SDS 496 Final Project Report

Aidan Kardan

December 1st, 2024

Introduction to Dataset and Objectives

US Stock Market Data: Includes 11 datasets from Kaggle with AMZN stock data and various predictors, spanning fundamental, technical and raw historical metrics.

Predictors: Financial ratios, valuation metrics, technical indicators (e.g., RSI, MACD), and historical OHLCV data.

Analysis Goals:

Regression Task: Predict AMZN stock prices, specifically, the close price, using the stock market data.

Classification Task: Predict stock movement (up/down) based on current information.

Data Extraction and Cleaning

After extracting the 11 datasets, and merging them into one dataframe, there were some issues that arose.

Data Repetition

Rationale for Handling Date Repetition

When combining data from multiple sources, each source contributed unique columns, resulting in multiple rows for the same Date. For any given Date, some columns were populated in one source but not in others, leading to overlapping but incomplete information.

Filling Missing Values:

Within all rows for the same date: Forward-fill was used to propagate the last valid value downward. Backward-fill was used to propagate the first valid value upward, addressing gaps at the start of the group.

Selecting the First Consolidated Row:

After filling missing values, the first row from each group was selected as the representative entry for that Date.

This process ensured:

Each Date had a single row in the final dataset. All available information across sources was preserved. Missing values were handled methodically, minimizing data loss while maintaining data integrity.

Forward-Fill Financial Ratios

Forward-filling financial ratios is standard in finance, as these metrics are typically updated quarterly after earnings reports. This approach preserves data integrity by propagating the most recent known values, reflecting real-world decision-making where investors rely on current and past information until new data becomes available.

Financial Ratios are also easy to spot in financial data because of their low frequency of occurence!

Missing Information

Rows missing open, high, low, close, or volume data indicate no trading activity occurred on those days. These rows can be safely removed from the dataset as they do not contribute meaningful information for analysis.

Price Column Removal

The Price column, reflecting stock values at earnings reports, was removed to avoid biases from forward-filling and redundancy with existing daily metrics (Open, High, Low).

Target Variable for Analysis

Using Close price as the target ensures clarity and relevance in our regression and classification analyses.

Tailored Analysis to Align with TA Window

Technical analysis indicators are available from January 2010 to July 2022. Therefore, the analysis is specifically tailored to this time frame to ensure consistency and accuracy.

Miscellaneous NA

Missing macroeconomic data is rare, occurring only on days when updates were not published. Forward-filling addresses these gaps, as these metrics change gradually, and the most recent value remains a reasonable approximation until new data is available. This reflects real-world reliance on the latest information.

At this point, the training and test data are ready for statistical analysis (no longer have any NA values).

Time Series Data Issues

When dealing with time series data, standard training-test splits with randomization, and Cross-Validation or Bootstrap techniques cannot be used for analysis. To fix this, we will utilize a time-series split/time-series CV, which preserves the integrity of the chronological order of observations by sequentially splitting the data.

This method ensures that future data points are never used to predict past ones, maintaining the temporal structure critical for accurate financial modeling.

Statistical Analysis

Exploratory Analysis

In this analysis, I explored a wide range of statistical and machine learning models to address the regression and classification problems. For regression, I utilized Linear Regression, Ridge Regression, Lasso Regression, PCA Regression, Random Forests, Gradient Boosting, Linear SVM, Non-linear SVM, KMeans Clustering, Hierarchical Clustering, and Model-Based (GMM) Clustering. For classification, I focused on Classification Trees.

This diverse selection of models was chosen to handle the inherent complexity of financial data, which is often characterized by high dimensionality, collinearity, non-linearity, and the presence of outliers. Each method was selected for its unique ability to address these challenges, identify patterns, and generalize well to unseen data.

While clustering methods are not directly predictive, as they do not use the response variable during training, they can still provide meaningful insights. If clusters naturally align with variations in the response variable, they can serve as useful approximations, and cluster labels can be incorporated as new predictors in regression models to enhance performance.

To begin, all 97 predictors were included to ensure no information was excluded prematurely. Baseline models were created to evaluate the full dataset before applying statistically backed feature selection methods to reduce dimensionality and prevent overfitting.

The goal of this exploratory analysis was to determine feature importance and identify the best combination of predictors for a mixed model.

A rigorous exploratory analysis was conducted to better understand the data, including:

Histogram Analysis: To assess the distribution of the response variable and predictors.

Scatterplot Analysis: To visualize relationships between predictors and the response variable.

Correlation Analysis: To identify collinear relationships among predictors.

From the histogram analysis, it was evident that the response variable needed to be log-transformed to better conform to normality.

Correlation analysis revealed strong collinearity among financial ratio predictors and similarly among technical analysis-based predictors.

Scatterplots showed that many technical analysis-based predictors exhibited a strong linear relationship with the response variable.

Feature Importance and Baseline Model Testing

Baseline models were constructed using the full set of predictors, both with the raw response variable and the log-transformed response. Feature importance was evaluated for each model to identify the top predictors contributing to performance. Baseline model performance was assessed using RMSE on the test set, providing a benchmark for further refinement.

Why RMSE Was Chosen:

RMSE was chosen because I am predicting close prices, and my focus is on the absolute difference between predictions and actual observations. RMSE is expressed in the same units as the close price, making it both intuitive and easy to understand.

Time Series Cross-Validation (Time Series Split):

To ensure robustness in training and testing, I used time series cross-validation (time series split). This method respects the temporal ordering of the data, ensuring that the training set always precedes the test set and avoids data leakage.

Mixed Model Development

After analyzing feature importance across methods, I identified the top 10 predictors from each method and selected unique predictors to create an optimal mixed model. This approach ensured that the model leveraged the most informative features across all methods while minimizing redundancy.

The mixed model was then evaluated across all baseline methods to assess its performance relative to the full predictor models. The goal of this step was to determine whether the mixed model, with a reduced yet diverse set of predictors, could maintain or improve predictive accuracy and generalization.

Final Comparison

The optimally mixed model was compared to the baseline models to assess improvements in prediction accuracy, generalization, and overall model efficiency. Among the models using the log-transformed response variable, Ridge and Lasso Regression demonstrated the lowest RMSE values, effectively managing multicollinearity through regularization. In contrast, models considering all predictors generally outperformed the mixed model in cases where the dataset's complexity required capturing hidden patterns.

Model-Specific Insights:

Ridge and Lasso Regression: These methods performed well with the full predictor set but saw reduced predictive power when applied to the mixed model. This suggests the reduced feature set outweighed the benefits of regularization.

Random Forest and Gradient Boosting: Tree-based models benefited significantly from the mixed model, as irrelevant predictors were removed, allowing the models to focus on the most important features.

