PART A – Regression modeling for business decision making

Link to supporting materials:
https://github.com/AIgulKZNZ/GD708_Assessment2.git

Task1. Data Preprocessing

For this assignment, I utilized the "Software Engineers Salaries" dataset, sourced from kaggle.com . To carry out the analysis, I decided to use Jupyter Notebook as my development environment. To complete this task, I've uploaded the dataset for the assignment and launched Anaconda Navigator on my local machine. Next, I created a new Jupyter Notebook.
To load a dataset, first, I need to import the pandas library. Pandas library is a powerful data manipulation and analysis tool for Python, providing data structures and functions needed to work with structured data. It allows us to efficiently load, manipulate, and analyze data in various formats, including CSV, Excel, and SQL databases. I used pandas' read_csv function to load the data from a CSV file into a DataFrame.

```
[2]: import pandas as pd
```

```
[30]: df=pd.read_csv("C:\\Users\\aseks\\Downloads\\Software Engineer Salaries - Software Engineer Salaries.csv")
```

```
[32]: df.head(5)
```

[32]:

	Company	Company Score	Job Title	Location	Date	Salary
0	ViewSoft	4.8	Software Engineer	Manassas, VA	8d	$68K–94K (Glassdoor est.)
1	Workiva	4.3	Software Support Engineer	Remote	2d	$61K–104K (Employer est.)
2	Garmin International, Inc.	3.9	C# Software Engineer	Cary, NC	2d	$95K–118K (Glassdoor est.)
3	Snapchat	3.5	Software Engineer, Fullstack, 1+ Years of Expe...	Los Angeles, CA	2d	$97K–145K (Employer est.)
4	Vitesco Technologies Group AG	3.1	Software Engineer	Seguin, TX	2d	$85K–108K (Glassdoor est.)

```
[34]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 870 entries, 0 to 869
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Company        866 non-null    object
 1   Company Score  788 non-null    float64
 2   Job Title      865 non-null    object
 3   Location       852 non-null    object
 4   Date           865 non-null    object
 5   Salary         759 non-null    object
dtypes: float64(1), object(5)
memory usage: 40.9+ KB
```

```
[36]: df.describe()
```

[36]:

	Company Score
count	788.000000
mean	3.895051
std	0.525235
min	1.000000
25%	3.600000
50%	3.900000
75%	4.200000
max	5.000000

a.

```
missing_values = df.isnull().sum()
print(missing_values)
```

```
Company           4
Company Score    82
Job Title         5
Location         18
Date              5
Salary          111
dtype: int64
```

Filling Missing Values: For these numeric columns, I filled any missing values with the mean of the respective column. This is a common imputation method that allows us to maintain the integrity of the dataset by filling gaps with a reasonable estimate based on the existing data.

```
numeric_columns = df.select_dtypes(include=['float64', 'int64']).columns
df_cleaned = df.copy()
df_cleaned[numeric_columns] = df_cleaned[numeric_columns].fillna(df_cleaned[numeric_columns].mean())
```

Non-numeric columns, such as categorical data (e.g., labels, categories), require a different approach: I filtered out the non-numeric columns to handle them separately.

```
non_numeric_columns = df.select_dtypes(exclude=['float64', 'int64']).columns
for col in non_numeric_columns:
    df_cleaned[col] = df_cleaned[col].fillna(df_cleaned[col].mode()[0])
```

For these columns, the most common approach is to fill missing values with the mode, which is the most frequent value in that column. This preserves the categorical distribution and avoids introducing bias.

Next, I removed outliers using the Interquartile Range (IQR) method, which identifies data points that lie significantly outside the typical range. Outliers can distort analysis results, so removing them helps maintain the accuracy of statistical measures. By focusing on the middle 50% of the data, the IQR method effectively eliminates these extreme values, ensuring a more reliable and representative dataset for further analysis.

```
def remove_outliers_iqr(df):
    numeric_df = df.select_dtypes(include=['float64', 'int64'])
    Q1 = numeric_df.quantile(0.25)
    Q3 = numeric_df.quantile(0.75)
    IQR = Q3 - Q1
    df_outliers_removed = df[~((numeric_df < (Q1 - 1.5 * IQR)) | (numeric_df > (Q3 + 1.5 * IQR))).any(axis=1)]
    return df_outliers_removed

df_no_outliers = remove_outliers_iqr(df_cleaned)
```

c.

