CAPSTONE PROJECT II

CONCRETE MANUFCTURING INDUSTRY ANALYSIS

Name: **Manvi Bajpai**

Course: **Postgraduate Program in Data Science and Analytics**

Batch: **PGA – 30**

Location: **Delhi-NCR**

Date of Submission: **04 September 2023**

ACKNOWLEDGEMENT

I want to sincerely thank everyone who has supported me in finishing my capstone assignment successfully. This project would not have been possible without your help.

I would want to express my gratitude to Imarticus Faculties for their ongoing direction, feedback, and inspiration during the project. Their advice and input have been helpful in forming this project.

Finally, I would like to thank my fellow classmates for their continuous support in getting the assignment finished. Once again, I want to thank you everyone for your encouragement and support.

Sincerely,

Manvi Bajpai

CERTIFICATE OF COMPLETION

I hereby, certify that the Capstone Project II titled “***CONCRETE MANUFCTURING INDUSTRY ANALYSIS”*** under the course ***Postgraduate Program in Data Science and Analytics*** was completed successfully by Manvi Bajpai (PGA: 30).

Mentor’s Name:
Location: **Delhi-NCR**
Date of Submission: **04 September 2023**

Table of Contents

CAPSTONE PROJECT II

WHY STUDY ABOUT CONCRETE MANUFACTURING INDUSTRY?

Cement is the "glue" that creates concrete when combined with sand, gravel, and water. Many parts of modern society are made possible by concrete, a necessary material that is one of the most widely utilized chemicals on the planet. Demand for cement and concrete is expected to rise further as infrastructure construction expands, including the requirement to create a strong foundation to withstand the extreme weather events forecast as a result of climate change.

In various fields of concrete technology, machine learning techniques have been used to characterize materials using image processing techniques, develop concrete mix designs using historical data, and predict the behavior of fresh concrete, hardening, and hardened concrete properties using laboratory data. The approaches have been developed further to analyses the durability and anticipate or identify cracks in the service life of concrete, and they have even been used to predict erosion and chemical attaches.

ML has been extensively used to evaluate, forecast, and model the properties of fresh, hardening, and hardened concrete in the field of concrete technology, which deals with the study of concrete materials characterization, fresh and hardened properties, behaviors, and its applications. Additionally, it has been utilized to optimize the performance of ecologically friendly components that are used to substitute cement using a forecast technique of the effect of adding these resources, such as fly ash. Son et al. employed ML to detect concrete on-site using automated color model photos with neural networks, which helped to measure the construction process and monitor the structural health. It has also been used during and after construction.

UNDERSTANDING THE DATA

PURPOSE OF THE STUDY

For most projects in the construction industry, concrete is most frequently used as the foundation.

Concrete technology provides instruction on all facets of concrete, including mix design, batching, mixing, transporting, placing, consolidating, finishing, and curing. It also covers all elements of concrete, from batching to mixing to transporting, placing, and finishing.

The purpose of this study is to identify the best cement/concrete product available on the market, evaluate its strength using various machine learning modelling techniques.

PROJECT FINDING

The goals of these machine learning models are as follows:

- **Step I:** Examining different machine learning modelling techniques for finding the high performing and effective concrete.

- **Step II:** Understanding how various ingredients used for the formation of concrete contributes to its quality and high performance.

- **Step III:** Finding the top two models that showcases at most accuracy and precision in concrete qualities and it's ingredients.

DATASET

https://www.kaggle.com/datasets/vinayakshanawad/cement-manufacturing-concrete-dataset

ABSTRACT

The dataset has been downloaded from Kaggle's Civil Engineering: Cement Manufacturing Dataset.

GIVEN

- The actual concrete compressive strength (MPa) for a given mixture under a specific age (days) was determined from laboratory. Data is in raw form (not scaled).

- The data has 8 quantitative input variables, and 1 quantitative output variable, and 1030 instances (observations).

ATTRIBUTE INFORMATION:

Concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear function of age and ingredients. These ingredients include cement, blast furnace slag, fly ash, water, superplasticizer, coarse aggregate, and fine aggregate.

- Cement: measured in kg in a m3 mixture
- Blast: measured in kg in a m3 mixture
- Fly ash: measured in kg in a m3 mixture
- Water: measured in kg in a m3 mixture
- Superplasticizer: measured in kg in a m3 mixture
- Coarse Aggregate: measured in kg in a m3 mixture
- Fine Aggregate: measured in kg in a m3 mixture
- Age: day (1~365)
- Concrete compressive strength measured in MPa

DURATION FOR PROJECT COMPLETION

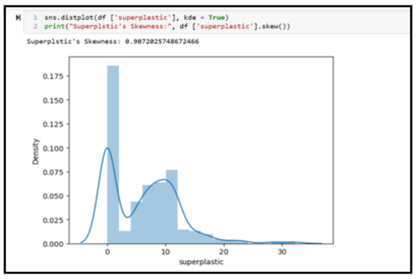
- Duration for coding of the machine learning models: 5-6 hrs.

- Duration for report and power point presentation: 3 hrs.

EXPLORATORY DATA ANALYSIS (EDA)

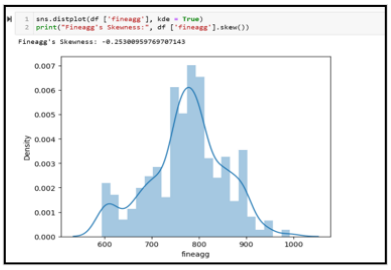
- No. of Independent Variables: 8 {Cement, Slag, Water, Coarseagg, Fineagg, Ash, Superplastic & Age}
- No. of Dependent Variables: 1 {Strength}
- Rows: 1030 & Columns: 9 {All records are numeric}
- Age – Integer Value and the rest of the attributes except age have Floating values.
- Outliers: Found in 5 independent variables {Slag, Water, Superplastic, Fineagg and Age}. Will replace those outliers with median for respective attributes.
- Missing Values: 0, hence no missing value treatment is required.
- None of the independent variables have a strong relationship among themselves. So, there is no need to drop any of the features or to create a composite feature.

UNIVARIATE DATA ANALYSIS



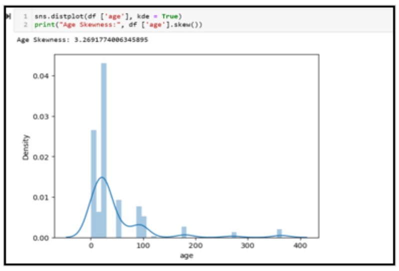
```
sns.distplot(df ['superplastic'], kde = True)
print("Superplstic's Skewness:", df ['superplastic'].skew())
```

```
Superplstic's Skewness: 0.9072025748672466
```



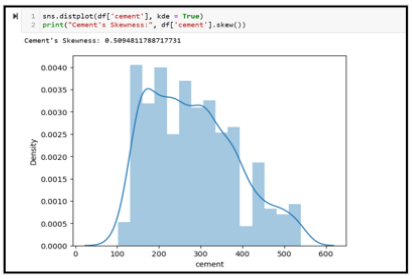
```
sns.distplot(df ['fineagg'], kde = True)
print("Fineagg's Skewness:", df ['fineagg'].skew())
```

```
Fineagg's Skewness: -0.25300959769707143
```



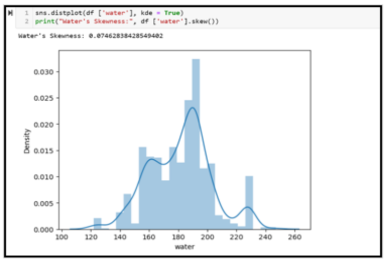
```
sns.distplot(df ['age'], kde = True)
print("Age Skewness:", df ['age'].skew())
```

```
Age Skewness: 3.2691774006345895
```



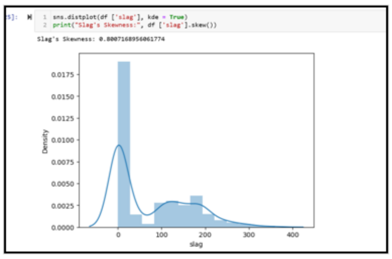
```
sns.distplot(df['cement'], kde = True)
print("Cement's Skewness:", df['cement'].skew())
```

```
Cement's Skewness: 0.5094811788717731
```



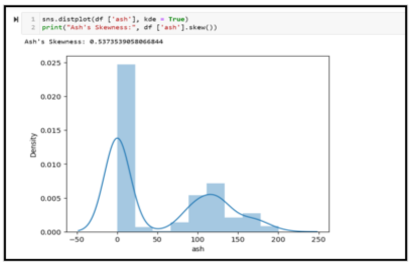
```
sns.distplot(df ['water'], kde = True)
print("Water's Skewness:", df ['water'].skew())
```

```
Water's Skewness: 0.07462838428549402
```



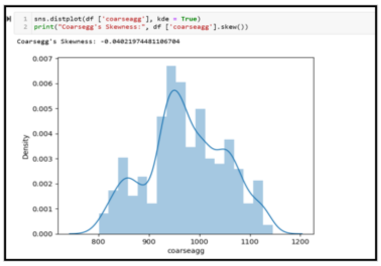
```
sns.distplot(df ['slag'], kde = True)
print("Slag's Skewness:", df ['slag'].skew())
```

```
Slag's Skewness: 0.8007168956061774
```



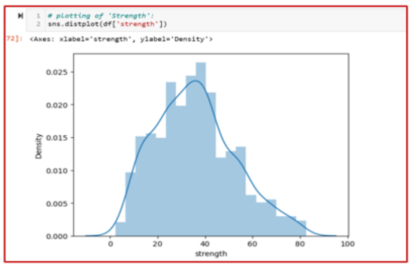
```
sns.distplot(df ['ash'], kde = True)
print("Ash's Skewness:", df ['ash'].skew())
```

```
Ash's Skewness: 0.5373539058066844
```



```
sns.distplot(df ['coarseagg'], kde = True)
print("Coarsegg's Skewness:", df ['coarseagg'].skew())
```

```
Coarsegg's Skewness: -0.04021974481106704
```



```
# plotting of 'Strength':
sns.distplot(df['strength'])
```

```
72]: <Axes: xlabel='strength', ylabel='Density'>
```

OBSERVATIONS

- Cement in comparison to other independent attributes: This attribute has no meaningful relationships to slag, ash, water, superplastic, coarse agg, fine agg, or age. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

