the square, the sum, and the square root to all be smaller. The numerator will also be smaller because of the presence of w^T, but in certain cases the decrease in the denominator will trump the decrease in the numerator, causing the distance to increase. This is seen in the following example:

```
import numpy as np

# weights and data points
w = np.array([0.3, 0.5, 0.6]).reshape(1, 3)
x = np.array([1, 0.1, 0.8]).reshape(3, 1)

# calculate distance to hyperplane
def dist(w, x):
    num = np.dot(w, x)[0]
    den = np.sqrt(np.sum(np.square(w)))
    return num/den

dist_1 = dist(w, x)
print(dist_1)

# incorrect data point weight is updated
w[0, 1] -= x[1, 0]

dist_2 = dist(w, x)
print(dist_2)
```

The output comes out as:

```
First distance: 0.9920397457475466
Updated weight distance: 1.049902415449747
```

We can see the updated distance did increase given these values.

Exercise 2

The Naive Bayes classifier I constructed for both parts a and b for this problem is given here:

```
import numpy as np

# data
x_data = np.array([[-1, -1, -1], [-1, -1, 1], [-1, 1, -1],
                [-1, 1, 1], [1, -1, -1], [1, -1, 1],
                [1, 1, -1], [1, 1, 1]])
y_labels = np.array([1, 0, 0, 1, 0, 1, 1, 0]).reshape(8,1)

# part b data
new_data = np.empty((8, 12))
for n in range(x_data.shape[0]):
    new_data[n, 0] = x_data[n, 0]
    new_data[n, 1] = x_data[n, 1]
    new_data[n, 2] = x_data[n, 2]
    new_data[n, 3] = x_data[n, 0] * x_data[n, 1]
    new_data[n, 4] = x_data[n, 0] * x_data[n, 2]
    new_data[n, 5] = x_data[n, 1] * x_data[n, 2]
    new_data[n, 6] = x_data[n, 0] * x_data[n, 1] * x_data[n, 2]
    new_data[n, 7] = x_data[n, 0] * x_data[n, 0] * x_data[n, 1]
    new_data[n, 8] = x_data[n, 0] * x_data[n, 1] * x_data[n, 1]
    new_data[n, 9] = x_data[n, 0] * x_data[n, 2] * x_data[n, 2]
    new_data[n, 10] = x_data[n, 1] * x_data[n, 1] * x_data[n, 2]
    new_data[n, 11] = x_data[n, 1] * x_data[n, 2] * x_data[n, 2]

class Evaluator():
    def __init__(self, x_data, labels, part):
        """
        x_data: np.array with x data
        labels: np.array with y (labels)
        part: string "a" or "b"
        """
        if part == "a":
            self.x_data = x_data
        elif part == "b":
            self.x_data = np.empty((8,12))
            for n in range(x_data.shape[0]):
                self.x_data[n, 0] = x_data[n, 0]
                self.x_data[n, 1] = x_data[n, 1]
                self.x_data[n, 2] = x_data[n, 2]
                self.x_data[n, 3] = x_data[n, 0] * x_data[n, 1]
                self.x_data[n, 4] = x_data[n, 0] * x_data[n, 2]
                self.x_data[n, 5] = x_data[n, 1] * x_data[n, 2]
                self.x_data[n, 6] = x_data[n, 0] * x_data[n, 1] *
                    x_data[n, 2]
                self.x_data[n, 7] = x_data[n, 0] * x_data[n, 0] *
                    x_data[n, 1]
                self.x_data[n, 8] = x_data[n, 0] * x_data[n, 1] *
                    x_data[n, 1]
                self.x_data[n, 9] = x_data[n, 0] * x_data[n, 2] *
                    x_data[n, 2]
                self.x_data[n, 10] = x_data[n, 1] * x_data[n, 1] *
                    x_data[n, 2]
                self.x_data[n, 11] = x_data[n, 1] * x_data[n, 2] *
                    x_data[n, 2]
        self.labels = labels
        self.all_data = np.append(self.x_data, self.labels, axis=1)
        self.num_pts = self.x_data.shape[0]
        self.pos_data = np.empty((0, self.all_data.shape[1]))
        self.neg_data = np.empty((0, self.all_data.shape[1]))
        for pt in range(self.num_pts):
            if self.all_data[pt, -1] == 1:
                self.pos_data = np.append([self.all_data[pt, :]],
                    self.pos_data, axis=0)
            else:
                self.neg_data = np.append([self.all_data[pt, :]],
                    self.neg_data, axis=0)
        self.num_pos = self.pos_data.shape[0]
        self.num_neg = self.neg_data.shape[0]
        self.p_pos = self.num_pos/self.num_pts
        self.p_neg = self.num_neg/self.num_pts

    def evaluate(self, data_pts):
        """
        data_pts: np.array of data points, shape (n, d)
        """
        self.results = np.empty((data_pts.shape[0], 2))

        for k in range(2):
            for n in range(data_pts.shape[0]):
                if k == 0:
                    self.neg = self.p_neg
                else:
                    self.pos = self.p_pos
                for d in range(data_pts.shape[1]):
                    p_pt_k = self.calculate_P(data_pts[n, d], k, d)
                    if k == 0:
                        self.neg *= p_pt_k
                    else:
                        self.pos *= p_pt_k
                if k == 0:
                    self.results[n, 0] = self.neg
                else:
```

```
                self.results[n, 1] = self.pos
        self.calculate_error()

    def calculate_P(self, data_pt_val, k, d):
        if k == 0:
            count_neg = 0
            for row in range(self.neg_data.shape[0]):
                if self.neg_data[row, d] == data_pt_val:
                    count_neg += 1
            return count_neg/self.num_neg
        elif k == 1:
            count_pos = 0
            for row in range(self.pos_data.shape[0]):
                if self.pos_data[row, d] == data_pt_val:
                    count_pos += 1
            return count_pos/self.num_pos

    def calculate_error(self):
        self.predictions = np.argmax(self.results, axis=1)
        print("Results from each test data point:")
        print(self.results)
        print("Predicted classes from test data:")
        print(self.predictions)
        print("Actual classes of test data:")
        print(self.labels.reshape(8,))
        count = 0
        for i in range(self.predictions.shape[0]):
            if self.predictions[i] == self.labels[i]:
                count += 1
        print(f"Accuracy: {round(count/self.predictions.shape[0]*100,
                4)}%")

# Execute
env = Evaluator(x_data, y_labels, "b")
env.evaluate(new_data)
```