For feature scaling, I utilized the Min-Max Scaler to normalize the numeric columns, ensuring that all values fall within the range of 0 to 1. This step is crucial to prevent features with larger ranges from disproportionately influencing the model's performance, thereby ensuring a balanced and accurate analysis.

```
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
```

```
scaler = MinMaxScaler()
df_no_outliers.loc[:, numeric_columns] = scaler.fit_transform(df_no_outliers.loc[:, numeric_columns])
```

For categorical variable encoding, I applied the Label Encoder to convert the Company, Job Title, Location, and Salary columns into numeric values. This transformation is essential as it allows categorical data to be effectively utilized in machine learning models by converting the categories into integer labels, ensuring compatibility with algorithms that require numeric input.

```
label_encoder = LabelEncoder()
df_no_outliers.loc[:, 'Company'] = label_encoder.fit_transform(df_no_outliers.loc[:, 'Company'])
df_no_outliers.loc[:, 'Job Title'] = label_encoder.fit_transform(df_no_outliers.loc[:, 'Job Title'])
df_no_outliers.loc[:, 'Location'] = label_encoder.fit_transform(df_no_outliers.loc[:, 'Location'])
df_no_outliers.loc[:, 'Salary'] = label_encoder.fit_transform(df_no_outliers.loc[:, 'Salary'])
```

To evaluate model performance, I split the dataset into training and testing sets, with 80% of the data used for training and 20% reserved for testing. This approach ensures that the model is trained on a substantial portion of the data while retaining a separate dataset for validating its accuracy on unseen data, thus providing a reliable assessment of the model's generalization capability.

```
X = df_no_outliers.drop(columns=['Salary'])
y = df_no_outliers['Salary']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print(df_no_outliers.head())
```

```
   Company  Company Score Job Title Location Date Salary
0      571       1.000000       294      174   8d    358
1      598       0.736842       466      243   2d    320
2      230       0.526316        40       55   2d    598
3      481       0.315789       418      168   2d    618
4      575       0.105263       294      267   2d    500
```

Task-2 Model Building with Hyper-Parameter Tuning

For this assessment, I initially ensured that the Date column was correctly treated as a string by using the astype(str) method. This conversion was necessary because the Date column contained mixed data types, and I needed to perform string operations on it. After converting the column to a string, I extracted the numeric part of the Date using a regular expression with the .str.extract(r'(\d+)') method. This allowed me to isolate the numerical values representing the number of days.

Next, I converted these extracted numeric values to a float data type using pd.to_numeric, ensuring that any non-numeric values were safely handled by setting them to NaN with the errors='coerce' parameter. I then filled any missing values (NaNs) with 0 using the fillna(0) method and converted the entire Date column to a float type using astype(float) to ensure consistency and avoid future errors during model training.

After cleaning the data, I split the dataset into training and testing sets using the train_test_split method, with 80% of the data allocated for training and 20% for testing. I then proceeded to build a Ridge Regression model and performed hyperparameter tuning using Grid Search. I specified a grid of alpha values and different solvers to find the best combination that minimized the mean squared error. Finally, I evaluated the model on the test set to assess its performance, reporting the best parameters, cross-validation score, and test mean squared error.

```
df_no_outliers.loc[:, 'Date'] = df_no_outliers.loc[:, 'Date'].astype(str)

df_no_outliers.loc[:, 'Date'] = df_no_outliers.loc[:, 'Date'].str.extract(r'(\d+)')
df_no_outliers.loc[:, 'Date'] = pd.to_numeric(df_no_outliers.loc[:, 'Date'], errors='coerce')
df_no_outliers.loc[:, 'Date'] = df_no_outliers.loc[:, 'Date'].fillna(0).astype(float)

X = df_no_outliers.drop(columns=['Salary'])
y = df_no_outliers['Salary']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error

ridge_model = Ridge()

# Hyperparameter Grid
param_grid = {
    'alpha': [0.01, 0.1, 1, 10, 100],
    'solver': ['auto', 'svd', 'cholesky', 'lsqr', 'sag', 'saga']
}

grid_search = GridSearchCV(estimator=ridge_model, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_score = grid_search.best_score_
y_pred = grid_search.predict(X_test)
test_mse = mean_squared_error(y_test, y_pred)

print("Best Hyperparameters:", best_params)
print("Best Cross-Validation Score:", best_score)
print("Test MSE:", test_mse)
```

Results:

```
Best Hyperparameters: {'alpha': 100, 'solver': 'lsqr'}
Best Cross-Validation Score: -33539.76000228875
Test MSE: 33482.23268286485
```

Based on the hyperparameter tuning results, I found that the best hyperparameters for the Ridge Regression model were an alpha value of 100 and the lsqr solver. The alpha parameter controls the regularization strength, and a value of 100 suggests that a higher level of regularization was necessary to optimize model performance. The lsqr solver, which is efficient for large datasets, was identified as the best choice for solving the regression problem.

The best cross-validation score achieved during the grid search was approximately -33,539.76. This score represents the mean squared error (MSE) during the cross-validation process, with a negative value indicating the loss function's value (since higher negative values in sklearn's scoring correspond to worse performance).

