Reid Jackson

CS4442 Assignment 1

Feb 16th 2020

(1)

CS4442 Assign 1

250914839
Reid Jackson

1.a) $f(w) = w^T x$

$x \in \mathbb{R}^n$ n-dim vector
$w \in \mathbb{R}^n$ n-dim column vector

$$f(w) = w^T x = [w_1\ w_2\ w_3 \dots w_n] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} w_i x_i$$

$$\frac{df}{dw_i} = x_i \quad \forall i = 1,2,3,\dots,n \Rightarrow \nabla f(w) = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = x$$

b.) $f(w) = tr(ww^T A)$

w defined above
$A \in \mathbb{R}^{n \times n}$ square matrix

Given property: $tr(AB) = tr(BA)$

$tr(ww^T A) = tr(w^T A w)$ $f(w) = tr(w^T A w)$

$$= [w_1\ w_2\ w_3 \dots w_n] \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix}$$

$$= \left[w_1 \sum_{j=1}^{n} a_{j1} \dots w_2 \sum_{j=1}^{n} a_{j2} \dots w_n \sum_{j=1}^{n} a_{jn}\right] \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix}$$

$$= \sum_{i=1}^{n} w_i^2 \sum_{j=1}^{n} a_{ji} \quad \forall i = 1,2,3,\dots,n$$

$$\Rightarrow \nabla f(w) = \begin{bmatrix} 2w_1 \sum_{j=1}^{n} a_{j1} \\ 2w_2 \sum_{j=1}^{n} a_{j2} \\ \vdots \\ 2w_n \sum_{j=1}^{n} a_{jn} \end{bmatrix}$$

c.) Hessian Matrix $H \in \mathbb{R}^{n \times n}$ of wrt w

$f(w) = tr(ww^TA)$

from b) $f(w) = tr(ww^TA) = tr(w^TAw)$

$$f(w) = \sum_{i=1}^{n} w_i^2 \sum_{j=1}^{n} a_{ji}$$

$$\frac{df}{dw_i} = 2w_i \sum_{j=1}^{n} a_{ji} \quad \forall i = 1,2,3,\ldots,n \qquad \frac{df}{dw_i^2} = 2\sum_{j=1}^{n} a_{ji} \quad \forall i = 1,2,3,\ldots,n$$

$$\frac{df}{dw_i dw_k} = 0 \quad \forall k \neq i$$

$$H = \begin{bmatrix} 2\sum_{j=1}^{n} a_{j1} & 0 & \cdots & 0 \\ 0 & 2\sum_{j=1}^{n} a_{j2} & \cdots & 0 \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & & \\ 0 & 0 & \cdots & 2\sum_{j=1}^{n} a_{jn} \end{bmatrix}$$

d.) Given $a = w^Tx$ $f(w) = \log(\sigma(w^Tx))$

$$\sigma(a) = \frac{1}{1+e^{-a}} \quad w^Tx = \sum_{i=1}^{n} w_i x_i$$

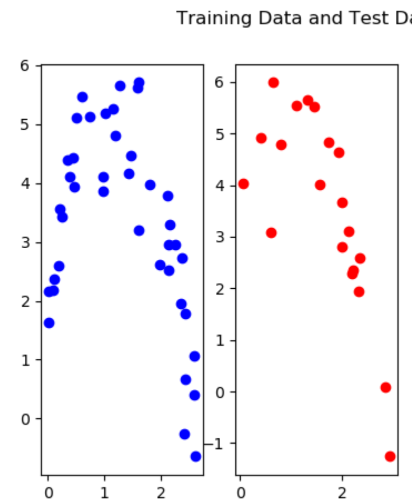
Then $f(w) = \log\left(\frac{1}{1+e^{-\sum_{i=1}^{n} w_i x_i}}\right)$ $f_w = -\log\left(1+e^{-\sum_{i=1}^{n} w_i x_i}\right)$

$$\frac{df}{dw_j} = \frac{d}{dw_j}\left(-\log\left(1+e^{-\sum_{i=1}^{n} w_i x_i}\right)\right) \quad \frac{df}{dw_j} = \frac{1}{1+e^{-\sum_{i=1}^{n} w_i x_i}} \left(\sum_{i=1} w_i x_i\right)(x_j) \quad \forall j = 1,2,3,\ldots,n$$

$$w^Tx = \sum_{i=1} w_i x_i$$

$$\nabla f(w) = \begin{bmatrix} \frac{1}{1+e^{-w^Tx}} (w^Tx)(x_1) \\ \frac{1}{1+e^{-w^Tx}} (w^Tx)(x_2) \\ \vdots \\ \frac{1}{1+e^{-w^Tx}} (w^Tx)(x_n) \end{bmatrix}$$

