CSE 546 HW2

Pushpak Sarkar

TOTAL POINTS

10 / 25

QUESTION 1

1 0 / 2

✓ **- 1.5 pts** Incorrect/missing characterization

✓ **- 0.5 pts** Incorrect/missing deterministic decision rule

QUESTION 2

8 pts

2.1 1 / 1

✓ **- 0 pts** Correct

2.2 1 / 1

✓ **- 0 pts** Correct

2.3 1 / 1

✓ **- 0 pts** Correct

2.4 0 / 1

✓ **- 1 pts** Wrong

2.5 0 / 1

✓ **- 1 pts** Incorrect

2.6 0 / 0.5

✓ **- 0.5 pts** Incorrect

2.7 0 / 0.5

✓ **- 0.5 pts** Incorrect

2.8 0 / 1

✓ **- 1 pts** Wrong

2.9 0 / 1

✓ **- 1 pts** Incorrect, didn't attempt

QUESTION 3

3 3.5 / 5

✓ **- 1 pts** comment of plot 2 is missing.

✓ **- 0.5 pts** x-axis in plot 1 should be in log scale.

QUESTION 4

4 3 / 5

✓ **- 1 pts** plot for errors v.s. lambda is not correct.

✓ **- 1 pts** the 10 features are missing

QUESTION 5

Programming: Binary Logistic Regression 5 pts

5.1 Gradients and Hessians 0.5 / 1

✓ **+ 0.25 pts** Gradient with respect to w correct

✓ **+ 0.25 pts** Hessian with respect to w correct

✓ **+ 0.25 pts** Gradient with respect to b correct

✓ **+ 0.25 pts** Hessian with respect to b correct

✓ **- 0.5 pts** Derivations missing

+ 0 pts Incorrect

+ 0 pts No answer submitted

5.2 Coding 0 / 4

✓ **- 0.5 pts** GD: Error in the loss function

✓ **- 0.5 pts** GD: Error in the classification error

✓ **- 0.5 pts** SGD: Error in the loss function

✓ **- 0.5 pts** SGD: Error in the classification error

✓ **- 0.5 pts** Batch SGD: Error in the loss function

✓ **- 0.5 pts** Batch SGD: Error in the classification error

✓ **- 0.5 pts** Newton's Method: Error in the loss function

✓ **- 0.5 pts** Newton's Method: Error in the classification error

CSE 546 HW 2

Pushpak Sarkar

November 2, 2018

Problem 1

$R(\delta)$
$= E_{XY,\delta}[1\{\delta(X) \neq Y\}]$
$= E_X[E_{Y|X}[1\{\delta(x) \neq Y\}|X = x]]$
$= E_{XY}[E_{\delta|XY}[1\{\delta(x) \neq Y\}|X = x, Y = y]]$
$= E_{XY}[1\{\delta(x) \neq Y\}P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[1 - \alpha(X, Y)|X = x, Y = y]$
$= E_X[E_{Y|X}[1 - \alpha(X, Y)|X = x]]$
$= E_X[\Sigma_{y=1}^K(1 - \alpha(X, Y))P(Y = y|X = x)]$
$= E_X[\Sigma_{y=1}^K P(Y = y|X = x) - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$
$= E_X[1 - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$

From the above expression, it is clear that to minimize $R(\delta)$, we need to maximize $\Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)$ over α where $\Sigma_{y=1}^K \alpha(x, y) = 1$

The coefficients of the variables in objective function are probabilities, so they are positive. To maximize, we can choose the coefficient of the largest α equal to 1 and set other coefficients equal to 0. There can be multiple Y's (for each x) which maximize $P(Y = y|X = x)$. Let Y^* denote the set of solutions that maximize $P(Y = y|X = x)$. For those $y^* \in Y^*$, the coefficient of maximum $\alpha(X, Y)$ is 1.

Problem 2

(a) The sample (x_i, y_i) is iid and given the classifier, the loss function $l(.)$ is bounded in the interval $[-1, 1]$. So the empirical risk is a random variable.

We can show, $E(\hat{R}_n(\tilde{f})) = R(\tilde{f})$ (due to the fact that expectation is a linear operator). Because the training data is iid and loss function is bounded, we can use the Hoeffding's inequality upper bound. Comparing the two inequalities, we can write the following -

$A = \epsilon$
$2exp(-\frac{2m\epsilon^2}{(b-a)^2}) = \delta$
Here, $b - a = 1 - (-1) = 2$, So, $2exp(-\frac{2m\epsilon^2}{4}) = \frac{\delta}{2}$
Taking logarithm on both sides, we get -
$-n\epsilon^2/2 = log(\delta/2)$
So, $\epsilon = A = \sqrt{\frac{2}{n}log(2/\delta)}$

(b) The same confidence interval will hold as f^* is the best classifier which minimizes true risk. Because we take expectation, it will not depend on the dataset.

(c) In this case the confidence interval will not hold as $\hat{f}$ is minimizes $\hat{R}_n(f)$. As it is dependent on the dataset, the empirical risk minimizer will change with a change of the dataset. So $\hat{f}$ will be different for each different dataset. We can write the following -
$\hat{R_n}(\hat{f}) = min_{f \in F}\hat{R}_n(f)$
$= min_{f \in F}\frac{1}{n}\Sigma_{i=1}^n 1(f(x_i) \neq y_i)$
To apply Hoeffding's inequality, $\hat{R_n}(\hat{f})$ needs to be written as the average of the iid terms, which is not the case here.

1 0 / 2

✓ - 1.5 pts Incorrect/missing characterization

✓ - 0.5 pts Incorrect/missing deterministic decision rule

CSE 546 HW 2

Pushpak Sarkar

November 2, 2018

Problem 1

$R(\delta)$
$= E_{XY,\delta}[1\{\delta(X) \neq Y\}]$
$= E_X[E_{Y|X}[1\{\delta(x) \neq Y\}|X = x]]$
$= E_{XY}[E_{\delta|XY}[1\{\delta(x) \neq Y\}|X = x, Y = y]]$
$= E_{XY}[1\{\delta(x) \neq Y\}P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[1 - \alpha(X, Y)|X = x, Y = y]$
$= E_X[E_{Y|X}[1 - \alpha(X, Y)|X = x]]$
$= E_X[\Sigma_{y=1}^K(1 - \alpha(X, Y))P(Y = y|X = x)]$
$= E_X[\Sigma_{y=1}^K P(Y = y|X = x) - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$
$= E_X[1 - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$

From the above expression, it is clear that to minimize $R(\delta)$, we need to maximize $\Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)$ over α where $\Sigma_{y=1}^K \alpha(x, y) = 1$

The coefficients of the variables in objective function are probabilities, so they are positive. To maximize, we can choose the coefficient of the largest α equal to 1 and set other coefficients equal to 0. There can be multiple Y's (for each x) which maximize $P(Y = y|X = x)$. Let Y^* denote the set of solutions that maximize $P(Y = y|X = x)$. For those $y^* \in Y^*$, the coefficient of maximum $\alpha(X, Y)$ is 1.

Problem 2

(a) The sample (x_i, y_i) is iid and given the classifier, the loss function $l(.)$ is bounded in the interval $[-1, 1]$. So the empirical risk is a random variable.

We can show, $E(\hat{R}_n(\tilde{f})) = R(\tilde{f})$ (due to the fact that expectation is a linear operator). Because the training data is iid and loss function is bounded, we can use the Hoeffding's inequality upper bound. Comparing the two inequalities, we can write the following -

$A = \epsilon$
$2exp(-\frac{2m\epsilon^2}{(b-a)^2}) = \delta$
Here, $b - a = 1 - (-1) = 2$, So, $2exp(-\frac{2m\epsilon^2}{4}) = \frac{\delta}{2}$
Taking logarithm on both sides, we get -
$-n\epsilon^2/2 = log(\delta/2)$
So, $\epsilon = A = \sqrt{\frac{2}{n}log(2/\delta)}$

(b) The same confidence interval will hold as f^* is the best classifier which minimizes true risk. Because we take expectation, it will not depend on the dataset.

(c) In this case the confidence interval will not hold as $\hat{f}$ is minimizes $\hat{R}_n(f)$. As it is dependent on the dataset, the empirical risk minimizer will change with a change of the dataset. So $\hat{f}$ will be different for each different dataset. We can write the following -
$\hat{R_n}(\hat{f}) = min_{f \in F}\hat{R}_n(f)$
$= min_{f \in F}\frac{1}{n}\Sigma_{i=1}^n 1(f(x_i) \neq y_i)$
To apply Hoeffding's inequality, $\hat{R_n}(\hat{f})$ needs to be written as the average of the iid terms, which is not the case here.

