Linear Regression

Regression analysis is sub category of Supervised Learning. Regression models are used predict target variable on a continuous scale. This feature makes it attractive for addressing many questions in data science application in industry, such as understanding relationships between variables, evaluating trends, or making forecasts.

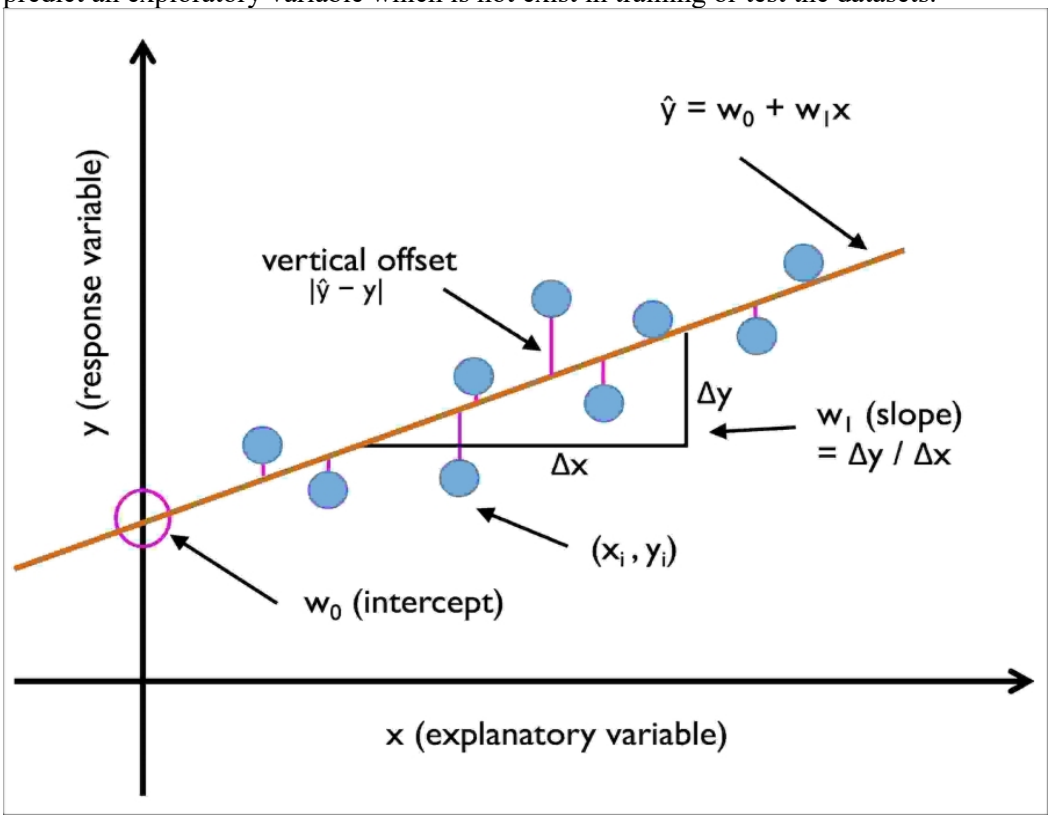
Here we are going to discuss main concept of regression analysis and implement model using python3 with real data example. To start model following topics need to cover:

1) Exploring and visualizing datasets
2) Looking at different approaches to implement linear regression models.
3) Training and test regression models that are robust to outlier.
4) Evaluating regression models that diagnosing common problem

Let's introduce simple linear regression which is a model to find relationship between a single feature called Exploratory variable X and a continuous variable response also called target variable y. then the equation of linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 x$$

Here the weight W_0 is y-axis intercept and W_1 is weight coefficient of the explanatory variable. The goal is to learn weight of linear equation to understand the relationships between feature and target variables and predict an exploratory variable which is not exist in training or test the datasets.