a.) To construct the Naive Bayes classifier above for part a of this problem, I began by defining the overall equation for a Naive Bayes classifier:

$$y_{guess} = argmax_Y\{P(Y) \cdot \prod_{j=1}^{n}(P(x_j \mid Y))\} \tag{2.1}$$

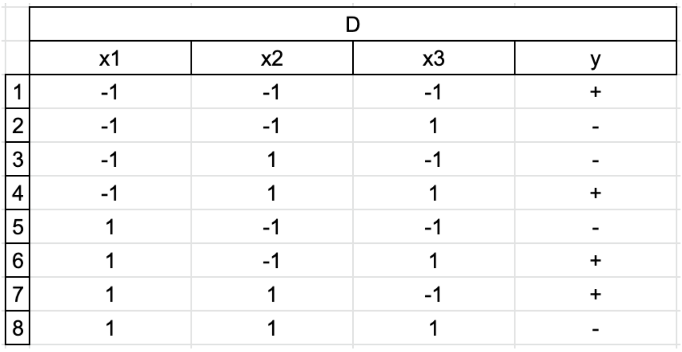
where

$$P(Y) = \frac{N_Y}{N_{total}}$$

$$P(x_j \mid Y) = \frac{N_{x_j}}{N_Y}$$

where N_Y is the number of data points classified as class y, N_{total} is the total number of data points, and N_{x_j} is the number of data points where feature x_j appears for class Y in the training set D. In the above code, $P(Y)$ is denoted as self.p_{pos/neg}, and $P(x_j \mid Y)$ is calculated in the method *calculate_P*(). The whole multiplication for each point is done out in the method *evaluate*(), and the argmax function is completed in the method *calculate_error*().