When I evaluated the model on the test set, the mean squared error (MSE) was approximately 33,482.23. This test MSE is close to the cross-validation score, indicating that the model has generalized well to the unseen data, and the regularization strategy was effective in preventing overfitting. These results

demonstrate that the model is performing consistently across both the training and testing datasets.

Task-3 Model Evaluation and Selection [22 Marks]

To evaluate the performance of the Ridge Regression model, I calculated the following metrics:

1. **Mean Absolute Error (MAE)**:

MAE measures the average magnitude of the errors in a set of predictions, without considering their direction (i.e., whether they are positive or negative).

2. **R-squared (R^2)**:

R^2 is a statistical measure that represents the proportion of the variance for the dependent variable that's explained by the independent variables in the model.

```
from sklearn.metrics import mean_absolute_error, r2_score
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Absolute Error (MAE):", mae)
print("R-squared (R²):", r2)
```

```
Mean Absolute Error (MAE): 162.63608205844002
R-squared (R²): 0.008310433702036213
```

Mean Absolute Error (MAE): 162.64

The MAE measures the average magnitude of errors in the predictions. In this case, an MAE of 162.64 indicates that, on average, the model's predictions are off by around 162.64 units from the actual salary values. This suggests that while the model is making predictions that are somewhat close to the actual values, the errors are still significant, meaning the model could be further improved.

R-squared (R^2): 0.0083

The R^2 value of 0.0083 indicates that the model explains only about 0.83% of the variance in the salary data. This is a very low R^2, meaning that the model is not capturing much of the relationship between the predictors and the target

variable. The model's ability to explain the variation in salaries is therefore quite limited, suggesting that either the features used are not very predictive of salary or that a different model might be better suited for this problem.

Interpretation:

- **MAE Interpretation**: The relatively high MAE indicates that the model's predictions are not very accurate, with an average error of 162.64 units.
- **R-squared Interpretation**: The very low R^2 value suggests that the model does not explain much of the variability in the salary data, indicating that the model might not be well-suited to this specific prediction task.

I performed 5-fold cross-validation to assess the model's generalization performance:

```
cv_scores = cross_val_score(ridge_model, X_train, y_train, cv=5, scoring='neg_mean_absolute_error')
mean_cv_score = -cv_scores.mean()
std_cv_score = cv_scores.std()

print("Mean CV Score (MAE):", mean_cv_score)
print("Standard Deviation of CV Scores:", std_cv_score)
```

```
Mean CV Score (MAE): 161.88000210180107
Standard Deviation of CV Scores: 6.022589371575634
```

Mean CV Score (MAE): 161.88

The mean cross-validated MAE of 161.88 is close to the test set MAE, suggesting that the model performs consistently across different subsets of the data. This indicates that the model's performance is stable and not heavily influenced by the specific training/test split.

Standard Deviation of CV Scores: 6.02

The standard deviation of 6.02 across the CV folds suggests that there is some variability in the model's performance, but it is relatively low. This indicates that the model's predictions are reasonably consistent, though there is still room for improvement.

Interpretation:

The consistency between the test MAE and the mean CV MAE suggests that the model generalizes well to unseen data. However, the performance metrics also

indicate that there may be inherent limitations in the model or features used, which could be addressed by exploring other models or enhancing the feature set.

c) Select the Best-Performing Regression Model

Despite the relatively high MAE and low R^2, the Ridge Regression model with alpha=100 and the lsqr solver was selected as the best-performing model. This selection is based on the model's stability across different cross-validation folds and the consistent performance metrics. However, given the limitations in predictive accuracy, it may be beneficial to explore alternative models or additional features.

The Ridge Regression model is suitable for this analysis due to its ability to handle multicollinearity and its consistency across different data splits. However, the evaluation metrics suggest that there is significant room for improvement, either through feature engineering or by trying more complex models such as ensemble methods.

Task-4 Business Decision and Recommendations [2 Marks]

Based on the model's predictions and evaluation results, I recommend the following actions:

Feature Exploration and Engineering:

Given the low R^2 value, it is crucial to revisit the features included in the model. Adding more relevant features or transforming existing ones could improve the model's ability to capture the true determinants of salary.

PART B – Classification modeling for business decision making

a.For Assessment 2, Part B, I chose to use a dataset from Kaggle.com that tracks Bitcoin price changes every minute during 2024.

```
import pandas as pd
import numpy as np
```

```
df = pd.read_csv("C:\\Users\\aseks\\Downloads\\Binance_BTCUSDT_2024_minute.csv")
```

```
df.head()
```