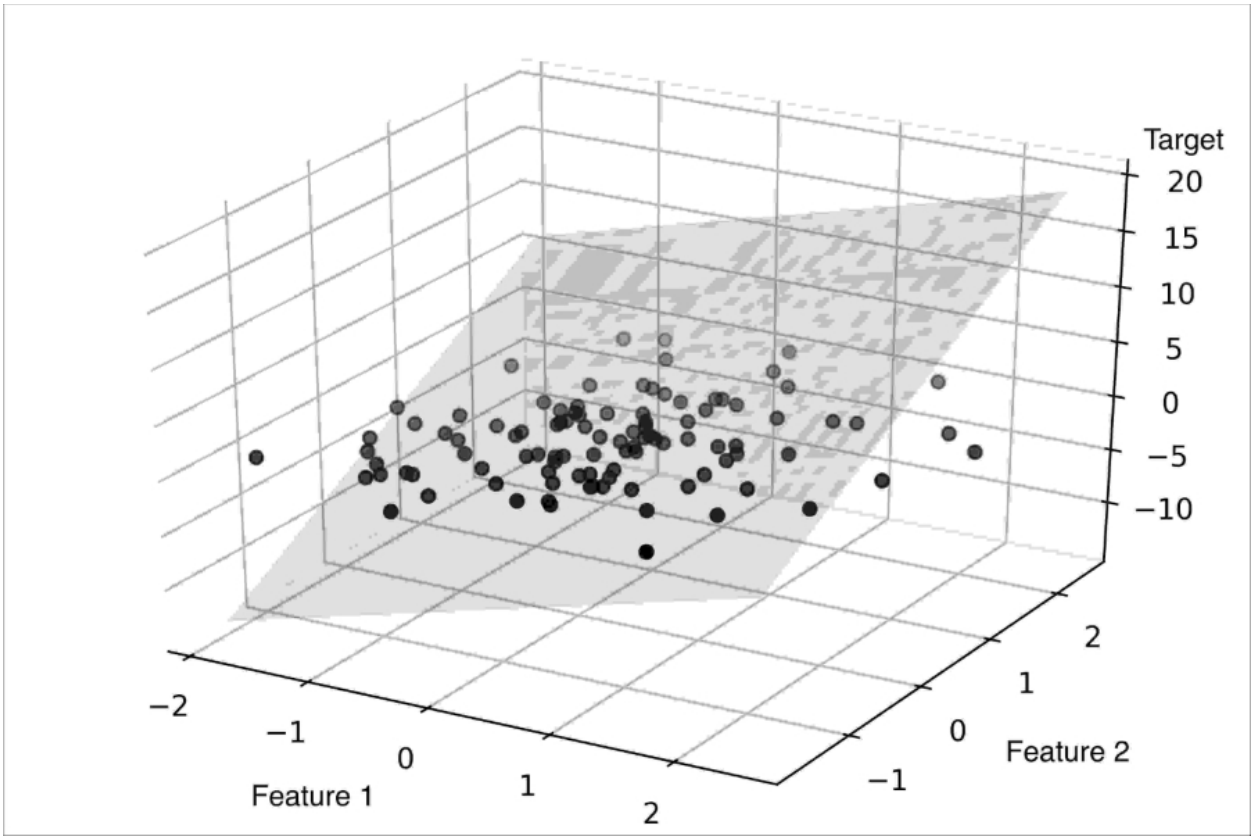


Based on the linear equation, linear regression can be understood as finding the best- fitting straight line through the sample feature points, as shown in above figure. The best fitting line also call regression line and the vertical distance from regression line to the sample feature points are called residuals or offsets which is used to calculate the error of prediction model.

Here we are going to discuss about multiple linear regression analysis.

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x$$

Here W_0 is the y-axis intercept with $X_0=1$ and $W_1, W_2, .. W_m$ are weights of sample feature. The following figure shows how the 2-dimensional, fitted hyperplane of multiple linear regression model with two features could look:



Now heading to prediction model with housing data. First we need to explore data which contains information about houses in suburbs of D.L Rubinfield in 1978. The housing data and description is available on my github account (https://github.com/jaanu). I use IBM Watson Studio to do this project. To start with IBM Watson Studio you need to create account in IBM cloud, its free.

In this section we load the housing data using pandas read_csv function which is fast and recommended tool for working with Tableau.

```
In [1]: import pandas as pd
        df=pd.read_csv('https://raw.githubusercontent.com/jaanu/Regression-Analysis/maste
        print("Data imported")

        Data imported

In [4]: # assig the column of dataset
        df.columns=['CRIM', 'ZN', 'INDUS', 'CHAS','NOX', 'RM', 'AGE', 'DIS', 'RAD','TAX',
        #check first few column of dataset
        df.head()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
1	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
2	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
3	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2
4	0.02985	0.0	2.18	0	0.458	6.430	58.7	6.0622	3	222	18.7	394.12	5.21	28.7

```
In [ ]:
```

At this point we have downloaded the data and read first five columns. Now let's check description of data. Let's have a look on description and type of data.

```
In [3]: #Look the data type of columns
        df.dtypes

Out[3]: CRIM       float64
        ZN         float64
        INDUS      float64
        CHAS         int64
        NOX        float64
        RM         float64
        AGE        float64
        DIS        float64
        RAD          int64
        TAX          int64
        PTRATIO    float64
        B          float64
        LSTAT      float64
        MEDV       float64
        dtype: object

In [4]: # data description will help us mungling dataset
        df.describe()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
count	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000	505.000000
mean	3.620667	11.350495	11.154257	0.069307	0.554728	6.284059	68.581584	3.794459	9.566337	408.459406	18.461782	356.594376	12.668257	22.529901
std	8.608572	23.343704	6.855868	0.254227	0.115990	0.703195	28.176371	2.107757	8.707553	168.629992	2.162520	91.367787	7.139950	9.205991
min	0.009060	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000	1.730000	5.000000
25%	0.082210	0.000000	5.190000	0.000000	0.449000	5.885000	45.000000	2.100000	4.000000	279.000000	17.400000	375.330000	7.010000	17.000000
50%	0.259150	0.000000	9.690000	0.000000	0.538000	6.208000	77.700000	3.199200	5.000000	330.000000	19.100000	391.430000	11.380000	21.200000
75%	3.678220	12.500000	18.100000	0.000000	0.624000	6.625000	94.100000	5.211900	24.000000	666.000000	20.200000	396.210000	16.960000	25.000000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000	37.970000	50.000000

