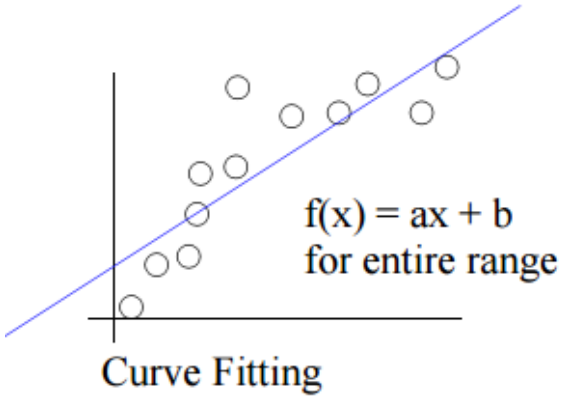
7.7 Curve Fitting and Error Function

Curve Fitting and Error Function

It is a classical regression problem, where we are given *N* number of training data points and our task is to find a function which can draw a curve or line across these data point in the best possible manner. Informally, the best possible line is the line which is closest to all the data points and is also in proximity to all the points which are going to appear in the future.



In the above graph we have 12 data points and we have drawn a line, of prediction, through these point using the equation $f(x)=ax+b$, where a and b are parameters representing slope and y intercept of the line respectively, whereas x is a data point.

Another way to look at above graph is to treat the circles as target value, x as predictor variable and the blue line as predicted value. So, our task is to draw a curve that predict these target values accurately and when extrapolated it predicts unseen target values as well.

Let's call the function, which draws the blue line, as hypothesis function

$$h_w(x) = w_0x_0 + w_1x_1.....w_nx_n = \widehat{y}$$

This hypothesis function produces a value $\widehat{y}$ which is our prediction corresponding to the actual value y shown by circles in above graph.

Also, we have total m data points and each data point is n dimensional, i.e., n attributes. w_0 to w_n are weights.

Above setup can be represented in matrix form as shown below

$$W = \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ w_n \end{bmatrix} X = \begin{bmatrix} x_0^1 & x_1^1 & . & . & x_n^1 \\ x_0^2 & x_1^2 & . & . & x_n^2 \\ . & . & . & . & . \\ . & . & . & . & . \\ x_0^m & x_1^m & . & . & x_n^m \end{bmatrix} Y = \begin{bmatrix} y^1 \\ y^2 \\ . \\ . \\ y^m \end{bmatrix} \widehat{Y} = \begin{bmatrix} \widehat{y^1} \\ \widehat{y^2} \\ . \\ . \\ \widehat{y^m} \end{bmatrix}$$

$W\ is\ weight\ vector\ of\ dimension\ n+1 \times 1$
$X\ is\ data\ matrix\ of\ dimension\ m \times n+1$
$Y\ is\ target\ vector\ of\ dimension\ m \times 1$
$\widehat{Y}\ is\ predicted\ vector\ of\ dimension\ m \times 1$

$$h_w(x) = W^T X^T = XW = \widehat{Y}$$

In the above data, matrix x_0^i are initialized by all 1, whereas initial values of weights in the weight vector are selected randomly.

Error function: It measures the difference between the predicted value and actual value. Most commonly used error function is squared error.

$$Error\ Function$$

$$E(w) = \frac{1}{2}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)^2$$

$$cost\ function$$

$$J(w_0, w_1 .... w_n) = \frac{1}{2m}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)^2$$

Above figure is graphical representation of error, it is the perpendicular distance between regression line and target variable(dots).

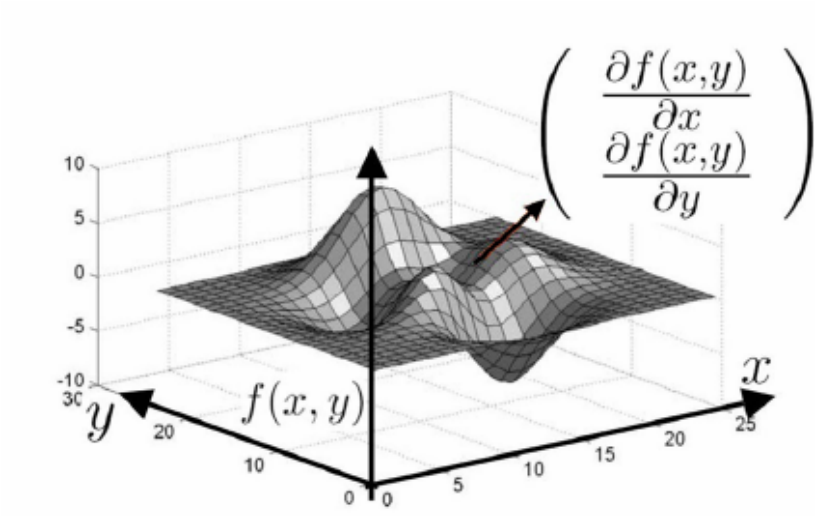
Our goal is to select a combination of weights which minimizes the error function or the cost function. But, how to do this in *n* dimensional space, brute force approach of trying and testing every possible value of weights will not work. We will take the help of calculus to find the optimal combination of weights and the process is called gradient descent.