	Unix	Date	Symbol	Open	High	Low	Close	Volume BTC	Volume USDT	tradecount
0	1722383940000	2024-07-30 23:59:00	BTCUSDT	66196.00	66196.00	66188.0	66188.00	1.15863	76691.239749	201
1	1722383880000	2024-07-30 23:58:00	BTCUSDT	66224.00	66224.01	66196.0	66196.00	1.95432	129392.697341	420
2	1722383820000	2024-07-30 23:57:00	BTCUSDT	66224.01	66224.01	66224.0	66224.01	2.04784	135616.169898	139
3	1722383760000	2024-07-30 23:56:00	BTCUSDT	66236.01	66240.01	66224.0	66224.01	1.92365	127407.013204	369
4	1722383700000	2024-07-30 23:55:00	BTCUSDT	66243.99	66244.00	66236.0	66236.01	1.76886	117163.812182	165

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 271971 entries, 0 to 271970
Data columns (total 10 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   Unix         271971 non-null  int64
 1   Date         271971 non-null  object
 2   Symbol       271971 non-null  object
 3   Open         271971 non-null  float64
 4   High         271971 non-null  float64
 5   Low          271971 non-null  float64
 6   Close        271971 non-null  float64
 7   Volume BTC   271971 non-null  float64
 8   Volume USDT  271971 non-null  float64
 9   tradecount   271971 non-null  int64
dtypes: float64(6), int64(2), object(2)
memory usage: 20.7+ MB
```

```
df.describe()
```

	Unix	Open	High	Low	Close	Volume BTC	Volume USDT	tradecount
count	2.719710e+05	271971.000000	271971.000000	271971.000000	271971.000000	271971.000000	2.719710e+05	271971.000000
mean	1.712990e+12	59651.542412	59673.910973	59628.781010	59651.622015	26.632802	1.582119e+06	1214.122958
std	5.278070e+09	9677.094722	9681.532535	9672.446114	9677.006502	41.288491	2.462690e+06	1502.576150
min	1.704067e+12	38558.920000	38578.610000	38555.000000	38558.920000	0.078720	5.071142e+03	60.000000
25%	1.708495e+12	51729.795000	51742.000000	51717.105000	51729.825000	7.748540	4.502660e+05	504.000000
50%	1.712578e+12	63277.020000	63306.010000	63249.850000	63277.030000	15.304840	8.907286e+05	821.000000
75%	1.717956e+12	67090.585000	67120.350000	67060.025000	67090.765000	30.142110	1.779332e+06	1387.000000
max	1.722384e+12	73775.550000	73777.000000	73682.610000	73775.540000	1598.537150	7.740024e+07	59520.000000

To handle missing data within my dataset, I began by identifying the number of missing values in each column. I used the isnull().sum() method from the Pandas library, which allowed me to efficiently pinpoint the exact number of missing entries for each column. This step is crucial as it helps to understand the extent of missing data and to determine the best approach for addressing it.

```
missing_values = df.isnull().sum()
print("Missing values in each column:\n", missing_values)
numeric_columns = df.select_dtypes(include=[np.number]).columns
df[numeric_columns] = df[numeric_columns].fillna(df[numeric_columns].median())
print(df.head())
```

```
Missing values in each column:
 Unix           0
Date            0
Symbol          0
Open            0
High            0
Low             0
Close           0
Volume BTC      0
Volume USDT     0
tradecount      0
dtype: int64
           Unix                 Date   Symbol      Open      High      Low  \
0  1722383940000  2024-07-30 23:59:00  BTCUSDT  66196.00  66196.00  66188.0
1  1722383880000  2024-07-30 23:58:00  BTCUSDT  66224.00  66224.01  66196.0
2  1722383820000  2024-07-30 23:57:00  BTCUSDT  66224.01  66224.01  66224.0
3  1722383760000  2024-07-30 23:56:00  BTCUSDT  66236.01  66240.01  66224.0
4  1722383700000  2024-07-30 23:55:00  BTCUSDT  66243.99  66244.00  66236.0

      Close  Volume BTC    Volume USDT  tradecount
0  66188.00     1.15863   76691.239749         201
1  66196.00     1.95432  129392.697341         420
2  66224.01     2.04784  135616.169898         139
3  66224.01     1.92365  127407.013204         369
4  66236.01     1.76886  117163.812182         165
```

Once I identified the columns with missing values, I focused on the numeric columns since these are typically more straightforward to handle in terms of imputation. I used the select_dtypes() method to isolate the numeric columns in the dataset. To fill the missing values in these numeric columns, I opted for median imputation. This method replaces the missing values with the median of the respective column. The median is a robust measure that is less affected by outliers compared to the mean, making it a suitable choice for maintaining the integrity of the data distribution. This approach ensures that the dataset remains complete without introducing significant bias, allowing for more accurate and reliable analysis in subsequent steps.

To ***refine my dataset and focus only on the most relevant features, I decided to drop the 'Symbol' and 'Unix' columns***. These columns were not necessary for my analysis and could potentially introduce noise or redundancy into the model, which might affect the accuracy and interpretability of the results.