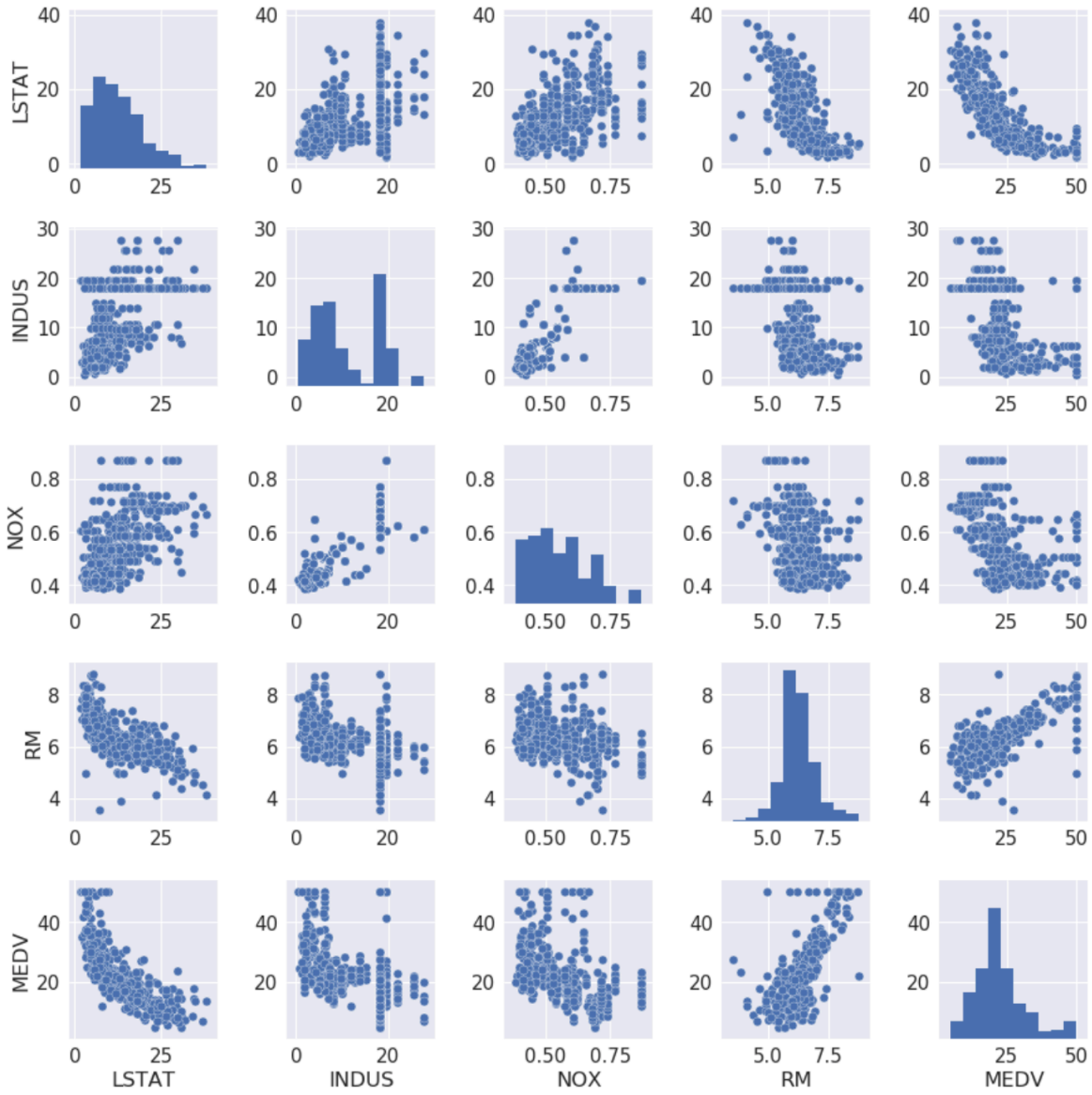
The features of the 506 samples and 14 feature in the Housing dataset are summarized here:

- `CRIM`: Per capita crime rate by town

- `ZN`: Proportion of residential land zoned for lots over 25,000 sq. ft.
- `INDUS`: Proportion of non-retail business acres per town
- `CHAS`: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- `NOX`: Nitric oxide concentration (parts per 10 million)
- `RM`: Average number of rooms per dwelling
- `AGE`: Proportion of owner-occupied units built prior to 1940
- `DIS`: Weighted distances to five Boston employment centers
- `RAD`: Index of accessibility to radial highways
- `TAX`: Full-value property tax rate per $10,000
- `PTRATIO`: Pupil-teacher ratio by town
- `B`: 1000(Bk - 0.63)^2, where *Bk* is the proportion of [people of African American descent] by town
- `LSTAT`: Percentage of lower status of the population
- `MEDV`: Median value of owner-occupied homes in $1000s

