Project Overview: Predicting Compressive Strength of Green Concrete

Introduction

Green concrete is an environment-friendly version of the traditional concrete, manufactured using waste materials as at least one of its constituents. The quality of green concrete used in construction projects is typically evaluated by its compressive strength, which contributes significantly to the durability and longevity of structures. The conventional method of testing concrete strength requires a 28-day period, making it a time-consuming process. Through the power of data science, this process can be expedited by creating a predictive model that estimates the compressive strength of green concrete based on its composition.

Objective

The primary goal of this project is to apply machine learning techniques to predict the compressive strength of green concrete. By taking into account the amount of each raw material used in the concrete mix, the model should be able to estimate the strength of the concrete, effectively assisting in maintaining construction quality standards, while also promoting sustainable practices.

Approach

This project follows the typical stages of a machine learning pipeline, including Data Exploration, Data Cleaning, Feature Engineering, Model Building, and Model Testing. A variety of machine learning algorithms were explored to find the most suitable one for this use-case.

Domain knowledge is indispensable for data scientists. It refers to specialized knowledge or expertise in a specific area of interest or field of study. In the context of this project, understanding the construction domain, particularly the factors that influence the quality of green concrete, was essential. It allowed for a more informed and effective application of machine learning techniques to solve this specific problem.

By applying machine learning techniques, I was able to build a predictive model that effectively predicts the compressive strength of green concrete. The resultant model helps ensure that construction projects maintain high quality standards by enabling quick and accurate testing of green concrete strength. This project not only demonstrated the utility of machine learning in enhancing efficiency but also highlighted the potential of data science in promoting sustainability in the construction industry.

Data Description

Dataset Overview

The project revolves around the "Concrete Compressive Strength" dataset, a multivariate dataset detailing the various components involved in the production of concrete and their impact on the final product's compressive strength. This dataset was generously donated by Prof. I-Cheng Yeh from Chung-Hua University and comprises 1030 observations across 9 distinct attributes.

Variables Breakdown

The dataset includes 8 quantitative input variables, all measured in kg in a m3 mixture, and one output variable, the concrete compressive strength, measured in MPa. Here's a brief description of each:

- **Cement (component 1)**: A key ingredient that provides strength and stability.
- **Blast Furnace Slag (component 2)**: Enhances the durability and resistance of the blocks.
- **Fly Ash (component 3)**: A byproduct of burning coal, it contributes to the strength and workability of the blocks.
- **Water (component 4)**: Essential for the chemical reaction that binds all components together.
- **Superplasticizer (component 5)**: Improves the workability and flow of the concrete mixture.
- **Coarse Aggregate (component 6)**: Reinforces and stabilizes the mixture.
- **Fine Aggregate (component 7)**: Fills in the gaps between the coarse aggregates for a smoother and cohesive mixture.
- **Age**: Refers to the number of days since the initial mixing, a crucial factor for determining strength and durability.

The target variable, **Concrete Compressive Strength**, represents the strength of the concrete blocks.

Insights from Data

By studying the relationship between these components and their quantities, we can gain invaluable insights into the optimal combination for producing high-quality concrete. This is pivotal, as the compressive strength of concrete determines its quality, and hence, the stability and durability of the structures it forms.

This dataset will serve as the foundation for building and optimizing a regression model, predicting the compressive strength of green concrete. The objective is to significantly reduce the time it takes to test the strength of the concrete and help in the creation of more environmentally friendly construction materials.

Click here for the dataset

Source : UCI Machine Learning Repository

Data Preprocessing

Importing Modules

In [34]:
```
# Import necessary modules

import numpy as np
import pandas as pd
import pandas_profiling as pp
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import cross_val_score, KFold
from sklearn.preprocessing import PolynomialFeatures
import xgboost as xgb

import warnings
warnings.filterwarnings('ignore')
```

To preprocess our data, the first step is to import it from the Astra DB. The data extracted from the database is raw and may contain inconsistencies, errors, or missing values. The process of data preprocessing helps to clean, format, and organize the raw data, thereby improving its quality and increasing the efficiency of our predictive models.

Once the data is imported, it undergoes a series of preprocessing steps, which might include:

- Data Cleaning: This step helps in dealing with the missing values and outliers in our dataset. This could be done either by deleting/ignoring such values or replacing them with a central value like mean, median, or mode.
- Data Transformation: The goal here is to improve the accuracy of our models. Techniques like scaling (bringing all values within a similar range), standardizing, or normalizing could be used.
- Feature Selection: All the features or variables in our dataset might not be useful in building predictive models. Feature selection methods can be used to identify and select the most important features.

In [102…
```
# Load the data Astra DB

from cassandra.cluster import Cluster
```

```
from cassandra.auth import PlainTextAuthProvider

cloud_config= {
    'secure_connect_bundle': 'D:\secure-connect-ml-projects.zip'
}
auth_provider = PlainTextAuthProvider('ylHAUuutJalGrfaMhsAGyYGS', 'UrdCS4S7jOrnzxgm
cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider)
session = cluster.connect()

row = session.execute("select release_version from system.local").one()
if row:
    print(row[0])
else:
    print("An error occurred.")
```

```
4.0.0.6816
```

In [103…

```
# Load data in dataframe

query = "SELECT * FROM concrete_strength_prediction.concrete_data"

rows = session.execute(query)

df = pd.DataFrame(list(rows))

df.head(5)
```

Out[103]:

	cement	concrete_compressive_strength	age	blast_furnace_slag	coarse_aggregate	fine_
0	182	21.5	14	45.2	1059.4	
1	237.5	26.26	7	237.5	932	
2	212.5	26.31	14	0	1007.8	
3	307	36.15	365	0	968	
4	168	39.23	100	42.1	1058.7	

Variable Information:

Given is the variable name, variable type, the measurement unit and a brief description. The concrete compressive strength is the regression problem. The order of this listing corresponds to the order of numerals along the rows of the database.

Name -- Data Type -- Measurement -- Description

Cement (component 1) -- quantitative -- kg in a m3 mixture -- Input Variable

Blast Furnace Slag (component 2) -- quantitative -- kg in a m3 mixture -- Input Variable

Fly Ash (component 3) -- quantitative -- kg in a m3 mixture -- Input Variable

Water (component 4) -- quantitative -- kg in a m3 mixture -- Input Variable

Superplasticizer (component 5) -- quantitative -- kg in a m3 mixture -- Input Variable

Coarse Aggregate (component 6) -- quantitative -- kg in a m3 mixture -- Input Variable

Fine Aggregate (component 7) -- quantitative -- kg in a m3 mixture -- Input Variable

Age -- quantitative -- Day (1~365) -- Input Variable

Concrete compressive strength -- quantitative -- MPa -- Output Variable

source : https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength

EDA

Auto EDA (Exploratory Data Analysis) is a powerful approach to quickly gain insights and understand the characteristics of a dataset. Two popular tools for performing Auto EDA are AutoPandas Profiling and AutoViz.

In [107…
```
df.dtypes
```

Out[107]:
```
cement                          object
concrete_compressive_strength   object
age                              int64
blast_furnace_slag              object
coarse_aggregate                object
fine_aggregate                  object
fly_ash                         object
superplasticizer                object
water                           object
dtype: object
```

In [108…
```
# change object features to float

columns_to_convert = ['cement', 'concrete_compressive_strength', 'blast_furnace_sla

for column in columns_to_convert:
    df[column] = df[column].astype(float)
```

In [109…
```
df.dtypes
```

Out[109]:
```
cement                          float64
concrete_compressive_strength   float64
age                               int64
blast_furnace_slag              float64
coarse_aggregate                float64
fine_aggregate                  float64
fly_ash                         float64
superplasticizer                float64
water                           float64
dtype: object
```

In [7]:
```
# profiling total report using pandas_profiling
```

```
profile_report = pp.ProfileReport(df)
profile_report.to_file("Profile_report.html")
profile_report
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]
Render HTML:   0%|          | 0/1 [00:00<?, ?it/s]
Export report to file:   0%|          | 0/1 [00:00<?, ?it/s]
```

Pandas Profiling Report

Overview

Overview | Alerts 8 | Reproduction

Dataset statistics

Number of variables	9
Number of observations	1030
Missing cells	0
Missing cells (%)	0.0%
Duplicate rows	11
Duplicate rows (%)	1.1%
Total size in memory	72.5 KiB
Average record size in memory	72.1 B

Variable types

Numeric	9

Variables

Select Columns

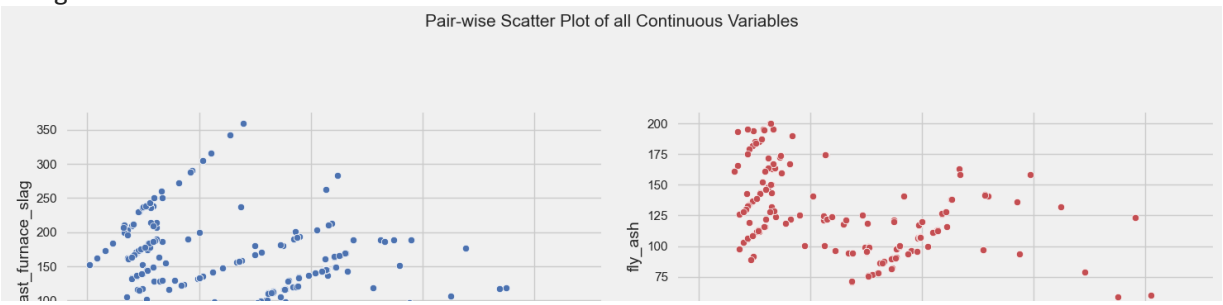
Out[7]:

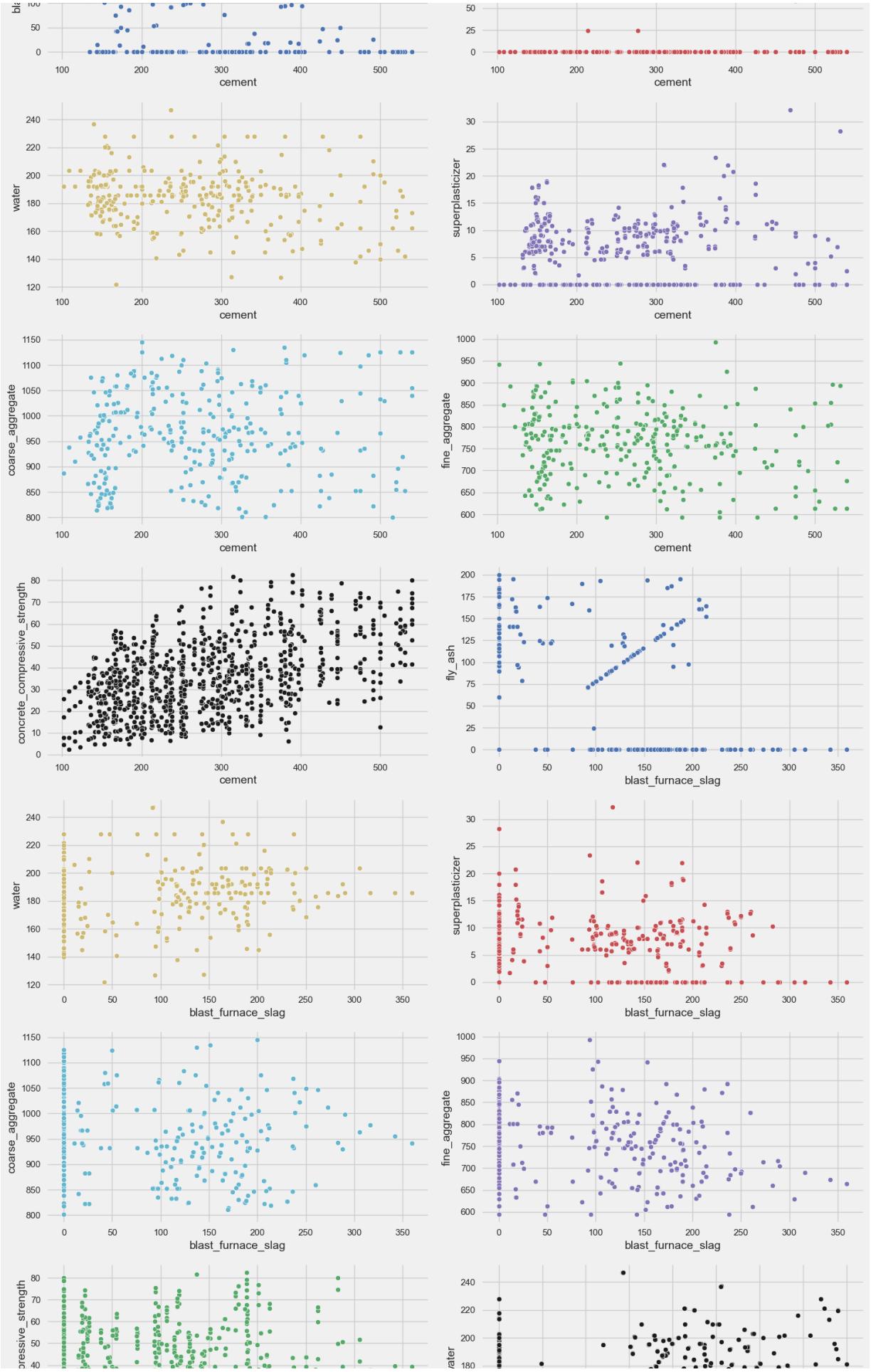
In [8]:
```
%matplotlib inline

from autoviz.AutoViz_Class import AutoViz_Class

plt.figure(figsize=(10, 5))
AV = AutoViz_Class()
df_av = AV.AutoViz(r"D:\INeuron_Projects\Concrete_Com Test Pred\concrete_data.csv")

plt.show()
```

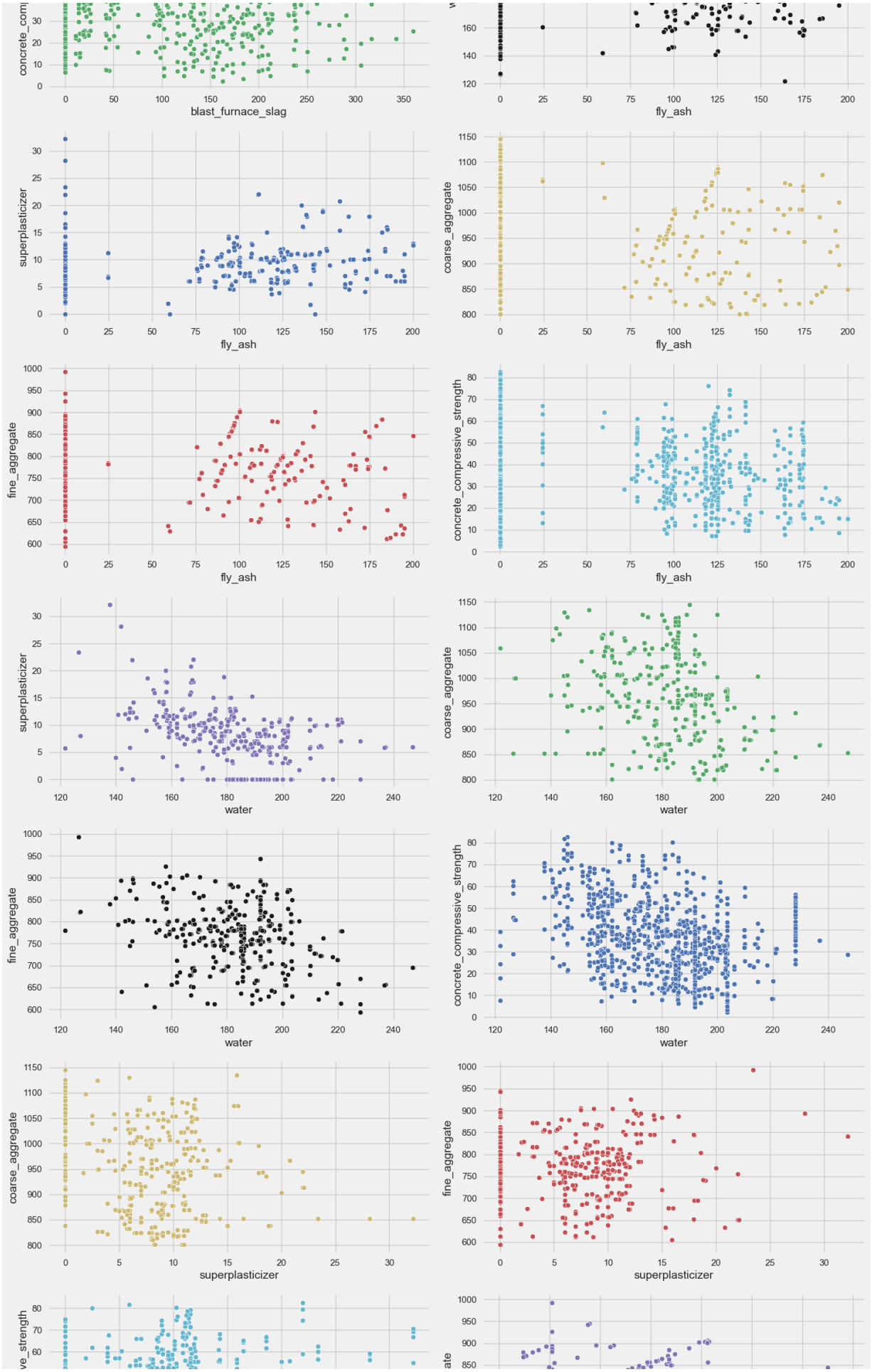
```
Shape of your Data Set loaded: (1030, 9)
#######################################################################################
###
######################## C L A S S I F Y I N G  V A R I A B L E S  ###################
###
#######################################################################################
###
Classifying variables in data set...
Data cleaning improvement suggestions. Complete them before proceeding to ML modelin
g.
```

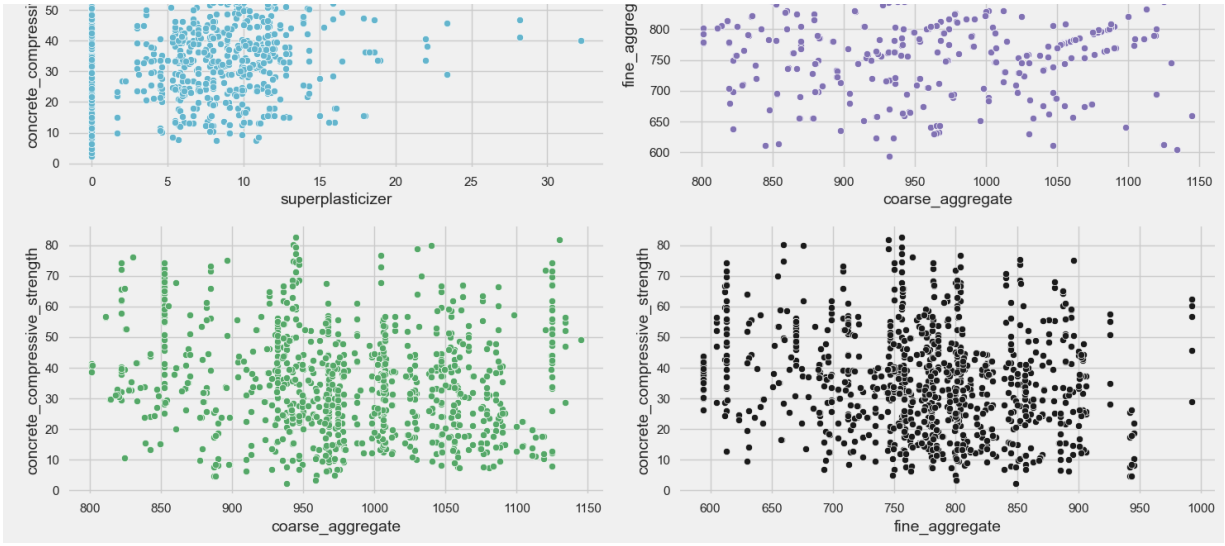
	Nullpercent	NuniquePercent	dtype	Nuniques	Nulls	nu categ
cement	0.000000	26.990291	float64	278	0	
blast_furnace_slag	0.000000	17.961165	float64	185	0	
fly_ash	0.000000	15.145631	float64	156	0	
water	0.000000	18.932039	float64	195	0	
superplasticizer	0.000000	10.776699	float64	111	0	
coarse_aggregate	0.000000	27.572816	float64	284	0	
fine_aggregate	0.000000	29.320388	float64	302	0	
age	0.000000	1.359223	int64	14	0	
concrete_compressive_strength	0.000000	82.038835	float64	845	0	

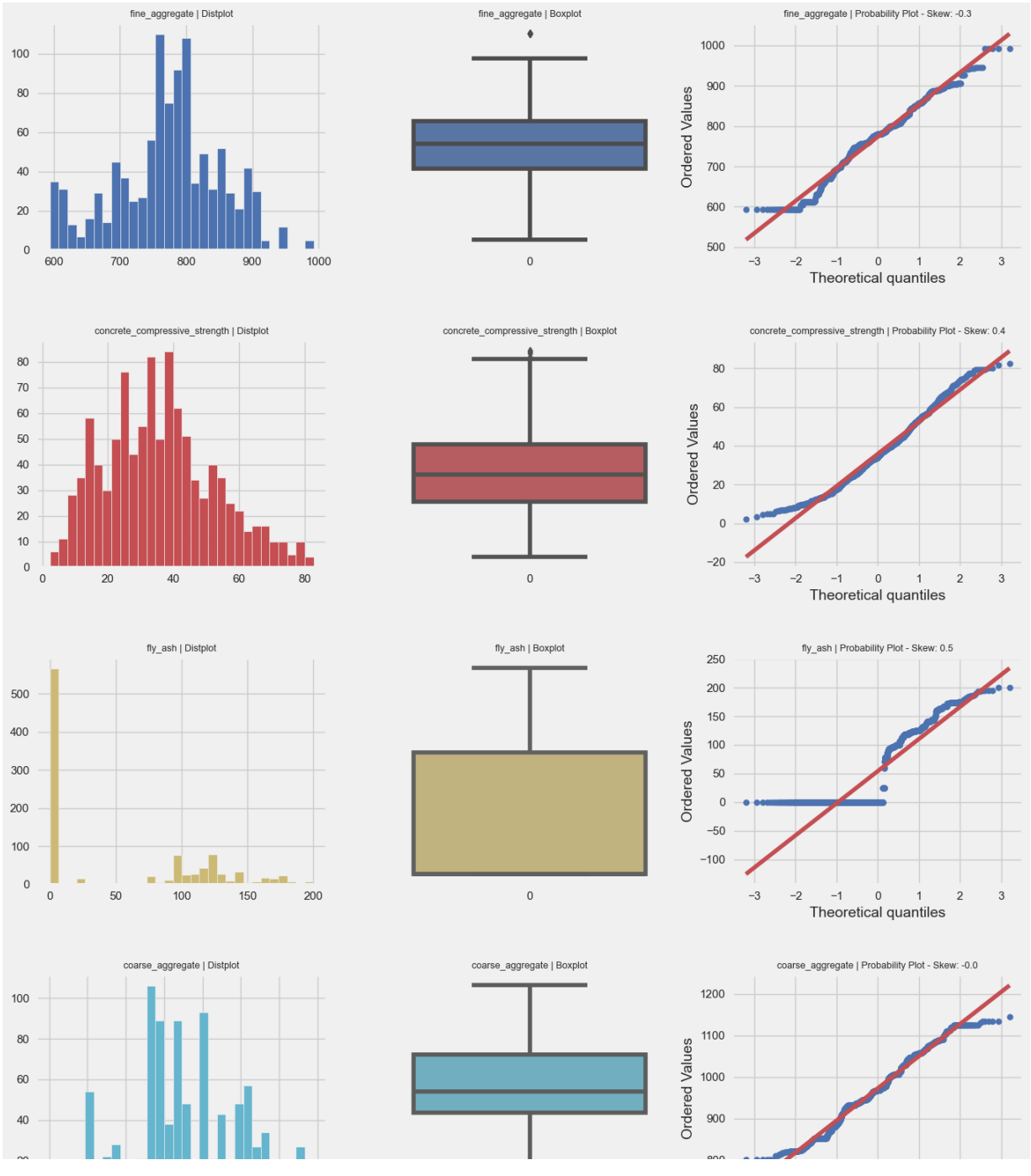
```
    9 Predictors classified...
        No variables removed since no ID or low-information variables found in data
set
Number of All Scatter Plots = 36
<Figure size 1000x500 with 0 Axes>
```

