Linear Regression (Python Implementation)

*we refer to dependent variables as **responses** and independent variables as **features** for simplicity. In order to provide a basic understanding of linear regression, we start with the most basic version of linear regression, i.e. **Simple linear regression***

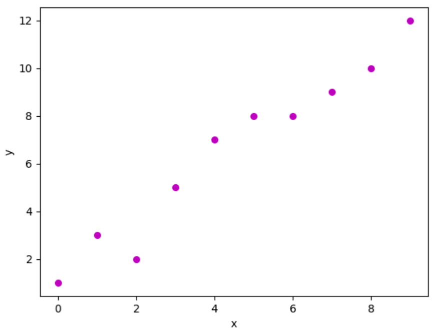
Simple Linear Regression

Simple linear regression is an approach for predicting a **response** using a **single feature**. It is one of the most basic machine learning models that a machine learning enthusiast gets to know about. In linear regression, we assume that the two variables i.e. dependent and independent variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature or independent variable(x). Let us consider a dataset where we have a value of response y for every feature x:

x	0	1	2	3	4	5	6	7	8	9
y	1	3	2	5	7	8	8	9	10	12

x as feature vector, i.e x = [x_1, x_2, …., x_n],
y as response vector, i.e y = [y_1, y_2, …., y_n]

for **n** observations (in the above example, n=10). A scatter plot of the above dataset looks like this:-



Now, the task is to find a **line that fits best** in the above scatter plot so that we can predict the response for any new feature values. (i.e a value of x not present in a dataset) This line is called a **regression line**. The equation of the regression line is represented as:

$$h(x_i) = \beta_0 + \beta_1 x_i$$

Here,

- h(x_i) represents the **predicted response value** for ith observation.
- b_0 and b_1 are regression coefficients and represent the **y-intercept** and **slope** of the regression line respectively.

To create our model, we must "learn" or estimate the values of regression coefficients b_0 and b_1. And once we've estimated these coefficients, we can use the model to predict responses!
In this article, we are going to use the principle of **Least Squares**.
Now consider:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i = h(x_i) + \varepsilon_i \Rightarrow \varepsilon_i = y_i - h(x_i)$$

Here, e_i is a **residual error** in ith observation. So, our aim is to minimize the total residual error. We define the squared error or cost function, J as:

$$J(\beta_0, \beta_1) = \frac{1}{2n} \sum_{i=1}^{n} \varepsilon_i^2$$

And our task is to find the value of b_0 and b_1 for which $J(b_0, b_1)$ is minimum! Without going into the mathematical details, we present the result here:

$$\beta_1 = \frac{SS_{xy}}{SS_{xx}}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

Where SS_{xy} is the sum of cross-deviations of y and x:

$$SS_{xy} = \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) = \sum_{i=1}^{n} y_i x_i - n\bar{x}\bar{y}$$

And SS_{xx} is the sum of squared deviations of x:

$$SS_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n(\bar{x})^2$$

Python Implementation of Linear Regression

We can use the Python language to learn the coefficient of linear regression models. For plotting the input data and best-fitted line we will use the matplotlib library. It is one of the most used Python libraries for plotting graphs.

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

def estimate_coef(x, y):
	# number of observations/points
	n = np.size(x)
```

```
    # mean of x and y vector
    m_x = np.mean(x)
    m_y = np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x

    return (b_0, b_1)

def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color = "m",
                marker = "o", s = 30)

    # predicted response vector
    y_pred = b[0] + b[1]*x

    # plotting the regression line
    plt.plot(x, y_pred, color = "g")

    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')

    # function to show plot
    plt.show()

def main():
    # observations / data
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    # estimating coefficients
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {} \
        \nb_1 = {}".format(b[0], b[1]))