2a



2b

Equation: y = -0.79337581x + 4.35891503

Average Error = 2.17394558

2c

Equation: y = -0.79337581x + 4.35891503

Average Error = 1.59595861

2b and 2c Python Code

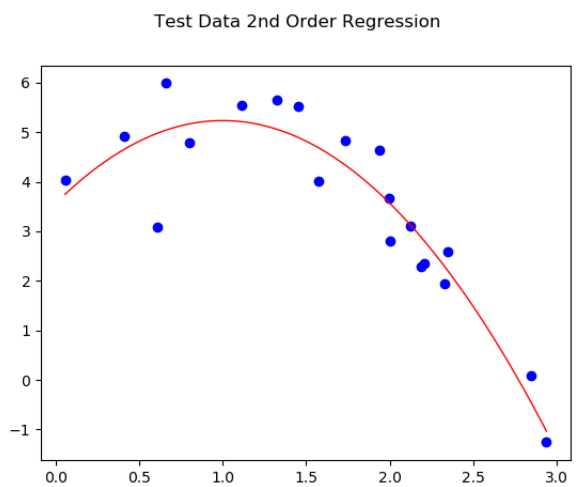
```
# Appends a column of ones to the features of training/test data, 1st order
onesColumn = np.ones((trainingF.shape[0]))
trainingFOnes = np.c_[trainingF, onesColumn]
onesColumn = np.ones((testF.shape[0]))
testFOnes = np.c_[testF, onesColumn]

# w = (X^T * X)^-1 * X^T * y, where X is s the data matrix augmented with a column of ones, and y is the column vector of target outputs.
# letting p = (X^T * X)^-1
# letting q = (X^T * X)^-1 * X^T

# does the first order regression of training data
trainingFOnesMatrix = np.asmatrix(trainingFOnes)
trainingOMatrix = np.asmatrix(trainingO)
p = np.linalg.inv((np.dot(trainingFOnes.T, trainingFOnes)))
q = np.dot(p, trainingFOnesMatrix.T)
# The matrix was turning from column vector to row vector, so Transposed back
w = np.dot(q, trainingOMatrix.T)
print(w)

# does the first order regression of test data
testFOnesMatrix = np.asmatrix(testFOnes)
testOMatrix = np.asmatrix(testO)
p = np.linalg.inv((np.dot(testFOnes.T, testFOnes)))
q = np.dot(p, testFOnesMatrix.T)
# The matrix was turning from column vector to row vector, so Transposed back
w = np.dot(q, testOMatrix.T)
print(w)

# Creates the x and y values of First Order Linear Regression for training
FirstOrderRegressionX = np.arange(np.amin(trainingF), np.amax(trainingF), 0.01)
FirstOrderRegressionY = w[0] * FirstOrderRegressionX + w[1]
# Uses the equation to find mean squared error for training
FirstOrderRegressionValues = w[0] * trainingF + w[1]
mse_sum = 0
for i in range(0, len(trainingF)):
    mse_sum += (trainingO[i] - FirstOrderRegressionValues.T[i])**2
mse = (mse_sum / len(trainingF))
print(mse)


# Creates the x and y values of First Order Linear Regression for test
FirstOrderRegressionX = np.arange(np.amin(testF), np.amax(testF), 0.01)
FirstOrderRegressionY = w[0] * FirstOrderRegressionX + w[1]
# Uses the equation to find mean squared error for test
FirstOrderRegressionValues = w[0] * testF + w[1]
mse_sum = 0
for i in range(0, len(testF)):
    mse_sum += (testO[i] - FirstOrderRegressionValues.T[i])**2
mse = (mse_sum / len(testF))
print(mse)
```

2d

Equation: y = $-2.29718151x^2 + 5.22039519x + 2.16940573$

Average Error = 0.4846845031271549

(also note the title is supposed to be training data)



Equation: y = $-1.67167226x^2 + 3.34520812x + 3.56132903$

Average Error = 0.44645401066211254

Given the better error values for the Second-Degree Regression, this is a better fit for the data over the Linear Regression (First Order).

