Linear_Regression

June 19, 2021

[1]:
```
%%capture
import sys
!{sys.executable} -m pip install ipynb
!{sys.executable} -m pip install scikit-learn
```

[2]:
```
%%capture
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn as sklearn

from ipynb.fs.full.CleaningData import getDataset
from ipynb.fs.full.CleaningData import getCovarianceVector
from sklearn.model_selection import train_test_split
from numpy import unravel_index

import warnings
warnings.filterwarnings('ignore')

pd.set_option("display.max_rows", 100)
pd.set_option("display.max_columns", None)

np.set_printoptions(suppress=True)  # When displaying a numpy array, the values␣
↪are NOT expressed in Scientific Notation
# :) Please uncomment to use (if desired):
# To use this you MIGHT need to run Jupyter Notebook with the following command:
↪
# jupyter notebook --NotebookApp.iopub_data_rate_limit=1.0e10
#np.set_printoptions(threshold=np.inf)  # Displays ALL rows and ALL columns -␣
↪takes slightly more compute time
```

0.0.1 Hyper Parameters

The three global variables below are used in the validation dataset to find the best hyper paramters to be used for the testing dataset.

[3]:
```
# Closed-form Solution
optimalLambda = 0
```

```
# Gradient Descent
optimalAlpha = 0
optimalStopping = 0
```

1 Get Dataset

The datatset is imported from CleaningData.ipynb. It prints all the columns that were deleted and the reason why. It also prints a dataframe of the first 100 records from the cleaned dataset.

[ ]:
```
# Store dataset in a dataframe df
df = getDataset(500)

# Remove the URL column because it is not a feature - the features are based␣
↪off the url
urlColumn = df['url']
del df['url']

# Print the first 100 records in the dataframe
# df.head(100)
```

2 Split Data

We used a package called **sklearn** to split the dataset into Training Data, Testing Data, and Validation Data.

The ratio is Training:Testing:Validation = 60:20:20

[5]:
```
# Create variables for ratios (60:20:20)
train_ratio = 0.6
validation_ratio = 0.2
test_ratio = 0.2

# trainX is split to be 60% of the entire data set and testX is 40% of the␣
↪entire dataset
trainX, testX, = train_test_split(df, test_size = 1 - train_ratio)

# test is now 40% of the initial data set
# testX is split further to create testX to be 20% and validationX (valX) to be␣
↪20% of the initial data set
valX, testX = train_test_split(testX, test_size = test_ratio/(test_ratio +␣
↪validation_ratio))

# The 'status' column is our target, so it is deleted from the features and␣
↪stored as the true y values
trainY = trainX['status']
del trainX['status']
```

```
testY = testX['status']
del testX['status']
valY = valX['status']
del valX['status']

# Print out the training data (x)
# :) Please uncomment to see the print statement (if desired):
#print(trainX)  # At the bottom, it shows that trainX contains 6888 rows and 56␣
↪columns
```

3 FUNCTIONS

Before performing Linear Regression, we've created functions that are used in both solutions (closed-form and gradient descent).

4 Create Design Matrix

[6]:
```
# Function to create design matrix using the data given (xValues)
def createDesignMatrix(xValues):

    # Initialise matrix filled with 1s
    number_rows = len(xValues.index)
    number_cols = len(xValues.columns) + 1
    designMat = np.ones((number_rows, number_cols), dtype=float)

    # For each feature (column) and url (row), add the value to the design␣
↪matrix
    for featureIndex in range(len(xValues.columns)):
        for urlIndex in range(len(xValues.index)):

            colName = xValues.columns[featureIndex]
            designMat[urlIndex][featureIndex + 1] = xValues[colName][xValues.
↪index[urlIndex]]

    return designMat

designMatrix = createDesignMatrix(trainX)
# :) Please uncomment to see the print statement (if desired):
#print(designMatrix)
```

5 Calculate Predicted y Values

The predicted y values are the values we predicted using the `thetaVector`.

The predicted y vales are calculated using the function: `y` = $theta_0 + theta_1 * x_1 + theta_2 * x_2 + \ldots + theta_n * x_n$

Predicted values are rounded because the `status` column (target) contains values of `0` (Legitimate) or `1` (Phishing).