2.1 1 / 1

✓ **- 0 pts** Correct

CSE 546 HW 2

Pushpak Sarkar

November 2, 2018

Problem 1

$R(\delta)$
$= E_{XY,\delta}[1\{\delta(X) \neq Y\}]$
$= E_X[E_{Y|X}[1\{\delta(x) \neq Y\}|X = x]]$
$= E_{XY}[E_{\delta|XY}[1\{\delta(x) \neq Y\}|X = x, Y = y]]$
$= E_{XY}[1\{\delta(x) \neq Y\}P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[1 - \alpha(X, Y)|X = x, Y = y]$
$= E_X[E_{Y|X}[1 - \alpha(X, Y)|X = x]]$
$= E_X[\Sigma_{y=1}^K(1 - \alpha(X, Y))P(Y = y|X = x)]$
$= E_X[\Sigma_{y=1}^K P(Y = y|X = x) - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$
$= E_X[1 - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$

From the above expression, it is clear that to minimize $R(\delta)$, we need to maximize $\Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)$ over α where $\Sigma_{y=1}^K \alpha(x, y) = 1$

The coefficients of the variables in objective function are probabilities, so they are positive. To maximize, we can choose the coefficient of the largest α equal to 1 and set other coefficients equal to 0. There can be multiple Y's (for each x) which maximize $P(Y = y|X = x)$. Let Y^* denote the set of solutions that maximize $P(Y = y|X = x)$. For those $y^* \in Y^*$, the coefficient of maximum $\alpha(X, Y)$ is 1.

Problem 2

(a) The sample (x_i, y_i) is iid and given the classifier, the loss function $l(.)$ is bounded in the interval $[-1, 1]$. So the empirical risk is a random variable.

We can show, $E(\hat{R}_n(\tilde{f})) = R(\tilde{f})$ (due to the fact that expectation is a linear operator). Because the training data is iid and loss function is bounded, we can use the Hoeffding's inequality upper bound. Comparing the two inequalities, we can write the following -

$A = \epsilon$
$2exp(-\frac{2m\epsilon^2}{(b-a)^2}) = \delta$
Here, $b - a = 1 - (-1) = 2$, So, $2exp(-\frac{2m\epsilon^2}{4}) = \frac{\delta}{2}$
Taking logarithm on both sides, we get -
$-n\epsilon^2/2 = log(\delta/2)$
So, $\epsilon = A = \sqrt{\frac{2}{n}log(2/\delta)}$

(b) The same confidence interval will hold as f^* is the best classifier which minimizes true risk. Because we take expectation, it will not depend on the dataset.

(c) In this case the confidence interval will not hold as $\hat{f}$ is minimizes $\hat{R}_n(f)$. As it is dependent on the dataset, the empirical risk minimizer will change with a change of the dataset. So $\hat{f}$ will be different for each different dataset. We can write the following -
$\hat{R_n}(\hat{f}) = min_{f \in F}\hat{R}_n(f)$
$= min_{f \in F}\frac{1}{n}\Sigma_{i=1}^n 1(f(x_i) \neq y_i)$
To apply Hoeffding's inequality, $\hat{R_n}(\hat{f})$ needs to be written as the average of the iid terms, which is not the case here.

2.2 1 / 1

✓ **- 0 pts** Correct

CSE 546 HW 2

Pushpak Sarkar

November 2, 2018

Problem 1

$R(\delta)$
$= E_{XY,\delta}[1\{\delta(X) \neq Y\}]$
$= E_X[E_{Y|X}[1\{\delta(x) \neq Y\}|X = x]]$
$= E_{XY}[E_{\delta|XY}[1\{\delta(x) \neq Y\}|X = x, Y = y]]$
$= E_{XY}[1\{\delta(x) \neq Y\}P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[P(\delta(X) \neq Y|X = x, Y = y)]$
$= E_{XY}[1 - \alpha(X, Y)|X = x, Y = y]$
$= E_X[E_{Y|X}[1 - \alpha(X, Y)|X = x]]$
$= E_X[\Sigma_{y=1}^K(1 - \alpha(X, Y))P(Y = y|X = x)]$
$= E_X[\Sigma_{y=1}^K P(Y = y|X = x) - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$
$= E_X[1 - \Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)]$

From the above expression, it is clear that to minimize $R(\delta)$, we need to maximize $\Sigma_{y=1}^K \alpha(X, Y)P(Y = y|X = x)$ over α where $\Sigma_{y=1}^K \alpha(x, y) = 1$

The coefficients of the variables in objective function are probabilities, so they are positive. To maximize, we can choose the coefficient of the largest α equal to 1 and set other coefficients equal to 0. There can be multiple Y's (for each x) which maximize $P(Y = y|X = x)$. Let Y^* denote the set of solutions that maximize $P(Y = y|X = x)$. For those $y^* \in Y^*$, the coefficient of maximum $\alpha(X, Y)$ is 1.

Problem 2

(a) The sample (x_i, y_i) is iid and given the classifier, the loss function $l(.)$ is bounded in the interval $[-1, 1]$. So the empirical risk is a random variable.

We can show, $E(\hat{R}_n(\tilde{f})) = R(\tilde{f})$ (due to the fact that expectation is a linear operator). Because the training data is iid and loss function is bounded, we can use the Hoeffding's inequality upper bound. Comparing the two inequalities, we can write the following -

$A = \epsilon$
$2exp(-\frac{2m\epsilon^2}{(b-a)^2}) = \delta$
Here, $b - a = 1 - (-1) = 2$, So, $2exp(-\frac{2m\epsilon^2}{4}) = \frac{\delta}{2}$
Taking logarithm on both sides, we get -
$-n\epsilon^2/2 = log(\delta/2)$
So, $\epsilon = A = \sqrt{\frac{2}{n}log(2/\delta)}$

(b) The same confidence interval will hold as f^* is the best classifier which minimizes true risk. Because we take expectation, it will not depend on the dataset.

(c) In this case the confidence interval will not hold as $\hat{f}$ is minimizes $\hat{R}_n(f)$. As it is dependent on the dataset, the empirical risk minimizer will change with a change of the dataset. So $\hat{f}$ will be different for each different dataset. We can write the following -
$\hat{R_n}(\hat{f}) = min_{f \in F}\hat{R}_n(f)$
$= min_{f \in F}\frac{1}{n}\Sigma_{i=1}^n 1(f(x_i) \neq y_i)$
To apply Hoeffding's inequality, $\hat{R_n}(\hat{f})$ needs to be written as the average of the iid terms, which is not the case here.

2.3 1 / 1

✓ - **0 pts** Correct

(d)

(e)

(f)

(g)

(h)

(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

2.4 0 / 1

✓ **- 1 pts** Wrong

(d)

(e)

(f)

(g)

(h)

(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

2.5 **0 / 1**

✓ **- 1 pts** Incorrect

(d)

(e)

(f)

(g)

(h)

(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

2.6 0 / 0.5

✓ - 0.5 pts Incorrect

(d)

(e)

(f)

(g)

(h)

(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

2.7 0 / 0.5

✓ - 0.5 pts Incorrect

(d)

(e)

(f)

(g)

(h)

(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

2.8 0 / 1

✓ - 1 pts Wrong

(d)

(e)

(f)

(g)

(h)

(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

2.9 0 / 1

✓ **- 1 pts** Incorrect, didn't attempt

(d)

(e)

(f)

(g)

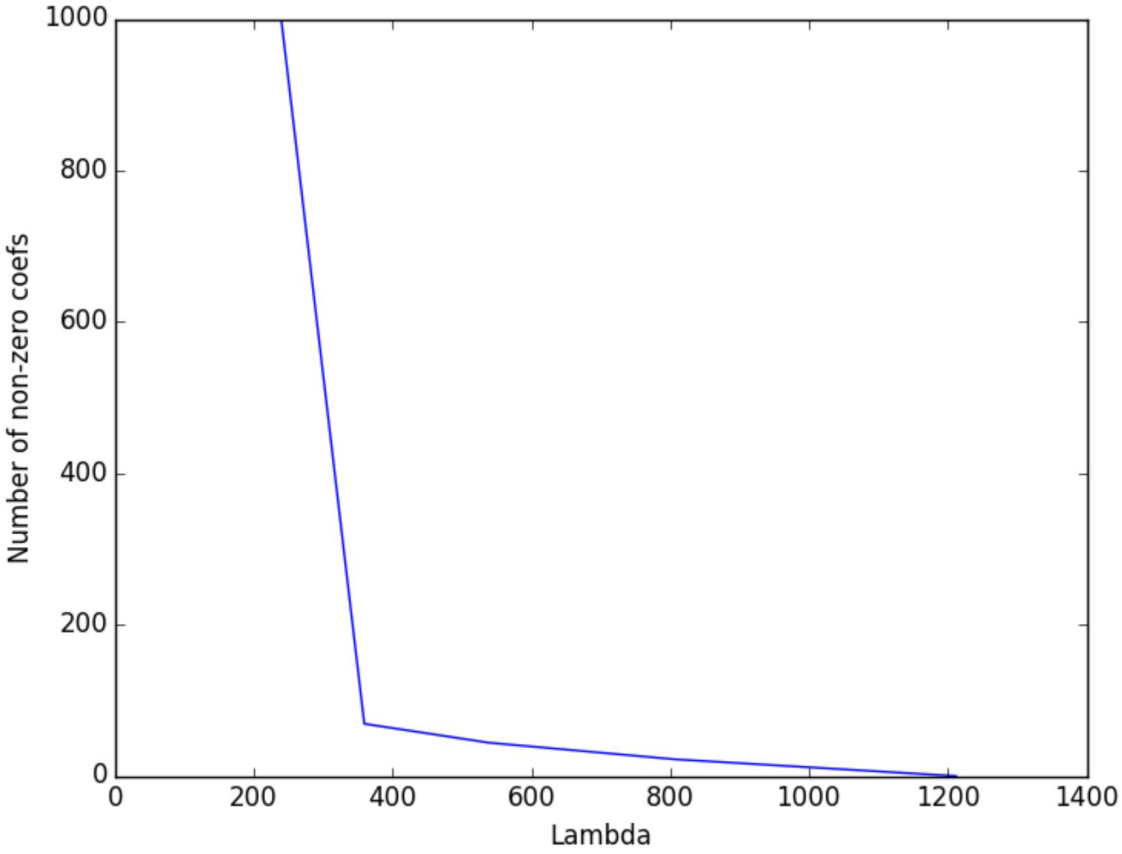
(h)