Linear SVM and PCA Regression: Both models showed improved performance with the mixed model, as feature selection reduced noise and emphasized key linear relationships.

Results

Full, Baseline Models

The most relevant and important results are highlighted below:

Linear Regression Average RMSE (Log Scale): 66247924.34874252

Linear Regression Average RMSE (Original Scale): inf

PCA Regression Average RMSE (Log Scale): 0.3535938673414937

PCA Regression Average RMSE (Original Scale): 1.424176664354599

Gradient Boosting Average RMSE (Log Scale): 0.24348653960017835

Gradient Boosting Average RMSE (Original Scale): 1.2756891464391382

Random Forest Average RMSE (Log Scale): 0.2407474488995537

Random Forest Average RMSE (Original Scale): 1.2721996992990103

Ridge Regression Average RMSE (Log Scale): 0.04601900778501562

Ridge Regression Average RMSE (Original Scale): 1.047094313711389

Lasso Regression Average RMSE (Log Scale): 0.041607199963150494

Lasso Regression Average RMSE (Original Scale): 1.0424849102056157

Linear SVM Average RMSE (Log Scale): 0.4667938446172881

Linear SVM Average RMSE (Original Scale): 1.5948725779569104

Non-linear SVM Average RMSE (Log Scale): 0.6068300772895748

Non-linear SVM Average RMSE (Original Scale): 1.8346066104355785

KMeans Clustering RMSE for Log Close (Log Scale): 0.23802173305004115

KMeans Clustering RMSE for Log Close (Original Scale): 1.2687367660449103

Hierarchical Clustering RMSE for Log Close (Log Scale): 0.3723168547812369

Hierarchical Clustering RMSE for Log Close (Original Scale): 1.4510926941163582

Model-Based Clustering RMSE for Log Close (Log Scale): 0.23845845586042463

Model-Based Clustering RMSE for Log Close (Original Scale): 1.2692909733396893

Optimally Mixed Models

Linear Regression RMSE Original Scale: 1.3166661668616408

Ridge RMSE Original Scale: 1.2355178856459195

Lasso RMSE Original Scale: 1.2071336836237347

Random Forest RMSE Original Scale: 1.2698637777706667

Gradient Boosting RMSE Original Scale: 1.2753331882329946

Linear SVM RMSE Original Scale: 1.5272381690448305

PCA Regression RMSE Original Scale: 1.2641572788408475

Conclusion and Future Directions

In this project, I analyzed a wide range of statistical and machine learning models to address regression and classification problems using financial data. The analysis began with an exploratory evaluation of all 97 predictors, identifying strong linear and non-linear relationships with the response variable. This was followed by the development and testing of baseline models across all predictors, using RMSE as the primary evaluation metric. Feature importance analysis led to the creation of a mixed model, leveraging the most informative predictors from multiple methods.

Among the models evaluated, Lasso Regression with all predictors emerged as the best-performing model, achieving the lowest RMSE. This result highlights Lasso's ability to handle multicollinearity while simplifying the model by shrinking irrelevant coefficients to zero. Ridge Regression also performed well, but its lack of inherent feature selection resulted in slightly reduced efficiency.

Tree-based methods like Random Forest and Gradient Boosting showed significant improvements when irrelevant predictors were removed, while PCA Regression and Linear SVM benefited from targeted feature selection to reduce noise.

Time series cross-validation was employed to ensure robust evaluation, respecting the temporal structure of the financial dataset and avoiding data leakage. This methodological rigor provided reliable insights into the predictive power and generalizability of each model.

While the academic objectives of this project have been fully met, this analysis has sparked curiosity about further enhancing stock price prediction through advanced techniques.

In particular, time series analysis (TSA) is a promising avenue to explore. TSA can incorporate lagged relationships, address non-stationarity, and identify temporal patterns to enhance predictive performance. Insights from TSA can be integrated with regression models, combining the strengths of traditional regression with time series dynamics.

Additionally, I plan to investigate Neural Networks (NNs) and Long Short-Term Memory (LSTM) networks to capture the complex, non-linear relationships and sequential dependencies inherent in financial data. Expanding the classification problem by leveraging insights from regression and clustering analyses will also allow for the development of a more comprehensive framework that balances feature importance, temporal trends, and non-linear dynamics.

In summary, this project has demonstrated the value of combining exploratory analysis, feature selection, and robust model evaluation techniques to address the challenges of complex financial datasets. While the current work fulfills its academic goals, the insights gained provide a strong foundation for future exploration and personal growth in advanced modeling techniques.

Further Statistical Analysis and Results

The full statistical analysis encompassed feature importance selection, model selection, and model evaluation. All steps, including data extraction, cleaning, and preprocessing, were carefully implemented to ensure robust and reproducible results.

The code used for these tasks will be provided below, the code provides insights into the methodologies and evaluation metrics that shaped the findings presented above.

Data Extraction from Kaggle Files

In [1]:
```
import pandas as pd

# Common file path and list of file endings
common_path = '/Users/aidanashrafi/Downloads/'
file_endings = [
    'market_indicators.csv',
    'AMZN_fundamentals_train.csv',
    'AMZN_fundamentals_test.csv',
    'AMZN_technicals_test.csv',
    'AMZN_technicals_train.csv',
    'revenue_profit.csv',
    'ratios.csv',
    'price_fundementals.csv',
    'assets_liabilities.csv',
    'price.csv',
    'fundementals.csv'
]

# Read and combine data
dataframes = [pd.read_csv(f"{common_path}{ending}") for ending in file_endir
full_data = pd.concat(dataframes, ignore_index=True)
```

Data Cleaning:

In [ ]:
```
# Ensure 'Date' is in datetime format
full_data['Date'] = pd.to_datetime(full_data['Date'])
# Remove duplicate columns
full_data = full_data.loc[:, ~full_data.columns.duplicated()]
# Sort the data by 'Date' to ensure chronological order
full_data.sort_values(by='Date', inplace=True)
# Reset the index for a clean DataFrame
full_data.reset_index(drop=True, inplace=True)
# Deleting Duplicate columns after inspection
full_data.drop(columns=['pb_ratio', 'ps_ratio', 'pfcf_ratio', 'price'], inpl

print(f"Number of columns after removing duplicates: {len(full_data.columns)

# Group by 'Date', fill missing values, and consolidate rows
full_data = (
    full_data.groupby('Date', as_index=False)
    .apply(lambda group: group.drop(columns=['Date']).ffill().bfill().iloc[0
    .reset_index(drop=True)
)
```