[7]:
```
# Function to calulate predicted y values
def calculateY(designMat, thetas):

    # Store predicted values in a vector
    predictedY = []

    # For every url
    for urlIndex in range(len(designMat)):

        # The function is really long because there are 57 features so we add␣
 ↪each term in the for loop
        # we begin by setting y to the theta0 (which is multiplied by 1 - the␣
 ↪first element in the design matrix
        # for each row)
        y = thetas[0]

        # For every feature (exluding the first element which is used above)
        for j in range(1, len(designMat[urlIndex])):

            y = y + designMat[urlIndex][j] * thetas[j]

        # Rount the predicted y value
        predictedY.append(round(y))

    return predictedY
```

6 Create Confusion Matrix

The functions `getConfusionMatrix()` calculates the confusion martix and `printConfusionMatrix()` prints it out.

[8]:
```
#Get confusion matrix data for making the matrix
def getConfusionMatrix(predictedY, trueY):

    # For all URLs
    amountDataPoints = len(predictedY)

    quad1 = 0
    quad2 = 0
    quad3 = 0
    quad4 = 0
    outliers = 0  # outliers is used for the case when the predicted y value is␣
 ↪neither a 0 or 1 (outliers)

```

```
    for i in range(amountDataPoints):

        if(trueY[trueY.index[i]] == 0):

            if (predictedY[i] == 0):
                quad1 += 1
            elif (predictedY[i] == 1):
                quad2 += 1
            else:
                outliers += 1

        else:

            if (predictedY[i] == 1):
                quad4 += 1
            elif (predictedY[i] == 0):
                quad3 += 1
            else:
                outliers += 1


    return [quad1, quad2, quad3, quad4, outliers]

#Printing out confusion matrix
def printConfusionMatrix(quad1, quad2, quad3, quad4, hyperParemeterSymbol, hyperParameterValue):

    print(hyperParemeterSymbol + " is: " + str(hyperParameterValue))
    print("===========================\n")

    print("===========================")
    print("Confusion Matrix\n")
    print("Class\t\tLegitimate\tPhishing")
    print("Legitimate\t"+str(quad1)+"\t\t"+str(quad2))
    print("Phishing\t"+str(quad3)+"\t\t"+str(quad4))
    print("\n=========================\n")

    accu = (quad1 + quad4)/(quad1 + quad2 + quad3 + quad4)
    print("===========================")
    print("Accuracy of: " + str(format(accu * 100,".5f")) + " %")
    print("===========================")

    print("\n")
```

7 TRAINING DATA

8 Closed-Form Solution

8.0.1 Create theta vector

Create vector which represents the **optimal solution for theta**

The formula to determine the optimal solution for theta is: `theta =` $((X)^Y X)^{-1}(X^T y)$

[9]:
```
# Function to create the optimal theta vector
def createOptimalTheta(lamda):

    # The identity matrix is used for regularisation
    identityMatrix = np.identity(designMatrix.shape[1])
    identityMatrix[0][0] = 0  # There is no regularisation on theta0

    # Performing calculation:
    transMatrix = designMatrix.transpose() #X^T

    calcXtX = transMatrix.dot(designMatrix)  #(X^T X)

    calcReg = calcXtX + (lamda * identityMatrix)  #(X^T X + lamda*identity) -␣
↪Regularisation
    calcInverse = np.linalg.inv(calcReg) #(X^T X + lamda*identity)^(-1)

    calcXtY = (transMatrix.dot(trainY)) #X^T y
    thetaVector = calcInverse.dot(calcXtY) # theta = (X^T X +␣
↪lamda*identity)^(-1) X^T y

    return thetaVector
```

8.0.2 Perform closed-form solution

[10]:
```
%%time
# Call function

# Lambda is a hyperparameter that is used for regularisation
# Lambda was tested at a few different values but had a very minimal impact on␣
↪the accuracy of the model.
lamda = 0.1
thetaVector = createOptimalTheta(lamda)

# :) Please uncomment to see the print statement (if desired):
#print(thetaVector)
```

```
Wall time: 4.99 ms
```

8.0.3 Calculate predicted y values

[11]:
```
# Call function
predictedY = calculateY(designMatrix, thetaVector)

# :) Please uncomment to see the print statement (if desired):
#print(predictedY)
```