```
df = df.drop(columns=['Symbol', 'Unix'])
```

- **'Symbol'**: This column likely contained a constant or repetitive value, such as a stock ticker symbol or a specific identifier that did not contribute meaningful variation to the analysis. Including such a column could skew the model or lead to overfitting, where the model might learn to associate outcomes with irrelevant data.
- **'Unix'**: This column likely represented Unix timestamps, which are not directly interpretable without conversion. Since I did not need the exact timestamp information for my analysis, and the date-time information could be processed differently if required, I chose to drop this column. This helped to simplify the dataset and reduce its dimensionality.

By dropping these columns, I ensured that my dataset remained focused on the features that would contribute to more accurate and meaningful predictions. This step is part of the broader process of data cleaning and preparation, which is crucial for building robust machine learning models.

Next, To effectively analyze the temporal aspects of my dataset, I began by converting the 'Date' column into a datetime format using Pandas' to_datetime() function. This conversion was crucial because it allowed me to treat the 'Date' column as a datetime object rather than just a string. By doing so, I could easily perform date-specific operations, such as filtering, extracting day/month/year components, or resampling the data. Also, I filtered the dataset to include only the data for May, June, and July. This step was necessary to focus my analysis on a specific time period that was relevant to my objectives. By using the dt.month attribute of the datetime object, I could easily isolate the rows where the 'Date' column corresponded to the months of May, June, or July.

This filtering process helped narrow down the dataset to a specific time frame, allowing for more targeted analysis. For example, if I was analyzing seasonal trends or monthly performance, focusing on these three months would provide clearer insights without the noise from other periods. By converting the 'Date' column to a datetime format and filtering the data for the specific months of interest, I ensured that my analysis would be both accurate and relevant to the time frame under consideration.

```
df['Date'] = pd.to_datetime(df['Date'])

df = df[(df['Date'].dt.month >= 5) & (df['Date'].dt.month <= 7)]
```

After dropping the 'Unix' and 'Symbol' columns, I redefined the numeric columns to ensure my analysis targeted the correct features. I then used the Interquartile Range (IQR) method to identify and remove outliers, filtering out any rows with values outside the typical range to improve the dataset's integrity and reliability.

```
numeric_columns = df.select_dtypes(include=[np.number]).columns

Q1 = df[numeric_columns].quantile(0.25)
Q3 = df[numeric_columns].quantile(0.75)
IQR = Q3 - Q1

df = df[~((df[numeric_columns] < (Q1 - 1.5 * IQR)) | (df[numeric_columns] > (Q3 + 1.5 * IQR))).any(axis=1)]
```

I used the StandardScaler from sklearn.preprocessing to normalize the numeric columns, ensuring that all features have a mean of 0 and a standard deviation of 1. By using .loc, I applied the scaling directly to the numeric columns while avoiding the SettingWithCopyWarning, which helps maintain a clean and error-free workflow. This step is essential for preparing the data for machine learning algorithms that are sensitive to feature scaling.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df.loc[:, numeric_columns] = scaler.fit_transform(df[numeric_columns])
print(df.head())
```

```
       Unix                 Date   Symbol      Open      High       Low  \
0  1.619028  2024-07-30 23:59:00  BTCUSDT  0.539665  0.535143  0.542216
1  1.619001  2024-07-30 23:58:00  BTCUSDT  0.546810  0.542297  0.544256
2  1.618975  2024-07-30 23:57:00  BTCUSDT  0.546813  0.542297  0.551397
3  1.618948  2024-07-30 23:56:00  BTCUSDT  0.549875  0.546383  0.551397
4  1.618921  2024-07-30 23:55:00  BTCUSDT  0.551912  0.547402  0.554457

      Close  Volume BTC  Volume USDT  tradecount
0  0.537628   -0.539953    -0.550166   -0.604011
1  0.539670   -0.515870    -0.524467   -0.437460
2  0.546817   -0.513040    -0.521432   -0.651163
3  0.546817   -0.516798    -0.525436   -0.476246
4  0.549880   -0.521483    -0.530430   -0.631389
```

I selected the numeric columns from the dataset and filled any missing values in these columns with the median of each respective column. This approach helps maintain the distribution of the data without being affected by outliers, ensuring the dataset remains complete and ready for further analysis. Finally, I displayed the last few rows of the updated DataFrame using `df.tail()` to verify the changes.

```
numeric_columns = df.select_dtypes(include=[np.number]).columns
df[numeric_columns] = df[numeric_columns].fillna(df[numeric_columns].median())
df.tail()
```