Python Code

```
# Gets the x^2 values and appends them to the features of training/test data
trainingFSquared = np.c_[(trainingF)**2, trainingF]
testFSquared = np.c_[(testF)**2, testF]
# Appends a column of ones to the features of training/test data, 2nd order
onesColumn = np.ones((trainingF.shape[0]))
trainingFOnes = np.c_[trainingFSquared, onesColumn]
onesColumn = np.ones((testF.shape[0]))
testFOnes = np.c_[testFSquared, onesColumn]
print(trainingFOnes.shape)

# w = (X^T * X)^-1 * X^T * y, where X is s the data matrix augmented with a column of ones, and y is the column vector of target outputs.
# letting p = (X^T * X)^-1
# letting q = (X^T * X)^-1 * X^T
does the second order regression of training data
trainingFOnesMatrix = np.asmatrix(trainingFOnes)
trainingOMatrix = np.ravel(np.asmatrix(trainingO).T)
# print(trainingFOnesMatrix)
p = np.linalg.inv(trainingFOnesMatrix.T @ trainingFOnesMatrix)
q = p @ trainingFOnesMatrix.T
# The matrix was turning from column vector to row vector, so Transposed back
w = q @ trainingOMatrix
w = np.ravel(np.asarray(w))
print(w)

# Creates the x and y values of Second Order Polynomial Regression for training
SecondOrderRegressionX = np.arange(np.amin(trainingF), np.amax(trainingF), 0.01)
SecondOrderRegressionY = ((w[0] * (SecondOrderRegressionX)**2) + (w[1] * SecondOrderRegressionX) + w[2])
# Uses the equation to find mean squared error for training
SecondOrderRegressionValues = []
for i in range(0, len(trainingF)):
    SecondOrderRegressionValues.append(w[0] * (trainingF[i])**2 + w[1] * trainingF[i] + w[2])

SORVMatrix = np.ravel(np.asarray(SecondOrderRegressionValues))
# print(SORVMatrix)
mse_sum = 0
for i in range(0, len(trainingF)):
    mse_sum += (trainingO[i] - SORVMatrix[i])**2
mse = (mse_sum / len(trainingF))
print(mse)

does the second order regression of test data
testFOnesMatrix = np.asmatrix(testFOnes)
testOMatrix = np.ravel(np.asmatrix(testO).T)
# print(trainingFOnesMatrix)
p = np.linalg.inv(testFOnesMatrix.T @ testFOnesMatrix)
q = p @ testFOnesMatrix.T
# The matrix was turning from column vector to row vector, so Transposed back
w = q @ testOMatrix
w = np.ravel(np.asarray(w))
print(w)
```

```
# Creates the x and y values of Second Order Polynomial Regression for test
SecondOrderRegressionX = np.arange(np.amin(testF), np.amax(testF), 0.01)
SecondOrderRegressionY = ((w[0] * (SecondOrderRegressionX)**2) + (w[1] * SecondOrderRegressionX) + w[2])
# Uses the equation to find mean squared error for training
SecondOrderRegressionValues = []
for i in range(0, len(testF)):
    SecondOrderRegressionValues.append(w[0] * (testF[i])**2 + w[1] * testF[i] + w[2])

SORVMatrix = np.ravel(np.asarray(SecondOrderRegressionValues))
# print(SORVMatrix)
mse_sum = 0
for i in range(0, len(testF)):
    mse_sum += (testO[i] - SORVMatrix[i])**2
mse = (mse_sum / len(testF))
print(mse)
```

2e

Equation: $y = 0.17701719x^3 - 2.99261207x^2 + 5.92195383x + 2.04696968$

Average Error = 0.48055213344532505

Equation: $y = 1.44636153e\text{-}03x^3 - 1.67845534x^2 + 3.35391872x + 3.55893345$

