Bitcoin closing value Prediction

By: Humaiz Javed

Contents

ABSTRACT

This project delves into the dynamic realm of cryptocurrency price prediction by employing a hybrid approach merging Artificial Neural Networks (ANN) and Support Vector Machines (SVM) and other machine learning algorithms. The study focuses on Bitcoin, a leading digital currency, leveraging Python-based tools and libraries for preprocessing and modeling. The preprocessing stage involves comprehensive data normalization and feature engineering to enhance model performance. The ANN architecture is tailored to capture intricate nonlinear patterns within historical Bitcoin price data, while SVM complements this by discerning complex relationships for classification tasks. Through rigorous evaluation and comparison of these models, this research aims to ascertain their efficacy in predicting Bitcoin price movements, contributing valuable insights to the challenging domain of cryptocurrency forecasting.

Additionally, the dataset undergoes rigorous cleaning procedures to handle missing values, outliers, and potential noise, bolstering the robustness of subsequent analyses. Feature engineering plays a pivotal role, involving the creation of lagged variables, rolling averages, and technical indicators tailored to cryptocurrency markets. This tailored feature set seeks to encapsulate intricate temporal patterns and dependencies within Bitcoin price data, empowering the models to extract meaningful insights.

INTRODUCTION

Cryptocurrency markets, especially Bitcoin, have captivated global attention with their volatility. Predicting Bitcoin prices is challenging due to various factors influencing its value. This project focuses on using several Machine Learning algorithms to forecast Bitcoin prices. But before diving into modeling, it emphasizes preparing the data for accurate predictions.

Data preprocessing is crucial for reliable predictions. This involves cleaning the dataset by handling missing values and outliers. Scaling techniques ensure fairness among different features, while crafting new features helps the models understand complex patterns in Bitcoin price movements. Additionally, incorporating sentiment analysis from social media and news sources enriches the dataset, considering external factors affecting Bitcoin's market behavior.

By combining these preprocessing techniques, the aim is to empower models to make informed predictions about Bitcoin price movements.

IMPLEMENTATION

Libraries

In [4]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import numpy as np
from scipy import stats
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVR
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.neural_network import MLPRegressor
```

Above are the Libraries that have been used in the project to ensure that the data from the csv file is collected, processed and plotted accurately. Sklearn and numpy ensure accurate plotting of the data and calculation where required.

Importing and understanding the dataset

In [6]:
```
data = pd.read_csv("C://Users//Dell//Desktop//main.csv")
```

In [7]:
```
data
```

Out[7]:

	Open Time	Open	High	Low	Close	Volume	Close Time	Quote asset volume	Number of trades	Taker buy base asset volume	Taker buy quote asset volume
0	1.609460e+12	28923.63	28961.66	28913.12	28961.66	27.457032	1.609460e+12	7.943820e+05	1292.0	16.777195	485390.8268
1	1.609460e+12	28961.67	29017.50	28961.01	29009.91	58.477501	1.609460e+12	1.695803e+06	1651.0	33.733818	978176.4682
2	1.609460e+12	29009.54	29016.71	28973.58	28989.30	42.470329	1.609460e+12	1.231359e+06	986.0	13.247444	384076.8545
3	1.609460e+12	28989.68	28999.85	28972.33	28982.69	30.360677	1.609460e+12	8.800168e+05	959.0	9.456028	274083.0751
4	1.609460e+12	28982.67	28995.93	28971.80	28975.65	24.124339	1.609460e+12	6.992262e+05	726.0	6.814644	197519.3749
...	...	...	...	...	...	...	...	...	...	...	...
188312	1.620790e+12	57517.42	57526.28	57485.00	57485.07	42.575735	1.620790e+12	2.448258e+06	1195.0	15.319691	880913.0908
188313	1.620790e+12	57485.07	57496.42	57466.75	57481.49	34.205467	1.620790e+12	1.966194e+06	1096.0	15.971891	918058.8162
188314	1.620790e+12	57477.18	57509.99	57458.18	57470.00	30.211789	1.620790e+12	1.736514e+06	955.0	13.054229	750364.5773
188315	1.620790e+12	57470.00	57470.01	57400.00	57450.90	45.354728	1.620790e+12	2.605080e+06	1559.0	12.615628	724559.2330
188316	1.620790e+12	57450.89	57475.66	57435.51	57450.19	14.168318	1.620790e+12	8.140594e+05	730.0	7.247751	416412.0222

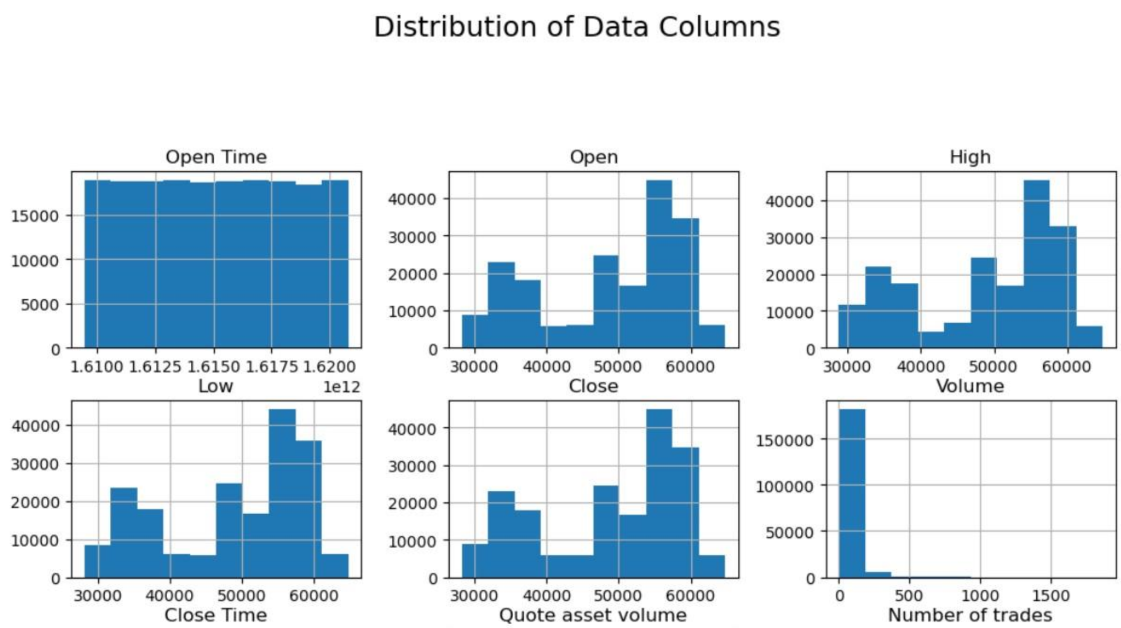
188317 rows × 11 columns

In [9]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 188317 entries, 0 to 188316
Data columns (total 11 columns):
 #   Column                        Non-Null Count   Dtype
---  ------                        --------------   -----
 0   Open Time                     188316 non-null  float64
 1   Open                          188315 non-null  float64
 2   High                          188304 non-null  float64
 3   Low                           188316 non-null  float64
 4   Close                         188313 non-null  float64
 5   Volume                        188307 non-null  float64
 6   Close Time                    188309 non-null  float64
 7   Quote asset volume            188312 non-null  float64
 8   Number of trades              188311 non-null  float64
 9   Taker buy base asset volume   188315 non-null  float64
 10  Taker buy quote asset volume  188316 non-null  float64
dtypes: float64(11)
memory usage: 15.8 MB
```

DISTRIBUTION OF DATA:

In [7]:
```
data.hist(figsize=(12, 10))
plt.suptitle("Distribution of Data Columns", y=1.02, size=18)
plt.show()
```



This generates a grid of histograms, each representing the distribution of values within individual columns of the dataset. The histograms provide a visual depiction of the spread and frequency of data across various features, aiding in the initial understanding of the dataset's characteristics. The title 'Distribution of

Data Columns' crowns this visual summary, offering a collective insight into the diverse distributions present in the dataset.

Detecting and Handling Missing Values

In [8]:
```
data.isnull().sum()
```

Out[8]:
```
Open Time                      1
Open                           2
High                          13
Low                            1
Close                          4
Volume                        10
Close Time                     8
Quote asset volume             5
Number of trades               6
Taker buy base asset volume    2
Taker buy quote asset volume   1
dtype: int64
```

In [9]:
```
medians = data.median()
data.fillna(medians, inplace=True)
```

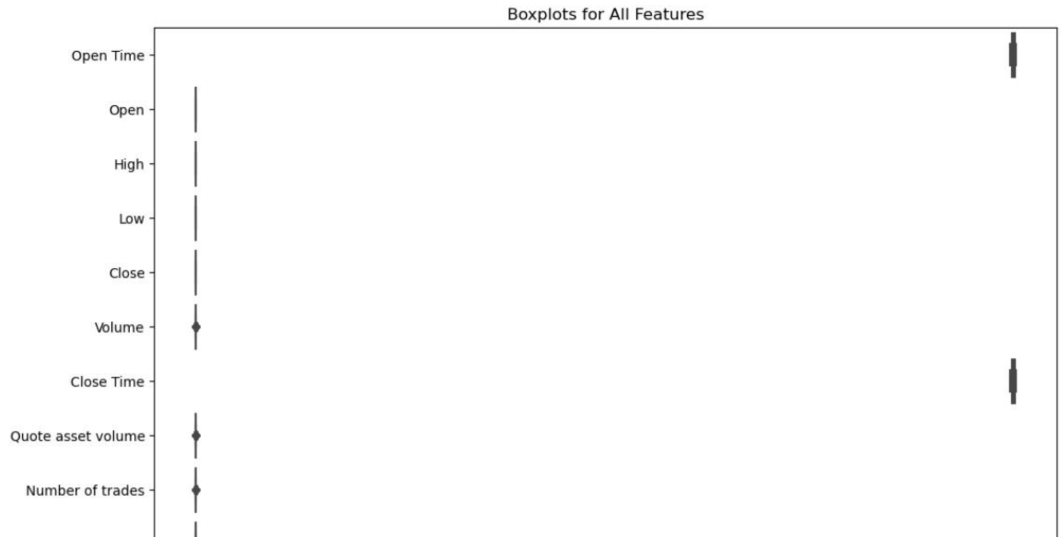
In [10]:
```
data.isnull().sum()
```

Out[10]:
```
Open Time                      0
Open                           0
High                           0
Low                            0
Close                          0
Volume                         0
Close Time                     0
Quote asset volume             0
Number of trades               0
Taker buy base asset volume    0
Taker buy quote asset volume   0
```

It calculates and returns the sum of missing or null values present in each column of the dataset 'data'. It provides a quick summary, indicating the count of missing values per column. calculates the median values for each column in the dataset data. Subsequently, it replaces any missing or null values in the dataset with the computed median values, After performing the imputation using the median values, this line of code checks for missing values in each column of the 'data' dataset. It counts and displays the sum of null values per column.

Detecting Outliers

```
In [11]: plt.figure(figsize=(12, 8))
         sns.boxplot(data=data, orient="h")
         plt.title("Boxplots for All Features")
         plt.show()
```



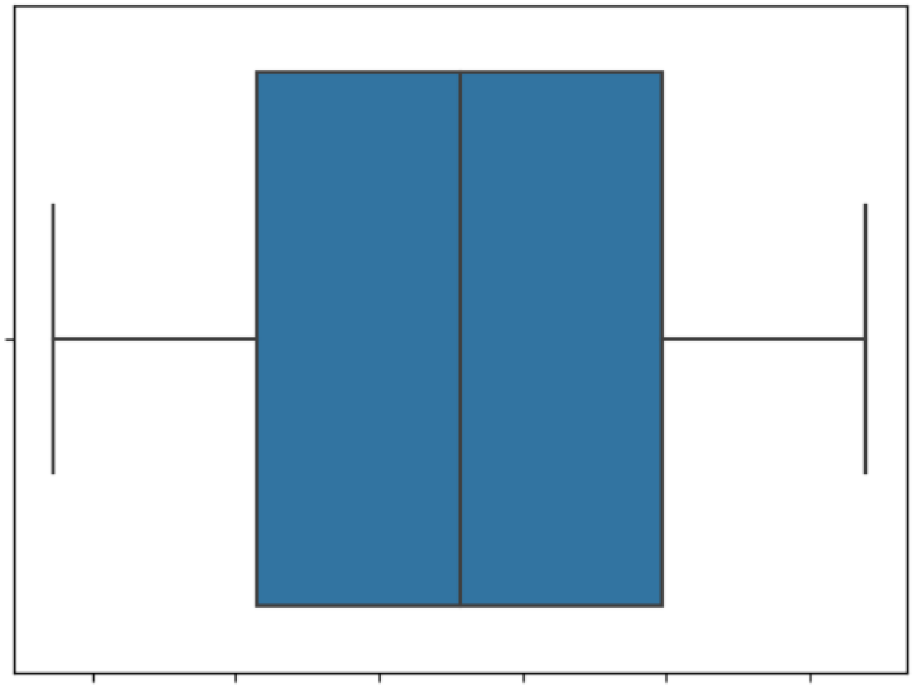
The above code generates a horizontal boxplot using Seaborn (sns) to visually identify outliers present across all features in the dataset. Each boxplot represents the distribution of values within individual columns. The length of the box signifies the interquartile range (IQR), while the whiskers extend to points within 1.5 times the IQR. Observations lying beyond the whiskers are considered outliers and are plotted individually. This visualization aids in comprehending the spread of data, identifying potential anomalies or extreme values that might require further investigation.

It utilizes Seaborn's **boxplot** function to generate a boxplot specifically for the 'Open Time' feature within the dataset. The resulting visualization displays the distribution of values contained in the 'Open Time' column.

This methodology, applied to various features within the dataset, offers a feature-specific perspective on data distribution, enabling a quick assessment of variability and the presence of outliers in each attribute.

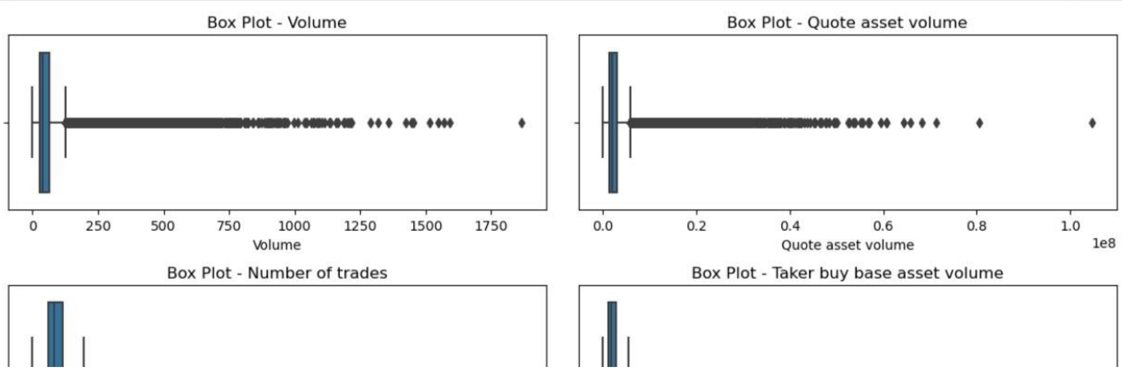
In [12]:
```python
sns.boxplot(x=data['Open Time'])
```

Out[12]: <Axes: xlabel='Open Time'>



Exploratory Data Analysis (EDA)

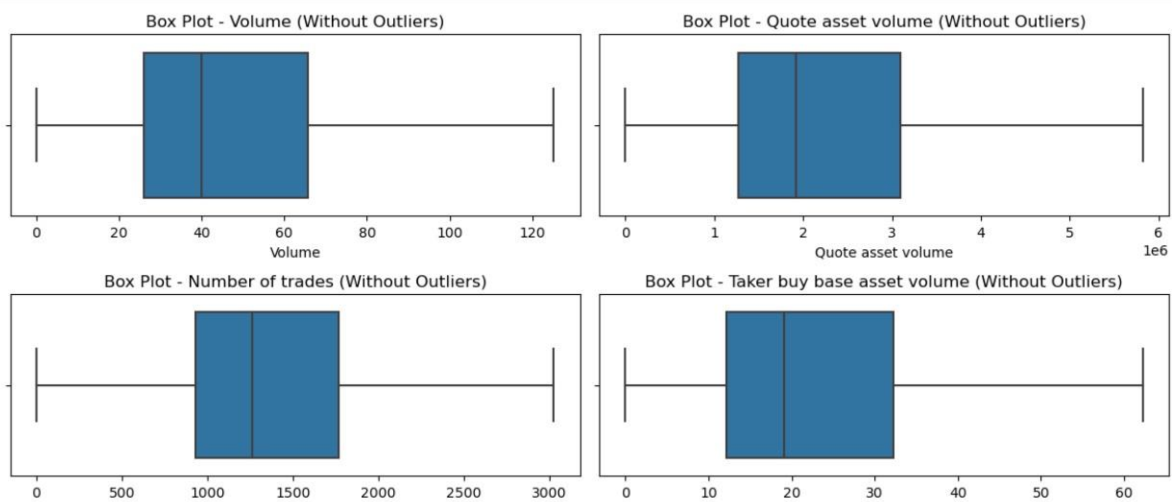
[23]:
```python
outlier_columns = ['Volume', 'Quote asset volume', 'Number of trades', 'Taker buy base asset volume', 'Taker buy quote asset volu
num_columns = len(outlier_columns)
num_rows = (num_columns + 1) // 2
plt.figure(figsize=(12, 8))
for i, column in enumerate(outlier_columns):
    plt.subplot(num_rows, 2, i + 1)
    sns.boxplot(x=data[column])
    plt.title(f'Box Plot - {column}')

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



This part conducts an Exploratory Data Analysis (EDA) by visualizing the distributions of numerical features in the dataset through boxplots. The 'outlier_columns' list contains the names of columns contains numerical data likely to have outliers.

it arranges these columns into a grid layout for visualization. For each column in 'outlier_columns', a boxplot is created using Seaborn's boxplot function. This visualization depicts the spread of values within each feature, showcasing median, quartiles, and potential outliers. Titles are added to each boxplot denoting the specific feature being represented. this approach allows for a systematic examination of the distribution characteristics across these selected numerical columns. It enables rapid identification of variability, central tendencies, and potential outlier presence within individual attributes.

```
n [24]: outlier_columns = ['Volume', 'Quote asset volume', 'Number of trades', 'Taker buy base asset volume', 'Taker buy quote asset vo
num_columns = len(outlier_columns)
num_rows = (num_columns + 1) // 2
plt.figure(figsize=(12, 8))
for i, column in enumerate(outlier_columns):
    plt.subplot(num_rows, 2, i + 1)
    sns.boxplot(x=data[column], showfliers=False)
    plt.title(f'Box Plot - {column} (Without Outliers)')
plt.tight_layout()
plt.show()
```



Showfliers omit the outliers and display the data without it as it can be seen in the above figure.

Detecting Noisy data

When we refer to "noisy data" for this dataset, we are specifically talking about data points that fall outside a certain range defined by the interquartile range (IQR). These points are commonly known as "outliers." Therefore such values won't be removed from the training dataset.

In [25]:
```python
outlier_threshold = 1.5
fig, axes = plt.subplots(nrows=len(data.columns), ncols=1, figsize=(10, 6 * len(data.columns)))
all_noisy_data = pd.DataFrame(columns=data.columns)
for i, column in enumerate(data.columns):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    noisy_data = data[(data[column] < Q1 - outlier_threshold * IQR) | (data[column] > Q3 + outlier_threshold * IQR)]
    axes[i].boxplot(data[column], vert=False)
    axes[i].set_title(f'Box Plot - {column}')
    all_noisy_data = pd.concat([all_noisy_data, noisy_data])
print("All Noisy Data Points:")
print(all_noisy_data)
plt.tight_layout()
plt.show()
```

```
All Noisy Data Points:
           Open Time      Open      High       Low     Close      Volume  \
14      1.609460e+12  28716.85  28764.23  28690.17  28752.80  156.587294
70      1.609460e+12  29117.19  29157.98  29115.76  29155.85  151.322379
71      1.609460e+12  29155.86  29200.00  29149.23  29193.47  201.436205
72      1.609460e+12  29193.46  29246.67  29187.87  29239.21  191.897707
75      1.609460e+12  29384.97  29385.00  29296.74  29357.15  285.135915
...              ...       ...       ...       ...       ...         ...
188264  1.620790e+12  57679.12  57719.64  57654.26  57713.33  214.481315
188267  1.620790e+12  57739.85  57814.14  57733.42  57785.02   92.674578
188277  1.620790e+12  57756.95  57870.70  57755.12  57859.92   99.550159
188278  1.620790e+12  57859.92  57900.00  57840.72  57889.65   73.089038
188279  1.620790e+12  57887.19  58000.01  57870.82  57909.47  486.246880
```

This code finds the outliers in each dataset column using the Interquartile Range (IQR) method. It calculates typical value ranges for each column and then flags values significantly far from these typical ranges as potential outliers.this displays boxplots for each column, highlighting any potential outliers found beyond a threshold. It also gathers all these suspected outliers into a table for easy reference, giving an overview of where these anomalies appear across the dataset. This helps spot and investigate data points that might not fit the usual patterns, aiding in further examination or cleaning of potentially noisy data.

Feature Selection (Correlation Coefficient Technique)

n [26]:
```python
target_variable = 'Close'
```

n [27]:
```python
numerical_features = data.select_dtypes(include='number').columns.tolist()
```

n [28]:
```python
data_num = data[numerical_features + [target_variable]]
```

n [29]:
```python
correlation_matrix = data_num.corr()
```

n [30]:
```python
print(correlation_matrix)
```

```
                           Open Time      Open      High       Low  \
Open Time                   1.000000  0.855567  0.855462  0.855685
Open                        0.855567  1.000000  0.999958  0.999985
High                        0.855462  0.999958  1.000000  0.999951
Low                         0.855685  0.999985  0.999951  1.000000
Close                       0.855554  0.999963  0.999948  0.999975
Volume                     -0.188588 -0.197823 -0.195731 -0.200297
Close Time                  0.999935  0.855540  0.855435  0.855658
Quote asset volume         -0.009224  0.011088  0.013250  0.008536
Number of trades           -0.072437 -0.052305 -0.050072 -0.054944
Taker buy base asset volume -0.176990 -0.183445 -0.180865 -0.184982
Taker buy quote asset volume -0.012316  0.008817  0.011463  0.007240
Close                       0.855554  0.999963  0.999948  0.999975

                              Close    Volume  Close Time  \
Open Time                  0.855554 -0.188588    0.999935
Open                       0.999963 -0.197823    0.855540
High                       0.999948 -0.195731    0.855435
Low                        0.999975 -0.200297    0.855658
Close                      1.000000 -0.197951    0.855527
Volume                    -0.197951  1.000000   -0.188598
```

This format is designed to analyze the relationships between numerical features and a specific target variable, 'Close', within the dataset. Initially, it selects all numerical columns, including the 'Close' column, creating a subset Subsequently, by utilizing the corr() function, it computes the correlation matrix ('correlation_matrix'). This correlation matrix offers a comprehensive view of how each numerical attribute correlates with the target variable, 'Close'. The numerical values within the matrix represent the degree and direction of association between every pair of numerical features.

```
In [40]: threshold = 0.5

In [41]: highly_correlated_features = set()

In [42]: for i in range(len(correlation_matrix.columns)):
             for j in range(i):
                 if abs(correlation_matrix.iloc[i, j]) > threshold:
                     colname = correlation_matrix.columns[i]
                     highly_correlated_features.add(colname)

In [44]: print("Highly correlated features:", highly_correlated_features)

         Highly correlated features: {'High', 'Close', 'Close Time', 'Taker buy base asset volume', 'Number of trades', 'Low', 'Taker bu
         y quote asset volume', 'Quote asset volume', 'Open'}
```

The code iterates through the correlation matrix, checking for highly correlated features based on a specified threshold. It examines pairs of columns in the matrix, and if the absolute correlation value surpasses the threshold, it adds the respective column names to a set named 'highly_correlated_features'. then, it displays the identified highly correlated features. This process helps pinpoint attributes strongly correlated with each other, assisting in potential feature selection or identification of multicollinearity issues within the dataset.

Regression Analysis (Linear Regression)

In [35]:
```
X = data[['Open', 'High', 'Low', 'Volume']]
y = data['Close']
```

In [36]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [37]:
```
model = LinearRegression()
```

In [38]:
```
model.fit(X_train, y_train)
```

Out[38]:
```
LinearRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [39]:
```
y_pred = model.predict(X_test)
```

In [40]:
```
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

In [41]:
```
print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
print(f'R-squared: {r2}')
```

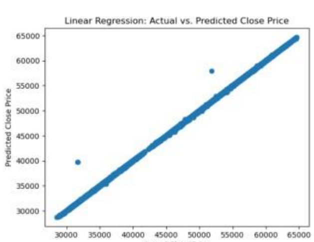
```
Mean Absolute Error: 23.24500075969155
Mean Squared Error: 5491.837867215096
R-squared: 0.9999435545729385
```

In [42]:
```
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Close Price')
plt.ylabel('Predicted Close Price')
```

This code implements a basic linear regression analysis on a dataset. It splits the data into predictors ('Open', 'High', 'Low', 'Volume') and the target ('Close'). Then, it divides the dataset into training and testing sets.

Using the training data, it trains a Linear Regression model to predict 'Close' prices based on the provided features. The model then predicts 'Close' prices on the test data. After predictions, it calculates three metrics - Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared (R^2). These metrics measure how accurately the model predicts 'Close' prices compared to the actual values.

The code displays these metrics, providing an assessment of the model's performance. Additionally, it generates a scatter plot showing how the model's predicted 'Close' prices align with the actual 'Close' prices, offering a visual understanding of the model's predictive capabilities.

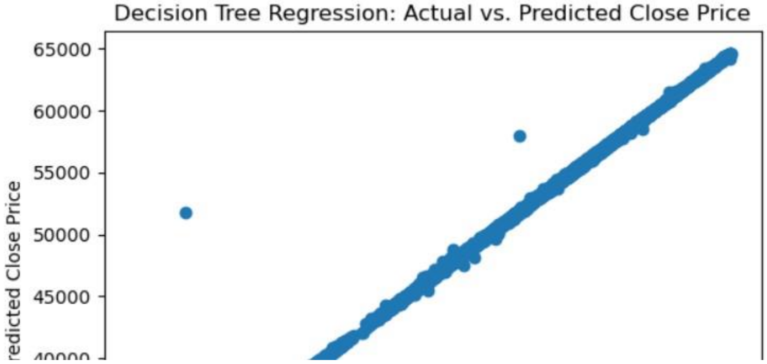


Decision Tree

```
[43]: decision_tree = DecisionTreeRegressor(random_state=42)

[44]: decision_tree.fit(X_train, y_train)

t[44]: DecisionTreeRegressor(random_state=42)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
[45]: y_pred = decision_tree.predict(X_test)

[46]: plt.scatter(y_test, y_pred)
      plt.xlabel('Actual Close Price')
      plt.ylabel('Predicted Close Price')
      plt.title('Decision Tree Regression: Actual vs. Predicted Close Price')
      plt.show()
```



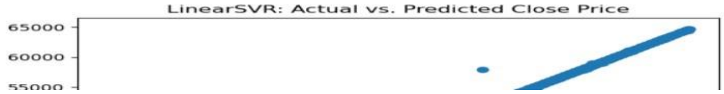
initializes a Decision Tree Regression model in Python using scikit-learn's DecisionTreeRegressor class. The random_state=42 parameter sets the random number generator's seed, ensuring reproducibility of results. A Decision Tree Regression model works by recursively partitioning the dataset into smaller subsets based on feature values. It creates a tree-like structure where each internal node represents a feature, each branch represents a decision based on that feature, and each leaf node represents the predicted output.

Support Vector Machine (SVM)

```
[49]: svr = LinearSVR()

[50]: svr.fit(X_train, y_train)

E:\humaiz1\anaconda\Lib\site-packages\sklearn\svm\_classes.py:32: FutureWarning: The default value of `dual` will cha
True` to ``auto`` in 1.5. Set the value of `dual` explicitly to suppress the warning.
  warnings.warn(
E:\humaiz1\anaconda\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, in
number of iterations.
  warnings.warn(

t[50]: LinearSVR()
```

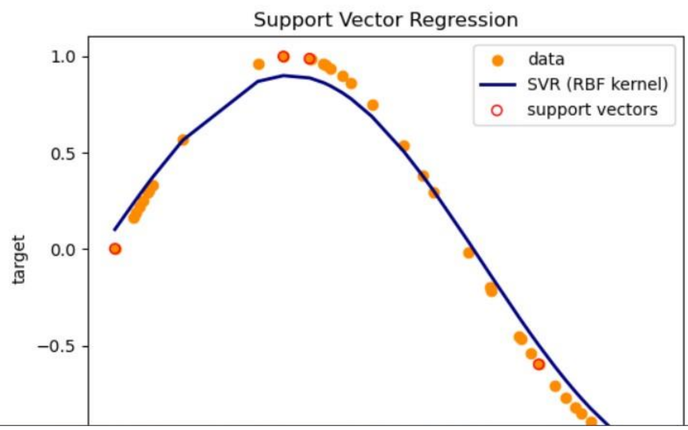
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
[51]: y_pred = svr.predict(X_test)

[52]: plt.scatter(y_test, y_pred)
      plt.xlabel('Actual Close Price')
      plt.ylabel('Predicted Close Price')
      plt.title('LinearSVR: Actual vs. Predicted Close Price')
      plt.show()
```



The SVM algorithm is also applied on the trained data which shows that all the algorithms can be efficiently produced in the dataset by using the above trained models.

```
[53]: X = np.sort(5 * np.random.rand(40, 1), axis=0)
      y = np.sin(X).ravel()
      svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1)
      svr_rbf.fit(X, y)
      y_pred = svr_rbf.predict(X)
      plt.scatter(X, y, color='darkorange', label='data')
      plt.plot(X, y_pred, color='navy', lw=2, label='SVR (RBF kernel)')
      plt.scatter(X[svr_rbf.support_], y[svr_rbf.support_], facecolors='none', edgecolors='r', label='support vectors')
      plt.xlabel('data')
      plt.ylabel('target')
      plt.title('Support Vector Regression')
      plt.legend()
      plt.show()
```



Above model demonstrates Support Vector Regression (SVR) using the radial basis function (RBF) kernel, a popular technique for nonlinear regression tasks. It generates synthetic data ('X' and 'y') by creating a sorted array of random values and computing the sine function of 'X' to serve as the target variable. The SVR model, initialized with the RBF kernel, is then trained on this synthetic data. The RBF kernel is effective for capturing nonlinear relationships between variables in SVR. The parameters ('C', 'gamma', 'epsilon') are set to control the model's complexity, influence of data points (gamma), and margin of error (epsilon).

After training, the model predicts the target variable ('y_pred') based on the 'X' values. The code visualizes the results using a scatter plot to represent the original data points and overlays the SVR model's predicted values as a curve. Additionally, it identifies the support vectors, representing the data points crucial for defining the SVR model.

Aritificial Neural Network (ANN)

```
54]: ann_model = MLPRegressor(hidden_layer_sizes=(100, 50), max_iter=1000, random_state=42)
ann_model.fit(X_train, y_train)
y_pred = ann_model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("Mean Absolute Error:", mae)
print("Mean Squared Error:", mse)
print("R-squared:", r2)
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Close Price')
plt.ylabel('Predicted Close Price')
plt.title('ANN: Actual vs. Predicted Close Price')
plt.show()
```

```
Mean Absolute Error: 34.80065188964728
Mean Squared Error: 5216.561050167356
R-squared: 0.9999463838839767
```

Model Prediction

```
5]: def predict_closing_value(user_input, model):
    relevant_features = ['Open', 'High', 'Low', 'Volume']
    user_values = np.array([user_input[feature] for feature in relevant_features]).reshape(1, -1)
    predicted_value = model.predict(user_values)
    return predicted_value[0]
def get_user_input():
    print("\nEnter values for each feature:")
    user_input = {}
    relevant_features = ['Open', 'High', 'Low', 'Volume']
    for feature in relevant_features:
        while True:
            try:
                value = float(input(f"{feature}: "))
                user_input[feature] = value
                break
            except ValueError:
                print("Invalid input. Please enter a numeric value.")
    return user_input
user_input = get_user_input()
prediction = predict_closing_value(user_input, model)
print("\nThe model predicts the closing value:", prediction)
```

```
Enter values for each feature:
Open: 28923.63
High: 28961.66
Low: 28913.12
Volume: 27.457032

The model predicts the closing value: 28942.51961678468
```

This code includes two functions to predict the closing value of a financial asset based on user input using a pre-trained machine learning model. The predict_closing_value function takes in user-inputted data and a machine learning model. It extracts relevant features ('Open', 'High', 'Low', 'Volume') from the user input, organizes them into an array, and predicts the closing value using the provided model.

The get_user_input function interacts with the user, prompting them to input values for each relevant feature. It ensures valid numeric inputs for each feature and stores them in a dictionary ('user_input').

The user_input dictionary is populated with user-provided values for 'Open', 'High', 'Low', and 'Volume'. Then, the predict_closing_value function utilizes this input alongside the model to forecast the closing value. Finally, the code prints the predicted closing value based on the provided user inputs and the model's prediction capabilities.

CONCLUSION

This project navigated the world of cryptocurrency price prediction, employing machine learning methodologies to forecast Bitcoin price movements. From preprocessing data to employing diverse models like Artificial Neural Networks (ANN), Support Vector Machines (SVM), and Decision Trees, we aimed to capture intricate patterns within Bitcoin's fluctuating prices.

Our analysis involved visual exploration, outlier detection, and correlation assessments among features. Regression models enabled price predictions, shedding light on potential trends and relationships among key attributes.

The project's pinnacle was a user-centric interface, allowing individuals to input financial parameters and obtain Bitcoin closing value predictions. This interactive tool aimed to bridge the gap between complex models and practical user applications for cryptocurrency price forecasting.

In summary, this project demonstrated the application of machine learning techniques to tackle the volatility and complexities inherent in forecasting Bitcoin prices