What is gradient?

It is the partial derivative of a function and is represented as follows

$$\nabla J(W) = \begin{Bmatrix} \frac{\delta J(W)}{\delta w_0} \\ \frac{\delta J(W)}{\delta w_1} \\ . \\ . \\ \frac{\delta J(W)}{\delta w_n} \end{Bmatrix}$$

$\nabla J(W)$ points towards the direction of maximum increase as shown in the below picture.



In gradient descent, we find gradient of the cost function, which gives us direction of maximum increase. To find minimum value of the cost function, we subtract this gradient from the original weights. But how can we subtract direction from a quantity like weight? In actual practice we multiply this direction with a very small number and then subtract it from the weights, this ensures that we always travel down the slope (descent) and find a local minimum. This very small number is called learning rate. The entire process of gradient descent is showing below.

$$Repeat\ until\ converge$$
$$\left\{ w_j^{new} := w_j^{old} - \alpha \frac{\delta}{\delta w_i} J(w_0, w_1, ...w_n) \right\}$$

$$gradient\ of\ cost\ function$$
$$\frac{\delta}{\delta w_i} J(w_0, w_1, ...w_n) = \frac{1}{m} \sum_{i=1}^{i=m} (h_w(x^i) - y^i) x_j^i$$

$$Repeat\ until\ converge$$
$$\left\{ w_j^{new} := w_j^{old} - \alpha \frac{1}{m} \sum_{i=1}^{i=m} (h_w(x^i) - y^i) x_j^i \right\}$$

In the above equation of gradient descent, α is the learning rate. We need to simultaneously update all the values of $w_i\ for\ i = 0\ to\ n$ and we need to keep on repeating the step until there is no significant

change in the weights, that is how we know that algorithm has converged. We can expand above equation to make it more clear how it works.

$$\left\{\begin{array}{l} w_0^{new} := w_0^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)x_0^i \\ \\ w_1^{new} := w_1^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)x_1^i \\ \\ w_2^{new} := w_2^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)x_2^i \\ \\ \vdots \\ \\ w_n^{new} := w_0^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)x_n^i \end{array}\right\}$$

Let's see how it works, recall that $h_w(x) = w_0x_0 + w_1x_1.....w_nx_n = \widehat{y}$. In the equation $\sum_{i=1}^{i=m}(h_w(x^i) - y^i)x_n^i$, *i* is an index to the data points, *i* = 1 represents first row of the data matrix **X**.
Similarly, *i* = 2 represents second row of the data matrix **X**. So $h_w(x^1), h_w(x^2), h_w(x^n)$ can be expanded in the following way

$$\begin{array}{l} h_w(x^1) = w_0x_0^1 + w_1x_1^1 + ...w_nx_n^1 = \widehat{y^1} \\ h_w(x^2) = w_0x_0^2 + w_1x_1^2 + ...w_nx_n^2 = \widehat{y^2} \\ .. \\ .. \\ h_w(x^n) = w_0x_0^n + w_1x_1^n + ...w_nx_n^n = \widehat{y^n} \end{array}$$

Putting the predicted values by the hypothesis function gives us the following equation

$$\left\{\begin{array}{l} w_0^{new} := w_0^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(\widehat{y^i} - y^i)x_0^i \\ \\ w_1^{new} := w_1^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(\widehat{y^i} - y^i)x_1^i \\ \\ w_2^{new} := w_2^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(\widehat{y^i} - y^i)x_2^i \\ \\ \vdots \\ \\ w_n^{new} := w_0^{old} - \alpha\frac{1}{m}\sum_{i=1}^{i=m}(\widehat{y^i} - y^i)x_n^i \end{array}\right\}$$

Problem Statement: We have data about various auto mobile and our task is to predict city-cycle fuel consumption in miles per gallon from 2 multivalued discrete and 5 continuous attributes. Details of attribute is given below

1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous
5. weight: continuous

6. acceleration: continuous
7. model year: multi-valued discrete

mpg is the target value, i.e., value which we have to predict using attribute number 2 to 7.

our task is to calculate weights which we can put in the below equation to predict the value of mpg

$$h_w(x) = w_0x_0 + w_1x_1.....w_nx_n = \widehat{y}$$

We will use gradient descent to find optimal weight which minimizes the cost function given below

$$cost\ function$$
$$J(w_0, w_1....w_n) = \frac{1}{2m}\sum_{i=1}^{i=m}(h_w(x^i) - y^i)^2$$