Average Error = 0.4464533911621823

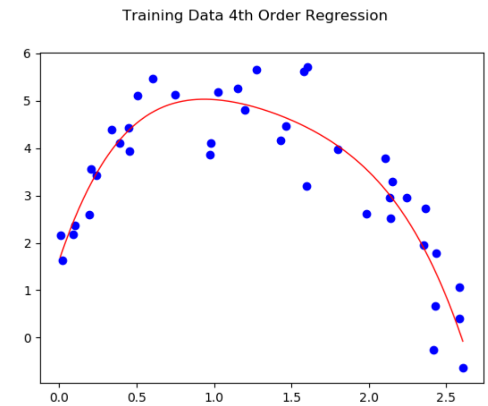
Based on the average errors, there is very little difference between 2nd and 3rd Order Regressions. Though, it is still much more accurate than 1st Order.

Python Code

```
# Gets the x^2 values and appends them to the features of training/test data
trainingFCubed = np.c_[(trainingF)**3, (trainingF)**2, trainingF]
testFCubed = np.c_[(testF)**3, (testF)**2, testF]
# Appends a column of ones to the features of training/test data, 2nd order
onesColumn = np.ones((trainingF.shape[0]))
trainingFOnes = np.c_[trainingFCubed, onesColumn]
onesColumn = np.ones((testF.shape[0]))
testFOnes = np.c_[testFCubed, onesColumn]
print(trainingFOnes)

# w = (X^T * X)^-1 * X^T * y, where X is s the data matrix augmented with a column of ones, and y is the column vector of target outputs.
# letting p = (X^T * X)^-1
# letting q = (X^T * X)^-1 * X^T
does the third order regression of training data
trainingFOnesMatrix = np.asmatrix(trainingFOnes)
trainingOMatrix = np.ravel(np.asmatrix(trainingO).T)
# print(trainingFOnesMatrix)
p = np.linalg.inv(trainingFOnesMatrix.T @ trainingFOnesMatrix)
q = p @ trainingFOnesMatrix.T
# The matrix was turning from column vector to row vector, so Transposed back
w = q @ trainingOMatrix
w = np.ravel(np.asarray(w))
print(w)

# Creates the x and y values of Third Order Polynomial Regression for training
ThirdOrderRegressionX = np.arange(np.amin(trainingF), np.amax(trainingF), 0.01)
ThirdOrderRegressionY = ((w[0] * (ThirdOrderRegressionX)**3) + (w[1] * (ThirdOrderRegressionX)**2) + (w[2] * ThirdOrderRegressionX) + w[3])
# Uses the equation to find mean squared error for training
ThirdOrderRegressionValues = []
for i in range(0, len(trainingF)):
    ThirdOrderRegressionValues.append(w[0] * (trainingF[i])**3 + w[1] * (trainingF[i])**2 + w[2] * trainingF[i] + w[3])

SORVMatrix = np.ravel(np.asarray(ThirdOrderRegressionValues))
# print(SORVMatrix)
mse_sum = 0
for i in range(0, len(trainingF)):
    mse_sum += (trainingO[i] - SORVMatrix[i])**2
mse = (mse_sum / len(trainingF))
print(mse)

does the third order regression of test data
testFOnesMatrix = np.asmatrix(testFOnes)
testOMatrix = np.ravel(np.asmatrix(testO).T)
# print(trainingFOnesMatrix)
p = np.linalg.inv(testFOnesMatrix.T @ testFOnesMatrix)
q = p @ testFOnesMatrix.T
# The matrix was turning from column vector to row vector, so Transposed back
w = q @ testOMatrix
w = np.ravel(np.asarray(w))
```

```
print(w)

# Creates the x and y values of Third Order Polynomial Regression for test
ThirdOrderRegressionX = np.arange(np.amin(testF), np.amax(testF), 0.01)
ThirdOrderRegressionY = ((w[0] * (ThirdOrderRegressionX)**3) + (w[1] * (ThirdOrderRegressionX)**2) + (w[2] * ThirdOrderRegressionX) + w[3])
# Uses the equation to find mean squared error for training
ThirdOrderRegressionValues = []
for i in range(0, len(testF)):
    ThirdOrderRegressionValues.append(w[0] * (testF[i])**3 + w[1] * (testF[i])**2 + w[2] * testF[i] + w[3])

SORVMatrix = np.ravel(np.asarray(ThirdOrderRegressionValues))
# print(SORVMatrix)
mse_sum = 0
for i in range(0, len(testF)):
    mse_sum += (testO[i] - SORVMatrix[i])**2
mse = (mse_sum / len(testF))
print(mse)
```