8.0.4 Print Confusion Matrix

[12]:
```
print("===========================")
print("Hyper Parameters:\n")

# Call functions
quad1, quad2, quad3, quad4, outliers = getConfusionMatrix(predictedY, trainY)
printConfusionMatrix(quad1, quad2, quad3, quad4, "lambda", lamda)

print("===========================")
print("Number of outiers = " + str(outliers))
```

```
===========================
Hyper Parameters:

lambda is: 0.1
===========================

===========================
Confusion Matrix

Class           Legitimate      Phishing
Legitimate      3137            264
Phishing        260             3217

=========================

===========================
Accuracy of: 92.38151 %
===========================


===========================
Number of outiers = 10
```

9 Gradient Descent

9.0.1 Function to calculate gradient descent

[13]:
```
#Gradient Descent Function
def gradientDescent(designMat, a_val, stoppingVal, y_set):

    # Initialise variables
    amountOfThetas = designMat.shape[1]
    newTheta = np.repeat(0.5, amountOfThetas)
    oldTheta = np.repeat(99999, amountOfThetas)

    # If the x values and the y values have a different number of rows
    if (len(designMat) != len(y_set)):
        print("The x values and the y values have different lengths")
        return []

    # Repeat until convergence
    while (np.linalg.norm(newTheta - oldTheta,2) > stoppingVal):

        # Calculate predicted y values with new theta
        for i in range(len(designMat)):

            oldTheta = newTheta

            y = newTheta[0]

            for j in range(1, len(designMat[i])):

                y = y + designMat[i][j] * newTheta[j]



                # FORMULA: theta = theta - a(predictedY - trueY)x
                var1 = a * (y - y_set[y_set.index[j]])  # a(predictedY - trueY)
                var = np.multiply(designMat[i], np.float64(var1))  #␣
↪a(predictedY - trueY)x
                newTheta = newTheta - var  # theta = theta - a(predictedY -␣
↪trueY)x

            # Converged
            if (np.linalg.norm(newTheta - oldTheta,2) > stoppingVal):
                break


    return newTheta
```

9.0.2 Perform gradient descent

[14]:
```
%%time
# Hyperparameter value (alpha) and (stoppingVal - used for the convergence)
a = 0.00001
stoppingVal = 0.001

# Call function
gradientTheta = gradientDescent(designMatrix, a, stoppingVal, trainY)

# :) Please uncomment to see the print statement (if desired):
#print(gradientTheta)
```

```
Wall time: 5min 38s
```

9.0.3 Calculate predicted y values

[15]:
```
# Call function
newY = calculateY(designMatrix, gradientTheta)

# :) Please uncomment to see the print statement (if desired):
#print(newY)
```

9.0.4 Print confusion matrix

[16]:
```
print("===========================")
print("Hyper Parameters:\n")
print("stoppingval is: " + str(stoppingVal))

quad1, quad2, quad3, quad4, outliers = getConfusionMatrix(newY, trainY)
printConfusionMatrix(quad1, quad2, quad3, quad4, "alpha", a)

print("===========================")
print("Number of outiers = " + str(outliers))
```

```
===========================
Hyper Parameters:

stoppingval is: 0.001
alpha is: 1e-05
===========================

===========================
Confusion Matrix

Class           Legitimate      Phishing
Legitimate      2736            668
Phishing        2609            875
```

```
=========================

===========================
Accuracy of: 52.42451 %
===========================



===========================
Number of outiers = 0
```

10 VALIDATION DATA

10.0.1 Closed-Form Solution

[17]:
```
%%time
# Get design matrix
designMatVal = createDesignMatrix(valX)

# Store different hyper parameters for lambda to find the highest accuracy
lamda = [0.001, 0.05, 0.1, 2, 5]
accuracyVec = np.array([0, 0, 0, 0, 0])

for lambdaIndex in range(len(lamda)):

    # Create theta vector
    thetaValidation = createOptimalTheta(lamda[lambdaIndex])

    # Calculate predicted y values
    predictedValY = calculateY(designMatVal, thetaValidation)

    # Get confusion matrix data to calculate accuracy
    quad1, quad2, quad3, quad4, outliers = getConfusionMatrix(predictedValY,␣
 ↪valY)
    # Calculate accuracy and store it
    accuracy = (quad1 + quad4)/(quad1 + quad2 + quad3 + quad4)
    accuracyVec[lambdaIndex] = accuracy

# Find index of highest accuracy and determine best hyper parameter
maxAccuracyIndex = np.argmax(accuracyVec, axis=0)

# Set optimalLambda to best optimal lambda
optimalLambda = lamda[maxAccuracyIndex]

# :) Please uncomment to see the print statement (if desired):
#print(optimalLambda)
```

```
Wall time: 3.45 s
```