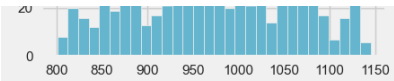



cement
water
superplasticizer
coarse_aggregate
fine_aggregate
concrete_compressive_strength
fly_ash
blast_furnace_slag

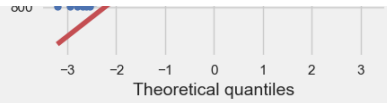

concrete_com
blast_furnace_slag
fly_ash
superplasticizer
coarse_aggregate
fine_aggregate
concrete_compressive_strength
water
ve_strength
ate

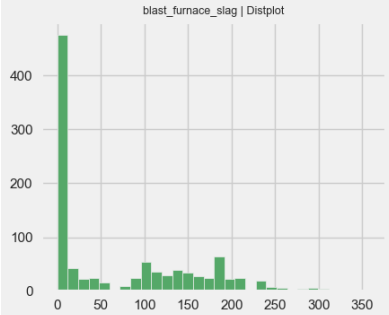

concrete_compressi
superplasticizer
fine_aggreg
coarse_aggregate
concrete_compressive_strength
coarse_aggregate
concrete_compressive_strength
fine_aggregate

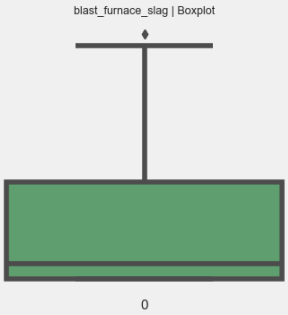

fine_aggregate | Distplot
fine_aggregate | Boxplot
fine_aggregate | Probability Plot - Skew: -0.3
Ordered Values
Theoretical quantiles
concrete_compressive_strength | Distplot
concrete_compressive_strength | Boxplot
concrete_compressive_strength | Probability Plot - Skew: 0.4
Ordered Values
Theoretical quantiles
fly_ash | Distplot
fly_ash | Boxplot
fly_ash | Probability Plot - Skew: 0.5
Ordered Values
Theoretical quantiles
coarse_aggregate | Distplot
coarse_aggregate | Boxplot
coarse_aggregate | Probability Plot - Skew: -0.0
Ordered Values

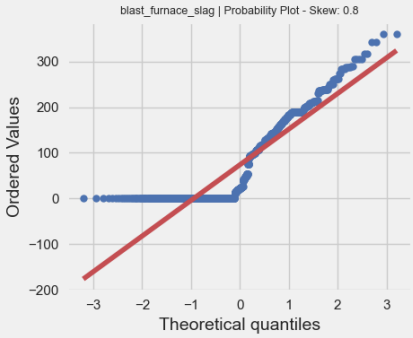

0
800 850 900 950 1000 1050 1100 1150

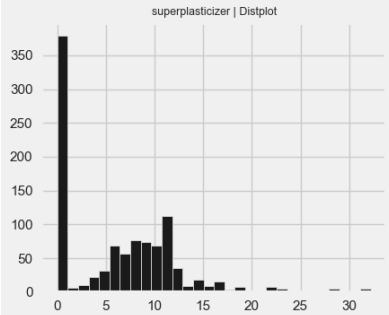

0

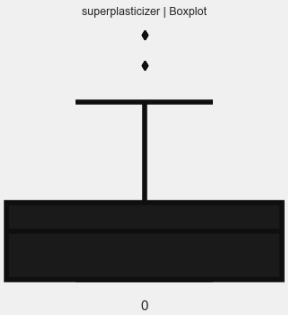

−3 −2 −1 0 1 2 3
Theoretical quantiles

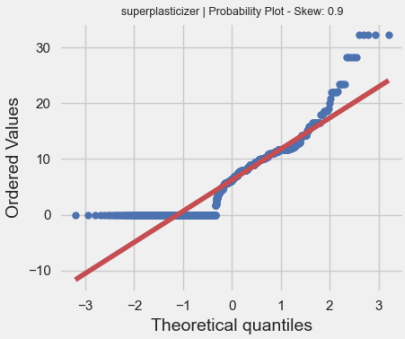

blast_furnace_slag | Distplot
0 100 200 300 400
0 50 100 150 200 250 300 350

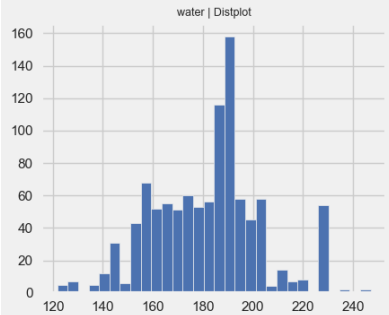

blast_furnace_slag | Boxplot
0

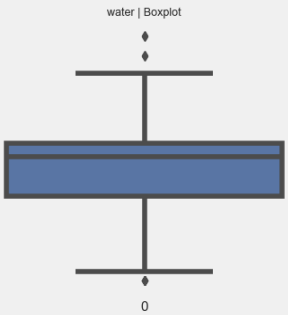

blast_furnace_slag | Probability Plot - Skew: 0.8
Ordered Values
300 200 100 0 −100 −200
−3 −2 −1 0 1 2 3
Theoretical quantiles

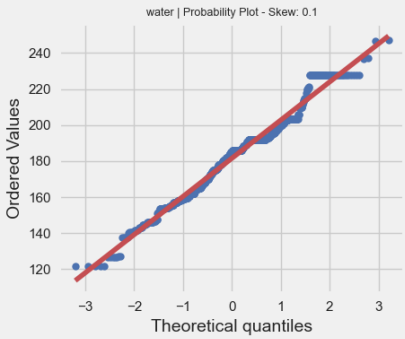

superplasticizer | Distplot
0 50 100 150 200 250 300 350
0 5 10 15 20 25 30

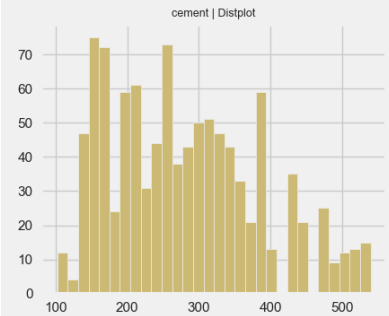

superplasticizer | Boxplot
0

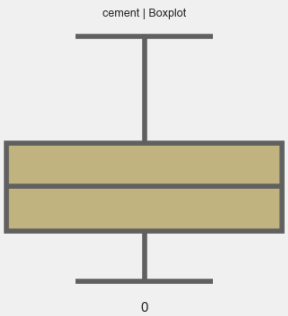

superplasticizer | Probability Plot - Skew: 0.9
Ordered Values
30 20 10 0 −10
−3 −2 −1 0 1 2 3
Theoretical quantiles

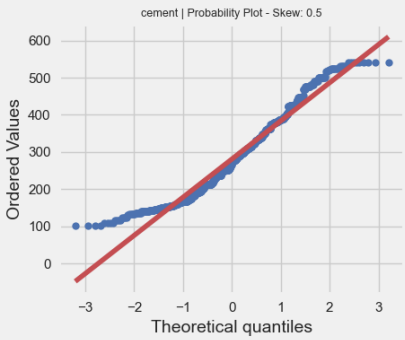

water | Distplot
0 20 40 60 80 100 120 140 160
120 140 160 180 200 220 240

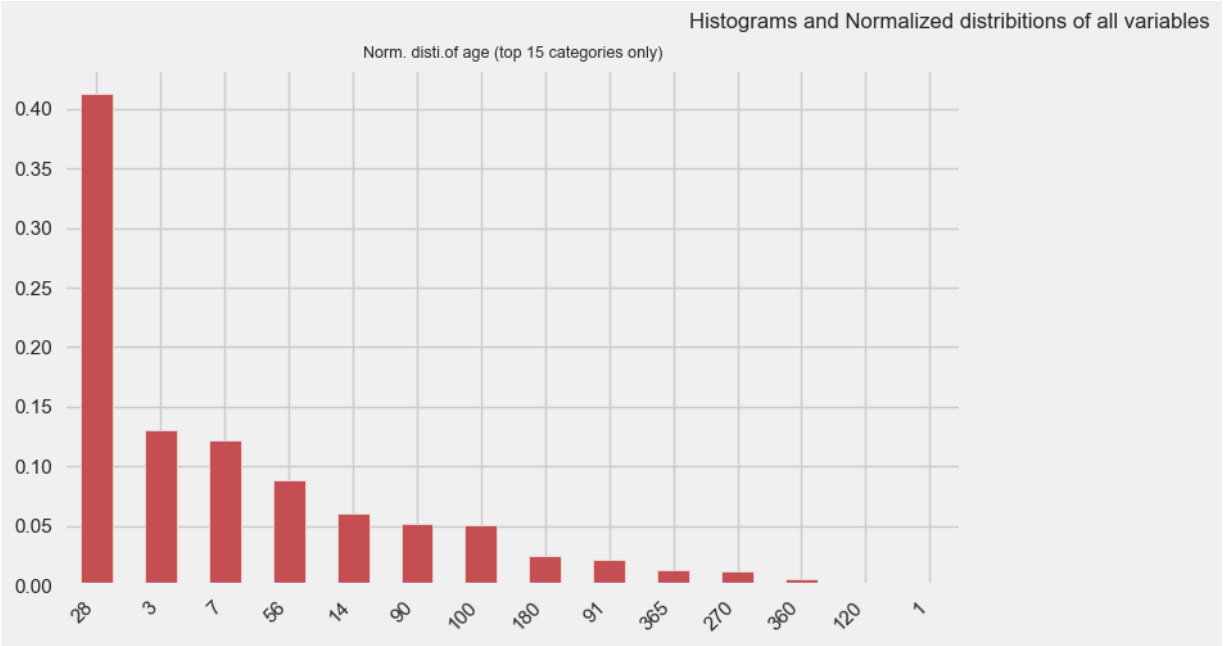

water | Boxplot
0

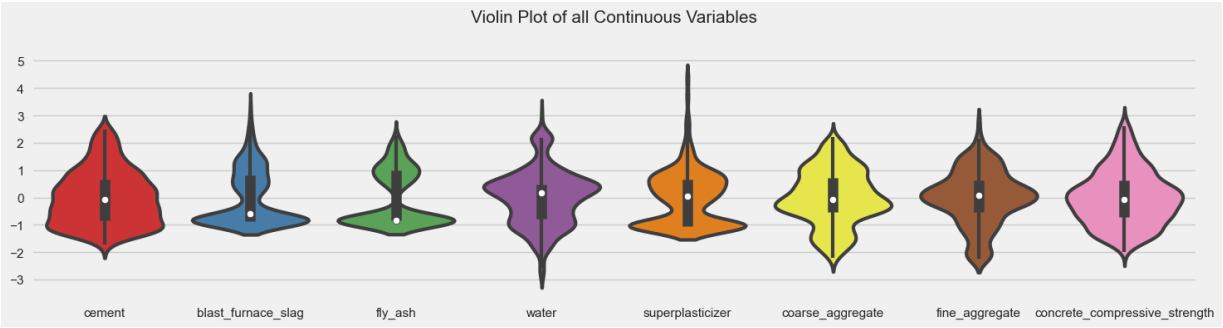

water | Probability Plot - Skew: 0.1
Ordered Values
240 220 200 180 160 140 120
−3 −2 −1 0 1 2 3
Theoretical quantiles

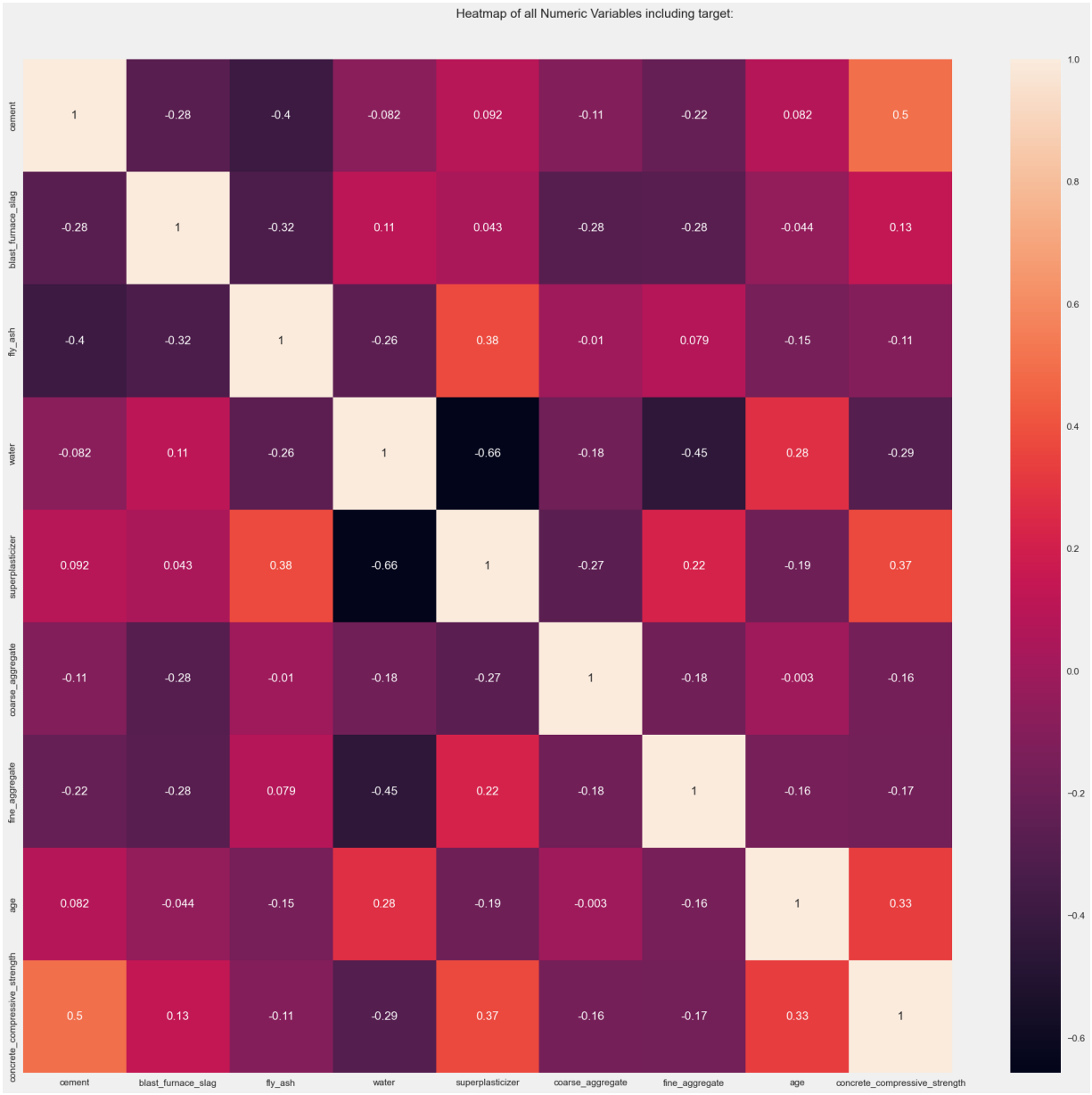

cement | Distplot
0 10 20 30 40 50 60 70
100 200 300 400 500


cement | Boxplot
0


cement | Probability Plot - Skew: 0.5
Ordered Values
600 500 400 300 200 100 0
−3 −2 −1 0 1 2 3
Theoretical quantiles


Histograms and Normalized distribitions of all variables
Norm. disti.of age (top 15 categories only)
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
28
3
7
56
14
90
100
180
91
365
270
360
120
1


Violin Plot of all Continuous Variables
5
4
3
2
1
0
−1
−2
−3
cement
blast_furnace_slag
fly_ash
water
superplasticizer
coarse_aggregate
fine_aggregate
concrete_compressive_strength


Heatmap of all Numeric Variables including target:
cement
blast_furnace_slag
fly_ash
water
superplasticizer
coarse_aggregate
fine_aggregate
age
concrete_compressive_strength
1 -0.28 -0.4 -0.082 0.092 -0.11 -0.22 0.082 0.5
-0.28 1 -0.32 0.11 0.043 -0.28 -0.28 -0.044 0.13
-0.4 -0.32 1 -0.26 0.38 -0.01 0.079 -0.15 -0.11
-0.082 0.11 -0.26 1 -0.66 -0.18 -0.45 0.28 -0.29
0.092 0.043 0.38 -0.66 1 -0.27 0.22 -0.19 0.37
-0.11 -0.28 -0.01 -0.18 -0.27 1 -0.18 -0.003 -0.16
-0.22 -0.28 0.079 -0.45 0.22 -0.18 1 -0.16 -0.17
0.082 -0.044 -0.15 0.28 -0.19 -0.003 -0.16 1 0.33
0.5 0.13 -0.11 -0.29 0.37 -0.16 -0.17 0.33 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6


```
All Plots done
Time to run AutoViz = 20 seconds
```

```
###################### AUTO VISUALIZATION Completed ########################
```

After analyzing the dataset, several observations were made:

1. Duplicate Rows: The dataset contains 11 duplicate rows, accounting for 1.1% of the total data. It is recommended to handle these duplicates to ensure accurate analysis.

2. Correlation: There is a strong positive correlation between water and superplasticizer variables. Similarly, age shows a high correlation with concrete compressive strength. These relationships indicate that changes in one variable may significantly impact the other.

3. Zeros: The variables blast_furnace_slag, fly_ash, and superplasticizer have a considerable number of zeros. These zeros may have implications for the analysis, and further investigation is required to understand their significance.

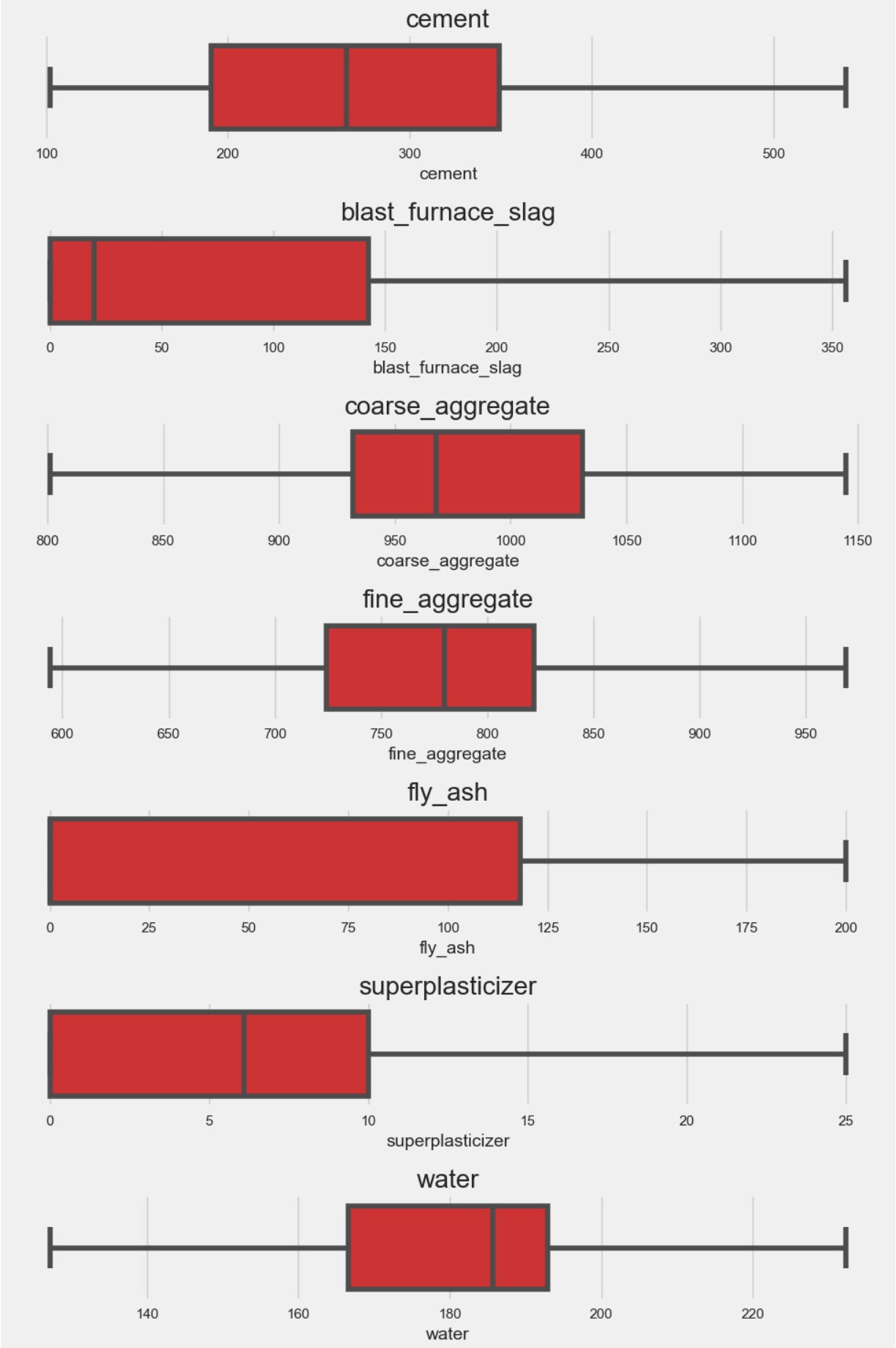
4. Outliers: The box plot analysis revealed the presence of outliers in the dataset. These outliers represent data points that significantly deviate from the majority of the data and may warrant further investigation to determine their impact on the analysis.

