Polynomial

July 1, 2024

1 Data Import

This section covers importing data from various sources.

[1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np
import matplotlib.pyplot as plt
```

1.0.1 Data Loading and Initial Exploration

In this section, we load the S&P 500 index data from a CSV file and display the first and last few rows to understand the dataset's structure.

[2]:
```
# import yfinance as yf
# df = yf.download('^SPX', start ='1995-12-27')

df = pd.read_csv('SPX1995.csv')
```

[3]:
```
df.head()
```

[3]:
```
         Date        Open        High         Low       Close   Adj Close  \
0  1995-01-03  459.209991  459.269989  457.200012  459.109985  459.109985
1  1995-01-04  459.130005  460.720001  457.559998  460.709991  460.709991
2  1995-01-05  460.730011  461.299988  459.750000  460.339996  460.339996
3  1995-01-06  460.380005  462.489990  459.470001  460.679993  460.679993
4  1995-01-09  460.670013  461.769989  459.739990  460.829987  460.829987

      Volume
0  262450000
1  319510000
2  309050000
3  308070000
4  278790000
```

[4]:
```
df.tail()
```

[4]:
```
            Date         Open         High          Low        Close  \
7336  2024-02-23  5100.919922  5111.060059  5081.459961  5088.799805
7337  2024-02-26  5093.000000  5097.660156  5068.910156  5069.529785
7338  2024-02-27  5074.600098  5080.689941  5057.290039  5078.180176
7339  2024-02-28  5067.200195  5077.370117  5058.350098  5069.759766
7340  2024-02-29  5085.359863  5104.990234  5061.890137  5096.270020

        Adj Close      Volume
7336  5088.799805  3672790000
7337  5069.529785  3683930000
7338  5078.180176  3925950000
7339  5069.759766  3789370000
7340  5096.270020  5219740000
```

[5]:
```
df.shape
```

[5]:
```
(7341, 7)
```

[6]:
```
df
```

[6]:
```
            Date         Open         High          Low        Close  \
0     1995-01-03   459.209991   459.269989   457.200012   459.109985
1     1995-01-04   459.130005   460.720001   457.559998   460.709991
2     1995-01-05   460.730011   461.299988   459.750000   460.339996
3     1995-01-06   460.380005   462.489990   459.470001   460.679993
4     1995-01-09   460.670013   461.769989   459.739990   460.829987
…            …            …            …            …            …
7336  2024-02-23  5100.919922  5111.060059  5081.459961  5088.799805
7337  2024-02-26  5093.000000  5097.660156  5068.910156  5069.529785
7338  2024-02-27  5074.600098  5080.689941  5057.290039  5078.180176
7339  2024-02-28  5067.200195  5077.370117  5058.350098  5069.759766
7340  2024-02-29  5085.359863  5104.990234  5061.890137  5096.270020

        Adj Close      Volume
0      459.109985   262450000
1      460.709991   319510000
2      460.339996   309050000
3      460.679993   308070000
4      460.829987   278790000
…             …           …
7336  5088.799805  3672790000
7337  5069.529785  3683930000
7338  5078.180176  3925950000
7339  5069.759766  3789370000
7340  5096.270020  5219740000
```

```
[7341 rows x 7 columns]
```

1.0.2 Data Cleaning: Handling Missing Values and Duplicates

This section focuses on identifying and addressing any missing or duplicated data entries to ensure the quality and reliability of the dataset for further analysis.

[7]:
```
missing_values = df.isnull().sum()
df_duplicated= df.duplicated().sum().any()

# here we drop rows if there is missing values
df_cleaned = df.dropna()

print("Missing values in each column:\n", missing_values)
print("\n \n duplicated values :  ", df_duplicated)
```

```
Missing values in each column:
 Date         0
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64