- Slag in comparison to other independent attributes: This attribute has no meaningful relationships with ash, water, superplastic, coarse, or fine agg, or age. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

- Ash in comparison to other independent attributes: This attribute has no meaningful relationships to age, water, superplastic, coarse or fine agg, or coarse or fine agg. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

- The independent qualities of water do not significantly relate to the independent attributes of age, superplastic, coarseagg, or fineagg. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

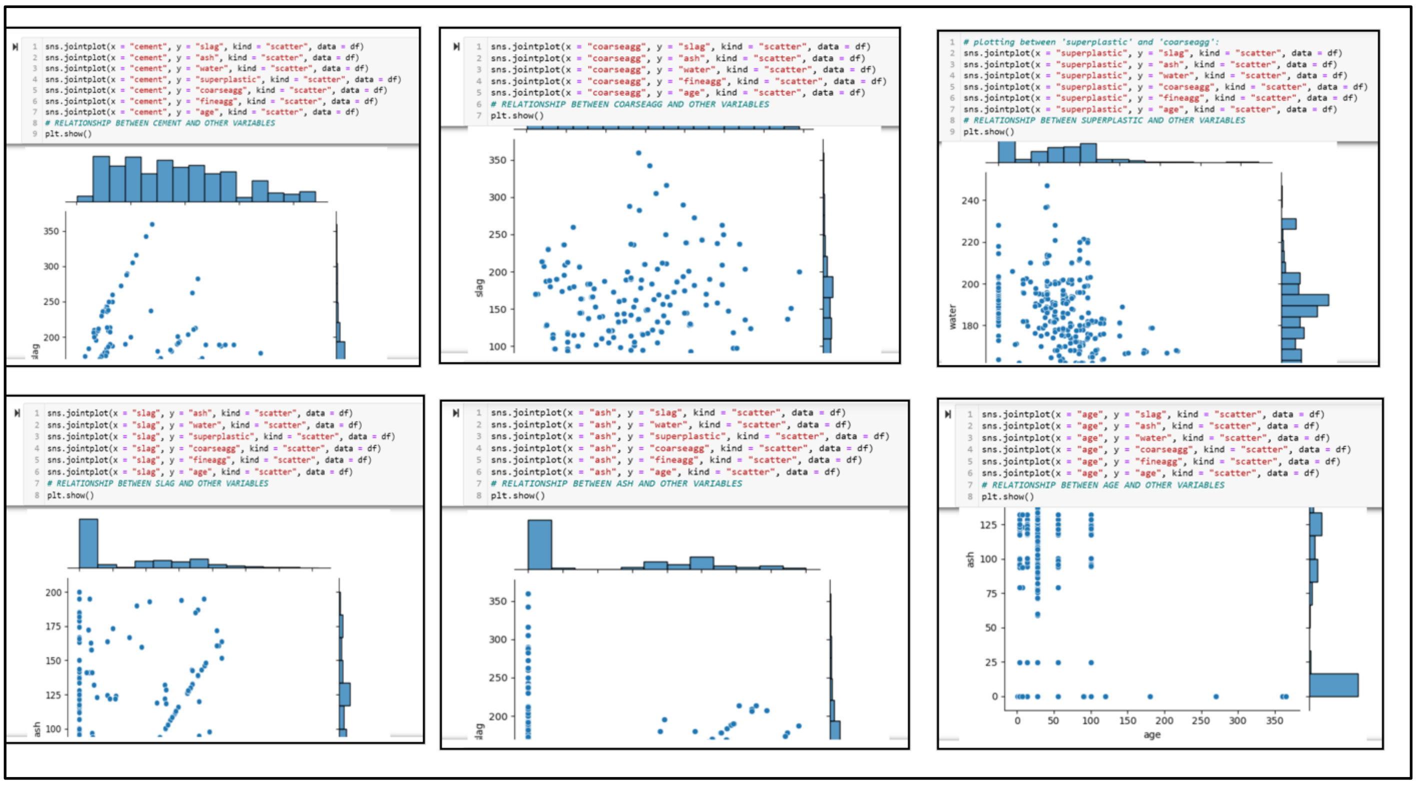
- Superplastic in comparison to other independent attributes: This feature does not significantly relate to coarseagg, fineagg, or age. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

- Coarseagg in comparison to other independent qualities: This attribute does not significantly relate to any other independent attributes. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

- Comparison of independent characteristics and fineagg. It is negatively correlated linearly with water. It does not significantly relate to any other qualities. It expanded almost like a cloud. The r value would have been close to zero if we had calculated it.

- Strength follows normal distribution.

BIVARIATE DATA ANALYSIS



```python
sns.jointplot(x = "cement", y = "slag", kind = "scatter", data = df)
sns.jointplot(x = "cement", y = "ash", kind = "scatter", data = df)
sns.jointplot(x = "cement", y = "water", kind = "scatter", data = df)
sns.jointplot(x = "cement", y = "superplastic", kind = "scatter", data = df)
sns.jointplot(x = "cement", y = "coarseagg", kind = "scatter", data = df)
sns.jointplot(x = "cement", y = "fineagg", kind = "scatter", data = df)
sns.jointplot(x = "cement", y = "age", kind = "scatter", data = df)
# RELATIONSHIP BETWEEN CEMENT AND OTHER VARIABLES
plt.show()
```

```python
sns.jointplot(x = "coarseagg", y = "slag", kind = "scatter", data = df)
sns.jointplot(x = "coarseagg", y = "ash", kind = "scatter", data = df)
sns.jointplot(x = "coarseagg", y = "water", kind = "scatter", data = df)
sns.jointplot(x = "coarseagg", y = "fineagg", kind = "scatter", data = df)
sns.jointplot(x = "coarseagg", y = "age", kind = "scatter", data = df)
# RELATIONSHIP BETWEEN COARSEAGG AND OTHER VARIABLES
plt.show()
```

```python
# plotting between 'superplastic' and 'coarseagg':
sns.jointplot(x = "superplastic", y = "slag", kind = "scatter", data = df)
sns.jointplot(x = "superplastic", y = "ash", kind = "scatter", data = df)
sns.jointplot(x = "superplastic", y = "water", kind = "scatter", data = df)
sns.jointplot(x = "superplastic", y = "coarseagg", kind = "scatter", data = df)
sns.jointplot(x = "superplastic", y = "fineagg", kind = "scatter", data = df)
sns.jointplot(x = "superplastic", y = "age", kind = "scatter", data = df)
# RELATIONSHIP BETWEEN SUPERPLASTIC AND OTHER VARIABLES
plt.show()
```

```python
sns.jointplot(x = "slag", y = "ash", kind = "scatter", data = df)
sns.jointplot(x = "slag", y = "water", kind = "scatter", data = df)
sns.jointplot(x = "slag", y = "superplastic", kind = "scatter", data = df)
sns.jointplot(x = "slag", y = "coarseagg", kind = "scatter", data = df)
sns.jointplot(x = "slag", y = "fineagg", kind = "scatter", data = df)
sns.jointplot(x = "slag", y = "age", kind = "scatter", data = df)
# RELATIONSHIP BETWEEN SLAG AND OTHER VARIABLES
plt.show()
```

```python
sns.jointplot(x = "ash", y = "slag", kind = "scatter", data = df)
sns.jointplot(x = "ash", y = "water", kind = "scatter", data = df)
sns.jointplot(x = "ash", y = "superplastic", kind = "scatter", data = df)
sns.jointplot(x = "ash", y = "coarseagg", kind = "scatter", data = df)
sns.jointplot(x = "ash", y = "fineagg", kind = "scatter", data = df)
sns.jointplot(x = "ash", y = "age", kind = "scatter", data = df)
# RELATIONSHIP BETWEEN ASH AND OTHER VARIABLES
plt.show()
```

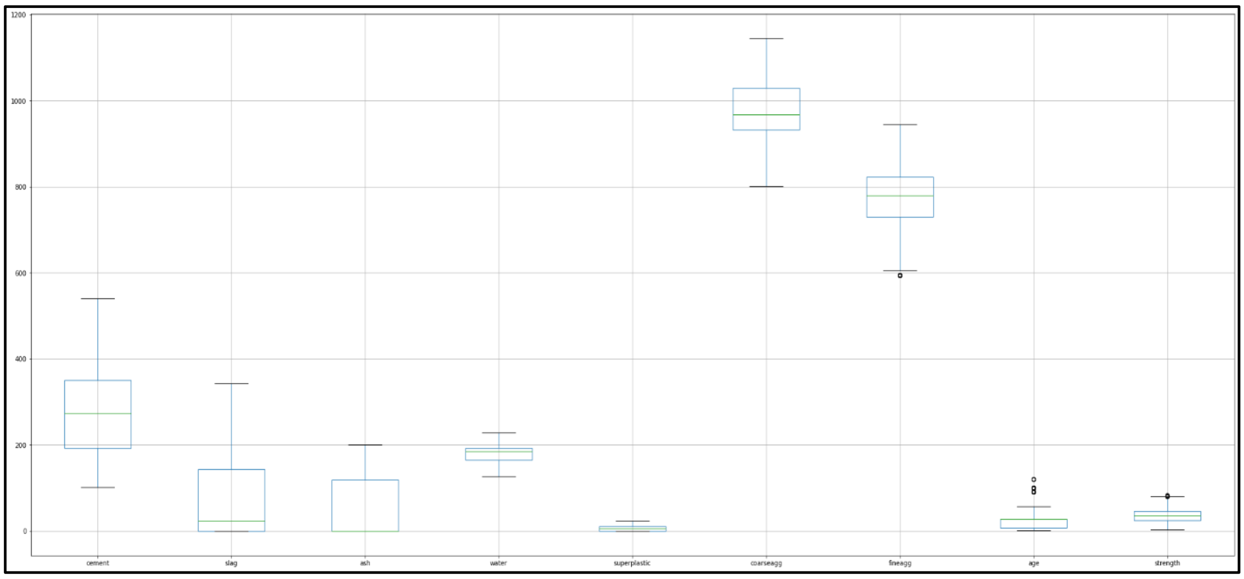
```python
sns.jointplot(x = "age", y = "slag", kind = "scatter", data = df)
sns.jointplot(x = "age", y = "ash", kind = "scatter", data = df)
sns.jointplot(x = "age", y = "water", kind = "scatter", data = df)
sns.jointplot(x = "age", y = "coarseagg", kind = "scatter", data = df)
sns.jointplot(x = "age", y = "fineagg", kind = "scatter", data = df)
sns.jointplot(x = "age", y = "age", kind = "scatter", data = df)
# RELATIONSHIP BETWEEN AGE AND OTHER VARIABLES
plt.show()
```

OBSERVATIONS

- It appears there is a straight relationship between cement and strength.
- The relationship is favourable, but what is clear is that there are various strengths available for a given value of cement; we are unsure which one to choose.
- Therefore, cement is not a very good predictor even though it has a positive link with strength. It is not a strong predictor.
- Strength vs. Slag: There is no discernible pattern.
- Ash vs. strength: Likewise, there is no discernible pattern.
- Water vs. power: There is no discernible pattern.
- Superplastic vs. strength: We have various values of strength for a particular value of age. As a result, it is a poor predictor.
- Strength vs. coarseagg: There is no discernible pattern.
- Strength vs. fineagg: There is no discernible pattern.
- Strength vs. Age: We have several levels of strength for a particular value of age.