Now we will create a **scatterplot matrix** that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. We use pairplot() function seaborn to visualize the plot. Here we use five features of dataset to plot the figure. Using this scatterplot matrix, we can quickly look how the data is distributed and whether it contains outliers. For example, we can see that there is a linear relationship between `RM` and house prices, `MEDV` (the fifth column of the fourth row). Same way we can find polynomial relationship between `LSTAT` and `MEDV` `(the fifth row of the fifth column).`

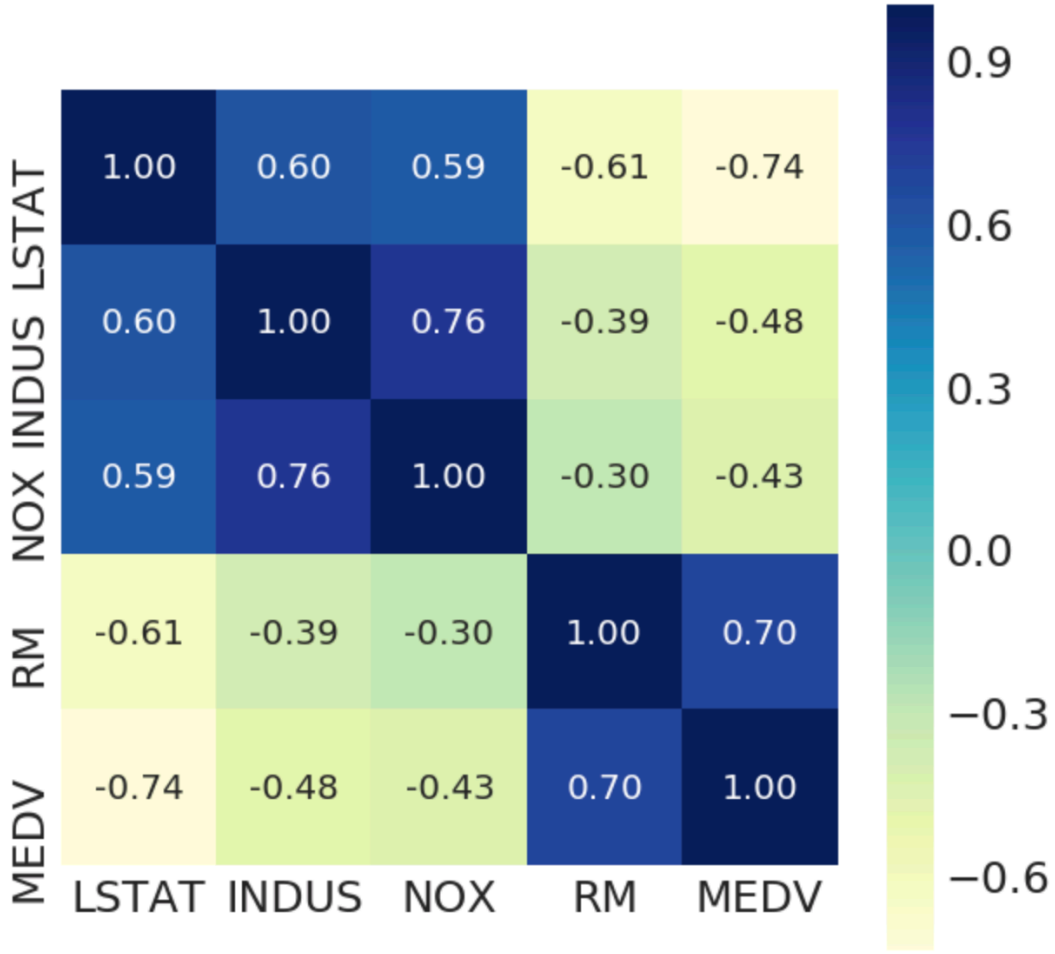
:
```
# create scatterplot matrix using seaborn that allows us to
#visualize the pair-wise correlation between the differnet features in
#this data setin one place.
import matplotlib.pyplot as plt
import seaborn as sns
cols=['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
sns.pairplot(df[cols],size=2.5)
plt.tight_layout()
plt.show()
```



You can use all feature of dataset to plot the scatter matrix using sns.pairplot(df,size=2.5)
Now create correlation matrix which is identical to covariance matrix computer from standardized features. The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficient** (often abbreviated as **Pearson's r**), which measure the linear dependence between pairs of features. The correlation coefficients are in the range -1 to 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$

we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use Seaborn's `heatmap` function to plot the correlation matrix array as a heat map:

```
: # create Correlation matrix
import numpy as np
cm=np.corrcoef(df[cols].values.T)
plt.figure(figsize=(10,10))
sns.set(font_scale=2.5)
hm=sns.heatmap(cm,cbar=True,cmap="YlGnBu", annot=True, square=True,fmt='.2f',annot_
plt.show()
```



To fit a linear regression model, we are interested in those features that have a high correlation with our target variable `MEDV`. Looking at the previous correlation matrix, we see that our target variable `MEDV` shows the largest correlation with the `LSTAT` variable (`-0.74`); however, as you might remember from inspecting the scatterplot matrix, there is a clear nonlinear relationship between `LSTAT` and `MEDV`. On the other hand, the correlation between `RM` and `MEDV` is also relatively high (`0.70`). Given the linear relationship between these two variables that we observed in the scatterplot, `RM` seems to be a good choice for an exploratory variable to introduce the concepts of a simple linear regression model in the following section.