Missing Values

Determine number of missing values for each variable

```
In [ ]: missing_summary = full_data.isnull().sum().sort_values(ascending=False)
        missing_percentage = (missing_summary / len(full_data)) * 100
        print(missing_percentage[missing_percentage > 0])
```

```
In [ ]: # Identify columns with more than 95% missing values
        columns_to_ffill = missing_percentage[missing_percentage > 95].index.tolist(
        # Apply forward-fill for these columns
        full_data[columns_to_ffill] = full_data[columns_to_ffill].ffill()

        # If necessary, apply backward-fill
        full_data[columns_to_ffill] = full_data[columns_to_ffill].bfill()
        # Check the results
        print("Columns processed with forward-fill", columns_to_ffill)
```

```
In [5]: # Check for missing values in core market data
        missing_market_data = full_data[['open', 'high', 'low', 'close', 'volume']].
        # Remove rows where all four columns are missing
        full_data = full_data[~missing_market_data].reset_index(drop=True)
```

```
In [ ]: # Split the data into training (2010-2020) and testing (2020-2023) periods
        train_data = full_data[(full_data['Date'] >= '2010-01-14') & (full_data['Dat
        test_data = full_data[(full_data['Date'] > '2019-12-31') & (full_data['Date'

        # Check the results
        print("Training Data Range:", train_data['Date'].min(), "-", train_data['Dat
        print("Testing Data Range:", test_data['Date'].min(), "-", test_data['Date']
```

```
In [ ]: missing_summary = train_data.isnull().sum().sort_values(ascending=False)
        missing_percentage = (missing_summary / len(full_data)) * 100
        print(missing_percentage[missing_percentage > 0])
```

```
In [ ]: # Missing Data Train
        # List of macroeconomic indicators
        macroeconomic_data = ['gold', 'treasury_5_years', 'treasury_10_years', 'trea

        train_data.loc[:, macroeconomic_data] = train_data.loc[:, macroeconomic_data

        missing_summary_train = train_data.isnull().sum().sort_values(ascending=Fals
        missing_percentage_train = (missing_summary_train / len(full_data)) * 100
        print(missing_percentage_train[missing_percentage_train > 0])
```

```
In [ ]: missing_summary_test = test_data.isnull().sum().sort_values(ascending=False)
        missing_percentage_test = (missing_summary_test / len(full_data)) * 100
        print(missing_percentage_test[missing_percentage_test > 0])
```

It is clear that there are no longer missing values in the training or test set, so we can proceed with the analysis.

Histograms

In [ ]:
```
# Histograms

import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt

# Generate histograms with a normal distribution overlay for numeric variabl
for column in train_data.columns:
    if train_data[column].dtype in ['int64', 'float64']:  # Only numeric col
        data = train_data[column].dropna() # Drop NaN values if any
        mean, std = data.mean(), data.std()  # Calculate mean and standard d

        plt.figure(figsize=(8, 4))

        # Plot histogram
        plt.hist(data, bins=30, density=True, edgecolor='k', alpha=0.7, labe

        # Overlay normal distribution
        xmin, xmax = plt.xlim()
        x = np.linspace(xmin, xmax, 100)
        p = norm.pdf(x, mean, std)
        plt.plot(x, p, 'r', linewidth=2, label='Normal Distribution')

        # Add title, labels, and legend
        plt.title(f"Histogram with Normal Distribution Overlay for {column}"
        plt.xlabel(column)
        plt.ylabel("Density")
        plt.legend()
        plt.grid(axis='y', linestyle='--', alpha=0.7)

        plt.show()
```

The histogram for close shows us that we likely want to log transform the response variable to achieve normality, depending on the model chosen for further analysis.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Apply log transformation to the response
train_data['log_close'] = np.log(train_data['close'])

# Calculate mean and standard deviation of the log-transformed response
mean_log_close = np.mean(train_data['log_close'])
std_log_close = np.std(train_data['log_close'])

# Generate points for the normal distribution
x = np.linspace(min(train_data['log_close']), max(train_data['log_close']),
normal_pdf = norm.pdf(x, mean_log_close, std_log_close)

# Plot histogram of the log-transformed response
```

```
plt.figure(figsize=(8, 5))
plt.hist(train_data['log_close'], bins=50, alpha=0.7, color='blue', edgecolo

# Overlay the normal distribution
plt.plot(x, normal_pdf, color='red', linewidth=2, label='Normal Distributio

# Add titles and labels
plt.title("Histogram of Log-Transformed Close with Normal Distribution")
plt.xlabel("Log(Close)")
plt.ylabel("Density")
plt.legend()
plt.grid(axis='y', alpha=0.75)
plt.show()
```

Scatterplots

In [ ]:
```
import matplotlib.pyplot as plt

# Generate scatterplots for each numeric variable with the log-transformed r
for column in train_data.columns:
    if column not in ['close', 'log_close'] and train_data[column].dtype in
        plt.figure(figsize=(8, 4))
        plt.scatter(train_data[column], train_data['log_close'], alpha=0.5)
        plt.title(f"Scatterplot of {column} vs Log(Close)")
        plt.xlabel(column)
        plt.ylabel('Log(Close)')
        plt.grid(alpha=0.5)
        plt.show()
```

Correlation Analysis

Pairwise Correlations

In [ ]:
```
# Pairwise Correlations
import seaborn as sns
import matplotlib.pyplot as plt

# Update numeric data to include log-transformed response
numeric_data = train_data.select_dtypes(include=['int64', 'float64']).drop(c
correlation_matrix = numeric_data.corr()

# Plot the heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm', fmt='.2f', lin
plt.title("Pairwise Correlations Heatmap")
plt.show()
```

Correlation with Target Transformed Response

In [ ]:
```
# Calculate the correlation of all numeric predictors with the log-transform
correlation_with_target = numeric_data.corrwith(train_data['log_close'])  #
```

```
# Set a correlation threshold (e.g., |correlation| >= 0.3)
high_corr_threshold = 0.3

# Filter predictors based on the threshold
selected_features_corr = correlation_with_target[abs(correlation_with_target

# Display the selected features
print("Features Selected by Correlation with Log(Close):")
print(selected_features_corr)

# Subset the dataset to include only the selected predictors
X_corr_filtered = train_data[selected_features_corr]

# Number of columns in the filtered dataset
num_columns = X_corr_filtered.shape[1]
print(f"Number of predictors selected: {num_columns}")
```

In [ ]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import numpy as np

X = numeric_data
y = train_data['log_close']

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store RMSE for each split
rmse_list = []

# Perform time series split and train/test evaluation for full linear regres
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a linear regression model
    lr_model = LinearRegression()
    lr_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = lr_model.predict(X_test)

    # Calculate RMSE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)

# Average RMSE across all splits
average_rmse_log_scale = sum(rmse_list) / len(rmse_list)
```

```
# Convert RMSE from log scale to original scale
average_rmse_original_scale = np.exp(average_rmse_log_scale)
```