OUTLIERS TREATMENT

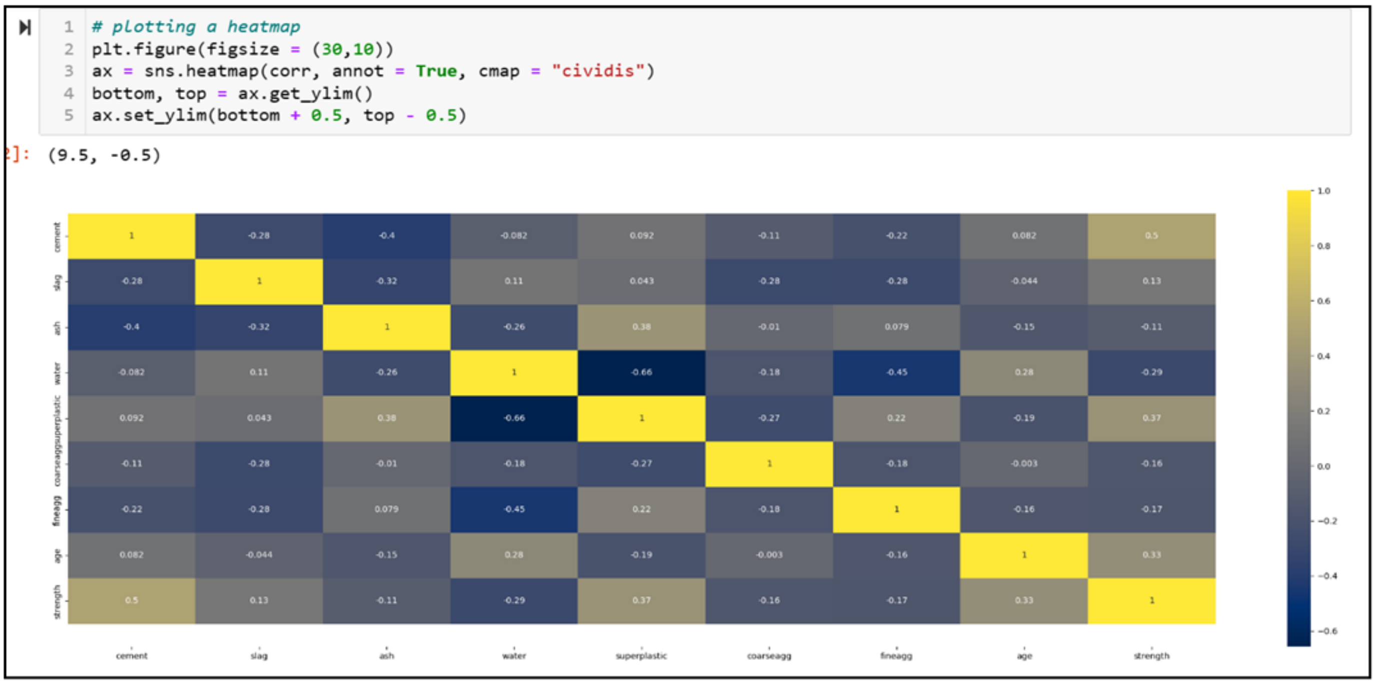
From the above analysis we know that the independent attributes slag, water, superplastic, fineagg and age does have outliers in them. We will replace those outliers with the median of the respective attributes.



MISSING VALUES TREATMENT

None of the independent attributes does have any missing values.

CORRELATION ANALYSIS: HEAT MAP



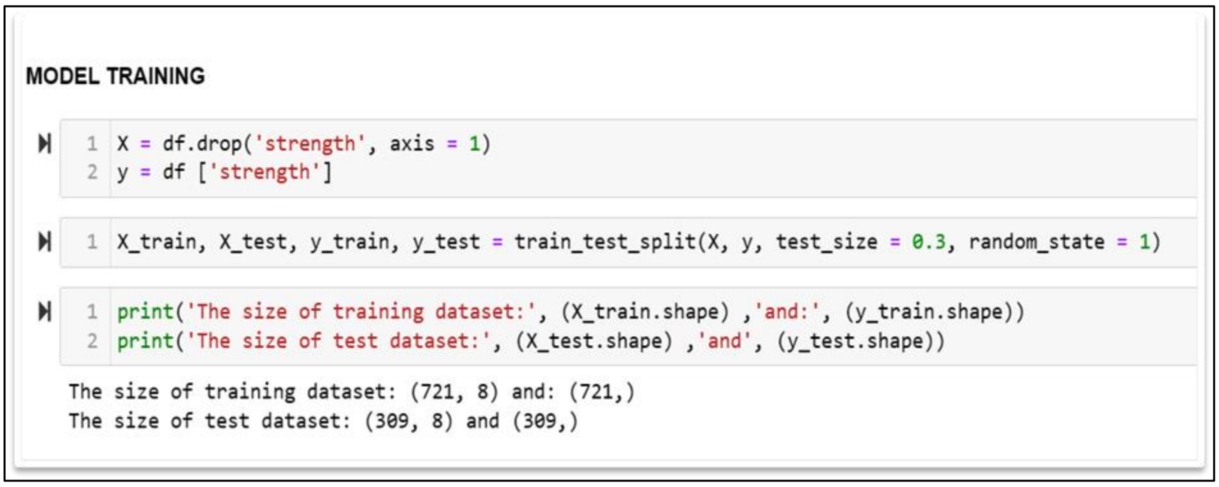
```
# plotting a heatmap
plt.figure(figsize = (30,10))
ax = sns.heatmap(corr, annot = True, cmap = "cividis")
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
```

```
]: (9.5, -0.5)
```

cement 1 -0.28 -0.4 -0.082 0.092 -0.11 -0.22 0.082 0.5
slag -0.28 1 -0.32 0.11 0.043 -0.28 -0.28 -0.044 0.13
ash -0.4 -0.32 1 -0.26 0.38 -0.01 0.079 -0.15 -0.11
water -0.082 0.11 -0.26 1 -0.66 -0.18 -0.45 0.28 -0.29
superplastic 0.092 0.043 0.38 -0.66 1 -0.27 0.22 -0.19 0.37
coarseagg -0.11 -0.28 -0.01 -0.18 -0.27 1 -0.18 -0.003 -0.16
fineagg -0.22 -0.28 0.079 -0.45 0.22 -0.18 1 -0.16 -0.17
age 0.082 -0.044 -0.15 0.28 -0.19 -0.003 -0.16 1 0.33
strength 0.5 0.13 -0.11 -0.29 0.37 -0.16 -0.17 0.33 1
cement slag ash water superplastic coarseagg fineagg age strength
1.0 0.8 0.6 0.4 0.2 0.0 −0.2 −0.4 −0.6

SCALING AND MODELLING OF DATA

X represents the independent variables whereas y the dependent variables.



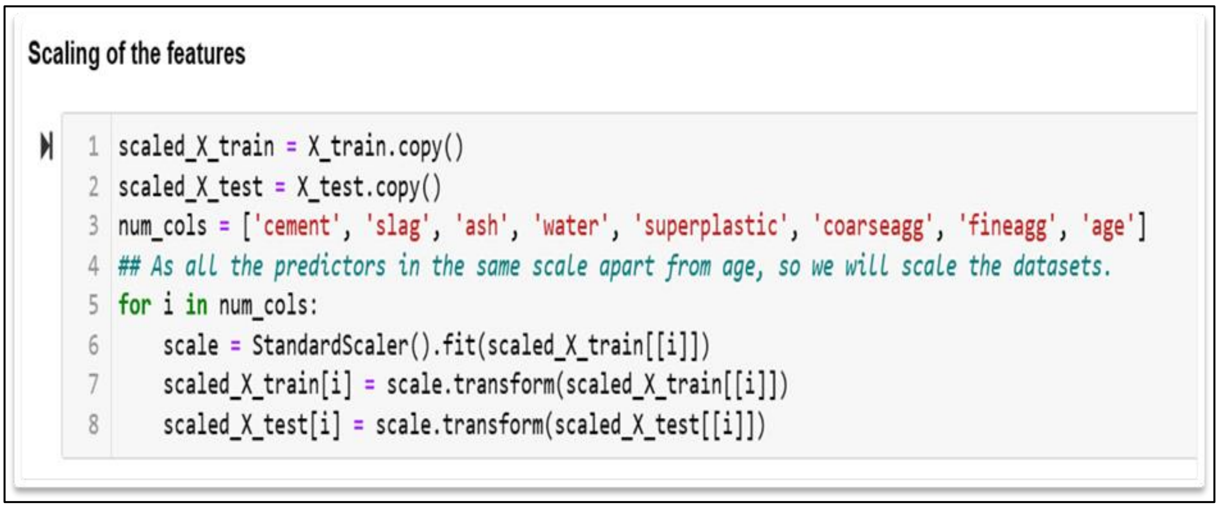
MODEL TRAINING

```
X = df.drop('strength', axis = 1)
y = df ['strength']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 1)
```

```
print('The size of training dataset:', (X_train.shape) ,'and:', (y_train.shape))
print('The size of test dataset:', (X_test.shape) ,'and', (y_test.shape))
```