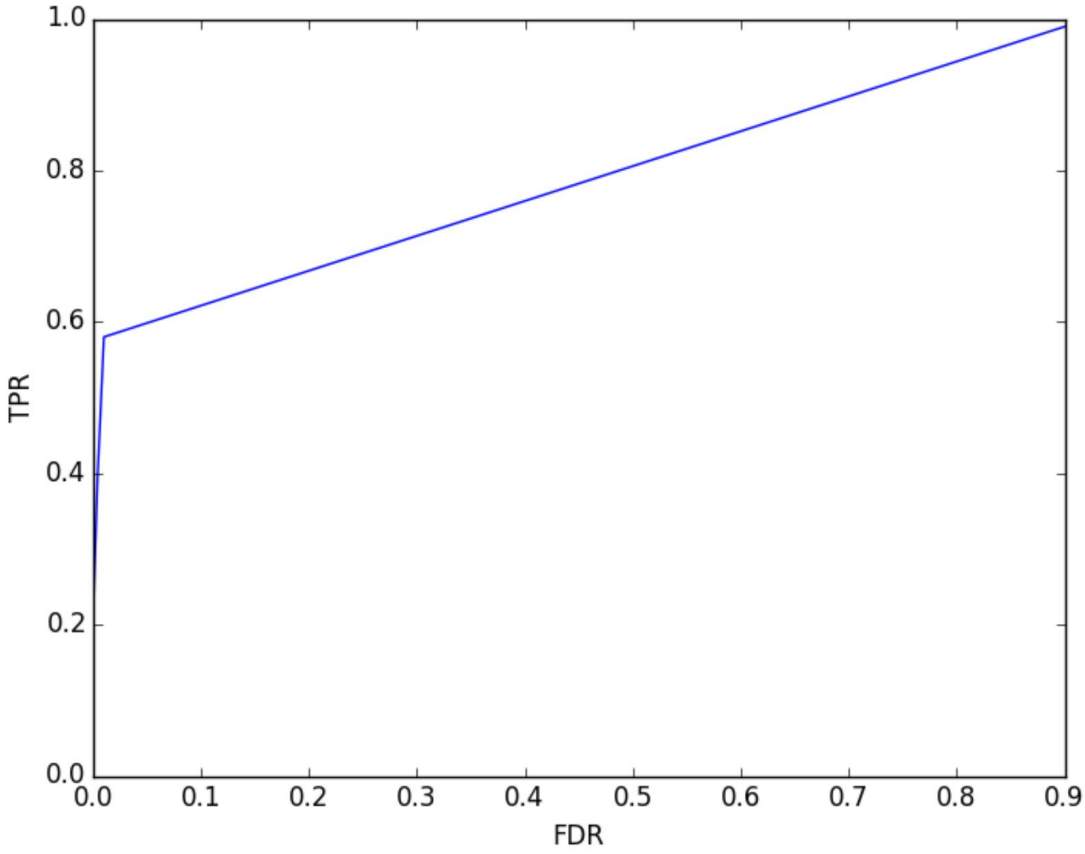
(i)

Problem 3

```
# Q3
import numpy as np
import matplotlib.pyplot as plt
import os
os.system('CLS')

k = 100
non_zero_coef = k
d = 1000
n = 500

sigma = 1

W = np.full(d, 0.0)
W[0:k] = 1.*(np.arange(k)+1)/k

epsilon = np.random.normal(0, sigma, n)
X = np.random.randn(n, d)
Y  = np.dot(X, W) + epsilon
Y = Y.reshape(n, 1)

lam_values = 2*abs(np.dot(X.T, (Y - np.mean(Y))))
lam = np.amax(lam_values)

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)

itr = 0
condition = True

while condition and itr <= 30:
    hyp_par = lam/(1.5**itr)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    while not conv_cond and counter <= 300:
        counter += 1
        b = np.mean(Y - np.dot(X, wvec))
        #print("b: ", b)
```

```
        W_est = list(wvec)

        for j in np.arange(d):
            a = 2*sum(X[:, j]**2)
            w_ex = np.copy(W_est)
            w_ex[j] = 0

            wx_prod = np.dot(X, w_ex).reshape(-1,1)
            diff = Y - (b + wx_prod)
            c = np.asscalar(2*np.dot(X[:, j], diff))
            c = 2*np.dot(X[:, j], diff)

            if c < - hyp_par:
                w = 1.*((c + hyp_par) / a)
            elif - hyp_par <= c <= hyp_par:
                w = 0.0
            else:
                w = 1.*((c - hyp_par) / a)

            wvec[j] = w

        if np.count_nonzero(wvec) == 0:
            break
        else:
            conv_cond = np.allclose(W_est, wvec, conv_limit)

        print("Convergence has been achieved: ", conv_cond)

    # Storing the optimal w for particular lambda
    W_mat = np.append(W_mat, wvec.reshape(1, d), axis=0)
    nzero_coefmat = np.append(nzero_coefmat, np.array([hyp_par, np.count_nonzero(wvec)]).reshape(1,

    condition = np.count_nonzero(wvec) <= d
    itr = itr + 1

plt.plot(nzero_coefmat[:, 0], nzero_coefmat[:, 1], label="Non-zero coef and lambda")
plt.xlabel("Lambda")
plt.ylabel("Number of non-zero coefs")
plt.show()

fdr_tpr = np.empty([0, 3])
for m in np.arange(W_mat.shape[0]):
    incorr_nonzero = np.count_nonzero(W_mat[m, 100:])/(np.count_nonzero(W_mat[m,] + 0.0000001))
    corr_nonzero = np.count_nonzero(W_mat[m, 0:99])/100
    fdr_tpr = np.append(fdr_tpr, np.array([W_mat[m, 0], incorr_nonzero, corr_nonzero]).reshape(1, 3)

print("FDR and TPR: \n", fdr_tpr)

plt.plot(fdr_tpr[:, 1], fdr_tpr[:, 2], label="FDR vs. TPR")
plt.xlabel("FDR")
plt.ylabel("TPR")
plt.show()
```


1000
800
600
400
200
0
Number of non-zero coefs
0
200
400
600
800
1000
1200
1400
Lambda



When the value of lambda is high, very few features are selected and as we keep on reducing the regularization parameter, more features are being selected.

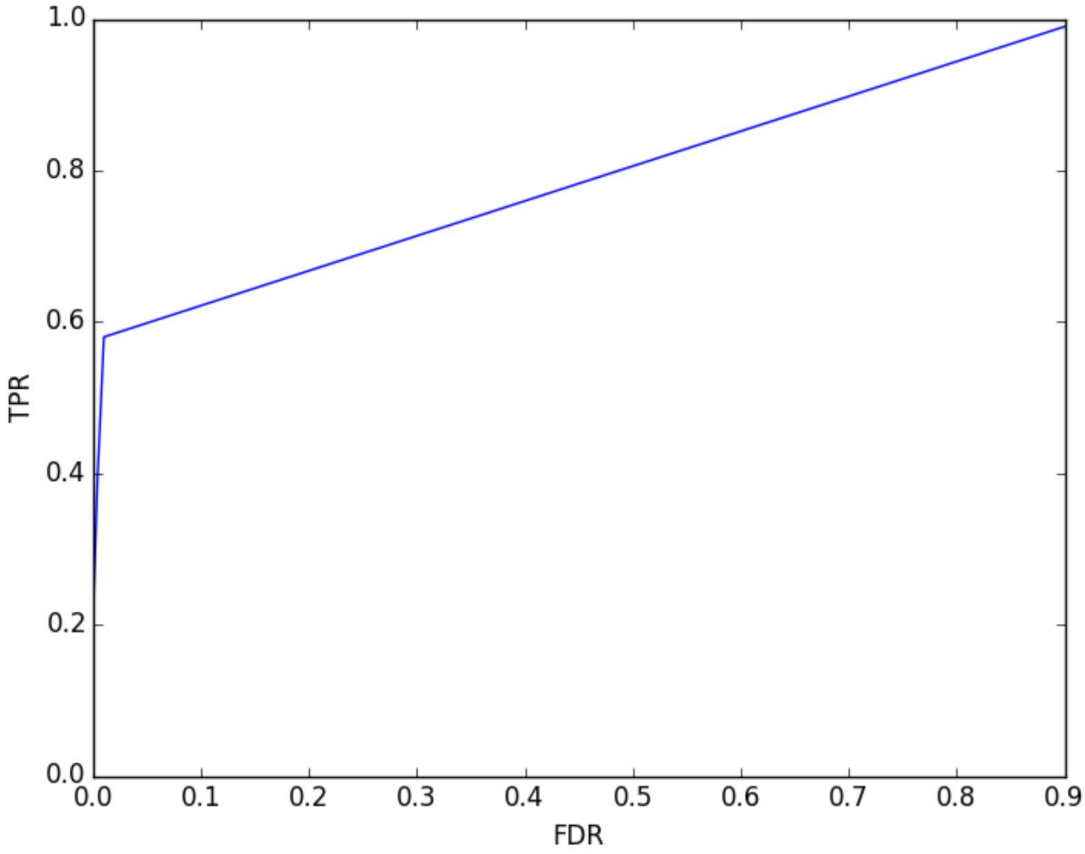
Problem 4

```
import numpy as np
import matplotlib.pyplot as plt

# Load a csv of floats:
X = np.genfromtxt('C:\\local\\yelp_data\\upvote_data.csv', delimiter=",")
# Load a text file of integers:
y = np.loadtxt('C:\\local\\yelp_data\\upvote_labels.txt', dtype=np.int)
# Load a text file of strings:
featureNames = open('C:\\local\\yelp_data\\upvote_features.txt').read().splitlines()

print("X shape", X.shape)
print("y shape", y.shape)

y = np.sqrt(y)

X_train = X[0:4000, :]
X_valid = X[4000:5000, :]
X_test = X[5000:6000, :]

y = y.reshape(-1, 1)

y_train = y[0:4000, :]
```

3 3.5 / 5

✓ - 1 pts comment of plot 2 is missing.