Considering these findings, further data preprocessing steps, such as handling duplicates, addressing zero values, and outlier treatment, should be performed to ensure the accuracy and reliability of the analysis.

In [9]:
```
# remove duplicate rows

df = df.drop_duplicates()
```

Treating Outliers

In [16]:
```
from feature_engine.outliers import Winsorizer

# Select the features to apply Winsorization
features = ['cement', 'blast_furnace_slag', 'coarse_aggregate', 'fine_aggregate', '

# Create the Winsorizer transformer
winsorizer = Winsorizer(capping_method='iqr', tail='both', fold=1.5, variables=feat

# Fit and transform the data
df[features] = winsorizer.fit_transform(df[features])

# Set the size of the figure
plt.figure(figsize=(10, 15))

# For each feature, create a subplot and draw a boxplot
for i, feature in enumerate(features, 1):
    plt.subplot(len(features), 1, i)
    sns.boxplot(x=df[feature])
    plt.title(feature)

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



In [17]:
```
# check outliers in each feature
```

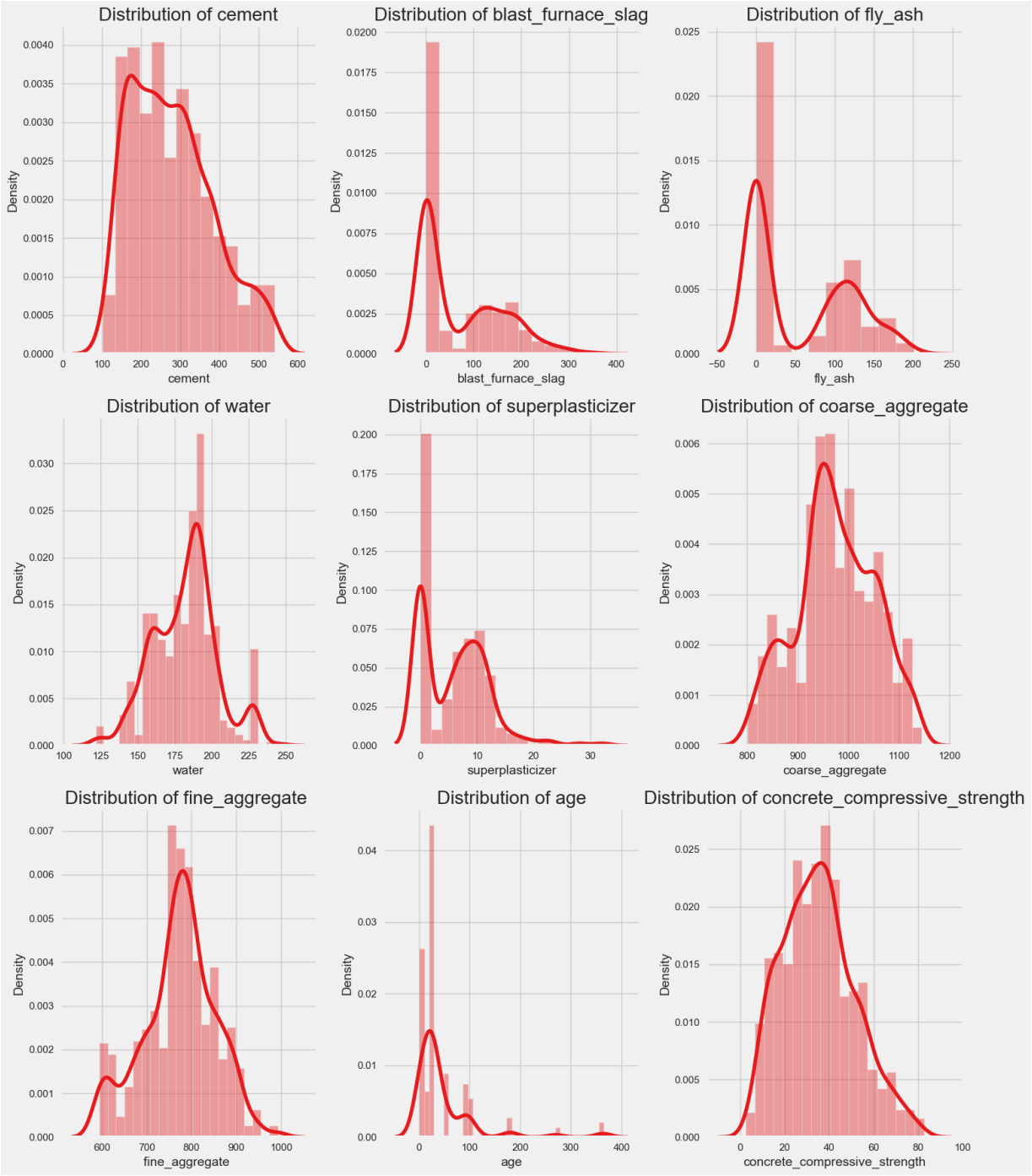
```
features = ['cement', 'blast_furnace_slag', 'fly_ash', 'water', 'superplasticizer',

# Calculate the lower and upper fences for outliers
for feature in features:
    Q1 = df[feature].quantile(0.25)
    Q3 = df[feature].quantile(0.75)
    IQR = Q3 - Q1
    lower_fence = Q1 - 1.5 * IQR
    upper_fence = Q3 + 1.5 * IQR

    # Count the number of outliers below the lower fence
    lower_outliers_count = df[df[feature] < lower_fence].shape[0]

    # Count the number of outliers above the upper fence
    upper_outliers_count = df[df[feature] > upper_fence].shape[0]

    print("Feature:", feature)
    print("Number of Lower Outliers:", lower_outliers_count)
    print("Number of Upper Outliers:", upper_outliers_count)
    print("----------------------------------------")#
```

```
Feature: cement
Number of Lower Outliers: 0
Number of Upper Outliers: 0
----------------------------------------
Feature: blast_furnace_slag
Number of Lower Outliers: 0
Number of Upper Outliers: 0
----------------------------------------
Feature: fly_ash
Number of Lower Outliers: 0
Number of Upper Outliers: 0
----------------------------------------
Feature: water
Number of Lower Outliers: 0
Number of Upper Outliers: 0
----------------------------------------
Feature: superplasticizer
Number of Lower Outliers: 0
Number of Upper Outliers: 0
----------------------------------------
Feature: coarse_aggregate
Number of Lower Outliers: 0
Number of Upper Outliers: 0
----------------------------------------
Feature: age
Number of Lower Outliers: 0
Number of Upper Outliers: 59
----------------------------------------
```

In [207…

```
features = ['cement', 'blast_furnace_slag', 'fly_ash', 'water', 'superplasticizer',

num_plots = len(features)
num_rows = num_plots // 3 + num_plots % 3
```

```
fig, axes = plt.subplots(nrows=num_rows, ncols=3, figsize=(15, 6 * num_rows))

for i, feature in enumerate(features):
    row = i // 3
    col = i % 3
    ax = axes[row, col]
    sns.distplot(df[feature], kde=True, ax=ax)
    ax.set_title(f"Distribution of {feature}")
    ax.set_xlabel(feature)
    ax.set_ylabel("Density")

# Remove any unused subplots
if num_plots % 3 != 0:
    for j in range(num_plots % 3, 3):
        fig.delaxes(axes[num_rows - 1, j])

plt.tight_layout()
plt.show()
```



Observations

After examining the data, several intriguing observations can be drawn regarding the distributions of the variables:

- Cement: The distribution appears to be almost normal, which indicates a balanced presence of this ingredient in the mixture across the dataset.
- Slag: The distribution of slag seems to exhibit three distinct peaks, or Gaussians, suggesting the presence of three groups within this variable. Additionally, the data is right-skewed, implying a significant number of higher-than-average values.

- Ash: The ash component also shows evidence of bimodality, with two distinct Gaussians, and is right-skewed. This suggests a substantial amount of observations have higher ash content.
- Water: The distribution of water showcases three Gaussians and is slightly left-skewed, indicating that the data contains a considerable number of observations with less water than the average.
- Superplastic: The distribution of this variable shows two clear Gaussians and is right-skewed, meaning there is a significant amount of mixtures with higher superplastic content.
- Coarse Aggregate: This variable displays three Gaussians, suggesting three groupings within the coarse aggregate content. The distribution appears to be approximately normal.
- Fine Aggregate: The fine aggregate content in the mixtures seems to follow a nearly normal distribution with a slight inclination towards two Gaussians, indicating a balanced spread of this variable.
- Age: The age variable is distinctly right-skewed, pointing towards a considerable number of observations having more days since the initial mixing. This variable also shows evidence of multiple Gaussians, suggesting the presence of various groups within the dataset based on the age.

These observations are instrumental in understanding the underlying structure and relationships within our data, informing our data preprocessing and model building steps.

Null Hypothesis (H0): The hypothesis that there is no significant difference or effect. In statistics, we usually assume the null hypothesis is true until we have enough evidence to reject it.

Alternative Hypothesis (Ha or H1): The hypothesis that there is a significant difference or effect. This is the hypothesis we are testing for, and it's considered as an alternative to the null hypothesis.

In the context of the Shapiro-Wilk test:

H0: "The data is drawn from a normal distribution."

Ha: "The data is not drawn from a normal distribution."

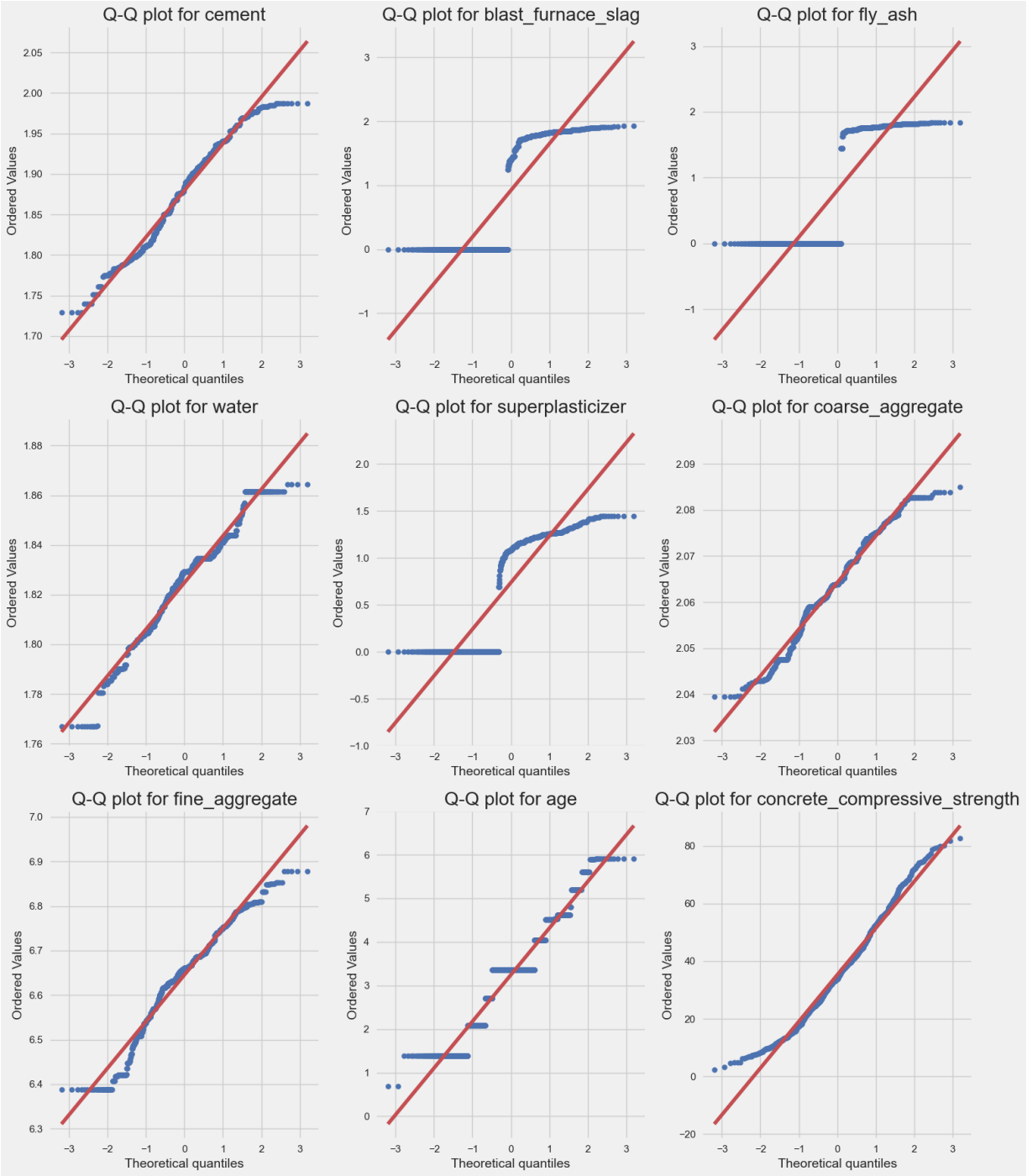
We use statistical tests to determine whether to reject the null hypothesis in favor of the alternative hypothesis. If the p-value is less than a chosen significance level (commonly 0.05), we reject the null hypothesis and conclude that the data is not normally distributed.

In [32]:
```
features = ['cement', 'blast_furnace_slag', 'fly_ash', 'water', 'superplasticizer',
            'coarse_aggregate', 'fine_aggregate', 'age', 'concrete_compressive_stre
```

```
num_plots = len(features)
num_rows = num_plots // 3 + num_plots % 3

fig, axes = plt.subplots(nrows=num_rows, ncols=3, figsize=(15, 6 * num_rows))

for i, feature in enumerate(features):
    row = i // 3
    col = i % 3
    ax = axes[row, col]

    data = df[feature]

    # perform Shapiro-Wilk test
    stat, p = stats.shapiro(data)

    # print test statistic and p-value
    print(f'Feature: {feature}')
    print('Test statistic =', stat)
    print('p-value =', p)

    if p > 0.05:
        print('Data appears to be normally distributed.\n')
    else:
        print('Data does not appear to be normally distributed.\n')

    # generate Q-Q plot in subplot
    stats.probplot(data, plot=ax)
    ax.set_title('Q-Q plot for ' + feature)

# Remove any unused subplots
if num_plots % 3 != 0:
    for j in range(num_plots % 3, 3):
        fig.delaxes(axes[num_rows - 1, j])

plt.tight_layout()
plt.show()
```

```
Feature: cement
Test statistic = 0.9779785871505737
p-value = 3.3278435562777986e-11
Data does not appear to be normally distributed.

Feature: blast_furnace_slag
Test statistic = 0.6973791718482971
p-value = 4.7765093857499655e-39
Data does not appear to be normally distributed.

Feature: fly_ash
Test statistic = 0.6571615934371948
p-value = 7.958254238593501e-41
Data does not appear to be normally distributed.

Feature: water
Test statistic = 0.9717737436294556
p-value = 4.563163540603765e-13
```

```
Data does not appear to be normally distributed.

Feature: superplasticizer
Test statistic = 0.7259781360626221
p-value = 1.1599685243569543e-37
Data does not appear to be normally distributed.

Feature: coarse_aggregate
Test statistic = 0.9790487289428711
p-value = 7.527087286796075e-11
Data does not appear to be normally distributed.

Feature: fine_aggregate
Test statistic = 0.9643646478652954
p-value = 5.79748566726301e-15
Data does not appear to be normally distributed.

Feature: age
Test statistic = 0.9258618950843811
p-value = 7.495046977684079e-22
Data does not appear to be normally distributed.

Feature: concrete_compressive_strength
Test statistic = 0.9817420244216919
p-value = 6.638498084576838e-10
Data does not appear to be normally distributed.
```

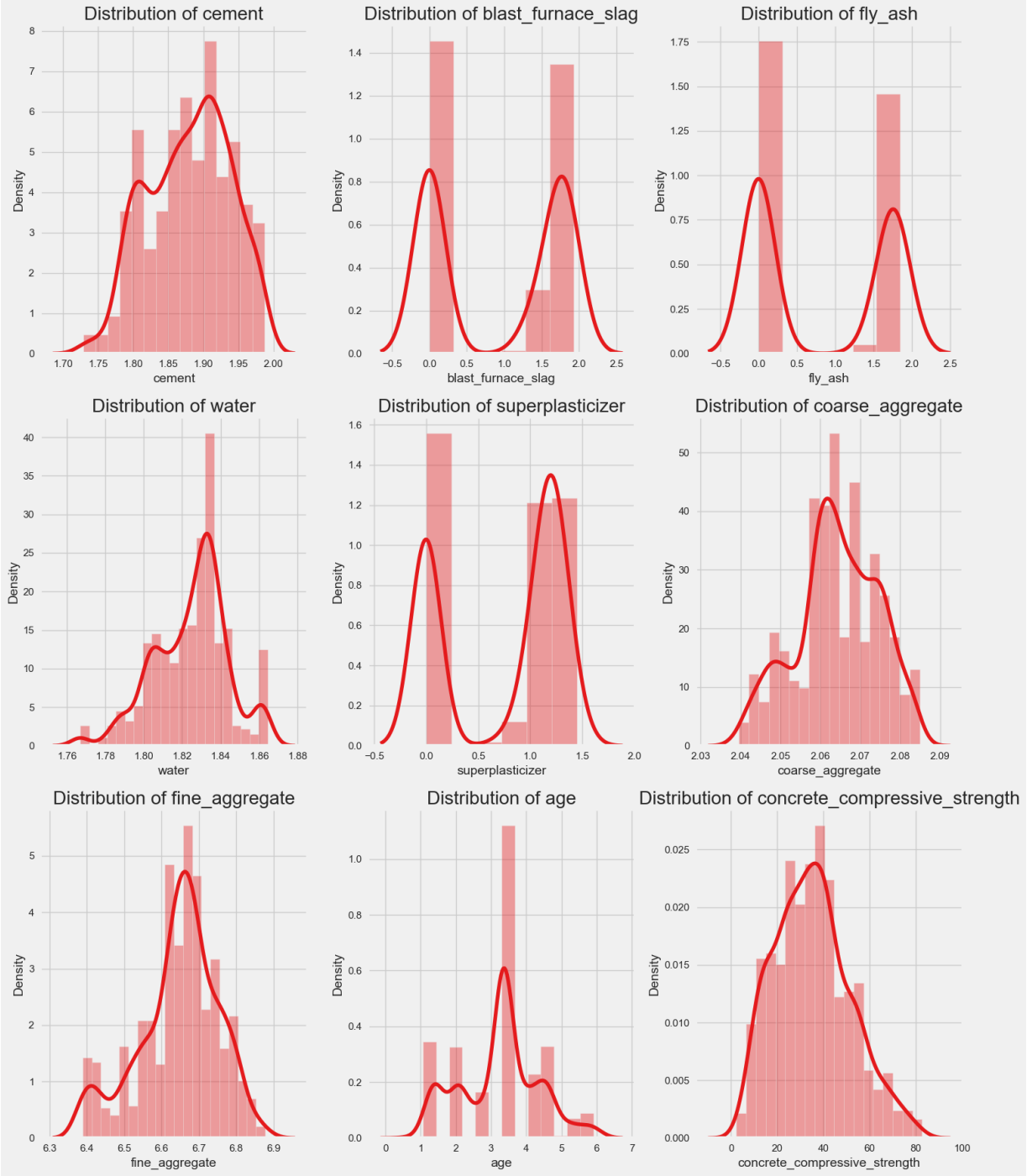


Applying log transforamtion

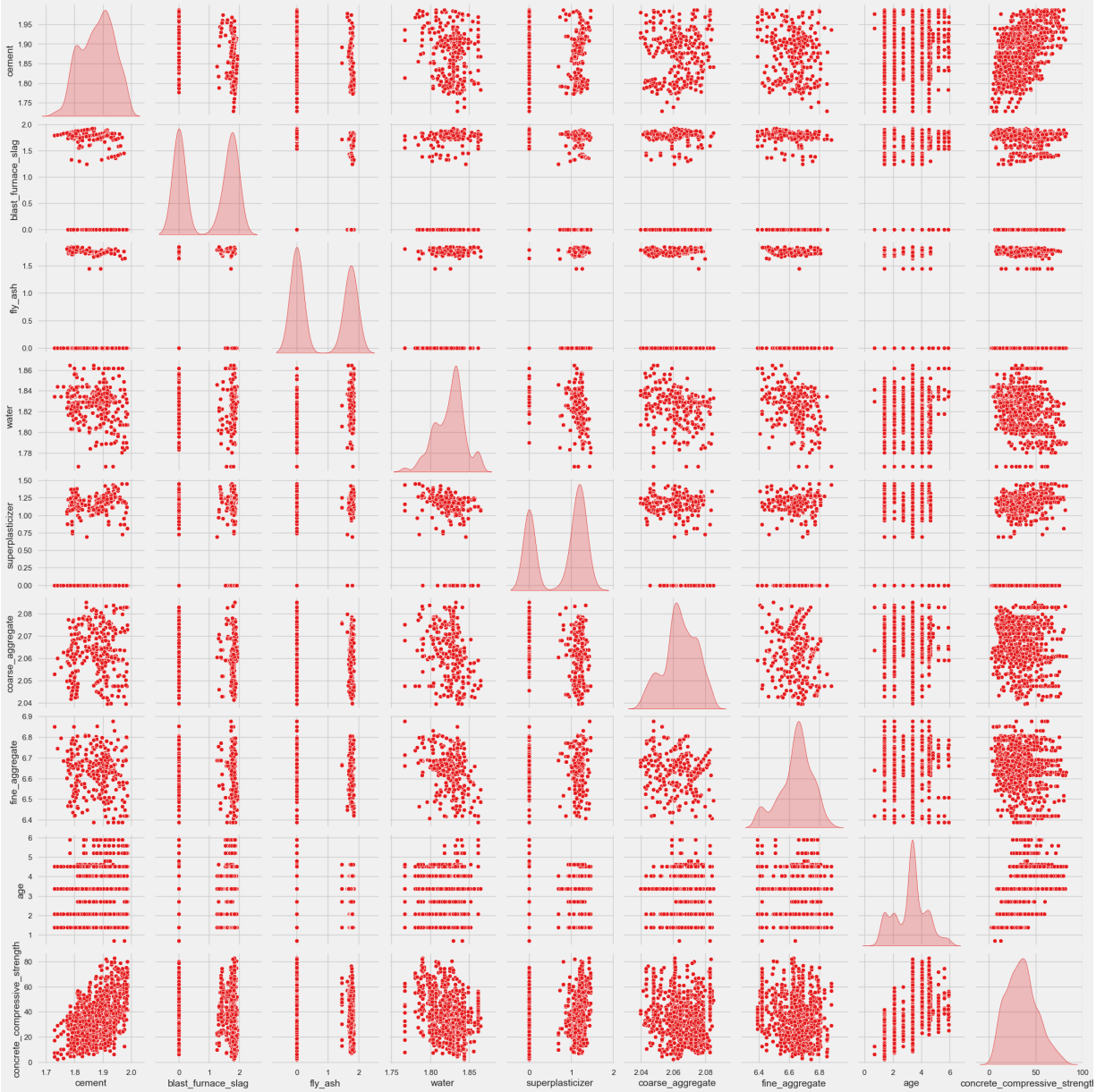
In [19]:
```
# Specify the features to apply log transformation
features = ['cement', 'blast_furnace_slag', 'fly_ash', 'water', 'superplasticizer',
       'coarse_aggregate', 'fine_aggregate', 'age'
       ]