2f



Equation: $y = -0.78497416x^4 + 4.29846914x^3 - 9.85657215x^2 + 9.73054414x + 1.6375279$

Average Error = 0.43664763409971397

Equation: $y = -0.78497416x^4 + 4.29846914x^3 -9.85657215x^2 + 9.73054414x + 1.6375279$

Average Error = 1.5584694832319532

Based on all the results, the 4th Order is better than 1st order but worse than 3rd and 2nd. And the order of accuracy is 3rd Order, 2nd Order, 4th Order then 1st Order.

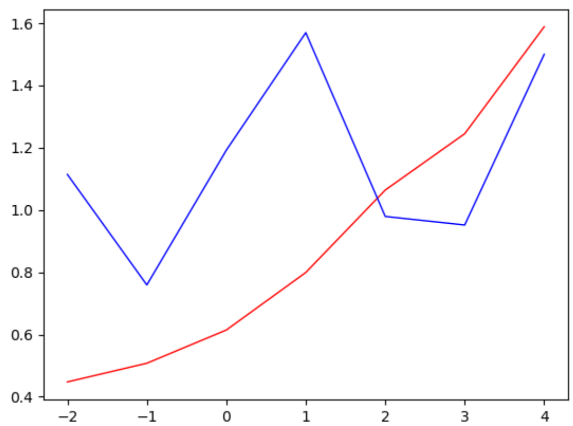
Python Code

```
# Gets the x^2 values and appends them to the features of training/test data
trainingF4TH = np.c_[(trainingF)**4, (trainingF)**3, (trainingF)**2,trainingF]
testF4TH = np.c_[(testF)**4, (testF)**3, (testF)**2, testF]
# Appends a column of ones to the features of training/test data, 2nd order
onesColumn = np.ones((trainingF.shape[0]))
trainingFOnes = np.c_[trainingF4TH, onesColumn]
onesColumn = np.ones((testF.shape[0]))
testFOnes = np.c_[testF4TH, onesColumn]
# print(trainingFOnes)

# # w = (X^T * X)^-1 * X^T * y, where X is s the data matrix augmented with a column of ones, and y is the column vector of target outputs.
# # letting p = (X^T * X)^-1
# # letting q = (X^T * X)^-1 * X^T
# # does the third order regression of training data
trainingFOnesMatrix = np.asmatrix(trainingFOnes)
trainingOMatrix = np.ravel(np.asmatrix(trainingO).T)
# # print(trainingFOnesMatrix)
p = np.linalg.inv(trainingFOnesMatrix.T @ trainingFOnesMatrix)
q = p @ trainingFOnesMatrix.T
# # The matrix was turning from column vector to row vector, so Transposed back
w = q @ trainingOMatrix
w = np.ravel(np.asarray(w))
print(w)

# Creates the x and y values of Fourth Order Polynomial Regression for training
FourthOrderRegressionX = np.arange(np.amin(trainingF), np.amax(trainingF), 0.01)
FourthOrderRegressionY = ((w[0] * (FourthOrderRegressionX)**4) + (w[1] * (FourthOrderRegressionX)**3) + (w[2] * (FourthOrderRegressionX)**2) + w[3] * FourthOrderRegressionX + w[4])
# Uses the equation to find mean squared error for training
FourthOrderRegressionValues = []
for i in range(0, len(trainingF)):
    FourthOrderRegressionValues.append(w[0] * (trainingF[i])**4 + w[1] * (trainingF[i])**3 + w[2] * trainingF[i]**2 + w[3] * trainingF[i] + w[4])

SORVMatrix = np.ravel(np.asarray(FourthOrderRegressionValues))
# print(SORVMatrix)
mse_sum = 0
for i in range(0, len(trainingF)):
    mse_sum += (trainingO[i] - SORVMatrix[i])**2
mse = (mse_sum / len(trainingF))
print(mse)

does the fourth order regression of test data
testFOnesMatrix = np.asmatrix(testFOnes)
testOMatrix = np.ravel(np.asmatrix(testO).T)
# print(trainingFOnesMatrix)
p = np.linalg.inv(testFOnesMatrix.T @ testFOnesMatrix)
q = p @ testFOnesMatrix.T
# The matrix was turning from column vector to row vector, so Transposed back
# w = q @ testOMatrix
w = np.ravel(np.asarray(w))
print(w)
```

```python
# Creates the x and y values of Fourth Order Polynomial Regression for test
FourthOrderRegressionX = np.arange(np.amin(testF), np.amax(testF), 0.01)
FourthOrderRegressionY = ((w[0] * (FourthOrderRegressionX)**4) + (w[1] * (FourthOrderRegressionX)**3) + (w[2] * (FourthOrderRegressionX)**2) + w[3] * FourthOrderRegressionX + w[4])
# Uses the equation to find mean squared error for training
FourthOrderRegressionValues = []
for i in range(0, len(testF)):
    FourthOrderRegressionValues.append(w[0] * (testF[i])**4 + w[1] * (testF[i])**3 + w[2] * testF[i]**2 + w[3] * testF[i] + w[4])

SORVMatrix = np.ravel(np.asarray(FourthOrderRegressionValues))
# print(SORVMatrix)
mse_sum = 0
for i in range(0, len(testF)):
    mse_sum += (testO[i] - SORVMatrix[i])**2
mse = (mse_sum / len(testF))
print(mse)
```