✓ - 0.5 pts x-axis in plot 1 should be in log scale.



When the value of lambda is high, very few features are selected and as we keep on reducing the regularization parameter, more features are being selected.

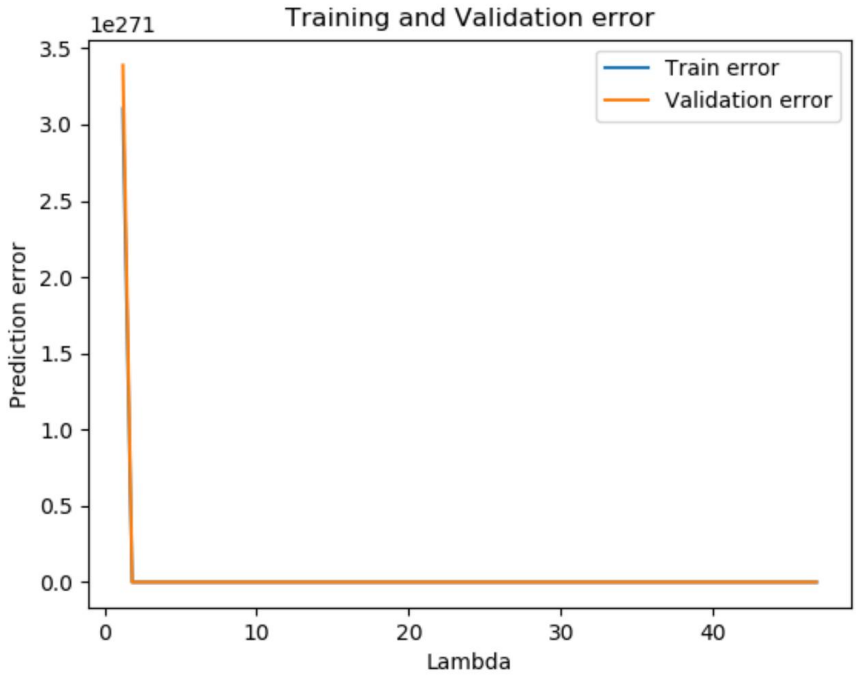
Problem 4

```
import numpy as np
import matplotlib.pyplot as plt

# Load a csv of floats:
X = np.genfromtxt('C:\\local\\yelp_data\\upvote_data.csv', delimiter=",")
# Load a text file of integers:
y = np.loadtxt('C:\\local\\yelp_data\\upvote_labels.txt', dtype=np.int)
# Load a text file of strings:
featureNames = open('C:\\local\\yelp_data\\upvote_features.txt').read().splitlines()

print("X shape", X.shape)
print("y shape", y.shape)

y = np.sqrt(y)

X_train = X[0:4000, :]
X_valid = X[4000:5000, :]
X_test = X[5000:6000, :]

y = y.reshape(-1, 1)

y_train = y[0:4000, :]
```

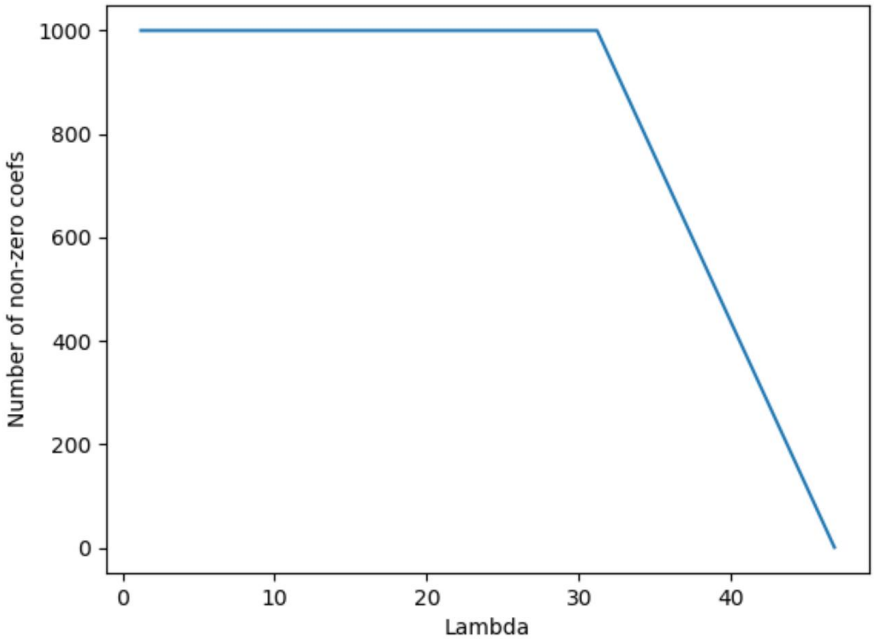
```
y_valid = y[4000:5000, :]
y_test = y[5000:6000, :]

lam_values = 2*abs(np.dot(X_train.T, (y_train - np.mean(y_train))))
lam = np.amax(lam_values)
print("Starting lambda: ", lam)
d = X_train.shape[1]

wvec = np.full(d, 0.0)
W_est = np.full(d, 5.0)
W_mat = np.empty([0, d])
nzero_coefmat = np.empty([0, 2])
errors_train_val = np.empty([0, 4])  # hyp par, train err, val err, itr num

conv_limit = 0.5
conv_cond = np.allclose(W_est, wvec, conv_limit)
itr = 0

condition = True

while condition and itr < 10:
    hyp_par = lam/(1.5**itr)
    itr = itr + 1
    print("\nlambda: ", hyp_par)
    print("This is lambda iteration: ", itr)

    conv_cond = False
    counter = 0
    #while not conv_cond and counter <= 5:
    while not conv_cond and counter <= 5:
        counter += 1
        #print("\n This is w convergence iteration: ", counter)
        #print("what wvec is used for b: ", wvec)
        b = np.mean(y_train - np.dot(X_train, wvec))
        #print("b: ", b)
        W_est = list(wvec)

        for j in np.arange(d):
            a = 2*sum(X_train[:, j]**2)
            w_ex = np.copy(W_est)
            w_ex[j] = 0

            wx_prod = np.dot(X_train, w_ex).reshape(-1,1)
            diff = y_train - (b + wx_prod)
            c = np.asscalar(2*np.dot(X_train[:, j], diff))
            c = 2*np.dot(X_train[:, j], diff)
            #print("c: ", c)

            if c < - hyp_par:
                w = 1.*((c + hyp_par) / a)
            elif - hyp_par <= c <= hyp_par:
                w = 0.0
            else:
                w = 1.*((c - hyp_par) / a)

            wvec[j] = w

        if np.count_nonzero(wvec) == 0:
```

```
            break
        else:
            conv_cond = np.allclose(W_est, wvec, conv_limit)

    pred_train = X_train@wvec.reshape(-1, 1)
    train_err = np.mean(np.square(y_train - pred_train))

    pred_valid = X_valid@wvec.reshape(-1, 1)
    valid_err = np.mean(np.square(y_valid - pred_valid))
    errors_train_val = np.append(errors_train_val, np.array([hyp_par, train_err, valid_err, itr]).re

    W_mat = np.append(W_mat, wvec.reshape(1, d), axis=0)
    nzero_coefmat = np.append(nzero_coefmat, np.array([hyp_par, np.count_nonzero(wvec)]).reshape(1,
    condition = np.count_nonzero(wvec) <= d


print("How many times while loop for lambda has run: ", itr)

plt.clf()
plt.plot(errors_train_val[:, 0], errors_train_val[:, 1], label="Train error")
plt.plot(errors_train_val[:, 0], errors_train_val[:, 2], label="Validation error")
plt.xlabel("Lambda")
plt.ylabel("Prediction error")
plt.title("Training and Validation error")
plt.legend(loc='best')
plt.show()
#plt.savefig('./HW2/errors_coef.png')

plt.plot(nzero_coefmat[:, 0], nzero_coefmat[:, 1], label="Non-zero coef and lambda")
plt.xlabel("Lambda")
plt.ylabel("Number of non-zero coefs")
plt.show()
#plt.savefig('./HW2/q4_nonzero_coef.png')

print("Error matrix:\n", errors_train_val)

ind = np.argmin(errors_train_val[:, 2])
print(ind)

pick_lam = errors_train_val[ind, 0]
itr_num = errors_train_val[ind, 3]

print("Best lambda: ", pick_lam)
print("Which iteration: ", int(itr_num))

print("Training error: ", errors_train_val[ind, 1])
print("Validation error: ", errors_train_val[ind, 2])

w_opt = W_mat[int(itr_num)-1, :]

pred_test = X_test@w_opt.reshape(-1, 1)
test_err = np.mean(np.square(y_test - pred_test))
print("Test error: ", test_err)

print("Optimal w:\n", w_opt)
sort_idx = np.argsort(w_opt)[::-1][0:9]
print("largest weight indices: ", sort_idx)
#w_sort = w_opt.argsort()[0:9]
```

```
print("Largest weight: ", w_opt[sort_idx])
#print("Features with largest weight: ", featureNames[sort_idx])
```