# Apply log transformation to the selected features
for feature in features:
    df[feature] = np.log1p(df[feature])
```

In [20]:
```
features = ['cement', 'blast_furnace_slag', 'fly_ash', 'water', 'superplasticizer',
```

```
num_plots = len(features)
num_rows = num_plots // 3 + num_plots % 3

fig, axes = plt.subplots(nrows=num_rows, ncols=3, figsize=(15, 6 * num_rows))

for i, feature in enumerate(features):
    row = i // 3
    col = i % 3
    ax = axes[row, col]
    sns.distplot(df[feature], kde=True, ax=ax)
    ax.set_title(f"Distribution of {feature}")
    ax.set_xlabel(feature)
    ax.set_ylabel("Density")

# Remove any unused subplots
if num_plots % 3 != 0:
    for j in range(num_plots % 3, 3):
        fig.delaxes(axes[num_rows - 1, j])

plt.tight_layout()
plt.show()
```



In [21]:
```
sns.pairplot(df, diag_kind='kde')
plt.show()
```



Diagonal Analysis

The diagonal of the correlation matrix provides insight into the distribution of each individual attribute:

- Cement: Displays an almost normal curve.
- Slag: Shows right skewness with two Gaussians and presents potential outliers.
- Ash: Exhibits right skewness with two Gaussians, also indicating possible outliers.
- Water: Presents multiple Gaussians, slightly left-skewed, with potential outliers.
- Superplastic: Indicates right skewness with multiple Gaussians and possible outliers.
- Coarse Aggregate: Shows a nearly normal distribution with three Gaussians.
- Fine Aggregate: Appears almost normal with an inclination towards two Gaussians.
- Age: Demonstrates right skewness with multiple Gaussians and potential outliers.
- Strength: Shows a distribution close to a normal curve.

The presence of outliers is an important observation in this analysis. Along with missing values, outliers can significantly influence the behavior of our predictive models.

Off-Diagonal Analysis: Relationship Between Independent Attributes

Examining scatter plots of various independent attributes against each other reveals:

- Cement, Slag, Ash, Coarse Aggregate: These attributes don't demonstrate any significant correlation with other variables - their data points disperse like a cloud, suggesting a correlation coefficient close to zero.
- Water: Exhibits a negative linear relationship with Superplastic and Fine Aggregate, but no significant correlation with other variables. The negative correlation with Superplastic is expected, as superplasticizers can reduce the water content in the mixture by up to 30% without affecting workability.
- Superplastic: Shows a negative linear relationship with Water only, but no significant correlation with other variables.
- Fine Aggregate: Exhibits a negative linear relationship with Water but no significant correlation with other variables.

The correlation analysis is critical as highly correlated variables (r close to 1 or -1) can introduce redundancy in our data. In such cases, we might prefer to keep one variable and drop the other, or create a composite dimension that combines these variables.

Relationship Between Dependent and Independent Attributes

When examining the relationship between the dependent variable, Strength, and independent attributes, we note the following:

- Strength vs. Cement: There is a positive but weak linear relationship. Multiple Strength values correspond to a given Cement value, making Cement a weak predictor for Strength.
- Strength vs. Slag, Ash, Age, Superplastic: No distinct trend or strong correlation is observed.
- Strength vs. Other Attributes: No other attributes show a strong linear relationship with Strength.

Consequently, none of the independent attributes appear to be strong predictors for the Strength attribute, indicating no evident linear relationship. This observation is crucial in understanding the limitations of our data for predictive modeling.

Splitting the data

In [22]:
```
X = df.drop(['concrete_compressive_strength'], axis=1)
y = df['concrete_compressive_strength']

# train test split the data

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2, random_stat

# Print the shapes of the train and test sets
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (804, 8)
y_train shape: (804,)
X_test shape: (201, 8)
y_test shape: (201,)
```

Linear Models

Exploring Linear Model Predictions Despite Non-Linear Relationships

In [35]:
```
# Create a LinearRegression model
linear_model = LinearRegression()

# Fit the model on the training data
linear_model.fit(X_train, y_train)

# Make predictions on the training and test sets
linear_ypred_train = linear_model.predict(X_train)
linear_ypred_test = linear_model.predict(X_test)

# Calculate the RMSE and R2 score for the test set
linear_rmse_test = mean_squared_error(y_test, linear_ypred_test, squared=False)
linear_r2_test = r2_score(y_test, linear_ypred_test)
linear_r2_train = r2_score(y_train, linear_ypred_train)
# Perform k-fold cross-validation
k = 5
kfold_linear = KFold(n_splits=k, random_state=42, shuffle=True)
cv_linear = cross_val_score(linear_model, X, y, cv=kfold_linear, scoring='r2')

# Print the results
print("Linear Regression (Train) - R^2:", linear_r2_train)
print("Linear Regression (Test) - R^2:", linear_r2_test)
print("Linear Regression (Test) - RMSE:", linear_rmse_test)
print("Linear Regression CV Score Mean (R^2):", cv_linear.mean())
```

```
Linear Regression (Train) - R^2: 0.7925995842681175
Linear Regression (Test) - R^2: 0.7900841279607375
```

```
Linear Regression (Test) - RMSE: 7.34931299244433
Linear Regression CV Score Mean (R^2): 0.789028441790786
```

In [36]:
```python
# L1 (Lasso) Regression

lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X_train, y_train)
lasso_ypred_train = lasso_model.predict(X_train)
lasso_ypred_test = lasso_model.predict(X_test)
lasso_r2_train = r2_score(y_train, lasso_ypred_train)
lasso_r2_test = r2_score(y_test, lasso_ypred_test)
lasso_coeffs = lasso_model.coef_

print("Lasso Regression (Train) - R^2:", lasso_r2_train)
print("Lasso Regression (Test) - R^2:", lasso_r2_test)
print("Lasso Coeffecients : ", lasso_coeffs)


# L2 (Ridge) Regression
ridge_model = Ridge(alpha=0.1)  # Adjust alpha as needed
ridge_model.fit(X_train, y_train)
ridge_predictions_train = ridge_model.predict(X_train)
ridge_predictions_test = ridge_model.predict(X_test)
ridge_rmse_test = mean_squared_error(y_test, ridge_predictions_test, squared=False)
ridge_r2_train = r2_score(y_train, ridge_predictions_train)
ridge_r2_test = r2_score(y_test, ridge_predictions_test)

# Elastic Net Regression
elastic_model = ElasticNet(alpha=0.1, l1_ratio=0.5)  # Adjust alpha and l1_ratio as
elastic_model.fit(X_train, y_train)
elastic_predictions_train = elastic_model.predict(X_train)
elastic_predictions_test = elastic_model.predict(X_test)
elastic_rmse_test = mean_squared_error(y_test, elastic_predictions_test, squared=Fa
elastic_r2_train = r2_score(y_train, elastic_predictions_train)
elastic_r2_test = r2_score(y_test, elastic_predictions_test)



print("Ridge Regression (Train) - R^2:", ridge_r2_train)
print("Ridge Regression (Test) - R^2:", ridge_r2_test)
print("Ridge Regression (Test) - RMSE:", ridge_rmse_test)
print("Elastic Net Regression (Train) - R^2:", elastic_r2_train)
print("Elastic Net Regression (Test) - R^2:", elastic_r2_test)
print("Elastic Net Regression (Test) - RMSE:", elastic_rmse_test)
```

```
Lasso Regression (Train) - R^2: 0.7361357887828031
Lasso Regression (Test) - R^2: 0.7483985712589172
Lasso Coeffecients :  [117.62391316   4.20804023  -2.51811787  -0.          11.59923
562
   0.          -0.           8.40388789]
Ridge Regression (Train) - R^2: 0.7859569375831417
Ridge Regression (Test) - R^2: 0.783103843405847
Ridge Regression (Test) - RMSE: 7.470506242015733
Elastic Net Regression (Train) - R^2: 0.5351507466786456
Elastic Net Regression (Test) - R^2: 0.5601327202340052
Elastic Net Regression (Test) - RMSE: 10.638611104911842
```

In [37]:
```
# Polynomial Regression
degree = 3  # Adjust the degree as needed
poly_features = PolynomialFeatures(degree=degree, include_bias=False)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)

poly_predictions_train = poly_model.predict(X_train_poly)  # Generate predictions o
poly_predictions = poly_model.predict(X_test_poly)

poly_rmse = mean_squared_error(y_test, poly_predictions, squared=False)
poly_r2_test = r2_score(y_test, poly_predictions)
poly_r2_train = r2_score(y_train, poly_predictions_train)  # Compute R^2 score on t


# K Fold Cross Validation

X_poly = poly_features.fit_transform(X)

k = 5
kfold_ploy = KFold(n_splits=k, random_state= 42, shuffle=True)
CV_score_poly = cross_val_score(poly_model,X_poly,y, scoring='r2', cv=kfold_ploy)



# Print the evaluation metrics
print("Polynomial Regression (Degree", degree, ") - RMSE:", poly_rmse)
print("Polynomial Regression (Degree", degree, ") - Train - R^2:", poly_r2_train)
print("Polynomial Regression (Degree", degree, ") - Test - R^2:", poly_r2_test)
print("CV_Score : ",CV_score_poly.mean())
```

```
Polynomial Regression (Degree 3 ) - RMSE: 4.955658467496733
Polynomial Regression (Degree 3 ) - Train - R^2: 0.9484633099482782
Polynomial Regression (Degree 3 ) - Test - R^2: 0.9045547045211716
CV_Score :  0.8035154458855788
```

In [38]:
```
degree = 3

# Create polynomial features
poly_features = PolynomialFeatures(degree=degree, include_bias=False)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

# Create Lasso regression model with L1 regularization
lasso_model = Lasso(alpha=0.1, max_iter=1000)  # Adjust the alpha and max_iter valu
lasso_model.fit(X_train_poly, y_train)
lasso_predictions = lasso_model.predict(X_test_poly)
lasso_rmse = mean_squared_error(y_test, lasso_predictions, squared=False)
lasso_r2 = r2_score(y_test, lasso_predictions)

# Create Ridge regression model with L2 regularization
ridge_model = Ridge(alpha=0.1)  # Adjust the alpha value as needed
ridge_model.fit(X_train_poly, y_train)
ridge_predictions = ridge_model.predict(X_test_poly)
```

```
ridge_rmse = mean_squared_error(y_test, ridge_predictions, squared=False)
ridge_r2 = r2_score(y_test, ridge_predictions)

# Perform k-fold cross-validation with Lasso and Ridge models
k = 5
kfold = KFold(n_splits=k, shuffle=True, random_state=42)

lasso_cv_scores = cross_val_score(lasso_model, X_train_poly, y_train, scoring='r2',
ridge_cv_scores = cross_val_score(ridge_model, X_train_poly, y_train, scoring='r2',

# Print the evaluation metrics
print("Polynomial Regression (Degree", degree, ") - RMSE (Lasso):", lasso_rmse)
print("Polynomial Regression (Degree", degree, ") - R^2 (Lasso):", lasso_r2)
print("Polynomial Regression (Degree", degree, ") - RMSE (Ridge):", ridge_rmse)
print("Polynomial Regression (Degree", degree, ") - R^2 (Ridge):", ridge_r2)
print("Lasso Regression CV Score:", lasso_cv_scores.mean())
print("Ridge Regression CV Score:", ridge_cv_scores.mean())
```

```
Polynomial Regression (Degree 3 ) - RMSE (Lasso): 6.74448381429475
Polynomial Regression (Degree 3 ) - R^2 (Lasso): 0.8232134486282252
Polynomial Regression (Degree 3 ) - RMSE (Ridge): 6.206340299512778
Polynomial Regression (Degree 3 ) - R^2 (Ridge): 0.8502996003362743
Lasso Regression CV Score: 0.8399145021762797
Ridge Regression CV Score: 0.8747372732055847
```

Exploring Advanced Regression Methods and Ensemble Learning

Having explored linear models and polynomial regression, it is now apparent that our dataset requires more complex predictive modeling. The polynomial model displayed some promise but was prone to overfitting, signifying it may not generalize well to unseen data. Therefore, we will experiment with other types of regression methods that can better handle the complexity and non-linearity of the data.

Firstly, we'll explore Gradient Descent-based regression models like XGBoost which is renowned for its effectiveness and speed, while offering great flexibility through various tunable parameters. It also has built-in regularization to prevent overfitting.

In addition, we'll investigate tree-based models like Decision Trees and Random Forests. These models can capture non-linear relationships and interactions between variables well, without needing explicit feature engineering.

We'll also experiment with Support Vector Machines (SVM), a powerful model capable of performing linear and non-linear regression. It uses the kernel trick to transform data, thus making it effective in high-dimensional spaces.

Lastly, we'll explore Hybrid Models or Ensemble methods where we combine the predictions from multiple models. This approach can potentially yield more robust and accurate predictions, leveraging the strengths of each individual model while mitigating their respective weaknesses.

In [39]:
```
#XG boost

xgb_model = xgb.XGBRegressor(objective ='reg:squarederror', n_estimators=100)



# Fit the model
xgb_model.fit(X_train, y_train)

# Make predictions
xgb_ypred_train = xgb_model.predict(X_train)
xgb_ypred_test = xgb_model.predict(X_test)

# Calculate metrics
xgb_rmse_test = mean_squared_error(y_test, xgb_ypred_test, squared=False)
xgb_r2_train = r2_score(y_train, xgb_ypred_train)
xgb_r2_test = r2_score(y_test, xgb_ypred_test)

# Perform  k fold cross-validation on XGBoost Regression
k = 5
kfold_XG = KFold(n_splits=k, random_state= 42, shuffle=True)
CV_score_XG = cross_val_score(xgb_model,X,y, scoring='r2', cv=kfold_XG)



print("XGBoost Regression (Train) - R^2:", xgb_r2_train)
print("XGBoost Regression (Test) - R^2:", xgb_r2_test)
print("XGBoost Regression (Test) - RMSE:", xgb_rmse_test)
print("XGBoost Regression CV Score :", CV_score_XG.mean())
```

```
XGBoost Regression (Train) - R^2: 0.9960415721489488
XGBoost Regression (Test) - R^2: 0.9311340549257805
XGBoost Regression (Test) - RMSE: 4.209459744802385
XGBoost Regression CV Score : 0.932439151630567
```

Hyperparmeter Tuning for best Parameters

In [217…
```
from sklearn.model_selection import GridSearchCV

# Define a parameter grid
param_grid = {
    'alpha': [0.001, 0.01, 0.1, 1, 10],
    'lambda': [0.001, 0.01, 0.1, 1, 10],
    'gamma': [0.001, 0.01, 0.1, 1, 10],
    'n_estimators': [50, 100],
    'max_depth': [2, 4, 6]
}

# Initialize an XGBoost Regressor
xgb_model = xgb.XGBRegressor(objective='reg:squarederror')

# Initialize the GridSearchCV object
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=5, n_jobs

# Fit the GridSearchCV object to the data
```

```
grid_search.fit(X_train, y_train)

# Print the best parameters
print(grid_search.best_params_)
```

```
Fitting 5 folds for each of 750 candidates, totalling 3750 fits
{'alpha': 0.01, 'gamma': 0.01, 'lambda': 1, 'max_depth': 6, 'n_estimators': 50}
```

In [218…
```
from sklearn.model_selection import RandomizedSearchCV

# Initialize the RandomizedSearchCV object
rand_search = RandomizedSearchCV(estimator=xgb_model, param_distributions=param_gri

# Fit the RandomizedSearchCV object to the data
rand_search.fit(X_train, y_train)

# Print the best parameters
print(rand_search.best_params_)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
{'n_estimators': 100, 'max_depth': 4, 'lambda': 1, 'gamma': 0.01, 'alpha': 0.001}
```

In [40]:
```
# remodeling according to Gridsearch CV
xgb_model = xgb.XGBRegressor(objective ='reg:squarederror', n_estimators=100, reg_a



# Fit the model
xgb_model.fit(X_train, y_train)

# Make predictions
xgb_ypred_train = xgb_model.predict(X_train)
xgb_ypred_test = xgb_model.predict(X_test)

# Calculate metrics
xgb_rmse_test = mean_squared_error(y_test, xgb_ypred_test, squared=False)
xgb_r2_train_remodel_grid = r2_score(y_train, xgb_ypred_train)
xgb_r2_test_remodel_grid = r2_score(y_test, xgb_ypred_test)

# Perform cross-validation on XGBoost Regression
k = 5
kfold_XG_remodel_grid = KFold(n_splits=k, random_state= 42, shuffle=True)
CV_score_XG_remodel_grid = cross_val_score(xgb_model,X,y, scoring='r2', cv=kfold_XG



print("XGBoost Regression (Train) - R^2:", xgb_r2_train_remodel_grid)
print("XGBoost Regression (Test) - R^2:", xgb_r2_test_remodel_grid)
print("XGBoost Regression (Test) - RMSE:", xgb_rmse_test)
print("XGBoost Regression CV Score :", CV_score_XG_remodel_grid.mean())
```

```
XGBoost Regression (Train) - R^2: 0.9823273202290895
XGBoost Regression (Test) - R^2: 0.9215236065330772
XGBoost Regression (Test) - RMSE: 4.493591845673046
XGBoost Regression CV Score : 0.9271543843681224
```

In [41]:
```
# remodeling according to randomsearch CV
```

```
xgb_model = xgb.XGBRegressor(objective ='reg:squarederror', n_estimators=100, reg_a



# Fit the model
xgb_model.fit(X_train, y_train)

# Make predictions
xgb_ypred_train = xgb_model.predict(X_train)
xgb_ypred_test = xgb_model.predict(X_test)

# Calculate metrics
xgb_rmse_test = mean_squared_error(y_test, xgb_ypred_test, squared=False)
xgb_r2_train_remodel_random = r2_score(y_train, xgb_ypred_train)
xgb_r2_test_remodel_random = r2_score(y_test, xgb_ypred_test)

# Perform cross-validation on XGBoost Regression
k = 5
kfold_XG_remodel_random = KFold(n_splits=k, random_state= 42, shuffle=True)
CV_score_XG_remodel_random = cross_val_score(xgb_model,X,y, scoring='r2', cv=kfold_


print("XGBoost Regression (Train) - R^2:", xgb_r2_train_remodel_random)
print("XGBoost Regression (Test) - R^2:", xgb_r2_test_remodel_random)
print("XGBoost Regression (Test) - RMSE:", xgb_rmse_test)
print("XGBoost Regression CV Score :", CV_score_XG_remodel_random.mean())
```

```
XGBoost Regression (Train) - R^2: 0.9828032275648817
XGBoost Regression (Test) - R^2: 0.918973439068376
XGBoost Regression (Test) - RMSE: 4.5660199836157345
XGBoost Regression CV Score : 0.9281685715158632
```

Implementation on Training, Validation and Test Sets

Building on the promising results from our chosen model, it is time to put it to a more rigorous test. We'll apply the model on our training, validation, and test datasets. The training data will be used to construct the model, while the validation set will allow us to tune our parameters without bias. Finally, the test data will offer an unbiased evaluation of the final model, demonstrating how well our model generalizes to unseen data. This three-way split approach further ensures that our model is robust, capable of learning effectively, and is not merely memorizing the training data.

In [42]:
```
# copy of df to df_copy
df_copy = df.copy()
```

In [43]:
```
Xcopy = df.drop(['concrete_compressive_strength'], axis=1)
ycopy = df['concrete_compressive_strength']

# Split the data into a temporary train set and a final test set
Xcopy_temp, Xcopy_test, ycopy_temp, ycopy_test = train_test_split(Xcopy, ycopy, tes
```

```
# Then split the temporary set into final train and validation sets
Xcopy_train, Xcopy_val, ycopy_train, ycopy_val = train_test_split(Xcopy_temp, ycopy

# Now we have training, validation, and test sets



print("Xcopy_train shape:", Xcopy_train.shape)
print("ycopy_train shape:", ycopy_train.shape)
print("Xcopy_val shape:", Xcopy_val.shape)
print("ycopy_val shape:", ycopy_val.shape)
print("Xcopy_test shape:", Xcopy_test.shape)
print("ycopy_test shape:", ycopy_test.shape)
```

```
Xcopy_train shape: (603, 8)
ycopy_train shape: (603,)
Xcopy_val shape: (201, 8)
ycopy_val shape: (201,)
Xcopy_test shape: (201, 8)
ycopy_test shape: (201,)
```

In [44]:

```
# Initialize the XGBoost Regressor with the selected parameters
xgb_model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=50, reg_alp

# Fit the model on the training set
xgb_model.fit(Xcopy_train, ycopy_train)

# Predict on training and validation sets
xgb_ypred_train = xgb_model.predict(Xcopy_train)
xgb_ypred_val = xgb_model.predict(Xcopy_val)

# Calculate metrics for the training and validation sets
xgb_rmse_val = mean_squared_error(ycopy_val, xgb_ypred_val, squared=False)
xgb_r2_train = r2_score(ycopy_train, xgb_ypred_train)
xgb_r2_val = r2_score(ycopy_val, xgb_ypred_val)

print("XGBoost Regression (Train) - R^2:", xgb_r2_train)
print("XGBoost Regression (Validation) - R^2:", xgb_r2_val)
print("XGBoost Regression (Validation) - RMSE:", xgb_rmse_val)