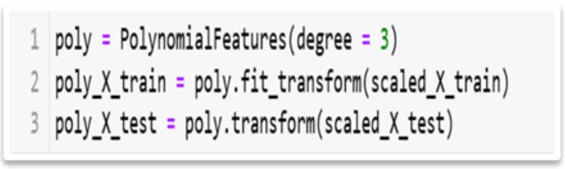
```
The size of training dataset: (721, 8) and: (721,)
The size of test dataset: (309, 8) and (309,)
```



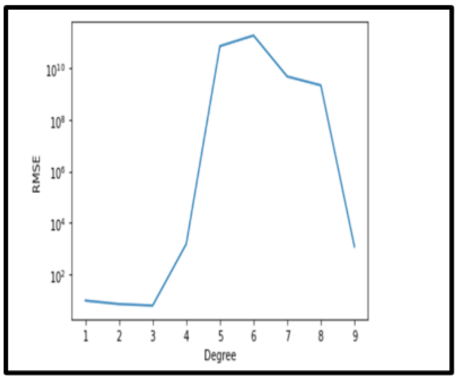
Scaling of the features

```
scaled_X_train = X_train.copy()
scaled_X_test = X_test.copy()
num_cols = ['cement', 'slag', 'ash', 'water', 'superplastic', 'coarseagg', 'fineagg', 'age']
## As all the predictors in the same scale apart from age, so we will scale the datasets.
for i in num_cols:
    scale = StandardScaler().fit(scaled_X_train[[i]])
    scaled_X_train[i] = scale.transform(scaled_X_train[[i]])
    scaled_X_test[i] = scale.transform(scaled_X_test[[i]])
```

Here, we attempted to use the idea of feature engineering to improve the accuracy of the models and lessen the problems with overfitting and underfitting. To enhance the performance of the linear regression model and comprehend the non-linear relationships existing within the dataset being used, polynomial feature engineering has been applied.

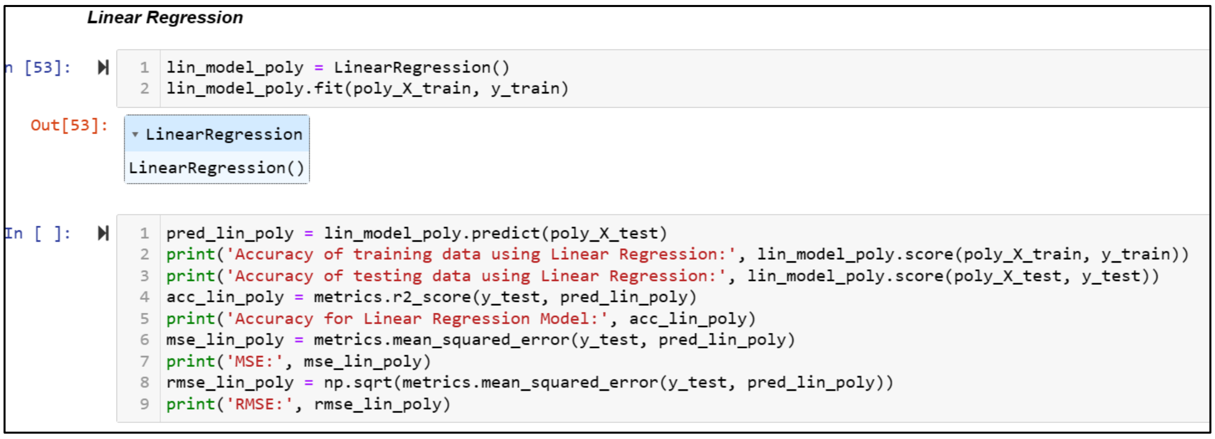


```
poly = PolynomialFeatures(degree = 3)
poly_X_train = poly.fit_transform(scaled_X_train)
poly_X_test = poly.transform(scaled_X_test)
```



MACHINE LEARNING MODELS

LINEAR REGRESSION MODEL

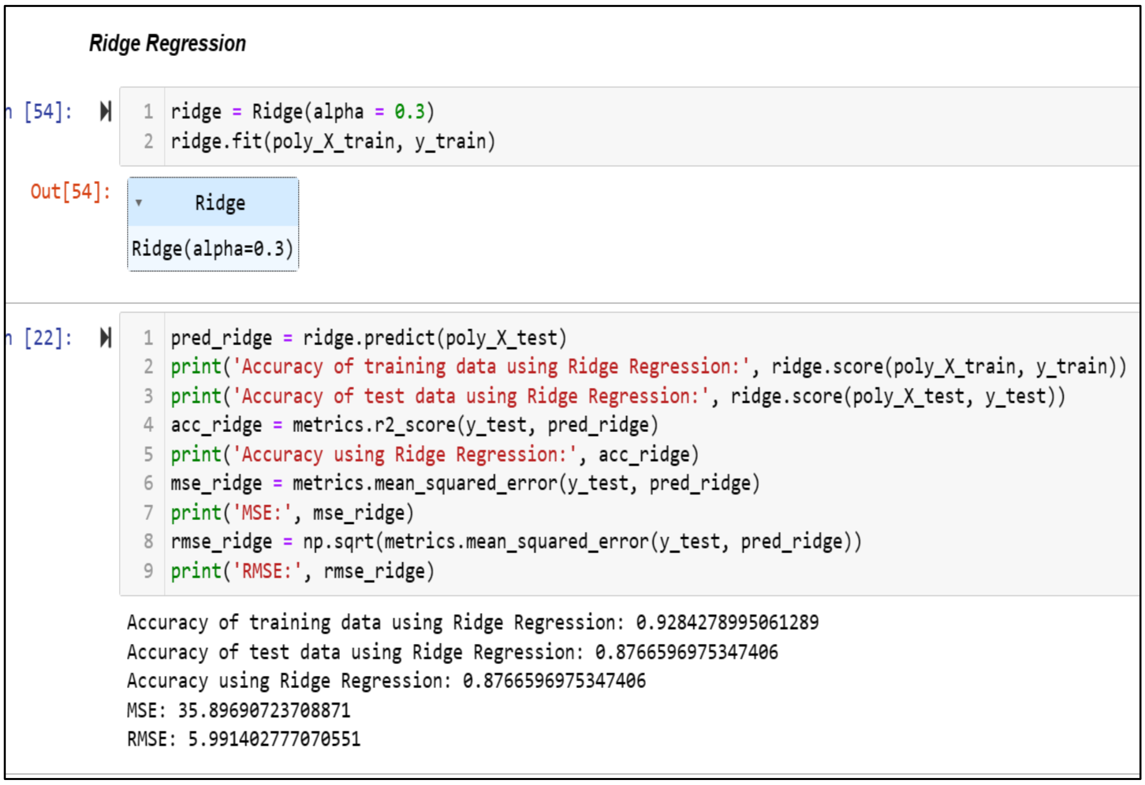


Linear Regression

```
n [53]:  lin_model_poly = LinearRegression()
         lin_model_poly.fit(poly_X_train, y_train)

Out[53]: ▾ LinearRegression
         LinearRegression()

In [ ]:  pred_lin_poly = lin_model_poly.predict(poly_X_test)
         print('Accuracy of training data using Linear Regression:', lin_model_poly.score(poly_X_train, y_train))
         print('Accuracy of testing data using Linear Regression:', lin_model_poly.score(poly_X_test, y_test))
         acc_lin_poly = metrics.r2_score(y_test, pred_lin_poly)
         print('Accuracy for Linear Regression Model:', acc_lin_poly)
         mse_lin_poly = metrics.mean_squared_error(y_test, pred_lin_poly)
         print('MSE:', mse_lin_poly)
         rmse_lin_poly = np.sqrt(metrics.mean_squared_error(y_test, pred_lin_poly))
         print('RMSE:', rmse_lin_poly)
```

Observation: The model can predict around 94% in training dataset and 87.5% in test data with an R2 score of 0.876 and RMSE of 6.01

RIDGE REGRESSOR



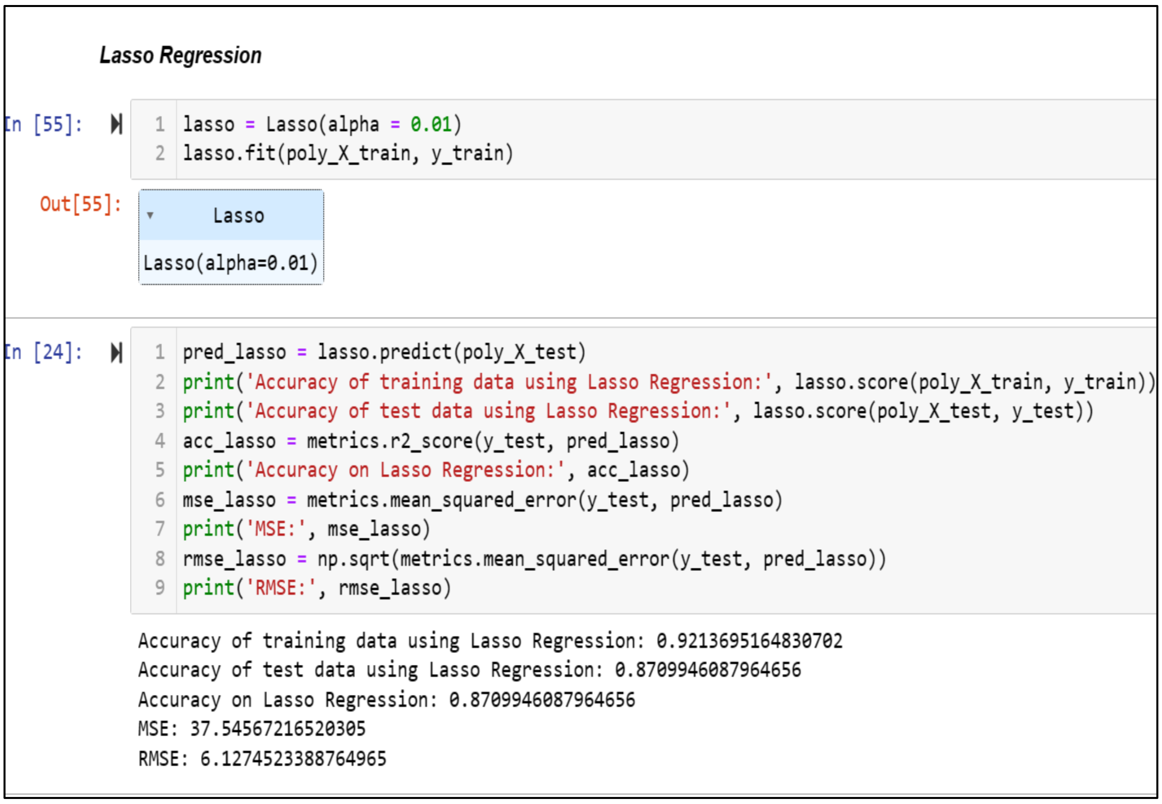
Ridge Regression

```
n [54]:  ridge = Ridge(alpha = 0.3)
         ridge.fit(poly_X_train, y_train)

Out[54]: ▾ Ridge
         Ridge(alpha=0.3)

n [22]:  pred_ridge = ridge.predict(poly_X_test)
         print('Accuracy of training data using Ridge Regression:', ridge.score(poly_X_train, y_train))
         print('Accuracy of test data using Ridge Regression:', ridge.score(poly_X_test, y_test))
         acc_ridge = metrics.r2_score(y_test, pred_ridge)
         print('Accuracy using Ridge Regression:', acc_ridge)
         mse_ridge = metrics.mean_squared_error(y_test, pred_ridge)
         print('MSE:', mse_ridge)
         rmse_ridge = np.sqrt(metrics.mean_squared_error(y_test, pred_ridge))
         print('RMSE:', rmse_ridge)

         Accuracy of training data using Ridge Regression: 0.9284278995061289
         Accuracy of test data using Ridge Regression: 0.8766596975347406
         Accuracy using Ridge Regression: 0.8766596975347406
         MSE: 35.89690723708871
         RMSE: 5.991402777070551
```