Our training set is defined below:



	D			
	x1	x2	x3	y
1	-1	-1	-1	+
2	-1	-1	1	-
3	-1	1	-1	-
4	-1	1	1	+
5	1	-1	-1	-
6	1	-1	1	+
7	1	1	-1	+
8	1	1	1	-

Figure 1: D for Q2a

Using equation 2.1 and the training data set points, an example calculation is shown below:

The calculation for the probability that data point 1 belongs to class "+" or "-" is:

$$\begin{aligned}
P(+ \mid x = (-1,-1,-1)) &= P(+) \cdot P(x_1 = -1 \mid Y = +) \cdot P(x_2 = -1 \mid Y = +) \cdot P(x_3 = -1 \mid Y = +) \\
&= \frac{4}{8} \cdot \frac{2}{4} \cdot \frac{2}{4} \cdot \frac{2}{4} = \frac{1}{16} = 0.0625 \\
P(- \mid x = (-1,-1,-1)) &= P(-) \cdot P(x_1 = -1 \mid Y = -) \cdot P(x_2 = -1 \mid Y = -) \cdot P(x_3 = -1 \mid Y = -) \\
&= \frac{4}{8} \cdot \frac{2}{4} \cdot \frac{2}{4} \cdot \frac{2}{4} = \frac{1}{16} = 0.0625
\end{aligned}$$

The final y_{guess} will be the argmax of these two values, which ends up in a tie. This exact calculation will hold for every point in this training set because of the following two relationships in this set D:

$$P(+) = P(-) = \frac{4}{8} = \frac{1}{2}$$

and

$$P(x_j = \{-1, 1\} \mid Y = \{+, -\}) = \frac{2}{4} = \frac{1}{2}$$

For the case of a tie, the argmax function will deliver the first class in the list, which will result in an accuracy of 50% whether it is + or - since there are both 4 + labels and 4 - labels.

Entering the following in to my Naive Bayes classifier results in the same results:

Input:

```
env = Evaluator(x_data, y_labels, "a")
env.evaluate(x_data)
```

Output:

```
Results from each test data point:
[[0.0625 0.0625]
 [0.0625 0.0625]
 [0.0625 0.0625]
 [0.0625 0.0625]
 [0.0625 0.0625]
 [0.0625 0.0625]
 [0.0625 0.0625]
 [0.0625 0.0625]]
Predicted classes from test data:
[0 0 0 0 0 0 0 0]
Actual classes of test data:
[1 0 0 1 0 1 1 0]
Accuracy: 50.0%
```

where 0 = "-", 1 = "+". We can see all values came out to $\frac{1}{2^4}$ and therefore all predictions were "-", resulting in an accuracy of 50%.

b.) Transforming the input space in to a higher-dimensional space defined as in the homework results in the following 12-d data set:

```
[[-1. -1. -1.  1.  1.  1. -1. -1. -1. -1. -1. -1.]
 [-1. -1.  1.  1. -1. -1.  1. -1. -1. -1.  1. -1.]
 [-1.  1. -1. -1.  1. -1.  1.  1. -1. -1. -1.  1.]
 [-1.  1.  1. -1. -1.  1. -1.  1. -1. -1.  1.  1.]
 [ 1. -1. -1. -1. -1.  1.  1. -1.  1.  1. -1. -1.]
 [ 1. -1.  1. -1.  1. -1. -1. -1.  1.  1.  1. -1.]
 [ 1.  1. -1.  1. -1. -1. -1.  1.  1.  1. -1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]]
```

The process to conduct the above calculations are the exact same, now just 13 multiplications instead of 4. To use this new data in my classifier and then test the same set, the following commands were entered, with the following output:

Input:

```
env = Evaluator(x_data, y_labels, "b")
env.evaluate(new_data)
```

Output:

```
Results from each test data point:
[[0.         0.00024414]
 [0.00024414 0.        ]
 [0.00024414 0.        ]
 [0.         0.00024414]
 [0.00024414 0.        ]
 [0.         0.00024414]
 [0.         0.00024414]
 [0.00024414 0.        ]]
Predicted classes from test data:
```

```
[1 0 0 1 0 1 1 0]
Actual classes of test data:
[1 0 0 1 0 1 1 0]
Accuracy: 100.0%
```

As we can see, transforming the data in to a higher dimension space made it so that the data can be perfectly classified. This is because of the way the positively and negatively classified data is split up. We can visualize the two below:

```
Positively classified data:
[[ 1.  1. -1.  1. -1. -1. -1.  1.  1.  1. -1.  1.  1.]
 [ 1. -1.  1. -1.  1. -1. -1. -1.  1.  1.  1. -1.  1.]
 [-1.  1.  1. -1. -1.  1. -1.  1. -1. -1.  1.  1.  1.]
 [-1. -1. -1.  1.  1.  1. -1. -1. -1. -1. -1. -1.  1.]]
Negatively classified data:
[[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.]
 [ 1. -1. -1. -1. -1.  1.  1. -1.  1.  1. -1. -1.  0.]
 [-1.  1. -1. -1.  1. -1.  1.  1. -1. -1. -1.  1.  0.]
 [-1. -1.  1.  1. -1. -1.  1. -1. -1. -1.  1. -1.  0.]]
```

We can see the reason that some the values of equation 2.1 are 0 for the incorrect label is due to column number 7. It is a column of -1's for label 1, or "+", and a column of 1's for label 0, or "-". This means that when we go to calculate any point labeled in the other class, $P(x_7 = 1 \mid +)$ = $P(x_7 = -1 \mid -)$ = 0, so when they are all multiplied together, it will result in a 0 value.