10.0.2 Gradient Decsent

[18]:
```
%%time
# Get design matrix
designMatVal = createDesignMatrix(valX)

# Store different hyper parameters for alpha and the stopping value to find the␣
↪highest accuracy
# Because there are two hyper parameters, we use a double for loop is so that␣
↪every alpha
# parameter is tested with every stopping value parameter.
# There are 4 values in each list, so it runs 16 times to find the accuracy -␣
↪this takes a loooonnggg time.
alpha = [0.00001, 0.01] #, 1, 5
stoppingVec = [0.0001, 0.1] #, 1, 5
accuracyVec = np.zeros((len(alpha), len(stoppingVec)))

# for each alpha parameter
for alphaIndex in range(len(alpha)):

    # for each stopping value parameter
    for stoppingIndex in range(len(stoppingVec)):

        # Create theta vector
        thetaValidation = gradientDescent(designMatVal, alpha[alphaIndex],␣
↪stoppingVec[stoppingIndex], valY)

        # Calculate predicted y values
        predictedValY = calculateY(designMatVal, thetaValidation)

        # Get confusion matrix data to calculate accuracy
        quad1, quad2, quad3, quad4, outliers =␣
↪getConfusionMatrix(predictedValY, valY)
        # Calculate accuracy and store it
        accuracy = (quad1 + quad4)/(quad1 + quad2 + quad3 + quad4)
        accuracyVec[alphaIndex][stoppingIndex] = accuracy


# Find value and index of highest accuracy and determine best hyper parameters
maxAccuracy = np.max(accuracyVec)
indexes = np.where(accuracyVec == maxAccuracy)

# Set optimalAlpha and optimalStopping to best optimal values
optimalAlpha = alpha[indexes[0][0]]
optimalStopping = stoppingVec[indexes[1][0]]

# :) Please uncomment to see the print statement (if desired):
```

```
#print(optimalAlpha)
#print(optimalStopping)
```

```
Wall time: 4h 26min 6s
```

11 TESTING DATA

11.0.1 Closed-Form Solution

[19]:
```
%%time
# Get design matrix
designMatTest = createDesignMatrix(testX)

# Calculate predicted y values
predictedTestY = calculateY(designMatTest, thetaVector)

print("===========================")
print("Hyper Parameters:\n")

# Get confusion matrix and print it
quad1, quad2, quad3, quad4, outliers = getConfusionMatrix(predictedTestY, testY)
printConfusionMatrix(quad1, quad2, quad3, quad4, "lambda", optimalLambda)

print("===========================")
print("Number of outiers = " + str(outliers))
```

```
===========================
Hyper Parameters:

lambda is: 0.001
===========================

===========================
Confusion Matrix

Class           Legitimate      Phishing
Legitimate      1072            79
Phishing        73              1068

=========================

===========================
Accuracy of: 93.36824 %
===========================



===========================
Number of outiers = 5
```

```
Wall time: 1.05 s
```

11.0.2 Gradient Descent

[20]:
```
%%time
# Get design matrix
designMatTest = createDesignMatrix(testX)

# Calculate predicted y values
predictedTestY = calculateY(designMatTest, gradientTheta)

# Get and print confusion matrix
print("===========================")
print("Hyper Parameters:\n")
print("stoppingval is: " + str(stoppingVal))

quad1, quad2, quad3, quad4, outliers = getConfusionMatrix(predictedTestY, testY)
printConfusionMatrix(quad1, quad2, quad3, quad4, "alpha", a)

print("===========================")
print("Number of outiers = " + str(outliers))
```

```
===========================
Hyper Parameters:

stoppingval is: 0.001
alpha is: 1e-05
===========================

===========================
Confusion Matrix

Class           Legitimate      Phishing
Legitimate      935             217
Phishing        870             275

=========================

===========================
Accuracy of: 52.67741 %
===========================



===========================
Number of outiers = 0
Wall time: 2.68 s
```

[ ]:
```

```