# Predict on the test set
xgb_ypred_test = xgb_model.predict(Xcopy_test)

# Calculate metrics for the test set
xgb_rmse_test = mean_squared_error(ycopy_test, xgb_ypred_test, squared=False)
xgb_r2_test = r2_score(ycopy_test, xgb_ypred_test)

print("XGBoost Regression (Test) - R^2:", xgb_r2_test)
print("XGBoost Regression (Test) - RMSE:", xgb_rmse_test)
```

```
XGBoost Regression (Train) - R^2: 0.9882951177169724
XGBoost Regression (Validation) - R^2: 0.9187064951900842
XGBoost Regression (Validation) - RMSE: 4.429901441824419
XGBoost Regression (Test) - R^2: 0.8938393352461491
XGBoost Regression (Test) - RMSE: 5.627643254142302
```

Evaluating Model with OLS and VIF

In the next stage of our analysis, we will leverage Ordinary Least Squares (OLS) and Variance Inflation Factor (VIF) for further model evaluation. OLS is a popular method for estimating the unknown parameters in a linear regression model, providing us with statistically robust results to interpret the significance of different features in our model.

On the other hand, VIF helps us quantify how much the variance (the square of the estimate's standard deviation) of an estimated regression coefficient is increased because of multicollinearity, ensuring that our model doesn't include redundant or highly correlated variables.

In [46]:
```
import statsmodels.formula.api as smf

model1 = smf.ols("y~X", data=df).fit()
model1.summary()
```

Out[46]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.793
Model:	OLS	**Adj. R-squared:**	0.791
Method:	Least Squares	**F-statistic:**	476.9
Date:	Sat, 03 Jun 2023	**Prob (F-statistic):**	0.00
Time:	01:32:15	**Log-Likelihood:**	-3438.3
No. Observations:	1005	**AIC:**	6895.
Df Residuals:	996	**BIC:**	6939.
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	494.4636	127.677	3.873	0.000	243.916	745.011
X[0]	136.7638	5.963	22.936	0.000	125.063	148.465
X[1]	4.8152	0.415	11.600	0.000	4.001	5.630
X[2]	-0.4810	0.445	-1.082	0.280	-1.354	0.392
X[3]	-270.7680	22.634	-11.963	0.000	-315.184	-226.352
X[4]	5.0038	0.763	6.558	0.000	3.506	6.501
X[5]	-68.9676	34.909	-1.976	0.048	-137.470	-0.465
X[6]	-17.4525	3.539	-4.932	0.000	-24.396	-10.509
X[7]	8.7599	0.215	40.662	0.000	8.337	9.183

Omnibus:	17.237	**Durbin-Watson:**	1.294

Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	26.551
Skew:	0.140	**Prob(JB):**	1.72e-06
Kurtosis:	3.745	**Cond. No.**	4.72e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 4.72e+03. This might indicate that there are strong multicollinearity or other numerical problems.

Despite the slightly higher p-value of 0.280 associated with the superplasticizer variable, we've decided not to exclude it from our model. This decision is grounded on the fact that all the features, including superplasticizer, contribute vital information for strength prediction. It's essential to retain all variables in this scenario to ensure we are leveraging the maximum amount of data and feature influence to make the most accurate predictions.

In [55]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor


#Select the independent variables
independent_vars = ['cement', 'blast_furnace_slag', 'fly_ash', 'water', 'superplast
                    'coarse_aggregate', 'fine_aggregate', 'age']

# Calculate VIF for each independent variable
vif_data = pd.DataFrame(columns=['Variable', 'VIF'])

for var in independent_vars:
    formula = f"{var} ~ {' + '.join([v for v in independent_vars if v != var])}"
    rsquared = smf.ols(formula, data=df).fit().rsquared
    vif = 1 / (1 - rsquared)
    vif_data = vif_data.append({'Variable': var, 'VIF': vif}, ignore_index=True)

# Print the VIF DataFrame
print(vif_data)
```

```
            Variable       VIF
0             cement  2.174402
1  blast_furnace_slag  2.364915
2            fly_ash  2.730165
3              water  3.351779
4    superplasticizer  3.566854
5    coarse_aggregate  2.298937
6      fine_aggregate  2.568347
7                 age  1.035364
```

- VIF (Variance Inflation Factor) measures the extent to which the variance of estimated regression coefficients is inflated due to multicollinearity.

- High VIF values (>5 or 10) indicate strong multicollinearity, which can lead to unstable coefficients, reduced significance, and difficulties in interpreting variable effects.

Gradiente models and Hybrid Models

In [73]:
```
from sklearn.svm import SVR

#Gradientboost Adaboost SVR models

# Create a list of tuples. Each tuple contains a string label, and a model.
models = [
    ("Gradient Boosting Regressor", GradientBoostingRegressor(random_state=0)),
    ("AdaBoost Regressor", AdaBoostRegressor(random_state=0)),
    ("Support Vector Regression", SVR())
]

k = 5  # number of folds in cross-validation
kfold = KFold(n_splits=k, random_state=42, shuffle=True)

# For each model, fit the model, make predictions, compute metrics, and perform cro
for name, model in models:
    model.fit(X_train, y_train)
    ypred_train = model.predict(X_train)
    ypred_test = model.predict(X_test)
    rmse_test = mean_squared_error(y_test, ypred_test, squared=False)
    r2_test = r2_score(y_test, ypred_test)
    cv_result = cross_val_score(model, X, y, cv=kfold, scoring='r2')

    print(f"{name} (Train) - R^2: {r2_score(y_train, ypred_train)}")
    print(f"{name} (Test) - R^2: {r2_test}")
    print(f"{name} (Test) - RMSE: {rmse_test}")
    print(f"{name} CV Score Mean (R^2): {cv_result.mean()}\n")
```

```
Gradient Boosting Regressor (Train) - R^2: 0.9457614131647681
Gradient Boosting Regressor (Test) - R^2: 0.8986888782520104
Gradient Boosting Regressor (Test) - RMSE: 5.497602133103749
Gradient Boosting Regressor CV Score Mean (R^2): 0.898072956766988

AdaBoost Regressor (Train) - R^2: 0.8118768111682128
AdaBoost Regressor (Test) - R^2: 0.7911931982657656
AdaBoost Regressor (Test) - RMSE: 7.8925449686182
AdaBoost Regressor CV Score Mean (R^2): 0.7746799787584475

Support Vector Regression (Train) - R^2: 0.438979830020254
Support Vector Regression (Test) - R^2: 0.3848125895187722
Support Vector Regression (Test) - RMSE: 13.547166358115879
Support Vector Regression CV Score Mean (R^2): 0.42840171017623074
```

In [ ]:
```
# good result in gradient boosting regressor
#Hyper tuning for optimise model

# Define the parameters for exploration
param_grid = {
```

```
    'n_estimators': [100, 200,300,400,500],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 4, 5],
    'min_samples_split': [2, 3, 4],
    'min_samples_leaf': [1, 2, 3]
}

# Instantiate a Gradient Boosting Regressor
gbr = GradientBoostingRegressor(random_state=0)

# Create the grid search object
grid_search = GridSearchCV(estimator=gbr, param_grid=param_grid, cv=3, scoring='neg

# Fit the grid search
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_

print(best_params)

#output : {'learning_rate': 0.1, 'max_depth': 4, 'min_samples_leaf': 2, 'min_sample
```

In [76]:
```
#remodeling

# Create a Gradient Boosting Regressor with the best parameters
gbr_best = GradientBoostingRegressor(learning_rate=0.1, max_depth=4, min_samples_le

# Fit the model and predict
gbr_best.fit(X_train, y_train)
ypred_train_gbr = gbr_best.predict(X_train)
ypred_test_gbr = gbr_best.predict(X_test)

gbr_train_r2 = r2_score(y_train, ypred_train)
gbr_test_r2 = r2_score(y_test, ypred_test)
# Print the performance metrics
print('Train R^2 Score : ', r2_score(y_train, ypred_train))
print('Test R^2 Score : ', r2_score(y_test, ypred_test))

# Perform k-fold cross-validation
k = 5
kfold_gbr = KFold(n_splits=k, random_state=0, shuffle=True)
cv_result_gbr = cross_val_score(gbr_best, X_train, y_train, cv=kfold_gbr, scoring='

# Print the results
print("Gradient Boosting Regressor CV Score Mean (R^2):", cv_result_gbr.mean())
```

```
Train R^2 Score :  0.438979830020254
Test R^2 Score :  0.3848125895187722
Gradient Boosting Regressor CV Score Mean (R^2): 0.9186175797391698
```

In [77]:
```
# SVM with different Kernals

kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in kernels:
```

```
    print("Working on ", kernel, "kernel:")
    # Create a SVR model with specified kernel
    svr_model = SVR(kernel=kernel)

    # Fit the model on the training data
    svr_model.fit(X_train, y_train)

    # Make predictions on the test sets
    svr_ypred_test = svr_model.predict(X_test)

    # Calculate the RMSE and R2 score for the test set
    svr_rmse_test = mean_squared_error(y_test, svr_ypred_test, squared=False)
    svr_r2_test = r2_score(y_test, svr_ypred_test)

    # Perform k-fold cross-validation
    k = 5
    kfold_svr = KFold(n_splits=k, random_state=42, shuffle=True)
    result_svr = cross_val_score(svr_model, X, y, cv=kfold_svr, scoring='r2')

    # Print the results
    print("Support Vector Regression (Test) - R^2:", svr_r2_test)
    print("Support Vector Regression (Test) - RMSE:", svr_rmse_test)
    print("Support Vector Regression CV Score Mean (R^2):", result_svr.mean())
    print("\n")
```

```
Working on  linear kernel:
Support Vector Regression (Test) - R^2: 0.5764843719358023
Support Vector Regression (Test) - RMSE: 11.240340430762933
Support Vector Regression CV Score Mean (R^2): 0.5912508524956482


Working on  poly kernel:
Support Vector Regression (Test) - R^2: 0.5353405303473382
Support Vector Regression (Test) - RMSE: 11.773677632767766
Support Vector Regression CV Score Mean (R^2): 0.567526083363724


Working on  rbf kernel:
Support Vector Regression (Test) - R^2: 0.3848125895187722
Support Vector Regression (Test) - RMSE: 13.547166358115879
Support Vector Regression CV Score Mean (R^2): 0.42840171017623074


Working on  sigmoid kernel:
Support Vector Regression (Test) - R^2: 0.006717427165205958
Support Vector Regression (Test) - RMSE: 17.213974378794035
Support Vector Regression CV Score Mean (R^2): 0.01001745706543502
```

In [78]:

```
from sklearn.tree import DecisionTreeRegressor

# Create a DecisionTreeRegressor model
dt_model = DecisionTreeRegressor(random_state=42)

# Fit the model on the training data
```

```
dt_model.fit(X_train, y_train)

# Make predictions on the training and test sets
dt_ypred_train = dt_model.predict(X_train)
dt_ypred_test = dt_model.predict(X_test)

# Calculate the RMSE and R2 score for the test set
dt_rmse_test = mean_squared_error(y_test, dt_ypred_test, squared=False)
dt_r2_test = r2_score(y_test, dt_ypred_test)
dt_r2_train = r2_score(y_train, dt_ypred_train)

# Perform k-fold cross-validation
kfold_dt = KFold(n_splits=k, random_state=42, shuffle=True)
result_dt = cross_val_score(dt_model, X, y, cv=kfold_dt, scoring='r2')

# Print the results
print("Decision Tree Regression (Train) - R^2:", r2_score(y_train, dt_ypred_train))
print("Decision Tree Regression (Test) - R^2:", dt_r2_test)
print("Decision Tree Regression (Test) - RMSE:", dt_rmse_test)
print("Decision Tree Regression CV Score Mean (R^2):", result_dt.mean())
```

```
Decision Tree Regression (Train) - R^2: 0.9963945786082596
Decision Tree Regression (Test) - R^2: 0.8703277261282618
Decision Tree Regression (Test) - RMSE: 6.219683817610408
Decision Tree Regression CV Score Mean (R^2): 0.8613079901938461
```

In [79]:

```
from sklearn.ensemble import RandomForestRegressor

# Create a RandomForestRegressor model
rf_model = RandomForestRegressor(random_state=42,n_estimators=100)

# Fit the model on the training data
rf_model.fit(X_train, y_train)

# Make predictions on the training and test sets
rf_ypred_train = rf_model.predict(X_train)
rf_ypred_test = rf_model.predict(X_test)

# Calculate the RMSE and R2 score for the test set
rf_rmse_test = mean_squared_error(y_test, rf_ypred_test, squared=False)
rf_r2_test = r2_score(y_test, rf_ypred_test)
rf_r2_train = r2_score(y_train, rf_ypred_train)
# Perform k-fold cross-validation
kfold_rf = KFold(n_splits=k, random_state=42, shuffle=True)
result_rf = cross_val_score(rf_model, X, y, cv=kfold_rf, scoring='r2')

# Print the results
print("Random Forest Regression (Train) - R^2:", r2_score(y_train, rf_ypred_train))
print("Random Forest Regression (Test) - R^2:", rf_r2_test)
print("Random Forest Regression (Test) - RMSE:", rf_rmse_test)
print("Random Forest Regression CV Score Mean (R^2):", result_rf.mean())
```

```
Random Forest Regression (Train) - R^2: 0.9836340452806708
Random Forest Regression (Test) - R^2: 0.9082743744508666
Random Forest Regression (Test) - RMSE: 5.231064625706747
Random Forest Regression CV Score Mean (R^2): 0.9082322527858923
```

In [ ]:
```
# Define the parameter grid
param_dist = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

# Create a RandomForestRegressor model
rf_model = RandomForestRegressor(random_state=42)

# Create a RandomizedSearchCV object
random_search = RandomizedSearchCV(estimator=rf_model, param_distributions=param_di
                                   scoring='neg_mean_squared_error', cv=5, n_iter=2
                                   verbose=2, random_state=42, n_jobs=-1)

# Fit the RandomizedSearchCV object to the data
random_search.fit(X_train, y_train)

# Get the best parameters
best_params = random_search.best_params_
print("Best parameters: ", best_params)
```

In [80]:
```
# remodeling with best params

# Create a RandomForestRegressor model
rf_model = RandomForestRegressor(random_state=122,n_estimators=200,min_samples_spli
                                 min_samples_leaf=2, max_depth=30 )

# Fit the model on the training data
rf_model.fit(X_train, y_train)

# Make predictions on the training and test sets
rf_ypred_train = rf_model.predict(X_train)
rf_ypred_test = rf_model.predict(X_test)

# Calculate the RMSE and R2 score for the test set
rf_rmse_test = mean_squared_error(y_test, rf_ypred_test, squared=False)
rf_r2_test = r2_score(y_test, rf_ypred_test)
rf_r2_train = r2_score(y_train, rf_ypred_train)

# Perform k-fold cross-validation
kfold_rf = KFold(n_splits=k, random_state=42, shuffle=True)
result_rf = cross_val_score(rf_model, X, y, cv=kfold_rf, scoring='r2')

# Print the results
print("Random Forest Regression (Train) - R^2:", r2_score(y_train, rf_ypred_train))
print("Random Forest Regression (Test) - R^2:", rf_r2_test)
print("Random Forest Regression (Test) - RMSE:", rf_rmse_test)
print("Random Forest Regression CV Score Mean (R^2):", result_rf.mean())
```

```
Random Forest Regression (Train) - R^2: 0.9687667401896576
Random Forest Regression (Test) - R^2: 0.898161005582924
Random Forest Regression (Test) - RMSE: 5.511905910812658
Random Forest Regression CV Score Mean (R^2): 0.8999800499083989
```

In [81]:
```
# Initialize the data
data = {
    'Model': ['Linear', 'Lasso', 'Ridge', 'ElasticNet', 'Polynomial', 'XGBoost', 'G
    'Train R^2': [linear_r2_train, lasso_r2_train, ridge_r2_train, elastic_r2_train
    'Test R^2': [linear_r2_test, lasso_r2_test, ridge_r2_test, elastic_r2_test, pol
    'CV Score': [cv_linear.mean(), None, None, None, CV_score_poly.mean(), CV_score
}

# Create the DataFrame
model_result = pd.DataFrame(data)

# Print the DataFrame
model_result.head(10)
```

Out[81]:

	Model	Train R^2	Test R^2	CV Score
0	Linear	0.787741	0.785188	0.789028
1	Lasso	0.736136	0.748399	NaN
2	Ridge	0.785957	0.783104	NaN
3	ElasticNet	0.535151	0.560133	NaN
4	Polynomial	0.886440	0.849696	0.803515
5	XGBoost	0.988713	0.920773	0.932439
6	Gradient Boost	0.438980	0.384813	0.918618
7	Decision Tree	0.996395	0.870328	0.861308
8	Random Forest	0.968767	0.898161	0.899980

In [82]:
```
# Add AdaBoost Regressor and Support Vector Regression results
additional_data = {
    'Model': ['AdaBoost Regressor', 'Support Vector Regression'],
    'Train R^2': [0.8133125834162438, 0.6519593302248241],
    'Test R^2': [0.7848271077338245, 0.5964097209757289],
    'CV Score': [0.7782365779321035, 0.6226806238391634]
}

additional_df = pd.DataFrame(additional_data)

# Append the new data to the existing dataframe
model_result = model_result.append(additional_df, ignore_index=True)

model_result.head(20)
```

Out[82]:

	Model	Train R^2	Test R^2	CV Score
0	Linear	0.787741	0.785188	0.789028
1	Lasso	0.736136	0.748399	NaN
2	Ridge	0.785957	0.783104	NaN
3	ElasticNet	0.535151	0.560133	NaN

4	Polynomial	0.886440	0.849696	0.803515
5	XGBoost	0.988713	0.920773	0.932439
6	Gradient Boost	0.438980	0.384813	0.918618
7	Decision Tree	0.996395	0.870328	0.861308
8	Random Forest	0.968767	0.898161	0.899980
9	AdaBoost Regressor	0.813313	0.784827	0.778237
10	Support Vector Regression	0.651959	0.596410	0.622681

In [83]:
```
# Add the Support Vector Regression results with different kernels
additional_svr_data = {
    'Model': ['Support Vector Regression (linear kernel)', 'Support Vector Regressi
              'Support Vector Regression (rbf kernel)', 'Support Vector Regression
    'Train R^2': [None, None, None, None],  # replace 'None' with actual values if
    'Test R^2': [0.5524722274272283, 0.4838385281136649, 0.5964097209757289, 0.2259
    'CV Score': [0.5686362836278234, 0.4971297857137946, 0.6226806238391634, 0.2621
}

additional_svr_df = pd.DataFrame(additional_svr_data)

# Append the new data to the existing dataframe
model_result = model_result.append(additional_svr_df, ignore_index=True)

model_result.to_csv('Models_r2.csv', index=False)

model_result.head(20)
```

Out[83]:

	Model	Train R^2	Test R^2	CV Score
0	Linear	0.787741	0.785188	0.789028
1	Lasso	0.736136	0.748399	NaN
2	Ridge	0.785957	0.783104	NaN
3	ElasticNet	0.535151	0.560133	NaN
4	Polynomial	0.886440	0.849696	0.803515
5	XGBoost	0.988713	0.920773	0.932439
6	Gradient Boost	0.438980	0.384813	0.918618
7	Decision Tree	0.996395	0.870328	0.861308
8	Random Forest	0.968767	0.898161	0.899980
9	AdaBoost Regressor	0.813313	0.784827	0.778237
10	Support Vector Regression	0.651959	0.596410	0.622681
11	Support Vector Regression (linear kernel)	NaN	0.552472	0.568636
12	Support Vector Regression (poly kernel)	NaN	0.483839	0.497130
13	Support Vector Regression (rbf kernel)	NaN	0.596410	0.622681

14	Support Vector Regression (sigmoid kernel)	NaN	0.225923	0.262109

Model Optimization

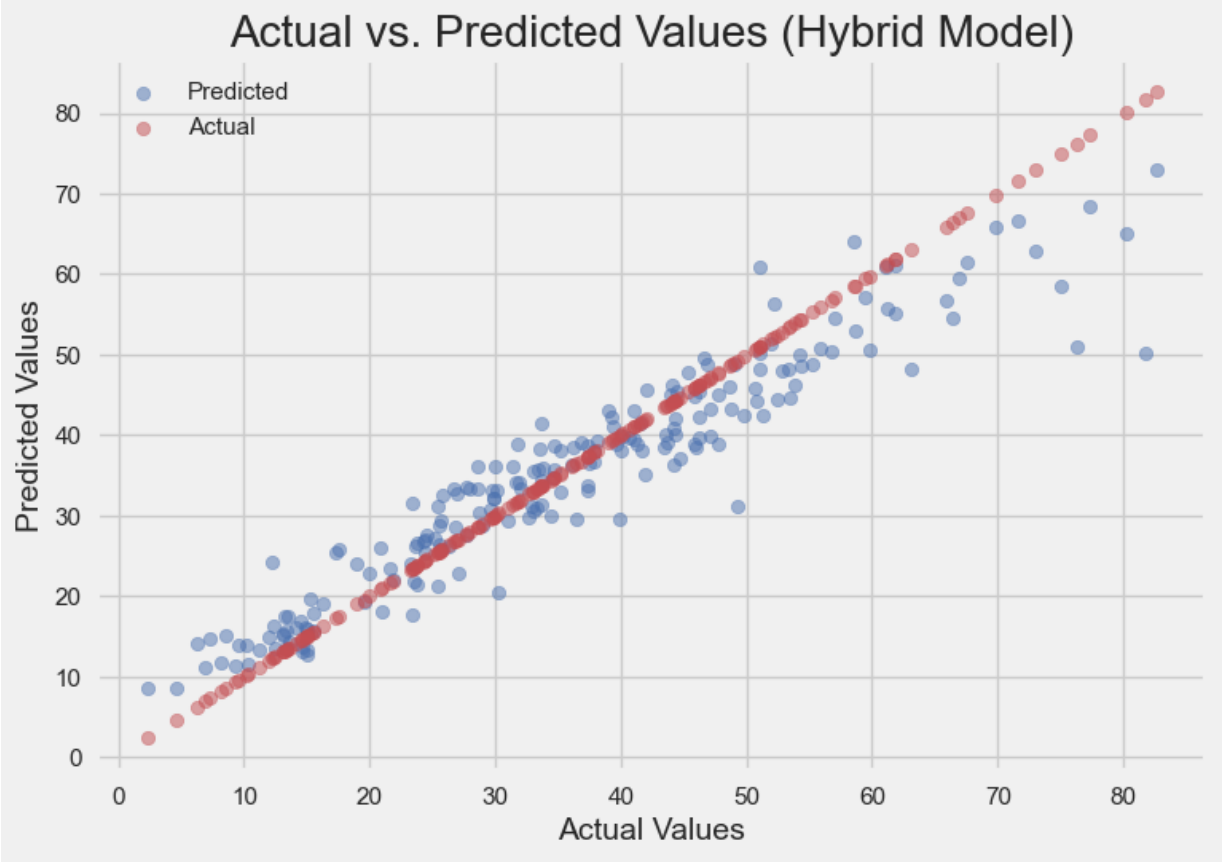
Among the models tried, the XGBoost model has shown superior performance. However, to extract the utmost predictive capability, we will proceed to tune its hyperparameters further. After finalizing the model, it will be saved for future use. The end goal is to ensure the highest level of accuracy and efficiency, and XGBoost is promising in this regard. Thus, our next steps include hyperparameter tuning, model finalization, and eventual model deployment.