In [ ]:
```
# Print results
print(f"Linear Regression Average RMSE (Log Scale): {average_rmse_log_scale}
print(f"Linear Regression Average RMSE (Original Scale): {average_rmse_origi
```

PCA Analysis with Transformed Target

In [ ]:
```
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Prepare data
X = numeric_data  # Predictors
y = train_data['log_close']  # Log-transformed response variable

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Perform PCA
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Determine explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_[:5]  # First 5 comp
cumulative_explained_variance = np.cumsum(pca.explained_variance_ratio_[:5])

# Display explained variance for the first 5 components
explained_variance_table = pd.DataFrame({
    "Principal Component": [f"PC{i+1}" for i in range(5)],
    "Explained Variance (%)": explained_variance_ratio * 100,
    "Cumulative Explained Variance (%)": cumulative_explained_variance * 100
})
print("Explained Variance Table (First 5 Components):")
print(explained_variance_table)
```

In [ ]:
```
# Visualize cumulative explained variance (all components)
cumulative_explained_variance_all = np.cumsum(pca.explained_variance_ratio_)
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(cumulative_explained_variance_all) + 1), cumulative_ex
plt.title("Cumulative Explained Variance by PCA Components")
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Explained Variance")
plt.grid()
plt.show()
```

In [ ]:
```
# Choose the number of components (e.g., explaining 95% variance)
n_components = np.argmax(cumulative_explained_variance_all >= 0.95) + 1
X_pca_reduced = X_pca[:, :n_components]

# Time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store MSPE for each split
rmse_list = []

# Perform time series split and train/test evaluation
for train_index, test_index in tscv.split(X_pca_reduced):
    X_train, X_test = X_pca_reduced[train_index], X_pca_reduced[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a linear regression model using PCA components
    lr_model = LinearRegression()
    lr_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = lr_model.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"PCA Regression Average RMSE (Log Scale): {average_rmse}")
print(f"PCA Regression Average RMSE (Original Scale): {np.exp(average_rmse)}
```

In [ ]:
```
# import numpy as np
# import pandas as pd

# Get loadings (coefficients of original variables in principal components)
loadings = pd.DataFrame(
    pca.components_.T,
    columns=[f"PC{i+1}" for i in range(len(pca.components_))],
    index=numeric_data.columns
 )

# Convert loadings into a DataFrame for readability, focusing on the first 5
loadings_df = loadings.iloc[:, :5].reset_index()
loadings_df.rename(columns={'index': 'Variable'}, inplace=True)

# Display the DataFrame
print("Principal Component Loadings (Top Components):")
print(loadings_df)
```

In [ ]:
```
# Loop through all principal components and plot cumulative contributions
for i in range(loadings.shape[1]):  # Number of components
    component_name = f"PC{i+1}"

    plt.figure(figsize=(12, 6))
    cumulative_contribution = loadings.abs().cumsum(axis=0)  # Cumulative su
    cumulative_contribution[component_name].sort_values(ascending=False).plo

    plt.title(f'Cumulative Contribution of Variables to {component_name}')
    plt.xlabel('Variables')
    plt.ylabel('Cumulative Contribution')
    plt.grid(axis='y')
    plt.show()
```

Gradient Boosting

Transformed Response

In [ ]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import pandas as pd
import matplotlib.pyplot as plt

# Prepare predictors and response
X = numeric_data.drop(columns=['close', 'log_close'], errors='ignore')  # Ex
y = train_data['log_close']  # Log-transformed response variable

# Time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store MSPE for each split
rmse_list = []

# Iterate through time series splits and fit Gradient Boosting
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train Gradient Boosting
    gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)
    gb_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = gb_model.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Gradient Boosting Average RMSE (Log Scale): {average_rmse}")
print(f"Gradient Boosting Average RMSE (Original Scale): {np.exp(average_rms
```

In [ ]:
```
# Feature Importance
gb_feature_importance = pd.Series(gb_model.feature_importances_, index=X.col
gb_feature_importance.sort_values(ascending=False, inplace=True)
```

```
# Plot feature importance
gb_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Top
plt.xlabel('Features')
plt.ylabel('Importance Score')
plt.grid(axis='y')
plt.show()
```

Normal Close Values

In [ ]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import pandas as pd
import matplotlib.pyplot as plt

# Prepare predictors and response
X = numeric_data.drop(columns=['close', 'log_close'], errors='ignore')  # Ex
y = train_data['close']  # Log-transformed response variable

# Time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store MSPE for each split
rmse_list = []

# Iterate through time series splits and fit Gradient Boosting
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train Gradient Boosting
    gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)
    gb_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = gb_model.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average RMSE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Gradient Boosting Average RMSE: {average_rmse}")
```

In [ ]:
```
# Feature Importance
gb_feature_importance = pd.Series(gb_model.feature_importances_, index=X.col
gb_feature_importance.sort_values(ascending=False, inplace=True)

# Plot feature importance
gb_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Top
plt.xlabel('Features')
plt.ylabel('Importance Score')
plt.grid(axis='y')
plt.show()
```

Random Forests

Transformed Response

In [ ]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import pandas as pd
import matplotlib.pyplot as plt

# Prepare predictors and response
X = numeric_data.drop(columns=['close', 'log_close'], errors='ignore')  # Ex
y = train_data['log_close']  # Log-transformed response variable

# Time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store MSPE for each split
rmse_list = []

# Iterate through time series splits and fit Random Forest
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train Random Forest
    rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
    rf_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = rf_model.predict(X_test)

    # Calculate MSPE (fixed for FutureWarning)
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Random Forest Average RMSE (Log Scale): {average_rmse}")
print(f"Random Forest Average RMSE (Original Scale): {np.exp(average_rmse)}"
```

In [ ]:
```
# Feature Importance
rf_feature_importance = pd.Series(rf_model.feature_importances_, index=X.col
rf_feature_importance.sort_values(ascending=False, inplace=True)

# Plot feature importance
rf_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Top
plt.xlabel('Features')
plt.ylabel('Importance Score')
plt.grid(axis='y')
plt.show()
```

Normal Close Values

```
In [ ]: from sklearn.ensemble import RandomForestRegressor
        from sklearn.model_selection import TimeSeriesSplit
        from sklearn.metrics import mean_squared_error
        import pandas as pd
        import matplotlib.pyplot as plt

        # Prepare predictors and response
        X = numeric_data.drop(columns=['close', 'log_close'], errors='ignore')  # Ex
        y = train_data['close']  # Log-transformed response variable