Ridge regression is the method used for the analysis of multicollinearity in multiple regression data. It is most suitable when a data set contains a higher number of predictor variables than the number of observations.

Observation: The performance of the model based on Ridge (and polynomial features) is around 94% in training dataset and 88% in test data with an R2 score of 0.88 and RMSE of 5.89

LASSO REGRESSOR



Lasso Regression

```
In [55]:
lasso = Lasso(alpha = 0.01)
lasso.fit(poly_X_train, y_train)
```

```
Out[55]:
▾ Lasso
Lasso(alpha=0.01)
```

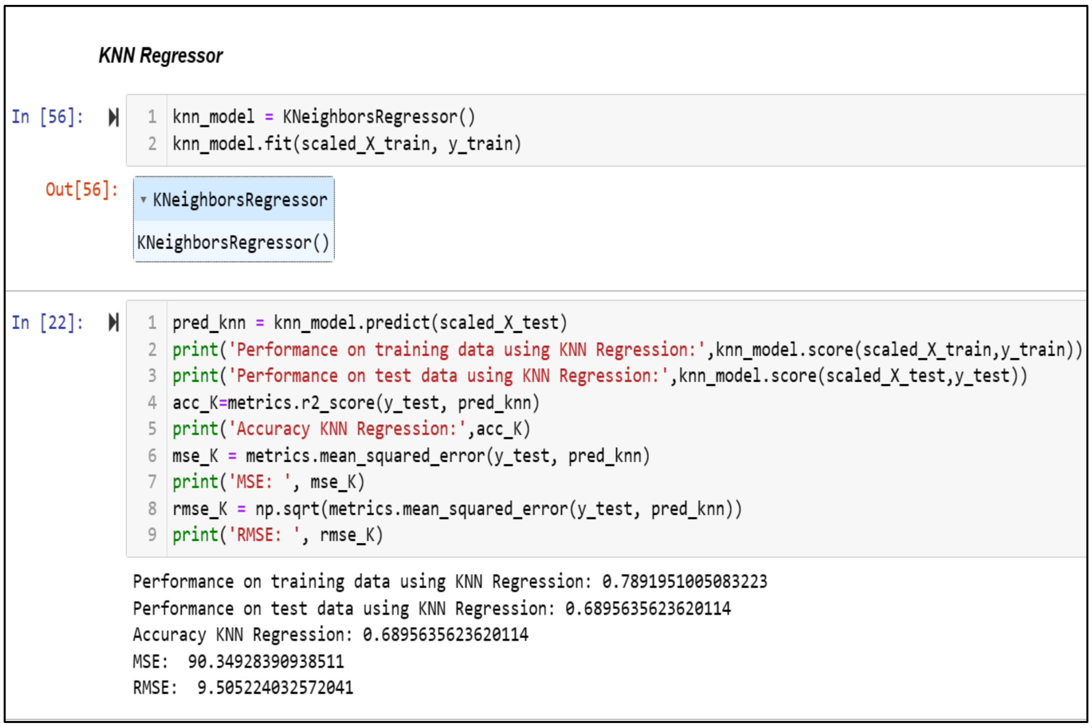
```
In [24]:
pred_lasso = lasso.predict(poly_X_test)
print('Accuracy of training data using Lasso Regression:', lasso.score(poly_X_train, y_train))
print('Accuracy of test data using Lasso Regression:', lasso.score(poly_X_test, y_test))
acc_lasso = metrics.r2_score(y_test, pred_lasso)
print('Accuracy on Lasso Regression:', acc_lasso)
mse_lasso = metrics.mean_squared_error(y_test, pred_lasso)
print('MSE:', mse_lasso)
rmse_lasso = np.sqrt(metrics.mean_squared_error(y_test, pred_lasso))
print('RMSE:', rmse_lasso)
```

```
Accuracy of training data using Lasso Regression: 0.9213695164830702
Accuracy of test data using Lasso Regression: 0.8709946087964656
Accuracy on Lasso Regression: 0.8709946087964656
MSE: 37.54567216520305
RMSE: 6.1274523388764965
```

The most 'classic' use of LASSO is that you have a large set of variables, but only a small number of them are truly important.

Observation: The performance of the model based on Lasso (and polynomial features) is around 93.29% on training dataset and 88.33% on test data with an R2 score of 0.88 and RMSE of 5.82

KNN REGRESSOR



KNN Regressor

```
In [56]:
knn_model = KNeighborsRegressor()
knn_model.fit(scaled_X_train, y_train)

Out[56]:
▾ KNeighborsRegressor
KNeighborsRegressor()
```

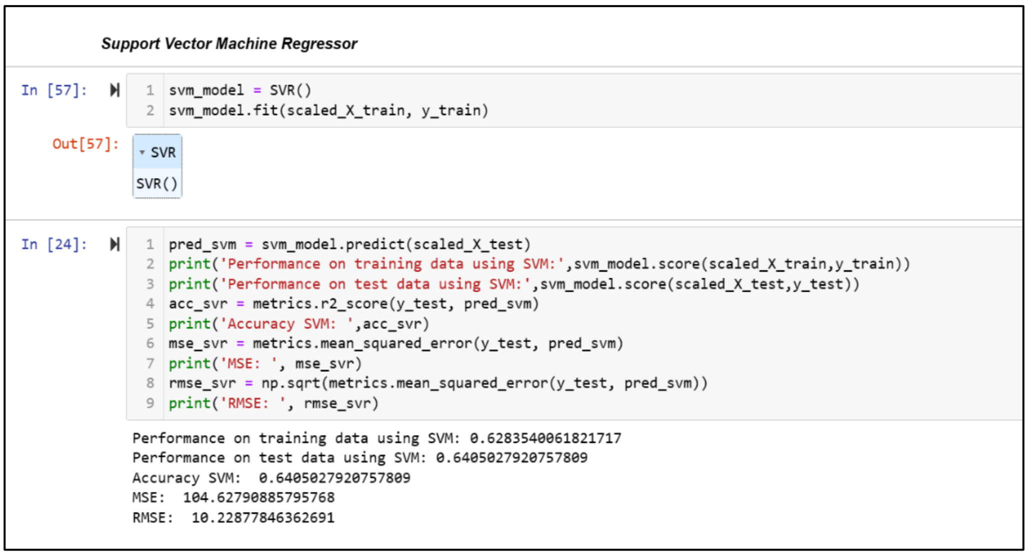
```
In [22]:
pred_knn = knn_model.predict(scaled_X_test)
print('Performance on training data using KNN Regression:',knn_model.score(scaled_X_train,y_train))
print('Performance on test data using KNN Regression:',knn_model.score(scaled_X_test,y_test))
acc_K=metrics.r2_score(y_test, pred_knn)
print('Accuracy KNN Regression:',acc_K)
mse_K = metrics.mean_squared_error(y_test, pred_knn)
print('MSE: ', mse_K)
rmse_K = np.sqrt(metrics.mean_squared_error(y_test, pred_knn))
print('RMSE: ', rmse_K)
```

```
Performance on training data using KNN Regression: 0.7891951005083223
Performance on test data using KNN Regression: 0.6895635623620114
Accuracy KNN Regression: 0.6895635623620114
MSE:  90.34928390938511
RMSE:  9.505224032572041
```

KNN models are easy to implement and handle non-linearities well. Fitting the model also tends to be quick: the computer doesn't have to calculate any particular parameters or values, after all.