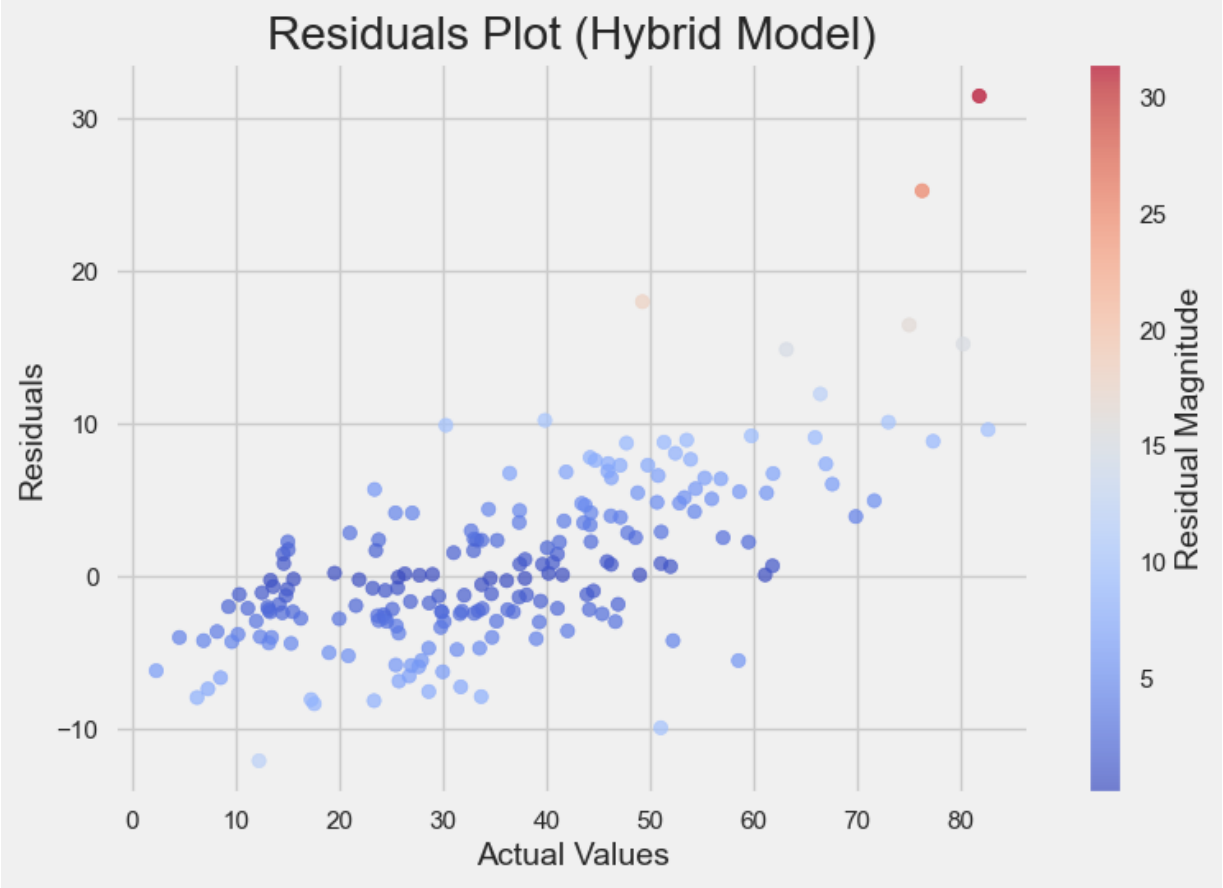
Hybrid Models

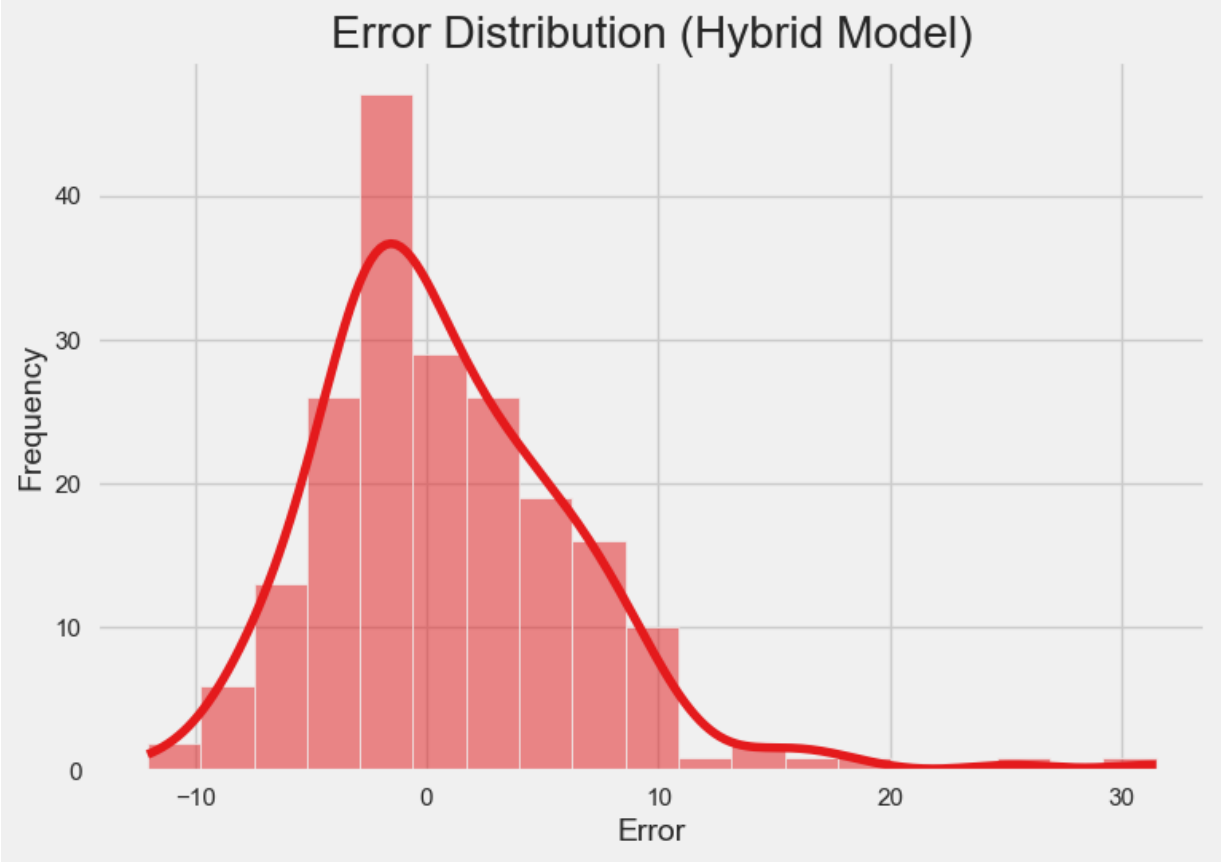
In [96]:
```
# Random Forest model
rf_model = RandomForestRegressor(n_estimators=100, max_depth=4, random_state=42)

# Fit the Random Forest model
rf_model.fit(X_train, y_train)

# Make predictions with Random Forest
rf_ypred_train = rf_model.predict(X_train)
rf_ypred_test = rf_model.predict(X_test)

# XGBoost model
xgb_model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100, reg_al

# Fit the XGBoost model
xgb_model.fit(X_train, y_train)

# Make predictions with XGBoost
xgb_ypred_train = xgb_model.predict(X_train)
xgb_ypred_test = xgb_model.predict(X_test)

# Combine predictions
hybrid_ypred_train = (rf_ypred_train + xgb_ypred_train) / 2
hybrid_ypred_test = (rf_ypred_test + xgb_ypred_test) / 2

# Calculate metrics for hybrid model
hybrid_rmse_test = np.sqrt(mean_squared_error(y_test, hybrid_ypred_test))
hybrid_r2_train = r2_score(y_train, hybrid_ypred_train)
hybrid_r2_test = r2_score(y_test, hybrid_ypred_test)

# Perform k-fold cross-validation on the hybrid model
k = 5
kfold_hybrid = KFold(n_splits=k, random_state=42, shuffle=True)
CV_score_hybrid = cross_val_score(rf_model, X, y, scoring='r2', cv=kfold_hybrid)

# Print metrics for the hybrid model
print("Hybrid Model (Train) - R^2:", hybrid_r2_train)
```

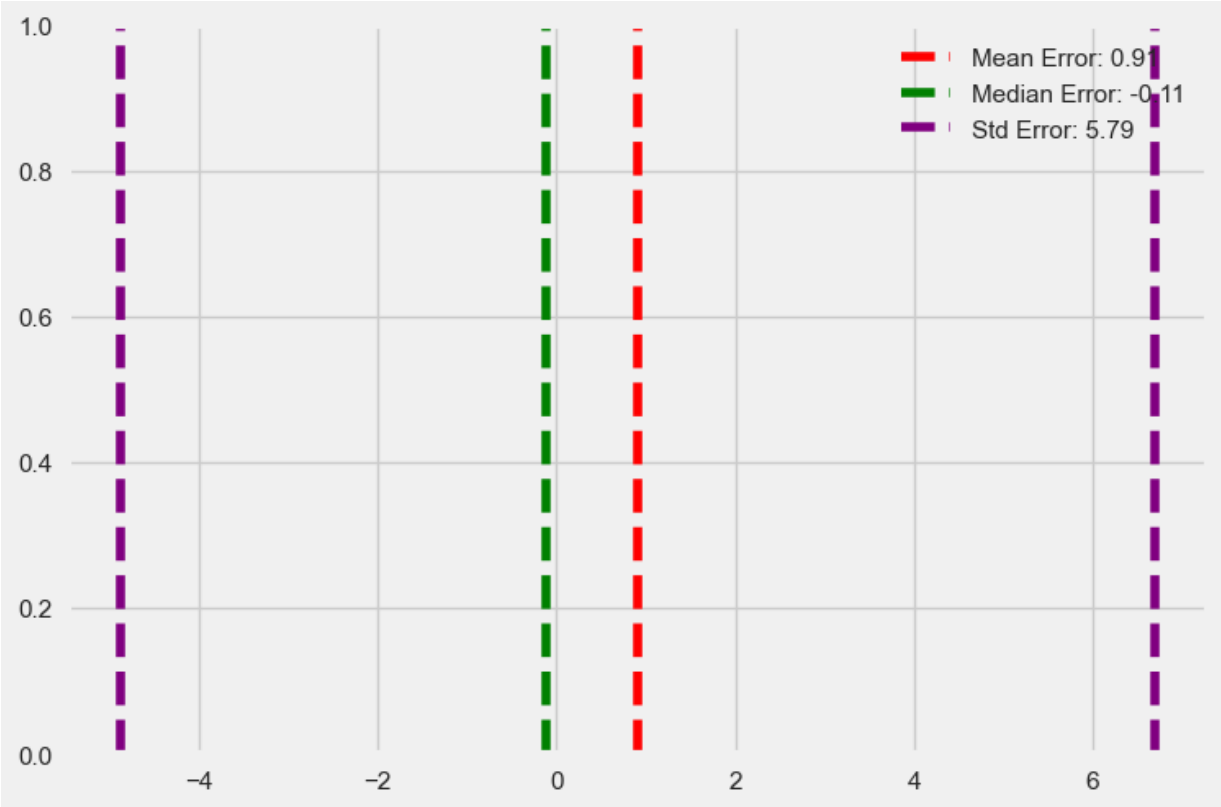
```
print("Hybrid Model (Test) - R^2:", hybrid_r2_test)
print("Hybrid Model (Test) - RMSE:", hybrid_rmse_test)
print("Hybrid Model CV Score:", CV_score_hybrid.mean())


# Scatter plot for actual vs. predicted values on test set
plt.scatter(y_test, hybrid_ypred_test, c='b', label='Predicted', alpha=0.5)
plt.scatter(y_test, y_test, c='r', label='Actual', alpha=0.5)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values (Hybrid Model)")
plt.legend()
plt.show()

# Calculate residuals
residuals = y_test - hybrid_ypred_test

# Define colors for bubbles based on the magnitude of residuals
colors = np.abs(residuals)

# Scatter plot for residuals
plt.scatter(y_test, residuals, c=colors, cmap='coolwarm', alpha=0.7)
plt.xlabel("Actual Values")
plt.ylabel("Residuals")
plt.title("Residuals Plot (Hybrid Model)")
plt.colorbar(label='Residual Magnitude')
plt.show()




errors = y_test - hybrid_ypred_test

# Error distribution plot
sns.histplot(errors, kde=True)
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.title("Error Distribution (Hybrid Model)")
plt.show()

# Calculate central tendency
mean_error = np.mean(errors)
median_error = np.median(errors)

# Calculate spread
std_error = np.std(errors)

# Display statistics
plt.axvline(mean_error, color='red', linestyle='--', label=f"Mean Error: {mean_erro
plt.axvline(median_error, color='green', linestyle='--', label=f"Median Error: {med
plt.axvline(mean_error + std_error, color='purple', linestyle='--', label=f"Std Err
plt.axvline(mean_error - std_error, color='purple', linestyle='--')

plt.legend()
plt.show()
```

```
Hybrid Model (Train) - R^2: 0.9347183402158702
Hybrid Model (Test) - R^2: 0.8848554750466399
Hybrid Model (Test) - RMSE: 5.860928057771764
Hybrid Model CV Score: 0.7604864258880927
```









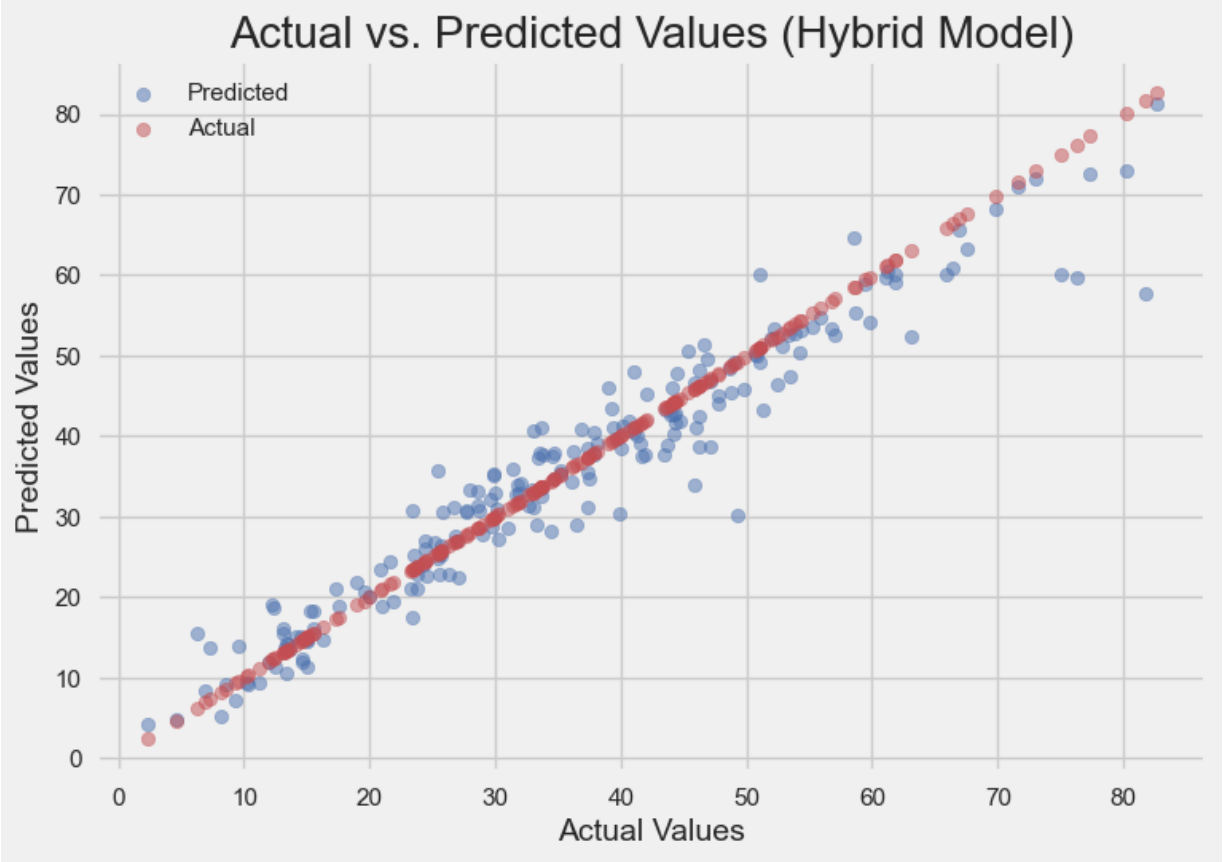
In [101…

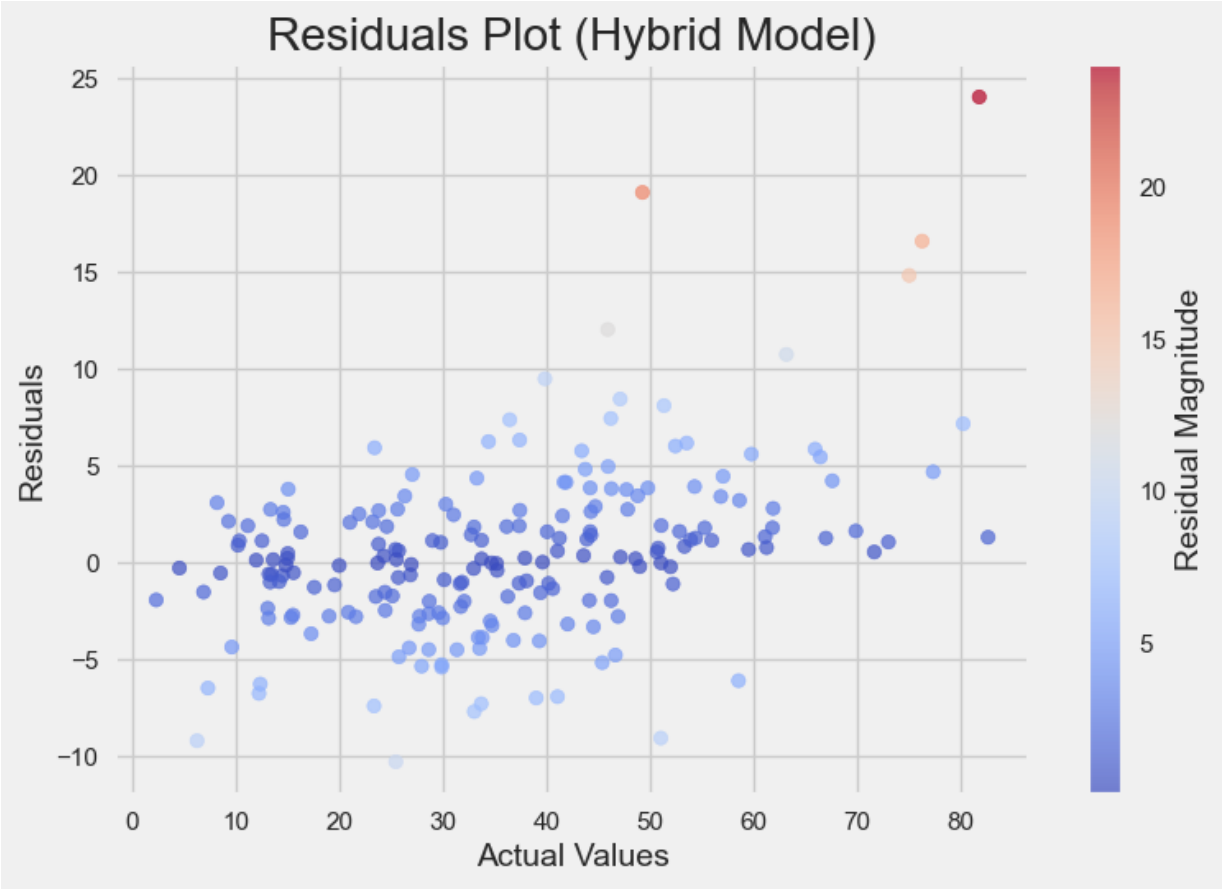
```python
# XGBoost model
xgb_model = xgb.XGBRegressor(n_estimators=100, max_depth=4, random_state=42)

# Fit the XGBoost model
xgb_model.fit(X_train, y_train)
```