    # plotting regression line
    plot_regression_line(x, y, b)

if __name__ == "__main__":
    main()
```

Multiple linear regression

Multiple linear regression attempts to model the relationship between **two or more features** and a response by fitting a linear equation to the observed data. Clearly, it is nothing but an extension of simple linear regression. Consider a dataset with **p** features(or independent variables) and one response(or dependent variable). Also, the dataset contains **n** rows/observations.

We define:

X (**feature matrix**) = a matrix of size **n X p** where x_{ij} denotes the values of the j^{th} feature for ith observation.

So,

$$\begin{pmatrix} x_{11} & \cdots & x_{1p} \\ x_{21} & \cdots & x_{2p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \vdots & x_{np} \end{pmatrix}$$

and

y (**response vector**) = a vector of size **n** where y_{i} denotes the value of response for ith observation.

$$y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

The **regression line** for **p** features is represented as:

$$h(x_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}$$

where h(x_i) is **predicted response value** for ith observation and b_0, b_1, …, b_p are the **regression coefficients**. Also, we can write:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \varepsilon_i$$
$$or$$
$$y_i = h(x_i) + \varepsilon_i \Rightarrow \varepsilon_i = y_i - h(x_i)$$

where e_i represents a **residual error** in ith observation. We can generalize our linear model a little bit more by representing feature matrix **X** as:

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}$$

So now, the linear model can be expressed in terms of matrices as:

$$y = X\beta + \varepsilon$$

where,

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ \beta_p \end{bmatrix}$$

and

$$\varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ . \\ . \\ \varepsilon_n \end{bmatrix}$$

Now, we determine an **estimate of b**, i.e. b’ using the **Least Squares method**. As already explained, the Least Squares method tends to determine b’ for which total residual error is minimized.
We present the result directly here:

$$\hat{\beta} = (X'X)^{-1}X'y$$

where ‘ represents the transpose of the matrix while -1 represents the matrix inverse.
Knowing the least square estimates, b’, the multiple linear regression model can now be estimated as:

$$\hat{y} = X\hat{\beta}$$

where y’ is the estimated response vector.

Python implementation of multiple linear regression techniques

```
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model, metrics

# load the boston dataset
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+",
                     skiprows=22, header=None)
X = np.hstack([raw_df.values[::2, :],
               raw_df.values[1::2, :2]])
y = raw_df.values[1::2, 2]

# splitting X and y into training and testing sets
X_train, X_test,\
    y_train, y_test = train_test_split(X, y,
                                       test_size=0.4,
                                       random_state=1)

# create linear regression object
reg = linear_model.LinearRegression()

# train the model using the training sets
```

```
reg.fit(X_train, y_train)

# regression coefficients
print('Coefficients: ', reg.coef_)

# variance score: 1 means perfect prediction
print('Variance score: {}'.format(reg.score(X_test, y_test)))

# plot for residual error

# setting plot style
plt.style.use('fivethirtyeight')

# plotting residual errors in training data
plt.scatter(reg.predict(X_train),
                        reg.predict(X_train) - y_train,
                        color="green", s=10,
                        label='Train data')

# plotting residual errors in test data
plt.scatter(reg.predict(X_test),
                        reg.predict(X_test) - y_test,
                        color="blue", s=10,
                        label='Test data')

# plotting line for zero residual error
plt.hlines(y=0, xmin=0, xmax=50, linewidth=2)

# plotting legend
plt.legend(loc='upper right')

# plot title
plt.title("Residual errors")

# method call for showing the plot
plt.show()
```

In the above example, we determine the accuracy score using **Explained Variance Score**. We define:
explained_variance_score = 1 – Var{y – y'}/Var{y}
where y' is the estimated target output, y is the corresponding (correct) target output, and Var is Variance, the square of the standard deviation. The best possible score is 1.0, lower values are worse.

Applications of Linear Regression:

- **Trend lines:** A trend line represents the variation in quantitative data with the passage of time (like GDP, oil prices, etc.). These trends usually follow a linear relationship. Hence, linear regression can be applied to predict future values. However, this method suffers from a lack of scientific validity in cases where other potential changes can affect the data.

- **Economics:** Linear regression is the predominant empirical tool in economics. For example, it is used to predict consumer spending, fixed investment spending, inventory investment, purchases of a country's exports, spending on imports, the demand to hold liquid assets, labor demand, and labor supply.
- **Finance:** The capital price asset model uses linear regression to analyze and quantify the systematic risks of an investment.
- **Biology:** Linear regression is used to model causal relationships between parameters in biological systems.

Practical Example using the breast cancer data set

Assume you want to implement a logistic regression model to classify breast cancer data and evaluate its performance using accuracy score

```
# import the necessary libraries
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load the breast cancer dataset
X, y = load_breast_cancer(return_X_y=True)
# split the train and test dataset
X_train, X_test,\
        y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=23)
# LogisticRegression
clf = LogisticRegression(random_state=0)
clf.fit(X_train, y_train)
# Prediction
y_pred = clf.predict(X_test)

acc = accuracy_score(y_test, y_pred)
print("Logistic Regression model accuracy (in %):", acc*100)
```

Accuracy is a commonly used metric for evaluating the performance of classification models. It represents the proportion of correct predictions made by the model. In this case, an accuracy score of 95.32% implies that the model is highly reliable in classifying breast cancer cases.