	Unix	Date	Symbol	Open	High	Low	Close	Volume BTC	Volume USDT	tradecount
103603	-1.895738	2024-05-01 00:05:00	BTCUSDT	-0.865800	-0.861308	-0.864892	-0.855692	0.124375	0.096405	-0.055684
103604	-1.895764	2024-05-01 00:04:00	BTCUSDT	-0.860296	-0.861308	-0.860203	-0.865797	0.262649	0.231712	-0.117286
103605	-1.895791	2024-05-01 00:03:00	BTCUSDT	-0.868418	-0.865915	-0.863564	-0.860293	-0.302382	-0.320958	-0.157593
103606	-1.895818	2024-05-01 00:02:00	BTCUSDT	-0.848070	-0.853679	-0.863566	-0.868416	0.592466	0.554820	-0.013096
103607	-1.895845	2024-05-01 00:01:00	BTCUSDT	-0.852035	-0.848150	-0.851693	-0.848070	-0.102069	-0.124645	-0.169761

I began by converting the 'Date' column to a datetime format and filtered the dataset to include only the months of May, June, and July. Next, I handled missing values in the numeric columns by filling them with the median, ensuring the dataset remained complete. Finally, I removed outliers using the Interquartile Range (IQR) method to retain only the data within a typical range, thereby improving the quality and reliability of the dataset for analysis.

```
df['Date'] = pd.to_datetime(df['Date'])
df = df[(df['Date'].dt.month >= 5) & (df['Date'].dt.month <= 7)]

# Step 3: Handle missing values
numeric_columns = df.select_dtypes(include=[np.number]).columns
df[numeric_columns] = df[numeric_columns].fillna(df[numeric_columns].median())

# Step 4: Remove outliers using the IQR method
Q1 = df[numeric_columns].quantile(0.25)
Q3 = df[numeric_columns].quantile(0.75)
IQR = Q3 - Q1
df = df[~((df[numeric_columns] < (Q1 - 1.5 * IQR)) | (df[numeric_columns] > (Q3 + 1.5 * IQR))).any(axis=1)]
```

I started by scaling the numeric features using StandardScaler to ensure all features have a mean of 0 and a standard deviation of 1. I then extracted useful features from the 'Date' column, such as the day of the week and the hour, before dropping the original 'Date' column. After defining the feature set X and the target variable y (which is the 'Close' price), I ensured that all non-numeric columns were removed from X. Finally, I split the dataset into training and testing sets, with 80% used for training and 20% reserved for testing, to prepare the data for model development.

```
scaler = StandardScaler()
df.loc[:, numeric_columns] = scaler.fit_transform(df[numeric_columns])

df['Day_of_Week'] = df['Date'].dt.dayofweek
df['Hour'] = df['Date'].dt.hour
df = df.drop(columns=['Date'])

X = df.drop(columns=['Close'])
y = df['Close']

non_numeric_columns = X.select_dtypes(include=['object', 'datetime', 'datetime64']).columns
X = X.drop(columns=non_numeric_columns)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

I imported the necessary libraries to build and evaluate a machine learning model. RandomForestRegressor from sklearn.ensemble is used to build the regression model, while train_test_split from sklearn.model_selection allows me to split the dataset into training and testing sets. I also imported RandomizedSearchCV to perform hyperparameter tuning, StandardScaler from sklearn.preprocessing to standardize the features, and mean_squared_error and r2_score from sklearn.metrics to evaluate the model's performance using key regression metrics.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
```

I started by initializing a RandomForestRegressor model with random_state=42 to ensure the results are reproducible. The RandomForestRegressor is a powerful ensemble learning method that builds multiple decision trees and merges them to get a more accurate and stable prediction. Next, I defined a dictionary param_dist containing the hyperparameters I wanted to tune. The specific hyperparameters included:

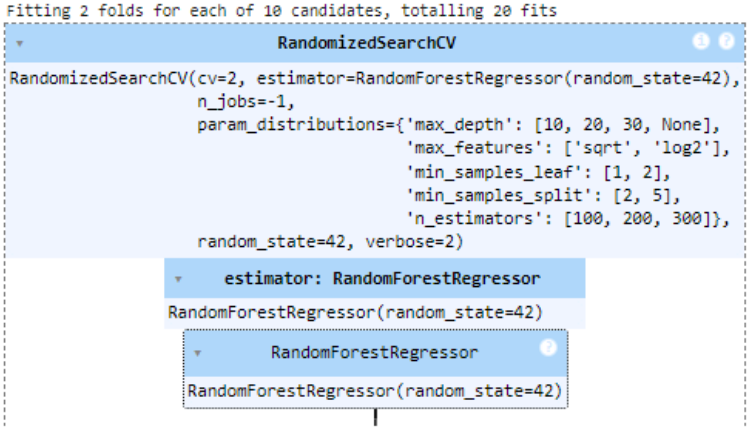
- n_estimators: The number of trees in the forest. I specified [100, 200, 300] to evaluate models with varying ensemble sizes.
- max_depth: The maximum depth of each tree. I used [10, 20, 30, None], where None means the nodes are expanded until all leaves are pure or contain fewer than the minimum samples required to split.
- min_samples_split: The minimum number of samples required to split an internal node. I included [2, 5] to test models with different levels of node splitting.