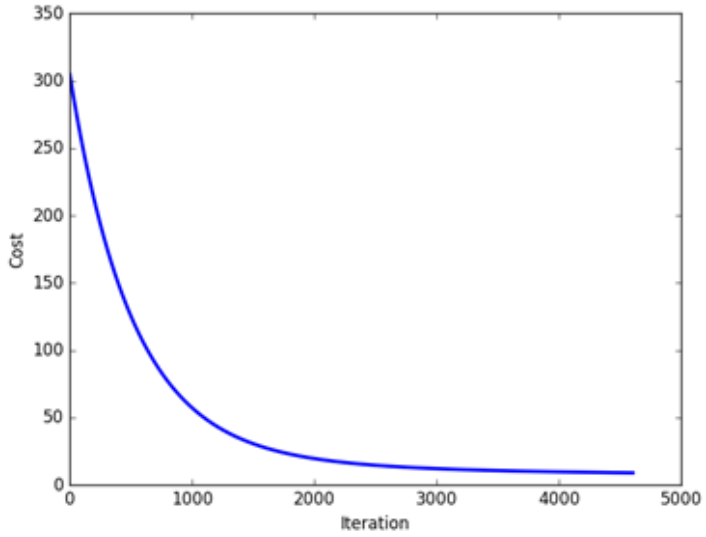
Steps:

1. Normalize the data: different attributes are having different scale, this step will bring all the attributes to same scale and hence makes the convergence faster. We have used mean normalization a variant of Z-score normalization.
$$x_{new} = \frac{x - \mu}{max - min}$$
2. Filter the data: We need to separate target and rest of the attributes and need to put x_0^i attribute in the data frame.
3. Initialize weight vector: We need to have some initial value of weights, these initial values can be zeros or some random values.
4. Calculate weights using gradient descent: need to keep track of convergence by comparing previous and new value of cost.

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

relativePath = os.getcwd() + "//Resources//"
dataFilePath = relativePath + "carMPG.csv"
data = pandas.read_csv(dataFilePath)

learningRate = 0.001  # learning rate
convergeThreshold = 0.001  # threshold to check condition of convergence

def meanNormalization(data, targetColName):  # we need to normalize the data so that each attribute is brought to same scale
    for i in data.columns:
        if i == targetColName:  # do not modify target values
            continue
        mean = data[i].mean()
        min = data[i].min()
        max = data[i].max()
        data[i] = data[i].apply(lambda d: float(d - mean) / float(max - min))

def filterData(dataFrame):  # we need to remove target value from the data frame and need to put x0 values.
    targetValues = dataFrame[[0]]
    colum = range(1, len(dataFrame.columns))
```

```
    data = dataFrame[colum]
    data.insert(0, "x0", [1] * len(dataFrame))
    return targetValues, data  # return target value and modified dataframe

def initializeWeightVector(allZeros, numberOfFeatures):  # weights has to be initialized. either to all zeros or to random values
    if allZeros:  # if all zero flag is true then make it all zero
        return np.zeros((numberOfFeatures, 1))
    else:
        return np.array(np.random.uniform(-2, 2, size=numberOfFeatures)).reshape(numberOfFeatures, 1)

def calCost(weight, data, target):  # calculate cost after each iteration to look for convergance
    m = len(data)
    predictedValue = np.dot(data, weight)
    return (((predictedValue - target) ** 2).sum()) / (2 * m)

def calGradient(data, weights, target, column, learningRate):  # calculate gradient
    m = len(data)
    predictedValue = np.dot(data, weights)
    return learningRate * ((((predictedValue - target) * (data[[column]].as_matrix())).sum()) / float(m))

def gradientDescent():
    prevCost = 0
    iterationCount = 0
    costList = []
    meanNormalization(data, "mpg")  # second parameter is the name of the target column
    target, dataMatrix = filterData(data)
    weights = initializeWeightVector(True, len(dataMatrix.columns))
    cost = calCost(weights, dataMatrix, target)
    while abs(prevCost - cost[0]) > convergeThreshold:  # check for convergence
        iterationCount += 1
        prevCost = cost[0]
        weightSub = []
        for i in range(len(data.columns)):
            weightSub.append(calGradient(dataMatrix, weights, target, i, learningRate))
        weights = weights - weightSub
        print "new Weights", weights.T
        cost = calCost(weights, dataMatrix, target)
        print "new cost", float(cost)
        costList.append(float(cost))
    print "Algo converged in ", iterationCount, "iteration"
    print "final weights ", weights.T
    return costList

def plotGraph(x, y):
    # Create a figure of size 8x6 inches, 80 dots per inch
    plt.figure(figsize=(8, 6), dpi=80)

    # Create a new subplot from a grid of 1x1
    plt.subplot(1, 1, 1)  # rows , column , serial number starting from 1
    # Plot cosine with a blue continuous line of width 1 (pixels)
    plt.plot(x, y, color="blue", linewidth=2.5, linestyle="-", label="cost")
    plt.show()

costList = gradientDescent()
plotGraph(range(1, len(costList) + 1), costList)
```

Below is the plot showing how the cost changes with iteration. Initially rate of change of cost is high but with each iteration change in the cost keeps on decreasing and at the minima it is almost zero. At this point we say that algorithm has converged and we break out of the loop.



Note: Please experiment with different values of learning rate and converge threshold. Also analyze the plot generated at the end of every run.