        # Time series split
        tscv = TimeSeriesSplit(n_splits=5)

        # Initialize list to store MSPE for each split
        rmse_list = []

        # Iterate through time series splits and fit Random Forest
        for train_index, test_index in tscv.split(X):
            X_train, X_test = X.iloc[train_index], X.iloc[test_index]
            y_train, y_test = y.iloc[train_index], y.iloc[test_index]

            # Train Random Forest
            rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
            rf_model.fit(X_train, y_train)

            # Predict on the test set
            y_pred = rf_model.predict(X_test)

            # Calculate MSPE (fixed for FutureWarning)
            rmse = mean_squared_error(y_test, y_pred, squared=False)
            rmse_list.append(rmse)
```

```
In [ ]: # Average MSPE across all splits
        average_rmse = sum(rmse_list) / len(rmse_list)
        print(f"Random Forest Average RMSE for Close: {average_rmse}")
```

```
In [ ]: # Feature Importance
        rf_feature_importance = pd.Series(rf_model.feature_importances_, index=X.col
        rf_feature_importance.sort_values(ascending=False, inplace=True)

        # Plot feature importance
        rf_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Top
        plt.xlabel('Features')
        plt.ylabel('Importance Score')
        plt.grid(axis='y')
        plt.show()
```

Classification Trees

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import accuracy_score
import pandas as pd
import matplotlib.pyplot as plt

# Prepare data for classification (example: predict if 'close' is increasing
train_data['Direction'] = (train_data['close'].diff() > 0).astype(int)  # 1

X = numeric_data  # Predictors
y = train_data['Direction']  # Binary target variable

# Time series cross-validation
tscv = TimeSeriesSplit(n_splits=5)

# Store feature importance and accuracy for each fold
feature_importances = []
accuracies = []

for train_index, test_index in tscv.split(X):
    # Split data while respecting time order
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a classification tree
    clf_tree = DecisionTreeClassifier(random_state=42, max_depth=5)
    clf_tree.fit(X_train, y_train)

    # Extract and store feature importance
    tree_feature_importance = pd.Series(clf_tree.feature_importances_, index
    feature_importances.append(tree_feature_importance)

    # Evaluate accuracy on the test set
    y_pred = clf_tree.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)

# Average feature importance across folds
average_importance = pd.concat(feature_importances, axis=1).mean(axis=1)
average_importance.sort_values(ascending=False, inplace=True)

# Plot the average feature importance
average_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Average
plt.xlabel('Features')
plt.ylabel('Average Importance Score')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Print average feature importance values
print("Average Classification Tree Feature Importance (Time Series CV):")
print(average_importance)
```

In [ ]:
```
# Print average accuracy across folds
print(f"Average Accuracy Across Folds: {sum(accuracies) / len(accuracies):.2
```

Regularization-Based Methods

Log Transformed Response

```
In [ ]: from sklearn.linear_model import RidgeCV
        from sklearn.preprocessing import StandardScaler
        from sklearn.model_selection import TimeSeriesSplit
        from sklearn.metrics import mean_squared_error
        import matplotlib.pyplot as plt
        import pandas as pd
        import numpy as np

        # Prepare data
        X = numeric_data  # Predictors
        y = train_data['log_close']  # Response variable

        # Standardize predictors
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # Define time series split
        tscv = TimeSeriesSplit(n_splits=5)

        # Ridge Regression with Time Series CV
        ridge_model = RidgeCV(alphas=[0.1, 1.0, 10.0, 100.0], cv=tscv)
        ridge_model.fit(X_scaled, y)

        # Predict on entire data (as RidgeCV optimizes internally)
        y_pred = ridge_model.predict(X_scaled)

        # Calculate MSPE on a manual test split for consistency
        rmse_list = []
        for train_index, test_index in tscv.split(X_scaled):
            X_train, X_test = X_scaled[train_index], X_scaled[test_index]
            y_train, y_test = y.iloc[train_index], y.iloc[test_index]

            # Use the pre-fitted ridge_model to predict
            y_pred = ridge_model.predict(X_test)

            # Calculate MSPE for the current fold
            rmse = mean_squared_error(y_test, y_pred, squared=False)
            rmse_list.append(rmse)
```

```
In [ ]: # Average MSPE across all splits
        average_rmse = sum(rmse_list) / len(rmse_list)
        print(f"Ridge Regression Average RMSE (Log Scale): {average_rmse}")
        print(f"Ridge Regression Average RMSE (Original Scale): {np.exp(average_rmse
```

```
In [ ]: # Extract and sort coefficients by magnitude
        ridge_coefficients = pd.Series(np.abs(ridge_model.coef_), index=X.columns)
        ridge_coefficients.sort_values(ascending=False, inplace=True)

        # Plot top coefficients
```

```
ridge_coefficients.head(20).plot(kind='bar', figsize=(12, 6), title="Top Ric
plt.xlabel('Features')
plt.ylabel('Absolute Coefficient Value')
plt.grid(axis='y')
plt.show()

# Print all coefficients sorted by magnitude
print("Ridge Regression Coefficients by Magnitude for Log Close:")
print(ridge_coefficients)
```

Normal Close Values

In [ ]:
```
from sklearn.linear_model import RidgeCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Prepare data
X = numeric_data  # Predictors
y = train_data['close']  # Response variable

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define time series split
tscv = TimeSeriesSplit(n_splits=5)

# Ridge Regression with Time Series CV
ridge_model = RidgeCV(alphas=[0.1, 1.0, 10.0, 100.0], cv=tscv)
ridge_model.fit(X_scaled, y)

# Predict on entire data (as RidgeCV optimizes internally)
y_pred = ridge_model.predict(X_scaled)

# Calculate MSPE on a manual test split for consistency
rmse_list = []
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Use the pre-fitted ridge_model to predict
    y_pred = ridge_model.predict(X_test)

    # Calculate MSPE for the current fold
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Ridge Regression Average MSPE for Close: {average_rmse}")
```

```
In [ ]: # Extract and sort coefficients by magnitude
        ridge_coefficients = pd.Series(np.abs(ridge_model.coef_), index=X.columns)
        ridge_coefficients.sort_values(ascending=False, inplace=True)

        # Plot top coefficients
        ridge_coefficients.head(20).plot(kind='bar', figsize=(12, 6), title="Top Ric
        plt.xlabel('Features')
        plt.ylabel('Absolute Coefficient Value')
        plt.grid(axis='y')
        plt.show()

        # Print all coefficients sorted by magnitude
        print("Ridge Regression Coefficients by Magnitude:")
        print(ridge_coefficients)
```