Exercise 3

It is given that point **x** is on the decision surface if $P(Y = 1 \mid x) = P(Y = 0 \mid x)$. We can then use Bayes theorem to solve this problem:

$$P(Y \mid x) = \frac{P(x \mid Y) \cdot P(Y)}{\sum_Y P(Y)} \tag{3.1}$$

In all parts of this problem, the denominator of $\sum_Y P(Y)$ will cancel out when we equate the two sides since it is a summation over the same set Y.

a.) For part a, equating equation 3.1 (after cancelling the denominator) will look like the following:

$$P(x \mid Y = 0) \cdot P(Y = 0) = P(x \mid Y = 1) \cdot P(Y = 1)$$

In this case, the $P(Y = 0)$ and $P(Y = 1)$ terms will also cancel since it's given that they are both equal to $\frac{1}{2}$. This leaves the following when plugging in to the given $P(x \mid Y = i)$ equation with the given values:

$$e^{-\frac{1}{2}\cdot(\mathbf{x}-(1,2))^T \cdot I \cdot (\mathbf{x}-(1,2))} = e^{-\frac{1}{2}\cdot(\mathbf{x}-(6,3))^T \cdot I \cdot (\mathbf{x}-(6,3))}$$

Clearly we can simple further by cancelling out the exponentials and $-\frac{1}{2}$ terms. Furthermore, the transposed $(\mathbf{x} - \mathbf{m})$ term multiplied with the identity matrix will results in itself, then multiplied again by its transpose. This will lead to the following:

$$(x_1 - 1)^2 + (x_2 - 2)^2 = (x_1 - 6)^2 + (x_2 - 3)^2$$

Expanding to...

$$x_1^2 - 2x_1 + 1 + x_2^2 - 4x_2 + 4 = x_1^2 - 12x_1 + 36 + x_2^2 - 6x_2 + 9$$

Finally simplifying to our optimal decision surface for 2-dimensional **x**:

$$f(\mathbf{x}) = 5x_1 + x_2 - 20 = 0$$

b.) For a general $m_0 = (m_{01}, m_{02})$ and $m_1 = (m_{11}, m_{12})$, $\sum_0 = \sum_1 = \sigma^2 \cdot I$, and $P(Y = 0) \neq P(Y = 1)$ the process and results are similar. In this case, we obviously cannot cancel out the $P(Y = i)$ terms since they are not equal.

We will begin with the simplified equation:

$$e^{-\frac{1}{2}\cdot(\mathbf{x}-(m_{01},m_{02}))^T \cdot (\sigma^2 \cdot I)^{-1} \cdot (\mathbf{x}-(m_{01},m_{02}))} \cdot P(Y = 0) = e^{-\frac{1}{2}\cdot(\mathbf{x}-(m_{11},m_{12}))^T \cdot (\sigma^2 \cdot I)^{-1} \cdot (\mathbf{x}-(m_{11},m_{12}))} \cdot P(Y = 1)$$

We can then log both sides and multiply by -2 to get:

$$(\mathbf{x} - (m_{01}, m_{02}))^T \cdot (\sigma^2 \cdot I)^{-1} \cdot (\mathbf{x} - (m_{01}, m_{02})) - 2log(P(Y = 0)) =$$

$$(\mathbf{x} - (m_{11}, m_{12}))^T \cdot (\sigma^2 \cdot I)^{-1} \cdot (\mathbf{x} - (m_{11}, m_{12})) - 2log(P(Y = 1))$$

Inverse of $\sigma^2 \cdot I$ is given as:

$$\begin{bmatrix} \sigma^{-2} & 0 \\ 0 & \sigma^{-2} \end{bmatrix}$$

Multiplying out the matrices and simplifying the σ^{-2} terms leads to:

$$(x_1 - m_{01})^2 + ((x_2 - m_{02})^2) - 2\sigma^2 log(P(Y = 0)) = (x_1 - m_{11})^2 + ((x_2 - m_{12})^2) - 2\sigma^2 log(P(Y = 1))$$

Which expands and simplifies to our final answer:

$$f(\mathbf{x}) = 2x_1(m_{01} - m_{11}) + 2x_1(m_{02} - m_{12}) + m_{11}^2 + m_{12}^2 - m_{01}^2 - m_{02}^2 - 2\sigma^2 log(\frac{P(Y = 1)}{P(Y = 0)}) = 0$$

We can double check this by plugging in the values from part a, which leads to the same answer as part a.

c.) For arbitrary covariance matrices $\sum_0$ and $\sum_1$ with σ_0^2 and σ_1^2, respectively, we will have an extra $\frac{1}{\sigma_{0/1}^2}$ term from the denominator of the original $P(x \mid Y = i)$ equation. The expansion will look like the following:

$$\frac{e^{-\frac{1}{2}\cdot(\mathbf{x}-(m_{01},m_{02}))^T\cdot(\sigma_0^2\cdot I)^{-1}\cdot(\mathbf{x}-(m_{01},m_{02}))} \cdot P(Y = 0)}{\sigma_0^2} = \frac{e^{-\frac{1}{2}\cdot(\mathbf{x}-(m_{11},m_{12}))^T\cdot(\sigma_1^2\cdot I)^{-1}\cdot(\mathbf{x}-(m_{11},m_{12}))} \cdot P(Y = 1)}{\sigma_1^2}$$

Logging and multiplying both sides by -2 leads to:

$$(\mathbf{x} - (m_{01}, m_{02}))^T \cdot (\sigma_0^2 \cdot I)^{-1} \cdot (\mathbf{x} - (m_{01}, m_{02})) - 2log(\frac{P(Y = 0)}{\sigma_0^2}) =$$
$$(\mathbf{x} - (m_{11}, m_{12}))^T \cdot (\sigma_1^2 \cdot I)^{-1} \cdot (\mathbf{x} - (m_{11}, m_{12})) - 2log(\frac{P(Y = 1)}{\sigma_1^2})$$

Expanding leads to:

$$\sigma_0^{-2}\cdot(x_1-m_{01})^2+\sigma_0^{-2}\cdot(x_2-m_{02})^2-2log(\frac{P(Y = 0)}{\sigma_0^2}) = \sigma_1^{-2}\cdot(x_1-m_{11})^2+\sigma_1^{-2}\cdot(x_2-m_{12})^2-2log(\frac{P(Y = 1)}{\sigma_1^2})$$

Simplifying to $\mathbf{x}$ terms on one side leads to our answer:

$$f(\mathbf{x}) = x_1^2(\sigma_0^{-2} - \sigma_1^{-2}) + x_1(-2m_{01}\sigma_0^{-2} + 2m_{11}\sigma_1^{-2}) + x_2^2(\sigma_0^{-2} - \sigma_1^{-2}) + x_2(-2m_{02}\sigma_0^{-2} + 2m_{12}\sigma_1^{-2}) - ...$$
$$...\sigma_1^{-2}(m_{11}^2 + m_{12}^2) + \sigma_0^{-2}(m_{01}^2 + m_{02}^2) + 2log(\frac{\sigma_0^2 P(Y = 1)}{\sigma_1^2 P(Y = 0)}) = 0$$

We can see that because σ_0 and σ_1 are different, the squared values of $\mathbf{x}$ will not cancel out, causing the optimal decision surface to be of a higher order shape, ex. curved vs linear, than seen in parts a and b.

Exercise 4

a.) For this problem, we want to minimize the sum of squares. We also know $p(y \mid x) = (w^T x)^{2y}$ for $y = 1$, and $(1 - w^T x)^2 \cdot (1 - y)$ for $y = 0$. This leads to the log-likelihood of our weights, **w**, as:

$$2 \cdot \sum_{i=1}^{n} y_i \cdot log(w^T x_i) + (1 - y_i) \cdot log(1 - w^T x_i)$$

