INT354: MACHINE LEARNING-I: NOTES

Unit IV

Predicting continuous target variables with regression analysis: Introducing Linear Regression, Fitting a Robust Regression Model using RANSAC, Relationship Using a Correlation Matrix, Exploratory Data Analysis, Regularized Methods for Regression, Polynomial Regression, Decision Tree, ARIMA

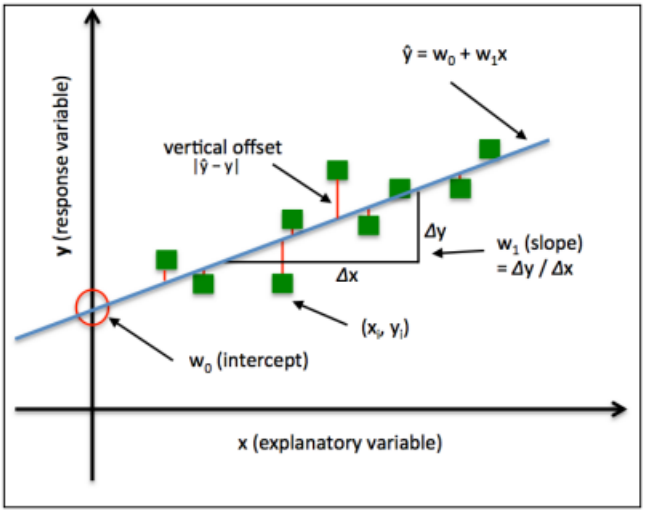
Predicting continuous target variables with regression analysis:

The goal of simple (*univariate*) linear regression is to model the relationship between a single feature (explanatory variable x) and a continuous valued *response* (target variable y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 x$$

Here, the weight w_0 represents the y axis intercepts and w_1 is the coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.

Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the sample points, as shown in the following figure:



This best-fitting line is also called the **regression line**, and the vertical lines from the regression line to the sample points are the so-called **offsets** or **residuals**—the errors of our prediction.

The special case of one explanatory variable is also called **simple linear regression**, but of course we can also generalize the linear regression model to multiple explanatory variables. Hence, this process is called **multiple linear regression**:

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

Here, w_0 is the y axis intercept with $x_0 = 1$.

Exploring the Housing Dataset

Before we implement our first linear regression model, we will introduce a new dataset, the **Housing Dataset**, which contains information about houses in the suburbs of Boston collected by D. Harrison and D.L. Rubinfeld in 1978. The *Housing Dataset* has been made freely available and can be downloaded from the *UCI machine learning repository* at `https://archive.ics.uci.edu/ml/datasets/Housing`.

The features of the 506 samples may be summarized as shown in the excerpt of the dataset description:

- **CRIM**: This is the per capita crime rate by town
- **ZN**: This is the proportion of residential land zoned for lots larger than 25,000 sq.ft.
- **INDUS**: This is the proportion of non-retail business acres per town
- **CHAS**: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)
- **NOX**: This is the nitric oxides concentration (parts per 10 million)
- **RM**: This is the average number of rooms per dwelling
- **AGE**: This is the proportion of owner-occupied units built prior to 1940
- **DIS**: This is the weighted distances to five Boston employment centers
- **RAD**: This is the index of accessibility to radial highways
- **TAX**: This is the full-value property-tax rate per $10,000
- **PTRATIO**: This is the pupil-teacher ratio by town
- **B**: This is calculated as *1000(Bk - 0.63)^2*, where Bk is the proportion of people of African American descent by town
- **LSTAT**: This is the percentage lower status of the population
- **MEDV**: This is the median value of owner-occupied homes in $1000s

---

For the rest of this chapter, we will regard the housing prices (MEDV) as our target variable—the variable that we want to predict using one or more of the 13 explanatory variables. Before we explore this dataset further, let's fetch it from the UCI repository into a pandas `DataFrame`:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data',
...                  header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

To confirm that the dataset was loaded successfully, we displayed the first five lines of the dataset, as shown in the following screenshot:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Visualizing the important characteristics of a dataset

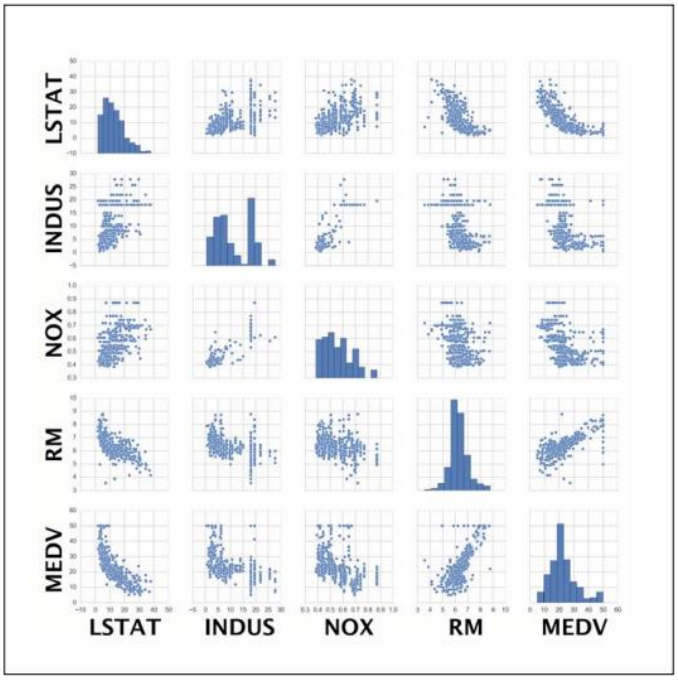
Exploratory Data Analysis (**EDA**) is an important and recommended first step prior to the training of a machine learning model. In the rest of this section, we will use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

First, 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. To plot the scatterplot matrix, we will use the `pairplot` function from the `seaborn` library (`http://stanford.edu/~mwaskom/software/seaborn/`), which is a Python library for drawing statistical plots based on matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style='whitegrid', context='notebook')
```

```
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5);
>>> plt.show()
```

As we can see in the following figure, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:



Importing the seaborn library modifies the default aesthetics of matplotlib for the current Python session. If you do not want to use seaborn's style settings, you can reset the matplotlib settings by executing the following command:

```
>>> sns.reset_orig()
```

Due to space constraints and for purposes of readability, we only plotted five columns from the dataset: **LSTAT**, **INDUS**, **NOX**, **RM**, and **MEDV**. However, you are encouraged to create a scatterplot matrix of the whole `DataFrame` to further explore the data.

Using this scatterplot matrix, we can now quickly eyeball how the data is distributed and whether it contains outliers. For example, we can see that there is a linear relationship between **RM** and the housing prices **MEDV** (the fifth column of the fourth row). Furthermore, we can see in the histogram (the lower right subplot in the scatter plot matrix) that the **MEDV** variable seems to be normally distributed but contains several outliers.

Note that in contrast to common belief, training a linear regression model does not require that the explanatory or target variables are normally distributed. The normality assumption is only a requirement for certain statistical tests and hypothesis tests that are beyond the scope of this book (Montgomery, D. C., Peck, E. A., and Vining, G. G. *Introduction to linear regression analysis*. John Wiley and Sons, 2012, pp.318–319).

To quantify the linear relationship between the features, we will now create a correlation matrix. A correlation matrix is closely related to the covariance matrix that we have seen in the section about **principal component analysis** (**PCA**) in *Chapter 4, Building Good Training Sets – Data Preprocessing*. Intuitively, we can interpret the correlation matrix as a rescaled version of the covariance matrix. In fact, the correlation matrix is identical to a covariance matrix computed from standardized data.

The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficients** (often abbreviated as **Pearson's r**), which measure the linear dependence between pairs of features. The correlation coefficients are bounded to the range -1 and 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$, respectively. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features x and y (numerator) divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Here, μ denotes the sample mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations, respectively.

We can show that the covariance between standardized features is in fact equal to their linear correlation coefficient.

Let's first standardize the features x and y, to obtain their z-scores which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we calculate the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean 0, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}(x'-0)(y'-0)$$

Through resubstitution, we get the following result:

$$\frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\frac{1}{n \cdot \sigma_x \sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

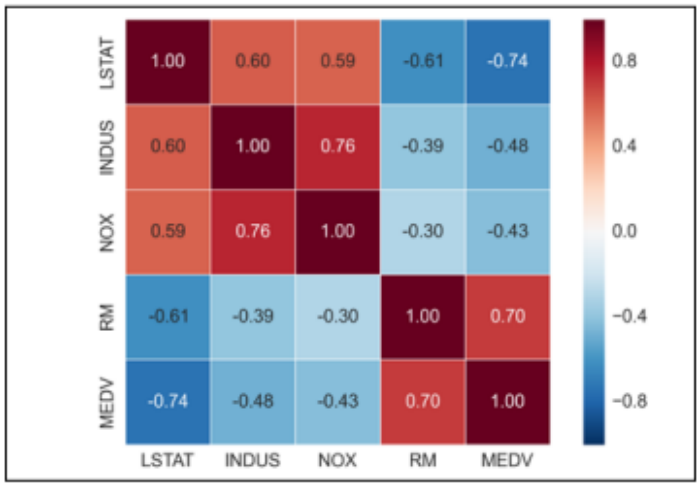
We can simplify it as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

In the following code example, 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:

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                  cbar=True,
...                  annot=True,
...                  square=True,
...                  fmt='.2f',
...                  annot_kws={'size': 15},
...                  yticklabels=cols,
...                  xticklabels=cols)
>>> plt.show()
```

As we can see in the resulting figure, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:



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 preceding 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 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) and given the linear relationship between those 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.

Implementing an ordinary least squares linear regression model

At the beginning of this chapter, we discussed that linear regression can be understood as finding the best-fitting straight line through the sample points of our training data. However, we have neither defined the term *best-fitting* nor have we discussed the different techniques of fitting such a model. In the following subsections, we will fill in the missing pieces of this puzzle using the **Ordinary Least Squares** (**OLS**) method to estimate the parameters of the regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the sample points.

Solving regression for regression parameters with gradient descent

Consider our implementation of the **ADAptive LInear NEuron** (**Adaline**) from *Chapter 2, Training Machine Learning Algorithms for Classification*; we remember that the artificial neuron uses a linear activation function and we defined a cost function $J(\cdot)$, which we minimized to learn the weights via optimization algorithms, such as **Gradient Descent** (**GD**) and **Stochastic Gradient Descent** (**SGD**). This cost function in Adaline is the **Sum of Squared Errors** (**SSE**). This is identical to the OLS cost function that we defined:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = w^T x$ (note that the term $1/2$ is just used for convenience to derive the update rule of GD). Essentially, OLS linear regression can be understood as Adaline without the unit step function so that we obtain continuous target values instead of the class labels -1 and 1. To demonstrate the similarity, let's take the GD implementation of *Adaline* from *Chapter 2, Training Machine Learning Algorithms for Classification*, and remove the unit step function to implement our first linear regression model:

```
    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)
```

If you need a refresher about how the weights are being updated—taking a step in the opposite direction of the gradient—please revisit the Adaline section in *Chapter 2, Training Machine Learning Algorithms for Classification*.

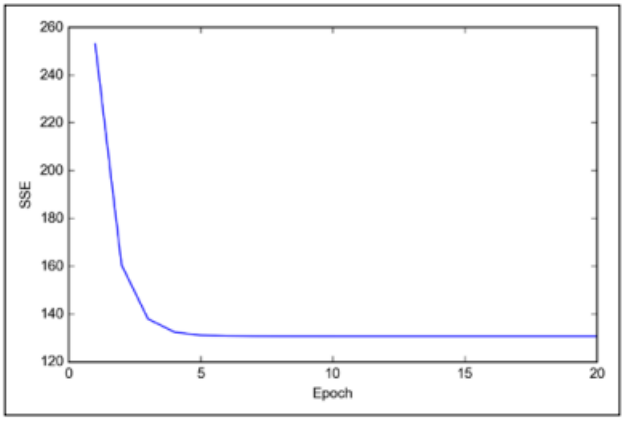
To see our `LinearRegressionGD` regressor in action, let's use the RM (number of rooms) variable from the Housing Data Set as the explanatory variable to train a model that can predict MEDV (the housing prices). Furthermore, we will standardize the variables for better convergence of the GD algorithm.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

```
>>> 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)
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

We discussed in *Chapter 2, Training Machine Learning Algorithms for Classification*, that it is always a good idea to plot the cost as a function of the number of epochs (passes over the training dataset) when we are using optimization algorithms, such as gradient descent, to check for convergence. To cut a long story short, let's plot the cost against the number of epochs to check if the linear regression has converged:

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

As we can see in the following plot, the GD algorithm converged after the fifth epoch:



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:

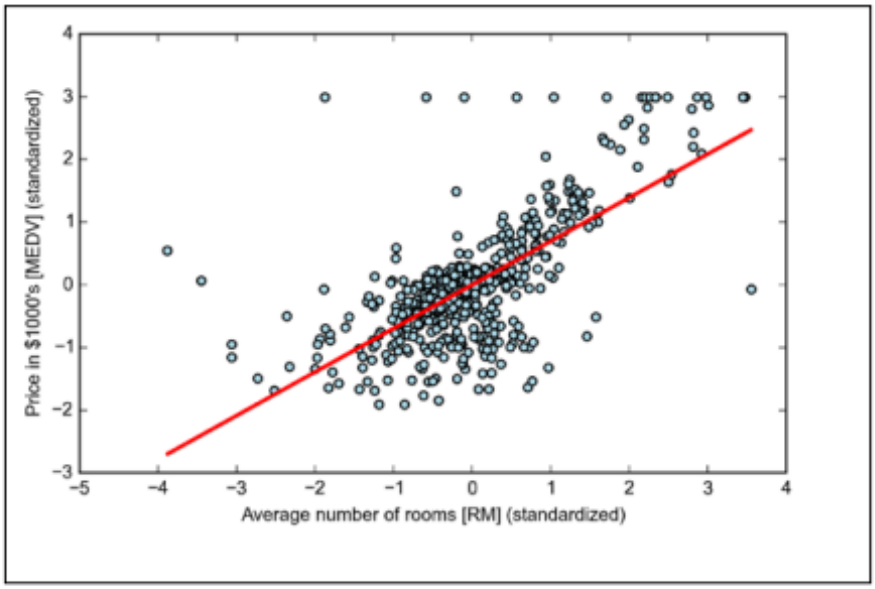
```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='blue')
...     plt.plot(X, model.predict(X), color='red')
        return None
```

```
...     return None
```

Now, we will use this `lin_regplot` function to plot the number of rooms against house prices:

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

As we can see in the following plot, the linear regression line reflects the general trend that house prices tend to increase with the number of rooms:



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. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we also observe a curious line $y = 3$, which suggests that the prices may have been clipped. In certain applications, it may also be important to report the predicted outcome variables on its original scale. To scale the predicted price outcome back on the **Price in $1000's** axes, we can simply apply the `inverse_transform` method of the `StandardScaler`:

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000's: %.3f" % \
...       sc_y.inverse_transform(price_std))
Price in $1000's: 10.840
```

In the preceding 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.