(a)



(b) Training error: 1.4414999999999965 Validation error: 1.409999999999996 Test error: 1.3569999999999969

(c)

Problem 5

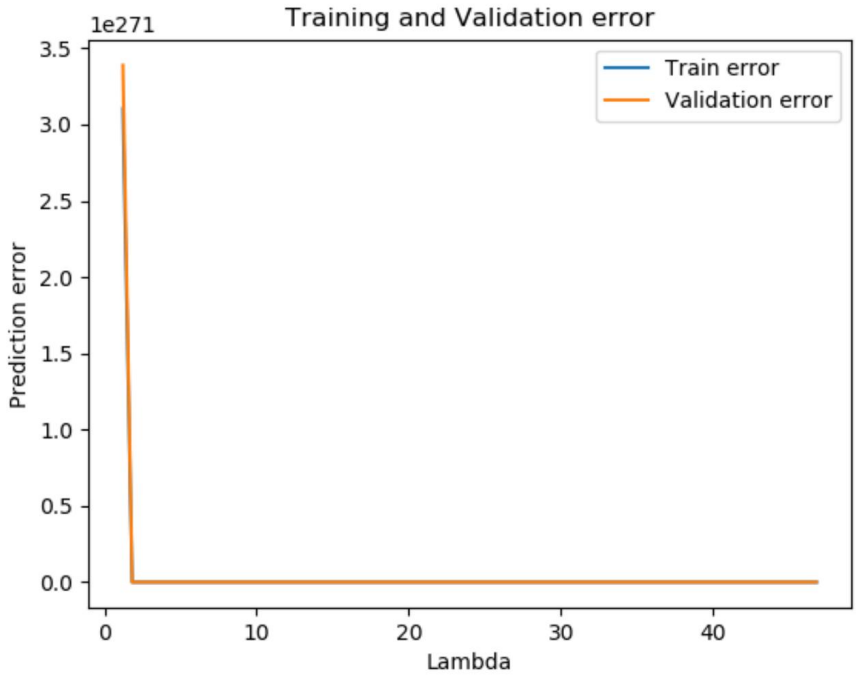
(a) The gradients are as follows -
The gradient with respect to b, is $\frac{\partial J}{\partial b} = \frac{1}{n}\Sigma_{i=1}^{n}\mu_i y_i - \bar{y}$

4 3 / 5

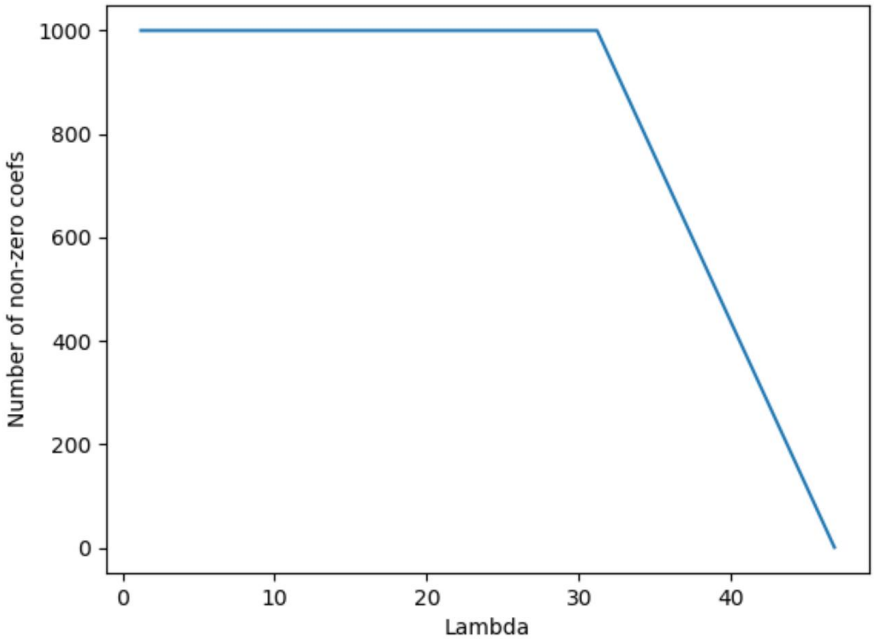
✓ - 1 pts plot for errors v.s. lambda is not correct.

✓ - 1 pts the 10 features are missing

```
print("Largest weight: ", w_opt[sort_idx])
#print("Features with largest weight: ", featureNames[sort_idx])
```



(a)



(b) Training error: 1.4414999999999965 Validation error: 1.409999999999996 Test error: 1.3569999999999969

(c)

Problem 5

(a) The gradients are as follows -
The gradient with respect to b, is $\frac{\partial J}{\partial b} = \frac{1}{n}\Sigma_{i=1}^{n}\mu_i y_i - \bar{y}$

The Hessian with respect to b, is
$\frac{\partial^2 J}{\partial b^2} = \frac{1}{n}\Sigma_{i=1}^{n} \frac{\partial \mu_i}{\partial b} y_i = \frac{exp(-y_i)(b+x_i^T w)y_i}{[1+exp(-y_i)(b+x_i^T w)]^2} = (\mu_i - \mu_i^2)y_i$
So, $\frac{\partial^2 J}{\partial b^2} = \frac{1}{n}\Sigma_{i=1}^{n}(\mu_i - \mu_i^2)y_i^2$

$\nabla_w J$ is a $d \times 1$vector with$\frac{\partial J}{\partial w_j}$ where j = 1,2,...d
$\frac{\partial J}{\partial w_j} = \frac{1}{n}\Sigma_{i=1}^{n}(\mu_i - 1)x_{ij}y_i + 2\lambda w_j$

The Hessian $\nabla_w^2 J$ is a $d \times d$ matrix where diagonal elements are of the form $\frac{\partial^2 J}{\partial w_j^2} = \frac{1}{n}\Sigma_{i=1}^{n}(\mu_i - \mu_i^2)x_{ij}^2 y_i^2 + 2\lambda$

Non-diagonal elements are of the form $\frac{\partial^2 J}{\partial w_j w_k} = \frac{1}{n}\Sigma_{i=1}^{n}(\mu_i - \mu_i^2)x_{ij}x_{ik}y_i^2$ where $i = 1, 2, .., n$, and $j, k = 1, 2, ..., d$

The following lines of code solve parts (b), (c), (d) and (e)

```
import numpy as np
from mnist import MNIST
import matplotlib.pyplot as plt

def load_dataset():
    mndata = MNIST('C:\\local\\MNIST_data\\')
    mndata.gz = True
    x_train, labels_train = map(np.array, mndata.load_training())
    x_test, labels_test = map(np.array, mndata.load_testing())
    x_train = x_train/255.0
    x_test = x_test/255.0
    return x_train, labels_train, x_test, labels_test

def mu(wvec, b, xvec, y):
    return 1/(1 + np.exp(-y*(b + np.dot(xvec, wvec))))

def grad_b(w, b, X, Y):
    sum = 0
    n = X.shape[0]
    for i in np.arange(n):
        sum = sum + mu(w, b, X[i, :], Y[i, 0])

    sum = sum / n
    sum = sum - np.mean(Y)
    return sum

def hess_b(w, b, X, Y):
    n = X.shape[0]
    prod = np.full(n, 0.0)
    for i in np.arange(n):
        temp = mu(w, b, X[i, :], Y[i, 0])
        prod[i] = temp*(1 - temp)*Y[i, 0]**2
    return np.mean(prod)

def grad_w(w, b, X, Y, lam):
    n = X.shape[0]
    d = X.shape[1]
    #grad_vec = np.full(d, 0.0)
    Mu = np.full(n, 0.0)
    sum = 0
    for i in np.arange(n):
        Mu[i] = mu(w, b, X[i, :], Y[i, 0]) - 1
```

```
    Mu = Mu.reshape(-1, 1)
    #print("Shape of Mu inside function: ", Mu.shape)

    Mu_Y = np.multiply(Mu, Y).reshape(-1, 1)  # this is of order (n X 1)

    grad_vec = (X.T@Mu_Y)/n + 2*lam*w.reshape(-1,1)
    grad_vec = grad_vec.reshape(-1, 1)
    return grad_vec

def hess_w(w, b, X, Y, lam):
    n = X.shape[0]
    d = X.shape[1]

    Mu = np.full(n, 0.0)
    sum = 0
    for i in np.arange(n):
        Mu[i] = mu(w, b, X[i, :], Y[i, 0]) - 1
    Mu = Mu.reshape(-1, 1)
    prod = np.diag(np.diag(Mu@(1- Mu).T))
    hess_mat = X.T@prod@X + 2*lam*np.identity(d)
    return hess_mat


def Jfun_val(wvec, b, X, Y, lam):
    temp1 = np.multiply(X@wvec.reshape(-1, 1) + b, Y)  # should be nX1
    #print("Elements of temp1", temp1[0:10,: ])
    temp2 = np.log(1 + np.exp(temp1))
    val = np.mean(temp2) + lam*np.linalg.norm(wvec)
    return val


X_train, train_labels, X_test, test_labels = load_dataset()
# print(train_labels.dtype)
# print(train_labels[0:20])

p = (train_labels == 7)
q = (train_labels == 2)
r = np.logical_or(p, q)
s = np.where(r)

train_labels = train_labels[s]
X_train = X_train[s]
t = np.where(train_labels == 7)
u = np.where(train_labels == 2)

train_labels.dtype = np.int8

# Y = 1 for 7 and Y = -1 for 2
train_labels[t] = 1
train_labels[u] = -1

train_labels = train_labels.reshape(-1,1)
#####################################################################
# Test set
p = (test_labels == 7)
q = (test_labels == 2)
r = np.logical_or(p, q)
s = np.where(r)
```

```
test_labels = test_labels[s]
X_test = X_test[s]
t = np.where(test_labels == 7)
u = np.where(test_labels == 2)

# Changing data from unsigned int to signed int
test_labels.dtype = np.int8
test_labels[t] = 1
test_labels[u] = -1
test_labels = test_labels.reshape(-1,1)

lam = 0.1
step_size = 0.1

######################################################################
# Gradient Descent Method
# Run it for the training set
X = X_train
Y = train_labels
n = X.shape[0]
d = X.shape[1]
wvec = np.full(d, 0.0)
b = 0
fun_val = np.empty([0, 3])  # iteration num, func val for train, func val for test
misclass_error = np.empty([0, 3])  # iteration num, misclass for train, misclass for test
norm_w = np.empty([0, 2])  # itr, norm of w

itr = 0
converged = False

while not converged:
    fun_val_old = Jfun_val(wvec, b, X, Y, lam)
    grad_wval = grad_w(wvec, b, X, Y, lam)
    grad_bval = grad_b(wvec, b, X, Y)

    w_update = wvec.reshape(-1, 1) - step_size*grad_wval
    b_update = b - step_size*grad_bval
    fun_val_new = Jfun_val(w_update, b_update, X, Y, lam)
    fun_val_test = Jfun_val(w_update, b_update, X_test, test_labels, lam)
    converged = np.allclose(fun_val_old, fun_val_new, 0.001)
    wvec = w_update
    b = b_update
    itr = itr + 1
    no_match_train = np.count_nonzero(train_labels - np.sign(X_train@wvec + b))
    misclass_err_train = no_match_train/X_train.shape[0]

    no_match_test = np.count_nonzero(test_labels - np.sign(X_test@wvec + b))
    misclass_err_test = no_match_test/X_test.shape[0]

    fun_val = np.append(fun_val, np.array([itr, fun_val_new, fun_val_test]).reshape(1, 3), axis=
    misclass_error = np.append(misclass_error, np.array([itr, misclass_err_train, misclass_err_
    norm_w = np.append(norm_w, np.array([itr, np.linalg.norm(wvec)]).reshape(1, 2), axis=0)


# Save estimated b and w's from training Set - to be used on test set prediction
wopt_train = wvec
bopt_train = b
```

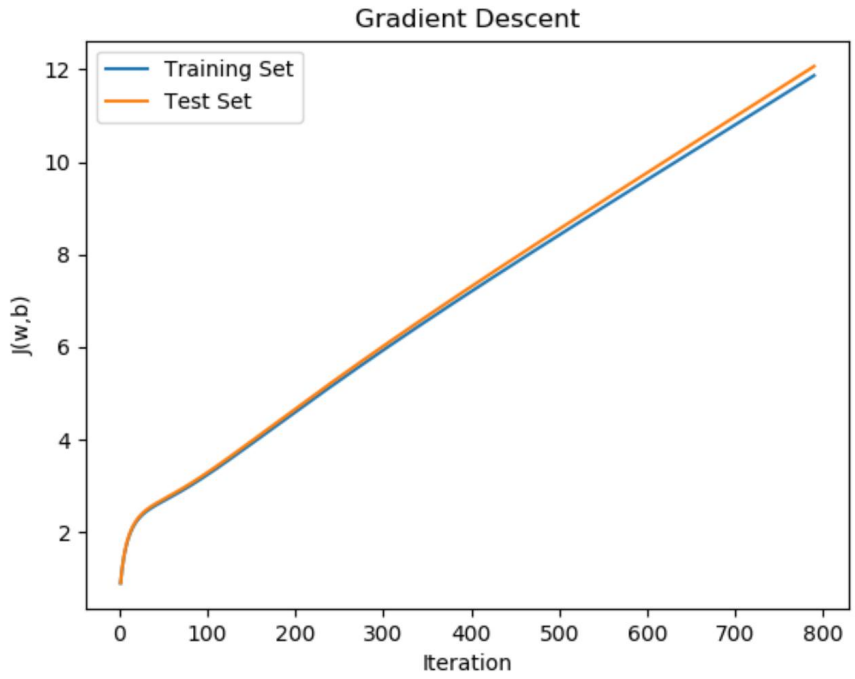
```
plt.clf()
plt.plot(fun_val[:, 0], fun_val[:, 1], label = "Training Set")
plt.plot(fun_val[:, 0], fun_val[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("J(w,b)")
plt.legend(loc='best')
plt.title("Gradient Descent")
plt.show()

plt.clf()
plt.plot(misclass_error[:, 0], misclass_error[:, 1], label = "Training Set")
plt.plot(misclass_error[:, 0], misclass_error[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("Misclassification error")
plt.title("Gradient Descent")
plt.legend(loc='best')
plt.show()


###################################################################### Stochastic Gradient Desce

step_size = 0.1
wvec = np.full(d, 0.0)
b = 0
fun_val = np.empty([0, 3])  # iteration num, func val for train, func val for test
misclass_error = np.empty([0, 3])  # iteration num, misclass for train, misclass for test
train_indx = np.random.choice(np.arange(X_train.shape[0]), 1, replace=False)

itr = 0
converged = False

while not converged:
    fun_val_old = Jfun_val(wvec, b, X_train[train_indx, :], train_labels[train_indx, :], lam)
    grad_wval = grad_w(wvec, b, X_train[train_indx, :], train_labels[train_indx, :], lam)
    grad_bval = grad_b(wvec, b, X_train[train_indx, :], train_labels[train_indx, :])

    w_update = wvec.reshape(-1, 1) - step_size*grad_wval
    b_update = b - step_size*grad_bval
    fun_val_new = Jfun_val(w_update, b_update, X_train[train_indx, :], train_labels[train_indx,
    fun_val_test = Jfun_val(w_update, b_update, X_test, test_labels, lam)
    converged = np.allclose(fun_val_old, fun_val_new, 0.001)
    wvec = w_update
    b = b_update
    itr = itr + 1
    no_match_train = np.count_nonzero(train_labels - np.sign(X_train@wvec + b))
    misclass_err_train = no_match_train/X_train.shape[0]

    no_match_test = np.count_nonzero(test_labels - np.sign(X_test@wvec + b))
    misclass_err_test = no_match_test/X_test.shape[0]

    fun_val = np.append(fun_val, np.array([itr, fun_val_new, fun_val_test]).reshape(1, 3), axis=
    misclass_error = np.append(misclass_error, np.array([itr, misclass_err_train, misclass_err_

# Plot func value on the same plot
plt.clf()
plt.plot(fun_val[:, 0], fun_val[:, 1], label = "Training Set")
plt.plot(fun_val[:, 0], fun_val[:, 2], label = "Test Set")
```