Log Transformed Response

```
In [ ]: from sklearn.linear_model import LassoCV
        from sklearn.preprocessing import StandardScaler
        from sklearn.model_selection import TimeSeriesSplit
        from sklearn.metrics import mean_squared_error
        import matplotlib.pyplot as plt
        import pandas as pd
        import numpy as np

        # Prepare data
        X = numeric_data  # Predictors
        y = train_data['log_close']  # Response variable

        # Standardize predictors
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # Define time series split
        tscv = TimeSeriesSplit(n_splits=5)

        # Lasso Regression with Time Series CV
        lasso_model = LassoCV(alphas=np.logspace(-4, 0, 50), cv=tscv, max_iter=10000
        lasso_model.fit(X_scaled, y)

        # Predict on entire dataset (fitted on time series CV)
        y_pred = lasso_model.predict(X_scaled)

        # Calculate MSPE on manual splits
        rmse_list = []
        for train_index, test_index in tscv.split(X_scaled):
            X_train, X_test = X_scaled[train_index], X_scaled[test_index]
            y_train, y_test = y.iloc[train_index], y.iloc[test_index]

            # Use pre-fitted LassoCV model to predict
            y_pred = lasso_model.predict(X_test)

            # Calculate MSPE for current fold
```

```
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Lasso Regression Average RMSE (Log Scale): {average_rmse}")
print(f"Lasso Regression Average RMSE (Original Scale): {np.exp(average_rmse
```

In [ ]:
```
# Extract and sort coefficients by magnitude
lasso_coefficients = pd.Series(np.abs(lasso_model.coef_), index=X.columns)
lasso_coefficients.sort_values(ascending=False, inplace=True)

# Plot top coefficients
lasso_coefficients.head(20).plot(kind='bar', figsize=(12, 6), title="Top Las
plt.xlabel('Features')
plt.ylabel('Absolute Coefficient Value')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Print all coefficients sorted by magnitude
print("Lasso Regression Coefficients by Magnitude for Log Close:")
print(lasso_coefficients)
```

In [ ]:
```
# Identify and print features removed by Lasso (coefficients = 0)
lasso_removed_features = lasso_coefficients[lasso_coefficients == 0]
print("\nFeatures Removed by Lasso (Coefficients = 0):")
print(lasso_removed_features)
```

In [ ]:
```
lasso_selected_features = lasso_coefficients[lasso_coefficients > 0]
print("\nFeatures kept by Lasso:")
print(lasso_selected_features)
```

Normal Close Values

In [ ]:
```
from sklearn.linear_model import LassoCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Prepare data
X = numeric_data  # Predictors
y = train_data['close']  # Response variable

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define time series split
tscv = TimeSeriesSplit(n_splits=5)

# Lasso Regression with Time Series CV
```

```
lasso_model = LassoCV(alphas=np.logspace(-4, 0, 50), cv=tscv, max_iter=10000
lasso_model.fit(X_scaled, y)

# Predict on entire dataset (fitted on time series CV)
y_pred = lasso_model.predict(X_scaled)

# Calculate MSPE on manual splits
rmse_list = []
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Use pre-fitted LassoCV model to predict
    y_pred = lasso_model.predict(X_test)

    # Calculate MSPE for current fold
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Lasso Regression Average RMSE for Close: {average_rmse}")
```

In [ ]:
```
# Extract and sort coefficients by magnitude
lasso_coefficients = pd.Series(np.abs(lasso_model.coef_), index=X.columns)
lasso_coefficients.sort_values(ascending=False, inplace=True)

# Plot top coefficients
lasso_coefficients.head(20).plot(kind='bar', figsize=(12, 6), title="Top Las
plt.xlabel('Features')
plt.ylabel('Absolute Coefficient Value')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Print all coefficients sorted by magnitude
print("Lasso Regression Coefficients by Magnitude for Close:")
print(lasso_coefficients)
```

In [ ]:
```
# Identify and print features removed by Lasso (coefficients = 0)
lasso_removed_features = lasso_coefficients[lasso_coefficients == 0]
print("\nFeatures Removed by Lasso (Coefficients = 0):")
print(lasso_removed_features)
```

In [ ]:
```
lasso_selected_features = lasso_coefficients[lasso_coefficients > 0]
print("\nFeatures kept by Lasso:")
print(lasso_selected_features)
```

SVMs

Linear SVM for feature importance

Log Transformed Respnse

In [ ]:
```
from sklearn.svm import LinearSVR
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Prepare data
X = numeric_data  # Predictors
y = train_data['log_close']  # Response variable

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define time series split
tscv = TimeSeriesSplit(n_splits=5)

# Train a Linear SVM regression model on the full dataset for feature import
svr_model = LinearSVR(random_state=42, max_iter=10000, C=0.1)  # Use smaller
svr_model.fit(X_scaled, y)

# Extract and sort coefficients by magnitude
svr_feature_importance = pd.Series(np.abs(svr_model.coef_), index=X.columns)
svr_feature_importance.sort_values(ascending=False, inplace=True)

# Plot top coefficients
svr_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Top
plt.xlabel('Features')
plt.ylabel('Absolute Coefficient Value')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Initialize list to store MSPE for each split
rmse_list = []

# Perform time series split and train/test evaluation for MSPE
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a Linear SVM regression model on the current split
    svr_model_split = LinearSVR(random_state=42, max_iter=10000, C=0.1)
    svr_model_split.fit(X_train, y_train)

    # Predict on the test set
    y_pred = svr_model_split.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
```

```
print(f"Linear SVM Average RMSE (Log Scale): {average_rmse}")
print(f"Linear SVM Average RMSE (Original Scale): {np.exp(average_rmse)}")
```

In [ ]:
```
# Print all coefficients sorted by magnitude
print("Linear SVM Coefficients by Magnitude for Log Close:")
print(svr_feature_importance)
```

Normal Close Values

In [ ]:
```
from sklearn.svm import LinearSVR
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Prepare data
X = numeric_data  # Predictors
y = train_data['close']  # Response variable

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define time series split
tscv = TimeSeriesSplit(n_splits=5)

# Train a Linear SVM regression model on the full dataset for feature import
svr_model = LinearSVR(random_state=42, max_iter=10000, C=0.1)  # Use smaller
svr_model.fit(X_scaled, y)

# Extract and sort coefficients by magnitude
svr_feature_importance = pd.Series(np.abs(svr_model.coef_), index=X.columns)
svr_feature_importance.sort_values(ascending=False, inplace=True)

# Plot top coefficients
svr_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="Top
plt.xlabel('Features')
plt.ylabel('Absolute Coefficient Value')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Initialize list to store MSPE for each split
rmse_list = []

# Perform time series split and train/test evaluation for MSPE
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a Linear SVM regression model on the current split
    svr_model_split = LinearSVR(random_state=42, max_iter=10000, C=0.1)
    svr_model_split.fit(X_train, y_train)
    # Predict on the test set
    y_pred = svr_model_split.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Linear SVM Average RMSE for Close: {average_rmse}")
```