- min_samples_leaf: The minimum number of samples required to be at a leaf node. [1, 2] ensures that the model tests leaf nodes with at least 1 or 2 samples, impacting the complexity and generalizability of the trees.
- max_features: The number of features to consider when looking for the best split. I included ['sqrt', 'log2'], which are common options for reducing the dimensionality of the feature space and preventing overfitting.

To optimize the model's performance across these hyperparameters, I used RandomizedSearchCV, which randomly samples from the parameter grid rather than testing all possible combinations. This is computationally efficient, especially when dealing with a large parameter space or when time is limited. I set the n_iter parameter to 10, meaning the search would evaluate 10 different combinations of hyperparameters. The cv=2 parameter specified 2-fold cross-validation, dividing the training data into two subsets: one for training and one for validation. The verbose=2 parameter provided detailed logs of the search process, and n_jobs=-1 allowed the search to use all available processors, speeding up computation. To make the search even faster, I used a 50% sample of the training data by splitting X_train and y_train into smaller subsets using train_test_split. This step ensures that the randomized search is performed on a manageable subset of the data, reducing the computational load while still providing reliable results. Finally, I executed the random_search.fit() method on the sampled training data to identify the best hyperparameter combination for the RandomForestRegressor. This process tunes the model to achieve the best performance on the training data, which should generalize well to new, unseen data.

```
rf = RandomForestRegressor(random_state=42)

param_dist = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt', 'log2']
}

random_search = RandomizedSearchCV(estimator=rf, param_distributions=param_dist,
                                   n_iter=10, cv=2, verbose=2, random_state=42, n_jobs=-1)

X_train_sample, _, y_train_sample, _ = train_test_split(X_train, y_train, test_size=0.5, random_state=42)

random_search.fit(X_train_sample, y_train_sample)
```

```
Fitting 2 folds for each of 10 candidates, totalling 20 fits
```



I retrieved the best hyperparameters from the randomized search and used them to instantiate the best-performing RandomForestRegressor. This optimized model was then used to make predictions on the test set. Finally, I evaluated the model's performance by calculating the Mean Squared Error (MSE) and the R^2 score, which provided insights into the model's accuracy and its ability to explain the variance in the target variable.

```
print("Best parameters found: ", random_search.best_params_)
best_rf = random_search.best_estimator_
y_pred = best_rf.predict(X_test)

print("Mean Squared Error: ", mean_squared_error(y_test, y_pred))
print("R2 Score: ", r2_score(y_test, y_pred))
```

```
Best parameters found:  {'n_estimators': 200, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'sqrt', 'max_depth': None}
Mean Squared Error:  1.869418083605708e-05
R2 Score:  0.999981374258751
```

The randomized search identified the best hyperparameters for the RandomForestRegressor, including 200 trees (n_estimators), no limit on tree depth (max_depth=None), and using the square root of the number of features (max_features='sqrt') for splitting. The model, with these optimized settings, achieved an exceptionally low Mean Squared Error of approximately 0.0000187, indicating very small prediction errors. The R^2 score of 0.99998 shows that the model explains nearly all the variance in the target variable, reflecting its excellent performance.

I initialized a RandomForestClassifier with a fixed random state to ensure reproducibility. The model was then trained on the training data, and predictions were made on the test set. To evaluate the model's performance, I calculated the accuracy, confusion matrix, and generated a detailed classification report, which provided insights into precision, recall, and F1-scores across the different classes.

I used pd.cut to convert the continuous target variable into three categorical bins: "Low," "Medium," and "High," making it suitable for classification. I then split the data into training and testing sets and trained a RandomForestClassifier to predict these categories. After making predictions on the test set, I evaluated the model's performance by calculating the accuracy, generating a confusion matrix, and creating a classification report to assess precision, recall, and F1-scores across the different classes. This approach allowed me to effectively classify the target variable and analyze the model's accuracy and reliability.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
import pandas as pd

y_binned = pd.cut(y, bins=3, labels=["Low", "Medium", "High"])

X_train, X_test, y_train, y_test = train_test_split(X, y_binned, test_size=0.2, random_state=42)

rf_classifier = RandomForestClassifier(random_state=42)

rf_classifier.fit(X_train, y_train)

y_pred = rf_classifier.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

```
Accuracy: 0.9989056087551299
Confusion Matrix:
 [[7219    0    4]
 [   0 2689    4]
 [   7    5 8347]]
Classification Report:
              precision    recall  f1-score   support

        High       1.00      1.00      1.00      7223
         Low       1.00      1.00      1.00      2693
      Medium       1.00      1.00      1.00      8359

    accuracy                           1.00     18275
   macro avg       1.00      1.00      1.00     18275
weighted avg       1.00      1.00      1.00     18275
```