 duplicated values :   False
```

1.0.3 Column Removal

In this section, we remove columns from the dataset that are not needed for our analysis.

[8]:
```
columns_to_drop = ['Adj Close']
df = df.drop(columns_to_drop, axis=1)
df
```

[8]:

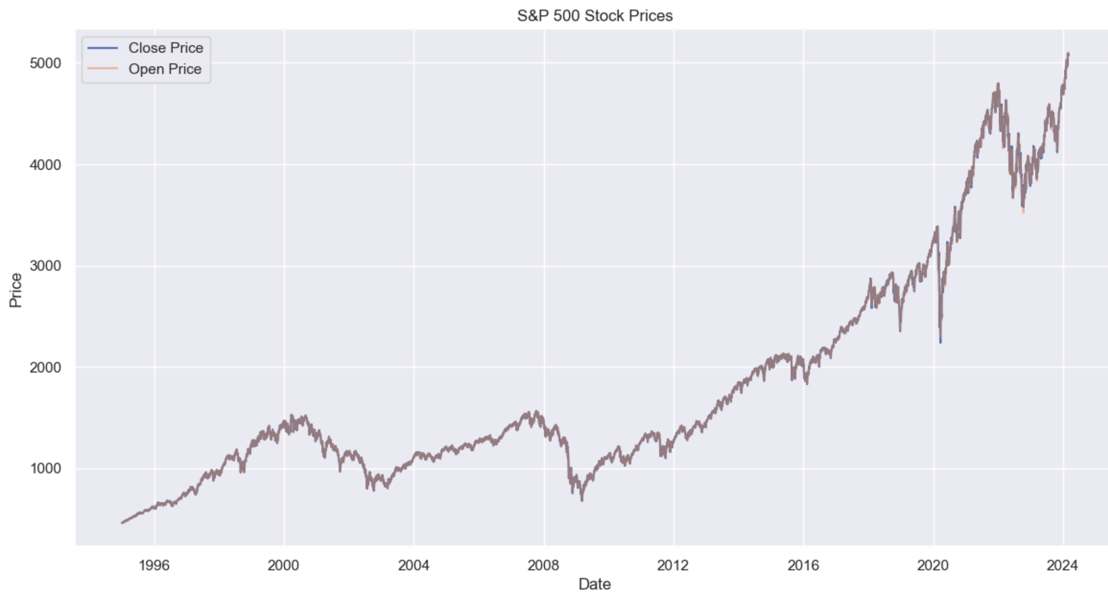
	Date	Open	High	Low	Close	\
0	1995-01-03	459.209991	459.269989	457.200012	459.109985	
1	1995-01-04	459.130005	460.720001	457.559998	460.709991	
2	1995-01-05	460.730011	461.299988	459.750000	460.339996	
3	1995-01-06	460.380005	462.489990	459.470001	460.679993	
4	1995-01-09	460.670013	461.769989	459.739990	460.829987	
…	…	…	…	…	…	
7336	2024-02-23	5100.919922	5111.060059	5081.459961	5088.799805	
7337	2024-02-26	5093.000000	5097.660156	5068.910156	5069.529785	
7338	2024-02-27	5074.600098	5080.689941	5057.290039	5078.180176	

```
7339  2024-02-28  5067.200195  5077.370117  5058.350098  5069.759766
7340  2024-02-29  5085.359863  5104.990234  5061.890137  5096.270020

          Volume
0      262450000
1      319510000
2      309050000
3      308070000
4      278790000
…          …
7336  3672790000
7337  3683930000
7338  3925950000
7339  3789370000
7340  5219740000

[7341 rows x 6 columns]
```

1.0.4 Visualization of S&P 500 Stock Prices

In this section, we convert the 'Date' column to datetime format for proper indexing and plot the S&P 500 closing and opening prices over time to visualize trends and patterns in the data.

[9]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

# Set the style of seaborn
sns.set(style='darkgrid')

# Convert 'Date' to datetime
df['Date'] = pd.to_datetime(df['Date'])

# Plotting the closing prices against the date
plt.figure(figsize=(14, 7))
plt.plot(df['Date'], df['Close'], label='Close Price')

plt.plot(df['Date'], df['Open'], label='Open Price', alpha=0.5)


# Labels and Title
plt.xlabel('Date')
plt.ylabel('Price')
plt.title('S&P 500 Stock Prices')
plt.legend()

# Show plot
plt.show()
```



1.0.5 Feature Preparation

Preparing the dataset for modeling by selecting the 'Open' and 'Volume' as features and 'Close' as the target variable. Converting 'Date' to a numerical format for use in polynomial features.

[10]:
```
# features = [ 'Open', 'High', 'Low', 'Volume']
# target = 'Close'

# # We split the data into features and target
# X = df[features]
# y = df[target]


X = df[['Open', 'High', 'Low', 'Volume']]
y = df['Close']
```

[ ]:
```

```

1.0.6 Data Splitting

Dividing the data into training and test sets to validate the performance of our model. Ensuring a fair distribution without shuffling due to the time-series nature of the data.

[11]:
```
#df['Date_ordinal'] = df['Date'].apply(lambda date: date.toordinal())

degree = 2
poly_features = PolynomialFeatures(degree=degree)
```

```
X_poly = poly_features.fit_transform(X)

# we Save the indexes before the split
original_indexes = df.index

# here we split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_poly, df['Close'],␣
↪test_size=0.2, shuffle=False)

print("the train data: --> ", X_train.shape)
print("the test data: --> ", X_test.shape)
```

```
the train data: -->  (5872, 15)
the test data: -->  (1469, 15)
```

1.0.7 Model Initialization and Fitting

Initializing the Linear Regression model and fitting it to the polynomial-transformed training data to capture non-linear patterns in stock prices.