In [ ]:
```
# Print all coefficients sorted by magnitude
print("Linear SVM Coefficients by Magnitude:")
print(svr_feature_importance)
```

Non-Linear SVM for feature importance

Log Transformed Response

In [ ]:
```
from sklearn.svm import SVR
from sklearn.inspection import permutation_importance
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(numeric_data)
y = train_data['log_close']  # Response variable

# Define time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store MSPE for each split
rmse_list = []

# Perform time series split and train/test evaluation
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a non-linear SVM model with RBF kernel
    svr_rbf_model = SVR(kernel='rbf', C=0.1, gamma='scale', epsilon=0.1, max
    svr_rbf_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = svr_rbf_model.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Non-linear SVM Average RMSE (Log Scale): {average_rmse}")
print(f"Non-linear SVM Average RMSE (Original Scale): {np.exp(average_rmse)}
```

In [ ]:
```
# Train the model on the full dataset for feature importance
svr_rbf_model.fit(X_scaled, y)

# Compute permutation importance
perm_importance = permutation_importance(svr_rbf_model, X_scaled, y, n_repea

# Extract feature importance
perm_feature_importance = pd.Series(perm_importance.importances_mean, index=
perm_feature_importance.sort_values(ascending=False, inplace=True)

# Plot top positive feature importances
perm_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="To
plt.xlabel('Features')
plt.ylabel('Permutation Importance Score')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Print all feature importances sorted by score
print("Non-Linear SVM (RBF Kernel) Feature Importance:")
print(perm_feature_importance)
```

In [ ]:
```
# List features with negative or zero importance
negative_features = perm_feature_importance[perm_feature_importance <= 0]
print("\nFeatures with Negative Importance for Log Close:")
print(negative_features)
```

Normal Close Values

In [ ]:
```
from sklearn.svm import SVR
from sklearn.inspection import permutation_importance
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(numeric_data)
y = train_data['close']  # Response variable

# Define time series split
tscv = TimeSeriesSplit(n_splits=5)

# Initialize list to store MSPE for each split
rmse_list = []

# Perform time series split and train/test evaluation
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train a non-linear SVM model with RBF kernel
    svr_rbf_model = SVR(kernel='rbf', C=0.1, gamma='scale', epsilon=0.1, max
    svr_rbf_model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = svr_rbf_model.predict(X_test)

    # Calculate MSPE
    rmse = mean_squared_error(y_test, y_pred, squared=False)
    rmse_list.append(rmse)
```

In [ ]:
```
# Average MSPE across all splits
average_rmse = sum(rmse_list) / len(rmse_list)
print(f"Non-Linear SVM (RBF Kernel) Average RMSE for Close: {average_rmse}")
```

In [ ]:
```
# Train the model on the full dataset for feature importance
svr_rbf_model.fit(X_scaled, y)

# Compute permutation importance
perm_importance = permutation_importance(svr_rbf_model, X_scaled, y, n_repea

# Extract feature importance
perm_feature_importance = pd.Series(perm_importance.importances_mean, index=
perm_feature_importance.sort_values(ascending=False, inplace=True)

# Plot top positive feature importances
perm_feature_importance.head(20).plot(kind='bar', figsize=(12, 6), title="To
plt.xlabel('Features')
plt.ylabel('Permutation Importance Score')
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Print all feature importances sorted by score
print("Non-Linear SVM (RBF Kernel) Feature Importance for Close:")
print(perm_feature_importance)
```

In [ ]:
```
# List features with negative or zero importance
negative_features = perm_feature_importance[perm_feature_importance <= 0]
print("\nFeatures with Negative Importance for Close:")
print(negative_features)
```

Clustering Methods

KMeans Clustering

In [ ]:
```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import pandas as pd
import matplotlib.pyplot as plt

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(numeric_data)

# Perform K-Means clustering
kmeans = KMeans(n_clusters=5, random_state=42, algorithm="elkan")
kmeans_clusters = kmeans.fit_predict(X_scaled)

# Ensure 'Date' is in datetime format
train_data1 = train_data.copy()  # Create a copy to avoid SettingWithCopyWar
train_data1.loc[:, 'KMeans_Cluster'] = kmeans_clusters
train_data1.loc[:, 'Date'] = pd.to_datetime(train_data['Date'])  # Ensure Da

# Summarize response ('close') by cluster
kmeans_summary = train_data1.groupby('KMeans_Cluster').mean()
print("K-Means Cluster Summary (Average Close Price):")
print(kmeans_summary)
```

In [ ]:
```
# Visualize cluster distribution over time
plt.figure(figsize=(12, 6))
for cluster in sorted(train_data1['KMeans_Cluster'].unique()):
    cluster_dates = train_data1[train_data1['KMeans_Cluster'] == cluster]['D
    plt.scatter(cluster_dates, [cluster] * len(cluster_dates), label=f"Clust
plt.title("K-Means Clusters Over Time")
plt.xlabel("Date")
plt.ylabel("Cluster")
plt.legend(title="Clusters")
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
for feature in numeric_data.columns:
    train_data1.boxplot(column=feature, by='KMeans_Cluster', grid=False)
    plt.title(f'{feature} by K-Means Cluster')
    plt.xlabel("Cluster")
    plt.ylabel(feature)
    plt.show()
```

Hierarchical Clustering

In [ ]:
```
from scipy.cluster.hierarchy import linkage, fcluster
import pandas as pd


# Perform hierarchical clustering
linked = linkage(X_scaled, method='ward')  # Perform clustering

# Extract flat clusters (e.g., 5 clusters)
hierarchical_clusters = fcluster(linked, t=5, criterion='maxclust')

# Add cluster labels to the dataset
train_data2 = train_data.copy()
train_data2.loc[:, 'Hierarchical_Cluster'] = hierarchical_clusters

# Summarize predictors and response ('close') by cluster
hierarchical_summary = train_data2.groupby('Hierarchical_Cluster').mean()
print("Hierarchical Cluster Summary:")
print(hierarchical_summary)
```

Model Based Clustering

In [ ]:
```
from sklearn.mixture import GaussianMixture
import pandas as pd
import matplotlib.pyplot as plt

# Perform Gaussian Mixture Clustering
gmm = GaussianMixture(n_components=5, random_state=42)
gmm_clusters = gmm.fit_predict(X_scaled)

# Add GMM cluster labels to the dataset
train_data3 = train_data.copy()
train_data3.loc[:, 'GMM_Cluster'] = gmm_clusters
```

```
# Visualize GMM clusters over time
plt.figure(figsize=(12, 6))
for cluster in sorted(train_data3['GMM_Cluster'].unique()):
    cluster_dates = train_data3[train_data3['GMM_Cluster'] == cluster]['Date
    plt.scatter(cluster_dates, [cluster] * len(cluster_dates), label=f"Clust
plt.title("GMM Clusters Over Time")
plt.xlabel("Date")
plt.ylabel("Cluster")
plt.legend(title="Clusters")
plt.grid(axis='y')
plt.show()
```