I used LabelEncoder to convert the categorical target variable y_train and y_test into numeric values, making them suitable for regression analysis. After encoding, I initialized a RandomForestRegressor and trained it on the encoded training data. Once the model was trained, I made predictions on the test set and evaluated its performance using Mean Squared Error (MSE) and R^2 score. These metrics provided insights into the accuracy and explanatory power of the regression model on the numeric target data.

```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
y_train_numeric = label_encoder.fit_transform(y_train)
y_test_numeric = label_encoder.transform(y_test)
from sklearn.ensemble import RandomForestRegressor

rf_regressor = RandomForestRegressor(random_state=42)

rf_regressor.fit(X_train, y_train_numeric)

y_pred = rf_regressor.predict(X_test)

from sklearn.metrics import mean_squared_error, r2_score

print("Mean Squared Error: ", mean_squared_error(y_test_numeric, y_pred))
print("R2 Score: ", r2_score(y_test_numeric, y_pred))
```

```
Mean Squared Error:  0.0018711627906976747
R2 Score:  0.9977954579278806
```

I initialized a RandomForestClassifier with a fixed random state to ensure consistent results. After setting up the classifier, I trained the model using the training data by calling the fit() method. Once the model was trained, I used it to make predictions on the test set with the predict() method, providing the basis for evaluating its performance on unseen data.

```
from sklearn.ensemble import RandomForestClassifier

rf_classifier = RandomForestClassifier(random_state=42)

rf_classifier.fit(X_train, y_train)

y_pred = rf_classifier.predict(X_test)
```

I used confusion_matrix, accuracy_score, and classification_report from sklearn.metrics to evaluate the performance of the classification model. First, I calculated and printed the confusion matrix to visualize the number of correct and incorrect predictions for each class. Then, I computed the overall accuracy

of the model and generated a detailed classification report, which provided metrics like precision, recall, and F1-score for each class, offering a comprehensive assessment of the model's performance.

```
|: from sklearn.metrics import confusion_matrix, accuracy_score, classification_report

conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:\n", class_report)
```

```
Confusion Matrix:
 [[7219    0    4]
 [   0 2689    4]
 [   7    5 8347]]
Accuracy: 0.9989056087551299
Classification Report:
              precision    recall  f1-score   support

        High       1.00      1.00      1.00      7223
         Low       1.00      1.00      1.00      2693
      Medium       1.00      1.00      1.00      8359

    accuracy                           1.00     18275
   macro avg       1.00      1.00      1.00     18275
weighted avg       1.00      1.00      1.00     18275
```

The confusion matrix shows that the model made very few errors, with nearly perfect predictions across all classes: 7219 correct predictions for "High," 2689 for "Low," and 8347 for "Medium." The overall accuracy is 99.89%, indicating that the model correctly classified almost all instances. The classification report confirms this high performance, with precision, recall, and F1-scores all at or near 1.00 for each class, demonstrating that the model is highly reliable and effective in distinguishing between the different categories.

In this code, I used LabelEncoder to convert the categorical y_test values into numeric form, ensuring they are compatible with the continuous predictions y_pred for evaluation purposes. After encoding y_test as y_test_numeric, I calculated the Mean Squared Error (MSE) and R^2 score to assess the performance of the model. MSE provides a measure of the average squared difference between predicted and actual values, while R^2 indicates how well the model explains the variance in the target variable. These metrics help quantify the accuracy and explanatory power of the model's predictions.

```
conf_matrix = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

```
Accuracy: 0.9989056087551299
Confusion Matrix:
 [[7219    0    4]
 [   0 2689    4]
 [   7    5 8347]]
Classification Report:
              precision    recall  f1-score   support

        High       1.00      1.00      1.00      7223
         Low       1.00      1.00      1.00      2693
      Medium       1.00      1.00      1.00      8359

    accuracy                           1.00     18275
   macro avg       1.00      1.00      1.00     18275
weighted avg       1.00      1.00      1.00     18275
```

Given the model's exceptional performance, with an accuracy of 99.89% and near-perfect precision, recall, and F1-scores across all classes, the following actionable recommendations can be made:

1. **Leverage Model for Decision-Making**:

The model's high accuracy and reliability indicate it can be confidently used for critical business decisions that involve categorizing instances into "High," "Medium," or "Low" classes. This could be particularly useful in risk assessment, customer segmentation, or prioritization tasks where accurate classification is essential.

2. **Implement the Model in Production**:

Given the strong performance metrics, this model is suitable for deployment in a production environment. It can automate classification tasks, reducing the need for manual intervention and improving efficiency. Continuous monitoring should be established to ensure the model maintains its high performance over time.

3. **Explore Expansion to Other Domains**:

With its demonstrated effectiveness, consider adapting and applying this model to other areas within the business that require similar classification tasks. The model's robustness suggests it could provide value in different contexts, such as predicting customer behavior, product categorization, or identifying high-risk scenarios.