3a

4th Order l2-regularized



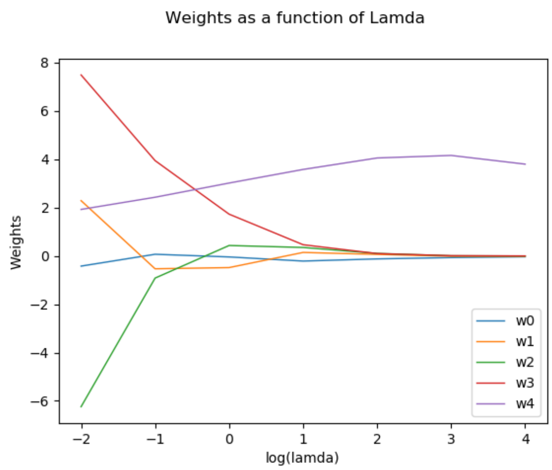
(Blue Line is the Error while Red is the Test Error)

From this, it can be concluded that lambda of 0.1, or lambda = e^{-1} gives the minimum, and is the best fitting for that data.

Python Code

```python
lamda = [0.01, 0.1, 1, 10, 100, 1000, 10000]
I = np.matrix([[1, 0, 0, 0 ,0],
               [0, 1, 0, 0, 0],
               [0, 0, 1, 0 ,0],
               [0, 0, 0, 1 ,0],
               [0, 0, 0, 0 ,0]])
# Gets the x^2 values and appends them to the features of training/test data
trainingF4TH = np.c_[(trainingF)**4, (trainingF)**3, (trainingF)**2,trainingF]
testF4TH = np.c_[(testF)**4, (testF)**3, (testF)**2, testF]
# Appends a column of ones to the features of training/test data, 2nd order
onesColumn = np.ones((trainingF.shape[0]))
trainingFOnes = np.c_[trainingF4TH, onesColumn]
onesColumn = np.ones((testF.shape[0]))
testFOnes = np.c_[testF4TH, onesColumn]
trainingFOnesMatrix = np.asmatrix(trainingFOnes)
trainingOMatrix = np.ravel(np.asmatrix(trainingO).T)

trainError = []
testError = []

for i in range (0, len(lamda)):
    w = (np.linalg.inv((trainingFOnesMatrix.T @ trainingFOnesMatrix) + lamda[i] * I) @ trainingFOnesMatrix.T @ trainingOMatrix)
    print(trainingFOnesMatrix.T @ trainingFOnesMatrix)
    w = np.ravel(np.asarray(w))

    FourthOrderRegressionValues = []
    for j in range(0, len(trainingF)):
        FourthOrderRegressionValues.append(w[0] * (trainingF[j])**4 + w[1] * (trainingF[j])**3 + w[2] * trainingF[j]**2 + w[3] * trainingF[j] + w[4])
    SORVTMatrix = np.ravel(np.asarray(FourthOrderRegressionValues))