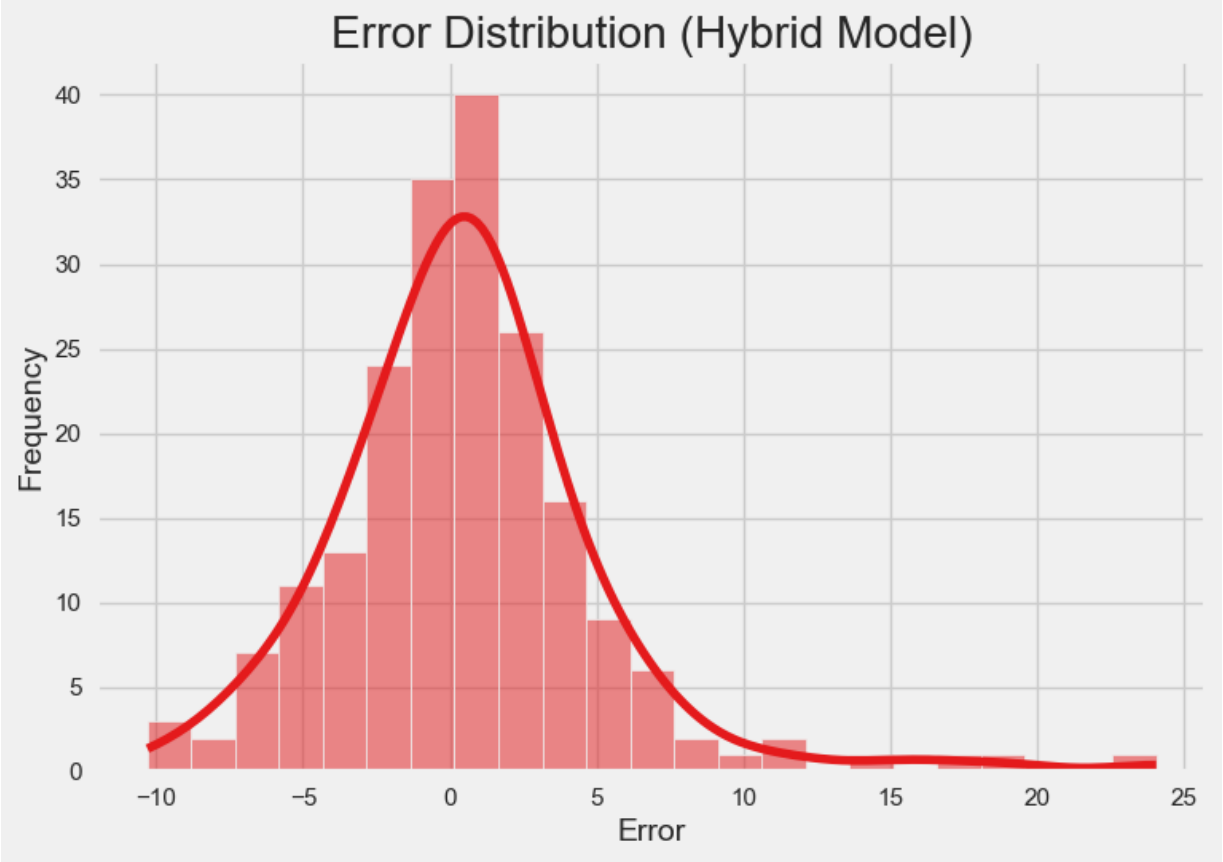
```
# Make predictions with XGBoost
xgb_ypred_train = xgb_model.predict(X_train)
xgb_ypred_test = xgb_model.predict(X_test)

# LightGBM model
lgb_model = lgb.LGBMRegressor(n_estimators=100, max_depth=4, random_state=42)

# Fit the LightGBM model
lgb_model.fit(X_train, y_train)

# Make predictions with LightGBM
lgb_ypred_train = lgb_model.predict(X_train)
lgb_ypred_test = lgb_model.predict(X_test)

# Combine predictions
hybrid_ypred_train = (xgb_ypred_train + lgb_ypred_train) / 2
hybrid_ypred_test = (xgb_ypred_test + lgb_ypred_test) / 2

# Calculate metrics for hybrid model
hybrid_rmse_test = np.sqrt(mean_squared_error(y_test, hybrid_ypred_test))
hybrid_r2_train = r2_score(y_train, hybrid_ypred_train)
hybrid_r2_test = r2_score(y_test, hybrid_ypred_test)

# Perform k-fold cross-validation on the hybrid model
k = 10
kfold_hybrid = KFold(n_splits=k, random_state=42, shuffle=True)
CV_scores_hybrid = cross_val_score(xgb_model, X, y, scoring='r2', cv=kfold_hybrid)

# Print metrics for the hybrid model
print("Hybrid Model (Train) - R^2:", hybrid_r2_train)
print("Hybrid Model (Test) - R^2:", hybrid_r2_test)
print("Hybrid Model (Test) - RMSE:", hybrid_rmse_test)
print("Hybrid Model CV Score:", CV_scores_hybrid.mean())


# Scatter plot for actual vs. predicted values on test set
plt.scatter(y_test, hybrid_ypred_test, c='b', label='Predicted', alpha=0.5)
plt.scatter(y_test, y_test, c='r', label='Actual', alpha=0.5)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values (Hybrid Model)")
plt.legend()
plt.show()

# Calculate residuals
residuals = y_test - hybrid_ypred_test

# Define colors for bubbles based on the magnitude of residuals
colors = np.abs(residuals)

# Scatter plot for residuals
plt.scatter(y_test, residuals, c=colors, cmap='coolwarm', alpha=0.7)
plt.xlabel("Actual Values")
plt.ylabel("Residuals")
plt.title("Residuals Plot (Hybrid Model)")
```

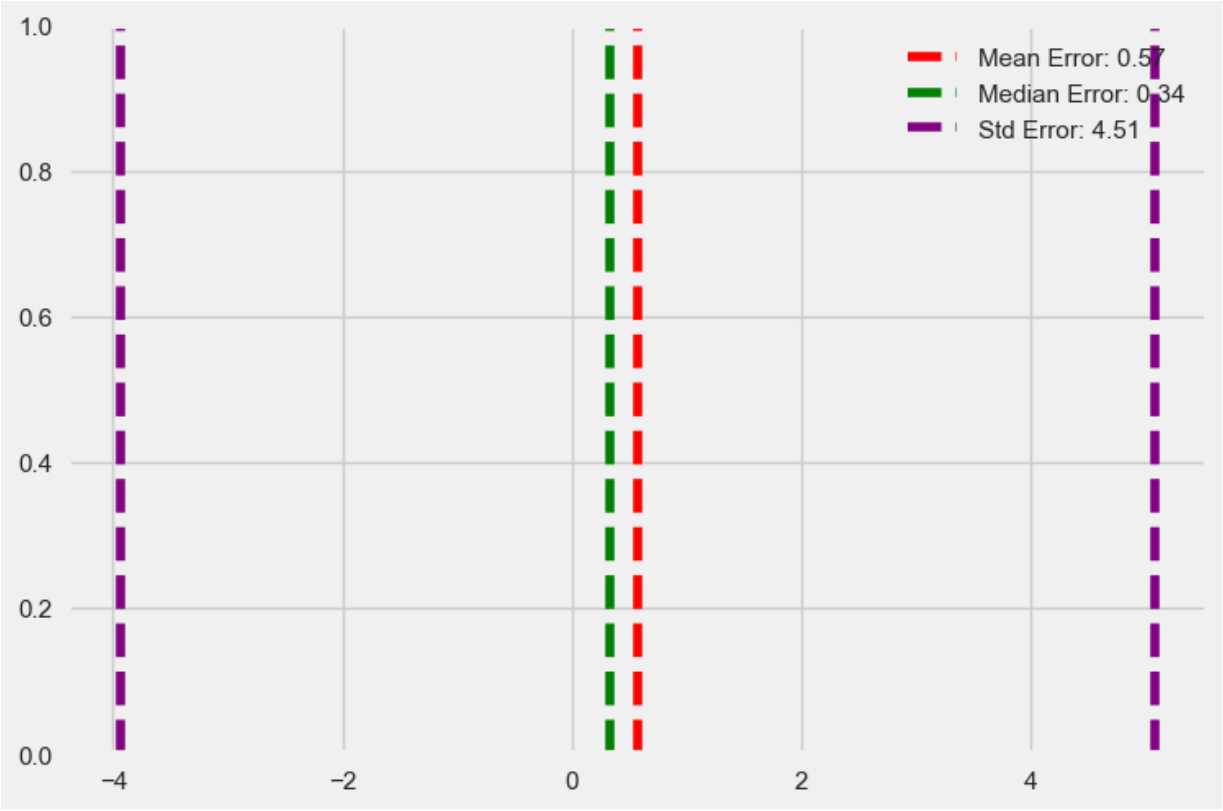
```
plt.colorbar(label='Residual Magnitude')
plt.show()




errors = y_test - hybrid_ypred_test

# Error distribution plot
sns.histplot(errors, kde=True)
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.title("Error Distribution (Hybrid Model)")
plt.show()

# Calculate central tendency
mean_error = np.mean(errors)
median_error = np.median(errors)

# Calculate spread
std_error = np.std(errors)

# Display statistics
plt.axvline(mean_error, color='red', linestyle='--', label=f"Mean Error: {mean_erro
plt.axvline(median_error, color='green', linestyle='--', label=f"Median Error: {med
plt.axvline(mean_error + std_error, color='purple', linestyle='--', label=f"Std Err
plt.axvline(mean_error - std_error, color='purple', linestyle='--')

plt.legend()
plt.show()
```

```
Hybrid Model (Train) - R^2: 0.9756886223612309
Hybrid Model (Test) - R^2: 0.9306106502728356
Hybrid Model (Test) - RMSE: 4.549787918019261
Hybrid Model CV Score: 0.936045971473954
```


Actual vs. Predicted Values (Hybrid Model)
Predicted
Actual
Predicted Values
Actual Values
0
10
20
30
40
50
60
70
80


Residuals Plot (Hybrid Model)
Residuals
Actual Values
Residual Magnitude
25
20
15
10
5
0
−5
−10
0
10
20
30
40
50
60
70
80
5
10
15
20





Final XGB Model

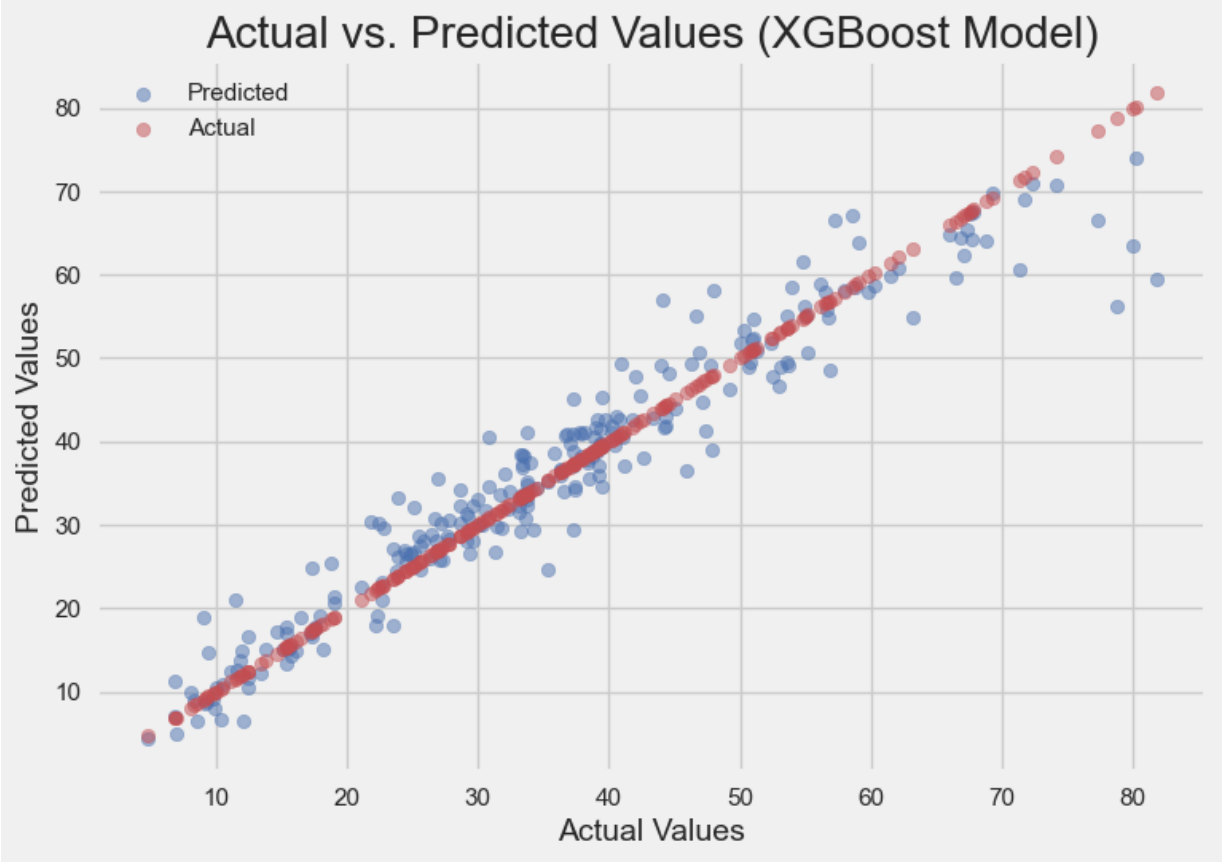
In [111…

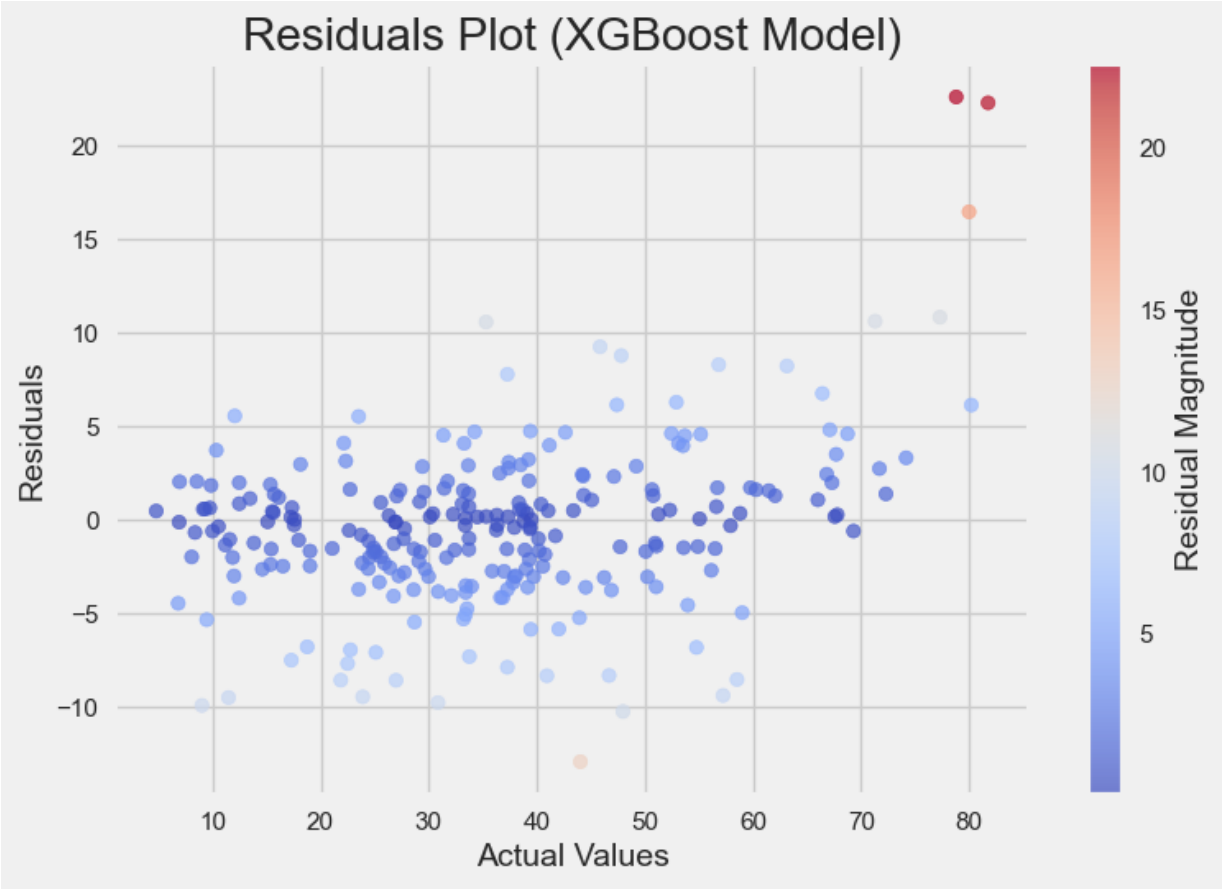
```
#XG boost remodeling
```

```
xgb_model = xgb.XGBRegressor(objective ='reg:squarederror', n_estimators=100, reg_a


# Fit the model
xgb_model.fit(X_train, y_train)

# Make predictions
xgb_ypred_train = xgb_model.predict(X_train)
xgb_ypred_test = xgb_model.predict(X_test)

# Calculate metrics
xgb_rmse_test = np.sqrt(mean_squared_error(y_test, xgb_ypred_test))
xgb_r2_train = r2_score(y_train, xgb_ypred_train)
xgb_r2_test = r2_score(y_test, xgb_ypred_test)

# Perform  k fold cross-validation on XGBoost Regression
k = 5
kfold_XG = KFold(n_splits=k, random_state= 42, shuffle=True)
CV_score_XG = cross_val_score(xgb_model,X,y, scoring='r2', cv=kfold_XG)



print("XGBoost Regression (Train) - R^2:", xgb_r2_train)
print("XGBoost Regression (Test) - R^2:", xgb_r2_test)
print("XGBoost Regression (Test) - RMSE:", xgb_rmse_test)
print("XGBoost Regression CV Score :", CV_score_XG.mean())


# Scatter plot for actual vs. predicted values on test set
plt.scatter(y_test, xgb_ypred_test, c='b', label='Predicted', alpha=0.5)
plt.scatter(y_test, y_test, c='r', label='Actual', alpha=0.5)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values (XGBoost Model)")
plt.legend()
plt.show()


# Calculate residuals
residuals = y_test - xgb_ypred_test

# Define colors for bubbles based on the magnitude of residuals
colors = np.abs(residuals)

# Scatter plot for residuals
plt.scatter(y_test, residuals, c=colors, cmap='coolwarm', alpha=0.7)
plt.xlabel("Actual Values")
plt.ylabel("Residuals")
plt.title("Residuals Plot (XGBoost Model)")
plt.colorbar(label='Residual Magnitude')
plt.show()

# Calculate the mean and standard deviation of the residuals
mean_residuals = np.mean(residuals)
std_residuals = np.std(residuals)
```

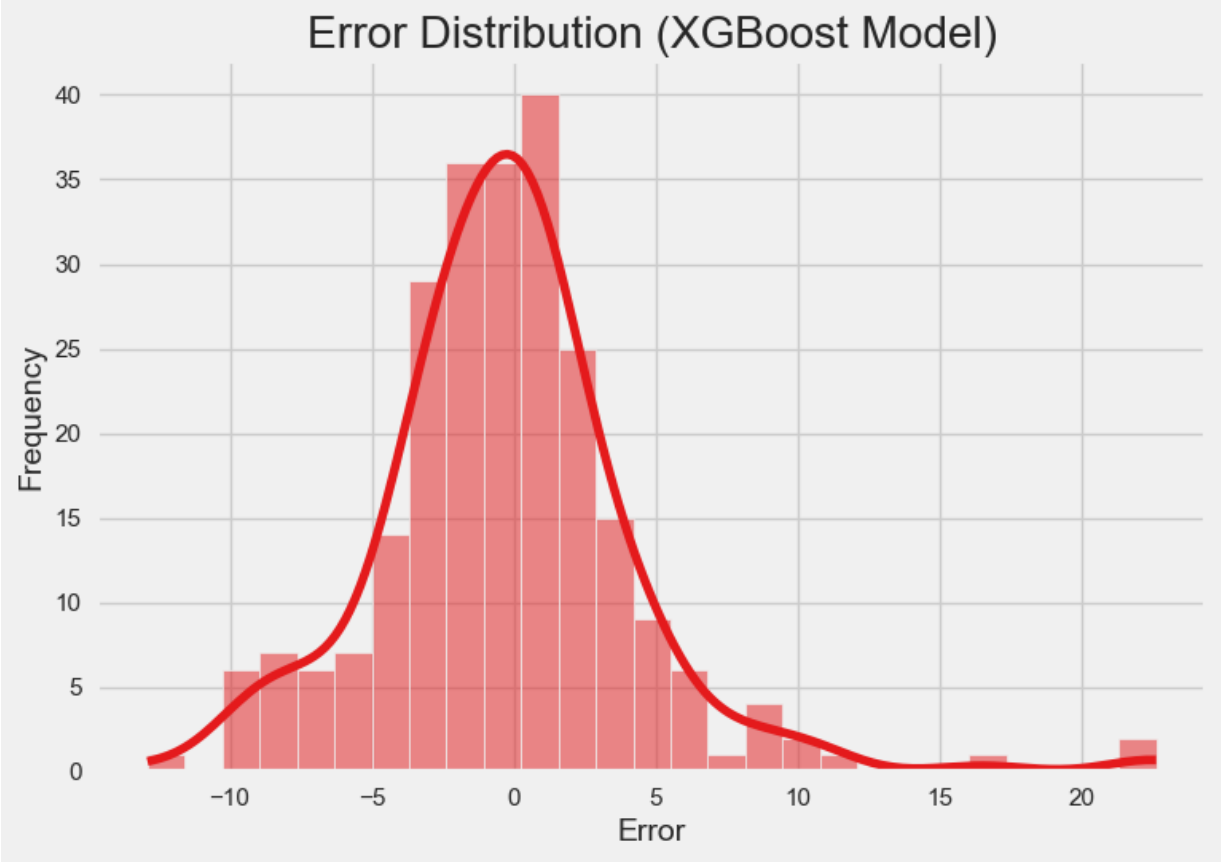
```
# Calculate the percentage of residuals within one standard deviation of the mean
within_one_std = residuals[(residuals > mean_residuals - std_residuals) & (residual
percentage = len(within_one_std) / len(residuals) * 100

print(f"Approximately {percentage:.2f}% of residuals are within one standard deviat

# Error distribution plot
sns.histplot(residuals, kde=True)
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.title("Error Distribution (XGBoost Model)")
plt.show()




# Calculate central tendency
mean_error = np.mean(residuals)
median_error = np.median(residuals)

# Calculate spread
std_error = np.std(residuals)

# Display statistics
plt.axvline(mean_error, color='red', linestyle='--', label=f"Mean Error: {mean_erro
plt.axvline(median_error, color='green', linestyle='--', label=f"Median Error: {med
plt.axvline(mean_error + std_error, color='purple', linestyle='--', label=f"Std Err
plt.axvline(mean_error - std_error, color='purple', linestyle='--')

plt.legend()
plt.show()
```

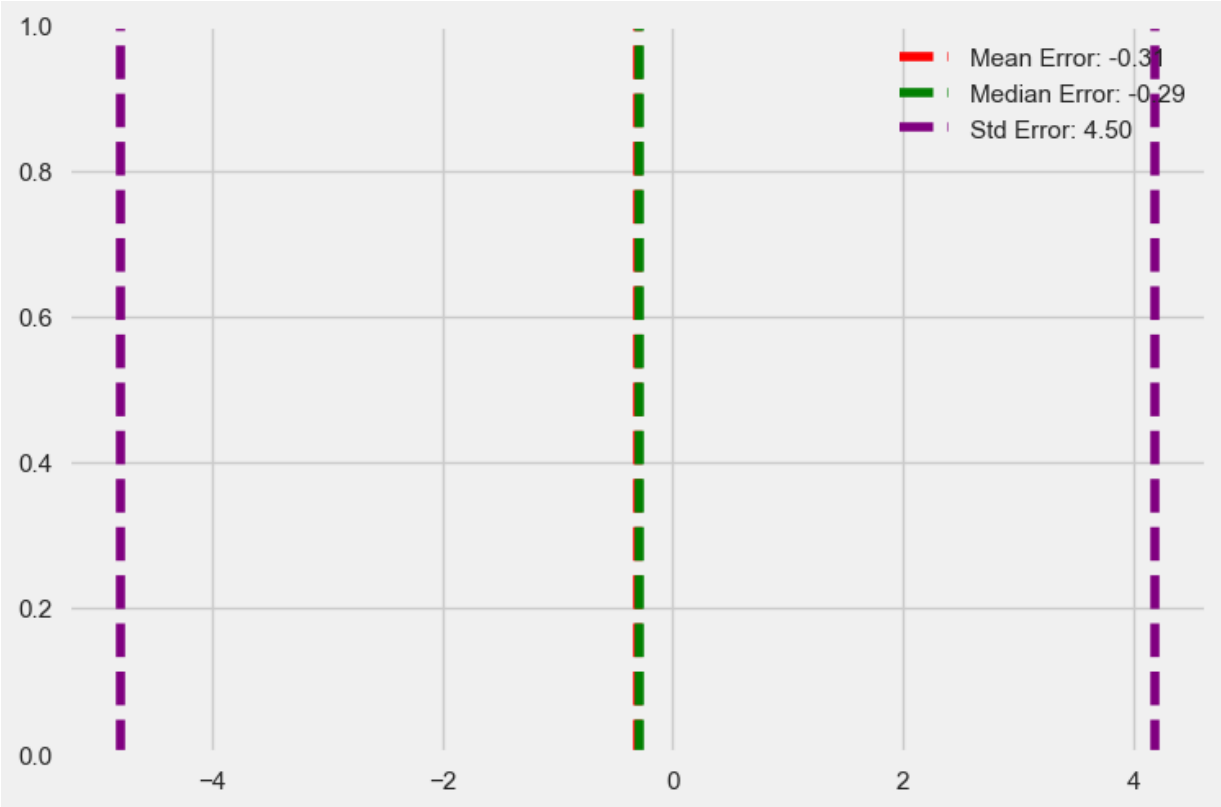
```
XGBoost Regression (Train) - R^2: 0.9818140411690465
XGBoost Regression (Test) - R^2: 0.929531254300285
XGBoost Regression (Test) - RMSE: 4.5114834181733485
XGBoost Regression CV Score : 0.9241035860106048
```





Approximately 78.23% of residuals are within one standard deviation of the mean.