[12]:
```
# Initialize the Linear Regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, y_train)
```

[12]:
```
LinearRegression()
```

1.0.8 Preparing Index Alignment Before Making Predictions

Before making predictions, it's important to retrieve and store the original indexes of our training and testing sets. This allows us to maintain a reference to the original dates of our data points, ensuring that after we make our predictions, we can accurately analyze and visualize the results in the context of their specific times in the dataset.

[13]:
```
# Retrieve the original indexes for train and test sets
train_indexes = original_indexes[:len(y_train)]
test_indexes = original_indexes[len(y_train):]

# You can now use train_indexes and test_indexes as they contain the original␣
↪DataFrame indexes
print(f'Training data index range: {train_indexes.min()} to {train_indexes.
↪max()}')
print(f'Testing data index range: {test_indexes.min()} to {test_indexes.max()}')
```

```
Training data index range: 0 to 5871
Testing data index range: 5872 to 7340
```

1.0.9 Model Predictions

After training our regression model, we proceed to make predictions on both the training and testing datasets. These predictions will allow us to evaluate the model's performance by comparing the predicted stock prices against the actual closing prices. It's crucial to ensure that the predictions align with the original data's timeline, hence the index retrieval before this step.

[14]:
```
# Predict on the training set for visualization purposes
y_train_pred = model.predict(X_train)

# Make predictions on the testing data
y_test_pred = model.predict(X_test)
```

1.0.10 Organizing and Inspecting Prediction Results

In this section, we consolidate the predictions with the actual values into structured DataFrames, aligning them with their corresponding dates. This organization is essential for an intuitive inspection of the model's predictive accuracy. It also lays the groundwork for subsequent analysis, such as calculating error metrics and visualizing the results.

[15]:
```
# Define train_dates and test_dates by indexing df['Date']
train_dates = df['Date'].iloc[:len(y_train)].reset_index(drop=True)
test_dates = df['Date'].iloc[-len(y_test):].reset_index(drop=True)

# Create DataFrames for the training and test data predictions with dates
train_results = pd.DataFrame({
    'Date': train_dates,
    'Actual_Close': y_train.reset_index(drop=True),
    'Predicted_Close': y_train_pred
})

test_results = pd.DataFrame({
    'Date': test_dates,
    'Actual_Close': y_test.reset_index(drop=True),
    'Predicted_Close': y_test_pred
})

# Now, let's try printing the head of these DataFrames to inspect
print("train_results \n")
print( train_results , "\n \n")

print("test_results \n")
print( test_results)
```

```
train_results

           Date  Actual_Close  Predicted_Close
0    1995-01-03    459.109985       667.145371
1    1995-01-04    460.709991       675.250452
```

```
2    1995-01-05    460.339996       673.398711
3    1995-01-06    460.679993       674.563202
4    1995-01-09    460.829987       670.228492
…         …           …              …
5867 2018-04-23   2670.290039      2913.400999
5868 2018-04-24   2634.560059      2925.766888
5869 2018-04-25   2639.399902      2846.865873
5870 2018-04-26   2666.939941      2912.266104
5871 2018-04-27   2669.909912      2878.095985

[5872 rows x 3 columns]
```

```
test_results
```

```
           Date  Actual_Close  Predicted_Close
0    2018-04-30   2648.050049      2880.951228
1    2018-05-01   2654.800049      2862.861414
2    2018-05-02   2635.669922      2834.173511
3    2018-05-03   2629.729980      2827.491682
4    2018-05-04   2663.419922      2999.733193
…         …           …              …
1464 2024-02-23   5088.799805      8617.675638
1465 2024-02-26   5069.529785      8548.768183
1466 2024-02-27   5078.180176      8455.260613
1467 2024-02-28   5069.759766      8470.372507
1468 2024-02-29   5096.270020      8424.977497

[1469 rows x 3 columns]
```

1.0.11 Performance Metrics Evaluation

In this segment, we compute and display the performance metrics for both the training and testing datasets. This evaluation involves Mean Absolute Error (MAE), Mean Squared Error (MSE) and Root Mean Squared Error (RMSE). These metrics help to quantify the accuracy of our model and reveal how well the predictions match up with the actual stock prices.