    FourthOrderRegressionValues = []
    for j in range(0, len(testF)):
        FourthOrderRegressionValues.append(w[0] * (testF[j])**4 + w[1] * (testF[j])**3 + w[2] * testF[j]**2 + w[3] * testF[j] + w[4])
    SORVMatrix = np.ravel(np.asarray(FourthOrderRegressionValues))

    mse_sumTrain = 0
    mse_sumTest = 0
    for j in range(0, len(trainingO)):
        mse_sumTrain += (trainingO[j] - SORVTMatrix[j])**2
        mseTrain = (mse_sumTrain / len(trainingF))

    for j in range(0, len(testO)):
        mse_sumTest += (testO[j] - SORVMatrix[j])**2
        mseTest = (mse_sumTest / len(testF))

    trainError.append(mseTrain)
    testError.append(mseTest)

for i in range (0, len(lamda)):
    lamda[i] = math.log10(lamda[i])
```

3b

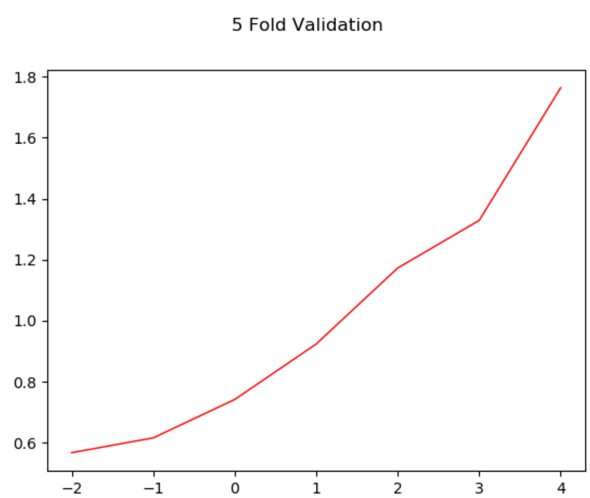


Lambda is spelt wrong here as the code interprets lambda as a preset value, lamda does not cause errors

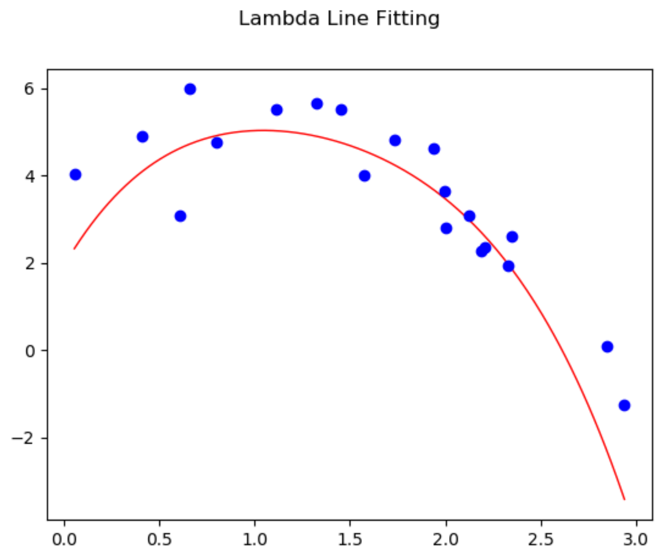
Python Code

```
# 3b code here, for weights ******
weightsArray = []
for i in range (0, len(lamda)):
    w = (np.linalg.inv((trainingFOnesMatrix.T @ trainingFOnesMatrix) + lamda[i] * I) @ trainingFOnesMatrix.T @ trainingOMatrix)
    w = np.ravel(np.asarray(w))
    weightsArray.append(w)

weightsArray = (np.asarray(weightsArray))

for i in range (0, len(lamda)):
    lamda[i] = math.log10(lamda[i])

for i in range (0, weightsArray.shape[1]):
    colourValue = "C" + str(i)
    labelString = "w" + str(i)
    plt.plot(lamda, weightsArray[:,i], color=colourValue, linewidth=1, label=labelString)

plt.legend(loc="lower right")
plt.xlabel('log(lamda)')
plt.ylabel('Weights')
```