Observation: The model based on KNN Regressor (with scaled variables) performs poorly as it has an accuracy of just 85.5% on training dataset and 78.3% on test data with an R2 score of 0.78 and RMSE of 7.94

SVR REGRESSOR



Support Vector Machine Regressor

```
In [57]:
svm_model = SVR()
svm_model.fit(scaled_X_train, y_train)

Out[57]:
▾ SVR
SVR()
```

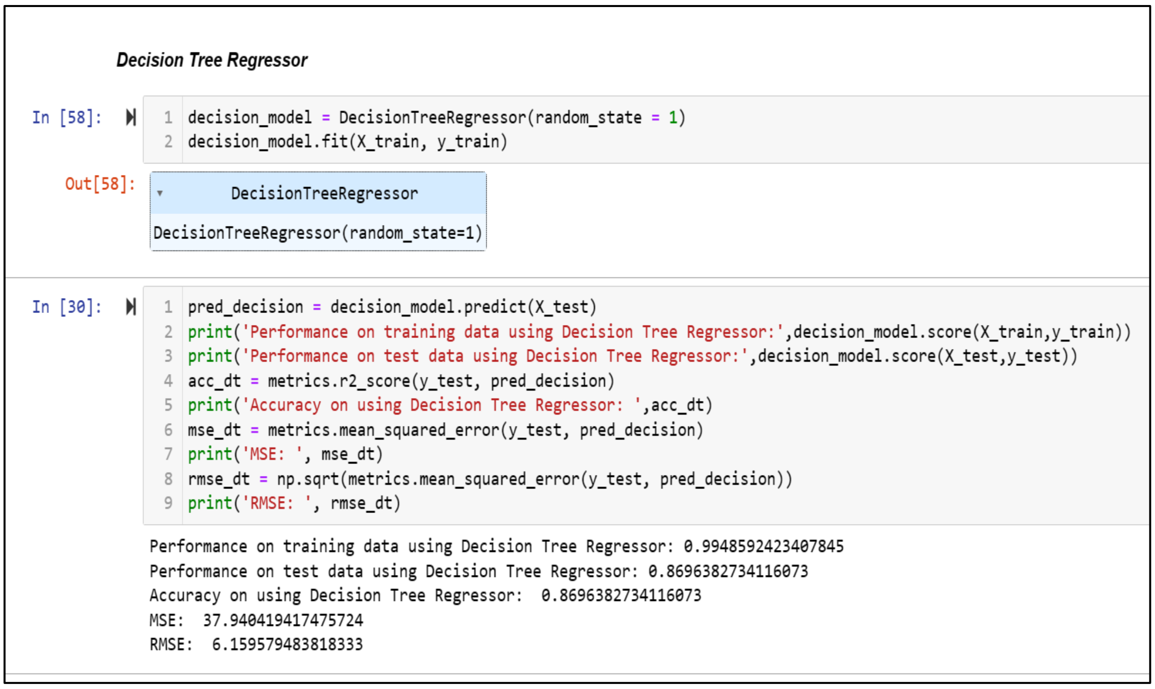
```
In [24]:
pred_svm = svm_model.predict(scaled_X_test)
print('Performance on training data using SVM:',svm_model.score(scaled_X_train,y_train))
print('Performance on test data using SVM:',svm_model.score(scaled_X_test,y_test))
acc_svr = metrics.r2_score(y_test, pred_svm)
print('Accuracy SVM: ',acc_svr)
mse_svr = metrics.mean_squared_error(y_test, pred_svm)
print('MSE: ', mse_svr)
rmse_svr = np.sqrt(metrics.mean_squared_error(y_test, pred_svm))
print('RMSE: ', rmse_svr)
```

```
Performance on training data using SVM: 0.6283540061821717
Performance on test data using SVM: 0.6405027920757809
Accuracy SVM:  0.6405027920757809
MSE:  104.62790885795768
RMSE:  10.22877846362691
```

SVR can handle non-linear relationships between the input variables and the target variable by using a kernel function to map the data to a higher-dimensional space. This makes it a powerful tool for regression tasks where there may be complex relationships between the input variables and the target variable.

Observation: The performance of the model based on SVR is quite poor as it just predicts only 71% on training dataset and 70% on test data (a fine case of Underfitting) with an R2 score of 0.70 and RMSE of 9.31

DECISION TREE



Decision Tree Regressor

In [58]:
```
decision_model = DecisionTreeRegressor(random_state = 1)
decision_model.fit(X_train, y_train)
```

Out[58]:
```
DecisionTreeRegressor
DecisionTreeRegressor(random_state=1)
```

In [30]:
```
pred_decision = decision_model.predict(X_test)
print('Performance on training data using Decision Tree Regressor:',decision_model.score(X_train,y_train))
print('Performance on test data using Decision Tree Regressor:',decision_model.score(X_test,y_test))
acc_dt = metrics.r2_score(y_test, pred_decision)
print('Accuracy on using Decision Tree Regressor: ',acc_dt)
mse_dt = metrics.mean_squared_error(y_test, pred_decision)
print('MSE: ', mse_dt)
rmse_dt = np.sqrt(metrics.mean_squared_error(y_test, pred_decision))
print('RMSE: ', rmse_dt)
```

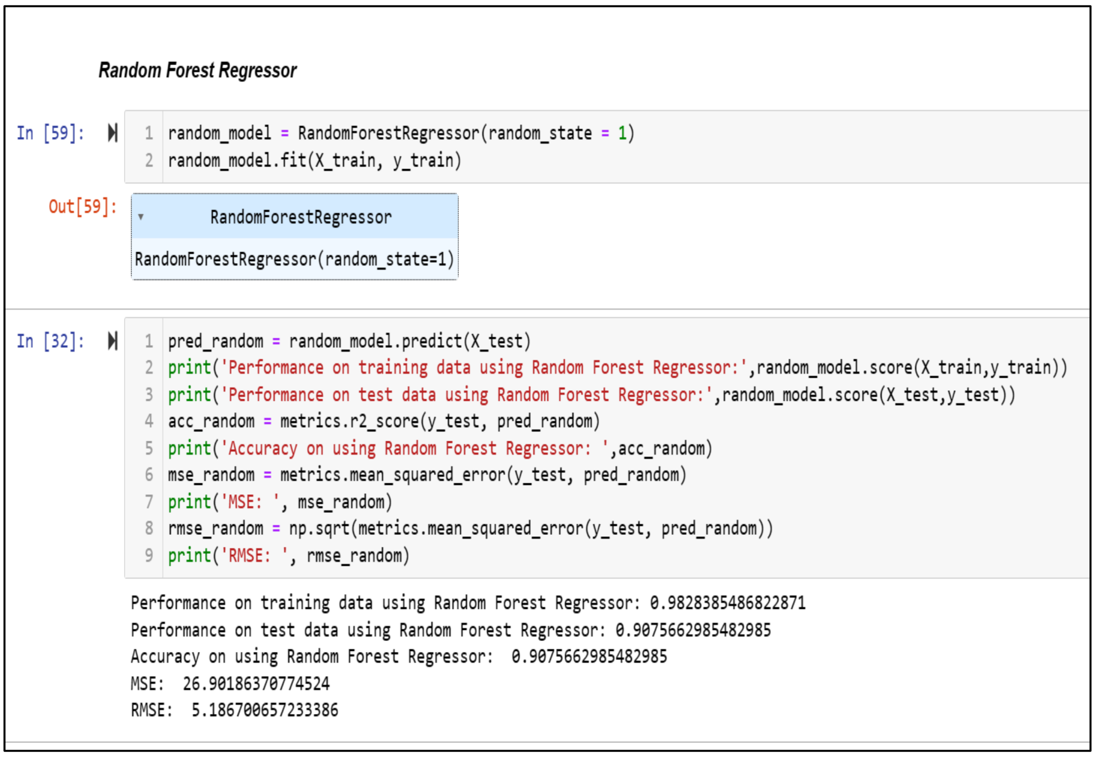
```
Performance on training data using Decision Tree Regressor: 0.9948592423407845
Performance on test data using Decision Tree Regressor: 0.8696382734116073
Accuracy on using Decision Tree Regressor:  0.8696382734116073
MSE:  37.940419417475724
RMSE:  6.159579483818333
```

Decision tree regression observes features of an object and trains a model in the structure of a tree to predict data in the future to produce meaningful continuous output.

Continuous output means that the output/result is not discrete, i.e., it is not represented just by a discrete, known set of numbers or values.

Observation: The performance of the model based on Decision Tree Regressor is around 99% on training dataset and just 85% on test data (a fine example of Overfitting) with an R2 score of 0.85 and RMSE of 6.56

RANDOM FOREST



Random Forest Regressor

In [59]:
```
random_model = RandomForestRegressor(random_state = 1)
random_model.fit(X_train, y_train)
```

Out[59]:
```
RandomForestRegressor
RandomForestRegressor(random_state=1)
```

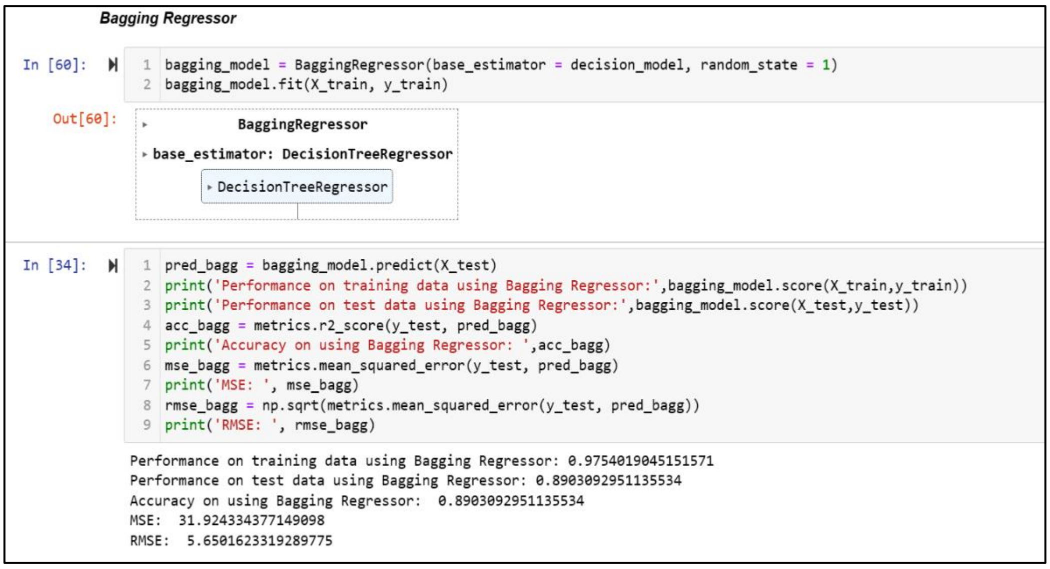
In [32]:
```
pred_random = random_model.predict(X_test)
print('Performance on training data using Random Forest Regressor:',random_model.score(X_train,y_train))
print('Performance on test data using Random Forest Regressor:',random_model.score(X_test,y_test))
acc_random = metrics.r2_score(y_test, pred_random)
print('Accuracy on using Random Forest Regressor: ',acc_random)
mse_random = metrics.mean_squared_error(y_test, pred_random)
print('MSE: ', mse_random)
rmse_random = np.sqrt(metrics.mean_squared_error(y_test, pred_random))
print('RMSE: ', rmse_random)
```

```
Performance on training data using Random Forest Regressor: 0.9828385486822871
Performance on test data using Random Forest Regressor: 0.9075662985482985
Accuracy on using Random Forest Regressor:  0.9075662985482985
MSE:  26.90186370774524
RMSE:  5.186700657233386
```

The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees.