On a side note, it is also worth mentioning that we technically don't have to update the weights of the intercept if we are working with standardized variables since the y axis intercept is always 0 in those cases. We can quickly confirm this by printing the weights:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

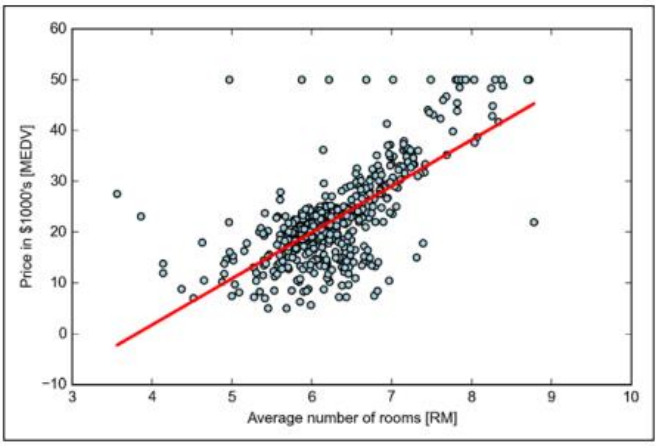
In the previous section, we implemented a working model for regression analysis. However, in a real-world application, we may be interested in more efficient implementations, for example, scikit-learn's `LinearRegression` object that makes use of the **LIBLINEAR** library and advanced optimization algorithms that work better with unstandardized variables. This is sometimes desirable for certain applications:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

As we can see by executing the preceding code, scikit-learn's `LinearRegression` model fitted with the unstandardized **RM** and **MEDV** variables yielded different model coefficients. Let's compare it to our own GD implementation by plotting MEDV against RM:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

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



Fitting a Robust Regression Model using RANSAC:

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. There are many statistical tests that can be used to detect outliers, which are beyond the scope of the book. However, removing outliers always requires our own judgment as a data scientist, as well as our domain knowledge.

As an alternative to throwing out outliers, we will look at a robust method of regression using the **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called *inliers*.

We can summarize the iterative RANSAC algorithm as follows:

1. Select a random number of samples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations has been reached; go back to step 1 otherwise.

Let's now wrap our linear model in the RANSAC algorithm using scikit-learn's `RANSACRegressor` object:

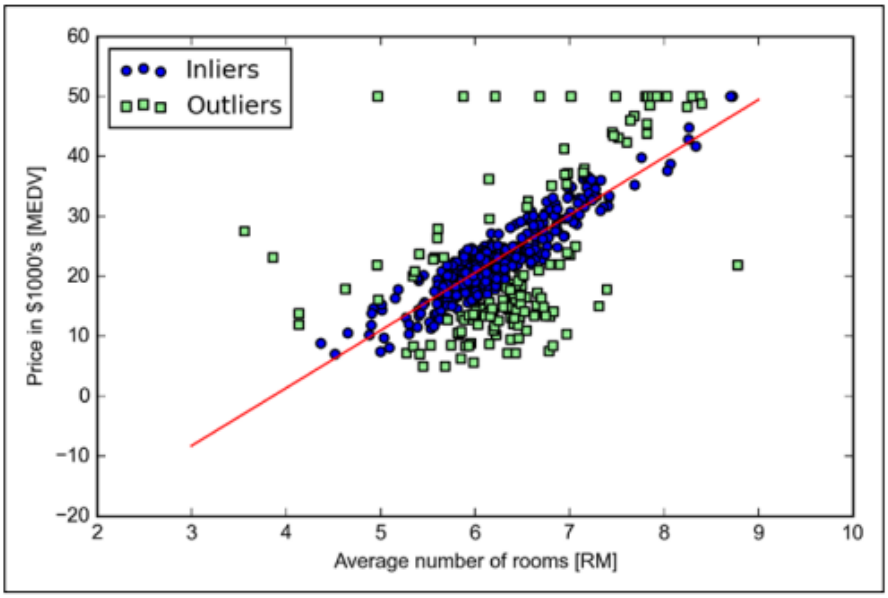
```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...                          max_trials=100,
...                          min_samples=50,
...                          residual_metric=lambda x: np.sum(np.abs(x), axis=1),
...                          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 `residual_metric` parameter, we provided a callable `lambda` function that simply calculates the 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. By default, scikit-learn uses the MAD estimate to select the inlier threshold, where **MAD** stands for the **Median Absolute Deviation** of the target values `y`. However, the choice of an appropriate value for the inlier threshold is problem-specific, which is one disadvantage of RANSAC. Many different approaches have been developed over the recent years to select a good inlier threshold automatically. You can find a detailed discussion in R. Toldo and A. Fusiello's. *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* (in Image Analysis and Processing–ICIAP 2009, pages 123–131. Springer, 2009).

After we have fitted 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='blue', marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='lightgreen', marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='red')
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [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:



When we print the slope and intercept of the model executing the following code, we can see that the linear regression line is slightly different from the fit that we obtained in the previous section without RANSAC:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 9.621
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -37.137
```

Using RANSAC, we reduced the potential effect of the outliers in this dataset, but we don't know if this approach has a positive effect on the predictive performance for unseen data. Thus, in the next section we will discuss how to evaluate a regression model for different approaches, which is a crucial part of building systems for predictive modeling.

Relationship Using a Correlation Matrix:

In the field of data science and machine learning, a correlation matrix aids in understanding relationships between variables. Correlation matrix represents how different variables interact with each other.
For someone who is navigating the complex landscape of data, understanding and harnessing the potential of correlation matrices is a skill that can significantly enhance their ability to drive meaningful insights. In this article, we will explore the step-by-step process of creating a correlation matrix in Python.

What is correlation?

Correlation is a statistical indicator that quantifies the degree to which two variables change in relation to each other. It indicates the strength and direction of the linear relationship between two variables. The correlation coefficient is denoted by "r", and it ranges from -1 to 1.

- If r = -1, it means that there is a perfect negative correlation.
- If r = 0, it means that there is no correlation between the two variables.
- If r = 1, it means that there is a perfect positive correlation.

There are two popular methods used to find the correlation coefficients:

Pearson's product-moment correlation coefficient

The Pearson correlation coefficient (r) is a measure of linear relationship between two variables.

$r = n(\sum xy) - (\sum x)(\sum y) / \sqrt{[n\sum x^2-(\sum x)^2][n\sum y^2-(\sum y)^2]}$

Here,

- n is the number of data points
- ∑xy is the sum of the product of corresponding values of x and y
- ∑x is the sum of all the values of x
- ∑y is the sum of all the values of y
- ∑x^2 is the sum of the squares of all values of x
- ∑y^2 is the sum of the squares of all the of y

Spearman's rank correlation coefficient

The Spearman's rank correlation coefficient is a measure of statistical dependence between two variables. It is based on the ranks of the data rather than the actual data values.

$\rho = 1 - \frac{6 \sum d^2}{n(n^2 -1)}$

Here,

- n is the number of paired observations
- d is the difference between the rank of corresponding values of the two variables.

What is a Correlation Matrix?

A correlation is a tabular representation that displays correlation coefficients, indicating the strength and direction of relationships between variables in a dataset. Within this matrix, each cell signifies the correlation between two specific variables. This tool serves multiple purposes, serving as a summary of data relationships, input for more sophisticated analyses, and a diagnostic aid for advanced analytical procedures. By presenting a comprehensive overview of inter-variable correlations, the matrix becomes invaluable in discerning patterns, guiding further analyses, and identifying potential areas of interest or concern in the dataset. Its applications extend beyond mere summary statistics, positioning it as a fundamental component in the preliminary stages of diverse and intricate data analyses.

Interpreting the correlation matrix

- Strong correlations, indicated by values close to 1 or -1, suggest a robust connection, while weak correlations, near 0, imply a less pronounced association. They are identifying these degrees of correlation aids in understanding the intensity of interactions within the dataset, facilitating targeted analysis and decision-making.
- Positive correlations (values > 0) signify that as one variable increases, the other tends to increase as well. Conversely, negative correlations (values < 0) imply an inverse relationship—when one variable increases, the other tends to decrease. Investigating these directional associations provides insights into how variables influence each other, crucial for formulating informed hypotheses and predictions.

How to create correlation matrix in Python?

A correlation matrix has been created using the following two libraries:

1. NumPy Library
2. Pandas Library

Creating a correlation matrix using NumPy Library

NumPy is a library for mathematical computations. It can be used for creating correlation matrices that helps to analyze the relationships between the variables through matric representation.

Example 1

Suppose an ice cream shop keeps track of total sales of ice creams versus the temperature on that day. To learn the correlation, we will use NumPy library.

In the following code snippet, x and y represent total sales in dollars and corresponding temperatures for each day of sale and **np.corrcoef()** function is sed to compute the correlation matrix.

Python3

```
import numpy as np

# x represents the total sale in dollars
x = [215, 325, 185, 332, 406, 522, 412,
     614, 544, 421, 445, 408],

# y represents the temperature on each day of sale
y = [14.2, 16.4, 11.9, 15.2, 18.5, 22.1,
     19.4, 25.1, 23.4, 18.1, 22.6, 17.2]

# create correlation matrix
matrix = np.corrcoef(x, y)
print(matrix)
```

Output:

```
[[1.         0.95750662]
 [0.95750662 1.        ]]
```

From the above matrix, if we see cell (0,1) and (1,0) both have the same value equal to 0.95750662 which lead us to conclude that whenever the temperature is high, we have more sales. Let's have a look at another example.

Example 2

Suppose we are given glucose level in boy respective to age. To find correlation between age(x) and glucose level in body(y), we will again use NumPy library. In the following code snippet, the variables x and y represent age and corresponding glucose levels.

The **np.corrcoef()** is used to calculate the correlation between the x and y variables.

Python3

```
import numpy as np

# x represents the age
x = [43, 21, 25, 42, 57, 59]

# y represents the glucose level corresponding to that age
y = [99, 65, 79, 75, 87, 81]

# correlation matrix
matrix = np.corrcoef(x, y)
print(matrix)
```

Output

```
[[1.       0.5298089]
 [0.5298089 1.       ]]
```

From the above correlation matrix, 0.5298089 or 52.98% that means the variable has a moderate positive correlation.

Creating correlation matrix using Pandas library

Pandas is a library with built-in functionalities using which user can analyze and interpret the relationships between variables.

Example 1:

To illustrate this example, we have created a data frame with three variables and have calculated the correlation matrix. In order to create a correlation matrix, we used corr() method on data frames.

Python3

```
import pandas as pd

# collect data
data = {
   'x': [45, 37, 42, 35, 39],
   'y': [38, 31, 26, 28, 33],
   'z': [10, 15, 17, 21, 12]
}

# form dataframe
dataframe = pd.DataFrame(data, columns=['x', 'y', 'z'])
print("Dataframe is : ")
print(dataframe)

# form correlation matrix
```

```
matrix = dataframe.corr()
print("Correlation matrix is : ")
print(matrix)
```

Output:

```
Dataframe is :
   x   y   z
0  45  38  10
1  37  31  15
2  42  26  17
3  35  28  21
4  39  33  12
Correlation matrix is :
          x         y         z
x  1.000000  0.518457 -0.701886
y  0.518457  1.000000 -0.860941
z -0.701886 -0.860941  1.000000
```

Example 2:

In this example, we will consider Iris dataset and find correlation between the features of the dataset.

Python3

```
from sklearn import datasets
import pandas as pd

#load iris dataset
dataset = datasets. load_iris ()
dataframe = pd. DataFrame (data = dataset. data, columns = dataset. feature_names)
dataframe ["target"] = dataset. target

#correlation matrix
matrix = dataframe.corr()
print(matrix)
```

Output:

```
                   sepal length (cm)  sepal width (cm)  petal length (cm)  \
sepal length (cm)           1.000000         -0.117570           0.871754
sepal width (cm)           -0.117570          1.000000          -0.428440
petal length (cm)           0.871754         -0.428440           1.000000
petal width (cm)            0.817941         -0.366126           0.962865
target                      0.782561         -0.426658           0.949035

                   petal width (cm)  target
sepal length (cm)          0.817941  0.782561
sepal width (cm)          -0.366126 -0.426658
petal length (cm)          0.962865  0.949035
petal width (cm)           1.000000  0.956547
target                     0.956547  1.000000
```

How to visualize correlation matrix in Python?

There are two popular libraries for data visualization, Matplotlib and Seaborn. Let's first visualize the code using Matplotlib using the iris dataset.

Correlation matrix using Matplotlib

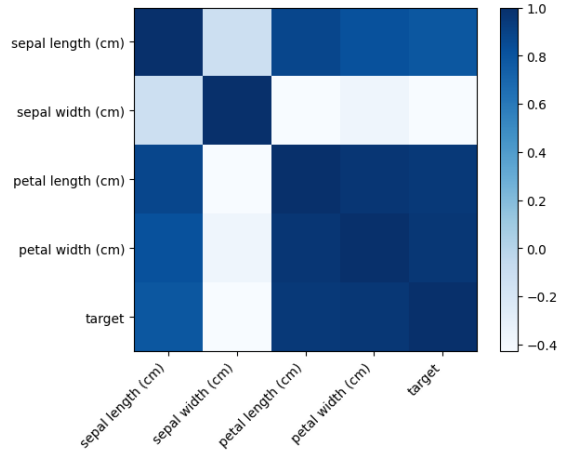
In the visualization part of the code,

- **plt.imshow()** function is used to create a heatmap.
- **plt.colobar()** is used to add the colorbar to the heat map.
- the name of the features of iris dataset are stored in the **'variables'**.
- **plt.xticks()** and **plt.yticks()** is used to add labels to the matrix.

Python3

```
import matplotlib.pyplot as plt
from sklearn import datasets
import pandas as pd
dataset = datasets. load_iris ()
dataframe = pd. DataFrame (data = dataset. data, columns = dataset. feature_names)
dataframe ["target"] = dataset. target
matrix = dataframe.corr()

#plotting correlation matrix
plt.imshow(matrix, cmap='Blues')

#adding colorbar
plt.colorbar()

#extracting variable names
variables = []
for i in matrix.columns:
  variables.append(i)

# Adding labels to the matrix
plt.xticks(range(len(matrix)), variables, rotation=45, ha='right')
plt.yticks(range(len(matrix)), variables)

# Display the plot
plt.show()
```

Output:



Heatmap of correlation matrix created using matplotlib

Correlation Matrix using Seaborn

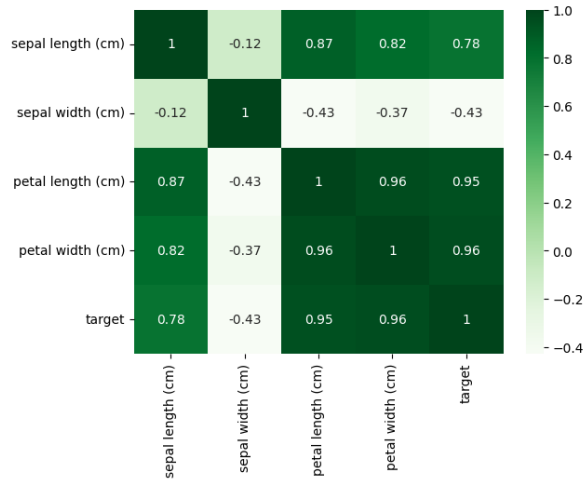
Let's visualize using Seaborn.

Python3

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
from sklearn import datasets
import pandas as pd
dataset = datasets. load_iris ()
dataframe = pd. DataFrame (data = dataset. data, columns = dataset. feature_names)
dataframe ["target"] = dataset. target
matrix = dataframe.corr()

#plotting correlation matrix
sns.heatmap(matrix, cmap="Greens", annot=True)
```

Output:



Heatmap of correlation values using Seaborn

Exploratory Data Analysis:

Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) refers to the method of studying and exploring record sets to apprehend their predominant traits, discover patterns, locate outliers, and identify relationships between variables. EDA is normally carried out as a preliminary step before undertaking extra formal statistical analyses or modeling.

The Foremost Goals of EDA

1. Data Cleaning: EDA involves examining the information for errors, lacking values, and inconsistencies. It includes techniques including records imputation, managing missing statistics, and figuring out and getting rid of outliers.

2. Descriptive Statistics: EDA utilizes precise records to recognize the important tendency, variability, and distribution of variables. Measures like suggest, median, mode, preferred deviation, range, and percentiles are usually used.

3. Data Visualization: EDA employs visual techniques to represent the statistics graphically. Visualizations consisting of histograms, box plots, scatter plots, line plots, heatmaps, and bar charts assist in identifying styles, trends, and relationships within the facts.

4. Feature Engineering: EDA allows for the exploration of various variables and their adjustments to create new functions or derive meaningful insights. Feature engineering can contain scaling, normalization, binning, encoding express variables, and creating interplay or derived variables.

5. Correlation and Relationships: EDA allows discover relationships and dependencies between variables. Techniques such as correlation analysis, scatter plots, and pass-tabulations offer insights into the power and direction of relationships between variables.
6. Data Segmentation: EDA can contain dividing the information into significant segments based totally on sure standards or traits. This segmentation allows advantage insights into unique subgroups inside the information and might cause extra focused analysis.
7. Hypothesis Generation: EDA aids in generating hypotheses or studies questions based totally on the preliminary exploration of the data. It facilitates form the inspiration for in addition evaluation and model building.
8. Data Quality Assessment: EDA permits for assessing the nice and reliability of the information. It involves checking for records integrity, consistency, and accuracy to make certain the information is suitable for analysis.

Types of EDA

Depending on the number of columns we are analyzing we can divide EDA into two types. EDA, or Exploratory Data Analysis, refers back to the method of analyzing and analyzing information units to uncover styles, pick out relationships, and gain insights. There are various sorts of EDA strategies that can be hired relying on the nature of the records and the desires of the evaluation. Here are some not unusual kinds of EDA:
1. Univariate Analysis: This sort of evaluation makes a speciality of analyzing character variables inside the records set. It involves summarizing and visualizing a unmarried variable at a time to understand its distribution, relevant tendency, unfold, and different applicable records. Techniques like histograms, field plots, bar charts, and precis information are generally used in univariate analysis.
2. Bivariate Analysis: Bivariate evaluation involves exploring the connection between variables. It enables find associations, correlations, and dependencies between pairs of variables. Scatter plots, line plots, correlation matrices, and move-tabulation are generally used strategies in bivariate analysis.
3. Multivariate Analysis: Multivariate analysis extends bivariate evaluation to encompass greater than variables. It ambitions to apprehend the complex interactions and dependencies among more than one variables in a records set. Techniques inclusive of heatmaps, parallel coordinates, aspect analysis, and primary component analysis (PCA) are used for multivariate analysis.
4. Time Series Analysis: This type of analysis is mainly applied to statistics sets that have a temporal component. Time collection evaluation entails inspecting and modeling styles, traits, and seasonality inside the statistics through the years. Techniques like line plots, autocorrelation analysis, transferring averages, and ARIMA (AutoRegressive Integrated Moving Average) fashions are generally utilized in time series analysis.
5. Missing Data Analysis: Missing information is a not unusual issue in datasets, and it may impact the reliability and validity of the evaluation. Missing statistics analysis includes figuring out missing values, know-how the patterns of missingness, and using suitable techniques to deal with missing data. Techniques along with lacking facts styles, imputation strategies, and sensitivity evaluation are employed in lacking facts evaluation.
6. Outlier Analysis: Outliers are statistics factors that drastically deviate from the general sample of the facts. Outlier analysis includes identifying and knowledge the presence of outliers, their capability reasons, and their impact at the analysis. Techniques along with box plots, scatter plots, z-rankings, and clustering algorithms are used for outlier evaluation.
7. Data Visualization: Data visualization is a critical factor of EDA that entails creating visible representations of the statistics to facilitate understanding and exploration. Various

visualization techniques, inclusive of bar charts, histograms, scatter plots, line plots, heatmaps, and interactive dashboards, are used to represent exclusive kinds of statistics. These are just a few examples of the types of EDA techniques that can be employed at some stage in information evaluation. The choice of strategies relies upon on the information traits, research questions, and the insights sought from the analysis.

Exploratory Data Analysis (EDA) Using Python Libraries

For the simplicity of the article, we will use a single dataset. We will use the employee data for this. It contains 8 columns namely – First Name, Gender, Start Date, Last Login, Salary, Bonus%, Senior Management, and Team. We can get the dataset here Employees.csv

Let's read the dataset using the Pandas read_csv() function and print the 1st five rows. To print the first five rows we will use the head() function.

Python3

```
import pandas as pd
import numpy as np
# read datasdet using pandas
df = pd.read_csv('employees.csv')
df.head()
```

Output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	8/6/1993	12:42 PM	97308	6.945	True	Marketing
1	Thomas	Male	3/31/1996	6:53 AM	61933	4.170	True	NaN
2	Maria	Female	4/23/1993	11:17 AM	130590	11.858	False	Finance
3	Jerry	Male	3/4/2005	1:00 PM	138705	9.340	True	Finance
4	Larry	Male	1/24/1998	4:47 PM	101004	1.389	True	Client Services

First five rows of the dataframe

Getting Insights About The Dataset

Let's see the shape of the data using the shape.

Python3

```
df.shape
```

Output:

```
(1000, 8)
```

This means that this dataset has 1000 rows and 8 columns.

Let's get a quick summary of the dataset using the pandas describe() method. The describe() function applies basic statistical computations on the dataset like extreme values, count of data points standard deviation, etc. Any missing value or NaN value is automatically skipped. describe() function gives a good picture of the distribution of data.

Python3

```
df.describe()
```

	Salary	Bonus %
count	1000.000000	1000.000000
mean	90662.181000	10.207555
std	32923.693342	5.528481
min	35013.000000	1.015000
25%	62613.000000	5.401750
50%	90428.000000	9.838500
75%	118740.250000	14.838000
max	149908.000000	19.944000

description of the dataframe

Python3

```
# information about the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   First Name       933 non-null    object 
 1   Gender           855 non-null    object 
 2   Start Date       1000 non-null   object 
 3   Last Login Time  1000 non-null   object 
 4   Salary           1000 non-null   int64  
 5   Bonus %          1000 non-null   float64
 6   Senior Management  933 non-null    object 
 7   Team             957 non-null    object 
dtypes: float64(1), int64(1), object(6)
memory usage: 62.6+ KB
```

Changing Dtype from Object to Datetime

Start Date is an important column for employees. However, it is not of much use if we can not handle it properly to handle this type of data pandas provide a special function datetime() from which we can change object type to DateTime format.

Python3

```
# convert "Start Date" column to datetime data type
df['Start Date'] = pd.to_datetime(df['Start Date'])
```

We can see the number of unique elements in our dataset. This will help us in deciding which type of encoding to choose for converting categorical columns into numerical columns.

Python3

```
df.nunique()
```

Output:

```
First Name           200
Gender                 2
Start Date           972
Last Login Time      720
Salary               995
Bonus %              971
Senior Management      2
Team                  10
dtype: int64
```

Handling Missing Values

You all must be wondering why a dataset will contain any missing values. It can occur when no information is provided for one or more items or for a whole unit. For Example, Suppose different users being surveyed may choose not to share their income, and some users may choose not to share their address in this way many datasets went missing. Missing Data is a very big problem in real-life scenarios. Missing Data can also refer to as NA(Not Available) values in pandas. There are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame :

- isnull()
- notnull()
- dropna()
- fillna()
- replace()
- interpolate()

Now let's check if there are any missing values in our dataset or not.

Example:

Python3

```
df.isnull().sum()
```

```
First Name             67
Gender                145
Start Date              0
Last Login Time         0
Salary                  0
Bonus %                 0
Senior Management      67
Team                   43
dtype: int64
```

Null values in dataframe

We can see that every column has a different amount of missing values. Like Gender has 145 missing values and salary has 0. Now for handling these missing values there can be several cases like dropping the rows containing NaN or replacing NaN with either mean, median, mode, or some other value.

Now, let's try to fill in the missing values of gender with the string "No Gender".

Python3

```
df["Gender"].fillna("No Gender", inplace = True)

df.isnull().sum()
```

```
First Name           67
Gender                0
Start Date            0
Last Login Time       0
Salary                0
Bonus %               0
Senior Management    67
Team                 43
dtype: int64
```

Null values in dataframe after filling Gender column

We can see that now there is no null value for the gender column. Now, Let's fill the senior management with the mode value.

Example:

Python3

```
mode = df['Senior Management'].mode().values[0]
df['Senior Management']= df['Senior Management'].replace(np.nan, mode)

df.isnull().sum()
```

Output:

```
First Name           67
Gender                0
Start Date            0
Last Login Time       0
Salary                0
Bonus %               0
Senior Management     0
Team                 43
dtype: int64
```

Null values in dataframe after filling S senior management column

Now for the first name and team, we cannot fill the missing values with arbitrary data, so, let's drop all the rows containing these missing values.

Example:

Python3

```
df = df.dropna(axis = 0, how ='any')

print(df.isnull().sum())
df.shape
```

Output:

```
First Name           0
Gender               0
Start Date           0
Last Login Time      0
Salary               0
Bonus %              0
Senior Management    0
Team                 0
dtype: int64

(899, 8)
```

Null values in dataframe after dropping all null values

We can see that our dataset is now free of all the missing values and after dropping the data the number of rows also reduced from 1000 to 899.

Note: For more information, refer to Working with Missing Data in Pandas.

Data Encoding

There are some models like Linear Regression which does not work with categorical dataset in that case we should try to encode categorical dataset into the numerical column. we can use different methods for encoding like Label encoding or One-hot encoding. pandas and sklearn provide different functions for encoding in our case we will use the LabelEncoding function from sklearn to encode the *Gender* column.

Python3

```
from sklearn.preprocessing import LabelEncoder
# create an instance of LabelEncoder
le = LabelEncoder()

# fit and transform the "Senior Management"
# column with LabelEncoder
df['Gender'] = le.fit_transform\
                    (df['Gender'])
```

Noe

Data visualization

Data Visualization is the process of analyzing data in the form of graphs or maps, making it a lot easier to understand the trends or patterns in the data.

Let's see some commonly used graphs –

Note: *We will use Matplotlib and Seaborn library for the data visualization. If you want to know about these modules refer to the articles –*

- *Matplotlib Tutorial*
- *Python Seaborn Tutorial*

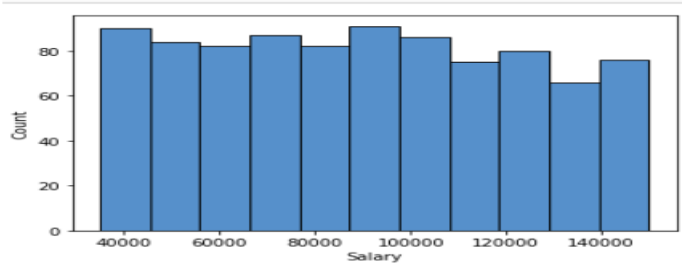
Histogram

It can be used for both uni and bivariate analysis.

Example:

Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt


sns.histplot(x='Salary', data=df, )
plt.show()
```



Histogram plot of salary column

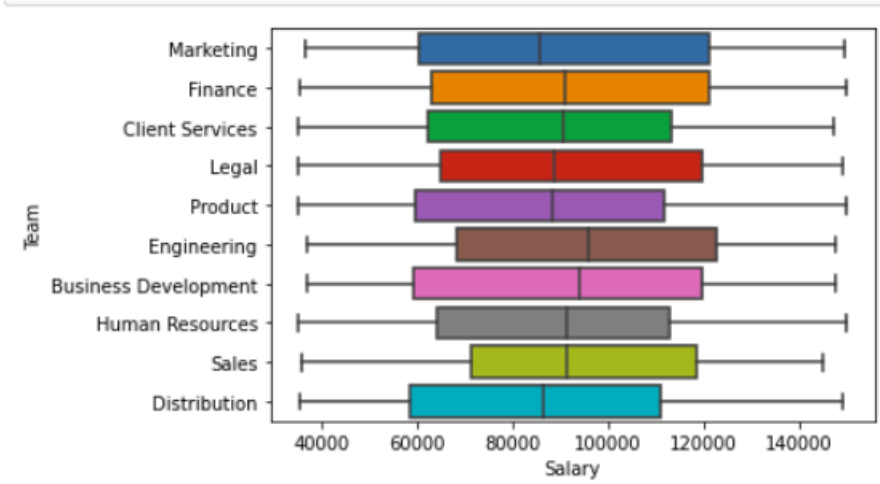
bivariate analyses.

Boxplot

It can also be used for univariate and bivariate analyses.

Example:

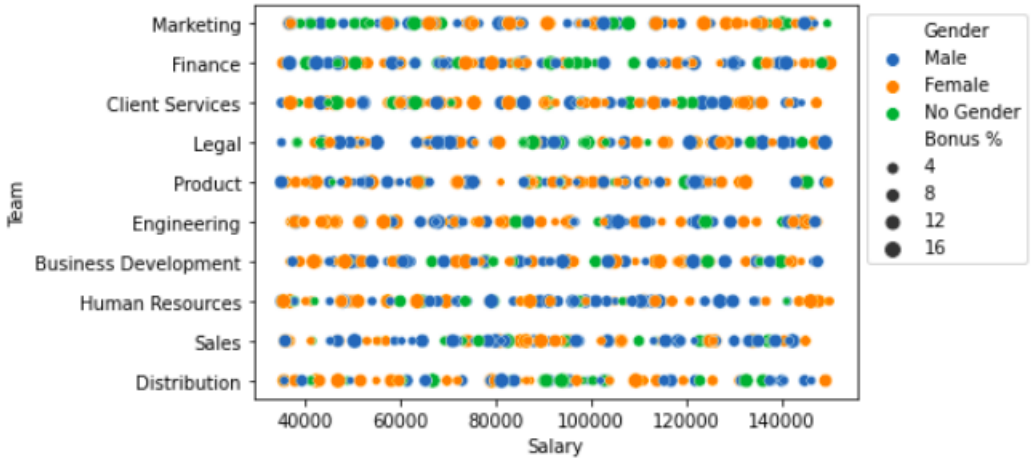
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt


sns.boxplot( x="Salary", y='Team', data=df, )
plt.show()
```



Boxplot of Salary and team column

Scatter Boxplot For Data Visualization
It can be used for bivariate analyses.
Example:

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt


sns.scatterplot( x="Salary", y='Team', data=df,
                hue='Gender', size='Bonus %')

# Placing Legend outside the Figure
plt.legend(bbox_to_anchor=(1, 1), loc=2)

plt.show()
```

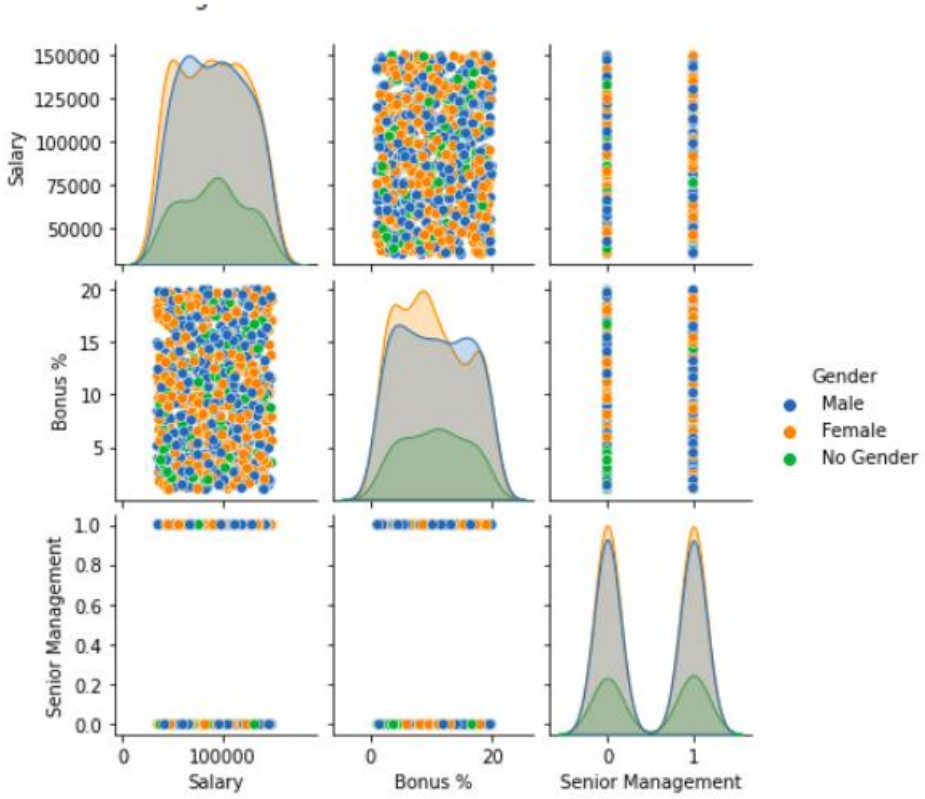


Scatter plot of salary and Team column

For multivariate analysis, we can use pairplot()method of the seaborn module. We can also use it for the multiple pairwise bivariate distributions in a dataset.

Example:

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt


sns.pairplot(df, hue='Gender', height=2)
```



Pairplot of columns of dataframe

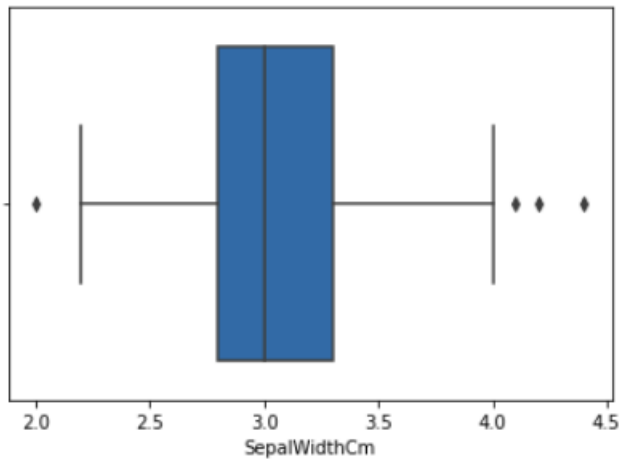
Handling Outliers

An Outlier is a data item/object that deviates significantly from the rest of the (so-called normal)objects. They can be caused by measurement or execution errors. The analysis for outlier detection is referred to as outlier mining. There are many ways to detect outliers, and the removal process of these outliers from the dataframe is the same as removing a data item from the panda's dataframe.

Let's consider the iris dataset and let's plot the boxplot for the SepalWidthCm column.

Example:

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('Iris.csv')

sns.boxplot(x='SepalWidthCm', data=df)
```



Boxplot of sample width column before outliers removal

In the above graph, the values above 4 and below 2 are acting as outliers.

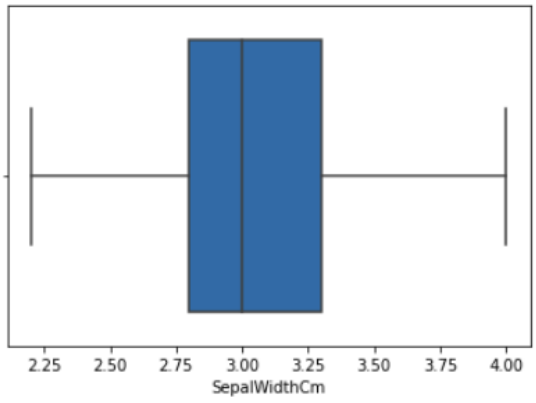
Removing Outliers

For removing the outlier, one must follow the same process of removing an entry from the dataset using its exact position in the dataset because in all the above methods of detecting the outliers end result is the list of all those data items that satisfy the outlier definition according to the method used.

Example: We will detect the outliers using IQR and then we will remove them. We will also draw the boxplot to see if the outliers are removed or not.

```
# Importing
import sklearn
from sklearn.datasets import load_boston
import pandas as pd
import seaborn as sns

# Load the dataset
df = pd.read_csv('Iris.csv')

# IQR
Q1 = np.percentile(df['SepalWidthCm'], 25,
                   interpolation = 'midpoint')

Q3 = np.percentile(df['SepalWidthCm'], 75,
                   interpolation = 'midpoint')
IQR = Q3 - Q1

print("Old Shape: ", df.shape)

# Upper bound
upper = np.where(df['SepalWidthCm'] >= (Q3+1.5*IQR))

# Lower bound
lower = np.where(df['SepalWidthCm'] <= (Q1-1.5*IQR))

# Removing the Outliers
df.drop(upper[0], inplace = True)
df.drop(lower[0], inplace = True)

print("New Shape: ", df.shape)

sns.boxplot(x='SepalWidthCm', data=df)
```

```
Old Shape:  (150, 6)
New Shape:  (146, 6)

<AxesSubplot:xlabel='SepalWidthCm'>
```



Boxplot of sample width after outlier removal

Note: for more information, refer Detect and Remove the Outliers using Python

These are some of the EDA we do during our data science project however it depends upon your requirement and how much data analysis we do.

Regularized Methods for Regression:

What is Regularization?

Regularization is one of the most important concepts of machine learning. It is a technique to prevent the model from overfitting by adding extra information to it.

Sometimes the machine learning model performs well with the training data but does not perform well with the test data. It means the model is not able to predict the output when deals with unseen data by introducing noise in the output, and hence the model is called overfitted. This problem can be deal with the help of a regularization technique.

This technique can be used in such a way that it will allow to maintain all variables or features in the model by reducing the magnitude of the variables. Hence, it maintains accuracy as well as a generalization of the model.

It mainly regularizes or reduces the coefficient of features toward zero. In simple words, "*In regularization technique, we reduce the magnitude of the features by keeping the same number of features.*"

How does Regularization Work?

Regularization works by adding a penalty or complexity term to the complex model. Let's consider the simple linear regression equation:

$$y= \beta0+\beta1x1+\beta2x2+\beta3x3+\cdots+\beta nxn +b$$

In the above equation, Y represents the value to be predicted

X1, X2, …Xn are the features for Y.

β0,β1,…..βn are the weights or magnitude attached to the features, respectively. Here represents the bias of the model, and b represents the intercept.

Linear regression models try to optimize the β0 and b to minimize the cost function. The equation for the cost function for the linear model is given below:

$$\sum_{i=1}^{M}\left(y_i - y'_i\right)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * Xij\right)^2$$

Now, we will add a loss function and optimize parameter to make the model that can predict the accurate value of Y. The loss function for the linear regression is called as **RSS or Residual sum of squares.**

Techniques of Regularization

There are mainly two types of regularization techniques, which are given below:

- **Ridge Regression**
- **Lasso Regression**

Ridge Regression

- Ridge regression is one of the types of linear regression in which a small amount of bias is introduced so that we can get better long-term predictions.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as **L2 regularization**.
- In this technique, the cost function is altered by adding the penalty term to it. The amount of bias added to the model is called **Ridge Regression penalty**. We can calculate it by multiplying with the lambda to the squared weight of each individual feature.
- The equation for the cost function in ridge regression will be:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}\beta_j{}^2$$

- In the above equation, the penalty term regularizes the coefficients of the model, and hence ridge regression reduces the amplitudes of the coefficients that decreases the complexity of the model.
- As we can see from the above equation, if the values of **λ tend to zero, the equation becomes the cost function of the linear regression model.** Hence, for the minimum value of λ, the model will resemble the linear regression model.
- A general linear or polynomial regression will fail if there is high collinearity between the independent variables, so to solve such problems, Ridge regression can be used.
- It helps to solve the problems if we have more parameters than samples.

Lasso Regression:

- Lasso regression is another regularization technique to reduce the complexity of the model. It stands for **Least Absolute and Selection Operator.**
- It is similar to the Ridge Regression except that the penalty term contains only the absolute weights instead of a square of weights.

- Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0.
- It is also called as **L1 regularization.** The equation for the cost function of Lasso regression will be:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}|\beta_j|$$

- Some of the features in this technique are completely neglected for model evaluation.
- Hence, the Lasso regression can help us to reduce the overfitting in the model as well as the feature selection.

Key Difference between Ridge Regression and Lasso Regression

- **Ridge regression** is mostly used to reduce the overfitting in the model, and it includes all the features present in the model. It reduces the complexity of the model by shrinking the coefficients.
- **Lasso regression** helps to reduce the overfitting in the model as well as feature selection.

Polynomial Regression:

- Polynomial Regression is a regression algorithm that models the relationship between a dependent(y) and independent variable(x) as nth degree polynomial. The Polynomial Regression equation is given below:

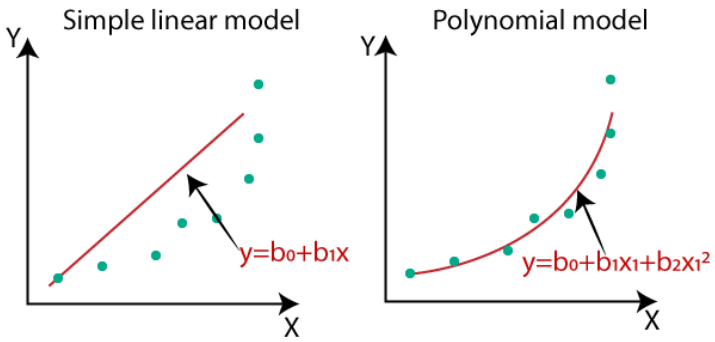
$y= b_0+b_1x_1+ b_2x_1^2+ b_2x_1^3+..... b_nx_1^n$

- It is also called the special case of Multiple Linear Regression in ML. Because we add some polynomial terms to the Multiple Linear regression equation to convert it into Polynomial Regression.
- It is a linear model with some modification in order to increase the accuracy.
- The dataset used in Polynomial regression for training is of non-linear nature.
- It makes use of a linear regression model to fit the complicated and non-linear functions and datasets.
- **Hence,** ***"In Polynomial regression, the original features are converted into Polynomial features of required degree (2,3,..,n) and then modeled using a linear model."***

Need for Polynomial Regression:

The need of Polynomial Regression in ML can be understood in the below points:

- If we apply a linear model on a **linear dataset**, then it provides us a good result as we have seen in Simple Linear Regression, but if we apply the same model without any modification on a **non-linear dataset**, then it will produce a drastic output. Due to which loss function will increase, the error rate will be high, and accuracy will be decreased.
- So for such cases, **where data points are arranged in a non-linear fashion, we need the Polynomial Regression model**. We can understand it in a better way using the below comparison diagram of the linear dataset and non-linear dataset.



- In the above image, we have taken a dataset which is arranged non-linearly. So if we try to cover it with a linear model, then we can clearly see that it hardly covers any data point. On the other hand, a curve is suitable to cover most of the data points, which is of the Polynomial model.
- Hence, *if the datasets are arranged in a non-linear fashion, then we should use the Polynomial Regression model instead of Simple Linear Regression.*

Note: A Polynomial Regression algorithm is also called Polynomial Linear Regression because it does not depend on the variables, instead, it depends on the coefficients, which are arranged in a linear fashion.

Equation of the Polynomial Regression Model:

Simple Linear Regression equation: $y = b_0+b_1x$ .....(a)

Multiple Linear Regression equation: $y= b_0+b_1x+ b_2x_2+ b_3x_3+....+ b_nx_n$ .....(b)

Polynomial Regression equation: $y= b_0+b_1x + b_2x^2+ b_3x^3+....+ b_nx^n$ .....(c)

When we compare the above three equations, we can clearly see that all three equations are Polynomial equations but differ by the degree of variables. The Simple and Multiple Linear

equations are also Polynomial equations with a single degree, and the Polynomial regression equation is Linear equation with the nth degree. So if we add a degree to our linear equations, then it will be converted into Polynomial Linear equations.

Note: To better understand Polynomial Regression, you must have knowledge of Simple Linear Regression.

Implementation of Polynomial Regression using Python:

Here we will implement the Polynomial Regression using Python. We will understand it by comparing Polynomial Regression model with the Simple Linear Regression model. So first, let's understand the problem for which we are going to build the model.

Problem Description: There is a Human Resource company, which is going to hire a new candidate. The candidate has told his previous salary 160K per annum, and the HR have to check whether he is telling the truth or bluff. So to identify this, they only have a dataset of his previous company in which the salaries of the top 10 positions are mentioned with their levels. By checking the dataset available, we have found that there is a **non-linear relationship between the Position levels and the salaries**. Our goal is to build a **Bluffing detector regression** model, so HR can hire an honest candidate. Below are the steps to build such a model.

Position	Level(X-variable)	Salary(Y-Variable)
Business Analyst	1	45000
Junior Consultant	2	50000
Senior Consultant	3	60000
Manager	4	80000
Country Manager	5	110000
Region Manager	6	150000
Partner	7	200000
Senior Partner	8	300000
C-level	9	500000
CEO	10	1000000

Steps for Polynomial Regression:

The main steps involved in Polynomial Regression are given below:

- Data Pre-processing
- Build a Linear Regression model and fit it to the dataset
- Build a Polynomial Regression model and fit it to the dataset
- Visualize the result for Linear Regression and Polynomial Regression model.
- Predicting the output.

Note: Here, we will build the Linear regression model as well as Polynomial Regression to see the results between the predictions. And Linear regression model is for reference.

Data Pre-processing Step:

The data pre-processing step will remain the same as in previous regression models, except for some changes. In the Polynomial Regression model, we will not use feature scaling, and also we will not split our dataset into training and test set. It has two reasons:

- The dataset contains very less information which is not suitable to divide it into a test and training set, else our model will not be able to find the correlations between the salaries and levels.
- In this model, we want very accurate predictions for salary, so the model should have enough information.

The code for pre-processing step is given below:

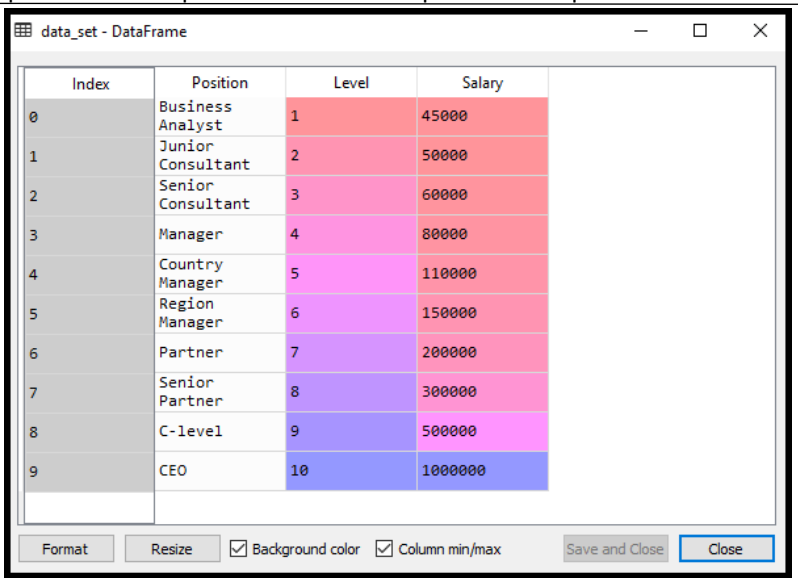
```
1. # importing libraries
2. import numpy as nm
3. import matplotlib.pyplot as mtp
4. import pandas as pd
5.
6. #importing datasets
7. data_set= pd.read_csv('Position_Salaries.csv')
8.
9. #Extracting Independent and dependent Variable
10. x= data_set.iloc[:, 1:2].values
11. y= data_set.iloc[:, 2].values
```

Explanation:

- In the above lines of code, we have imported the important Python libraries to import dataset and operate on it.
- Next, we have imported the dataset '**Position_Salaries.csv**', which contains three columns (Position, Levels, and Salary), but we will consider only two columns (Salary and Levels).
- After that, we have extracted the dependent(Y) and independent variable(X) from the dataset. For x-variable, we have taken parameters as [:,1:2], because we want 1 index(levels), and included :2 to make it as a matrix.

Output:

By executing the above code, we can read our dataset as:



Index	Position	Level	Salary
0	Business Analyst	1	45000
1	Junior Consultant	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Region Manager	6	150000
6	Partner	7	200000
7	Senior Partner	8	300000
8	C-level	9	500000
9	CEO	10	1000000

As we can see in the above output, there are three columns present (Positions, Levels, and Salaries). But we are only considering two columns because Positions are equivalent to the levels or may be seen as the encoded form of Positions.

Here we will predict the output for level **6.5** because the candidate has 4+ years' experience as a regional manager, so he must be somewhere between levels 7 and 6.

Building the Linear regression model:

Now, we will build and fit the Linear regression model to the dataset. In building polynomial regression, we will take the Linear regression model as reference and compare both the results. The code is given below:

```
1. #Fitting the Linear Regression to the dataset
2. from sklearn.linear_model import LinearRegression
3. lin_regs= LinearRegression()
4. lin_regs.fit(x,y)
```

In the above code, we have created the Simple Linear model using **lin_regs** object of **LinearRegression** class and fitted it to the dataset variables (x and y).

Output:

```
Out[5]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Building the Polynomial regression model:

Now we will build the Polynomial Regression model, but it will be a little different from the Simple Linear model. Because here we will use **PolynomialFeatures** class of **preprocessing** library. We are using this class to add some extra features to our dataset.

```
1. #Fitting the Polynomial regression to the dataset
```

2. from sklearn.preprocessing **import** PolynomialFeatures
3. poly_regs= PolynomialFeatures(degree= 2)
4. x_poly= poly_regs.fit_transform(x)
5. lin_reg_2 =LinearRegression()
6. lin_reg_2.fit(x_poly, y)

In the above lines of code, we have used **poly_regs.fit_transform(x)**, because first we are converting our feature matrix into polynomial feature matrix, and then fitting it to the Polynomial regression model. The parameter value(degree= 2) depends on our choice. We can choose it according to our Polynomial features.

After executing the code, we will get another matrix **x_poly**, which can be seen under the variable explorer option:

	0	1	2
0	1	1	1
1	1	2	4
2	1	3	9
3	1	4	16
4	1	5	25
5	1	6	36
6	1	7	49
7	1	8	64
8	1	9	81
9	1	10	100

Next, we have used another LinearRegression object, namely **lin_reg_2**, to fit our **x_poly** vector to the linear model.

Output:

```
Out[11]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

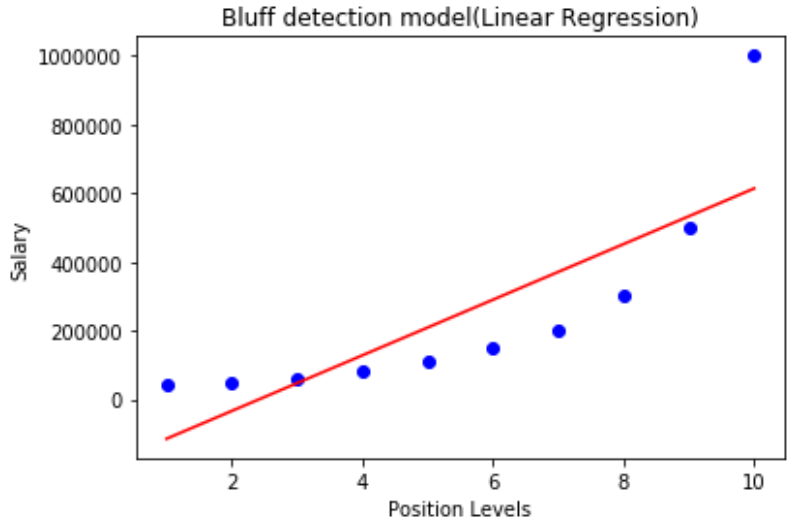
Visualizing the result for Linear regression:

Now we will visualize the result for Linear regression model as we did in Simple Linear Regression. Below is the code for it:

1. #Visulaizing the result **for** Linear Regression model
2. mtp.scatter(x,y,color="blue")
3. mtp.plot(x,lin_regs.predict(x), color="red")
4. mtp.title("Bluff detection model(Linear Regression)")

5. mtp.xlabel("Position Levels")
6. mtp.ylabel("Salary")
7. mtp.show()

Output:



In the above output image, we can clearly see that the regression line is so far from the datasets. Predictions are in a red straight line, and blue points are actual values. If we consider this output to predict the value of CEO, it will give a salary of approx. 600000$, which is far away from the real value.

So we need a curved model to fit the dataset other than a straight line.

Visualizing the result for Polynomial Regression

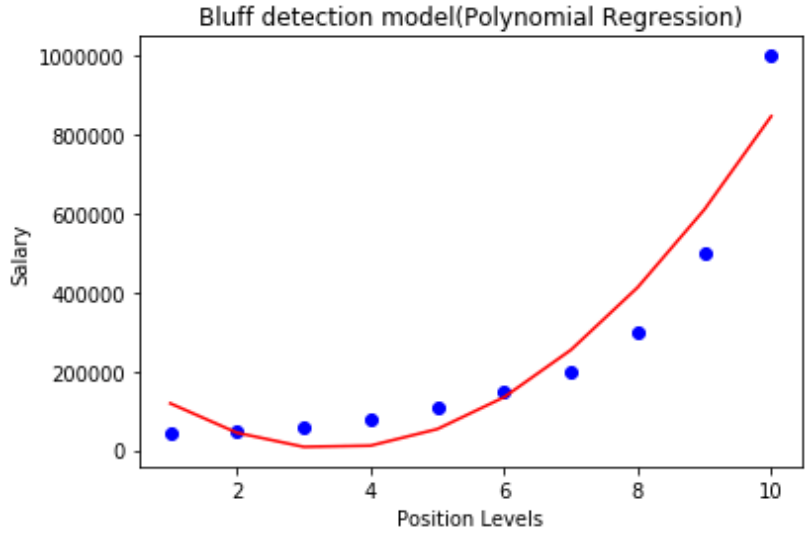
Here we will visualize the result of Polynomial regression model, code for which is little different from the above model.

Code for this is given below:

1. #Visulaizing the result **for** Polynomial Regression
2. mtp.scatter(x,y,color="blue")
3. mtp.plot(x, lin_reg_2.predict(poly_regs.fit_transform(x)), color="red")
4. mtp.title("Bluff detection model(Polynomial Regression)")
5. mtp.xlabel("Position Levels")
6. mtp.ylabel("Salary")
7. mtp.show()

In the above code, we have taken lin_reg_2.predict(poly_regs.fit_transform(x), instead of x_poly, because we want a Linear regressor object to predict the polynomial features matrix.

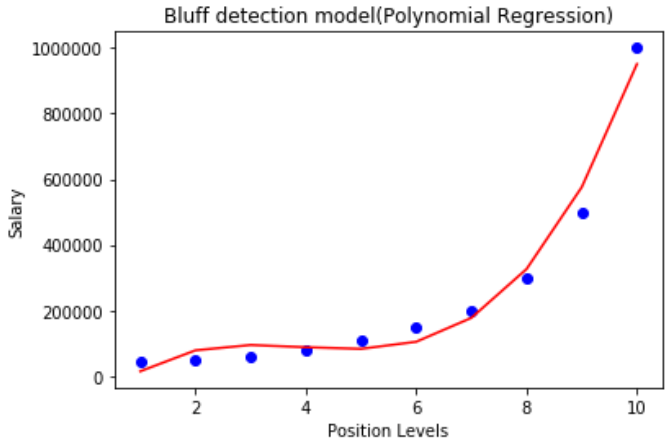
Output:



As we can see in the above output image, the predictions are close to the real values. The above plot will vary as we will change the degree.

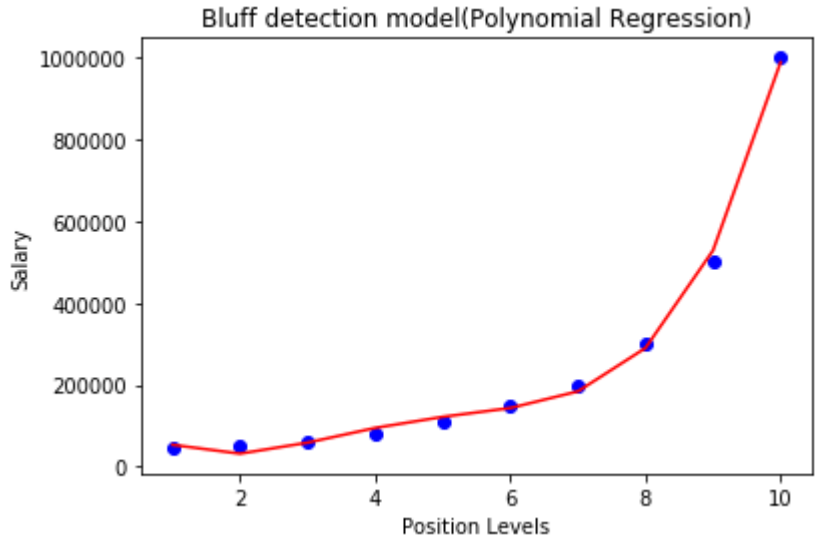
For degree= 3:

If we change the degree=3, then we will give a more accurate plot, as shown in the below image.



SO as we can see here in the above output image, the predicted salary for level 6.5 is near to 170K$-190k$, which seems that future employee is saying the truth about his salary.

Degree= 4: Let's again change the degree to 4, and now will get the most accurate plot. Hence we can get more accurate results by increasing the degree of Polynomial.



Predicting the final result with the Linear Regression model:

Now, we will predict the final output using the Linear regression model to see whether an employee is saying truth or bluff. So, for this, we will use the **predict()** method and will pass the value 6.5. Below is the code for it:

```
1. lin_pred = lin_regs.predict([[6.5]])
2. print(lin_pred)
```

Output:

```
[330378.78787879]
```

Predicting the final result with the Polynomial Regression model:

Now, we will predict the final output using the Polynomial Regression model to compare with Linear model. Below is the code for it:

```
1. poly_pred = lin_reg_2.predict(poly_regs.fit_transform([[6.5]]))
2. print(poly_pred)
```

Output:

```
[158862.45265153]
```

As we can see, the predicted output for the Polynomial Regression is [158862.45265153], which is much closer to real value hence, we can say that future employee is saying true.

Decision Tree:

A decision tree in machine learning is a versatile, interpretable algorithm used for predictive modelling. It structures decisions based on input data, making it suitable for both

classification and regression tasks. This article delves into the components, terminologies, construction, and advantages of decision trees, exploring their applications and learning algorithms.

Decision Tree in Machine Learning

A decision tree is a type of supervised learning algorithm that is commonly used in machine learning to model and predict outcomes based on input data. It is a tree-like structure where each internal node tests on attribute, each branch corresponds to attribute value and each leaf node represents the final decision or prediction. The decision tree algorithm falls under the category of supervised learning. They can be used to solve both **regression** and **classification problems**.

Decision Tree Terminologies

There are specialized terms associated with decision trees that denote various components and facets of the tree structure and decision-making procedure. :

- **Root Node:** A decision tree's root node, which represents the original choice or feature from which the tree branches, is the highest node.
- **Internal Nodes (Decision Nodes)**: Nodes in the tree whose choices are determined by the values of particular attributes. There are branches on these nodes that go to other nodes.
- **Leaf Nodes (Terminal Nodes)**: The branches' termini, when choices or forecasts are decided upon. There are no more branches on leaf nodes.
- **Branches (Edges)**: Links between nodes that show how decisions are made in response to particular circumstances.
- **Splitting**: The process of dividing a node into two or more sub-nodes based on a decision criterion. It involves selecting a feature and a threshold to create subsets of data.
- **Parent Node**: A node that is split into child nodes. The original node from which a split originates.
- **Child Node**: Nodes created as a result of a split from a parent node.
- **Decision Criterion**: The rule or condition used to determine how the data should be split at a decision node. It involves comparing feature values against a threshold.
- **Pruning**: The process of removing branches or nodes from a decision tree to improve its generalization and prevent overfitting.

Understanding these terminologies is crucial for interpreting and working with decision trees in machine learning applications.

How Decision Tree is formed?

The process of forming a decision tree involves recursively partitioning the data based on the values of different attributes. The algorithm selects the best attribute to split the data at each internal node, based on certain criteria such as information gain or Gini impurity. This splitting process continues until a stopping criterion is met, such as reaching a maximum depth or having a minimum number of instances in a leaf node.

Why Decision Tree?

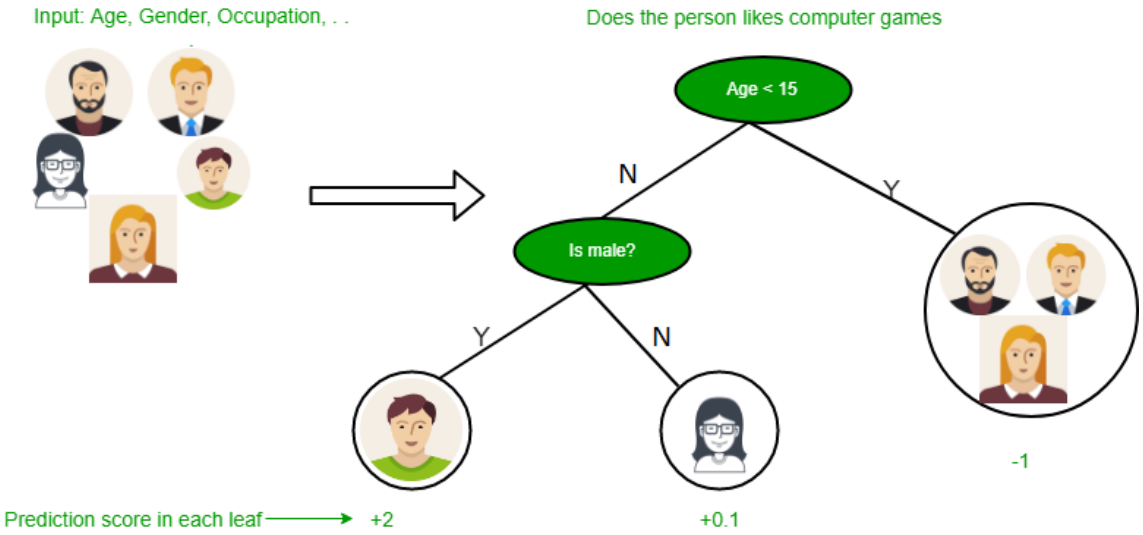
Decision trees are widely used in machine learning for a number of reasons:

- Decision trees are so versatile in simulating intricate decision-making processes, because of their interpretability and versatility.
- Their portrayal of complex choice scenarios that take into account a variety of causes and outcomes is made possible by their hierarchical structure.
- They provide comprehensible insights into the decision logic, decision trees are especially helpful for tasks involving categorization and regression.

- They are proficient with both numerical and categorical data, and they can easily adapt to a variety of datasets thanks to their autonomous feature selection capability.
- Decision trees also provide simple visualization, which helps to comprehend and elucidate the underlying decision processes in a model.

Decision Tree Approach

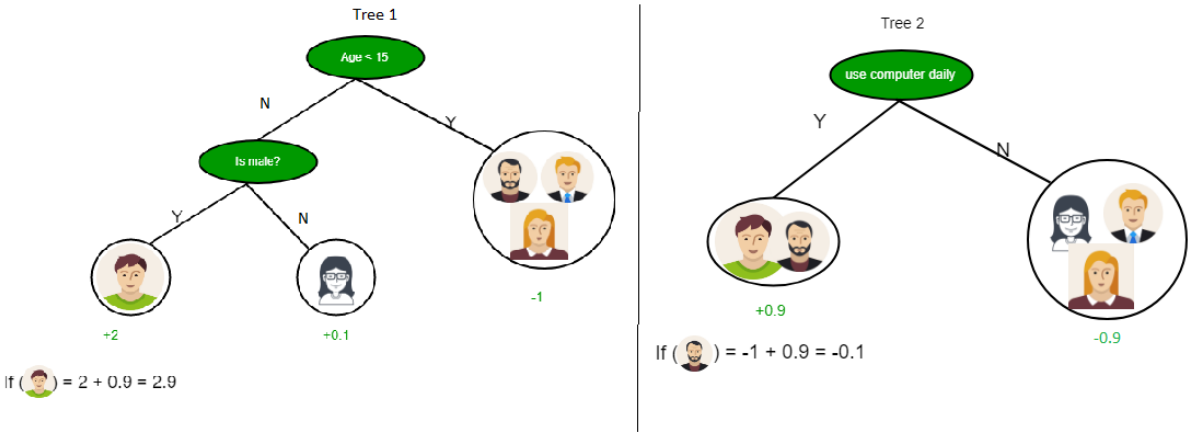
Decision tree uses the tree representation to solve the problem in which each leaf node corresponds to a class label and attributes are represented on the internal node of the tree. We can represent any boolean function on discrete attributes using the decision tree.



Below are some assumptions that we made while using the decision tree:

At the beginning, we consider the whole training set as the root.

- Feature values are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.
- On the basis of attribute values, records are distributed recursively.
- We use statistical methods for ordering attributes as root or the internal node.



As you can see from the above image the Decision Tree works on the Sum of Product form which is also known as *Disjunctive Normal Form*. In the above image, we are predicting the use of computer in the daily life of people. In the Decision Tree, the major challenge is the identification of the attribute for the root node at each level. This process is known as attribute selection. We have two popular attribute selection measures:

1. Information Gain
2. Gini Index

1. Information Gain:

When we use a node in a decision tree to partition the training instances into smaller subsets the entropy changes. Information gain is a measure of this change in entropy.

- Suppose S is a set of instances,
- A is an attribute
- Sv is the subset of S
- *v* represents an individual value that the attribute *A* can take and Values (A) is the set of all possible values of A, then

$$Gain(S, A) = Entropy(S) - \sum_{v}^{A} \frac{|S_v|}{|S|}.Entropy(S_v)$$

Entropy: is the measure of uncertainty of a random variable, it characterizes the impurity of an arbitrary collection of examples. The higher the entropy more the information content.

Suppose S is a set of instances, A is an attribute, Sv is the subset of S with A = v, and Values (A) is the set of all possible values of A, then

$$Gain(S, A) = Entropy(S) - \sum_{v \epsilon Values(A)} \frac{|S_v|}{|S|}.Entropy(S_v)$$

Example:

For the set X = {a,a,a,b,b,b,b,b}
Total instances: 8
Instances of b: 5
Instances of a: 3

$$\begin{aligned}\text{Entropy } H(X) &= \left[\left(\frac{3}{8}\right)\log_2\frac{3}{8} + \left(\frac{5}{8}\right)\log_2\frac{5}{8}\right] \\ &= -[0.375(-1.415) + 0.625(-0.678)] \\ &= -(-0.53 - 0.424) \\ &= 0.954\end{aligned}$$

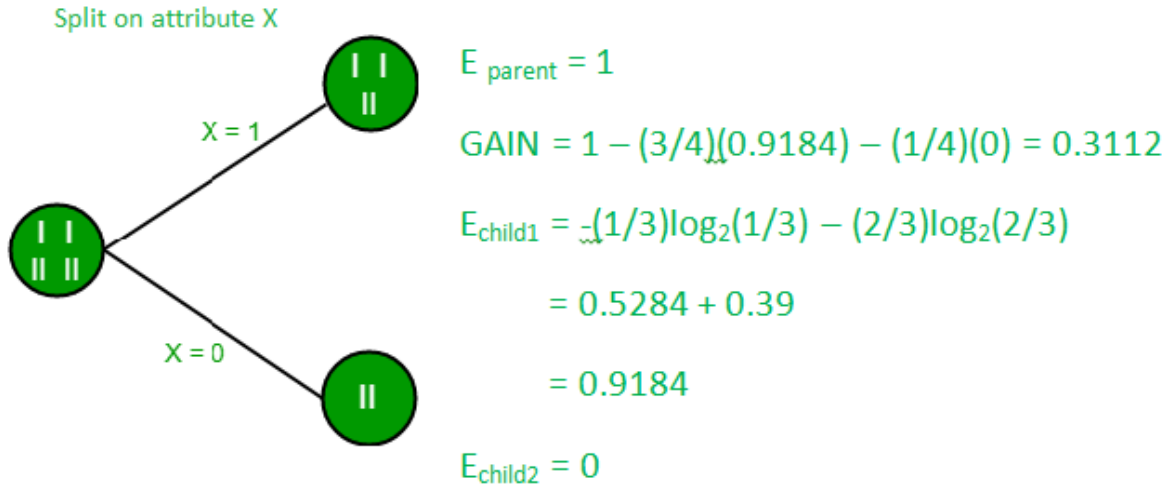
Building Decision Tree using Information Gain The essentials:

- Start with all training instances associated with the root node
- Use info gain to choose which attribute to label each node with
- *Note:* No root-to-leaf path should contain the same discrete attribute twice
- Recursively construct each subtree on the subset of training instances that would be classified down that path in the tree.
- If all positive or all negative training instances remain, the label that node “yes” or “no” accordingly
- If no attributes remain, label with a majority vote of training instances left at that node
- If no instances remain, label with a majority vote of the parent’s training instances.

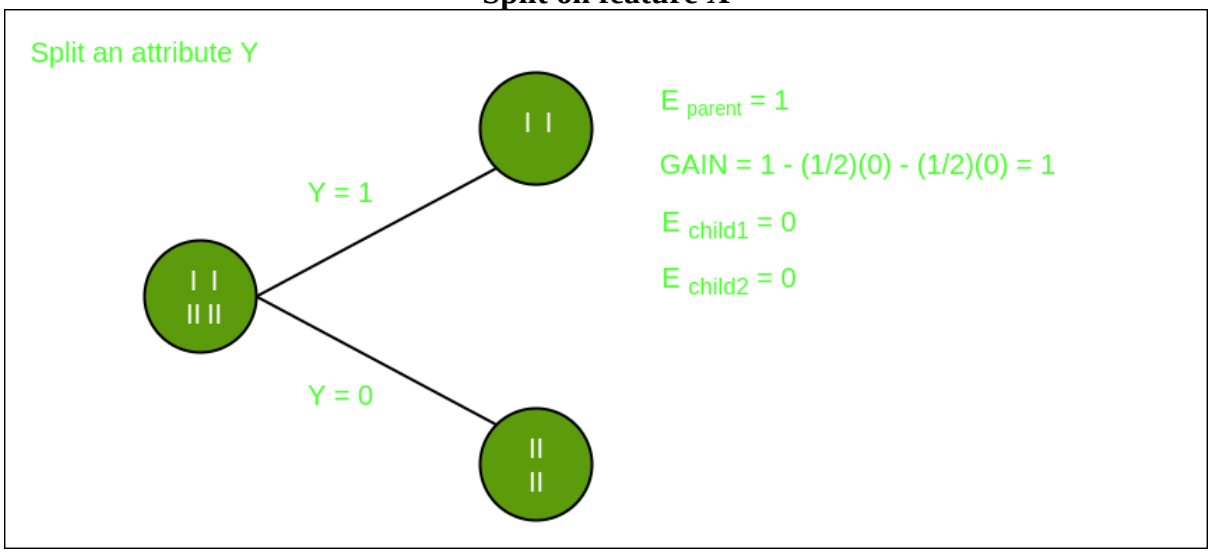
Example: Now, let us draw a Decision Tree for the following data using Information gain. **Training set: 3 features and 2 classes**

X	Y	Z	C
1	1	1	I
1	1	0	I
0	0	1	II
1	0	0	II

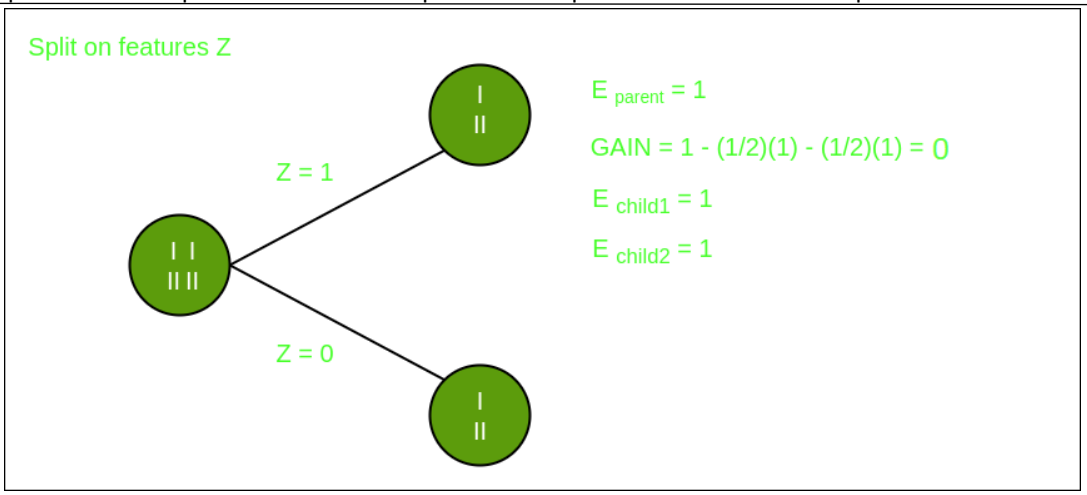
Here, we have 3 features and 2 output classes. To build a decision tree using Information gain. We will take each of the features and calculate the information for each feature.



Split on feature X

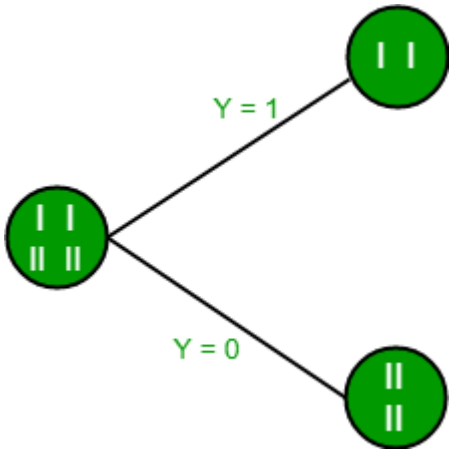


Split on feature Y



Split on feature Z

From the above images, we can see that the information gain is maximum when we make a split on feature Y. So, for the root node best-suited feature is feature Y. Now we can see that while splitting the dataset by feature Y, the child contains a pure subset of the target variable. So we don't need to further split the dataset. The final tree for the above dataset would look like this:



2. Gini Index

- Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified.
- It means an attribute with a lower Gini index should be preferred.
- Sklearn supports "Gini" criteria for Gini Index and by default, it takes "gini" value.
- The Formula for the calculation of the Gini Index is given below.

The Gini Index is a measure of the inequality or impurity of a distribution, commonly used in decision trees and other machine learning algorithms. It ranges from 0 to 1, where 0 represents perfect equality (all values are the same) and 1 represents perfect inequality (all values are different).

Some additional features and characteristics of the Gini Index are:

- It is calculated by summing the squared probabilities of each outcome in a distribution and subtracting the result from 1.
- A lower Gini Index indicates a more homogeneous or pure distribution, while a higher Gini Index indicates a more heterogeneous or impure distribution.

- In decision trees, the Gini Index is used to evaluate the quality of a split by measuring the difference between the impurity of the parent node and the weighted impurity of the child nodes.
- Compared to other impurity measures like entropy, the Gini Index is faster to compute and more sensitive to changes in class probabilities.
- One disadvantage of the Gini Index is that it tends to favor splits that create equally sized child nodes, even if they are not optimal for classification accuracy.
- In practice, the choice between using the Gini Index or other impurity measures depends on the specific problem and dataset, and often requires experimentation and tuning.

Example of a Decision Tree Algorithm

Forecasting Activities Using Weather Information

- **Root node**: Whole dataset
- **Attribute** : "Outlook" (sunny, cloudy, rainy).
- **Subsets**: Overcast, Rainy, and Sunny.
- **Recursive Splitting**: Divide the sunny subset even more according to humidity, for example.
- **Leaf Nodes**: Activities include "swimming," "hiking," and "staying inside."

Beginning with the entire dataset as the root node of the decision tree:

- Determine the best attribute to split the dataset based on information gain, which is calculated by the formula: Information gain = Entropy(parent) – [Weighted average] * Entropy(children), where entropy is a measure of impurity or disorder of a set of examples, and the weighted average is based on the number of examples in each child node.
- Create a new internal node that corresponds to the best attribute and connects it to the root node. For example, if the best attribute is "outlook" (which can have values "sunny", "overcast", or "rainy"), we create a new node labeled "outlook" and connect it to the root node.
- Partition the dataset into subsets based on the values of the best attribute. For example, we create three subsets: one for instances where the outlook is "sunny", one for instances where the outlook is "overcast", and one for instances where the outlook is "rainy".
- Recursively repeat steps 1-4 for each subset until all instances in a given subset belong to the same class or no further splitting is possible. For example, if the subset of instances where the outlook is "overcast" contains only instances where the activity is "hiking", we assign a leaf node labeled "hiking" to this subset. If the subset of instances where the outlook is "sunny" is further split based on the humidity attribute, we repeat steps 2-4 for this subset.
- Assign a leaf node to each subset that contains instances that belong to the same class. For example, if the subset of instances where the outlook is "rainy" contains only instances where the activity is "stay inside", we assign a leaf node labeled "stay inside" to this subset.
- Make predictions based on the decision tree by traversing it from the root node to a leaf node that corresponds to the instance being classified. For example, if the outlook is "sunny" and the humidity is "high", we traverse the decision tree by following the "sunny" branch and then the "high humidity" branch, and we end up at a leaf node labeled "swimming", which is our predicted activity.

Advantages of Decision Tree

- Easy to understand and interpret, making them accessible to non-experts.

- Handle both numerical and categorical data without requiring extensive preprocessing.
- Provides insights into feature importance for decision-making.
- Handle missing values and outliers without significant impact.
- Applicable to both classification and regression tasks.

Disadvantages of Decision Tree

- **Disadvantages** include the potential for overfitting
- Sensitivity to small changes in data, limited generalization if training data is not representative
- Potential bias in the presence of imbalanced data.

Conclusion

Decision trees, a key tool in machine learning, model and predict outcomes based on input data through a tree-like structure. They offer interpretability, versatility, and simple visualization, making them valuable for both categorization and regression tasks. While decision trees have advantages like ease of understanding, they may face challenges such as overfitting. Understanding their terminologies and formation process is essential for effective application in diverse scenarios.

ARIMA :

A Sequence of recording a metric over the constant time intervals is known as **Time Series**.

Based on the frequency, a Time Series can be classified into the following categories:

1. Yearly (For example, Annual Budget)
2. Quarterly (For example, Expenses)
3. Monthly (For example, Air Traffic)
4. Weekly (For example, Sale Quantity)
5. Daily (For instance, Weather)
6. Hourly (For example, Stocks Price)
7. Minutes wise (For example, Inbound Calls in a Call Centre)
8. Seconds wise (For example, Web Traffic)

Once we are done with Time Series Analysis, we have to forecast it in order to predict the future values that the series will be going to take.

However, what is the need for forecasting?

Since forecasting a Time Series, such as Sales and Demand, is often of incredible commercial value, which increases the need for forecasting.

Time Series Forecasting is generally used in many manufacturing companies as it drives the primary business planning, procurement, and production activities. Any forecasts' errors will undulate throughout the chain of the supply or any business framework, for that stuff. Thus, it is significant in order to get accurate predictions saving the costs, and is critical to success.

The Concepts and Techniques behind Time Series forecasting can also be applied in any business, including manufacturing.

The Time Series forecasting can be broadly classified into two categories:

1. **Univariate Time Series Forecasting:** The Univariate Time Series Forecasting is a forecasting of time series where we utilize the former values of the time series only in order to guess the forthcoming values.
2. **Multi-Variate Time Series Forecasting:** The Multi-Variate Time Series Forecasting is a forecasting of time series where we utilize the predictors other than the series, also known as exogenous variables, in order to forecast.

In the following tutorial, we will understand the specific type of method known as **ARIMA modeling**.

Auto Regressive Integrated Moving Average, abbreviated as **ARIMA**, is an Algorithm for forecasting that is centered on the concept that the data in the previous values of the time series can alone be utilized in order to predict the future values.

Let us understand the ARIMA Models in detail.

An Introduction to ARIMA Models

ARIMA, abbreviated for **'Auto Regressive Integrated Moving Average'**, is a class of models that 'demonstrates' a given time series based on its previous values: its lags and the lagged errors in forecasting, so that equation can be utilized in order to forecast future values.

We can model any Time Series that are non-seasons exhibiting patterns and not a random white noise with **ARIMA models**.

There are three terms characterizing An ARIMA model:

p, q, and d

where,

- **p** = the order of the AR term
- **q** = the order of the MA term
- **d** = the number of differences required to make the time series stationary

If a Time Series has seasonal patterns, we have to insert seasonal periods, and it becomes **SARIMA**, short for **'Seasonal ARIMA'**.

Now, before understanding "**the order of AR term**", let us discuss 'd' term.

What are 'p', 'q', and 'd' in the ARIMA model?

The primary step is **to make the time series stationary** in order to build an ARIMA model. This is because the term **'Auto Regressive'** in ARIMA implies a Linear Regression Model using its lags as predictors. And as we already know, Linear Regression Models work well for independent and non-correlated predictors.

In order to make a series stationary, we will utilize the most common approach that is to subtract the past value from the present value. Sometimes, depending on the series complexity, multiple subtractions may be required.

Therefore, the value of d is the minimum number of subtractions required to make the series stationary. And if the time series is already stationary, thus d becomes 0.

Now, let us understand the terms '**p**' and '**q**'.

The '**p**' is the order of the **'AR' (Auto-Regressive) term**, which means that the number of lags of Y to be utilized as predictors. At the same time, '**q**' is the order of the **'MA' (Moving Average) term**, which means that the number of lagged forecast errors should be used in the ARIMA Model.

Now, let us understand what 'AR' and 'MA' models are in detail.

Understanding Auto-Regressive (AR) and Moving Average (MA) Models

In the following section, we will discuss the AR and MA models and the actual mathematical formula for these models.

A Pure AR (Auto-Regressive only) Model is a model which relies only on its own lags. Hence, we can also conclude that it is a function of the 'lags of Y_t'

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \dots + \beta_p Y_{t-p} + \epsilon_t$$

where, Y_{t-1} is the lag1 of the series. β_1 is the coefficient of lag1 and is the term of intercept that is calculated by the model.

Similarly, a Pure MA (Moving Average only) model is a model where Y_t relies only on the lagged predicted errors.

$$Y_t = \alpha + \epsilon_1 + \emptyset_1 \epsilon_{t-1} + \emptyset_2 \epsilon_{t-2} + \dots + \emptyset_q \epsilon_{t-q}$$

Where, the error terms are the AR models errors of the corresponding lags. The errors ϵ_t and ϵ_{t-1} are the errors from the equations given below:

$$Y_t = \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \dots + \beta_0 Y_0 + \epsilon_t$$

$$Y_{t-1} = \beta_2 Y_{t-2} + \beta_3 Y_{t-3} + \dots + \beta_0 Y_0 + \epsilon_{t-1}$$

Thus, we have concluded Auto-Regressive (AR) and Moving Average (MA) models, respectively.

Let us now understand the equation of an ARIMA Model.

An ARIMA model is a model where the series of time was subtracted at least once in order to make it stationary, and we combine the Auto-Regressive (AR) and the Moving Average (MA) terms. Hence, we got the following equation:

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \dots + \beta_p Y_{t-p} \epsilon_t + \emptyset_1 \epsilon_{t-1} + \emptyset_2 \epsilon_{t-2} + \dots + \emptyset_q \epsilon_{t-q}$$

ARIMA Model in words:

Forecasted Y_t = Constant + Linear Combination Lags of Y (up to p lags) + Linear Combination of Lagged Predicted Errors (up to q lags)

Thus, the objective of this model is to find the values of **p, q**, and **d**. However, how can we find one?

Let us begin with finding the '**d**' in the ARIMA Model.

Finding the order of differencing 'd' in the ARIMA Model

The primary purpose of differencing in the ARIMA model is to make the Time Series stationary.

However, we have to take care of not over-differencing the series as an over-differenced series may also be stationary, which will affect the parameter of the model later.

Now, let us understand the appropriate differencing order.

The most appropriate differencing order is the minimum differencing needed in order to achieve an almost stationary series roaming around a defined mean and the ACF plat reaching Zero relatively faster.

In case the autocorrelations are positive for multiple lags (generally, ten or more), the series requires further differencing. In contrast, if lag 1 autocorrelated itself pretty negatively, then the series is possibly over-differenced.

In cases where we cannot actually decide between two differencing orders, then we have to choose the order providing the minor standard deviation in the differenced series.

Let us consider an example to check if the series is stationary. We will use the **Augmented Dickey-Fuller Test (adfuller())** from the **statsmodels** package of Python Programming Language.

Example:

```
from statsmodels.tsa.stattools import adfuller
from numpy import log
import pandas as pd

mydata = pd.read_csv('mydataset.csv', names = ['value'], header = 0)

res = adfuller( mydata.value.dropna())
print('Augmented Dickey-Fuller Statistic: %f' % res[0])
print('p-value: %f' % res[1])
```

Output:

```
Augmented Dickey-Fuller Statistic: -2.464240
p-value: 0.124419
```

Explanation:

In the above example, we have imported the **adfuller** module along with the **numpy**'s log module and **pandas**. We have then used the **pandas** library to read the CSV file. We have then used the **adfuller** method and printed the values to the user.

It is necessary to check whether the series is stationary or not. If not, we have to use difference; else, **d** becomes **zero**.

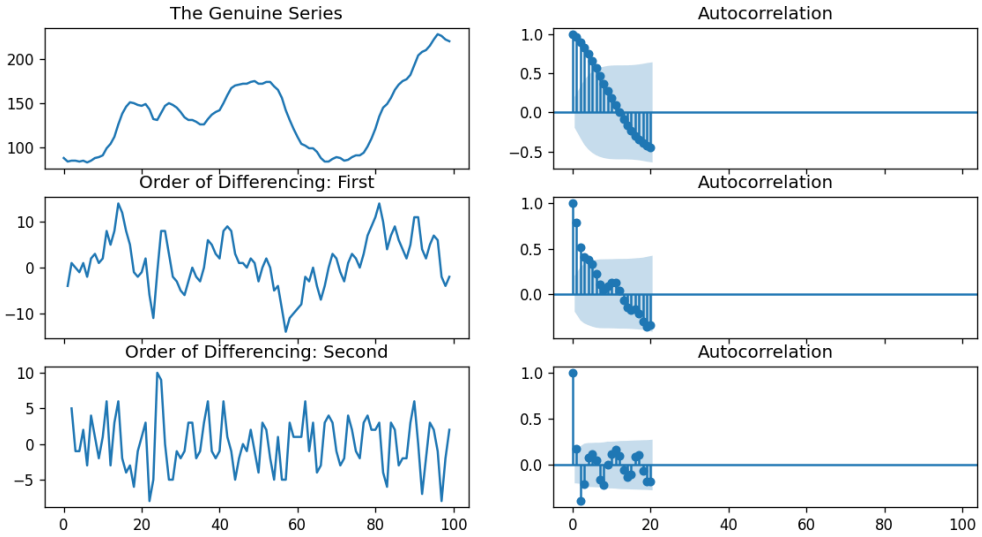
The **Augmented Dickey-Fuller (ADF)** test's null hypothesis is that the time series is not stationary. Thus, if the ADF test's p-value is less than the significance level (0.05), then we will reject the null hypothesis and infer that the time series is definitely stationary. As we can observe, the p-value is more significant than the level of significance. Therefore, we can difference the series and check the plot of autocorrelation as shown below.

Example:

```
import numpy as np, pandas as pd
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt

plt.rcParams.update({'figure.figsize' : (9,7), 'figure.dpi' : 120})

# Importing data
df = pd.read_csv('mydataset.csv', names = ['value'], header = 0)

```

```
# The Genuine Series
fig, axes = plt.subplots(3, 2, sharex = True)
axes[0, 0].plot(df.value); axes[0, 0].set_title('The Genuine Series')
plot_acf(df.value, ax = axes[0, 1])

# Order of Differencing: First
axes[1, 0].plot(df.value.diff()); axes[1, 0].set_title('Order of Differencing: First')
plot_acf(df.value.diff().dropna(), ax = axes[1, 1])

# Order of Differencing: Second
axes[2, 0].plot(df.value.diff().diff()); axes[2, 0].set_title('Order of Differencing: Second')
plot_acf(df.value.diff().diff().dropna(), ax = axes[2, 1])

plt.show()
```

Output:



Explanation:

In the above example, we have imported the required libraries and modules. We have then imported the data and plot different graphs. We have plotted the original series graph, first-order differencing, and second-order differencing along with their autocorrelation graphs. As we can observe, the time series has reached stationarity with two differencing orders. However, when we have a look at the plot of autocorrelation for the Second order of differencing, the lag going into the far negative zone pretty faster, indicating the series might have been over differenced.

Hence, we will tentatively be fixing the differencing order because the series is not properly stationary, or we can say that the series has weak stationarity.

This can be done as shown below.

Example:

```
1. from pmdarima.arima.utils import ndiffs
2. import pandas as pd
3.
4. df = pd.read_csv('mydataset.csv', names = ['value'], header = 0)
5. X = df.value
6.
7. # Augmented Dickey Fuller Test
8. adftest = ndiffs(X, test = 'adf')
9.
10. # KPSS Test
11. kpsstest = ndiffs(X, test = 'kpss')
12.
13. # PP Test
14. pptest = ndiffs(X, test = 'pp')
15.
16. print("ADF Test =", adftest)
17. print("KPSS Test =", kpsstest)
18. print("PP Test =", pptest)
```

Output:

```
ADF Test = 2
KPSS Test = 0
PP Test = 2
```

Explanation:

In the above example, we have imported the **ndiffs** method of the **pmdarima** module. We have then imported the dataset and defined 'X' as the object containing the values from the dataset. We used the **ndiffs** method to perform ADF, KPSS, and PP Tests and printed their results to the users.

Finding the order of the Auto-Regressive (AR) term (p)

In the following section, we will discuss the steps to check whether the model requires any **Auto-Regressive (AR) terms**. The number of AR terms needed can be found by studying the **Partial Autocorrelation (PACF) plot**.

We can consider **Partial Autocorrelation** as the correlation between the series and its lag once we exclude the contributions from the intermediate lags. Thus, **PACF** tends to convey the pure correlation between the series and its lag. Hence, we can identify whether that lag is required in the Auto-Regressive (AR) term or not.

Partial Autocorrelation of lag(k) of a series is the coefficient of that lag in the Auto-Regression Equation of Y.

$$Y_t = \alpha_0 + \alpha_1 Y_{t-1} + \alpha_2 Y_{t-2} + \alpha_3 Y_{t-3}$$

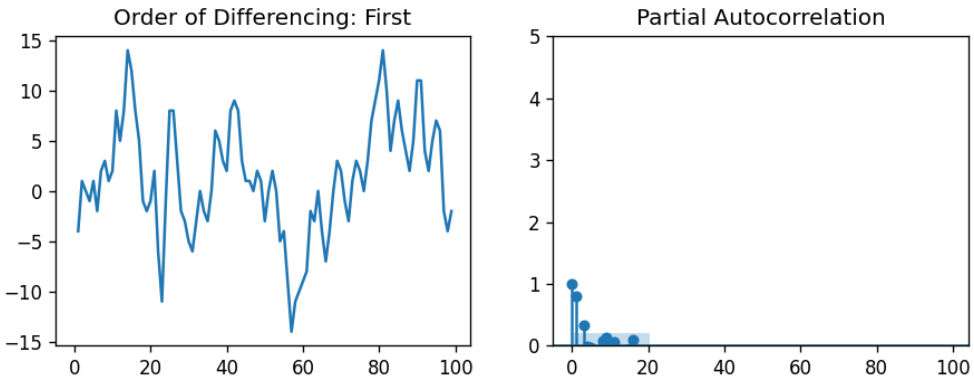
Now, let us understand how to find the number of AR terms?

As we know, any Autocorrelation in a stationary series can be rectified by inserting enough AR terms. Thus, we can initially take the order of the Auto-Regressive (AR) term equivalent to as many lags that cross the limit of significance in the PACF Plot.

Example:

```
import numpy as np, pandas as pd
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt

plt.rcParams.update({'figure.figsize':(9,3), 'figure.dpi':120})

# importing data
df = pd.read_csv('mydataset.csv', names = ['value'], header = 0)

fig, axes = plt.subplots(1, 2, sharex = True)
axes[0].plot(df.value.diff()); axes[0].set_title('Order of Differencing: First')
axes[1].set(ylim = (0,5))
plot_pacf(df.value.diff().dropna(), ax = axes[1])

plt.show()
```

Output:



Explanation:

In the above example, we have imported the required libraries, modules, and datasets. We have then plotted the graphs to represent the First Order Differencing and its partial autocorrelation.

As a result, we can observe that the PACF lag 1 is pretty significant above the line of significance. Lag 2 also appears to be substantial, entirely maintaining to cross the limit of significance (blue region). However, we will be conservative and fix the p as one tentatively.

Finding the Order of the Moving Average (MA) term (q)

Similar to what we have looked at earlier at the PACF plot for the number of Auto-Regressive (AR) Terms, we can use the ACF plot to find the number of Moving Average (MA) Terms. A Moving Average (MA) term is, theoretically, the lagged forecast's error.

The ACF plot expresses the number of Moving Average (MA) terms needed to remove the autocorrelation in the stationary series.

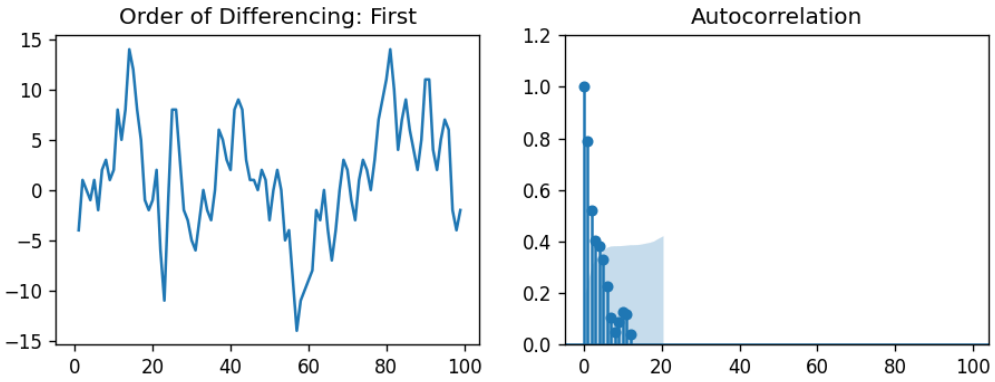
Let us consider the following example to understanding the autocorrelation plot of the differenced series.

Example:

```
1. import numpy as np, pandas as pd
2. from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
3. import matplotlib.pyplot as plt
4.
5. plt.rcParams.update({'figure.figsize' : (9,3), 'figure.dpi' : 120})
6.
7. # importing data
8. mydata = pd.read_csv('mydataset.csv', names = ['value'], header = 0)
9.
10. fig, axes = plt.subplots(1, 2, sharex = True)
11. axes[0].plot(mydata.value.diff()); axes[0].set_title('Order of Differencing: First')
```

```
12. axes[1].set(ylim = (0, 1.2))
13. plot_acf(mydata.value.diff().dropna(), ax = axes[1])
14.
15. plt.show()
```

Output:



Explanation:

In the above example, we have imported the required libraries, modules, and datasets. We have then plotted the graphs to represent the First Order Differencing and its Autocorrelation. As a result, we can observe that some lags are pretty above the line of significance. So, let us fix q as 2, tentatively. We can also use the simpler model in case of any doubt that adequately demonstrates the Y.

Handling the A Slightly Under or Over-Differenced Time Series

Sometimes, a situation may occur where the series is slightly under-differenced, and differencing it one time extra makes the series somewhat over-differenced. In such cases, we have to add one or multiple additional Auto-Regressive (AR) Terms for the slightly under-differenced Time Series and add an extra Moving Average (MA) Term for the slightly over-differenced Time Series.

Once we have discussed most of the topics, let us begin creating an ARIMA Model for Time Series Forecasting.

Building the ARIMA Model

Once we have determined the values of p, q, and d, we will try creating the ARIMA model. The implementation of the **ARIMA()** module is shown below:

Example:

```
1. import numpy as np, pandas as pd
2. from statsmodels.tsa.arima_model import ARIMA
```

```
3.
4.  # importing data
5.  mydata = pd.read_csv('mydataset.csv', names = ['value'], header = 0)
6.
7.  # Creating ARIMA model
8.  mymodel = ARIMA(mydata.value, order = (1, 1, 2))
9.  modelfit = mymodel.fit(disp = 0)
10. print(modelfit.summary())
```

Output:

```
ARIMA Model Results
==============================================================================
Dep. Variable:               D.value   No. Observations:                   99
Model:                 ARIMA(1, 1, 2)   Log Likelihood                -253.790
Method:                       css-mle   S.D. of innovations              3.119
Date:                Thu, 15 Apr 2021   AIC                            517.579
Time:                        21:10:37   BIC                            530.555
Sample:                             1   HQIC                           522.829

==================================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------
const              1.1202      1.290      0.868      0.385      -1.409       3.649
ar.L1.D.value      0.6351      0.257      2.469      0.014       0.131       1.139
ma.L1.D.value      0.5287      0.355      1.489      0.136      -0.167       1.224
ma.L2.D.value     -0.0010      0.321     -0.003      0.998      -0.631       0.629
                                    Roots
==============================================================================
                  Real          Imaginary           Modulus         Frequency
------------------------------------------------------------------------------
AR.1            1.5746           +0.0000j            1.5746            0.0000
MA.1           -1.8850           +0.0000j            1.8850            0.5000
MA.2          545.5472           +0.0000j          545.5472            0.0000
------------------------------------------------------------------------------
```

Explanation:

In the above example, we have imported the new module called **ARIMA** from the **statsmodels** class and create the ARIMA model of the order 1, 1, and 2. We have then printed the summary of the model to the user. As we can observe, the overview of the model reveals a lot of details. The middle table is the table of coefficients where the **'coef'** values act as the related terms' weights.

We can also notice that the MA2 term's coefficient tends to zero, and the P-Value in the '**P > |z|**' column is exceedingly insignificant. The P-Value should be less than 0.05, ideally for the corresponding X to be significant.

Now, let us try rebuilding the model without the MA2 term.

Example:

```
1. import numpy as np, pandas as pd
2. from statsmodels.tsa.arima_model import ARIMA
3.
4. # importing data
5. mydata = pd.read_csv('mydataset.csv', names = ['value'], header = 0)
6.
7. # Creating ARIMA model
8. mymodel = ARIMA(mydata.value, order = (1, 1, 1))
9. modelfit = mymodel.fit(disp = 0)
10. print(modelfit.summary())
```

Output:

```
ARIMA Model Results
==============================================================================
Dep. Variable:               D.value   No. Observations:                   99
Model:                 ARIMA(1, 1, 1)   Log Likelihood                -253.790
Method:                       css-mle   S.D. of innovations              3.119
Date:                Thu, 15 Apr 2021   AIC                            515.579
Time:                        21:34:00   BIC                            525.960
Sample:                             1   HQIC                           519.779

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.1205      1.286      0.871      0.384      -1.400       3.641
ar.L1.D.value  0.6344      0.087      7.317      0.000       0.464       0.804
ma.L1.D.value  0.5297      0.089      5.932      0.000       0.355       0.705
                                    Roots
==============================================================================
                  Real          Imaginary           Modulus         Frequency
------------------------------------------------------------------------------
AR.1            1.5764           +0.0000j            1.5764            0.0000
MA.1           -1.8879           +0.0000j            1.8879            0.5000
------------------------------------------------------------------------------
```

Explanation:

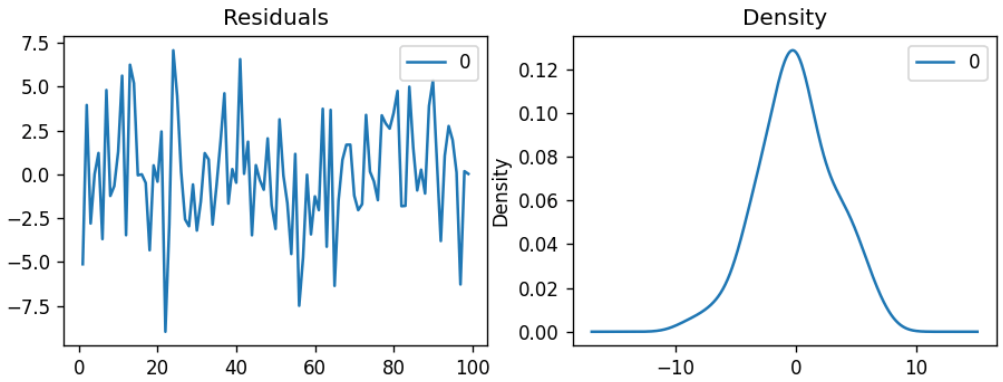
In the above example, we have reduced the AIC of the model, which is actually good. We can also observe that the AR1 and MA1 terms' P-Values' have been improved and are highly significant (<< 0.05).

Now, let us plot the residuals in order to ensure that there are no patterns such as constant mean and variance.

Example:

```
1. import numpy as np, pandas as pd
2. from statsmodels.tsa.arima_model import ARIMA
3. import matplotlib.pyplot as plt
4.
5. plt.rcParams.update({'figure.figsize' : (9,3), 'figure.dpi' : 120})
6.
7. # importing data
8. mydata = pd.read_csv('mydataset.csv', names = ['value'], header = 0)
9.
10. # Creating ARIMA model
11. mymodel = ARIMA(mydata.value, order = (1, 1, 1))
12. modelfit = mymodel.fit(disp = 0)
13.
14. # Plotting Residual Errors
15. myresiduals = pd.DataFrame(modelfit.resid)
16. fig, ax = plt.subplots(1,2)
17. myresiduals.plot(title = "Residuals", ax = ax[0])
18. myresiduals.plot(kind = 'kde', title = 'Density', ax = ax[1])
19. plt.show()
```

Output:



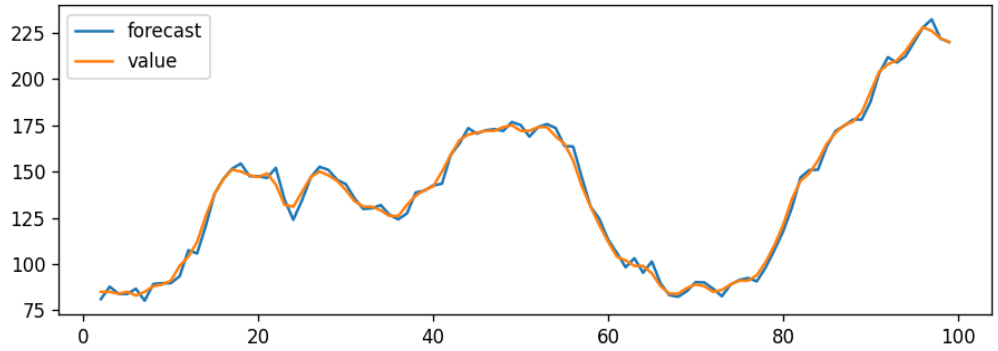
Explanation:

In the above example, we have plotted the residual errors and density graphs. We can observe that the residual errors look fair with around zero mean and uniform variance. Let us plot the graph representing the actuals vs. fitted values with the help of the **plot_predict()** function.

Example:

```
1. import numpy as np, pandas as pd
2. from statsmodels.tsa.arima_model import ARIMA
3. import matplotlib.pyplot as plt
4.
5. plt.rcParams.update({'figure.figsize' : (9,3), 'figure.dpi' : 120})
6.
7. # importing data
8. mydata = pd.read_csv('mydataset.csv', names = ['value'], header = 0)
9.
10. # Creating ARIMA model
11. mymodel = ARIMA(mydata.value, order = (1, 1, 1))
12. modelfit = mymodel.fit(disp = 0)
13.
14. # Actual vs Fitted
15. modelfit.plot_predict(dynamic = False)
16. plt.show()
```

Output:



Explanation:

In the above example, we have now plotted the 'actuals vs fitted' graph and set **dynamic = False**. As a result, the in-sample lagged values are utilized for forecasting.

Thus, the model gets trained up until the past value makes the following forecast. Therefore, it can create the fitted forecast, and actuals appear preciously delicate.