3c



As this graph shows, the lambda value of -2 in log(lambda) is better than the previous lambda of 0, which makes lambda 0.01 instead of 0.1.



Here is the test data fit with a l2-regularized 4th-order polynomial regression line, with a lambda of 0.01.

Python Code

```
# 3c, first part to find lamda value
valData = []
trainingData = []
valDataO = []
trainingDataO = []
errorValues = np.empty([7,5])
for i in range(0,5):
    valData.extend(trainingF[i*8:i*8+8])
    valDataO.extend(trainingO[i*8:i*8+8])
    if(i == 0):
        trainingData.extend(trainingF[8:len(trainingF)])
        trainingDataO.extend(trainingO[8:len(trainingF)])
    elif(i == 4):
        trainingData.extend(trainingF[0:i*8])
        trainingDataO.extend(trainingO[0:i*8])
    else:
        trainingData.extend(trainingF[0:i*8])
        trainingDataO.extend(trainingO[0:i*8])
        trainingData.extend(trainingF[i*8+8:len(trainingF)])
        trainingDataO.extend(trainingO[i*8+8:len(trainingF)])

    trainingDataNP = np.asarray(trainingData)
    valDataNP = np.asarray(valData)
    trainingDataONP = np.matrix(trainingDataO).T

    valDataONP = np.asmatrix(valDataO).T

    trainingF4TH = np.c_[(trainingDataNP)**4, (trainingDataNP)**3, (trainingDataNP)**2, trainingDataNP]
    onesColumn = np.ones((trainingF4TH.shape[0]))
    trainingFOnes = np.c_[trainingF4TH, onesColumn]
    trainingFOnesMatrix = np.asmatrix(trainingFOnes)


    for j in range (0, len(lamda)):
        w = (np.linalg.inv((trainingFOnesMatrix.T @ trainingFOnesMatrix) + lamda[j] * I) @ trainingFOnesMatrix.T @ trainingDataONP)
        w = np.ravel(np.asarray(w))

        mse = 0
        mse_sum = 0
        FourthOrderRegressionValues = []
        for k in range(0, len(valData)):
            FourthOrderRegressionValues.append(w[0] * (valData[k])**4 + w[1] * (valData[k])**3 + w[2] * valData[k]**2 + w[3] * valData[k] + w[4])
        valWMatrix = np.ravel(np.asarray(FourthOrderRegressionValues))

        for k in range(0, len(valDataONP)):
            mse_sum += (valWMatrix[k] - valDataONP[k])**2
        mse = (mse_sum / len(valDataONP))

        errorValues[j][i] = np.ravel(mse)
```

```
    # print("training: " + str(trainingFOnesMatrix.shape))
    # print("val: " + str(trainingOMatrix.shape))
    valData = []
    trainingData = []
    valDataO = []
    trainingDataO = []

avgError = []
for i in range(errorValues.shape[0]):
    avgError.append(np.mean(errorValues[i,:]))

for i in range (0, len(lamda)):
    lamda[i] = math.log10(lamda[i])
#3c part 2, plotting regression with found lambda value

w = (np.linalg.inv((trainingFOnesMatrix.T @ trainingFOnesMatrix) + 0.01 * I) @ trainingFOnesMatrix.T @ trainingOMatrix).T
# Creates the x and y values of Fourth Order Polynomial Regression for test
FourthOrderRegressionX = np.arange(np.amin(testF), np.amax(testF), 0.01)
FourthOrderRegressionY = ((w[0] * (FourthOrderRegressionX)**4) + (w[1] * (FourthOrderRegressionX)**3) + (w[2] * (FourthOrderRegressionX)**2) + w[3] * FourthOrderRegressionX + w[4])
```