[21]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np

# Function to calculate MAPE
def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

# Calculate metrics for the training set
train_mae = mean_absolute_error(train_results['Actual_Close'],␣
 ↪train_results['Predicted_Close'])
```

```
train_rmse = np.sqrt(mean_squared_error(train_results['Actual_Close'],␣
↪train_results['Predicted_Close']))
train_mape = mean_absolute_percentage_error(train_results['Actual_Close'],␣
↪train_results['Predicted_Close'])

# Calculate metrics for the testing set
test_mae = mean_absolute_error(test_results['Actual_Close'],␣
↪test_results['Predicted_Close'])
test_rmse = np.sqrt(mean_squared_error(test_results['Actual_Close'],␣
↪test_results['Predicted_Close']))
test_mape = mean_absolute_percentage_error(test_results['Actual_Close'],␣
↪test_results['Predicted_Close'])

# Print out the metrics for the training set
print("Training set metrics:")
print(f"Mean Absolute Error (MAE): {train_mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {train_rmse:.2f}")
print(f"Mean Absolute Percentage Error (MAPE): {train_mape:.2f}%")

# Print out the metrics for the testing set
print("\nTesting set metrics:")
print(f"Mean Absolute Error (MAE): {test_mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {test_rmse:.2f}")
print(f"Mean Absolute Percentage Error (MAPE): {test_mape:.2f}%")
```

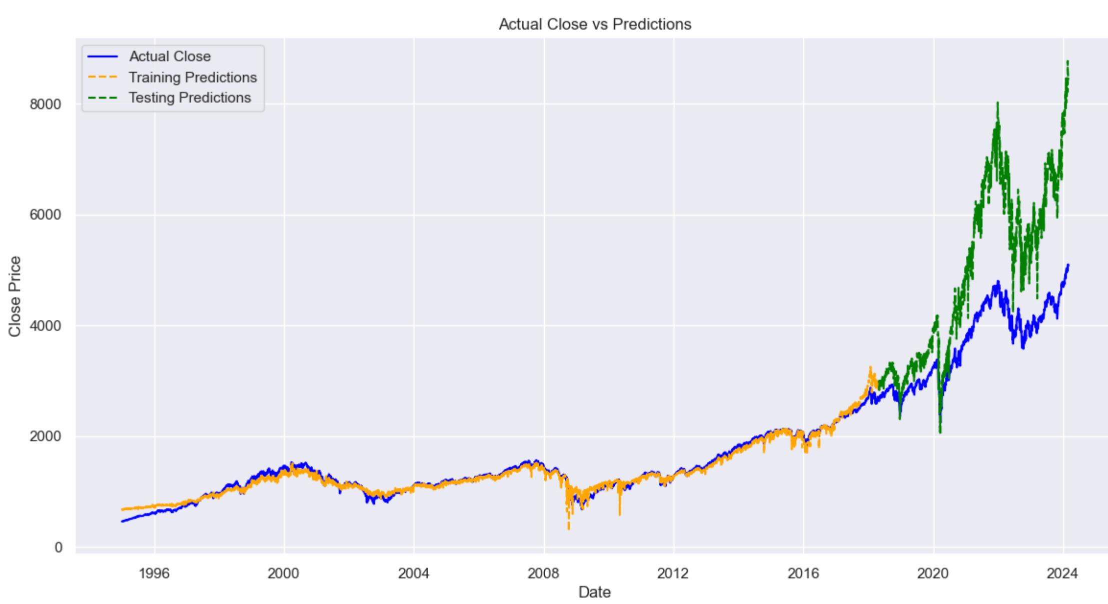
```
Training set metrics:
Mean Absolute Error (MAE): 61.77
Root Mean Squared Error (RMSE): 81.99
Mean Absolute Percentage Error (MAPE): 5.77%

Testing set metrics:
Mean Absolute Error (MAE): 1301.87
Root Mean Squared Error (RMSE): 1576.16
Mean Absolute Percentage Error (MAPE): 32.09%
```

1.0.12 Visualization of Model Predictions Against Actual Data

In accordance with our project's aim to assess machine learning model efficacy, this visualization plots predicted stock prices from our model against the actual S&P 500 closing prices. The graph provides a visual representation of the model's performance over time, showcasing the alignment of predictions with real-world data. This step is crucial for a comprehensive evaluation, allowing for a clear, intuitive understanding of the model's predictive capabilities in both training and testing phases.

[17]:
```
import matplotlib.pyplot as plt
import pandas as pd

# Combine train and test results into a single DataFrame
```

```python
combined_results = pd.concat([train_results, test_results])

# Convert 'Date' to datetime and sort by date to ensure correct plotting order
combined_results['Date'] = pd.to_datetime(combined_results['Date'])
combined_results.sort_values('Date', inplace=True)

# Set 'Date' as the index for plotting
combined_results.set_index('Date', inplace=True)

# Plot the actual close prices
plt.figure(figsize=(14,7))
plt.plot(combined_results['Actual_Close'], label='Actual Close', color='blue')

# Plot the training predictions - we use loc to select the train date range
plt.plot(train_results['Date'], train_results['Predicted_Close'],␣
↪label='Training Predictions', color='orange', linestyle='--')

# Plot the testing predictions - we use loc to select the test date range
plt.plot(test_results['Date'], test_results['Predicted_Close'], label='Testing␣
↪Predictions', color='green', linestyle='--')

# Add labels and title
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.title('Actual Close vs Predictions')
plt.legend()
plt.show()
```



[ ]:

```

```