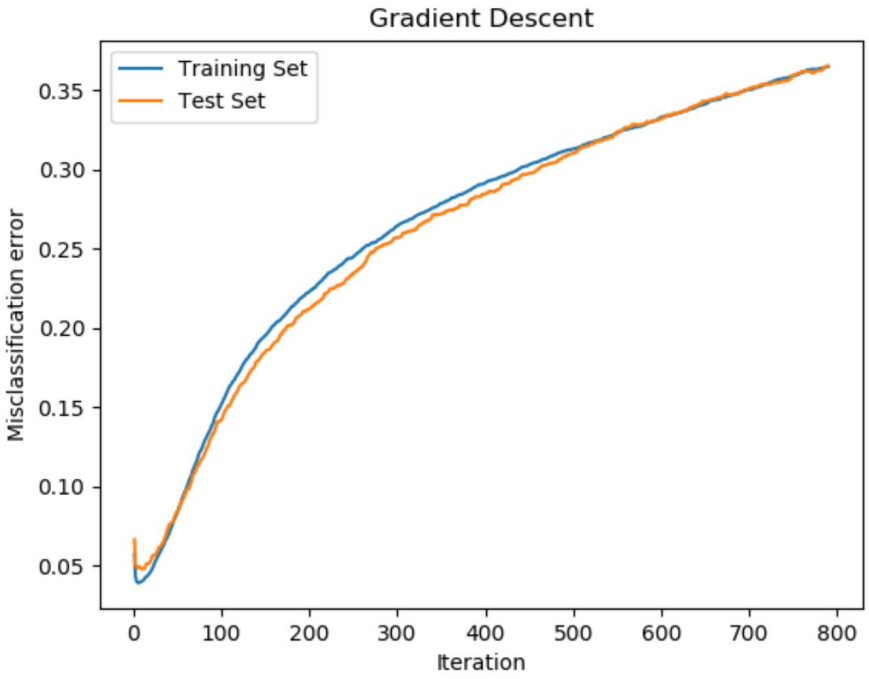
```
plt.xlabel("Iteration")
plt.ylabel("J(w,b)")
plt.legend(loc='best')
plt.title("Stochastic Gradient Descent with batch=1")
plt.show()

plt.clf()
plt.plot(misclass_error[:, 0], misclass_error[:, 1], label = "Training Set")
plt.plot(misclass_error[:, 0], misclass_error[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("Misclassification error")
plt.legend(loc='best')
plt.title("Stochastic Gradient Descent with batch=1")
plt.savefig('./HW2/misclass_train_test_sgd1.png')


# Stochastic Gradient Descent with batch size = 100
step_size = 0.1
wvec = np.full(d, 0.0)
b = 0
fun_val = np.empty([0, 3])  # iteration num, func val for train, func val for test
misclass_error = np.empty([0, 3])  # iteration num, misclass for train, misclass for test
train_indx = np.random.choice(np.arange(X_train.shape[0]), 100, replace=False)

itr = 0
converged = False

while not converged:
    fun_val_old = Jfun_val(wvec, b, X_train[train_indx, :], train_labels[train_indx, :], lam)
    grad_wval = grad_w(wvec, b, X_train[train_indx, :], train_labels[train_indx, :], lam)
    grad_bval = grad_b(wvec, b, X_train[train_indx, :], train_labels[train_indx, :])

    w_update = wvec.reshape(-1, 1) - step_size*grad_wval
    b_update = b - step_size*grad_bval
    fun_val_new = Jfun_val(w_update, b_update, X_train[train_indx, :], train_labels[train_indx,
    fun_val_test = Jfun_val(w_update, b_update, X_test, test_labels, lam)
    converged = np.allclose(fun_val_old, fun_val_new, 0.001)
    wvec = w_update
    b = b_update
    itr = itr + 1
    no_match_train = np.count_nonzero(train_labels - np.sign(X_train@wvec + b))
    misclass_err_train = no_match_train/X_train.shape[0]

    no_match_test = np.count_nonzero(test_labels - np.sign(X_test@wvec + b))
    misclass_err_test = no_match_test/X_test.shape[0]

    fun_val = np.append(fun_val, np.array([itr, fun_val_new, fun_val_test]).reshape(1, 3), axis=
    misclass_error = np.append(misclass_error, np.array([itr, misclass_err_train, misclass_err_


# Plot func value on the same plot
plt.clf()
plt.plot(fun_val[:, 0], fun_val[:, 1], label = "Training Set")
plt.plot(fun_val[:, 0], fun_val[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("J(w,b)")
plt.legend(loc='best')
plt.title("Stochastic Gradient Descent with batch=100")
```

```
plt.show()

plt.clf()
plt.plot(misclass_error[:, 0], misclass_error[:, 1], label = "Training Set")
plt.plot(misclass_error[:, 0], misclass_error[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("Misclassification error")
plt.title("Stochastic Gradient Descent with batch=100")
plt.legend(loc='best')
plt.show()

######################################################################
# Newton method
# We need the Hessian for it. The Hessian for this case can be expressed
# as H = X_TSX where S = diag(mu_i(1 - mu_i))
# Run it for the training set
n = X_train.shape[0]
d = X_train.shape[1]
wvec = np.full(d, 0.0)
b = 0
fun_val = np.empty([0, 3])  # iteration num, func val for train, func val for test
misclass_error = np.empty([0, 3])  # iteration num, misclass for train, misclass for test

itr = 0
converged = False

while not converged:
    fun_val_old = Jfun_val(wvec, b, X_train, train_labels, lam)
    grad_wval = grad_w(wvec, b, X_train, train_labels, lam)
    hess_wval = hess_w(wvec, b, X_train, train_labels, lam)
    grad_bval = grad_b(wvec, b, X_train, train_labels)
    hess_bval = hess_b(wvec, b, X_train, train_labels)
    hess_inv = np.linalg.solve(hess_wval, np.identity(d))
    w_update = wvec.reshape(-1, 1) - step_size*(hess_inv@grad_wval)
    b_update = b - (step_size*grad_bval)/hess_bval
    fun_val_new = Jfun_val(w_update, b_update, X, Y, lam)
    fun_val_test = Jfun_val(w_update, b_update, X_test, test_labels, lam)
    converged = np.allclose(fun_val_old, fun_val_new, 0.001)
    wvec = w_update
    b = b_update
    itr = itr + 1
    no_match_train = np.count_nonzero(train_labels - np.sign(X_train@wvec + b))
    misclass_err_train = no_match_train/X_train.shape[0]

    no_match_test = np.count_nonzero(test_labels - np.sign(X_test@wvec + b))
    misclass_err_test = no_match_test/X_test.shape[0]

    fun_val = np.append(fun_val, np.array([itr, fun_val_new, fun_val_test]).reshape(1, 3), axis=
    misclass_error = np.append(misclass_error, np.array([itr, misclass_err_train, misclass_err_

# Plot func value on the same plot
plt.clf()
plt.plot(fun_val[:, 0], fun_val[:, 1], label = "Training Set")
plt.plot(fun_val[:, 0], fun_val[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("J(w,b)")
plt.title("Function Value: Newton's Method")
plt.legend(loc='best')
```

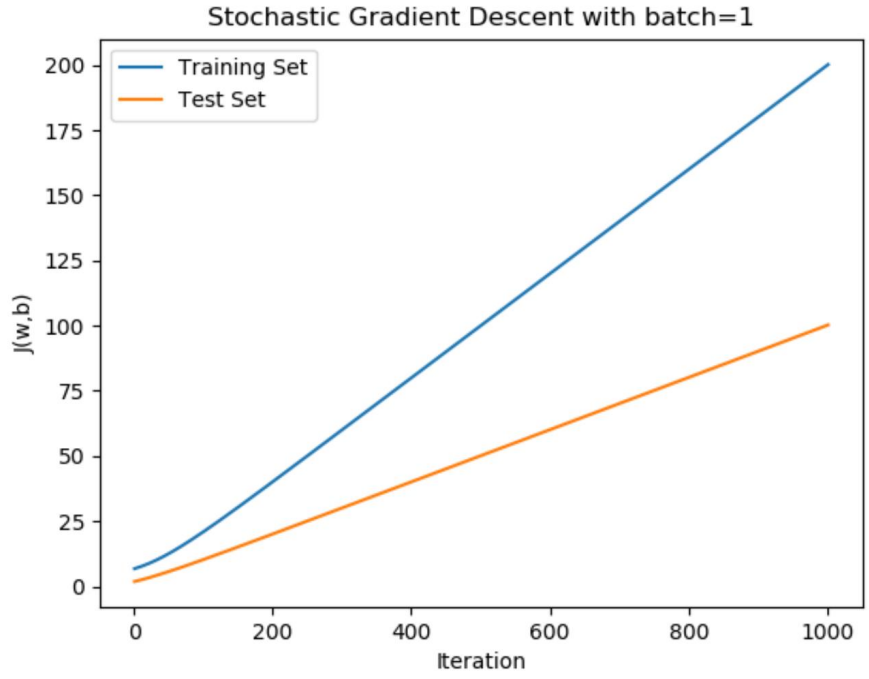
```
plt.show()

plt.clf()
plt.plot(misclass_error[:, 0], misclass_error[:, 1], label = "Training Set")
plt.plot(misclass_error[:, 0], misclass_error[:, 2], label = "Test Set")
plt.xlabel("Iteration")
plt.ylabel("Misclassification error")
plt.title("Misclassification error: Newton's Method")
plt.legend(loc='best')
plt.show()
```

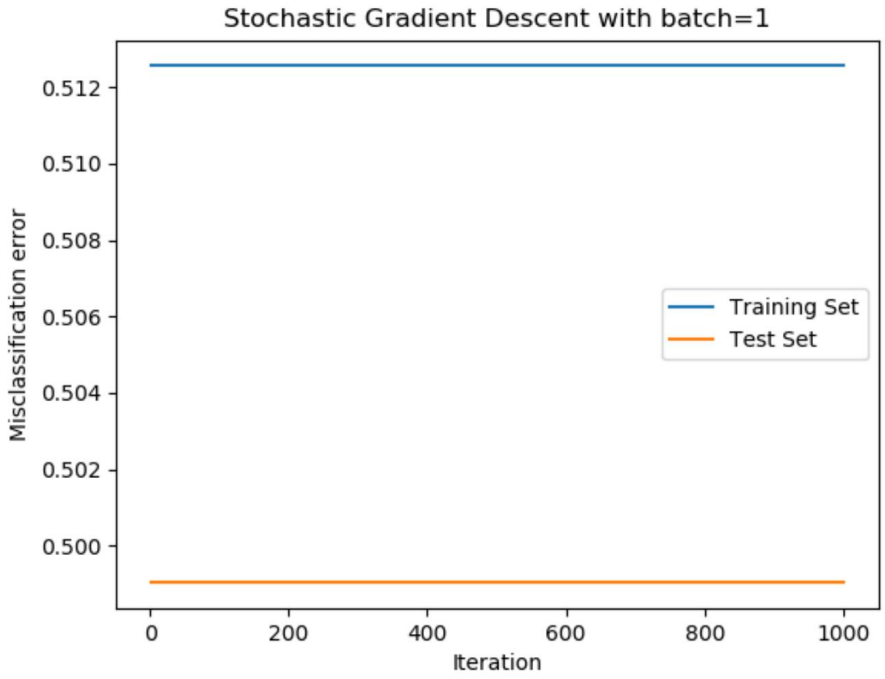
(b) Find the relevant code for part (b) above



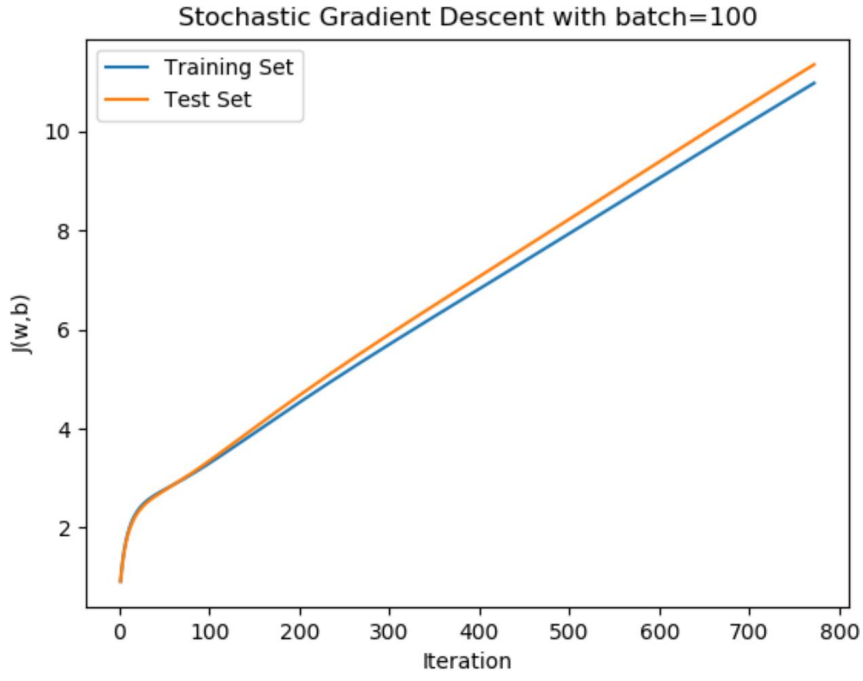


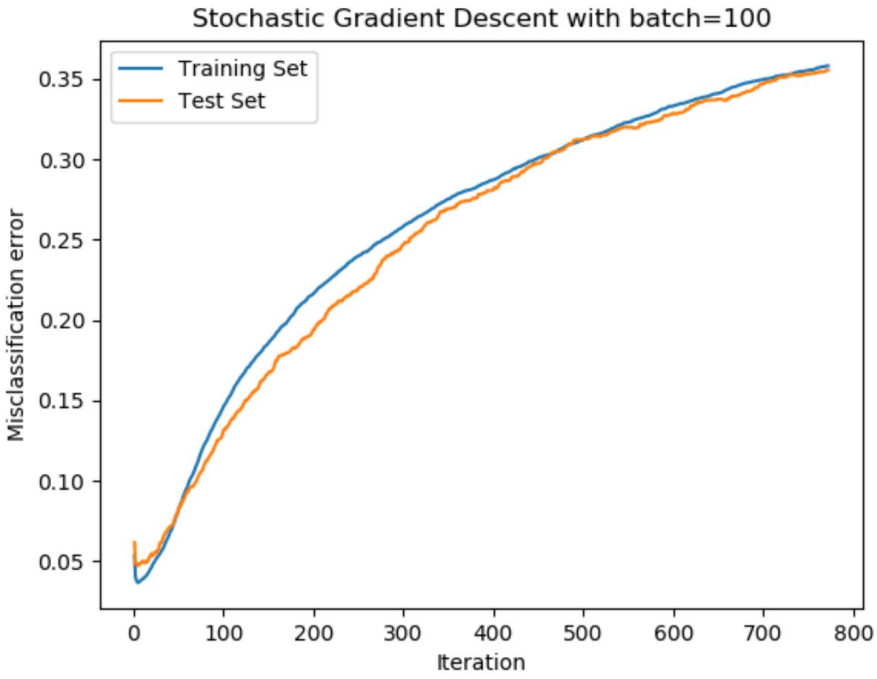
(c) Find the relevant code for part (c) above



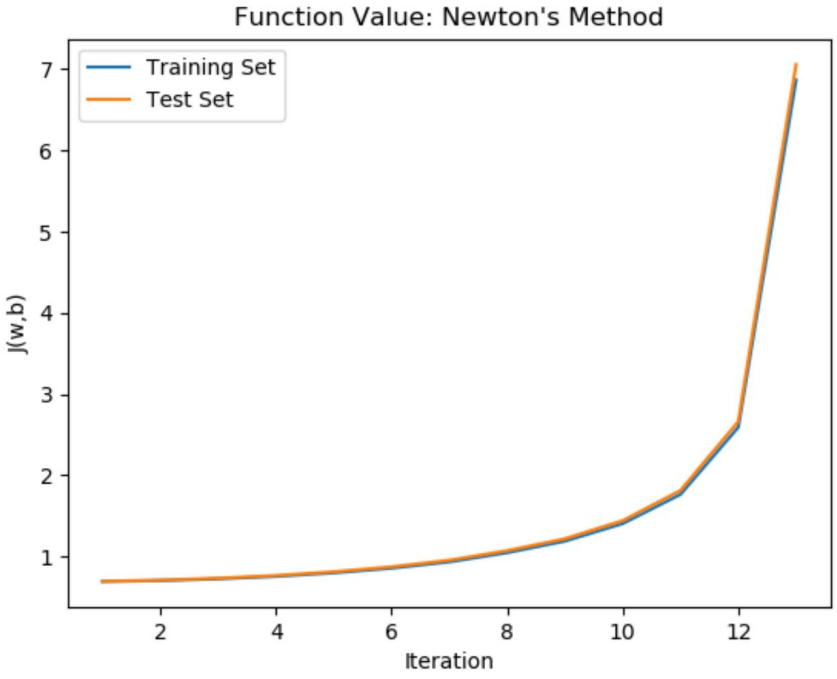


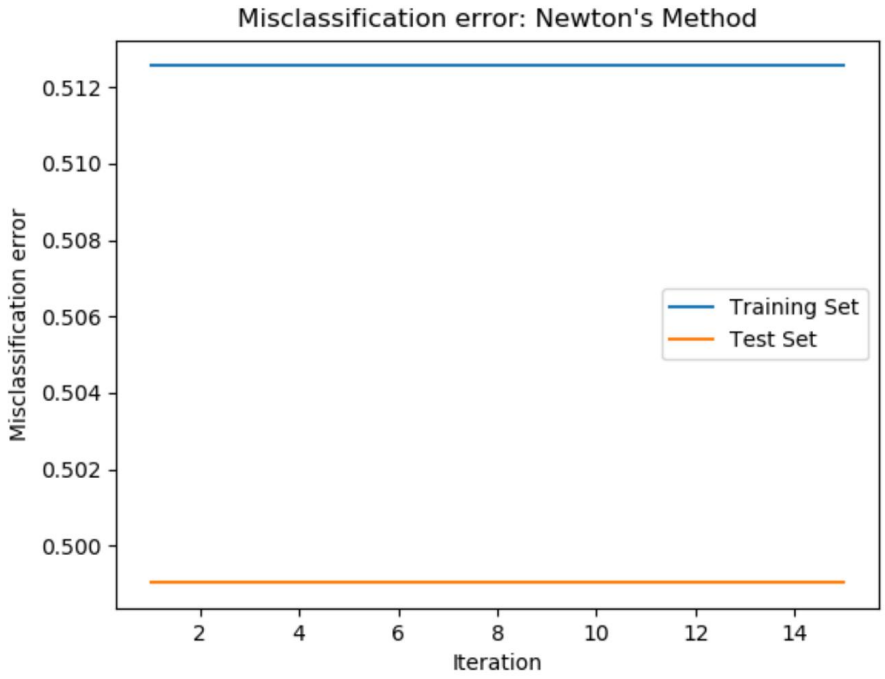
(d) Find the relevant code for part (d) above





(e) Find the relevant code for part (e) above





*** Study group partners: Vidur Vij, Samin Jalali

5.1 Gradients and Hessians 0.5 / 1

- ✓ **+ 0.25 pts** Gradient with respect to w correct
- ✓ **+ 0.25 pts** Hessian with respect to w correct
- ✓ **+ 0.25 pts** Gradient with respect to b correct
- ✓ **+ 0.25 pts** Hessian with respect to b correct
- ✓ **- 0.5 pts** Derivations missing
  - **+ 0 pts** Incorrect
  - **+ 0 pts** No answer submitted

5.2 Coding 0 / 4

- ✓ **- 0.5 pts** GD: Error in the loss function
- ✓ **- 0.5 pts** GD: Error in the classification error
- ✓ **- 0.5 pts** SGD: Error in the loss function
- ✓ **- 0.5 pts** SGD: Error in the classification error
- ✓ **- 0.5 pts** Batch SGD: Error in the loss function
- ✓ **- 0.5 pts** Batch SGD: Error in the classification error
- ✓ **- 0.5 pts** Newton's Method: Error in the loss function
- ✓ **- 0.5 pts** Newton's Method: Error in the classification error