Through simple algebra and log relations, this leads to:

$$2 \cdot \sum_{i=1}^{n} (y_i - 1) \cdot (w^T x_i) + log(w^T x_i)$$

We now want the derivative with respect to all w_j. Taking this derivative leads to:

$$\begin{aligned}\frac{dll(w)}{dw_j} &= 2 \cdot \sum_{i=1}^{n} (y_i - 1) \cdot (x_{ij}) + w^T x_i \cdot x_{ij}^{-1} \cdot x_{ij} \\ &= 2 \cdot \sum_{i=1}^{n} x_{ij}(y_i - 1 - \frac{w^T x_i}{x_{ij}}) \\ &= 2 \cdot \sum_{i=1}^{n} x_{ij}(y_i - w^T x_i)\end{aligned}$$

From here, we can see that our $w^T x_i$ term is equivalent to our **p** term. We can also see that the summation across the i^{th} data point of x_{ij} is equivalent to a feature vector of the j^{th} feature. We can then further simplify this equation to:

$$2 \cdot \mathbf{f}_j^T (\mathbf{y} - \mathbf{p})$$

Across every i^{th} data point, this simplifies further and we get our update rule:

$$w^{t+1} = w^t + \eta \cdot \nabla \mathbf{w}$$

where

$$\nabla \mathbf{w} = 2 \cdot X^T (\mathbf{y} - \mathbf{p}) \tag{4.1}$$

This update rule is implemented in part 4c.

b.) For this problem, we start with the distance from a data point **x, y** to a hyperplane as described in the problem:

$$r = \frac{w_0 + \sum_{j=1}^{d} w_j x_j}{\sqrt{\sum_{j=1}^{d} w_j^2}} \tag{4.2}$$

Given the problem statement, we want to minimize the sum of squares of this distance. We begin with the likelihood with respect to **w**, similar to last problem:

$$l(\mathbf{w}) = \prod_{i=1}^{n} (\frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}})^2 \cdot (1 - \frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}})^2$$

Taking the log likelihood of this to simplify leads to:

$$= 2\sum_{i=1}^{n} log(\frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}}) + log(1 - \frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}})$$

Using the law of logs this simplifies to:

$$= 2\sum_{i=1}^{n} log(\frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}} - \frac{(w^T x_i + w_0)^2}{\sum_{j=1}^{d} w_j^2})$$

Now we want to take the derivative of this with respect to our parameter we want to estimate, **w**:

$$\frac{dll(\mathbf{w})}{d\mathbf{w}} = 2\sum_{i=1}^{n} \frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}} \cdot x_{ij} - \frac{2(w^T x_i + w_0)(x_{ij})(w_j)}{\sum_{j=1}^{d} w_j^2}$$

which, similar to last problem, simplifies by pulling out the x_{ij} term:

$$= 2\sum_{i=1}^{n} x_{ij}(\frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}} - \frac{2(w^T x_i + w_0)(w_j)}{\sum_{j=1}^{d} w_j^2})$$

This x_{ij} term will again create a feature vector of the j^{th} feature when summed over $i = 1$ to n, although for the simplicity of implementing this for part c, we will keep it in this form. This ends as our update rule for minimizing the sum of squared distance to the hyperplane:

$$w^{t+1} = w^t + \eta \cdot \nabla\mathbf{w}$$

where

$$\nabla\mathbf{w} = 2\sum_{i=1}^{n} x_{ij}(\frac{w^T x_i + w_0}{\sqrt{\sum_{j=1}^{d} w_j^2}} - \frac{2(w^T x_i + w_0)(w_j)}{\sum_{j=1}^{d} w_j^2}) \tag{4.3}$$

c.) I used the following code to implement my update rules, with just changing the way $\nabla\mathbf{w}$ was calculated:

```
import numpy as np

# Initialize Data Params
n = 15
d = 8
learning_rate = 0.01

# Create data
x_data = np.random.rand(n, d)
x_data_1 = np.append(np.ones((n, 1)), x_data, axis=1)
# n x d+1
x_data_T = np.transpose(x_data_1)
# d+1 x n
w_actual = np.random.rand(d+1, 1)
# d+1 x 1
```

```
y_data = np.dot(x_data_1, w_actual)
# (n x d+1) * (d+1 x 1) = n x 1
w_star = np.dot(np.dot(np.linalg.inv((np.dot(x_data_T, x_data_1))), x_dat
# d+1 x 1

def calc_delt_w(x, y, w):
    p = np.dot(x, w)
    delt_w = 2*np.dot(np.transpose(x), y-p)
    return delt_w

def calc_R2(w_gd, w_star):
    num = np.sum(np.square(w_star - w_gd))
    y_m = np.mean(w_gd)
    den = np.sum(np.square(w_gd - y_m))
    return 1 - num/den

# Now we want to estimate our w vector
w_init = np.random.rand(d+1, 1)          # initialize to 0 vector
num_epochs = 1000
for ep in range(num_epochs):
    if ep == 0:
        w = w_init
    delt_w = calc_delt_w(x_data_1, y_data, w)
    w += learning_rate*delt_w
r2 = calc_R2(w, w_star)

print("W actual:")
print(w_actual)
print("\nW grad descent:")
print(w)
print("\nW*:")
print(w_star)
print(f"R squared: {r2}")
```