Let's implement linear regression model using gradient descent optimization algorithm. We will do from scratch to implement the model. The following code is for LinearRegressionGD:

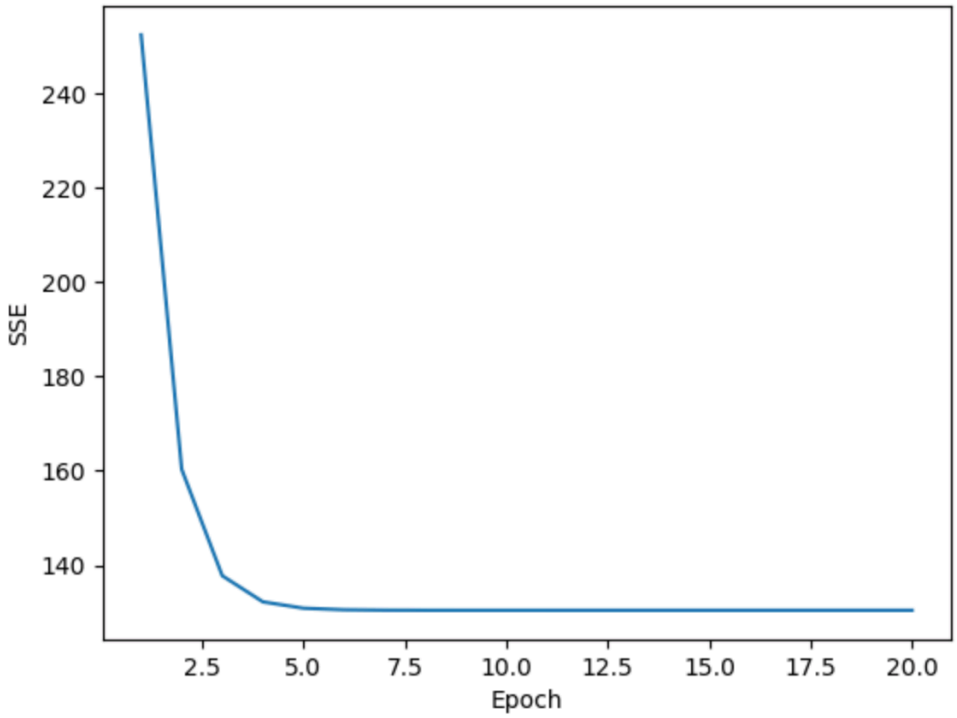
```
# linear regression using Gradient descent
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

To see our `LinearRegressionGD()` regressor in action, let's use the `RM` (number of rooms) variable from the Housing dataset as the explanatory variable and train a model that can predict `MEDV` (house prices). Furthermore, we will standardize the variables for better convergence of the GD algorithm. The code is as follows:

```
#let's use RM (number of rooms) variable
#from housing dataset to predict MEDV(House prices)
X=df[['RM']].values
y=df['MEDV'].values
from sklearn.preprocessing import StandardScaler
sc_x=StandardScaler()
sc_y=StandardScaler()
X_std=sc_x.fit_transform(X)
y_std=sc_y.fit_transform(y[:,np.newaxis]).flatten()
lr=LinearRegressionGD()
lr.fit(X_std,y_std)
```

Notice the workaround regarding `y_std`, using `np.newaxisx` and `flatten`. Most transformers in scikit-learn expect data to be stored in two-dimensional arrays. In the previous code example, the use of `np.newaxis` in `y[:, np.newaxis]` added a new dimension to the array. Then, after the `StandardScaler` returned the scaled variable, we converted it back to the original one-dimensional array representation using the `flatten()` method for our convenience.
Now plot cost function find whether it is converged or not. Following code has plot below graph.

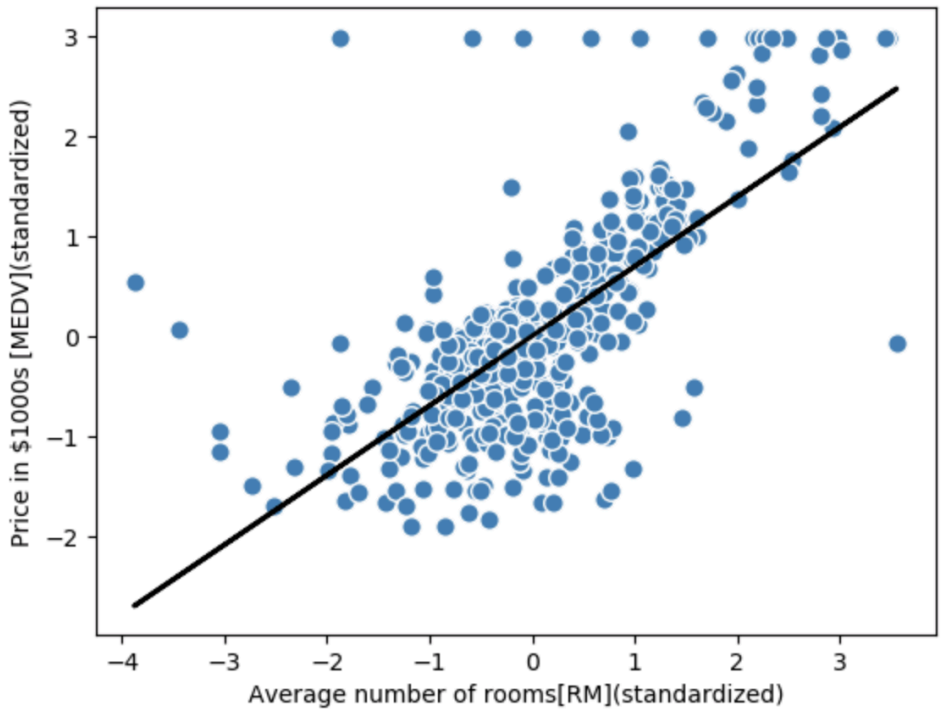
```
sns.reset_orig() # resets matplotlib style
plt.plot(range(1, lr.n_iter+1), lr.cost_)
plt.ylabel('SSE')
plt.xlabel('Epoch')
plt.show()
```



Next, let's visualize how well the linear regression line fits the training data. To do so, we will define a simple helper function that will plot a scatterplot of the training samples and add the regression line:

```
def lin_regplot(X,y,model):
    plt.scatter(X,y,c='steelblue',edgecolor='white',s=70)
    plt.plot(X,model.predict(X),color='black',lw=2)
    return None
```

```
lin_regplot(X_std, y_std,lr)
plt.xlabel('Average number of rooms[RM](standardized)')
plt.ylabel('Price in $1000s [MEDV](standardized)')
plt.show()
```



Although this observation makes intuitive sense, the data also tells us that the number of rooms does not explain the house prices very well in many cases.

Interestingly, we also observe that several data points lined up at $y = 3$, which suggests that the prices may have been clipped

In this code example, we used the previously trained linear regression model to predict the price of a house with five rooms. According to our model, such a house is worth $10,840.

```
num_rooms_std = sc_x.transform([[5.0]])
price_std = lr.predict(num_rooms_std)
print("Price in $1000s: %.3f" %sc_y.inverse_transform(price_std))
```

```
Price in $1000s: 10.840
```

```
print('Slope: %.3f' % lr.w_[1])
```

```
Slope: 0.695
```

```
print('Intercept: %.3f' % lr.w_[0])
```

```
Intercept: 0.000
```

As previously, we implemented a working model for regression analysis; however, in a real-world application we may be interested in more efficient implementations. For example, many of scikit-learn's estimators for regression make use of the **LIBLINEAR** library, advanced optimization algorithms, and other code optimizations that work better with unstandardized variables, which is sometimes desirable for certain applications:

```
# scikit estimator for regression make use od LIBLINEAR
#library which work better with unstandarized variables
from sklearn.linear_model import LinearRegression
slr=LinearRegression()
slr.fit(X,y)
print('Slope: %.3f' % slr.coef_[0])
```

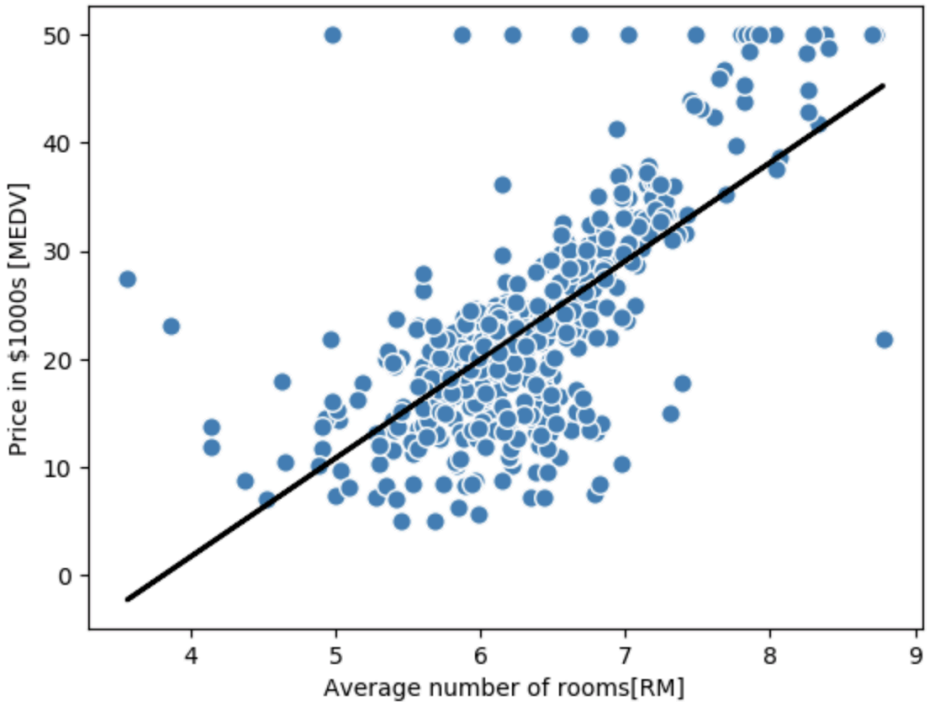
```
Slope: 9.103
```

```
print('Intercept: %.3f' % slr.intercept_)
```

```
Intercept: -34.677
```

Now, when we plot the training data and our fitted model by executing this code, we can see that the overall result looks identical to our GD implementation:

```
#Let's compare it to our GD implementation by plotting MEDV against RM:
lin_regplot(X,y,slr)
plt.xlabel('Average number of rooms[RM]')
plt.ylabel('Price in $1000s [MEDV]')
plt.show()
```



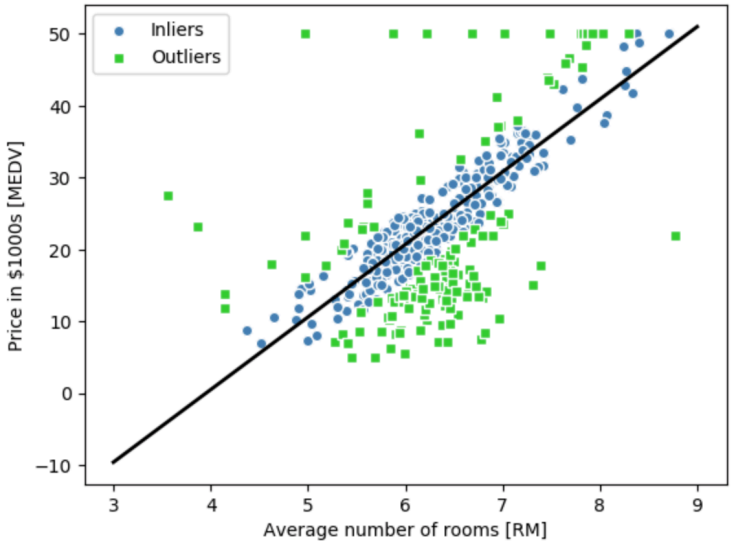
Linear regression models can be heavily impacted by the presence of outliers. In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients. To remove outliers, we will look at a robust method of regression usingthe **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called **inliers**. Let us now wrap our linear model in the RANSAC algorithm using scikit-learn's `RANSACRegressor` class:

```
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor(LinearRegression(), max_trials=100, min_samples=50, loss='absolute_loss', residual_threshold=5.0, random_state=0)
ransac.fit(X, y)
```

We set the maximum number of iterations of the `RANSACRegressor` to 100, and using `min_samples=50`, we set the minimum number of the randomly chosen samples to be at least 50. Using the `'absolute_loss'` as an argument for the `residual_metric` parameter, the algorithm computes absolute vertical distances between the fitted line and the sample points. By setting the `residual_threshold` parameter to `5.0`, we only allowed samples to be included in the inlier set if their vertical distance to the fitted line is within 5 distance units, which works well on this particular dataset.

After we fit the RANSAC model, let's obtain the inliers and outliers from the fitted RANSAC-linear regression model and plot them together with the linear fit:

```
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)
line_X = np.arange(3, 10, 1)
line_y_ransac = ransac.predict(line_X[:, np.newaxis])
plt.scatter(X[inlier_mask], y[inlier_mask],c='steelblue', edgecolor='white', marker='o', label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask],c='limegreen', edgecolor='white', marker='s', label='Outliers')
plt.plot(line_X, line_y_ransac, color='black', lw=2)
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000s [MEDV]')
plt.legend(loc='upper left')
plt.show()
```



As we can see in the following scatterplot, the linear regression model was fitted on the detected set of inliers, shown as circles in above picture.

```
: print('Slope: %.3f' % ransac.estimator_.coef_[0])
```

```
Slope: 10.099
```

```
: print('Intercept: %.3f' % ransac.estimator_.intercept_)
```

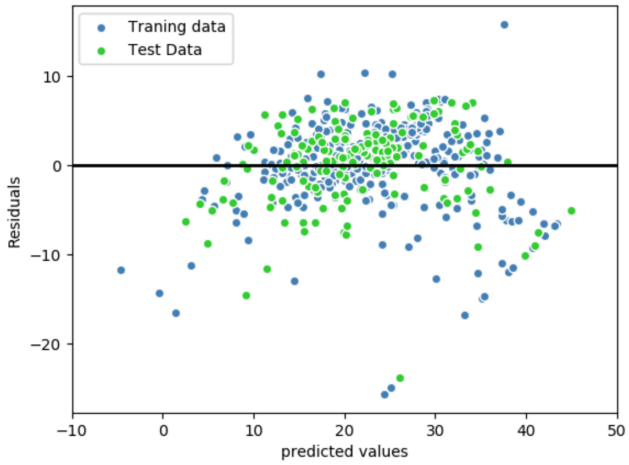
```
Intercept: -39.915
```

Using RANSAC model we can remove outliers. In the next section we will look at different approaches to evaluating a regression model, which is a crucial part of building systems for predictive modeling. Here we use all variables in the dataset and train a multiple regression model:

```
#Evaluating performance of linear regression model
#we will now use all variables in the dataset and train a multiple regression model:
from sklearn.model_selection import train_test_split
X=df.iloc[:,:-1].values
y=df['MEDV'].values
X_train,X_test, y_train, y_test=train_test_split(X,y,test_size=0.3,random_state=0)
slr=LinearRegression()
slr.fit(X_train,y_train)
y_train_pred=slr.predict(X_train)
y_test_pred=slr.predict(X_test)
```

Using the following code, we will now plot a residual plot where we simply subtract the true target variables from our predicted responses:

```
#we will now plot a residual plot where we simply subtract the true target
#variables from our predicted responses:
plt.scatter(y_train_pred,y_train_pred-y_train,c='steelblue',marker='o',edgecolor='white',label='Traning data')
plt.scatter(y_test_pred,y_test_pred-y_test,c='limegreen',marker='o',edgecolor='white',label='Test Data')
plt.xlabel('predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0,xmin=-10,xmax=50,color='black',lw=2)
plt.xlim([-10,50])
plt.show()
```



Let's compute the MSE of our training and test predictions:

```
#error of out traing and test predictions
from sklearn.metrics import mean_squared_error
print('MSE train: %.3f, test: %.3f' %(mean_squared_error(y_train, y_train_pred), mean_squared_error(y_test, y_test_pred)))
```

```
MSE train: 22.390, test: 21.382
```

```
from sklearn.metrics import r2_score
print('R^2 train: %.3f, test: %.3f' %(r2_score(y_train, y_train_pred),r2_score(y_test, y_test_pred)))
```

```
R^2 train: 0.741, test: 0.728
```

So, in this model we used multiple linear regression model to predict our target. There is polynomial regression model where we can use quadric equation. In our housing dataset MEDV and LSTAT is an example of polynomial regression model or nonlinear regression. By executing the following code, we will model the relationship between house prices and `LSTAT` (percent lower status of the population) as using second degree (quadratic) and third degree (cubic) polynomials and compare it to a linear fit.

We load STAT into X and MEDV into y for our polynomial regression model.

```
from sklearn.preprocessing import PolynomialFeatures
X = df[['LSTAT']].values
y=df[['MEDV']].values
regr=LinearRegression()
```

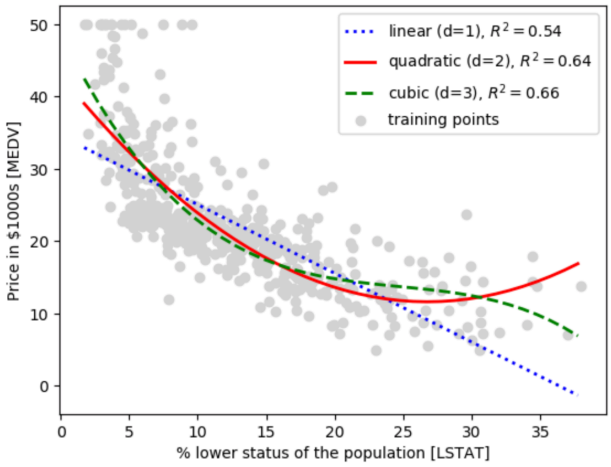
Now create the quadratic features and fit the feature into the model.

```
#create quadratic features
quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)
X_quad = quadratic.fit_transform(X)
X_cubic = cubic.fit_transform(X)
```

```
#fit features
X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
regr=regr.fit(X,y)
y_lin_fit=regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X))
regr = regr.fit(X_quad, y)
y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
quadratic_r2 = r2_score(y, regr.predict(X_quad))
regr = regr.fit(X_cubic, y)
y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
cubic_r2 = r2_score(y, regr.predict(X_cubic))
```

The resulting plot is as follows:

```
# plot results
plt.scatter(X, y, label='training points', color='lightgray')
plt.plot(X_fit, y_lin_fit,label='linear (d=1), $R^2=%.2f$' %linear_r2,color='blue', lw=2, linestyle=':')
plt.plot(X_fit, y_quad_fit,label='quadratic (d=2), $R^2=%.2f$' %quadratic_r2,color='red',lw=2,linestyle='-')
plt.plot(X_fit, y_cubic_fit,label='cubic (d=3), $R^2=%.2f$' %cubic_r2,color='green',lw=2, linestyle='--')
plt.xlabel('% lower status of the population [LSTAT]')
plt.ylabel('Price in $1000s [MEDV]')
plt.legend(loc='upper right')
plt.show()
```



As we can see, the cubic fit captures the relationship between house prices and LSTAT better than the linear and quadratic fit.

In this discussion we have gone through Linear Regression and polynomial regression model. There are other ways we can handle nonlinear datasets using Random forest and decision tree regression model. Linear Regression model used to predict continuous variable. In this tutorial we have used gradient descent to optimization algorithm. Linear regression is for predicting outcomes based on linear combinations of features. It's great when you want interpretable feature weights, want possible causal interpretations, and don't think there are interaction effects. Time complexity: O(C2n), with C features and N data points. It's dominated by a matrix multiplication. Memory Consumption: The feature matrix can get large when you have sparse features! Use sparse encodings when possible.