In [ ]:
```
# Summarize predictors and response ('close') by cluster
gmm_summary = train_data3.groupby('GMM_Cluster').mean()
print("Gaussian Mixture Model Cluster Summary:")
print(gmm_summary)
```

In [ ]:
```
from sklearn.metrics import mean_squared_error
import numpy as np
import pandas as pd

# Assume `train_data` has the following cluster labels already added:
# 'KMeans_Cluster', 'Hierarchical_Cluster', 'GMM_Cluster'

# Prepare response variable
y = train_data['log_close']

# Initialize a dictionary to store MSPE for each clustering method
rmse_dict = {}

# Evaluate MSPE for K-Means Clustering
kmeans_means = train_data1.groupby('KMeans_Cluster')['log_close'].transform(
kmeans_rmse = mean_squared_error(y, kmeans_means, squared=False)
rmse_dict['KMeans'] = kmeans_rmse

# Evaluate MSPE for Hierarchical Clustering
hierarchical_means = train_data2.groupby('Hierarchical_Cluster')['log_close'
hierarchical_rmse = mean_squared_error(y, hierarchical_means, squared=False)
rmse_dict['Hierarchical'] = hierarchical_rmse

# Evaluate MSPE for Model-Based Clustering
gmm_means = train_data3.groupby('GMM_Cluster')['log_close'].transform('mean'
gmm_rmse = mean_squared_error(y, gmm_means, squared=False)
rmse_dict['Model-Based'] = gmm_rmse
```

In [ ]:
```
# Display MSPE for all clustering methods
print("Clustering RMSEs:")
for method, rmse in rmse_dict.items():
    print(f"{method} Clustering RMSE for Log Close (Log Scale): {rmse}")
    print(f"{method} Clustering RMSE for Log Close (Original Scale): {np.exp
    print(f"\n")
```

In [ ]:
```
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegresso
from sklearn.svm import LinearSVR
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import numpy as np
import pandas as pd


train_data.loc[:, 'KMeans_Cluster'] = kmeans_clusters
train_data.loc[:, 'Hierarchical_Cluster'] = hierarchical_clusters
train_data.loc[:, 'GMM_Cluster'] = gmm_clusters
# Prepare data
X = train_data[["low", "vpt", "high", "EP", "gold", "PSAR", "sma_50-sma_200"
                "treasury_5_years", "total_share_holder_equity", "open", "bo
                "shares_outstanding", "price_to_book_ratio", "ttm_sales_per_
                "current_liabilities", "price_to_sales_ratio", "current_rati
                "return_on_equity", "cash_on_hand", "operating_income", "eps
                "ebitda", "return_on_tangible_equity", "decreasing", "increa
                "macd_signal", "stoch", "MACD_10_26_9", "roc", "treasury_30_
                "KMeans_Cluster"]]

y = train_data["log_close"]

# Standardize features (for models requiring scaling)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Time series cross-validation
tscv = TimeSeriesSplit(n_splits=5)

# Initialize MSPE dictionary
rmse_results = {}

# Linear Regression
linear_rmse = []
linear_model = LinearRegression()
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    linear_model.fit(X_train, y_train)
    y_pred = linear_model.predict(X_test)
    linear_rmse.append(mean_squared_error(y_test, y_pred, squared=False))
rmse_results["Linear Regression"] = np.exp(np.mean(linear_rmse))

# Ridge Regression
ridge_model = RidgeCV(alphas=[0.1, 1.0, 10.0])
ridge_rmse = []
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    ridge_model.fit(X_train, y_train)
    y_pred = ridge_model.predict(X_test)
    ridge_rmse.append(mean_squared_error(y_test, y_pred, squared=False))
rmse_results["Ridge"] = np.exp(np.mean(ridge_rmse))

# Lasso Regression
```

```python
lasso_model = LassoCV(alphas=np.logspace(-4, 0, 50), max_iter=10000)
lasso_rmse = []
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    lasso_model.fit(X_train, y_train)
    y_pred = lasso_model.predict(X_test)
    lasso_rmse.append(mean_squared_error(y_test, y_pred, squared=False))
rmse_results["Lasso"] = np.exp(np.mean(lasso_rmse))

# Random Forest
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_rmse = []
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    rf_model.fit(X_train, y_train)
    y_pred = rf_model.predict(X_test)
    rf_rmse.append(mean_squared_error(y_test, y_pred, squared=False))
rmse_results["Random Forest"] = np.exp(np.mean(rf_rmse))

# Gradient Boosting
gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)
gb_rmse = []
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    gb_model.fit(X_train, y_train)
    y_pred = gb_model.predict(X_test)
    gb_rmse.append(mean_squared_error(y_test, y_pred, squared=False))
rmse_results["Gradient Boosting"] = np.exp(np.mean(gb_rmse))

# Linear SVM
svr_model = LinearSVR(random_state=42, max_iter=10000, C=0.1)
svr_rmse = []
for train_index, test_index in tscv.split(X_scaled):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    svr_model.fit(X_train, y_train)
    y_pred = svr_model.predict(X_test)
    svr_rmse.append(mean_squared_error(y_test, y_pred, squared=False))
rmse_results["Linear SVM"] = np.exp(np.mean(svr_rmse))

from sklearn.decomposition import PCA

# PCA Regression
pca_rmse = []
explained_variance_threshold = 0.95  # Set the threshold for explained varia

# Fit PCA on the entire dataset to determine the number of components for 95
pca_full = PCA().fit(X_scaled)
cumulative_explained_variance = np.cumsum(pca_full.explained_variance_ratio_
n_components = np.argmax(cumulative_explained_variance >= explained_variance

# Use the determined number of components for PCA
pca = PCA(n_components=n_components)
```

```
pca_model = LinearRegression()

for train_index, test_index in tscv.split(X_scaled):
    # Apply PCA on training and test sets
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Fit PCA on training data and transform both training and test data
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)

    # Train Linear Regression model on PCA-transformed data
    pca_model.fit(X_train_pca, y_train)
    y_pred = pca_model.predict(X_test_pca)

    # Calculate RMSE
    pca_rmse.append(mean_squared_error(y_test, y_pred, squared=False))

# Add PCA Regression RMSE to the results dictionary
rmse_results["PCA Regression"] = np.exp(np.mean(pca_rmse))
```

In [ ]:
```
# Output MSPE results
rmse_results
```