For 3 randomly generated sets of data points and weights, I received the following results:

Trial 1:

```
W actual:
[[0.2966381 ]
 [0.22325538]
 [0.09909381]
 [0.63591618]
 [0.3866431 ]
 [0.22800516]
 [0.95442526]
 [0.45488444]
 [0.56392892]]
```

```
W grad descent:
 [[0.2890117 ]
  [0.22666231]
  [0.10470047]
  [0.64225766]
  [0.38872633]
  [0.22750061]
  [0.9508408 ]
  [0.4531393 ]
  [0.56507701]]

W*:
 [[0.2966381 ]
  [0.22325538]
  [0.09909381]
  [0.63591618]
  [0.3866431 ]
  [0.22800516]
  [0.95442526]
  [0.45488444]
  [0.56392892]]

 R squared: 0.9997004009324237
```

Trial 2:

```
W actual:
 [[0.47830688]
  [0.1523674 ]
  [0.67539233]
  [0.25802045]
  [0.45019195]
  [0.46129102]
  [0.04620609]
  [0.19435741]
  [0.65385798]]

W grad descent:
 [[0.46740268]
  [0.16674473]
  [0.67818196]
  [0.25873461]
  [0.43986107]
  [0.46219533]
  [0.04995053]
  [0.20211565]
  [0.65398005]]

W*:
```

```
[[0.47830688]
  [0.1523674 ]
  [0.67539233]
  [0.25802045]
  [0.45019195]
  [0.46129102]
  [0.04620609]
  [0.19435741]
  [0.65385798]]

R squared: 0.9986519420624336
```

Trial 3:

```
W actual:
 [[0.72964428]
  [0.10198406]
  [0.63409011]
  [0.01389148]
  [0.71612242]
  [0.04652104]
  [0.20915056]
  [0.70681711]
  [0.04117807]]

W grad descent:
 [[ 0.71142321]
  [ 0.10119302]
  [ 0.64685351]
  [-0.02481796]
  [ 0.71714158]
  [ 0.08602245]
  [ 0.25027124]
  [ 0.72980791]
  [-0.01184145]]

W*:
 [[0.72964428]
  [0.10198406]
  [0.63409011]
  [0.01389148]
  [0.71612242]
  [0.04652104]
  [0.20915056]
  [0.70681711]
  [0.04117807]]

R squared: 0.990569065909239
```

where W* is calculated as:

$$(X^T X)^{-1} X^T y \tag{4.4}$$

As you can see based on the R^2 values, this update rule works very well in simulated data with 1000 epochs.

For part b, the update rule was changed to equation 4.3 above:

```
def calc_delt_w2(x, y, w):
    sq_den = np.sum(np.square(w))
    sqrt_sq_den = np.sqrt(sq_den)
    n_1 = np.dot(x, w)
    term_1 = n_1/sqrt_sq_den
    term_2 = 2*n_1/sq_den
    delt_w = 2*np.dot(np.transpose(x), term_1 - term_2)
    return delt_w
```

Unfortunately, I was not able to get reliable results for part b of the problem. My simulations were coming out similar to this:

```
W actual:
 [[0.53913462]
  [0.20436367]
  [0.62140634]
  [0.76408987]
  [0.24365719]
  [0.86399367]]

W grad descent:
 [[-21.15237906]
  [-10.90263878]
  [-10.78791457]
  [-10.87509657]
  [-10.86121594]
  [-10.97823031]]

W*:
 [[0.53913462]
  [0.20436367]
  [0.62140634]
  [0.76408987]
  [0.24365719]
  [0.86399367]]

R squared:  -11.771586576194485
```

Clearly my weights are not updating as they should/as I would like them to for a high number of epochs. My main source of error is my derivation of the update rule, which I believe may be off by an *i*, *j*, square or something similar. Unfortunately I was not able to locate my error in time, so please let me know if you see it anywhere.