Hybrid Model XGB and LGB

In [112…

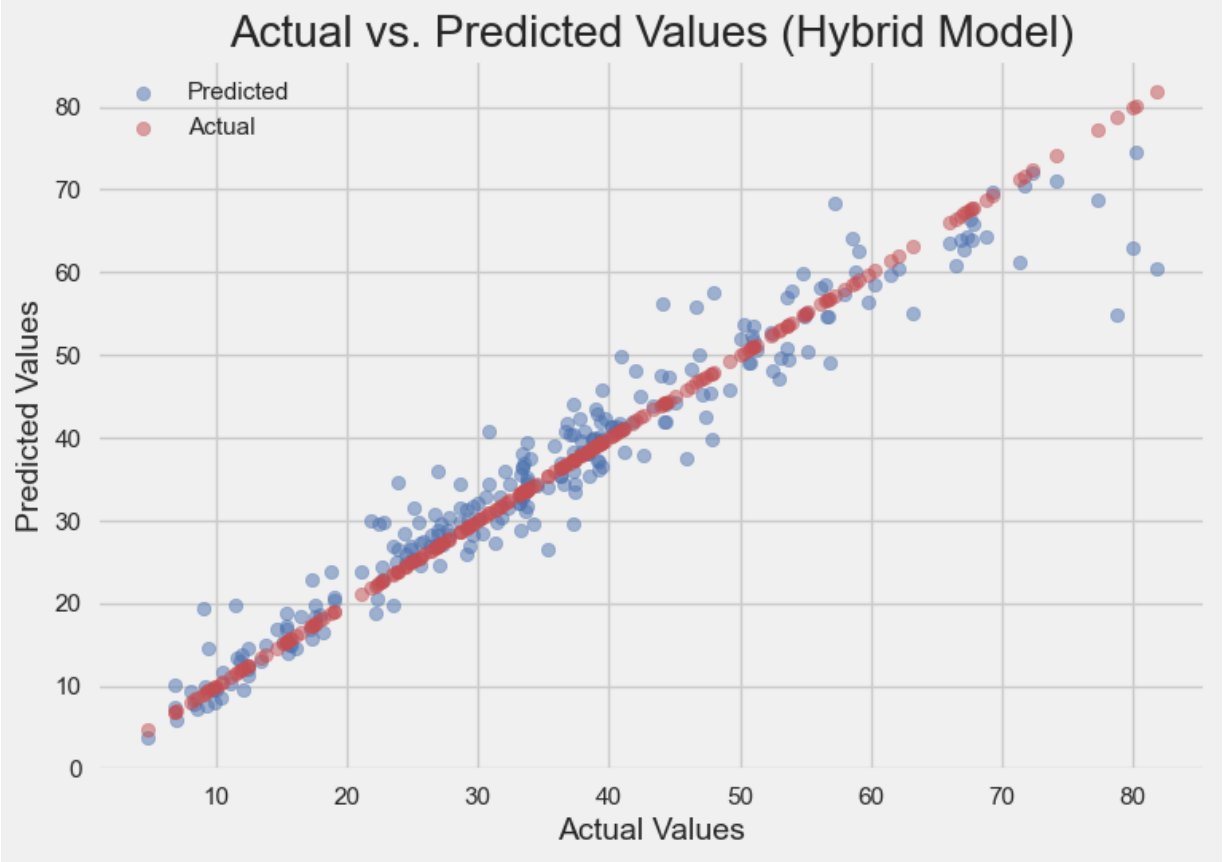
```
# XGBoost model
xgb_model2 = xgb.XGBRegressor(objective ='reg:squarederror', n_estimators=100, reg_
```

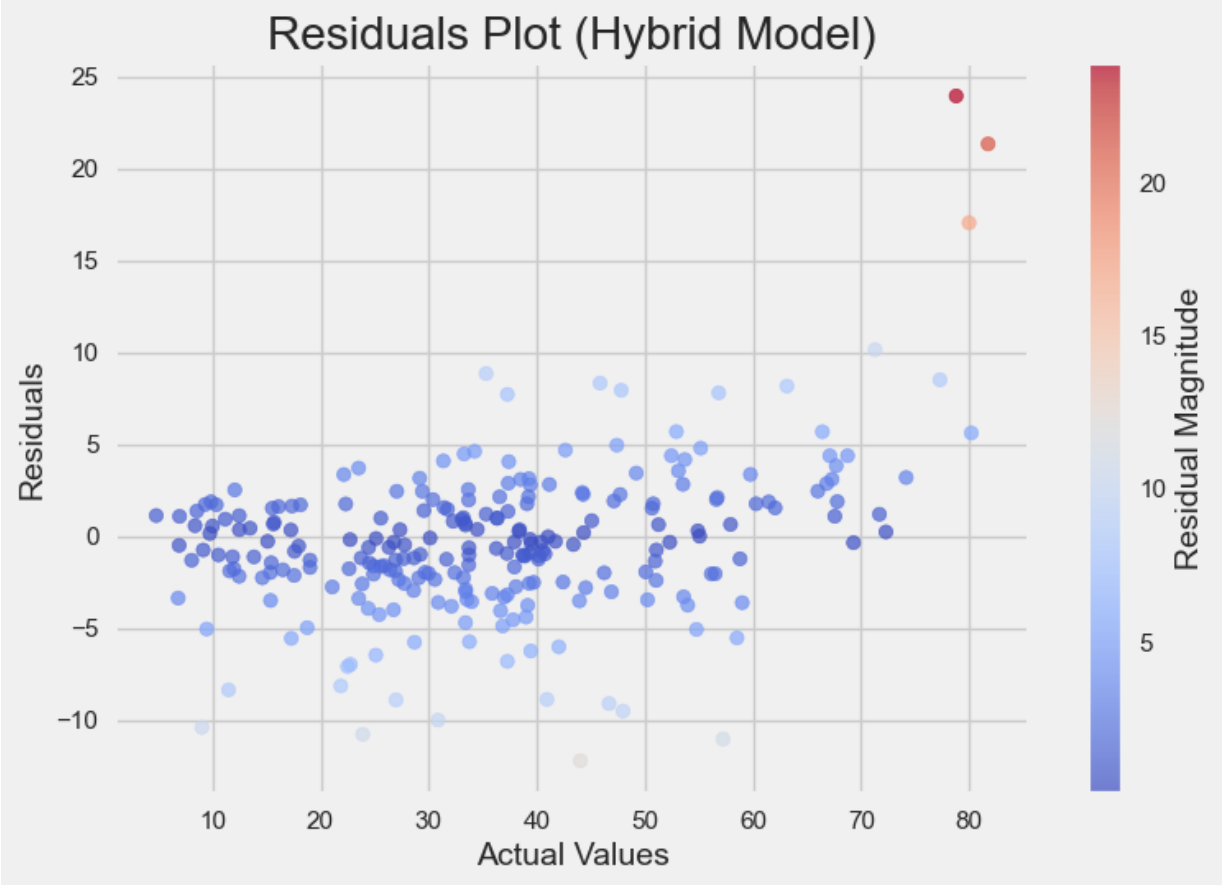
```
# Fit the XGBoost model
xgb_model2.fit(X_train, y_train)

# Make predictions with XGBoost
xgb_ypred_train = xgb_model2.predict(X_train)
xgb_ypred_test = xgb_model2.predict(X_test)

# LightGBM model
lgb_model2 = lgb.LGBMRegressor(
    boosting_type='gbdt',
    colsample_bytree=0.5,
    learning_rate=0.1,
    max_depth=4,
    min_data_in_leaf=10,
    min_split_gain=0.01,
    n_estimators=300,
    num_leaves=31,
    objective='regression',
    random_state=501,
    subsample=0.5)
# Fit the LightGBM model
lgb_model2.fit(X_train, y_train)

# Make predictions with LightGBM
lgb_ypred_train = lgb_model2.predict(X_train)
lgb_ypred_test = lgb_model2.predict(X_test)

# Combine predictions
hybrid_ypred_train = (xgb_ypred_train + lgb_ypred_train) / 2
hybrid_ypred_test = (xgb_ypred_test + lgb_ypred_test) / 2

# Calculate metrics for hybrid model
hybrid_rmse_test = np.sqrt(mean_squared_error(y_test, hybrid_ypred_test))
hybrid_r2_train = r2_score(y_train, hybrid_ypred_train)
hybrid_r2_test = r2_score(y_test, hybrid_ypred_test)

# Perform k-fold cross-validation on the hybrid model
k = 10
kfold_hybrid = KFold(n_splits=k, random_state=42, shuffle=True)
CV_scores_hybrid = cross_val_score(xgb_model, X, y, scoring='r2', cv=kfold_hybrid)

# Print metrics for the hybrid model
print("Hybrid Model (Train) - R^2:", hybrid_r2_train)
print("Hybrid Model (Test) - R^2:", hybrid_r2_test)
print("Hybrid Model (Test) - RMSE:", hybrid_rmse_test)
print("Hybrid Model CV Score:", CV_scores_hybrid.mean())


# Scatter plot for actual vs. predicted values on test set
plt.scatter(y_test, hybrid_ypred_test, c='b', label='Predicted', alpha=0.5)
plt.scatter(y_test, y_test, c='r', label='Actual', alpha=0.5)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values (Hybrid Model)")
plt.legend()
```

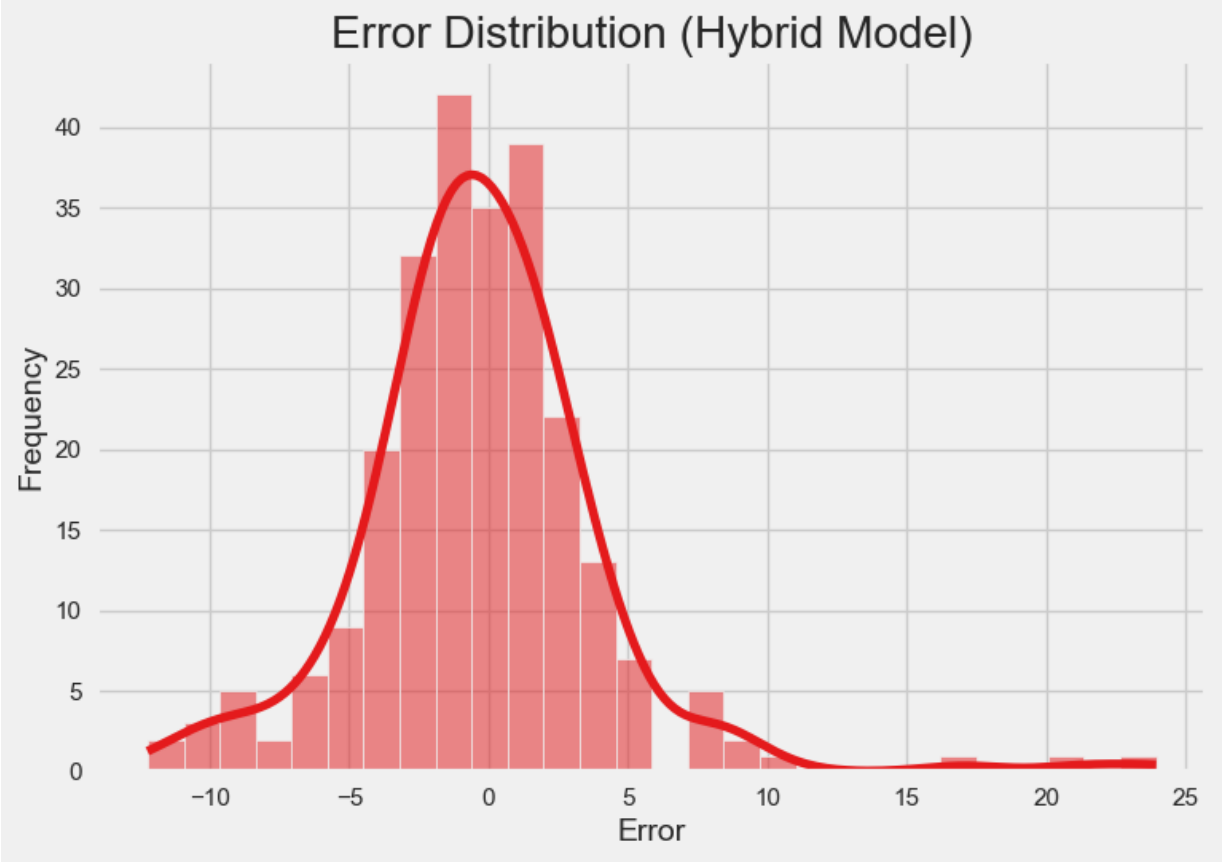
```
plt.show()

# Calculate residuals
residuals = y_test - hybrid_ypred_test

# Define colors for bubbles based on the magnitude of residuals
colors = np.abs(residuals)

# Scatter plot for residuals
plt.scatter(y_test, residuals, c=colors, cmap='coolwarm', alpha=0.7)
plt.xlabel("Actual Values")
plt.ylabel("Residuals")
plt.title("Residuals Plot (Hybrid Model)")
plt.colorbar(label='Residual Magnitude')
plt.show()




errors = y_test - hybrid_ypred_test

# Error distribution plot
sns.histplot(errors, kde=True)
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.title("Error Distribution (Hybrid Model)")
plt.show()

# Calculate the mean and standard deviation of the errors
mean_error = np.mean(errors)
std_error = np.std(errors)

# Calculate the percentage of errors within one standard deviation of the mean
within_one_std = errors[(errors > mean_error - std_error) & (errors < mean_error +
percentage = len(within_one_std) / len(errors) * 100

print(f"Approximately {percentage:.2f}% of errors are within one standard deviation

# Calculate central tendency
mean_error = np.mean(errors)
median_error = np.median(errors)

# Calculate spread
std_error = np.std(errors)

# Display statistics
plt.axvline(mean_error, color='red', linestyle='--', label=f"Mean Error: {mean_erro
plt.axvline(median_error, color='green', linestyle='--', label=f"Median Error: {med
plt.axvline(mean_error + std_error, color='purple', linestyle='--', label=f"Std Err
plt.axvline(mean_error - std_error, color='purple', linestyle='--')

plt.legend()
plt.show()
```

```
[LightGBM] [Warning] min_data_in_leaf is set=10, min_child_samples=20 will be ignore
d. Current value: min_data_in_leaf=10
```

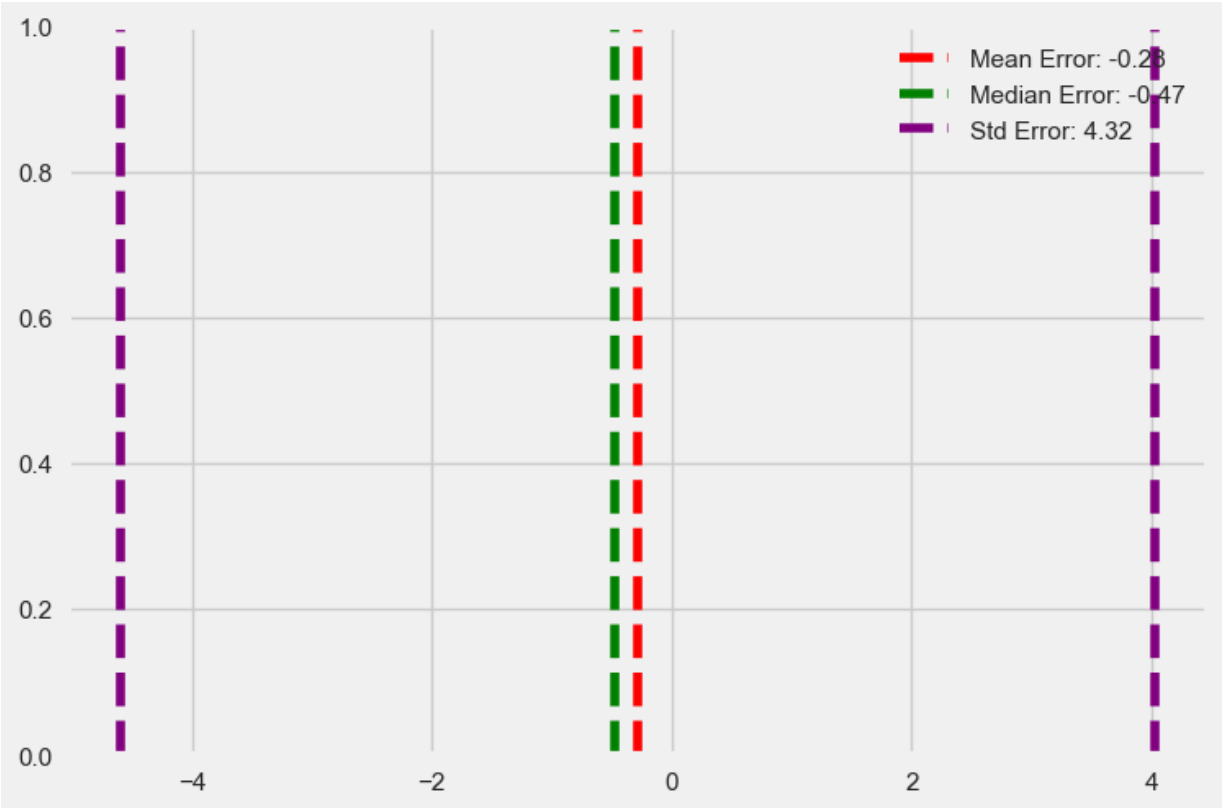
```
Hybrid Model (Train) - R^2: 0.9833186417189904
Hybrid Model (Test) - R^2: 0.9351907752156532
Hybrid Model (Test) - RMSE: 4.326527894643483
Hybrid Model CV Score: 0.9290091276963839
```








```
Approximately 79.44% of errors are within one standard deviation of the mean.
```



In [ ]:
```
import pickle

# Save models
```

```
pickle.dump(xgb_model2, open("xgb_model2.sav", "wb"))
pickle.dump(lgb_model2, open("lgb_model2.sav", "wb"))
```

In [ ]:
```
# Load models
xgb_model2_loaded = pickle.load(open("xgb_model2.sav", "rb"))
lgb_model2_loaded = pickle.load(open("lgb_model2.sav", "rb"))

# Make predictions with each model
xgb_ypred2 = xgb_model2_loaded.predict(X)
lgb_ypred2 = lgb_model2_loaded.predict(X)

# Average predictions
hybrid_ypred2 = (xgb_ypred2 + lgb_ypred2) / 2
```

In [ ]:
```
#save model 2


filename = 'final_model.sav'
pickle.dump(xgb_model, open(filename,'wb'))
```

In [ ]:
```
#loding the save model2

loaded_model = pickle.load(open('final_model.sav', 'rb'))
```

Throughout this project, I've conducted a comprehensive exploratory data analysis (EDA), followed by rigorous data preprocessing, which led to the training of various regression models. After careful evaluation, I identified two standout performers: XGBoost Regression and a hybrid model that combines the strengths of XGBoost and LightGBM.

The selected models demonstrated exceptional performance, each achieving noteworthy accuracy on the test datasets, and maintaining this high level of performance during a 10-fold cross-validation process. Moreover, the hybrid model exhibited lower mean and median error rates compared to its counterparts, indicating a higher degree of reliability and consistency.

Nonetheless, understanding that there's always room for improvement, I decided to further refine these models. I am currently applying different feature engineering techniques and advanced data preprocessing methods. Techniques such as Principal Component Analysis (PCA) for dimensionality reduction, Synthetic Minority Over-sampling Technique (SMOTE) for addressing imbalanced data, and recursive feature elimination (RFE) for feature selection are under consideration. Each method is aimed at improving the model's performance by managing overfitting, enhancing the balance of the dataset, and focusing the model's attention on the most significant features.

Upon fine-tuning these models, I am saving them for deployment in a real-world testing environment. This crucial step enables the optimization of these models in a live setting.

In conclusion, this project has demonstrated the robust potential of machine learning techniques for predicting concrete strength with high accuracy. However, it's worth noting

that the actual strength may deviate by a margin of ±5 MPa due to various factors not accounted for in the models. Consequently, the model's predictions should be used as guiding estimates rather than absolute figures.

As I continue to refine these models with advanced techniques and explore other potential models, I'm eager to share further developments. The valuable experience gained from this project lays a solid foundation for continued work in this field. As we move forward, I look forward to the enhancements and insights that this continuous improvement will bring.

In [ ]: