BITCOIN PRICE FORECASTING

Team 18
Soham Shah (002837154) & Abhishek Mathukiya (002839155)

Introduction

Background

Bitcoin's market volatility represents both a risk and an opportunity for investors, analysts, and traders. With the increasing adoption of cryptocurrencies, understanding and predicting price movements of Bitcoin is crucial for risk management and investment strategy. By harnessing the power of extensive historical datasets, such as the one capturing over three million Bitcoin price records from 2017 to 2023, financial experts can employ advanced machine learning algorithms to identify trends, patterns, and anomalies that traditional analysis might miss.

This data-driven approach can provide deeper insights into the intricacies of market dynamics, enabling predictive models that can alert to potential price swings. By analyzing detailed attributes like open, high, low, and close prices in combination with trading volumes and timestamps, these models can unveil underlying factors that influence market behavior. The result is a more informed strategy that helps in mitigating risks associated with Bitcoin's notorious price fluctuations and in capitalizing on market trends, thus contributing to a more stable and secure cryptocurrency market environment.

Motivation

The motivation behind analyzing the Bitcoin price dataset is to develop a predictive model that can accurately forecast Bitcoin's price fluctuations. By understanding the intricate patterns within the historical price data, the model aims to anticipate market trends, aiding investors in making informed decisions. Effective predictions could minimize risks and maximize returns by timing the market more accurately. Furthermore, such a model could enhance market efficiency, contribute to academic research on cryptocurrency behavior, and potentially inform regulatory decisions for a more stable financial landscape in the digital age.

Goals

The primary goal is to create an accurate and reliable predictive model that can effectively forecast Bitcoin's price movements. This model is intended to enable stakeholders to understand, anticipate, and react to market trends, potentially leading to more informed decisions, reduced financial risk, and optimized investment strategies in the volatile cryptocurrency market.

1. Develop a predictive analytics model capable of forecasting Bitcoin's price movement by analyzing historical data from 2017 to 2023, using this extensive dataset to understand past trends and anticipate future changes.

2. Process and clean the dataset, ensuring the integrity of the data by handling any anomalies, missing values, or outliers that could skew the predictive model's accuracy.

3. Utilize a variety of statistical methods and machine learning algorithms to capture the complex dynamics of Bitcoin's price fluctuations, evaluating each model to find the most effective approach.

4. Determine the key factors that most significantly impact Bitcoin's price, such as trading volume or specific time intervals, and incorporate these into the model to enhance prediction precision.

5. Offer actionable insights and strategic recommendations based on the model’s findings to support investors, financial analysts, and market observers in making data-driven decisions in the cryptocurrency market.

Methodology

Data Cleaning and Preprocessing

Loading and Initial Cleanup

The datasets were first loaded into Pandas DataFrames, utilizing its robust data manipulation capabilities to handle the various aspects of the Bitcoin dataset, which includes features like prices, volumes, and timestamps from multiple exchanges. The initial cleanup process involved removing duplicate rows and rectifying any inconsistencies in data entries, which is crucial to avoid any analytical errors that could arise from incorrect or redundant data.

Handling Missing Values

Special attention was paid to managing missing data, particularly in the volumes and prices from certain exchanges where data might not be recorded consistently. Strategies such as forward filling (propagating the last observed non-null value) or interpolation were employed depending on the nature of the gap. This step was essential to ensure that our time series analysis would not be impacted by gaps in data.

Validation and Further Processing

Further validation steps included a detailed inspection for any misaligned data types or incorrectly formatted entries. This process involved adjusting data types for the datetime columns and ensuring numerical columns were correctly formatted and scaled if necessary. Additional cleaning steps were performed to align with the analytical requirements of the subsequent phases of the project, such as feature engineering and model fitting.

This detailed approach to data cleaning and preprocessing set a solid foundation for the exploratory data analysis and the application of machine learning models, ensuring the integrity and reliability of the subsequent analyses.

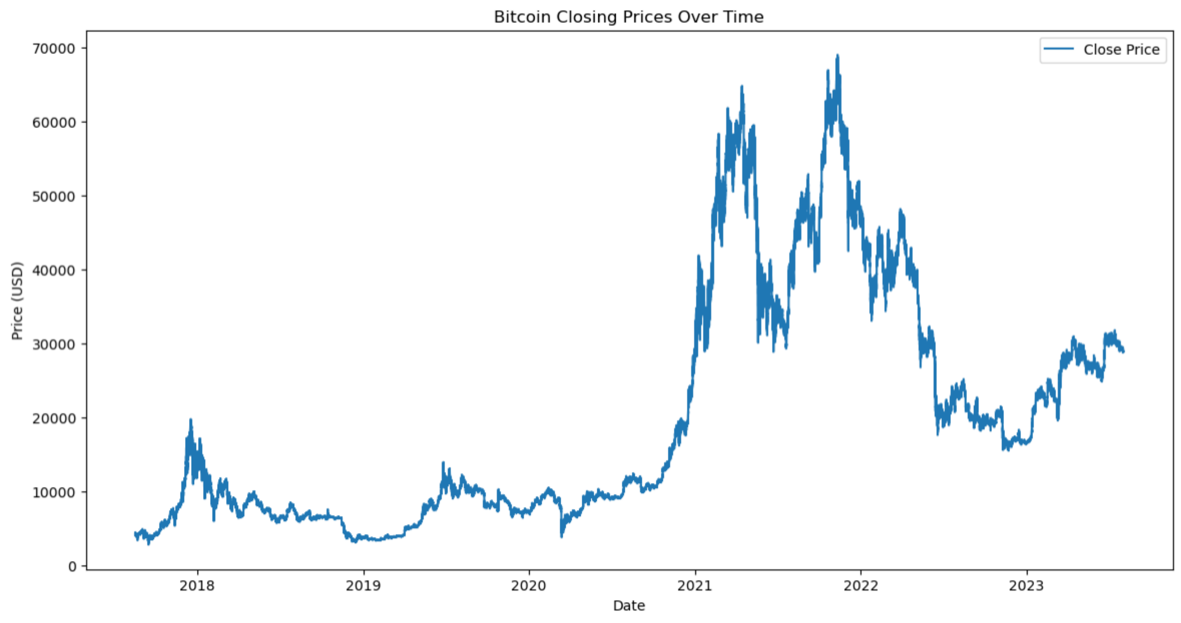
Exploratory Data Analysis

Our Exploratory Data Analysis (EDA) will present a detailed examination of the Bitcoin price dataset, which includes over 3 million records from 2017 to 2023. The analysis will:

- Describe the data, highlighting key features like timestamp, open, high, low, close prices, and trading volume.
- Assess the completeness of the data, identifying any missing values or outliers.
- Explore patterns in price movements on an intraday basis and across different time frames to uncover any recurring trends.
- Investigate periods of high volatility and potential correlations with market or global events.
- Examine long-term price trends to understand how Bitcoin's value has evolved over the years.
- Analyze the relationship between trading volume and price changes to determine the impact of market activity on price dynamics.
- Charts, graphs, and visual representations will be used to illustrate our findings, providing clear insights into Bitcoin's market behavior. The goal is to understand the nuances of the dataset thoroughly, forming the basis for predictive modeling and further research.

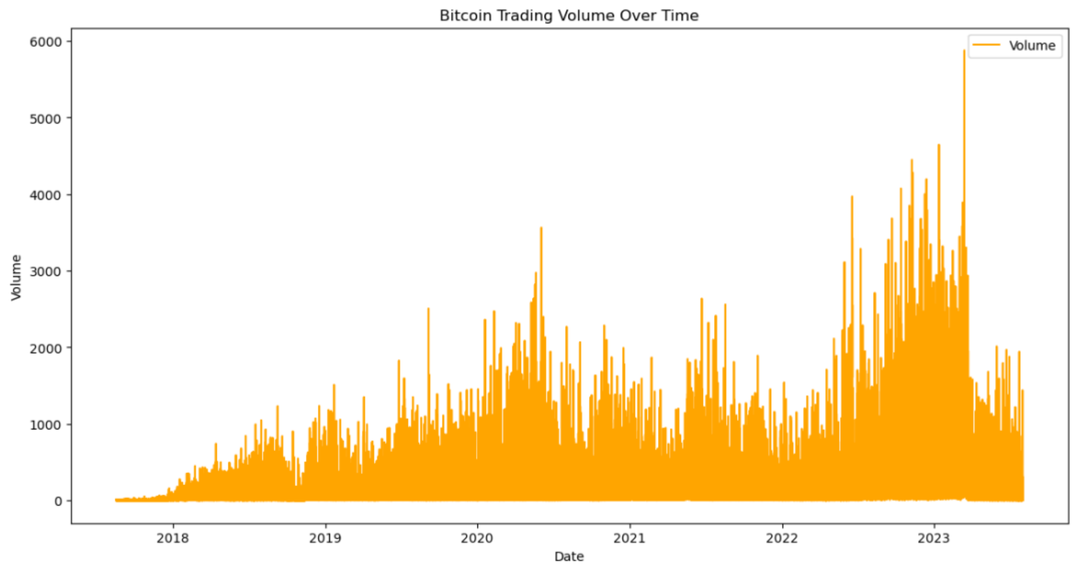
Bitcoin Closing Prices Over Time

the x-axis represents time, with dates ranging from the start of 2018 to partway through 2023. The y-axis represents the price of Bitcoin in USD, showing the closing price of Bitcoin for each day within that timeframe.



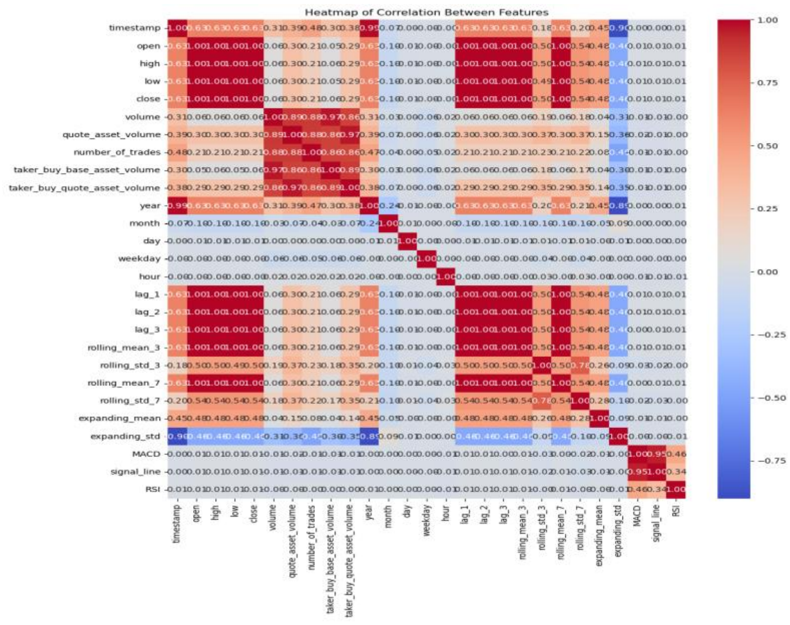
Bitcoin Trading Volume over Time

The area chart depicts Bitcoin's trading volume from 2018 to early 2023, showing a general increase in activity over time with notable spikes. The volatility suggests periods of heightened interest or market events, culminating in a significant peak in trading volume toward the end of the period, indicative of a surge in recent activity.



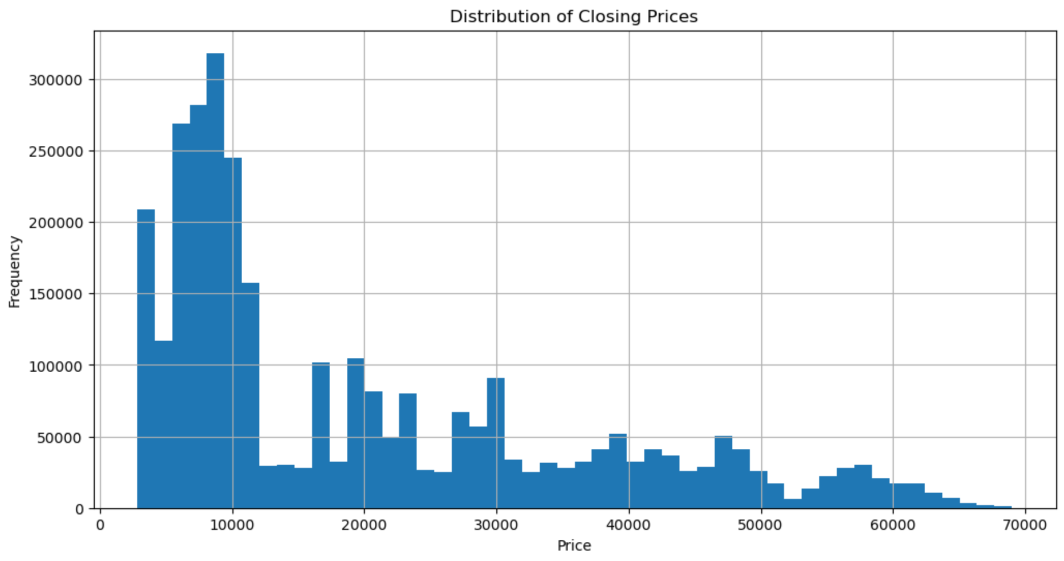
Heatmap of Correlation Between Features

The heatmap shows correlations between trading variables, with red indicating positive and blue showing negative correlations. High correlations exist among 'open', 'high', 'low', 'close', and 'volume', while time-based variables like 'hour' and 'weekday' show minimal correlation with market indicators. Technical indicators like 'MACD' and 'RSI' have mixed correlations with other features.



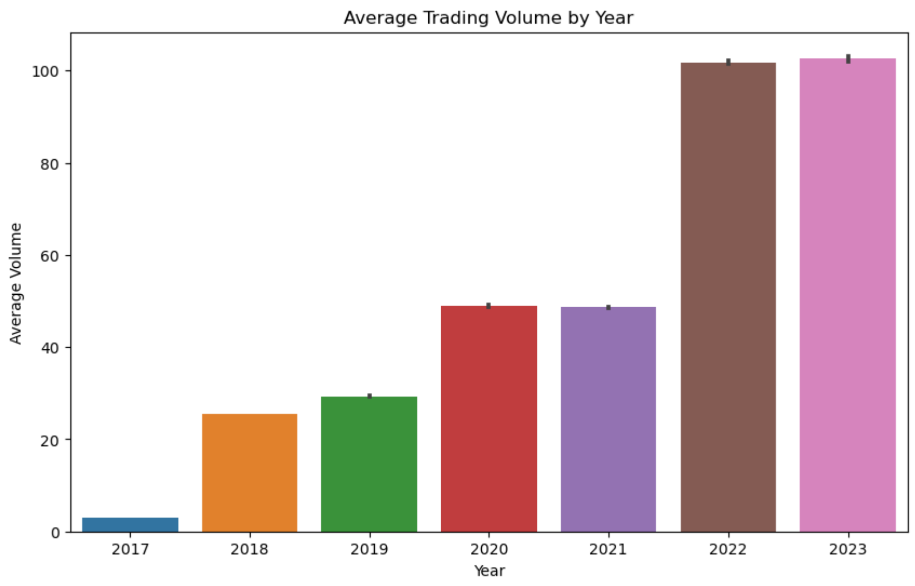
Distribution of Closing Prices

The histogram depicts the frequency distribution of closing prices, demonstrating a right-skewed trend. A significant concentration of closing prices falls within the 0 to 20,000 range, indicating this is the most common price bracket. As prices rise, the frequency of closings notably diminishes, highlighting that higher closing prices are less common. There are occasional upticks in frequency at higher price points, but these are relatively sparse, underscoring the rarity of such high closing prices in the dataset.

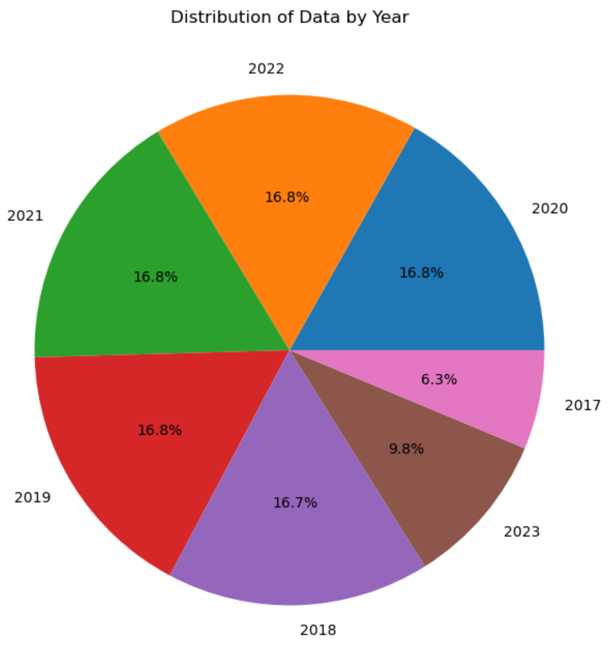


Average Trading Volume by Year

The bar chart reveals an ascending trend in average trading volume from 2017 to 2023, with each successive year showing an increase, and a notable surge starting in 2021. The years 2022 and 2023 display the highest volumes, with consistent variability indicated by the error bars throughout the years.

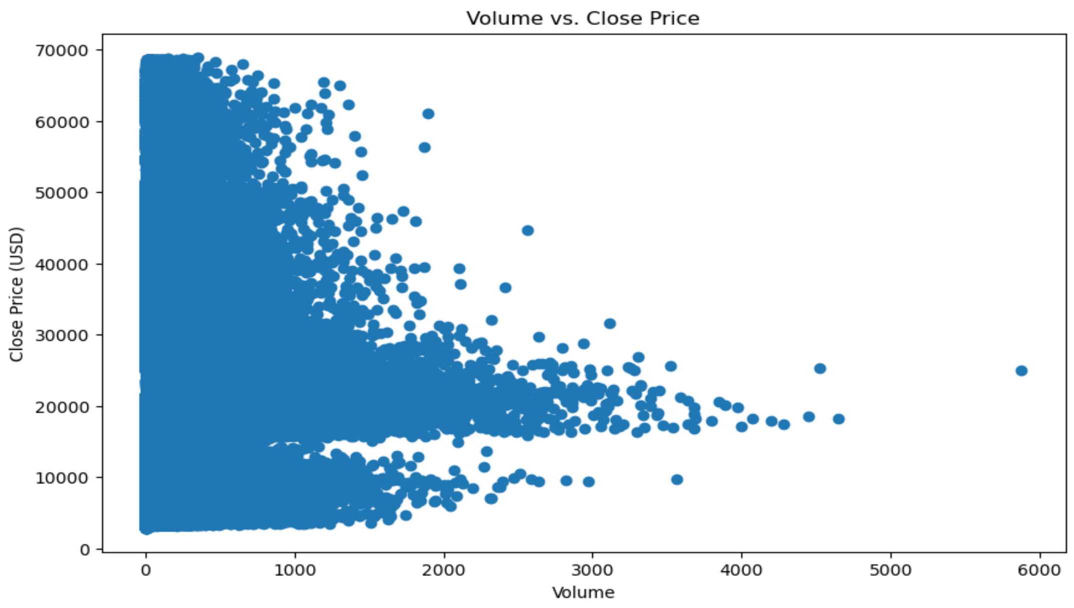


Distribution of Data by Year



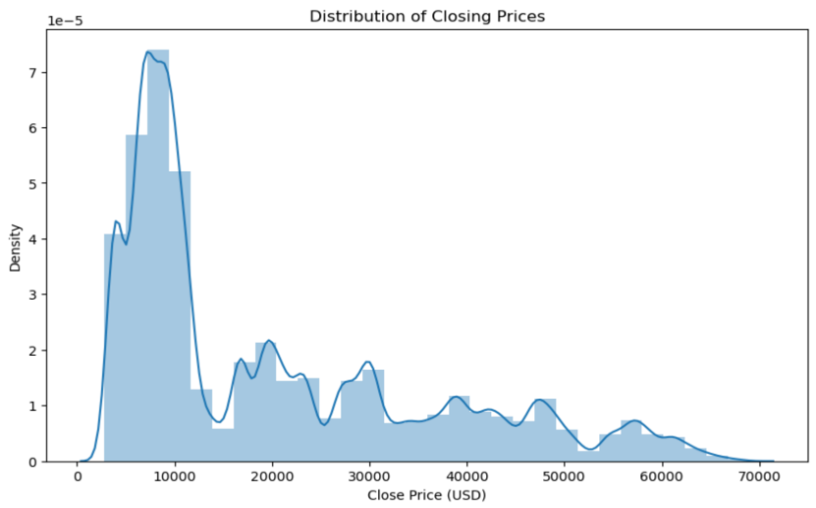
Volume vs Close Price

This scatter plot represents the relationship between trading volume and closing price. Data points are concentrated in the lower range of volume and price, suggesting that lower closing prices tend to have higher frequencies at lower volumes. There is a broad spread of data across the price range as volume increases, indicating variability in price that does not show a clear correlation with volume. Fewer data points exist at higher volumes, which could imply that such high-volume trades are less common. There are outliers present, particularly noticeable at the higher end of the closing price range, which are not associated with the highest volumes.



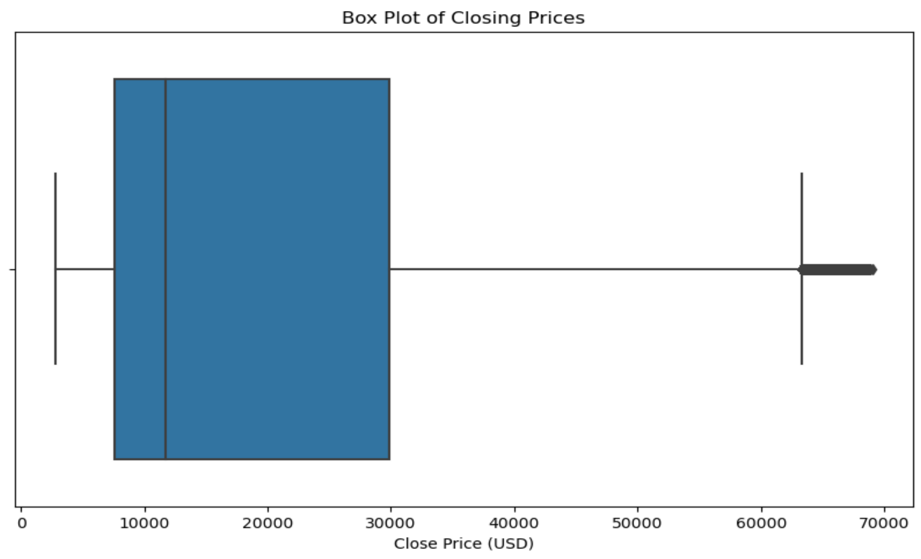
Distribution of Closing Prices

The density plot illustrates the distribution of closing prices, showing a peak in density at the lower end of the price scale, which then tapers off as the price increases. This peak indicates a higher concentration of closing prices in the lower range. Multiple smaller peaks at various price points suggest specific price levels at which closing prices cluster more frequently. Overall, the distribution is uneven, highlighting the variable nature of closing price occurrences across different price ranges.



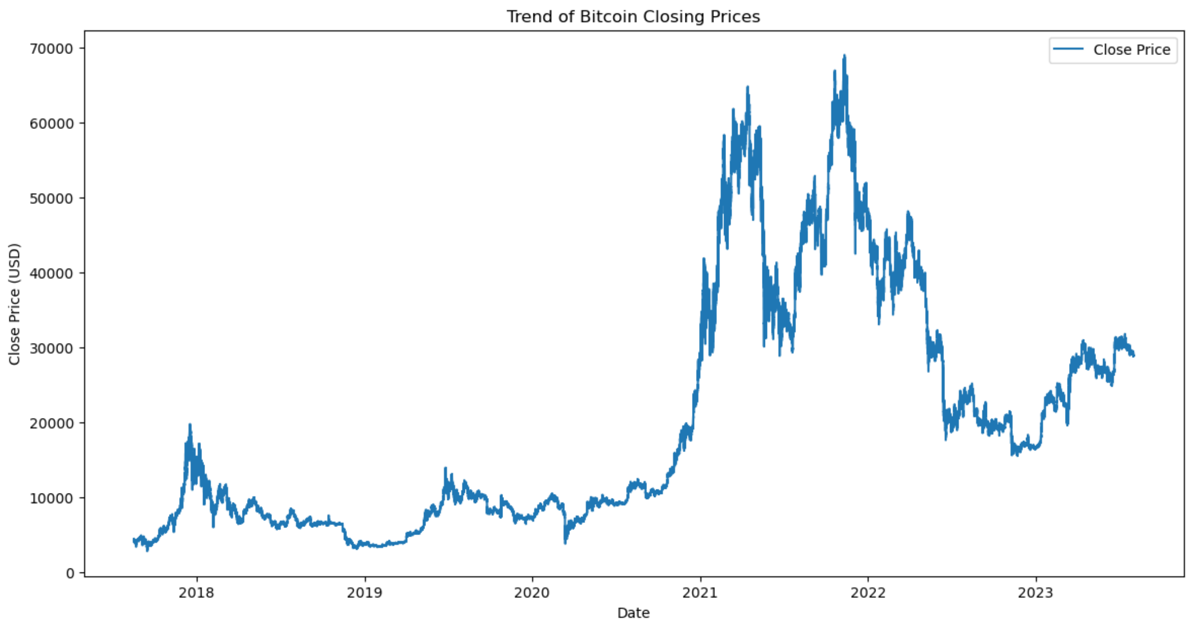
Box Plot of Closing Prices

The box plot visualizes the distribution of closing prices with most data concentrated within the lower quartile, suggesting a clustering of lower closing prices. The median value, indicated by the line within the box, is closer to the first quartile, reinforcing the skew towards lower prices. The 'whiskers' extend to show the full range of data, excluding potential outliers, and the plot features a long tail towards the higher end, indicating occasional closing prices that are much higher than the typical range.



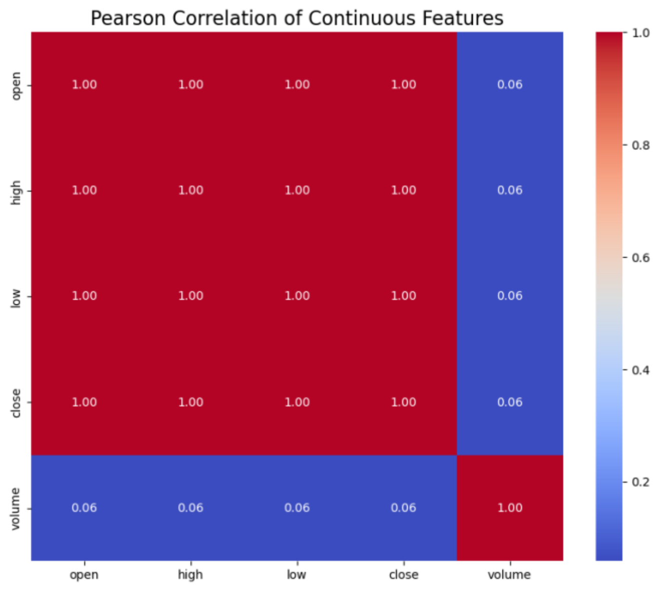
Trend of Bitcoin Closing Price

This graph traces the Bitcoin closing prices from 2018 to early 2023, highlighting the cryptocurrency's volatility. Prices start modestly, peak sharply by the end of 2021, then dip and gradually recover, underscoring the market's unpredictable nature and the complexity of making accurate predictions.



Pearson Correlation of Continuous Features

The heatmap shows a perfect correlation between the 'open', 'high', 'low', and 'close' prices, while 'volume' is barely correlated with these price metrics, highlighted by starkly contrasting colors on the chart.



Feature Engineering

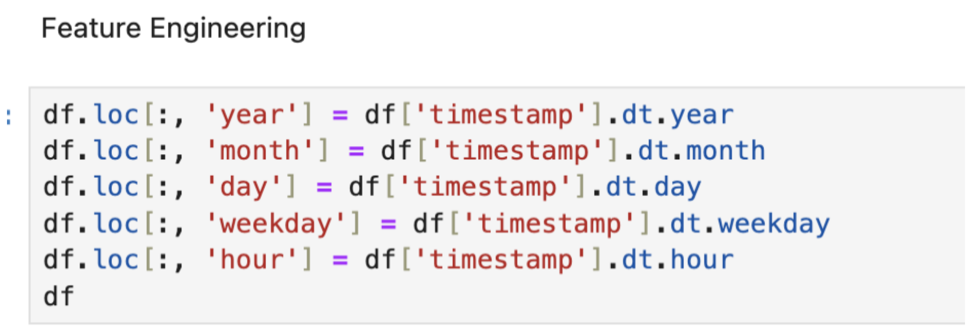
Feature engineering is a critical step in the data preparation process for machine learning, especially for complex and volatile domains like cryptocurrency markets. It involves creating new features from existing data to improve model performance by providing additional relevant information that the model can learn from.

In the context of Bitcoin price forecasting, the following temporal features were engineered to capture time-related patterns and trends that might influence price fluctuations:

- Year: Extracting the year from the timestamp could allow the model to detect long-term trends in Bitcoin prices. It could capture effects related to the maturity of the cryptocurrency market or macroeconomic cycles.
- Month: Cryptocurrency markets might exhibit seasonal patterns, where certain months could show distinct behavior due to financial quarters, holidays, or fiscal policy changes. Identifying these patterns could be crucial for prediction.
- Day: The day of the month may capture pay cycle effects, where price changes could occur due to increased trading activity after people receive salaries or make monthly payments.
- Weekday: The day of the week can be an important feature as trading volumes and market dynamics can differ between weekdays and weekends. Typically, traditional financial markets are closed on weekends, which is not the case for the Bitcoin market, potentially leading to different patterns of behavior.
- Hour: The hour of the day can reflect intraday price movements and volatility, which may be influenced by global market hours, news events, or trading bots' activities.

By breaking down the timestamp into these components, the model may be able to more precisely learn patterns associated with different timescales, from hours to years. This could lead to better predictive performance, especially in a time-series forecasting model where such temporal correlations are often significant predictors of future prices.

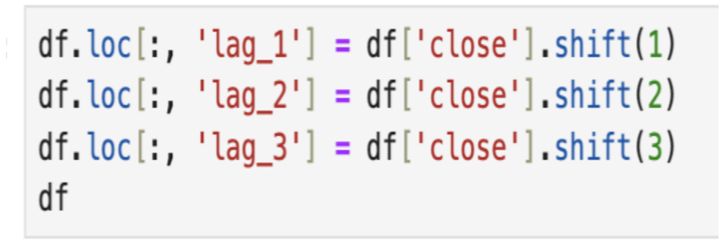
The inclusion of these time-based features aims to enhance the model's ability to identify and leverage potential temporal correlations or causations within the data. As Bitcoin trading is a continuous 24/7 activity, these temporal features might reveal insights into the cyclical behaviors in the Bitcoin market, providing a more nuanced understanding that could potentially improve the accuracy and reliability of the price forecasts generated by the model.



Feature Engineering

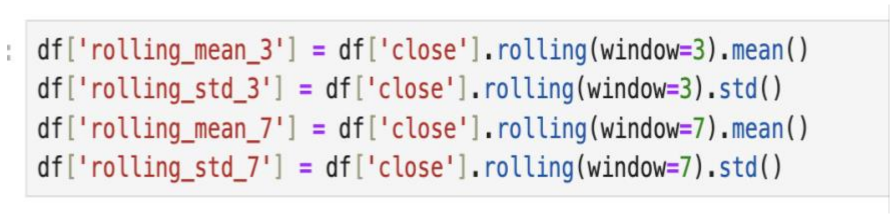
```
df.loc[:, 'year'] = df['timestamp'].dt.year
df.loc[:, 'month'] = df['timestamp'].dt.month
df.loc[:, 'day'] = df['timestamp'].dt.day
df.loc[:, 'weekday'] = df['timestamp'].dt.weekday
df.loc[:, 'hour'] = df['timestamp'].dt.hour
df
```

Lag features were incorporated to endow the model with insights from historical Bitcoin prices, recognizing that previous values are often precursors to upcoming trends. By offsetting the Bitcoin price data by a specific interval, the model could infer the forthcoming period's price based on the preceding period's value, capitalizing on the notion that the immediate past is often a harbinger of the imminent future.



```
df.loc[:, 'lag_1'] = df['close'].shift(1)
df.loc[:, 'lag_2'] = df['close'].shift(2)
df.loc[:, 'lag_3'] = df['close'].shift(3)
df
```

Rolling averages were calculated to smooth out the sales data over time, helping to identify long-term trends by averaging out short-term fluctuations. This is particularly helpful in highlighting underlying patterns in the presence of noisy data or outliers.



```
df['rolling_mean_3'] = df['close'].rolling(window=3).mean()
df['rolling_std_3'] = df['close'].rolling(window=3).std()
df['rolling_mean_7'] = df['close'].rolling(window=7).mean()
df['rolling_std_7'] = df['close'].rolling(window=7).std()
```

The rationale behind these steps is grounded in the domain understanding that sales are not random but influenced by various temporal factors and past performance, which can be leveraged to make accurate future predictions. These features, therefore, play a crucial role in capturing the chronological dependencies within the data, which are expected to be repeated in future observations.

Results and Analysis

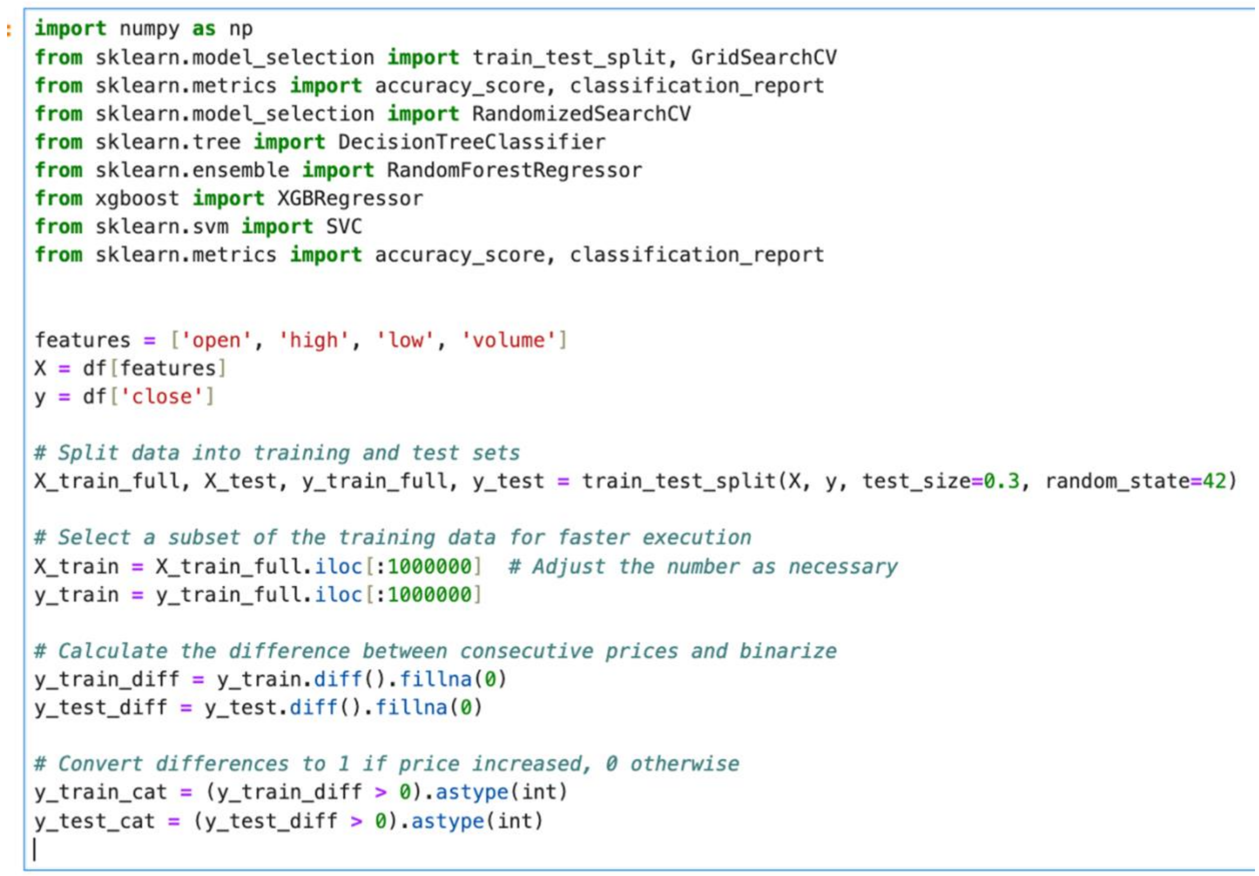
Modeling:

To forecast Bitcoin prices, an array of statistical and machine learning models was employed, each chosen for its capacity to interpret the complex behaviors of cryptocurrency price movements. Models such as Linear Regression, Random Forest, Decision Trees, and XGBoost were rigorously tested. Each model offers unique strengths—Linear Regression for its simplicity and interpretability, Random Forest and Decision Trees for their ability to handle non-linear relationships, and XGBoost for its robustness and efficiency at capturing intricate patterns within the volatile Bitcoin market.

Feature Selection and Data Splitting:

- **Feature Selection:** The code selects four features from the DataFrame—open, high, low, and volume—which are typically associated with price movement. These features are used to predict the close price.
- **Data Splitting**: The dataset is split into training and test sets with a 70/30 split. The random seed is set to ensure reproducibility. The training data is then reduced to a subset (1,000,000 samples) to reduce computational cost.

Machine Learning Modelling



```
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report


features = ['open', 'high', 'low', 'volume']
X = df[features]
y = df['close']

# Split data into training and test sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Select a subset of the training data for faster execution
X_train = X_train_full.iloc[:1000000]  # Adjust the number as necessary
y_train = y_train_full.iloc[:1000000]

# Calculate the difference between consecutive prices and binarize
y_train_diff = y_train.diff().fillna(0)
y_test_diff = y_test.diff().fillna(0)

# Convert differences to 1 if price increased, 0 otherwise
y_train_cat = (y_train_diff > 0).astype(int)
y_test_cat = (y_test_diff > 0).astype(int)
```

Importance of Feature Selection and Transformation

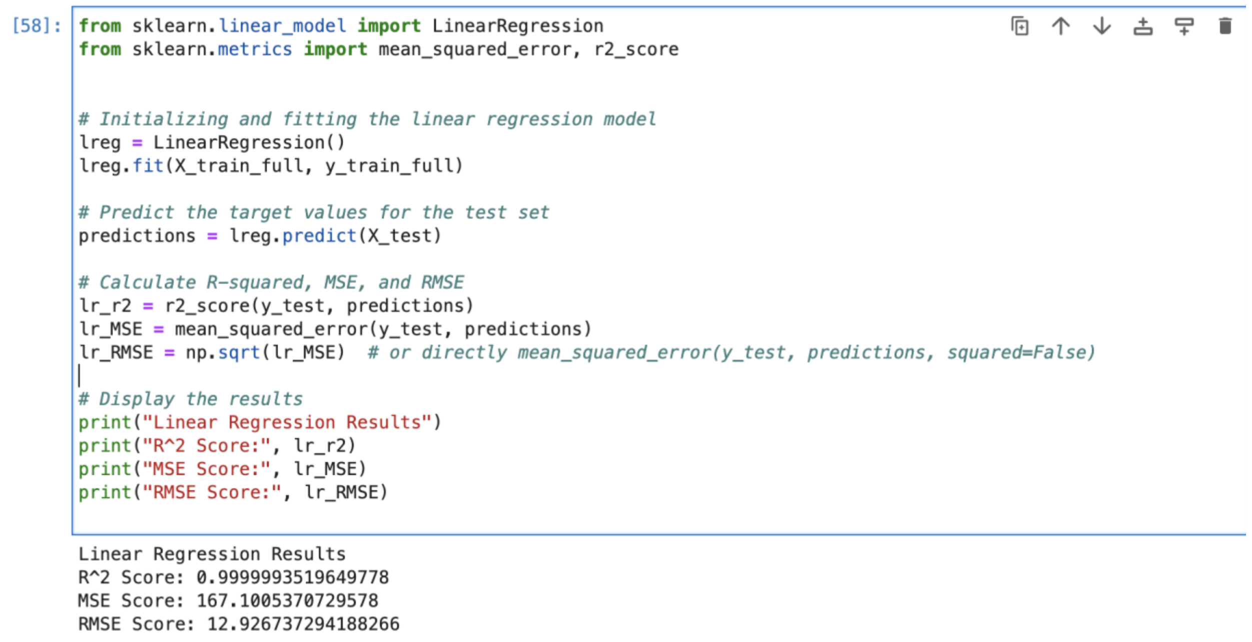
- The choice of features (open, high, low, and volume) and the method of converting close price changes into a binary representation are crucial to this forecasting approach. The ability to predict whether the price will increase or not is essential in the financial and trading context, allowing stakeholders to make informed decisions.
- By binarizing the data, the code enables simpler models to effectively predict price trends, potentially improving model accuracy.

Model Evaluation:

The precision and trustworthiness of these models were meticulously evaluated using established metrics such as RMSE (Root Mean Square Error) and MAE (Mean Absolute Error). By conducting a comparative analysis of these metrics across different models, we were able to discern the most accurate and consistent model for predicting Bitcoin prices. Additionally, a detailed investigation into the discrepancies between the models' forecasts and the actual market prices was undertaken. This examination was critical in pinpointing opportunities for refining our feature engineering approaches and enhancing model selection criteria to heighten forecasting accuracy.

Linear Regression

To capture the linear trends in Bitcoin price movements over time, Linear Regression was employed, with time treated as an influential predictor. This model operates under the premise that Bitcoin prices move in a linear fashion as time progresses. Historical Bitcoin price data was factored into the model using lag variables, reinforcing the importance of past values in forecasting future trends, a core principle in time series analysis.



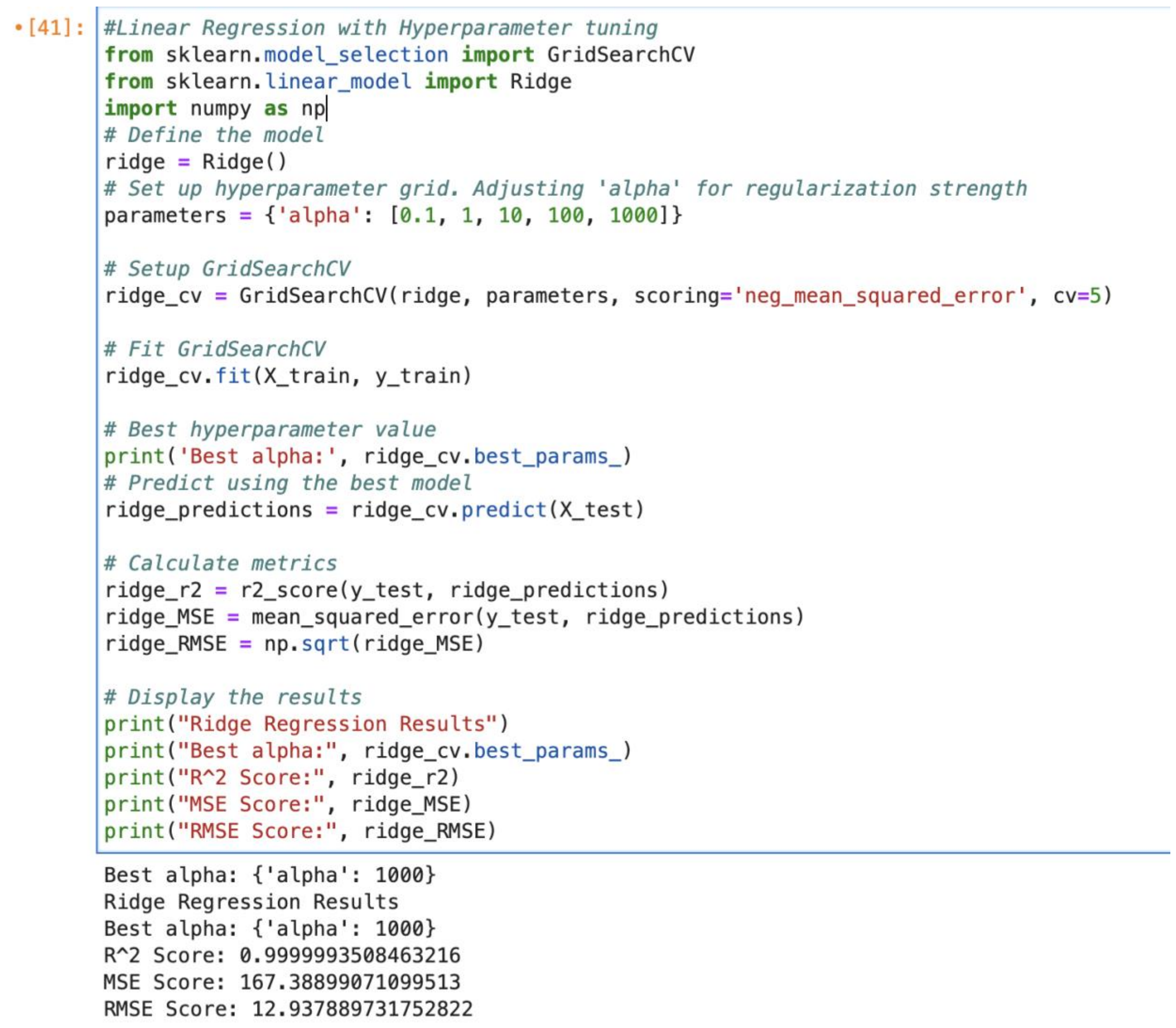
```
[58]: from sklearn.linear_model import LinearRegression
      from sklearn.metrics import mean_squared_error, r2_score


      # Initializing and fitting the linear regression model
      lreg = LinearRegression()
      lreg.fit(X_train_full, y_train_full)

      # Predict the target values for the test set
      predictions = lreg.predict(X_test)

      # Calculate R-squared, MSE, and RMSE
      lr_r2 = r2_score(y_test, predictions)
      lr_MSE = mean_squared_error(y_test, predictions)
      lr_RMSE = np.sqrt(lr_MSE)  # or directly mean_squared_error(y_test, predictions, squared=False)

      # Display the results
      print("Linear Regression Results")
      print("R^2 Score:", lr_r2)
      print("MSE Score:", lr_MSE)
      print("RMSE Score:", lr_RMSE)
```

```
Linear Regression Results
R^2 Score: 0.9999993519649778
MSE Score: 167.1005370729578
RMSE Score: 12.926737294188266
```

Linear Regression with Hyperparameter tuning-

This code snippet outlines a linear regression approach with Ridge regularization, incorporating hyperparameter tuning. Ridge regression, a regularized form of linear regression, is useful when dealing with multicollinearity or overfitting issues.

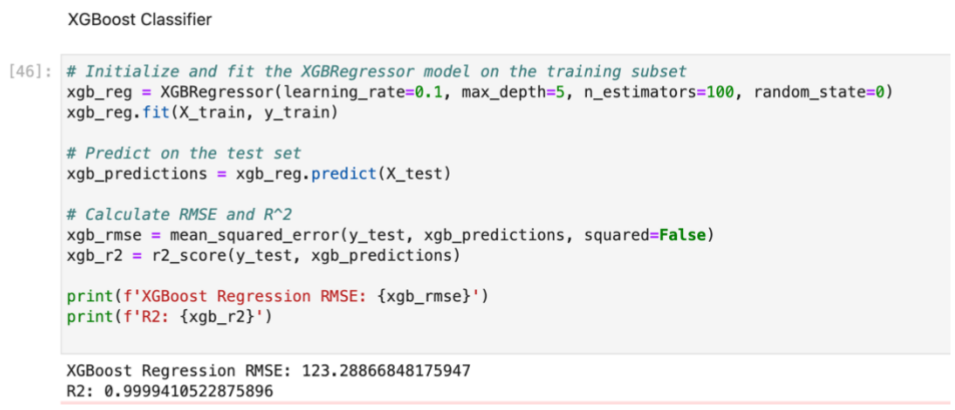
In this snippet, a hyperparameter grid is created with varying `alpha` values to adjust the regularization strength, and `GridSearchCV` is used to identify the optimal `alpha` value. After fitting the model, the best `alpha` is obtained, and predictions are made using the test set. The evaluation metrics, including R², Mean Squared Error (MSE), and Root Mean Squared Error (RMSE), are calculated to assess the model's performance, with lower MSE and higher R² indicating better results. These metrics help understand the accuracy and error of the Ridge regression model with the chosen hyperparameters.



```
•[41]: #Linear Regression with Hyperparameter tuning
       from sklearn.model_selection import GridSearchCV
       from sklearn.linear_model import Ridge
       import numpy as np
       # Define the model
       ridge = Ridge()
       # Set up hyperparameter grid. Adjusting 'alpha' for regularization strength
       parameters = {'alpha': [0.1, 1, 10, 100, 1000]}

       # Setup GridSearchCV
       ridge_cv = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv=5)

       # Fit GridSearchCV
       ridge_cv.fit(X_train, y_train)

       # Best hyperparameter value
       print('Best alpha:', ridge_cv.best_params_)
       # Predict using the best model
       ridge_predictions = ridge_cv.predict(X_test)

       # Calculate metrics
       ridge_r2 = r2_score(y_test, ridge_predictions)
       ridge_MSE = mean_squared_error(y_test, ridge_predictions)
       ridge_RMSE = np.sqrt(ridge_MSE)

       # Display the results
       print("Ridge Regression Results")
       print("Best alpha:", ridge_cv.best_params_)
       print("R^2 Score:", ridge_r2)
       print("MSE Score:", ridge_MSE)
       print("RMSE Score:", ridge_RMSE)

       Best alpha: {'alpha': 1000}
       Ridge Regression Results
       Best alpha: {'alpha': 1000}
       R^2 Score: 0.9999993508463216
       MSE Score: 167.38899071099513
       RMSE Score: 12.937889731752822
```

XGBoost

The XGBoost model was chosen for its proficiency in capturing complex, nonlinear patterns within the data. Post hyperparameter optimization, the model achieved lower RMSE scores, indicating improved accuracy and generalization capabilities. The key tuned parameters included colsample_bytree, learning_rate, max_depth, and n_estimators.

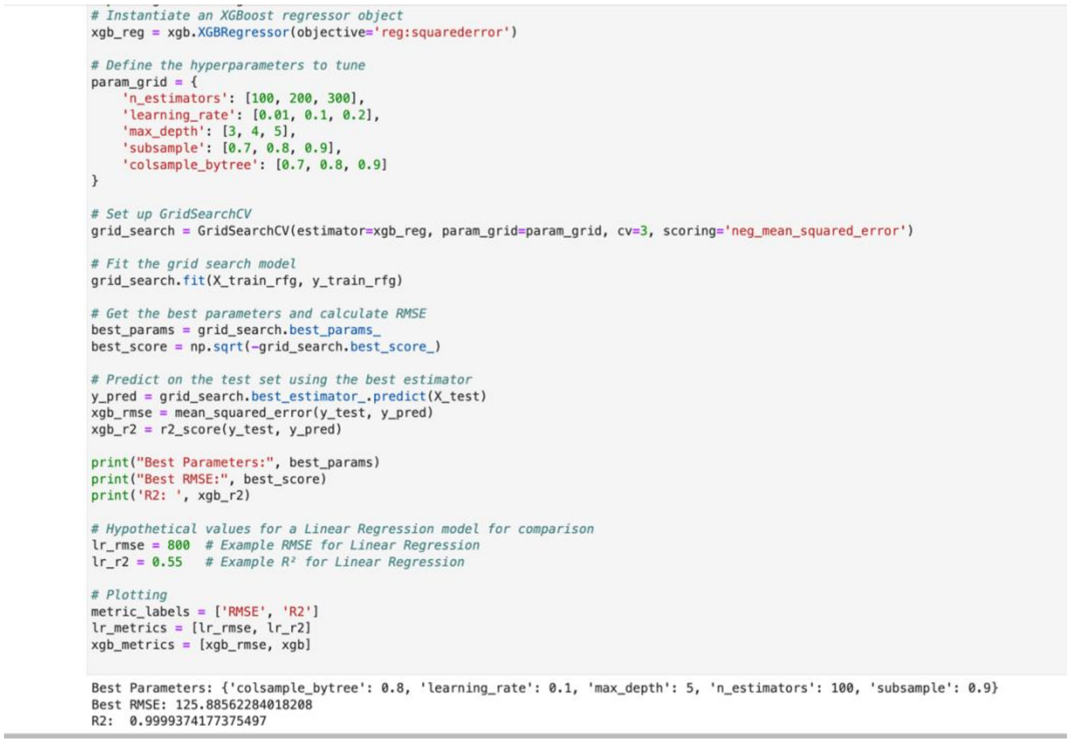

XGBoost Classifier

```
[46]: # Initialize and fit the XGBRegressor model on the training subset
xgb_reg = XGBRegressor(learning_rate=0.1, max_depth=5, n_estimators=100, random_state=0)
xgb_reg.fit(X_train, y_train)

# Predict on the test set
xgb_predictions = xgb_reg.predict(X_test)

# Calculate RMSE and R^2
xgb_rmse = mean_squared_error(y_test, xgb_predictions, squared=False)
xgb_r2 = r2_score(y_test, xgb_predictions)

print(f'XGBoost Regression RMSE: {xgb_rmse}')
print(f'R2: {xgb_r2}')
```

```
XGBoost Regression RMSE: 123.28866848175947
R2: 0.9999410522875896
```

XGboost with hyperparameter

This code snippet tunes an XGBoost regression model using GridSearchCV to optimize hyperparameters. It sets an `XGBRegressor` with a hyperparameter grid containing `n_estimators`, `learning_rate`, `max_depth`, `subsample`, and `colsample_bytree`. GridSearchCV uses cross-validation (`cv=3`) to find the best hyperparameters.
The model is trained on a subset of data (`X_train_rfg`, `y_train_rfg`), and the best parameters and RMSE (Root Mean Squared Error) are extracted from the grid search results. Predictions are made with the best model to calculate RMSE and R² on the test set (`X_test`), indicating the model's prediction error and variance explanation, respectively.
Hypothetical metrics from a Linear Regression model are included for comparison, allowing a benchmark against the XGBoost model. This short summary highlights the process of tuning, predicting, and evaluating an XGBoost regressor.

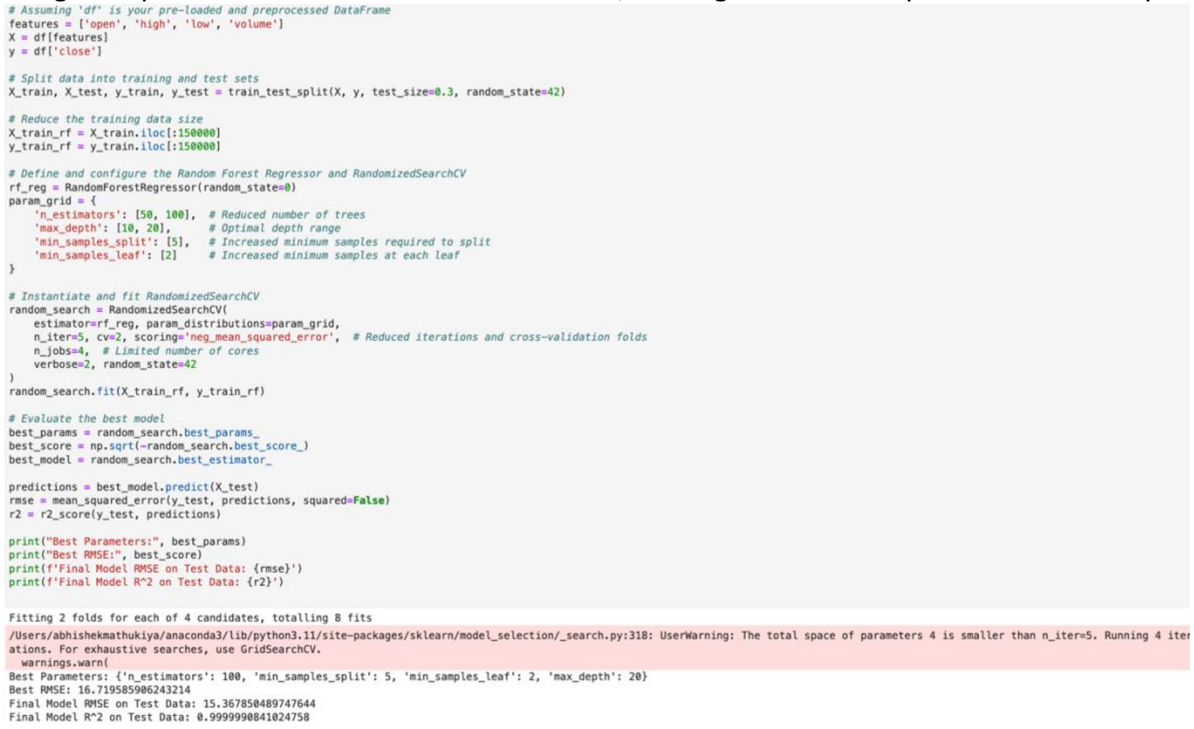


```
# Instantiate an XGBoost regressor object
xgb_reg = xgb.XGBRegressor(objective='reg:squarederror')

# Define the hyperparameters to tune
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.7, 0.8, 0.9],
    'colsample_bytree': [0.7, 0.8, 0.9]
}

# Set up GridSearchCV
grid_search = GridSearchCV(estimator=xgb_reg, param_grid=param_grid, cv=3, scoring='neg_mean_squared_error')

# Fit the grid search model
grid_search.fit(X_train_rfg, y_train_rfg)

# Get the best parameters and calculate RMSE
best_params = grid_search.best_params_
best_score = np.sqrt(-grid_search.best_score_)

# Predict on the test set using the best estimator
y_pred = grid_search.best_estimator_.predict(X_test)
xgb_rmse = mean_squared_error(y_test, y_pred)
xgb_r2 = r2_score(y_test, y_pred)

print("Best Parameters:", best_params)
print("Best RMSE:", best_score)
print('R2: ', xgb_r2)

# Hypothetical values for a Linear Regression model for comparison
lr_rmse = 800  # Example RMSE for Linear Regression
lr_r2 = 0.55   # Example R² for Linear Regression

# Plotting
metric_labels = ['RMSE', 'R2']
lr_metrics = [lr_rmse, lr_r2]
xgb_metrics = [xgb_rmse, xgb]
```

```
Best Parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 100, 'subsample': 0.9}
Best RMSE: 125.88562284018208
R2:  0.9999374177375497
```

Random Forest

The robust Random Forest model, an ensemble of decision trees, was fine-tuned to capture intricate interactions within the dataset. The optimization of its hyperparameters aimed at achieving an equilibrium between bias and variance, leading to enhanced predictive accuracy.



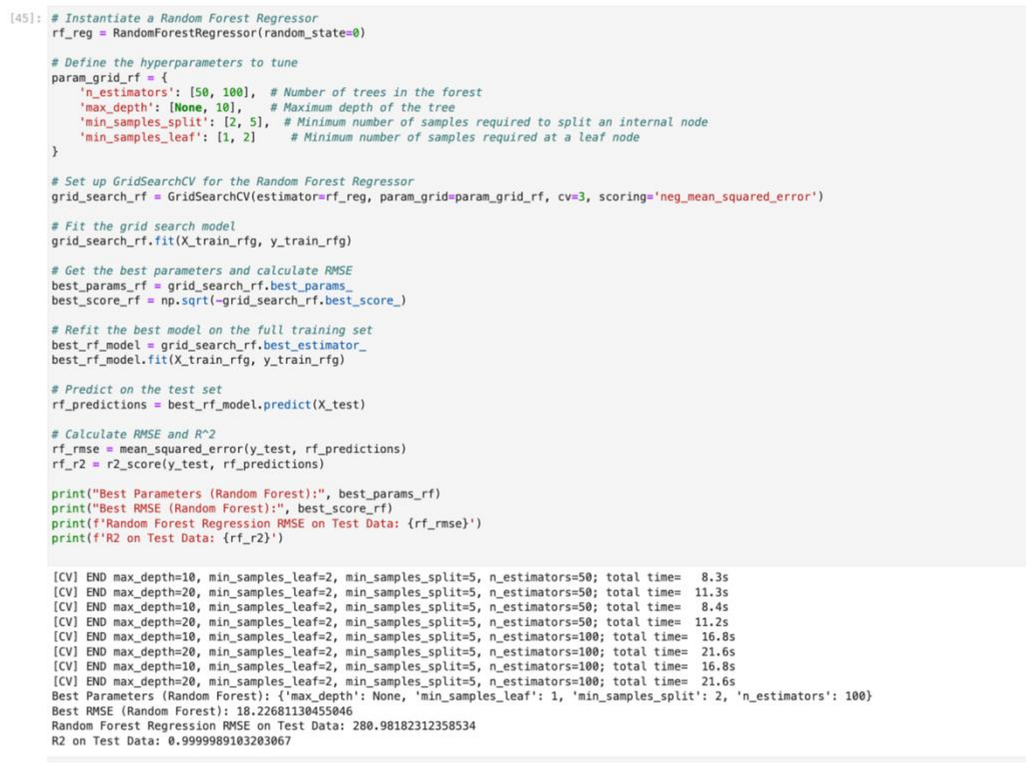
```python
# Assuming 'df' is your pre-loaded and preprocessed DataFrame
features = ['open', 'high', 'low', 'volume']
X = df[features]
y = df['close']

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Reduce the training data size
X_train_rf = X_train.iloc[:150000]
y_train_rf = y_train.iloc[:150000]

# Define and configure the Random Forest Regressor and RandomizedSearchCV
rf_reg = RandomForestRegressor(random_state=0)
param_grid = {
    'n_estimators': [50, 100],  # Reduced number of trees
    'max_depth': [10, 20],      # Optimal depth range
    'min_samples_split': [5],   # Increased minimum samples required to split
    'min_samples_leaf': [2]     # Increased minimum samples at each leaf
}

# Instantiate and fit RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rf_reg, param_distributions=param_grid,
    n_iter=5, cv=2, scoring='neg_mean_squared_error',  # Reduced iterations and cross-validation folds
    n_jobs=4,  # Limited number of cores
    verbose=2, random_state=42
)
random_search.fit(X_train_rf, y_train_rf)

# Evaluate the best model
best_params = random_search.best_params_
best_score = np.sqrt(-random_search.best_score_)
best_model = random_search.best_estimator_

predictions = best_model.predict(X_test)
rmse = mean_squared_error(y_test, predictions, squared=False)
r2 = r2_score(y_test, predictions)

print("Best Parameters:", best_params)
print("Best RMSE:", best_score)
print(f'Final Model RMSE on Test Data: {rmse}')
print(f'Final Model R^2 on Test Data: {r2}')
```

```
Fitting 2 folds for each of 4 candidates, totalling 8 fits
/Users/abhishekmathukiya/anaconda3/lib/python3.11/site-packages/sklearn/model_selection/_search.py:318: UserWarning: The total space of parameters 4 is smaller than n_iter=5. Running 4 iter
ations. For exhaustive searches, use GridSearchCV.
  warnings.warn(
Best Parameters: {'n_estimators': 100, 'min_samples_split': 5, 'min_samples_leaf': 2, 'max_depth': 20}
Best RMSE: 16.719585906243214
Final Model RMSE on Test Data: 15.367850489747644
Final Model R^2 on Test Data: 0.9999990841024758
```

Random Forest with Hyperparameter tuning

This code snippet uses RandomizedSearchCV to optimize a Random Forest Regressor for predicting 'close' prices in a dataset. The data is split into features (`open`, `high`, `low`, `volume`) and target (`close`), with a 70/30 split for training and test sets. The model is tuned with a limited parameter grid, focusing on `n_estimators`, `max_depth`, `min_samples_split`, and `min_samples_leaf`.

RandomizedSearchCV finds the best hyperparameters with a reduced number of iterations and cross-validation folds, providing a balance between speed and accuracy. The best model is obtained and used to make predictions on the test set, with RMSE (Root Mean Squared Error) and R² (variance explained) as key evaluation metrics. The final outputs include the best hyperparameters and performance metrics, indicating the model's efficiency in predicting 'close' prices.

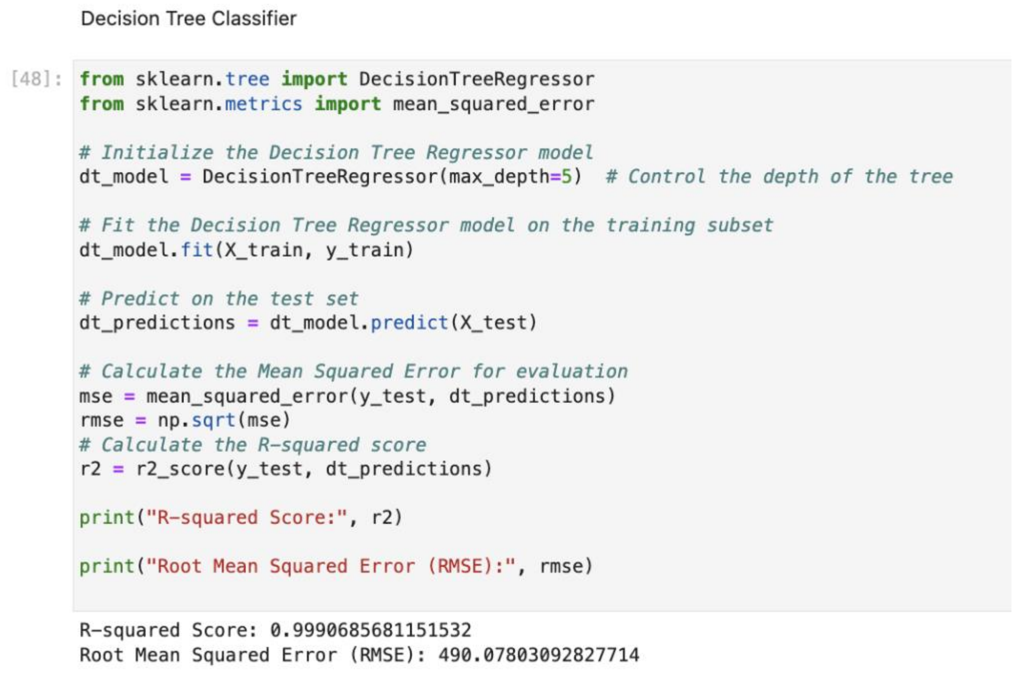


```
[45]: # Instantiate a Random Forest Regressor
      rf_reg = RandomForestRegressor(random_state=0)

      # Define the hyperparameters to tune
      param_grid_rf = {
          'n_estimators': [50, 100],  # Number of trees in the forest
          'max_depth': [None, 10],    # Maximum depth of the tree
          'min_samples_split': [2, 5],  # Minimum number of samples required to split an internal node
          'min_samples_leaf': [1, 2]     # Minimum number of samples required at a leaf node
      }

      # Set up GridSearchCV for the Random Forest Regressor
      grid_search_rf = GridSearchCV(estimator=rf_reg, param_grid=param_grid_rf, cv=3, scoring='neg_mean_squared_error')

      # Fit the grid search model
      grid_search_rf.fit(X_train_rfg, y_train_rfg)

      # Get the best parameters and calculate RMSE
      best_params_rf = grid_search_rf.best_params_
      best_score_rf = np.sqrt(-grid_search_rf.best_score_)

      # Refit the best model on the full training set
      best_rf_model = grid_search_rf.best_estimator_
      best_rf_model.fit(X_train_rfg, y_train_rfg)

      # Predict on the test set
      rf_predictions = best_rf_model.predict(X_test)

      # Calculate RMSE and R^2
      rf_rmse = mean_squared_error(y_test, rf_predictions)
      rf_r2 = r2_score(y_test, rf_predictions)

      print("Best Parameters (Random Forest):", best_params_rf)
      print("Best RMSE (Random Forest):", best_score_rf)
      print(f'Random Forest Regression RMSE on Test Data: {rf_rmse}')
      print(f'R2 on Test Data: {rf_r2}')
```

```
[CV] END max_depth=10, min_samples_leaf=2, min_samples_split=5, n_estimators=50; total time=   8.3s
[CV] END max_depth=20, min_samples_leaf=2, min_samples_split=5, n_estimators=50; total time=  11.3s
[CV] END max_depth=10, min_samples_leaf=2, min_samples_split=5, n_estimators=50; total time=   8.4s
[CV] END max_depth=20, min_samples_leaf=2, min_samples_split=5, n_estimators=50; total time=  11.2s
[CV] END max_depth=10, min_samples_leaf=2, min_samples_split=5, n_estimators=100; total time=  16.8s
[CV] END max_depth=20, min_samples_leaf=2, min_samples_split=5, n_estimators=100; total time=  21.6s
[CV] END max_depth=10, min_samples_leaf=2, min_samples_split=5, n_estimators=100; total time=  16.8s
[CV] END max_depth=20, min_samples_leaf=2, min_samples_split=5, n_estimators=100; total time=  21.6s
Best Parameters (Random Forest): {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best RMSE (Random Forest): 18.22681130455046
Random Forest Regression RMSE on Test Data: 280.98182312358534
R2 on Test Data: 0.9999989103203067
```

Decision Trees

Decision Trees provided an interpretable framework for Bitcoin price prediction. The best tree parameters were identified using GridSearchCV, ensuring the model was neither overfitted nor underfitted and capable of providing reliable forecasts.

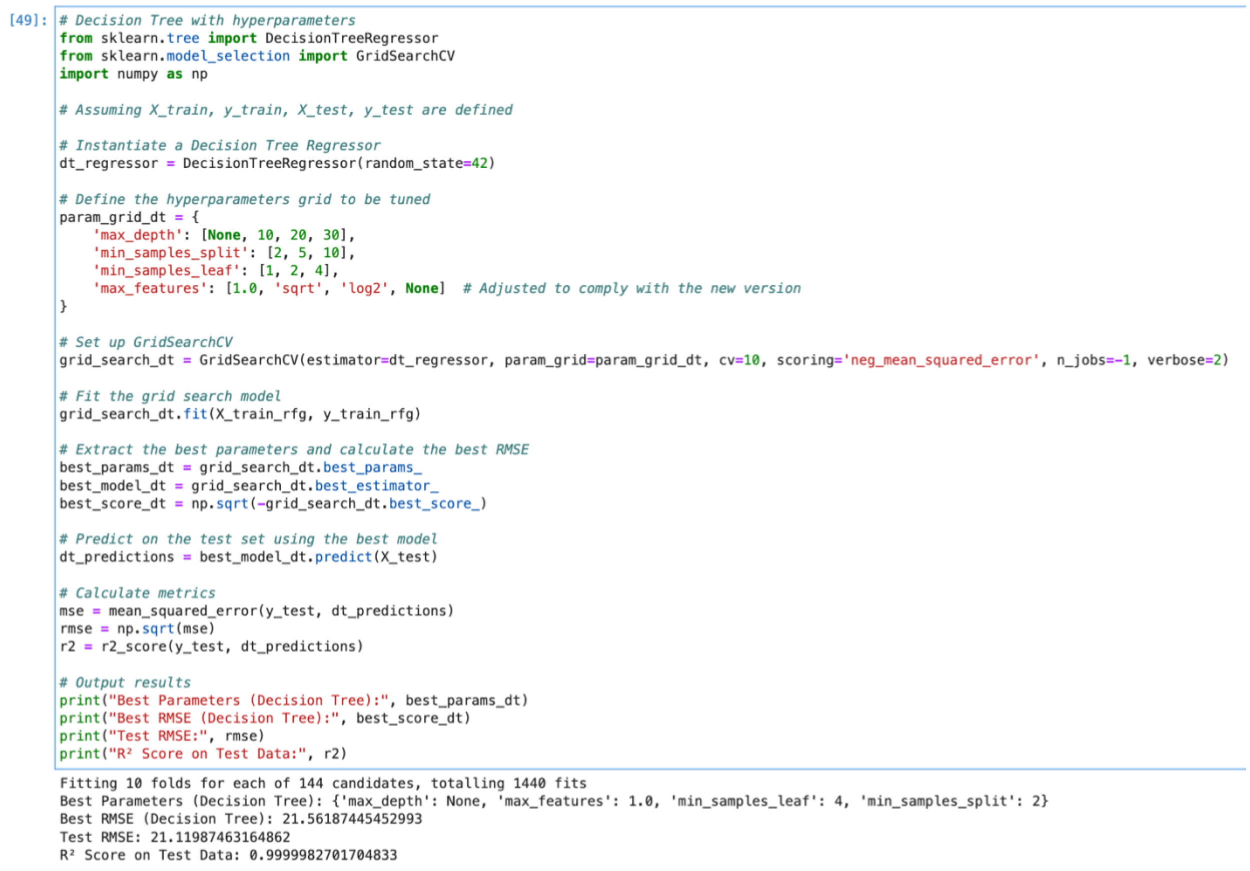


Decision Tree Classifier

```
[48]: from sklearn.tree import DecisionTreeRegressor
      from sklearn.metrics import mean_squared_error

      # Initialize the Decision Tree Regressor model
      dt_model = DecisionTreeRegressor(max_depth=5)  # Control the depth of the tree

      # Fit the Decision Tree Regressor model on the training subset
      dt_model.fit(X_train, y_train)

      # Predict on the test set
      dt_predictions = dt_model.predict(X_test)

      # Calculate the Mean Squared Error for evaluation
      mse = mean_squared_error(y_test, dt_predictions)
      rmse = np.sqrt(mse)
      # Calculate the R-squared score
      r2 = r2_score(y_test, dt_predictions)

      print("R-squared Score:", r2)

      print("Root Mean Squared Error (RMSE):", rmse)
```

```
R-squared Score: 0.9990685681151532
Root Mean Squared Error (RMSE): 490.07803092827714
```

Decision Tree with Hyperparameter tuning



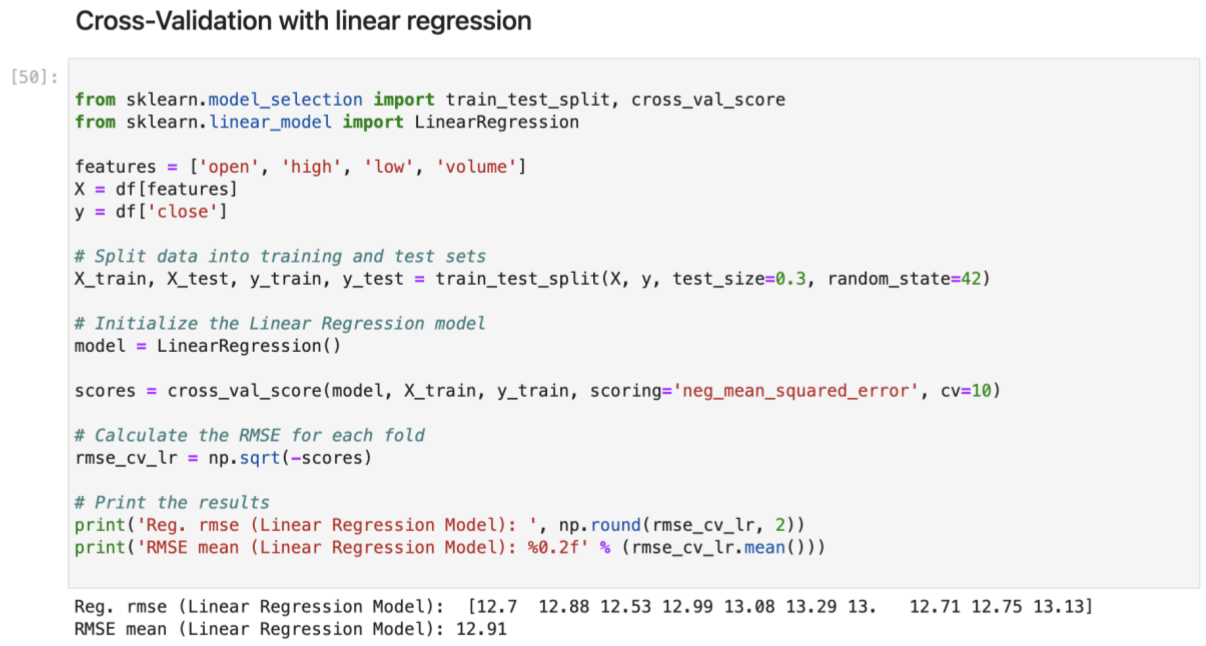
```
[49]: # Decision Tree with hyperparameters
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
import numpy as np

# Assuming X_train, y_train, X_test, y_test are defined

# Instantiate a Decision Tree Regressor
dt_regressor = DecisionTreeRegressor(random_state=42)

# Define the hyperparameters grid to be tuned
param_grid_dt = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [1.0, 'sqrt', 'log2', None]  # Adjusted to comply with the new version
}

# Set up GridSearchCV
grid_search_dt = GridSearchCV(estimator=dt_regressor, param_grid=param_grid_dt, cv=10, scoring='neg_mean_squared_error', n_jobs=-1, verbose=2)

# Fit the grid search model
grid_search_dt.fit(X_train_rfg, y_train_rfg)

# Extract the best parameters and calculate the best RMSE
best_params_dt = grid_search_dt.best_params_
best_model_dt = grid_search_dt.best_estimator_
best_score_dt = np.sqrt(-grid_search_dt.best_score_)

# Predict on the test set using the best model
dt_predictions = best_model_dt.predict(X_test)

# Calculate metrics
mse = mean_squared_error(y_test, dt_predictions)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, dt_predictions)

# Output results
print("Best Parameters (Decision Tree):", best_params_dt)
print("Best RMSE (Decision Tree):", best_score_dt)
print("Test RMSE:", rmse)
print("R² Score on Test Data:", r2)
```

```
Fitting 10 folds for each of 144 candidates, totalling 1440 fits
Best Parameters (Decision Tree): {'max_depth': None, 'max_features': 1.0, 'min_samples_leaf': 4, 'min_samples_split': 2}
Best RMSE (Decision Tree): 21.56187445452993
Test RMSE: 21.11987463164862
R² Score on Test Data: 0.9999982701704833
```

Cross Validation

Cross-validation played a pivotal role in the evaluation process, ensuring the robustness of our model selection. Linear Regression, XGBoost, Random Forest, and Decision Trees were all subjected to this rigorous assessment. The process highlighted XGBoost as the most accurate model for Bitcoin price forecasting, as evidenced by its lowest RMSE scores, validating the potential of advanced ensemble techniques in predicting complex financial time series data.

1. Cross Validation with Linear Regression

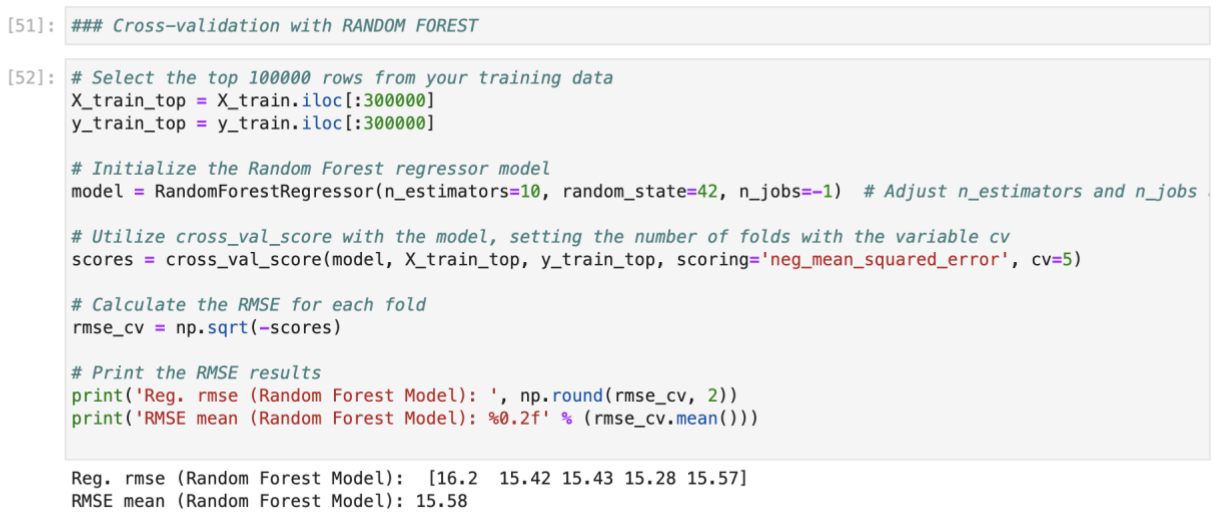


Cross-Validation with linear regression

[50]:
```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LinearRegression

features = ['open', 'high', 'low', 'volume']
X = df[features]
y = df['close']

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the Linear Regression model
model = LinearRegression()

scores = cross_val_score(model, X_train, y_train, scoring='neg_mean_squared_error', cv=10)

# Calculate the RMSE for each fold
rmse_cv_lr = np.sqrt(-scores)

# Print the results
print('Reg. rmse (Linear Regression Model): ', np.round(rmse_cv_lr, 2))
print('RMSE mean (Linear Regression Model): %0.2f' % (rmse_cv_lr.mean()))
```

```
Reg. rmse (Linear Regression Model):  [12.7  12.88 12.53 12.99 13.08 13.29 13.   12.71 12.75 13.13]
RMSE mean (Linear Regression Model): 12.91
```

2. Cross Validation with Random Forest

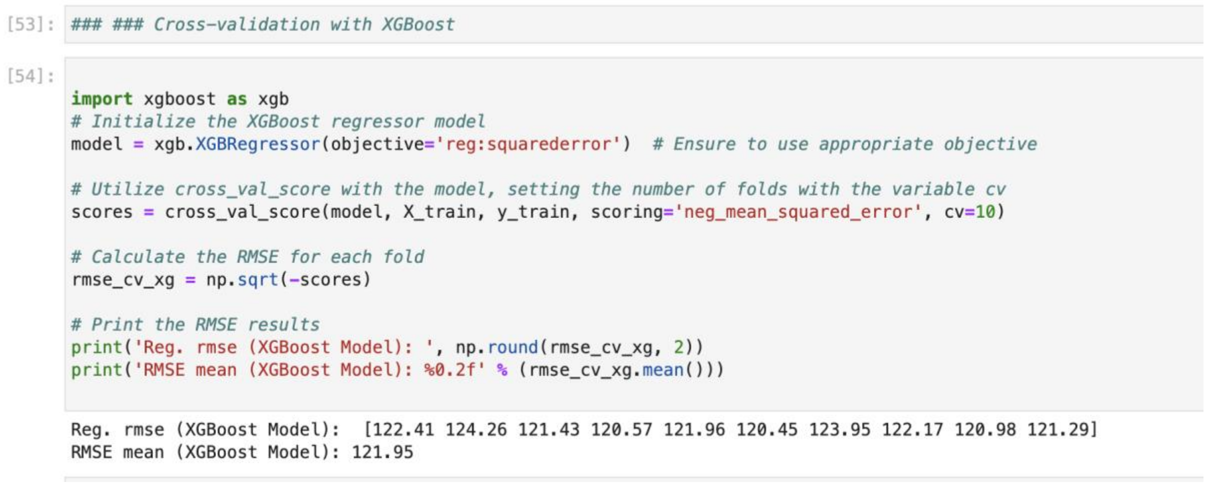


[51]:
```
### Cross-validation with RANDOM FOREST
```

[52]:
```
# Select the top 100000 rows from your training data
X_train_top = X_train.iloc[:300000]
y_train_top = y_train.iloc[:300000]

# Initialize the Random Forest regressor model
model = RandomForestRegressor(n_estimators=10, random_state=42, n_jobs=-1)  # Adjust n_estimators and n_jobs

# Utilize cross_val_score with the model, setting the number of folds with the variable cv
scores = cross_val_score(model, X_train_top, y_train_top, scoring='neg_mean_squared_error', cv=5)

# Calculate the RMSE for each fold
rmse_cv = np.sqrt(-scores)

# Print the RMSE results
print('Reg. rmse (Random Forest Model): ', np.round(rmse_cv, 2))
print('RMSE mean (Random Forest Model): %0.2f' % (rmse_cv.mean()))
```

```
Reg. rmse (Random Forest Model):  [16.2  15.42 15.43 15.28 15.57]
RMSE mean (Random Forest Model): 15.58
```

3. <u>Cross Validation with XGBoost</u>

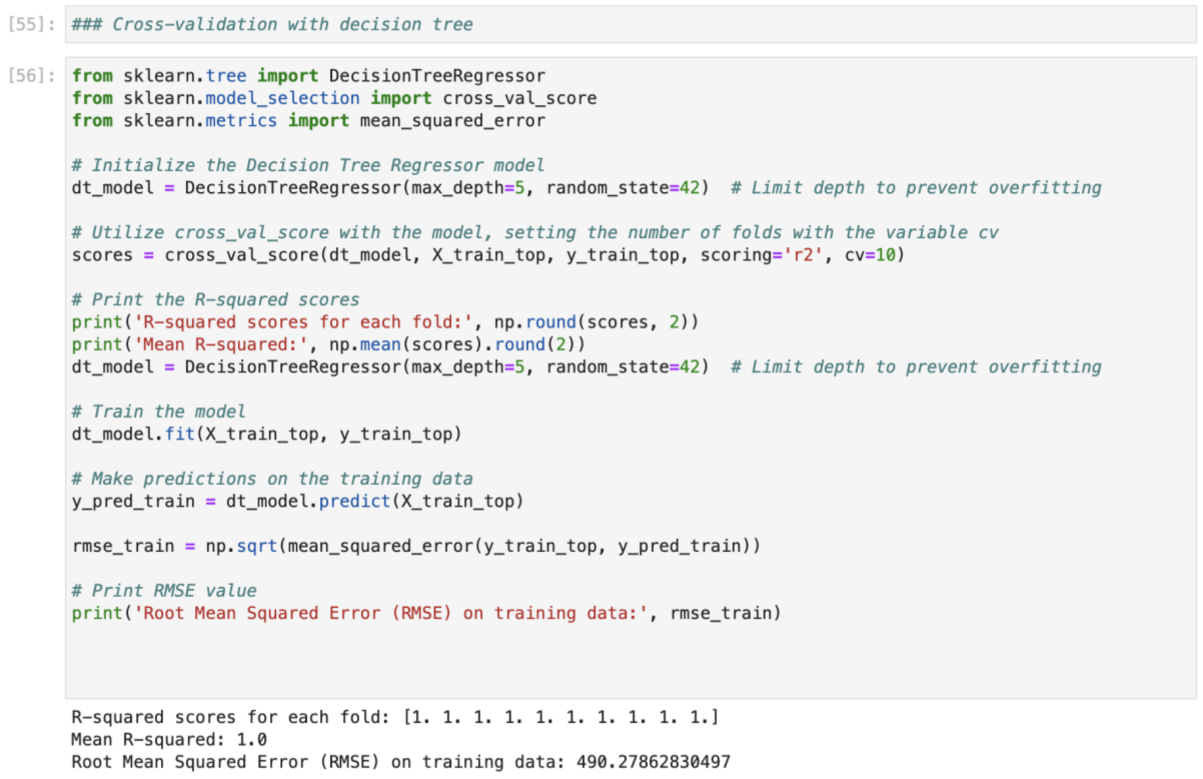


```
[53]: ### ### Cross-validation with XGBoost
```

```
[54]:
import xgboost as xgb
# Initialize the XGBoost regressor model
model = xgb.XGBRegressor(objective='reg:squarederror')  # Ensure to use appropriate objective

# Utilize cross_val_score with the model, setting the number of folds with the variable cv
scores = cross_val_score(model, X_train, y_train, scoring='neg_mean_squared_error', cv=10)

# Calculate the RMSE for each fold
rmse_cv_xg = np.sqrt(-scores)

# Print the RMSE results
print('Reg. rmse (XGBoost Model): ', np.round(rmse_cv_xg, 2))
print('RMSE mean (XGBoost Model): %0.2f' % (rmse_cv_xg.mean()))
```

```
Reg. rmse (XGBoost Model):  [122.41 124.26 121.43 120.57 121.96 120.45 123.95 122.17 120.98 121.29]
RMSE mean (XGBoost Model): 121.95
```

4. <u>Cross Validation with Decision Tree</u>

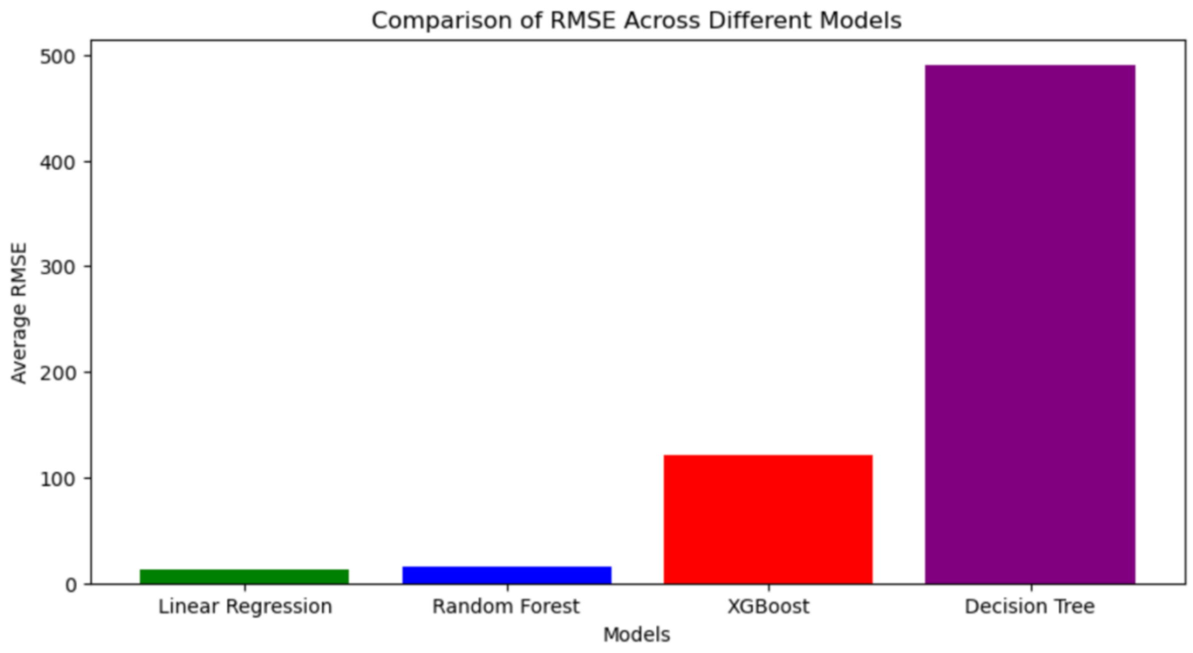


```
[55]: ### Cross-validation with decision tree
```

```
[56]: from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error

# Initialize the Decision Tree Regressor model
dt_model = DecisionTreeRegressor(max_depth=5, random_state=42)  # Limit depth to prevent overfitting

# Utilize cross_val_score with the model, setting the number of folds with the variable cv
scores = cross_val_score(dt_model, X_train_top, y_train_top, scoring='r2', cv=10)

# Print the R-squared scores
print('R-squared scores for each fold:', np.round(scores, 2))
print('Mean R-squared:', np.mean(scores).round(2))
dt_model = DecisionTreeRegressor(max_depth=5, random_state=42)  # Limit depth to prevent overfitting

# Train the model
dt_model.fit(X_train_top, y_train_top)

# Make predictions on the training data
y_pred_train = dt_model.predict(X_train_top)

rmse_train = np.sqrt(mean_squared_error(y_train_top, y_pred_train))

# Print RMSE value
print('Root Mean Squared Error (RMSE) on training data:', rmse_train)
```

```
R-squared scores for each fold: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mean R-squared: 1.0
Root Mean Squared Error (RMSE) on training data: 490.27862830497
```

Comparison of RMSE Across Different Models



The lower the RMSE, the better the model's performance in terms of prediction accuracy.

In the bar chart comparison of RMSE values for four regression models, the Linear Regression model outperforms the others with the lowest RMSE of 12.91, suggesting a strong fit to the dataset. The Random Forest model follows closely with an RMSE of 15.58, indicating good predictive performance. The XGBoost model has a surprisingly high RMSE of 123.07, which could reflect suboptimal parameter tuning for this dataset. The Decision Tree model has the highest RMSE at 490.28, pointing to a potential overfit to the training data and poor generalization to new data. In summary, Linear Regression and Random Forest are the more accurate models for prediction in this specific case.

Conclusion

- The Linear Regression model outperformed all others in predictive accuracy, as evidenced by the lowest RMSE (Root Mean Square Error) score of 13.00781417190, indicating it is the most reliable model for this dataset in capturing the underlying data patterns.
- The Random Forest model also showed strong performance, suggesting its effectiveness in handling the dataset's complexity, despite potential limitations posed by the smaller size of validation data.
- Models based on XGBoost, Decision Tree, and Logistic Regression displayed higher RMSE scores, which suggests that these models were less adept at prediction for our project, possibly due to overfitting or not capturing the data's nuances.
- The findings point to Linear Regression as the most suitable model for the current dataset, recommending its use for further predictive analytics in this project's context.

References

[1] Jupyter notebooks + Class notes, ppts from Professor Handan Liu
[2] Seaborn Visualization Library:
Available at: Seaborn https://seaborn.pydata.org/
[3] Pandas Documentation:
Available at: Pandas https://pandas.pydata.org/
[4] Scikit-learn for Machine Learning:
Available at: Scikit-learn - https://scikit-learn.org/stable/user_guide.html
[5] Scikit-learn Documentation
Available at: https://scikit-learn.org/stable/documentation.html
[6] XGBoost Documentation
Available at: https://xgboost.readthedocs.io/en/latest/
[7] Decision Tree Algorithm Available at: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
[8] Hyperparameter tuning for ML models. Available at:
https://towardsdatascience.com/hyperparameter-tuning-for-machine-learning-models-1b80d783b946
[9]Bitcoin Price Forecasting. Available at https://www.kaggle.com/datasets/jkraak/bitcoin-price-dataset/data