Observation: The performance of the model based on Random Forest Regressor is around 97% on training dataset and 90% on test data with an R2 score of 0.9 and RMSE of 5.32

BAGGING



Bagging Regressor

In [60]:
```
bagging_model = BaggingRegressor(base_estimator = decision_model, random_state = 1)
bagging_model.fit(X_train, y_train)
```

Out[60]:
```
BaggingRegressor
base_estimator: DecisionTreeRegressor
    DecisionTreeRegressor
```

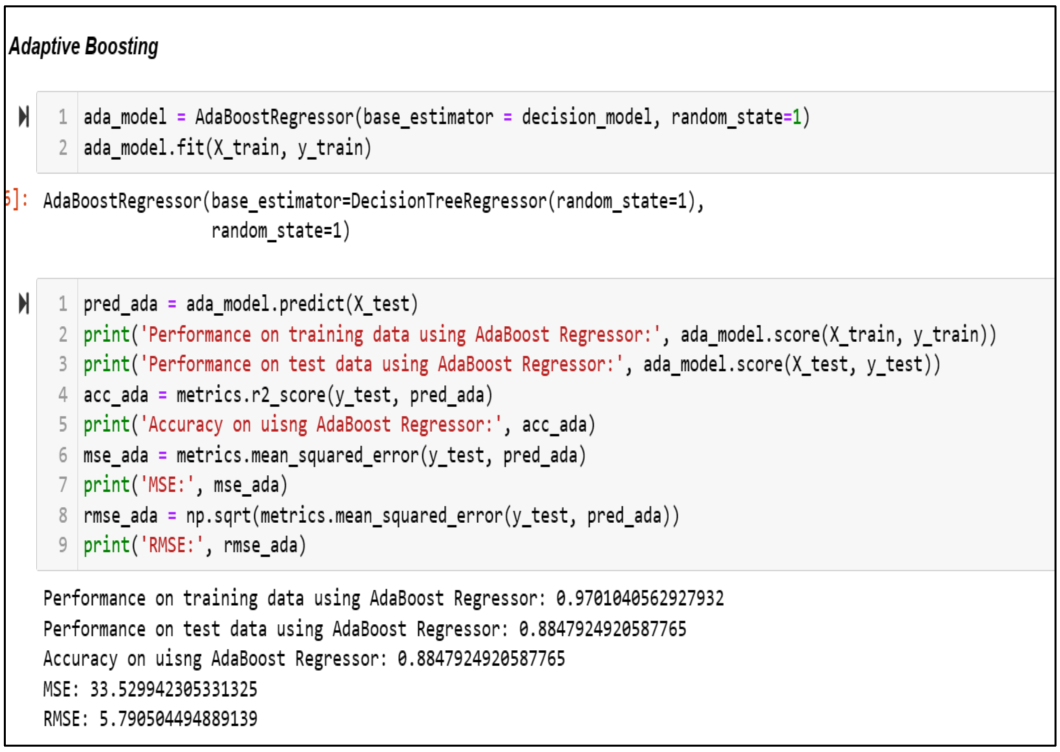
In [34]:
```
pred_bagg = bagging_model.predict(X_test)
print('Performance on training data using Bagging Regressor:',bagging_model.score(X_train,y_train))
print('Performance on test data using Bagging Regressor:',bagging_model.score(X_test,y_test))
acc_bagg = metrics.r2_score(y_test, pred_bagg)
print('Accuracy on using Bagging Regressor: ',acc_bagg)
mse_bagg = metrics.mean_squared_error(y_test, pred_bagg)
print('MSE: ', mse_bagg)
rmse_bagg = np.sqrt(metrics.mean_squared_error(y_test, pred_bagg))
print('RMSE: ', rmse_bagg)
```

```
Performance on training data using Bagging Regressor: 0.9754019045151571
Performance on test data using Bagging Regressor: 0.8903092951135534
Accuracy on using Bagging Regressor:  0.8903092951135534
MSE:  31.924334377149098
RMSE:  5.6501623319289775
```

Observation: The model based on Bagging (and a Decision Tree based model as an estimator) performs 97% on training data and 89% on test data with a R2 score of 0.89 and RMSE of 5.74.

Bagging avoids overfitting of data and is used for both regression and classification models, specifically for decision tree algorithms.

ADABOOST



Adaptive Boosting

```
ada_model = AdaBoostRegressor(base_estimator = decision_model, random_state=1)
ada_model.fit(X_train, y_train)
```

```
]: AdaBoostRegressor(base_estimator=DecisionTreeRegressor(random_state=1),
                    random_state=1)
```

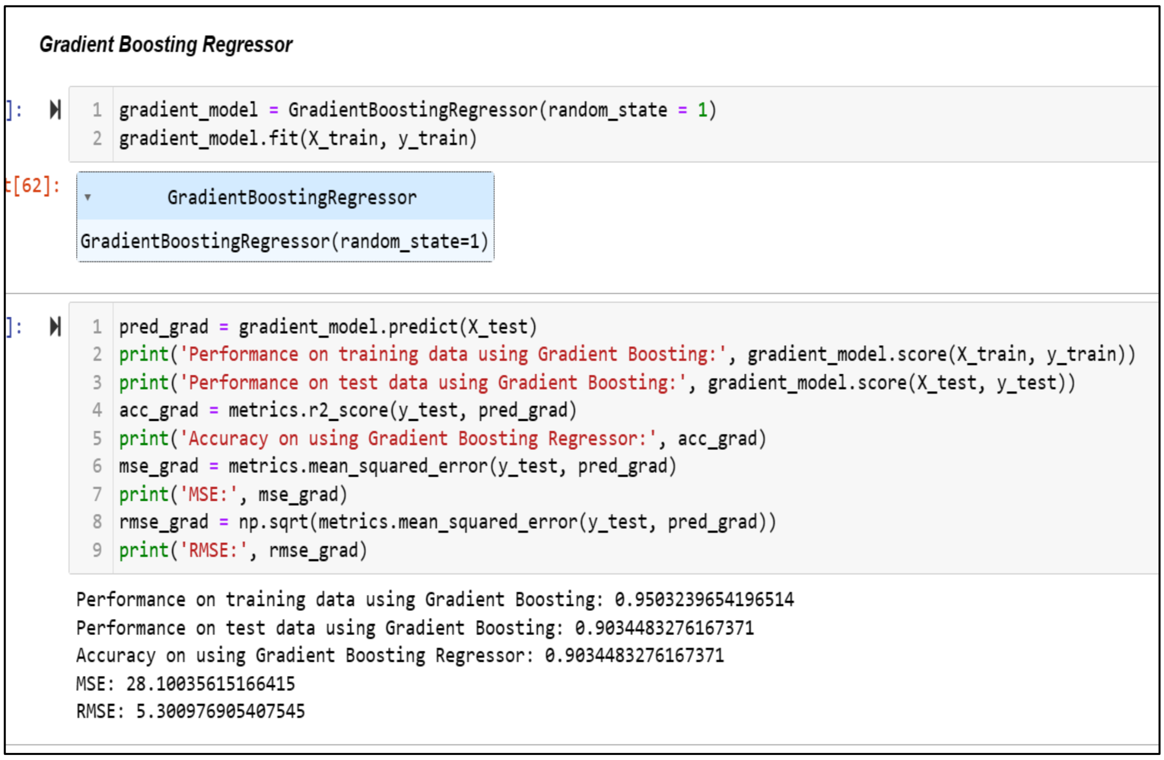
```
pred_ada = ada_model.predict(X_test)
print('Performance on training data using AdaBoost Regressor:', ada_model.score(X_train, y_train))
print('Performance on test data using AdaBoost Regressor:', ada_model.score(X_test, y_test))
acc_ada = metrics.r2_score(y_test, pred_ada)
print('Accuracy on uisng AdaBoost Regressor:', acc_ada)
mse_ada = metrics.mean_squared_error(y_test, pred_ada)
print('MSE:', mse_ada)
rmse_ada = np.sqrt(metrics.mean_squared_error(y_test, pred_ada))
print('RMSE:', rmse_ada)
```

```
Performance on training data using AdaBoost Regressor: 0.9701040562927932
Performance on test data using AdaBoost Regressor: 0.8847924920587765
Accuracy on uisng AdaBoost Regressor: 0.8847924920587765
MSE: 33.529942305331325
RMSE: 5.790504494889139
```

Switching from linear regression to ensembles of decision stumps (aka AdaBoost) allows us to capture many of these non-linear relationships, which translates into better prediction accuracy on the problem of interest

Observation: The model based on Adaptive Boosting Regressor (and Decision Tree Regressor as a base model) performs correctly 97% on training data while 88% on test data with a R2 score of 0.88 and RMSE of 5.79

GRADIENT BOOSTING



Gradient Boosting Regressor

```
]:  gradient_model = GradientBoostingRegressor(random_state = 1)
    gradient_model.fit(X_train, y_train)
```

```
t[62]:  GradientBoostingRegressor

        GradientBoostingRegressor(random_state=1)
```

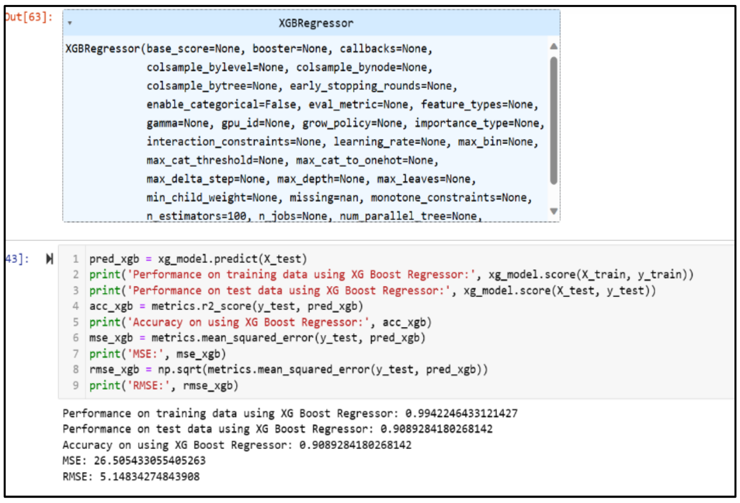
```
]:  pred_grad = gradient_model.predict(X_test)
    print('Performance on training data using Gradient Boosting:', gradient_model.score(X_train, y_train))
    print('Performance on test data using Gradient Boosting:', gradient_model.score(X_test, y_test))
    acc_grad = metrics.r2_score(y_test, pred_grad)
    print('Accuracy on using Gradient Boosting Regressor:', acc_grad)
    mse_grad = metrics.mean_squared_error(y_test, pred_grad)
    print('MSE:', mse_grad)
    rmse_grad = np.sqrt(metrics.mean_squared_error(y_test, pred_grad))
    print('RMSE:', rmse_grad)
```

```
Performance on training data using Gradient Boosting: 0.9503239654196514
Performance on test data using Gradient Boosting: 0.9034483276167371
Accuracy on using Gradient Boosting Regressor: 0.9034483276167371
MSE: 28.10035615166415
RMSE: 5.300976905407545
```

Gradient boosting trees can be more accurate than random forests. Because we train them to correct each other's errors, they are capable of capturing complex patterns in the data.

Observation: The model based on Gradient Boosting Regression performs accurately about 94% on training data and 89.2% on test data with a R2 score of 0.89 and RMSE of 5.59

XG BOOST



```
ut[63]:  XGBRegressor

         XGBRegressor(base_score=None, booster=None, callbacks=None,
                      colsample_bylevel=None, colsample_bynode=None,
                      colsample_bytree=None, early_stopping_rounds=None,
                      enable_categorical=False, eval_metric=None, feature_types=None,
                      gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
                      interaction_constraints=None, learning_rate=None, max_bin=None,
                      max_cat_threshold=None, max_cat_to_onehot=None,
                      max_delta_step=None, max_depth=None, max_leaves=None,
                      min_child_weight=None, missing=nan, monotone_constraints=None,
                      n_estimators=100, n_jobs=None, num_parallel_tree=None,
```

```
43]:  pred_xgb = xg_model.predict(X_test)
      print('Performance on training data using XG Boost Regressor:', xg_model.score(X_train, y_train))
      print('Performance on test data using XG Boost Regressor:', xg_model.score(X_test, y_test))
      acc_xgb = metrics.r2_score(y_test, pred_xgb)
      print('Accuracy on using XG Boost Regressor:', acc_xgb)
      mse_xgb = metrics.mean_squared_error(y_test, pred_xgb)
      print('MSE:', mse_xgb)
      rmse_xgb = np.sqrt(metrics.mean_squared_error(y_test, pred_xgb))
      print('RMSE:', rmse_xgb)
```

```
Performance on training data using XG Boost Regressor: 0.9942246433121427
Performance on test data using XG Boost Regressor: 0.9089284180268142
Accuracy on using XG Boost Regressor: 0.9089284180268142
MSE: 26.505433055405263
RMSE: 5.14834274843908
```

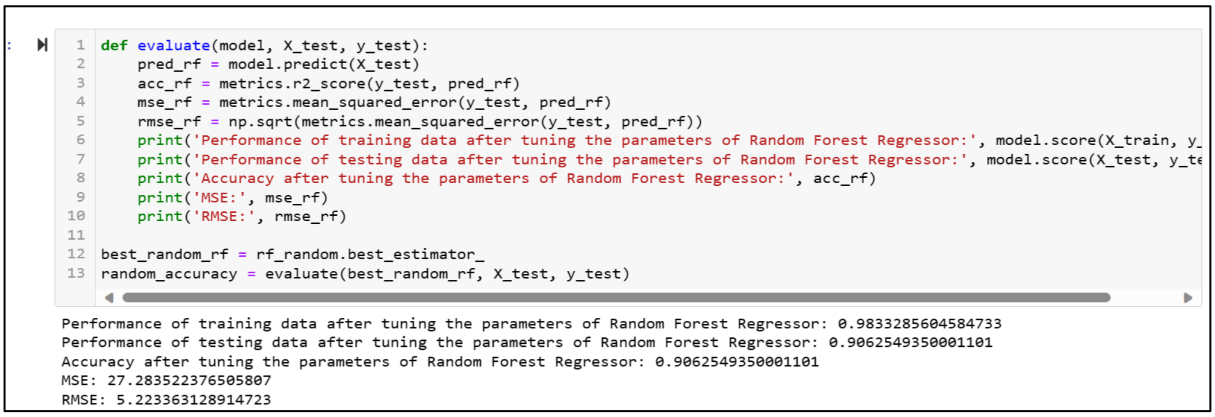
XG Boost performs better than a normal gradient boosting algorithm and that is why it is much faster than that also.

The model based on XG Boost Regressor performs accurately about 99.11% on training data and just 89.9% on test data with a R2 score of 0.899 and RMSE of 5.4

HYPERPARAMETER TUNING

When you locate a model that is "appropriate" for your task or create a model's architecture, you must modify hyperparameters to ensure that the model can produce accurate predictions. It occurs when modifying the model's architecture was unable to make it perform better.

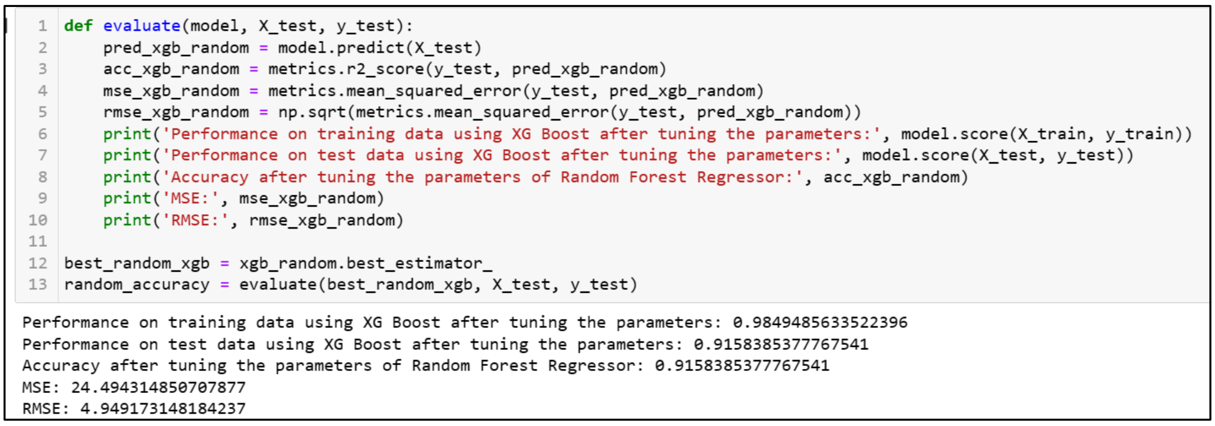
You can choose from a variety of methods for adjusting hyperparameters, including grid search and random search, depending on your preferences. The key idea is that the "appropriate" model may vary depending on the application.



```python
def evaluate(model, X_test, y_test):
    pred_rf = model.predict(X_test)
    acc_rf = metrics.r2_score(y_test, pred_rf)
    mse_rf = metrics.mean_squared_error(y_test, pred_rf)
    rmse_rf = np.sqrt(metrics.mean_squared_error(y_test, pred_rf))
    print('Performance of training data after tuning the parameters of Random Forest Regressor:', model.score(X_train, y_
    print('Performance of testing data after tuning the parameters of Random Forest Regressor:', model.score(X_test, y_te
    print('Accuracy after tuning the parameters of Random Forest Regressor:', acc_rf)
    print('MSE:', mse_rf)
    print('RMSE:', rmse_rf)

best_random_rf = rf_random.best_estimator_
random_accuracy = evaluate(best_random_rf, X_test, y_test)
```

```
Performance of training data after tuning the parameters of Random Forest Regressor: 0.9833285604584733
Performance of testing data after tuning the parameters of Random Forest Regressor: 0.9062549350001101
Accuracy after tuning the parameters of Random Forest Regressor: 0.9062549350001101
MSE: 27.283522376505807
RMSE: 5.223363128914723
```

Observation: After training and predicting the model with Random Forest Regressor for their best parameter we were able to improve the performance of the model by a little bit. The model now correctly predicts 98% on training data and 90.367% on test data with an R2 score of 0.90366 and RMSE of 5.29.



```python
def evaluate(model, X_test, y_test):
    pred_xgb_random = model.predict(X_test)
    acc_xgb_random = metrics.r2_score(y_test, pred_xgb_random)
    mse_xgb_random = metrics.mean_squared_error(y_test, pred_xgb_random)
    rmse_xgb_random = np.sqrt(metrics.mean_squared_error(y_test, pred_xgb_random))
    print('Performance on training data using XG Boost after tuning the parameters:', model.score(X_train, y_train))
    print('Performance on test data using XG Boost after tuning the parameters:', model.score(X_test, y_test))
    print('Accuracy after tuning the parameters of Random Forest Regressor:', acc_xgb_random)
    print('MSE:', mse_xgb_random)
    print('RMSE:', rmse_xgb_random)

best_random_xgb = xgb_random.best_estimator_
random_accuracy = evaluate(best_random_xgb, X_test, y_test)
```

```
Performance on training data using XG Boost after tuning the parameters: 0.9849485633522396
Performance on test data using XG Boost after tuning the parameters: 0.9158385377767541
Accuracy after tuning the parameters of Random Forest Regressor: 0.9158385377767541
MSE: 24.494314850707877
RMSE: 4.949173148184237
```

Observation: After training and predicting the model with XG Boost Regression for their best parameter we were able to improve the performance of the model. The model now correctly predicts 98.49% on training data and 91.58% on test data with an R2 score of 0.9158 and RMSE of 4.94.

CONCLUSION

- We can see that out of the twelve models tried out here the top three performers were (in descending order according to the R2 score obtained): Random Forest Regressor, XG Boost Regressor and Gradient Boosting Regressor.

- These three models also have the least RMSE when compared to the models developed by using other algorithms.

- Tree based models have performed better as compared to other models. We must always tune our parameters of a model before stating that so and so model is the best performing one (as was observed in the case of XG Boost Regression).

- We need to use parameters based on polynomial features shows improvement when polynomial features were employed by Linear Regression.