In [287…

```
# Install and Import Necessary Libraries and Packages

import pandas as pd
import pandas_datareader as pdr
import pandas_datareader.data as web
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
import ta as ta
import requests
import os
from bs4 import BeautifulSoup
import statsmodels.api as sm
from sklearn.linear_model import LinearRegression
from scipy import stats
import rpy2.robjects as ro
from rpy2.robjects import pandas2ri
from rpy2.robjects import conversion
from scipy.stats import skew, kurtosis
pandas2ri.activate()
```

Before diving into the core research project, it's valuable to take a comprehensive look at the ta package. This package offers a wide array of built-in technical indicators, which can serve as powerful tools for developing and testing trading strategies.

However, understanding the mathematical foundations and the various parameters available within the ta package is crucial to tailoring these indicators effectively to fit specific research objectives. While ta offers a robust set of tools, there will undoubtedly be cases where further customization is required to meet more complex research needs. This is where creating custom technical indicators based on specific financial criteria becomes essential.

By combining the built-in tools with personalized adjustments, it becomes possible to address nuanced research questions and trading strategies that go beyond standard implementations. Nevertheless, exploring and experimenting with different strategies using the capabilities of ta is not only feasible but also provides a solid foundation for generating insights and enhancing decision-making.

Quantitative Research Project Plan

Step-by-Step Process

0: Problem Selection, Data Extraction, Scraping, and Cleaning (Completed)

- **Objective**: Obtain comprehensive data for any ticker in the S&P 500 over any date and time range, including OHLCV (Open, High, Low, Close, Volume) data and all indicators from the `ta` package.

- **Data Quality**: Ensured consistency in data cleaning, handled missing values, and managed outliers effectively.
- **Data Sources**: Used reliable data sources (e.g., Yahoo Finance API) for the necessary data frequency (daily, hourly, etc.).

1: Exploratory Data Analysis (EDA) (Completed)

- **Statistical Analysis**:
  - Computed **correlations**, **skewness**, and **kurtosis** among variables.
  - Checked for **multicollinearity** using **Variance Inflation Factor (VIF)**.
- **Visualizations**:
  - Generated **scatterplots**, **histograms**, and **heatmaps** to visualize data distributions and relationships.
- **Feature Distributions**:
  - Assessed normality of feature distributions and applied transformations (e.g., **Box-Cox**, **log transformations**) as needed.

2: Predictor Selection (Completed)

- **Feature Selection Techniques**:
  - Used **VIF** to eliminate highly collinear predictors.
  - Applied **ElasticNet regularization** for automatic variable selection and regularization.

3: Time Series Analysis (TSA) with ARIMA (Completed)

- **Objective**: Modeled and forecasted the time-dependent structure using ARIMA.
- **Stationarity**:
  - Tested for stationarity using the **ADF test** and applied differencing.
- **Model Estimation**:
  - Fitted ARIMA and **ARIMAX** models to capture both autoregressive and moving average components.
- **Model Selection**:
  - Used **AIC/BIC** to compare models and selected the best-performing ones.

4: Advanced TSA: Volatility Modeling with GARCH/EGARCH (Completed)

- **Objective**: Modeled volatility clustering and heteroscedasticity using **GARCH** and **EGARCH** models.

- **Model Estimation**:
  - Fitted **GARCH(1,1)** and **EGARCH(1,1)** models with both **Normal** and **Student-t** distributions.
  - **EGARCH(1,1) with Student-t distribution** was selected as the best model based on **AIC/BIC** and residual diagnostics.

5: Integration of EGARCH Model into Regression Framework (Completed)

- **Objective**: Enhanced robust and regularized regression models by incorporating volatility estimates from the EGARCH model.
- **EGARCH-Enhanced Regression Models**:
  - Integrated **EGARCH volatility** as an additional predictor in robust methods (Huber, Quantile, LTS) and regularized regression models (Ridge, Lasso, ElasticNet).
- **Results**:
  - Some models, such as **Ridge** and **LTS**, showed modest improvements, while others, like **ElasticNet** and **Lasso**, worsened after EGARCH integration.

6: Regularized Regression Models (Completed)

- **Objective**: Predicted the target variable using regularized regression methods.
- **Models Fit**:
  - **Ridge**, **Lasso**, and **ElasticNet** models were applied.
- **Evaluation**:
  - **MSPE** was used to evaluate the predictive performance. The **Full Model** performed best among the regularized approaches, although adding EGARCH did not lead to significant improvements.

Future Methodologies and Additions

Residual Diagnostics and Error Distribution Analysis (In Progress)

- **Objective**: Conduct residual diagnostics and assess model fit.
- **Error Assumptions**:
  - Compare residual distributions using **Q-Q plots** and perform diagnostic tests like **Ljung-Box** and **ARCH** tests to better understand the behavior of residuals post-EGARCH integration.

Non-Parametric and Machine Learning Methods (Planned)

- **Objective**: Explore non-linear relationships and interactions using **machine learning algorithms**.
- **Methods to Apply**:
  - **Random Forests** and **Gradient Boosting Machines (GBMs)** to identify potential non-linear patterns that EGARCH and traditional regressions missed.

Ensemble Methods for Model Combination (Planned)

- **Objective**: Combine models to improve overall predictive performance.
- **Techniques**:
  - **Stacking** and **blending** models, allowing for improved accuracy by combining insights from both traditional regression and machine learning models.

Model Validation and Backtesting (Planned)

- **Objective**: Ensure model robustness through **backtesting** and **cross-validation**.
- **Techniques**:
  - Use **rolling window validation** and **walk-forward validation** to assess the model's predictive capacity over time and across different market regimes.
  - Evaluate the model's performance using financial metrics like **Sharpe Ratio** and **Value at Risk (VaR)**.

Advanced Optimization and Simulation Techniques (Planned)

- **Objective**: Enhance parameter estimation and evaluate model uncertainty using advanced methods.
- **Optimization Techniques**:
  - Implement **Simulated Annealing**, **Genetic Algorithms**, or **Particle Swarm Optimization** to improve model fitting.
- **Monte Carlo Simulations**:
  - Use simulations to stress-test models under various market conditions and assess their risk profiles.

Distribution Fitting and Simulation (Planned)

- **Objective**: Fit and simulate stock price distributions using advanced techniques.
- **Methods**:
  - Apply **Inverse CDF**, **Acceptance-Rejection methods**, and **Kernel Density Estimation** for distribution fitting, to better model real-world stock price movements.

Completed Methodologies

- **Data Extraction, Cleaning, and Transformation**
- **Exploratory Data Analysis (EDA)**
- **Variable Transformations and Selection**
- **Time Series Analysis (ARIMA)**
- **Advanced TSA with GARCH/EGARCH**
- **EGARCH Integration with Robust and Regularized Regression Models**
- **Residual Diagnostics and Error Analysis**

Remaining Objectives

- **Non-Parametric Methods** (Random Forests, GBM)
- **Model Combination and Ensemble Methods**
- **Backtesting Framework and Risk Management Metrics**
- **Advanced Optimization Techniques** (Simulated Annealing, MCMC)
- **Monte Carlo Simulations for Risk Assessment**
- **Distribution Fitting and Simulation Methods**

Practical Implications and Moving Forward

- The results from integrating EGARCH models into regularized and robust regression frameworks did not yield the expected improvements. In particular, **ElasticNet** and **Lasso** models worsened after incorporating volatility predictions, while **Ridge** and **LTS** saw only modest gains.
- This implies that while volatility modeling is essential, its straightforward inclusion as a predictor may not be the most effective way to improve regression models' performance.

In Progress:

- I plan to explore **non-parametric methods** like Random Forests and GBM to capture non-linear relationships that EGARCH and linear models may have missed.
- **Ensemble methods** will also be tested to combine different models for better predictive accuracy.

The journey so far has been very insightful, revealing that traditional regression and volatility modeling may not always yield the best results in complex financial datasets. However, the lessons learned from this exploration will guide my next steps toward building a more robust quantitative strategy.

In [279…
```
# Understanding ta library more comprehensively
# Uncomment code to run
# print(dir(ta))
```

In [279…
```
# Determine submodules of interest
# Uncomment code to run
# for submodule in dir(ta):
    # if not submodule.startswith("__"):
       #  print(submodule)
```

In [279…
```
# List of Trend-based Technical Indicators and how to customize arguments and p
# Remove # to learn more
# help(ta.trend)

# List of Volatility-based Technical Indicators and how to customize arguments
# Remove # to learn more
# help(ta.volatility)

# List of Momentum Based Technical Indicators and how to customize arguments ar
# Remove # to learn more
# help(ta.momentum)

# Extensive list of Volume-based Technical Indicators and how to customize argu
# Remove # to learn more
# help(ta.volume)

# Popular Technical Indicators bundled together
# Remove # to learn more
# help(ta.wrapper)

# Remove # to learn more
# help(ta.add_others_ta)

# Remove # to learn more
# help(ta.add_trend_ta)

# Remove # to learn more
# help(ta.add_volatility_ta)

# Remove # to learn more
# help(ta.add_volume_ta)

# Remove # to learn more
# help(ta.others)

# Remove # to learn more
# help(ta.utils)
```

In [280…
```
# In depth analysis of ta package and submodules is done
```

In [280…
```
# Initially accessed list of tickers for S&P 500 stocks using slickcharts
# Used headers to simulate a request from a web browser
# Create a function that grabs all S&P 500 Tickers
# Creating this function in anticipation of using it to extract data easily fr
```

In [280…
```
def sp_500_tickers():
```

```
    # Grab url
    url = 'https://www.slickcharts.com/sp500'
    # Simulate a request from a web browser
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/5
    }
    #Error Handling
    try:
        response = requests.get(url, headers=headers)
        response.raise_for_status()  # Ensure the request was successful
    except requests.exceptions.HTTPError as http_err:
        print("HTTP Error has occurred")
        return []
    except Exception as err:
        print("Error has occurred")
        return []

    # Parse the HTML content of the webpage
    html_parser = BeautifulSoup(response.text, 'html.parser')
    # Scrape html file using inspect and determine how to extract table
    # Table was labeled 'table table-hover table-borderless table-sm'
    table = html_parser.find('table', {'class': 'table table-hover table-borde

    if table is None:
        print("Could not find the table on the webpage.")
        return []
    tickers = []
    ticker_column_index = 2  # The index of the column containing the ticker sy

    # Go through all table rows
    for row in table.find_all('tr')[1:]:  # Skip the header row
        cells = row.find_all('td') # Extract the columns of each row
        if len(cells) > ticker_column_index:
            ticker = cells[ticker_column_index].text.strip()
            tickers.append(ticker)

    return tickers
```

In [ ]:
```
tickers = sp_500_tickers()
print(f"Number of tickers fetched: {len(tickers)}")
```

In [280…
```
# Remove # to learn more
# help(yf.Ticker)
```

In [280…
```
def validate_tickers(tickers): # Function to validate the tickers by fetching r
    valid_tickers = []

    for ticker in tickers:
# Ran into problem here
# Determined the root cause was the syntax carried over from Wikipedia was not
# Made sure that data has been read in wikipedia and yfinance together and data
        yf_ticker = ticker.replace('.', '-')
        stock = yf.Ticker(yf_ticker)
        hist = stock.history(period="1m")
        if not hist.empty:
            valid_tickers.append(yf_ticker)  # Append the ticker, not the data

    return valid_tickers
```

In [ ]:
```
# Validate the tickers
valid_tickers = validate_tickers(tickers)
print(f"Number of valid tickers: {len(valid_tickers)}")
```

In [280…
```
def fetch_historical_data(valid_tickers, start_date='2009-01-01', end_date='20
    all_data = {}  # Initialize all_data dictionary

    for ticker in valid_tickers:
        stock = yf.Ticker(ticker)

        try:
            # Fetch historical market data for the specified period
            hist = stock.history(start=start_date, end=end_date)
            if not hist.empty:
                all_data[ticker] = hist
            else:
                print(f"No data found for {ticker}.")

        except Exception as e:
            print(f"Error fetching historical data for {ticker}: {e}")

    return all_data
```

In [ ]:
```
# Fetch full historical data for the validated tickers
historical_data = fetch_historical_data(valid_tickers)
print(f"Number of tickers with historical data: {len(historical_data)}")
```

In [ ]:
```
# Select several stocks for visualization to make sure data has been extracted

selected_tickers = ['AAPL', 'MSFT', 'GOOGL']  # Selected stock tickers for visu
plt.figure(figsize=(14, 8))

for ticker in selected_tickers:
    if ticker in historical_data:
        plt.plot(historical_data[ticker].index, historical_data[ticker]['Close

plt.title('Historical Close Prices of Selected Stocks from Slick')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.grid(True)
plt.show()
```

In [ ]:
```
def sp_500_tickers_wiki(): # Function to scrape S&P 500 tickers from Wikipedia
    url = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies' # Extrac
    response = requests.get(url)
    response.raise_for_status()

    # Parse the HTML content of the webpage
    html_parser = BeautifulSoup(response.text, 'html.parser')

    # Scrape HTML file using inspect and determine how to extract table
    table = html_parser.find('table', {'id': 'constituents'})

    # Verify if the table was found
    if table is None:
        print("Could not find the table on the webpage.")
        return []
```

```
    # List to store the extracted ticker symbols
    tickers = []
    ticker_column_index = 0  # The index of the column containing the ticker sy

    # Go through all table rows
    for row in table.find_all('tr')[1:]:  # Skip the header row
        cells = row.find_all('td')  # Extract the columns of each row
        if len(cells) > ticker_column_index:
            ticker = cells[ticker_column_index].text.strip()
            tickers.append(ticker)

    return tickers

# Example usage, and testing to make sure the process works
tickers_wiki = sp_500_tickers_wiki()

print(f"Number of tickers fetched: {len(tickers)}")

# Function to validate the tickers by fetching minimal data using yfinance
def validate_tickers_wiki(tickers_wiki):
    valid_tickers = []

    for ticker in tickers_wiki:
        yf_ticker = ticker.replace('.', '-')
        stock = yf.Ticker(yf_ticker)
        hist = stock.history(period="1m")
        if not hist.empty:
            valid_tickers.append(yf_ticker)  # Append the ticker, not the data

    return valid_tickers

# Validate the tickers
valid_tickers_wiki = validate_tickers_wiki(tickers_wiki)
print(f"Number of valid tickers: {len(valid_tickers_wiki)}")

# Function to fetch historical data for validated tickers using yfinance
def fetch_historical_data_wiki(valid_tickers_wiki, start_date='2009-01-01', en
    all_data = {}  # Initialize all_data dictionary

    for ticker in valid_tickers_wiki:
        stock = yf.Ticker(ticker)

        try:
            # Fetch historical market data for the specified period
            hist = stock.history(start=start_date, end=end_date)
            if not hist.empty:
                all_data[ticker] = hist
            else:
                print(f"No data found for {ticker}.")

        except Exception as e:
            print(f"Error fetching historical data for {ticker}: {e}")

    return all_data

# Fetch full historical data for the validated tickers
historical_data_wiki = fetch_historical_data_wiki(valid_tickers_wiki)
print(f"Number of tickers with historical data: {len(historical_data_wiki)}")
```

```
# Select several stocks for visualization to make sure data has been extracted
selected_tickers_wiki = ['AAPL', 'MSFT', 'GOOGL']  # Selected stock tickers fo
plt.figure(figsize=(14, 8))

for ticker in selected_tickers_wiki:
    if ticker in historical_data_wiki:
        plt.plot(historical_data_wiki[ticker].index, historical_data_wiki[tick

plt.title('Historical Close Prices of Selected Stocks from Wiki')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.grid(True)
plt.show()
```

In [281…

```
def apply_all_indicators(data):
    # Ensure necessary columns are present
    if 'Open' in data.columns and 'High' in data.columns and 'Low' in data.colu
        # Add all technical indicators using ta
        data = ta.add_all_ta_features(
            data, open="Open", high="High", low="Low", close="Close", volume="
    else:
        print("Required columns are missing from the data.")
    return data
```

In [281…

```
def fetch_and_enhance_data(tickers, start_date, end_date):
    # Initialize dictionary to store the historical data
    all_data = {}

    # Fetch data for each ticker
    for ticker in tickers:
        try:
            stock = yf.Ticker(ticker)
            historical_data = stock.history(start=start_date, end=end_date)
            if not historical_data.empty:
                # Apply all technical indicators
                all_data[ticker] = apply_all_indicators(historical_data)
        except Exception as e:
            print(f"Error processing {ticker}: {e}")

    return all_data
```

In [ ]:

```
# Define your tickers and time periods
tickers = ['AAPL', 'MSFT', 'GOOGL']  # Example subset for testing
start_date = '2010-01-01'
end_date = '2023-12-29'

# Fetch and enhance data
enhanced_data = fetch_and_enhance_data(tickers, start_date, end_date)

# Looking at documentation, others_ta indicator others_cr is calculated only us
# This could potentially ruin the model because of perfect multicollinearity.
enhanced_data['AAPL'] = enhanced_data['AAPL'].drop(columns=['others_cr'])
# Now you can inspect or save the enhanced data
if 'AAPL' in enhanced_data:
    print(enhanced_data['AAPL'].head())  # Display some of the data
```

In [ ]:

```
print(enhanced_data['AAPL'].columns)
```

Send Data to R:

ro.globalenv['variable_name_in_r'] = pandas2ri.py2rpy(pandas_df_in_python)

Run R Code:

ro.r(''' R code here ''')

Retrieve Data from R:

python_df = pandas2ri.rpy2py(ro.globalenv['variable_name_in_r'])

In [ ]:
```
# Grab the specific stock of interest
aapl_data = enhanced_data['AAPL']
# Create a Date column
aapl_data = aapl_data.reset_index()

# Check for any missing or null values
print(aapl_data.isnull().sum())
# Convert 'volume_obv' to a larger data type, like float64 because computer ca
aapl_data['volume_obv'] = aapl_data['volume_obv'].astype(np.float64)

# Now convert to R
aapl_r_df = pandas2ri.py2rpy(aapl_data)

# Assign df to the R environment for further use
ro.globalenv['aapl_r_df'] = aapl_r_df

# Set up R for complex statistical analysis
ro.r('''
    library(randomForest)  # For nonlinear relationships
    library(MASS)          # For LTS, stepAIC
    library(car)           # Companion to Applied Regression
    library(forecast)      # Time Series forecasting
    library(tseries)
    library(lmtest)        # For regression diagnostics
    library(ggplot2)       # For visualization
    library(data.table)    # For fast data manipulation
    library(dplyr)         # For data manipulation
    library(boot)          # For bootstrap
    library(glmnet)        # For ridge/lasso regression
    library(quantreg)      # For quantile regression
    library(leaps)         # For model selection, regsubsets
    library(robustbase)    # For robust linear regression
    library(caret)         # For cross-validation and model training
    library(MCMCpack)      # For MCMC regression
    library(mgcv)          # For generalized additive model
''')
```

In [ ]:
```
# Ensure that we are operating on the desired data in R
ro.r('''
    summary <- summary(aapl_r_df)
    print(summary)
''')
```

In [ ]:
```
# The first thing we want to do, is split the data into training and test set
ro.r('''
     aapl_r_df$Date <- as.Date(aapl_r_df$Date)
     train_data <- aapl_r_df[aapl_r_df$Date < as.Date("2021-01-01"), ]
     test_data <- aapl_r_df[aapl_r_df$Date >= as.Date("2021-01-01"), ]
     # Show the first few rows of train dataset
     print(head(train_data))
     ''')
```

In [ ]:
```
ro.r(''' # Show first few rows of test dataset
     print(head(test_data))
     ''')
```

In [ ]:
```
ro.r(''' # Check for missing values
    sum(is.na(train_data))
''')
```

In [ ]:
```
ro.r(''' # Convert to data.table
    library(data.table)
    setDT(train_data)  # Convert the train data to data.table
    setDT(test_data)   # Convert the test data to data.table
    class(train_data)  # Check to make sure conversion was done

''')
```

In [ ]:
```
ro.r('''
    class(test_data)
''')
```

In [ ]:
```
ro.r('''
    # Split the data into y ~ response and X ~ design matrix (predictors)

    # Response
    response <- train_data[,.(Close)]
    print(head(response))

    # Design Matrix
    predictors <- train_data[, !"Close", with = FALSE]
    print(head(predictors))

    # For correlation and histogram creation
    predictors <- as.data.frame(predictors)


    ''')
```

In [ ]:
```
ro.r(''' # Data Cleaning
    # Create response variable
    response <- train_data[,.(Close)]

    # Create design matrix (predictors)
    predictors <- train_data[,!'Close', with = FALSE]
    predictors <- as.data.frame(predictors)

    response_df <- as.data.frame(response)
    response_numeric <- as.numeric(train_data$Close)
```

```
    # Extract numeric columns from predictors
    numeric_predictors <- predictors[, sapply(predictors, is.numeric)]

    # Verify to make sure non-numeric columns (like Date) have been removed
    print(head(numeric_predictors))
     ''')
```

In [ ]:
```
# Step 1: Perform the correlation-based filtering in R
ro.r('''
    # Initialize a vector to store relevant variable names
    relevant_predictors <- c()

    # Loop through each predictor (column), calculate its correlation to the re
    for (colname in colnames(numeric_predictors)) {
        # Correlation to response (Close)
        correlation <- cor(numeric_predictors[[colname]], response$Close, use =

        # If the absolute correlation meets the threshold, print and store
        if (abs(correlation) > 0.2) {
            cat("Analyzing:", colname, " ")
            cat("Correlation with Close:", correlation, "\n")
            # Append the relevant predictors to the vector
            relevant_predictors <- c(relevant_predictors, colname)
        }
    }

    # Extract only the relevant predictors from numeric_predictors
    corr_based_filtered_predictors <- numeric_predictors[, relevant_predictors
    print(head(corr_based_filtered_predictors))
''')
```

In [ ]:
```
ro.r(''' # Data parsing check
       # Expecting 93
       print(ncol(train_data))
       # Expecting 92
       print(ncol(predictors))
       # Expecting 91
       print(ncol(numeric_predictors))
       # Expecting 42
       print(ncol(corr_based_filtered_predictors))
       ''')
```

In [ ]:
```
ro.r(''' # Number of missing values
       sum(is.na(corr_based_filtered_predictors))
       ''')
```

In [ ]:
```
import rpy2.robjects as ro # Step 2: Generate histograms based on filtered pre
from rpy2.robjects.lib import grdevices
from IPython.display import Image, display


def display_histograms():
    # Fetch column names from the filtered predictors
    colnames = list(ro.r('colnames(corr_based_filtered_predictors)'))

    # Add the 'Close' column separately as response
    colnames.append("Close")
```

```
    for colname in colnames:
        colname_str = str(colname)  # Convert to Python string
        filename = f"histogram_{colname_str}.png"  # Save each histogram as a

        # Determine which dataset to use (filtered predictors or response)
        if colname_str == 'Close':
            # For 'Close', use the numeric vector
            dataset = "response_numeric"
            ro.r(f'''
            # Open a PNG plotting device in R
            png(file="{filename}", width=512, height=512)
            # Generate the histogram using R
            hist({dataset}, probability = TRUE,
                 main="Histogram of Close",
                 xlab="Close",
                 col="lightblue",
                 border="black")
            # Add a normal distribution curve
            curve(dnorm(x, mean=mean({dataset}, na.rm=TRUE),
                 sd=sd({dataset}, na.rm=TRUE)),
                 col="red", lwd=2, add=TRUE)
            dev.off()
            ''')
        else:
            # For the predictors, use corr_based_filtered_predictors dataset
            dataset = "corr_based_filtered_predictors"
            ro.r(f'''
            # Open a PNG plotting device in R
            png(file="{filename}", width=512, height=512)
            # Generate the histogram using R
            hist({dataset}[["{colname_str}"]], probability = TRUE,
                 main=paste("Histogram of", "{colname_str}"),
                 xlab="{colname_str}",
                 col="lightblue",
                 border="black")
            # Add a normal distribution curve
            curve(dnorm(x, mean=mean({dataset}[["{colname_str}"]], na.rm=TRUE)
                 sd=sd({dataset}[["{colname_str}"]], na.rm=TRUE)),
                 col="red", lwd=2, add=TRUE)
            dev.off()
            ''')

        # Display the histogram in Python
        display(Image(filename))

# Call the function to generate and display histograms
display_histograms()
```

Data Analysis Insight: After inspecting the histograms for the predictor variables, it is evident that the majority exhibit right-skewness. To stabilize the skewed distributions and improve normality, we will apply log transformations to these variables. This will allow us to better analyze the relationships between predictors and the response variable.

Dynamic Programming Approach: To demonstrate flexibility and dynamic programming skills, I will perform the Box-Cox transformation analysis in Python. This approach enables me to systematically evaluate whether a log transformation or an alternative power transformation is appropriate for each variable, based on their distribution. By leveraging

the interoperability between Python and R, I can efficiently switch between these languages to handle different aspects of the analysis and visualization to suit my personal strengths.

In [ ]:
```
# Load R object into Python
data = ro.r('corr_based_filtered_predictors')
ncol_data = ro.r('colnames(corr_based_filtered_predictors)')

# Convert R dataframe to a pandas dataframe using conversion
with conversion.localconverter(ro.default_converter + pandas2ri.converter):
    data = ro.conversion.rpy2py(data)

# Ensure it's a pandas dataframe
data = pd.DataFrame(data)

# Create dictionaries to store columns and the data associated with columns
positive_only_indicators = {}
needs_log = {}
no_log_transform = {}
unsure_indicators = {}

# Loop through each column in the dataframe
for column in data.columns:
    # Add small value to indicators that range from [0,100]
    if (data[column] <= 0).any():
        data[column] += 0.001

    # After adjusting, check if the column has only positive values for Box-Cox
    if (data[column] > 0).all():
        # Apply Box-Cox transformation
        _, lambda_val = stats.boxcox(data[column].values)  # Use .values to pas
        positive_only_indicators[column] = data[column]  # Store the data for t

        # Check if the lambda suggests log transformation (lambda ~ 0) or no t
        if -0.125 < lambda_val < 0.125:
            print(f"{column} is best log transformed (lambda ~ 0)")
            needs_log[column] = data[column]  # Store the data for log transfo
        elif 0.875 < lambda_val < 1.125:
            print(f"{column} does not need transformation (lambda ~ 1)")
            no_log_transform[column] = data[column]  # Store the data for no lo
        else:
            print(f"{column} is best transformed with lambda: {lambda_val}")
            unsure_indicators[column] = data[column]  # Store the data for unsu

# Convert 'ncol_data' (an R vector) to a Python list
ncol_data = list(ncol_data)

# Find and print the names of the columns that are in 'ncol_data' but not in 'p
non_positive_indicators = list(set(ncol_data) - set(positive_only_indicators.ke

# Update unsure_indicators with non-positive columns
for col in non_positive_indicators:
    unsure_indicators[col] = data[col]
```

In [ ]:
```
# Expecting total number of indicators based on initial correlation filtering
print(f"Total number of correlation filtered indicators: {len(ncol_data)}")
# Expecting total number of indicators that are positive and pass initial filte
print(f"Total number of positive-only indicators: {len(positive_only_indicators
# Print number of non-positive indicators
```

```
print(f"Total number of non-positive indicators: {len(non_positive_indicators)}
# Print number of non-positive indicators
print(f"Total number of indicators needing log transformation: {len(needs_log)}
# Print number of no log transformation indicators
print(f"Total number of indicators with no log transformation: {len(no_log_tra
# Print number of unsure indicators
print(f"Total number of indicators unsure: {len(unsure_indicators)}")
# Print names of non-positive indicators
print(f"Non-positive indicators:({non_positive_indicators})")
```

In [ ]:
```
print("\nLog needing indicators:")
print('\n'.join(needs_log.keys()))
```

In [ ]:
```
print("\nNo log transformation indicators:")
print('\n'.join(no_log_transform.keys()))
```

In [ ]:
```
print("\nUnsure indicators:")
print('\n'.join(unsure_indicators.keys()))
```

Now we will evaluate the unsure indicators using skew and kurtosis and from here, fit a full model to finalize initial base model creation, having started with 90+ predictors for the response variable.

In [ ]:
```
for column in unsure_indicators:
    print(f"{column}: Skewness = {skew(data[column])}, Kurtosis = {kurtosis(da
```

Transformation Diagnosis

Skewness and Kurtosis Results

- **Skewness in range [-0.5,0.5]** -> Fairly symmetric
- **Kurtosis = 3** -> Normal distribution

1. **High**: Highly right-skewed (1.89) with moderately heavy tails (3.75). **Apply log transformation.**
2. **Volume**: Highly right-skewed (1.88) with heavy tails (5.02). **Apply log transformation.**
3. **volatility_bbh**: Highly right-skewed (1.90) with moderately heavy tails (3.77). **Apply log transformation.**
4. **volatility_bbw**: Moderately right-skewed (1.05) with light tails (1.37). **Transformation not critical, but log transformation may help normalize.**
5. **volatility_kcw**: Highly right-skewed (2.03) with very heavy tails (7.04). **Apply log transformation and consider handling outliers.**
6. **volatility_dch**: Highly right-skewed (1.94) with heavy tails (4.01). **Apply log transformation.**
7. **trend_macd_signal**: Highly right-skewed (1.84) with extremely heavy tails (12.03). **Apply log transformation and handle outliers separately**.
8. **volume_adi**: Highly left-skewed (-1.19) with near-normal tails (2.62). **Apply cube transformation or square transformation to reduce left skew.**

9. **trend_kst_sig**: Extremely left-skewed (-4.60) with extremely heavy tails (32.65). **Apply square transformation and handle outliers.**
10. **momentum_ppo**: Slightly left-skewed (-0.48) with light tails (0.68). **Transformation not necessary.**
11. **momentum_ao**: Highly right-skewed (1.42) with extremely heavy tails (11.67). **Apply log transformation and handle outliers.**
12. **momentum_ppo_signal**: Slightly left-skewed (-0.48) with light tails (0.70). **Transformation not necessary.**
13. **volume_vpt**: Near symmetric (0.14) with light tails (0.63). **No transformation necessary.**
14. **volume_obv**: Moderately left-skewed (-0.83) with light tails (1.65). **Apply square root transformation.**
15. **trend_kst**: Extremely left-skewed (-4.41) with extremely heavy tails (32.38). **Apply square transformation and handle outliers.**
16. **trend_macd**: Highly right-skewed (1.81) with extremely heavy tails (12.37). **Apply log transformation and manage outliers.**

Summary:

- **Log transformations**: For most highly right-skewed variables.
- **Square or cube transformations**: For highly left-skewed variables.
- **Handle outliers**: For variables with extremely heavy tails (e.g., kurtosis > 10).

In [ ]:
```
import rpy2.robjects as ro #  Data Preparation: Applying Transformations
from rpy2.robjects.lib import grdevices
from IPython.display import Image, display
import numpy as np
import pandas as pd
from rpy2.robjects import pandas2ri
pandas2ri.activate()

# Convert to df for easier analysis
needs_log_df = pd.DataFrame(needs_log)
no_log_transform_df = pd.DataFrame(no_log_transform)
unsure_indicators_df = pd.DataFrame(unsure_indicators)

# List of columns to add to needs_log_df (log transformation)
additional_log_columns = ["High", "Volume", "volatility_bbh", "volatility_dch"

# Add these columns to needs_log_df
needs_log_df = pd.concat([needs_log_df, data[additional_log_columns]], axis=1)

# List of all columns that need cube transformation instead of cube root
cube_columns = ["volume_adi", "volume_obv", "trend_kst_sig", "trend_kst", "mom

# Apply cube transformation
cube_df = data[cube_columns].apply(lambda x: np.power(x, 3))  # Cube transforma

# List of columns to be added as normal predictors (no transformation needed)
normal_predictors_columns = ["volume_vpt", "momentum_ppo", "momentum_ppo_signa

# Create normal_predictors_df
```

```
normal_predictors_df = data[normal_predictors_columns]

# Apply log transformation to needs_log_df
needs_log_df = needs_log_df.apply(np.log)

# Combine all DataFrames (log transformed, cubed, and normal predictors)
final_df = pd.concat([needs_log_df, cube_df, normal_predictors_df], axis=1)

# Add the response variable and apply log transformation
response_r = ro.r('train_data[["Close"]]')  # Fetch the correct response from I
response_r = np.log(response_r)  # Log transformation of the response

response_r = pd.DataFrame(response_r)
final_df['log_Close'] = response_r.to_numpy()[:, 0]

# Drop any rows with missing values
final_df = final_df.dropna(axis=0)

# Ensure no columns are missing from the transformations
all_transformed_columns = list(needs_log_df.columns) + list(cube_df.columns) +

# Find and print the names of the columns that are tricky based on box cox, ske
predictors_need_care = list(set(data.columns) - set(all_transformed_columns))

# Check if predictors_need_care is empty
if not predictors_need_care:
    print("All columns have been transformed successfully.")
else:
    print(f"The following columns still need care: {predictors_need_care}")
    predictors_need_care_df = data[predictors_need_care]  # Create df for trick
```

In [ ]:

```
#  Data Visualization: Renaming Columns and Displaying Histograms
def display_transformed_histograms():
    # Create a copy of final_df to avoid modifying the original data
    transformed_df = final_df.copy()

    # Get the column names from final_df
    colnames = final_df.columns

    # Loop through each column and apply transformation logic
    for colname in colnames:
        colname_str = str(colname)

        # Determine the type of transformation applied
        if colname_str in needs_log_df.columns:
            transform_type = 'Log-Transformed'
            new_colname = f"log_{colname_str}"
            transformed_df.rename(columns={colname_str: new_colname}, inplace='
        elif colname_str in cube_df.columns:
            transform_type = 'Cube Transformed'
            new_colname = f"cube_{colname_str}"
            transformed_df.rename(columns={colname_str: new_colname}, inplace='
        elif colname_str == "log_Close":  # Explicitly handle log_Close
            transform_type = 'Log-Transformed'
            new_colname = colname_str
        else:
            transform_type = 'Untransformed'
            new_colname = colname_str

        # Create a filename for the histogram
```

```
        filename = f"histogram_{new_colname}.png"

        # Convert the transformed DataFrame to an R DataFrame
        transformed_df_r = pandas2ri.py2rpy(transformed_df)

        # Assign the R dataframe to a variable in R
        ro.globalenv['transformed_df'] = transformed_df_r

        # Open a PNG plotting device in R
        grdevices.png(file=filename, width=512, height=512)

        # Prepare the R code for plotting
        ro.r(f'''
        # Extract the data column
        data_vector <- transformed_df[["{new_colname}"]]

        # Plot the histogram
        hist(data_vector, probability = TRUE,
             main=paste("{transform_type} Histogram of", "{colname}"),
             xlab=paste("{transform_type}", "{colname}"),
             col="lightblue",
             border="black")

        # Add a normal distribution curve
        curve(dnorm(x, mean=mean(data_vector, na.rm=TRUE), sd=sd(data_vector, r
              col="red", lwd=2, add=TRUE)
        ''')

        # Close the PNG device
        grdevices.dev_off()

        # Display the histogram in Python
        display(Image(filename))

    return transformed_df

# Call the function to display histograms and return the transformed DataFrame
transformed_df = display_transformed_histograms()
```

In [ ]:
```
transformed_df.head()
# this is as desired
```

Cube Root transformations were not good, went back and did Cube transformations, these were better. Some variables likely have outliers making simple transformations ineffective. We need to take a closer look at these variables later, potentiall using robust regression methods to take care of outliers.

In [ ]:
```
# Convert needs_log_df to R DataFrame and move to R environment
needs_log_df_r = pandas2ri.py2rpy(needs_log_df.dropna()) # Use dropna to handl
ro.globalenv['needs_log_df'] = needs_log_df_r

ro.globalenv['final_df'] = final_df

print(ro.r('head(needs_log_df)'))
```

In [ ]:
```
# Convert normal_predictors_df to R DataFrame and move to R environment
normal_predictors_df_r = pandas2ri.py2rpy(normal_predictors_df.dropna()) # Use
```

```
ro.globalenv['normal_predictors_df'] = normal_predictors_df_r
print(ro.r('head(normal_predictors_df)'))
```

In [ ]:
```
# Convert cube_root_df to R DataFrame and move to R environment
cube_df_r = pandas2ri.py2rpy(cube_df.dropna()) # Use dropna to handle potentia
ro.globalenv['cube_df'] = cube_df_r
print(ro.r('head(cube_df)'))
```

In [ ]:
```
print(ro.r('head(final_df)')) # In R, final_df has log_Close
```

In [ ]:
```
print(ro.r('head(transformed_df)')) # In R, tranformed_df adjusts column names

# Transformed response and predictor
print(ro.r('ncol(transformed_df)'))
```

In [ ]:
```
# Retrieve the test_data from the R environment with adjustments
test_data = pandas2ri.rpy2py(ro.globalenv['test_data'])

# Function to apply the same transformations to the test set
def transform_test_set(test_data):
    # Create a copy of test_data to avoid modifying the original test data
    transformed_test_df = test_data.copy()

    # Get the column names from the training set (transformed_df)
    colnames = transformed_df.columns  # This includes the transformed names

    # Apply transformations to the test set based on training set transformati
    for colname in test_data.columns:
        colname_str = str(colname)

        # Check if the column was log-transformed in the training set
        if f"log_{colname_str}" in colnames:
            # Apply log transformation and rename the column
            transformed_test_df[f"log_{colname_str}"] = np.log(transformed_tes
            transformed_test_df.drop(columns=[colname_str], inplace=True)  # D

        # Check if the column was cube root-transformed in the training set
        elif f"cube_{colname_str}" in colnames:
            # Apply cube root transformation and rename the column
            transformed_test_df[f"cube_{colname_str}"] = transformed_test_df[c

            transformed_test_df.drop(columns=[colname_str], inplace=True)  # D

    # Return the transformed test set
    return transformed_test_df

# Apply the function to transform the test set
transformed_test_df = transform_test_set(test_data)

print(transformed_test_df.head())
# Now the test set (transformed_test_df) has the same transformations and colu
```

In [ ]:
```
# Convert final, transformed test df to an R DataFrame
transformed_test_df_r = pandas2ri.py2rpy(transformed_test_df)

# Assign the R dataframe to a variable in R
ro.globalenv['transformed_test_df'] = transformed_test_df_r
```

```
ro.r('''

    transformed_test_df = as.data.table(transformed_test_df)
    print(head(transformed_test_df))

    ''')
```

In [ ]:
```
# Ensure transformed_test_df_clean only contains columns present in transforme
def align_columns(df_source, df_target):
    # Find common columns between source and target
    common_columns = df_source.columns.intersection(df_target.columns)
    # Align target dataframe to source dataframe's common columns
    df_target_aligned = df_target[common_columns]
    return df_target_aligned

# Apply the alignment to both training and test sets
transformed_test_df_clean = align_columns(transformed_df, transformed_test_df)

# Print the first few rows of the aligned test dataframe
print(transformed_test_df_clean.head())

# Check the new shapes of the aligned dataframes
print("Training set number of columns:", transformed_df.shape[1])
print("Test set number of aligned columns:", transformed_test_df_clean.shape[1
```

In [ ]:
```
# This has everything as desired, all predictors and the response based on cor
print(ro.r('ncol(transformed_df)'))

# This does not yet
print(ro.r('ncol(transformed_test_df)'))

# Transform the R dataframe to reflect the changes made in Python
ro.globalenv['transformed_test_df'] = transformed_test_df_clean
print(ro.r('ncol(transformed_test_df)'))

# Problem has been resolved
```

In [ ]:
```
ro.r(''' # Fit a linear model with all predictors
    # Fit a linear model with all predictors
    full_model <- lm(log_Close ~ ., data = transformed_df)
    # Grab summary statistics
    print(summary(full_model))
''')


ro.r('''
    set.seed(123)
    # Exponentiate the values in the test set to get them back on the real sca
    y_test <- transformed_test_df$log_Close
    exp_y_test <- exp(transformed_test_df$log_Close)

    # Use the full model to predict on the test data
    predictions_full_test <- predict(full_model, newdata = transformed_test_df

    # Exponentiate the predicted values to get them on the real scale
    predictions_full_test_exp <- exp(predictions_full_test)

    # Apply smearing correction to account for bias
    residuals_full <- y_test - predictions_full_test  # Residuals on the log s
```

```
    correction_factor <- mean(exp(residuals_full))  # Smearing correction facto
    predictions_full_test_exp_smearing <- predictions_full_test_exp * correcti

    # Calculate Full Model MSPE on the log scale (without exponentiating)
    full_model_mspe_log <- mean((y_test - predictions_full_test)^2)

    # Calculate Full Model MSPE on the real scale (after smearing correction)
    full_model_mspe_real <- mean((exp_y_test - predictions_full_test_exp_smear

    # Print the results
    cat("Full Model MSPE (Log Scale):", full_model_mspe_log, "\n")
    cat("Full Model MSPE (Real Scale, Smearing Corrected):", full_model_mspe_r
''')
```

In [ ]:
```
ro.r(''' # Calculate VIF for all predictors
    # VIF calculation function is in the 'car' package
    library(car)

    # Calculate VIF for the full model
    vif_values <- vif(full_model)

    # Identify variables with high VIF
    high_vif <- names(vif_values[vif_values > 10])

    # Identify variables with low VIF
    low_vif <- names(vif_values[vif_values <= 10])

    # Display results
    cat("Variables with high VIF:\n")
    print(high_vif)
    cat("Variables with low VIF:\n")
    print(low_vif)
    ''')
```

We will look at the model that results from VIF to see how it is.

In [ ]:
```
ro.r(''' # Fit a linear model with VIF predictors
    vif_model <- lm(log_Close ~ log_Volume + cube_volume_adi + cube_volume_obv
    # Print summary statistics
    print(summary(vif_model))
''')
```

In [ ]:
```
ro.r(''' # Get the actual test values in log scale
    y_test <- transformed_test_df$log_Close
    exp_y_test <- exp(transformed_test_df$log_Close)

    # Predict on the test data using the VIF model
    predictions_test <- predict(vif_model, newdata = transformed_test_df)

    # Exponentiate the predictions to get them on the real scale
    predictions_test_exp <- exp(predictions_test)

    # Calculate MSPE for the log scale (no need to exponentiate)
    vif_mspe <- mean((y_test - predictions_test)^2)

    # Calculate MSPE for the real scale
    vif_real_mspe <- mean((exp_y_test - predictions_test_exp)^2)
```

```
    # Print the results
    cat("VIF MSPE (Log Scale):", vif_mspe, "\n")
    cat("Full Model MSPE (Log Scale):", full_model_mspe_log, "\n")

''')
```

Model Performance: Full Model vs. VIF Model

We evaluated two regression models for predicting financial data: the **full model** (with all available transformed and untransformed predictors based on correlation filtering) and a **simplified VIF model** (with predictors filtered using correlation, histogram analysis, and VIF elimination). Both models were trained on log-transformed data and tested on unseen data to assess their generalization abilities.

Methodology:

- **Full Model**: Used all predictors, capturing a broader range of relationships.
- **VIF Model**: Utilized a reduced set of predictors, focusing on minimizing multicollinearity and simplifying the model.

Test Set Performance:

- **VIF Model MSPE**:
  - Log Scale: `0.0834`
- **Full Model MSPE**:
  - Log Scale: `4.23e-05`

Conclusion:

The **full model** significantly outperformed the VIF model, especially on the real scale. This is likely due to the full model's ability to capture the inherent complexities and subtle relationships present in financial data. While the VIF model was simplified to reduce multicollinearity, it likely missed important predictors, leading to higher prediction error on unseen data.

Regularization Methods

Now I will do regularized regression methods, Lasso, Ridge, and ElasticNet.

In [ ]:
```
ro.r(''' # Ridge
# Load necessary libraries
library(glmnet)
library(caret)

# For reproducibility
set.seed(123)

# transformed_df uses the training data
# Convert predictors and response using model.matrix
```

```
x <- model.matrix(log_Close ~ . - 1, transformed_df)  # Create design matrix, 
y <- transformed_df$log_Close  # The response variable


# Ridge Regression (alpha = 0)
ridge_model <- cv.glmnet(x, y, alpha = 0)
print("Ridge Regression:")
print(ridge_model)

# Optimal lambda for the Ridge model
best_lambda_ridge <- ridge_model$lambda.min
cat("Best lambda for Ridge:", best_lambda_ridge, "\\n")

''')
```

In [ ]:
```
ro.r(''' # Ridge Coefficients
set.seed(123)
# Get Ridge coefficients
ridge_coefs <- coef(ridge_model, s = best_lambda_ridge)
print("Ridge Coefficients:")
print(ridge_coefs)
     ''')
```

In [ ]:
```
ro.r(''' # Lasso
set.seed(123)
# Lasso Regression (alpha = 1)
lasso_model <- cv.glmnet(x, y, alpha = 1)
print("Lasso Regression:")
print(lasso_model)

# Optimal lambda for the Ridge model
best_lambda_lasso <- lasso_model$lambda.min
cat("Best lambda for Lasso:", best_lambda_lasso, "\\n")
     ''')
```

In [ ]:
```
ro.r(''' # Coefficients from the Lasso model
lasso_coefs <- coef(lasso_model, s = best_lambda_lasso)
print(lasso_coefs)
     ''')
```

In [ ]:
```
ro.r('''   # ElasticNet
# Set seed for reproducibility
set.seed(123)

x <- model.matrix(log_Close ~ . - 1, transformed_df)  # Create design matrix, 
y <- transformed_df$log_Close  # The response variable


# Define the alpha values to loop over (e.g., from 0 to 1 in steps of 0.1)
alpha_values <- seq(0, 1, by = 0.1)

# Initialize variables to store the best results
best_alpha <- NULL
best_lambda <- NULL
lowest_mse <- Inf  # Set initial MSE to infinity

# Loop through each alpha value
for (alpha_value in alpha_values) {
```

```
    # Fit ElasticNet model for each alpha
    elasticnet_model <- cv.glmnet(x, y, alpha = alpha_value)

    # Get the best lambda for this alpha
    best_lambda_for_alpha <- elasticnet_model$lambda.min

    # Predict using the best lambda
    predictions <- predict(elasticnet_model, s = best_lambda_for_alpha, newx =

    # Calculate MSE
    mse <- mean((y - predictions)^2)

    # Output the appropriate message for each model type
    if (alpha_value == 0){
        cat("Ridge Regression: MSE:", mse, " | Best Lambda:", best_lambda_for_a
    } else if (alpha_value == 1){
        cat("Lasso Regression: MSE:", mse, " | Best Lambda:", best_lambda_for_a
    } else {
        cat("Alpha:", alpha_value, " | MSE:", mse, " | Best Lambda:", best_laml
    }

    # Update the best alpha and lambda if this model has the lowest MSE
    if (mse < lowest_mse) {
        best_alpha <- alpha_value
        best_lambda <- best_lambda_for_alpha
        lowest_mse <- mse
    }
}

# Print the best alpha, lambda, and corresponding MSE
cat("\nBest Alpha:", best_alpha, "\n")
cat("Best Lambda:", best_lambda, "\n")
cat("Lowest MSE:", lowest_mse, "\n")
''')
```

In [ ]:
```
ro.r(''' # Coefficients from the ElasticNet model
elasticnet_coefs <- coef(elasticnet_model, s = best_lambda_enet)
print(elasticnet_coefs)
     ''')
```

In [ ]:
```
ro.r(''' # MSPE
set.seed(123)

# Check and align test data to match training data columns
transformed_test_df <- transformed_test_df[, colnames(transformed_df), drop = F

# Compute MSPE on the test set
test_x <- model.matrix(log_Close ~ . - 1, transformed_test_df)  # Prepare test
test_y <- transformed_test_df$log_Close  # True values from the test set

# Use best lambda and model (ElasticNet, Ridge, or Lasso)
elasticnet_predictions <- predict(elasticnet_model, s = best_lambda, newx = tes
ridge_predictions <- predict(ridge_model, s = best_lambda_ridge, newx = test_x
lasso_predictions <- predict(lasso_model, s = best_lambda_lasso, newx = test_x
vif_predictions <- predict(vif_model, newdata = transformed_test_df)  # VIF mod

# Calculate MSPE for each method
elasticnet_mspe <- mean((test_y - elasticnet_predictions)^2)
ridge_mspe <- mean((test_y - ridge_predictions)^2)
```

```
lasso_mspe <- mean((test_y - lasso_predictions)^2)
vif_mspe <- mean((test_y - vif_predictions)^2)

# Output the results
cat("ElasticNet MSPE (Log Scale):", elasticnet_mspe, "\n")
cat("Ridge MSPE (Log Scale):", ridge_mspe, "\n")
cat("Lasso MSPE (Log Scale):", lasso_mspe, "\n")
cat("VIF MSPE (Log Scale):", vif_mspe, "\n")
cat("Full Model MSPE (Log Scale):", full_model_mspe_log, "\n")

library(MASS)
# Fit robust regression model
robust_model <- rlm(log_Close ~ . , data = transformed_df)
# Predict on the test data
robust_preds <- predict(robust_model, newdata = transformed_test_df)
robust_mspe <- mean((test_y - robust_preds)^2)
cat("Huber Regression MSPE (Log Scale):", robust_mspe, "\n")

# Fit the LTS regression model
lts_model <- lqs(log_Close ~ ., data = transformed_df)
lts_preds <- predict(lts_model, newdata = transformed_test_df)
lts_mspe <- mean((test_y - lts_preds)^2)
cat("Least Trimmed Squares Regression MSPE (Log Scale):", lts_mspe, "\n")

# Fit the quantile regression model at the median (tau = 0.5)
library(quantreg)
qr_model <- rq(log_Close ~ . , data = transformed_df, tau = 0.5)
qr_preds <- predict(qr_model, newdata = transformed_test_df)
qr_mspe <- mean((test_y - qr_preds)^2)
cat("Quantile Regression MSPE (Log Scale):", qr_mspe, "\n")

# Fit penalized quantile regression
library(rqPen)
pen_qr_model <- rq.fit.lasso(x = test_x, y = test_y, tau = 0.5)
pen_qr_preds <- test_x %*% pen_qr_model$coefficients
pen_qr_mspe <- mean((test_y - pen_qr_preds)^2)
cat("Penalized Quantile Regression MSPE (Log Scale):", pen_qr_mspe, "\n")
''')
```

In [285…

```
ro.r(''' # MSPE Real Scale using bias correction
set.seed(123)

# Check and align test data to match training data columns
transformed_test_df <- transformed_test_df[, colnames(transformed_df), drop = F

# Compute MSPE on the test set
test_x <- model.matrix(log_Close ~ . - 1, transformed_test_df)  # Prepare test
test_y <- transformed_test_df$log_Close  # True values from the test set

# Smearing function to apply bias correction
smearing_bias_correction <- function(log_preds, log_true) {
  residuals_log <- log_true - log_preds
  correction_factor <- mean(exp(residuals_log))  # Smearing correction factor
  return(correction_factor)
}

# Function to calculate MSPE on the real scale
calculate_mspe_original <- function(log_preds, log_true) {
  pred_original <- exp(log_preds)
  true_original <- exp(log_true)
```

```
  # Apply smearing bias correction
  correction_factor <- smearing_bias_correction(log_preds, log_true)
  adjusted_preds <- pred_original * correction_factor

  # Calculate MSPE on the original scale
  mspe_original <- mean((true_original - adjusted_preds)^2)
  return(mspe_original)
}

# ElasticNet predictions and MSPE
elasticnet_predictions <- predict(elasticnet_model, s = best_lambda, newx = tes
elasticnet_mspe_real <- calculate_mspe_original(elasticnet_predictions, test_y

# Ridge predictions and MSPE
ridge_predictions <- predict(ridge_model, s = best_lambda_ridge, newx = test_x
ridge_mspe_real <- calculate_mspe_original(ridge_predictions, test_y)

# Lasso predictions and MSPE
lasso_predictions <- predict(lasso_model, s = best_lambda_lasso, newx = test_x
lasso_mspe_real <- calculate_mspe_original(lasso_predictions, test_y)

# VIF Model predictions and MSPE
vif_predictions <- predict(vif_model, newdata = transformed_test_df)  # VIF mod
vif_mspe_real <- calculate_mspe_original(vif_predictions, test_y)

# Full Model MSPE (Real Scale)
full_model_mspe_real <- calculate_mspe_original(predictions_full_test, test_y)

# Robust Model MSPE
robust_preds <- predict(robust_model, newdata = transformed_test_df)
robust_mspe_real <- calculate_mspe_original(robust_preds, test_y)

# LTS Model MSPE
lts_preds <- predict(lts_model, newdata = transformed_test_df)
lts_mspe_real <- calculate_mspe_original(lts_preds, test_y)

# Quantile Regression Model MSPE
qr_preds <- predict(qr_model, newdata = transformed_test_df)
qr_mspe_real <- calculate_mspe_original(qr_preds, test_y)

# Penalized Quantile Regression Model MSPE
pen_qr_preds <- test_x %*% pen_qr_model$coefficients
pen_qr_mspe_real <- calculate_mspe_original(pen_qr_preds, test_y)
''')
```

In [ ]:

```
ro.r(''' # Output the results (Real Scale)
cat("ElasticNet MSPE (Real Scale):", elasticnet_mspe_real, "\n")
cat("Ridge MSPE (Real Scale):", ridge_mspe_real, "\n")
cat("Lasso MSPE (Real Scale):", lasso_mspe_real, "\n")
cat("VIF MSPE (Real Scale):", vif_mspe_real, "\n")
cat("Full Model MSPE (Real Scale):", full_model_mspe_real, "\n")
cat("Huber Regression MSPE (Real Scale):", robust_mspe_real, "\n")
cat("Least Trimmed Squares MSPE (Real Scale):", lts_mspe_real, "\n")
cat("Quantile Regression MSPE (Real Scale):", qr_mspe_real, "\n")
cat("Penalized Quantile Regression MSPE (Real Scale):", pen_qr_mspe_real, "\n"
     ''')
```

Analysis of Robust and Regularized Regression Methods

In my analysis, I applied robust regression methods—Quantile Regression (QR), Least Trimmed Squares (LTS), and Huber's method—and regularized regression methods—Lasso, Ridge, and Elastic Net regression models. My goal was to address outliers and multicollinearity, evaluating the predictive performance of these regression models.

Results

Robust Regression Methods

Model	MSPE (Log Scale)	MSPE (Original Scale)
Quantile Regression	3.6021e-05	0.8046161
Huber Regression	3.789034e-05	0.8522115
Penalized Quantile Regression	7.939872e-05	1.758258
Least Trimmed Squares Regression	0.0007393806	15.15739

Regularized Regression Methods

Model	MSPE (Log Scale)	MSPE (Original Scale)
Ridge Regression	0.0006943541	16.37199
Lasso Regression	0.002135133	4.930664
Elastic Net Regression	0.002135133	4.930664
VIF Model	0.08338693	3076.539
Full Model	4.225889e-05	0.9543641

Interpretation and Conclusions

Robust Regression Methods

- **Quantile Regression (QR)** had the lowest MSPE on both the log and original scales, with **3.6021e-05 (Log)** and **0.8046161 (Original)**, making it the top performer.
- **Huber Regression** followed closely, with an MSPE of **3.789034e-05 (Log)** and **0.8522115 (Original)**, indicating its robustness against outliers.
- **Penalized Quantile Regression** showed a higher MSPE than QR and Huber, with **7.939872e-05 (Log)** and **1.758258 (Original)**.

- **Least Trimmed Squares (LTS) Regression** performed worst among robust methods, with **0.0007393806 (Log)** and **15.15739 (Original)**.

Regularized Regression Methods

- **Ridge Regression** had a relatively low MSPE on the log scale (**0.0006943541**) but was less effective on the original scale (**16.37199**).
- **Lasso** and **Elastic Net Regression** showed similar performance, with **0.002135133 (Log)** and **4.930664 (Original)**.
- The **VIF Model** performed the worst, with an MSPE of **0.08338693 (Log)** and **3076.539 (Original)**, indicating suboptimal predictions.
- The **Full Model** was the strongest regularized method, with **4.225889e-05 (Log)** and **0.9543641 (Original)**.

Mathematical Reasoning

- **Quantile Regression (QR)** is effective at estimating the median relationship between the predictors and the response. It minimizes the influence of extreme values by focusing on the median rather than the mean, which explains its superior performance on both scales.
- **Huber Regression** combines squared error for small residuals and absolute error for large residuals. This approach makes it resistant to outliers while preserving the efficiency of least squares for typical data points.
- **Regularized Regression (Ridge, Lasso, and Elastic Net)** applies penalties to the regression coefficients. Ridge regression uses the L2 penalty to shrink coefficients of correlated predictors, while Lasso applies an L1 penalty to enforce sparsity. Elastic Net is a combination of both L1 and L2 penalties, balancing shrinkage and variable selection.
- The **VIF Model** was built by removing predictors with high multicollinearity based on the variance inflation factor. However, this simplistic approach likely resulted in the removal of useful predictors, leading to poor predictive performance.

Practical Reasoning

Financial data often contains extremes and high volatility, which can distort traditional models. Robust methods like Quantile and Huber regression are less influenced by outliers, making them well-suited to handle the irregularities in financial data. This is why these models performed better than regularized methods, which may be more sensitive to such fluctuations.

Takeaways

1. **Quantile Regression and Huber Regression outperformed the regularized methods**, making them the most effective for this dataset.
2. **Ridge Regression performed well on the log scale**, but it struggled when transformed back to the original scale.
3. **The Full Model showed the best performance among regularized methods**, while the **VIF Model** underperformed.

Transition to Time Series Analysis (TSA)

Following these analyses, the next logical step is to incorporate time-dependent patterns through **Time Series Analysis (TSA)**. By applying **ARIMA** and **GARCH** models, we aim to capture autocorrelation and volatility clustering, respectively. This transition will allow us to account for the temporal structure in stock prices and improve model performance further.

Time Series Analysis (TSA)

While the regression models provided valuable insights, they may not fully capture temporal dependencies inherent in time series data. **Time Series Analysis** techniques are specifically designed to model and forecast data where observations are correlated over time.

Next Steps:

1. **Exploratory Time Series Analysis:**
    - Visualize the time series plots of the response variable to identify any underlying **trends**, **seasonality**, and **autocorrelation** patterns.
    - Use tools like the **Autocorrelation Function (ACF)** and **Partial Autocorrelation Function (PACF)** plots to help determine the order of the AR and MA components in an ARIMA model.
2. **Stationarity Assessment:**
    - Apply stationarity tests such as the **Augmented Dickey-Fuller (ADF) test** to check for stationarity.
    - If the data is non-stationary, use **differencing** or transformations (e.g., log transformations) to achieve stationarity.
3. **Modeling Temporal Dependencies:**
    - Fit **ARIMA (AutoRegressive Integrated Moving Average)** or **SARIMA (Seasonal ARIMA)** models to capture autocorrelation and other temporal structures.
    - Consider integrating a **GARCH (Generalized Autoregressive Conditional Heteroskedasticity)** model to handle volatility clustering and model variance over time.

4. **Forecast Evaluation:**

    - Compare the forecast performance of TSA models (e.g., ARIMA, SARIMA, GARCH) with the static regression models using **forecast accuracy metrics** like **Mean Squared Prediction Error (MSPE)**, **Mean Absolute Error (MAE)**, and **Mean Absolute Percentage Error (MAPE)**.

By incorporating Time Series Analysis techniques, I will better understand the dynamics of the data over time, leading to more accurate and robust forecasts.

Further Justifications

Given the inherent autocorrelation in stock market data, the next step is to conduct Time Series Analysis (TSA) to capture the temporal patterns. By applying ARIMA and potentially integrating GARCH models, we aim to model both the time-dependent trends and volatility in the data. This should help alleviate the burden on other predictors, allowing them to better capture the underlying relationships.

Once TSA is incorporated, we will re-evaluate the performance of robust and regularized regression methods, now accounting for time-series noise. Following TSA, we plan to explore Principal Component Analysis (PCA) and tree-based models (e.g., Random Forests) to further address multicollinearity, improve feature selection, and enhance prediction accuracy.

This approach ensures a comprehensive model that leverages both temporal dynamics and feature-based insights for more robust predictive performance.

In [ ]:
```
# Create a function to save R plots to a PNG file and display them in Python

from IPython.display import Image, display
def display_r_plot(filename):
    display(Image(filename))


ro.r('''
# Convert the 'Close' column to a time series object
close_ts <- ts(aapl_r_df$Close, start = c(2009, 1), frequency = 252) # Assuming

# Transform the response variable as we did previously from Exploratory Analysis
close_ts <- log(close_ts)
# Verify we have created a time series object
print(class(close_ts))

# Create a new column in the dataframe to hold log_close
aapl_r_df$log_Close <- log(aapl_r_df$Close)
# Check the basic structure of the time series data
print(summary(close_ts))
plot(close_ts, main = "Close Price Time Series", ylab = "Price", xlab = "Time"
    ''')

ro.r('''
    # Decompose the time series into trend, seasonal, and residual components
```

```
    decomposed_close <- decompose(close_ts)

    # Plot the decomposition
    png(filename = "decomposition_plot.png", width = 800, height = 600)
    plot(decomposed_close)
    dev.off()
''')

# Display the decomposition plot
display_r_plot("decomposition_plot.png")
ro.r('''

# Plot the 'Close' price
png(filename = "Close_price.png", width = 600, height = 600)

# Plot the close prices directly without storing it in a variable
plot(aapl_r_df$Date, aapl_r_df$log_Close, type = "l", col = "blue",
     main = "Log Close Price Over Time", xlab = "Date", ylab = "Close Price")

# Close the PNG device
dev.off()

# Check for stationarity using the Augmented Dickey-Fuller (ADF) test
adf_test <- adf.test(aapl_r_df$log_Close)
print(adf_test)
''')

# Display the plot with the corresponding filename
display_r_plot("Close_price.png")
ro.r('''
# Apply differencing to make the series stationary
differenced_close <- diff(close_ts)

# Plot the differenced series
png(filename = "differenced_close.png", width = 800, height = 600)
plot(differenced_close, main="Differenced Log Close Price Time Series", ylab="L
dev.off()
''')
# Display the plot
display_r_plot("differenced_close.png")


# Perform the ADF test on the differenced series
ro.r('''
adf_test_diff <- adf.test(differenced_close)
print(adf_test_diff)
''')
# ACF plot example
ro.r('''
    # Open PNG device with a custom filename for ACF
    png(filename = "acf_plot.png", width = 800, height = 600)

    # Create ACF plot
    acf(aapl_r_df$Close, main="ACF of Log Close Prices")

    # Close the PNG device
    dev.off()
''')

# Display the ACF plot
```

```
display_r_plot("acf_plot.png")

# PACF plot example
ro.r('''
    # Open PNG device with a custom filename for PACF
    png(filename = "pacf_plot.png", width = 800, height = 600)

    # Create PACF plot
    pacf(aapl_r_df$Close, main="PACF of Log Close Prices")

    # Close the PNG device
    dev.off()
''')

# Display the PACF plot
display_r_plot("pacf_plot.png")
```

Our ACF is slowly decreasing, while the PACF cuts off at lag 2, indicating the presence of an autoregressive (AR) process. We have also differenced the data once to achieve stationarity, which suggests an integration order of 1. Based on the structure of the ACF, where lag 1 appears almost 1, an additional moving average (MA) component might also be necessary to capture the full dynamics of the data.

In [ ]:

```
ro.r(''' # ARIMA testing
# Set up ranges for AR and MA components
p_values <- c(1, 2)  # Possible values for AR (from PACF analysis)
q_values <- c(0, 1, 2, 3)  # Possible values for MA (from ACF analysis)
d <- 1  # The differencing order has already been determined

# Initialize variables to store the best model and criteria
best_aic <- Inf
best_bic <- Inf
best_model_aic <- NULL
best_model_bic <- NULL

# Loop over p and q values
for (p in p_values) {
  for (q in q_values) {
    # Try fitting the ARIMA model with current p, d, q
    arima_model <- tryCatch({arima(differenced_close, order = c(p, d, q))}, er

    # Check if the model fitting was successful
    if (!is.null(arima_model)) {
      # Calculate AIC and BIC for the model
      current_aic <- AIC(arima_model)
      current_bic <- BIC(arima_model)

      # Update the best model based on AIC
      if (current_aic < best_aic) {
        best_aic <- current_aic
        best_model_aic <- arima_model
      }

      # Update the best model based on BIC
      if (current_bic < best_bic) {
        best_bic <- current_bic
        best_model_bic <- arima_model
      }
```

```
      # Print the result for this model
      cat("ARIMA(", p, ",", d, ",", q, "): AIC =", current_aic, ", BIC =", cur
    }
  }
}

# Print the best models
cat("\nBest model based on AIC:\n")
print(best_model_aic)
cat("\nBest model based on BIC:\n")
print(best_model_bic)
''')
```

Time Series Analysis (TSA) of AAPL Close Prices

Summary of TSA Process

I conducted time series analysis (TSA) on the **AAPL Close prices**, focusing on selecting the best-fitting **ARIMA** model. My main goal was to model the temporal dependencies in the differenced and log-transformed series and identify the optimal model based on **AIC** and **BIC** criteria.

Preprocessing Steps

1. **Log Transformation**:
    - I log-transformed the **Close** prices to stabilize variance and improve normality in the time series data.
2. **Differencing**:
    - To achieve stationarity, I differenced the log-transformed series once, which was confirmed by the **Augmented Dickey-Fuller (ADF) test**.
    - The test indicated that the differenced series was stationary, with a **p-value of 0.01**.

ACF and PACF Analysis

- **ACF**: The slow decay in the ACF plot suggested the need for a **moving average (MA)** component. Lag 1 showed a strong correlation close to 1, supporting an **MA(1)** process.
- **PACF**: The sharp cut-off in the PACF after lag 2 indicated that an **AR(1)** or **AR(2)** process could be appropriate.

ARIMA Model Selection

Based on the ACF and PACF plots, I tested several **ARIMA(p, d, q)** models with different AR (p) and MA (q) components. The differencing (d) was fixed at 1.

ARIMA Model Results:

ARIMA Model (p, d, q)	AIC	BIC
ARIMA(1, 1, 0)	-16903.04	-16890.71
ARIMA(1, 1, 1)	**-18368.09**	**-18349.59**
ARIMA(1, 1, 2)	-18366.17	-18341.51
ARIMA(1, 1, 3)	-18367.07	-18336.24
ARIMA(2, 1, 0)	-17295.14	-17276.64
ARIMA(2, 1, 1)	-18366.09	-18341.43
ARIMA(2, 1, 2)	-18364.15	-18333.32
ARIMA(2, 1, 3)	-18363.13	-18326.14

Best ARIMA Model

- **Best Model based on AIC**: **ARIMA(1, 1, 1)** with AIC = **-18368.09**.
- **Best Model based on BIC**: **ARIMA(1, 1, 1)** with BIC = **-18349.59**.

ARIMA(1, 1, 1) Model Coefficients:

- The **AR(1)** coefficient is **-0.0431** with a standard error of **0.0168**.
- The **MA(1)** coefficient is **-1.0000** with a standard error of **0.0012**.
- The **sigma^2** (error variance) is estimated as **0.0003154**.

Conclusion

- The **ARIMA(1, 1, 1)** model was identified as the best fit for the differenced log-transformed data based on both AIC and BIC.
- This model includes both an autoregressive (AR) and a moving average (MA) component, effectively capturing the underlying time dependencies in the data.
- The low error variance and strong fit to the data make this model suitable for further time-series forecasting.

Next Steps

1. **Residual Diagnostics**: I will perform residual checks to ensure no remaining autocorrelation.
2. **Advanced Models**: I will consider using **ARIMAX**, **GARCH**, or **EGARCH** to capture additional patterns, such as volatility clustering.
3. **Forecasting**: I plan to use the selected ARIMA model to forecast future stock prices and assess its performance.

This analysis provides a thorough statistical basis for the model selection, ensuring I choose the most appropriate time-series model for predictive purposes.

In [ ]:
```
ro.r(''' # Extract residuals from the best ARIMA model

residuals_arima <- residuals(best_model_aic)

# Plot residuals
png(filename = "residuals_plot.png", width = 800, height = 600)
plot(residuals_arima, main="Residuals from ARIMA(1, 1, 1)", ylab="Residuals", x
dev.off()
''')
display_r_plot("residuals_plot.png")

ro.r('''
# Check for autocorrelation in the residuals using ACF
png(filename = "residuals_acf.png", width = 800, height = 600)
acf(residuals_arima, main="ACF of Residuals from ARIMA(1, 1, 1)")
dev.off()
''')
display_r_plot("residuals_acf.png")
```

In [ ]:
```
ro.r(''' # Perform Ljung-Box test to check for any remaining autocorrelation
ljung_box_test <- Box.test(residuals_arima, lag=10, type="Ljung-Box")
print(ljung_box_test)
     ''')
```

In [ ]:
```
ro.r(''' # ARIMA residuals
arima_211 <- arima(differenced_close, order =c(2,1,1))
residuals_arima_211 <- residuals(arima_211)

# Plot residuals from ARIMA(2, 1, 1)
png(filename = "residuals_arima_211.png", width = 800, height = 600)
plot(residuals_arima_211, main="Residuals from ARIMA(2, 1, 1)", ylab="Residuals
dev.off()
''')
display_r_plot("residuals_arima_211.png")

# Check the ACF of the residuals
ro.r('''
png(filename = "acf_residuals_arima_211.png", width = 800, height = 600)
acf(residuals_arima_211, main="ACF of Residuals from ARIMA(2, 1, 1)")
dev.off()
''')
display_r_plot("acf_residuals_arima_211.png")
```

In [ ]:
```
ro.r(''' # Perform the Ljung-Box test to check for remaining autocorrelation
ljung_box_test_211 <- Box.test(residuals_arima_211, lag=10, type="Ljung-Box")
print(ljung_box_test_211)
''')
```

In [ ]:
```
ro.r(''' # ARIMA summary
    # Print ARIMA(1,1,1) summary
    print(summary(best_model_aic))
    # Print ARIMA(2,1,1) summary to see if change solved issues
    print(summary(arima_211))
     ''')
```

ARIMA Model Selection and Analysis

In the initial stages of Time Series Analysis, I explored various ARIMA models to identify the best fit for the data using AIC and BIC criteria. Both criteria converged on the same ARIMA model: ARIMA(1, 1, 1). **Since AIC and BIC agreed on the best model, this suggests that further exploration of ARIMA models (e.g., adding more AR or MA terms) would likely lead to overfitting without significant improvement in model accuracy.**

Upon analyzing the residuals from the selected ARIMA(1, 1, 1) model, it became clear that some autocorrelation remains, indicating that the model did not fully capture all the dependencies in the data. Despite this, adding additional AR or MA terms would not necessarily address these issues, as the model selection process already chose the best possible model based on statistical criteria.

Given that the series has already been differenced to achieve stationarity, there is no need for further differencing.

ARIMAX Model

ARIMAX (AutoRegressive Integrated Moving Average with Exogenous Variables) extends ARIMA by incorporating external predictors. It is typically used when external factors significantly influence the target variable beyond autoregressive and moving average components.

Purpose: Introduced ARIMAX to assess whether adding external variables (filtered via VIF to avoid collinearity issues with computations) could enhance the model's performance by addressing volatility or autocorrelation in the differenced close data. Given that ARIMA(1,1,1) residuals displayed volatility spikes, the aim is to see if exogenous variables could help.

Thus, I will proceed by exploring more advanced models to capture the remaining patterns in the data, rather than trying additional ARIMA variations.

In [ ]:
```
ro.r(''' # Ensure differenced_close_ts and x_arimax cover the same time period

# Slice differenced_close_ts to match training data (assuming it's time-series

differenced_close_train <- differenced_close[1:nrow(transformed_df)]  # Adjust

# Create the x_arimax matrix using only the predictors filtered from VIF analys
# Variables left after VIF filtering were: log_Volume, cube_volume_adi, cube_vo
# Calculate correlation matrix for the selected variables
cor_matrix <- cor(transformed_df[, c("log_Volume", "cube_volume_obv", "volume_v

# Print the correlation matrix
print(cor_matrix)

# Given the correlation matrix, I tried log_Volume + cube_momentum_ao, and comp
x_arimax <- model.matrix(log_Close ~ log_Volume - 1, transformed_df)

# Fit the ARIMAX model with the selected predictors (exclude intercept)
```

```
arimax_model <- arima(differenced_close_train, order = c(1, 1, 1), xreg = x_ar

# Summarize the ARIMAX model
print(summary(arimax_model))

# Extract residuals from the ARIMAX model
residuals_arimax <- residuals(arimax_model)

# Plot residuals from ARIMAX model
png(filename = "residuals_arimax_regularized.png", width = 800, height = 600)
plot(residuals_arimax, main="Residuals from ARIMAX(1, 1, 1)", ylab="Residuals"
dev.off()
''')

# Display the residuals plot
display_r_plot("residuals_arimax_regularized.png")

ro.r('''
# Check the ACF of the residuals to check for any autocorrelation
png(filename = "acf_residuals_arimax_regularized.png", width = 800, height = 6(
acf(residuals_arimax, main="ACF of Residuals from ARIMAX(1, 1, 1)")
dev.off()
''')
# Display the ACF plot
display_r_plot("acf_residuals_arimax_regularized.png")
```

In [ ]:
```
ro.r(''' # Perform Ljung-Box test to check for autocorrelation in residuals
ljung_box_test_arimax <- Box.test(residuals_arimax, lag = 10, type = "Ljung-Box
print(ljung_box_test_arimax)
''')
```

ARIMAX Model Analysis and Conclusion

Why ARIMAX Did Not Improve Performance

After establishing that the **ARIMA(1,1,1)** model adequately handled stationarity by ensuring a zero-mean and constant variance, I introduced the **ARIMAX** model to assess whether adding external variables (specifically VIF-filtered variables) could improve the model's performance. However, the ARIMAX model did not lead to meaningful improvements for two primary reasons:

1. **Singularity Problems**: When fitting ARIMAX with multiple predictors or collinear variables, the model encountered singularity issues. Singularity occurs when predictors are highly collinear, leading to unreliable coefficient estimates. In this case, the external variables added little new information beyond what the ARIMA model already captured.

2. **Volatility Clustering**: The ARIMAX residuals still showed signs of autocorrelation, indicating unresolved volatility clustering. ARIMAX models are designed to capture mean shifts due to external variables, but they are not equipped to address time-varying volatility, which was evident in the data.

Conclusion and Next Steps

The ARIMAX model did not improve performance because the core issue lies in volatility clustering rather than mean shifts. Since ARIMAX is intended to account for changes in the mean from external shocks, it did not address the volatility in the residuals. Thus, there is no justification for testing additional exogenous variables at this stage.

Instead, I will shift focus to more advanced models designed for volatility, specifically **GARCH** and **EGARCH**, which are better suited for handling time-varying volatility. These models should resolve the remaining issues observed in the residuals.

Justification for Not Testing Additional Variables

- **Volatility Clustering**: A **Box-Ljung Test** returned a p-value of **6.213e-07**, confirming that autocorrelation is still present in the residuals. This indicates that the primary issue is volatility, not mean shifts, making it inefficient to continue testing more external variables.
- **Singularity**: The singularity problems further support that the added predictors do not provide enough new information to justify additional testing with ARIMAX.

Note: I will revisit ARIMAX using the **VIX** as an external variable, but for now, the focus remains on volatility modeling using **GARCH** and **EGARCH**.

Transition to GARCH/EGARCH

Given the presence of **volatility clustering**, I will now transition to **GARCH** and **EGARCH** models. These models are designed to capture **time-varying volatility**, which ARIMA and ARIMAX failed to address. Financial data often exhibit clusters of volatility that cannot be explained by shifts in the mean alone, making these models particularly suitable for this analysis.

Why GARCH and EGARCH?

The **GARCH(1,1)** model captures conditional volatility, while the **EGARCH(1,1)** model accounts for **asymmetric effects**, where negative shocks have a larger impact on volatility. These features are essential for modeling financial market behaviors where large price swings tend to cluster.

Approaches to Handling Mean and Variance

1. **GARCH(1,1) without a mean model**: The mean and variance are handled separately, with ARIMA addressing the mean structure and GARCH modeling the volatility.
2. **GARCH(1,1) with ARMA(1,1)**: This integrates both the mean and variance components into one model, capturing both autocorrelation and volatility within a unified framework.

By testing these models and their variations (with Normal and Student-t distributions), I aim to identify the most effective model for capturing both mean and volatility dynamics in the dataset.

In [ ]:
```
ro.r(''' # GARCH modeling
# Step 1: Define GARCH(1,1) with no mean model specification
garch_spec <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(0, 0), include.mean = FALSE), # No ARMA mean
  distribution.model = "norm"
)

# Step 2: Fit the GARCH(1,1) model using the differenced log close prices (tra
garch_fit <- ugarchfit(spec = garch_spec, data = differenced_close_train)

# Step 3: Print the summary of the GARCH(1,1) model
print(garch_fit)

# Step 4: Extract residuals for diagnostics
garch_residuals <- residuals(garch_fit, standardize = TRUE)

# Convert residuals to a time series object
garch_residuals_ts <- ts(garch_residuals, start = c(2009, 1), frequency = 252)

# Step 5: Plot GARCH residuals with the correct time index
png(filename = "garch_residuals_fixed.png", width = 800, height = 600)
plot(garch_residuals_ts, main="Residuals from GARCH(1,1)", ylab="Standardized R
dev.off()

# Step 6: ACF of GARCH residuals
png(filename = "garch_acf_residuals_fixed.png", width = 800, height = 600)
acf(garch_residuals_ts, main="ACF of GARCH(1,1) Residuals")
dev.off()
''')
# Display the residuals plot
display_r_plot("garch_residuals_fixed.png")

# Display the ACF plot
display_r_plot("garch_acf_residuals_fixed.png")

ro.r('''
# Step 1: Define GARCH(1,1) model with ARMA(1,1) mean model specification
garch_spec_arma <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(1, 1), include.mean = TRUE),  # ARMA(1,1) + (
  distribution.model = "norm"
)

# Step 2: Fit ARMA(1,1) + GARCH(1,1) model using the differenced data, so it is
garch_fit_arma <- ugarchfit(spec = garch_spec_arma, data = differenced_close_t

# Step 3: Print the summary of the ARMA(1,1) + GARCH(1,1) model
print(garch_fit_arma)

''')
ro.r('''
# Step 4: Extract residuals for diagnostics
arma_garch_residuals <- residuals(garch_fit_arma, standardize = TRUE)
```

```
# Convert residuals to a time series object (adjust dates as needed)
arma_garch_residuals_ts <- ts(arma_garch_residuals, start = c(2009, 1), freque

# Step 5: Plot ARMA(1,1) + GARCH(1,1) residuals
png(filename = "arma_garch_residuals.png", width = 800, height = 600)
plot(arma_garch_residuals_ts, main="Residuals from ARMA(1,1) + GARCH(1,1)", yla
dev.off()
''')
display_r_plot('arma_garch_residuals.png')


ro.r('''
# Step 6: ACF of ARMA(1,1) + GARCH(1,1) residuals
png(filename = "arma_garch_acf_residuals.png", width = 800, height = 600)
acf(arma_garch_residuals_ts, main="ACF of ARMA(1,1) + GARCH(1,1) Residuals")
dev.off()

''')
display_r_plot('arma_garch_acf_residuals.png')
```

In [ ]:
```
ro.r(''' # Autocorrelation significance test
# Perform Ljung-Box test to check for autocorrelation in GARCH(1,1) residuals
ljung_box_test_garch <- Box.test(garch_residuals, lag = 10, type = "Ljung-Box"
print(ljung_box_test_garch)
''')

ro.r('''
# Perform Ljung-Box test to check for autocorrelation in ARMA(1,1) + GARCH(1,1
ljung_box_test_arma_garch <- Box.test(arma_garch_residuals, lag = 10, type = "L
print(ljung_box_test_arma_garch)
''')
```

In [ ]:
```
ro.r(''' # Normal vs std distribution
# Model 1: GARCH(1,1) + ARMA(1,1) with normal distribution
garch_spec_normal <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(1, 1), include.mean = TRUE),  # ARMA(1,1)
  distribution.model = "norm"
)

garch_fit_normal <- ugarchfit(spec = garch_spec_normal, data = differenced_clos
aic_normal <- infocriteria(garch_fit_normal)["Akaike",]
bic_normal <- infocriteria(garch_fit_normal)["Bayes",]
print(paste("AIC (Normal):", aic_normal))
print(paste("BIC (Normal):", bic_normal))


# Model 2: GARCH(1,1) + ARMA(1,1) with Student-t distribution
garch_spec_student <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(1, 1), include.mean = TRUE),  # ARMA(1,1)
  distribution.model = "std"  # Student-t distribution
)

garch_fit_student <- ugarchfit(spec = garch_spec_student, data = differenced_cl
aic_student <- infocriteria(garch_fit_student)["Akaike",]
bic_student <- infocriteria(garch_fit_student)["Bayes",]
print(paste("AIC (Student-t):", aic_student))
```

```
print(paste("BIC (Student-t):", bic_student))

# Model 3: GARCH(1,1) with no mean model (Normal distribution)
garch_spec_no_mean_normal <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(0, 0), include.mean = FALSE),  # No mean mode
  distribution.model = "norm"
)

garch_fit_no_mean_normal <- ugarchfit(spec = garch_spec_no_mean_normal, data =
aic_no_mean_normal <- infocriteria(garch_fit_no_mean_normal)["Akaike",]
bic_no_mean_normal <- infocriteria(garch_fit_no_mean_normal)["Bayes",]
print(paste("AIC (No Mean, Normal):", aic_no_mean_normal))
print(paste("BIC (No Mean, Normal):", bic_no_mean_normal))

# Model 4: GARCH(1,1) with no mean model (Student-t distribution)
garch_spec_no_mean_student <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(0, 0), include.mean = FALSE),  # No mean mode
  distribution.model = "std"
)

garch_fit_no_mean_student <- ugarchfit(spec = garch_spec_no_mean_student, data
aic_no_mean_student <- infocriteria(garch_fit_no_mean_student)["Akaike",]
bic_no_mean_student <- infocriteria(garch_fit_no_mean_student)["Bayes",]
print(paste("AIC (No Mean, Student-t):", aic_no_mean_student))
print(paste("BIC (No Mean, Student-t):", bic_no_mean_student))


# Model 5: EGARCH(1,1) with Normal distribution
egarch_spec_normal <- ugarchspec(
  variance.model = list(model = "eGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(1, 1), include.mean = TRUE),  # ARMA(1,1)
  distribution.model = "norm"
)

egarch_fit_normal <- ugarchfit(spec = egarch_spec_normal, data = differenced_c
aic_egarch_normal <- infocriteria(egarch_fit_normal)["Akaike",]
bic_egarch_normal <- infocriteria(egarch_fit_normal)["Bayes",]
print(paste("AIC (EGARCH Normal):", aic_egarch_normal))
print(paste("BIC (EGARCH Normal):", bic_egarch_normal))

# Model 6: EGARCH(1,1) with Student-t distribution
egarch_spec_student <- ugarchspec(
  variance.model = list(model = "eGARCH", garchOrder = c(1, 1)),
  mean.model = list(armaOrder = c(1, 1), include.mean = TRUE),  # ARMA(1,1)
  distribution.model = "std"
)

egarch_fit_student <- ugarchfit(spec = egarch_spec_student, data = differenced_
aic_egarch_student <- infocriteria(egarch_fit_student)["Akaike",]
bic_egarch_student <- infocriteria(egarch_fit_student)["Bayes",]
print(paste("AIC (EGARCH Student-t):", aic_egarch_student))
print(paste("BIC (EGARCH Student-t):", bic_egarch_student))
    ''')
```

In [ ]:

```
ro.r(''' # Extract residuals for each model
garch_residuals_normal <- residuals(garch_fit_normal, standardize = TRUE)
garch_residuals_student <- residuals(garch_fit_student, standardize = TRUE)
garch_residuals_no_mean_student <- residuals(garch_fit_no_mean_student, standa
```

```
garch_residuals_no_mean_normal <- residuals(garch_fit_no_mean_normal, standardi
garch_residuals_student <- residuals(garch_fit_student, standardize = TRUE)
egarch_residuals_normal <- residuals(egarch_fit_normal, standardize = TRUE)
egarch_residuals_student <- residuals(egarch_fit_student, standardize = TRUE)


# Perform Ljung-Box test for all models

ljung_box_garch_normal <- Box.test(garch_residuals_normal, lag = 10, type = "Lj
print(ljung_box_garch_normal)

ljung_box_garch_student <- Box.test(garch_residuals_student, lag = 10, type = '
print(ljung_box_garch_student)

ljung_box_garch_normal <- Box.test(garch_residuals_no_mean_normal, lag = 10, ty
print(ljung_box_garch_normal)

ljung_box_garch_student <- Box.test(garch_residuals_no_mean_student, lag = 10,
print(ljung_box_garch_student)

ljung_box_egarch_normal <- Box.test(egarch_residuals_normal, lag = 10, type = '
print(ljung_box_egarch_normal)

ljung_box_egarch_student <- Box.test(egarch_residuals_student, lag = 10, type =
print(ljung_box_egarch_student)
''')
```

Advanced Time Series Analysis (TSA): GARCH and EGARCH Models

Summary of GARCH/EGARCH Process

After conducting ARIMA and ARIMAX modeling, I found that **volatility clustering** remained a core issue that those models could not address. Thus, I moved forward with **GARCH** and **EGARCH** models to better handle time-varying volatility in the data.

GARCH and EGARCH Models Tested

To fully capture both mean and volatility, I tested several GARCH and EGARCH variants. Some models incorporated an **ARMA(1,1)** mean model, while others did not.

- **Models with ARMA(1,1)**: These models use an **ARIMA(1,1,1)** framework to handle autocorrelation and differencing of the series.
- **Models without ARMA(1,1)**: These models are purely volatility-focused, meaning they do not include a mean structure like ARIMA.

Here are the six models tested:

1. **GARCH(1,1) with Normal Distribution**: Includes **ARMA(1,1)**.
2. **GARCH(1,1) with Student-t Distribution**: Includes **ARMA(1,1)**.
3. **GARCH(1,1) with No Mean Model (Normal Distribution)**: No ARMA component.
4. **GARCH(1,1) with No Mean Model (Student-t Distribution)**: No ARMA component.

5. **EGARCH(1,1) with Normal Distribution**: Includes **ARMA(1,1)**.
6. **EGARCH(1,1) with Student-t Distribution**: Includes **ARMA(1,1)**.

Model Selection Criteria: AIC and BIC

Model	AIC	BIC
GARCH(1,1) with Normal	-5.3826	-5.3697
GARCH(1,1) with Student-t	-5.4908	-5.4758
GARCH(1,1) No Mean, Normal	-5.3709	-5.3645
GARCH(1,1) No Mean, Student-t	-5.4792	-5.4707
EGARCH(1,1) with Normal	-5.4152	-5.4002
EGARCH(1,1) with Student-t	**-5.5138**	**-5.4967**

Best Model Based on AIC and BIC:

- **Best Model based on AIC**: **EGARCH(1,1) with Student-t distribution**.
- **Best Model based on BIC**: **EGARCH(1,1) with Student-t distribution**.

Box-Ljung Test Results for Residual Autocorrelation

Model	Box-Ljung Test (p-value)
GARCH(1,1) with Normal	0.1418
GARCH(1,1) with Student-t	0.1473
GARCH(1,1) No Mean, Normal	0.1432
GARCH(1,1) No Mean, Student-t	0.1522
EGARCH(1,1) with Normal	0.131
EGARCH(1,1) with Student-t	**0.1572**

Interpretation:

- All models have **p-values > 0.05**, suggesting to **fail to reject the null hypothesis** that there is no autocorrelation in the residuals. This indicates that all models adequately captured the volatility in the data.
- **EGARCH(1,1) with Student-t distribution** produced the best AIC, BIC, and passed the Box-Ljung test, making it the most appropriate model for this dataset.

Why EGARCH(1,1) with Student-t Distribution is the Best Model

- **EGARCH (Exponential GARCH)**: This model can capture **asymmetric effects**, where negative shocks tend to increase volatility more than positive shocks.
- **Student-t Distribution**: Its heavy tails better capture large movements in stock prices than the normal distribution.

- **ARIMA(1,1,1) + EGARCH(1,1)**: By incorporating the **ARIMA(1,1,1)** model, I addressed the autocorrelation and stationarity in the mean, while **EGARCH(1,1)** captured the time-varying volatility.

- **Conclusion**: The **EGARCH(1,1) with Student-t distribution** model provides the best fit based on statistical criteria (AIC, BIC) and residual diagnostics, effectively handling both the mean structure and volatility clustering in the AAPL data.

Integration of EGARCH Model into the Predictive Framework

Having identified the **EGARCH(1,1) with Student-t distribution** model as the best-performing time-series model, I am now incorporating its volatility predictions into the regression framework. The goal is to enhance the performance of both robust and regularized regression methods by accounting for the time-varying volatility captured by EGARCH. This step will allow the model to handle not just the mean structure, but also the heteroscedasticity inherent in the data, improving predictive accuracy.

In [ ]:
```
ro.r(''' # Integrate EGARCH into regression models
set.seed(123)
# Step 1: Generate Volatility Predictions from EGARCH(1,1) with Student-t dist
egarch_volatility <- sigma(egarch_fit_student)  # Extract the volatility from

# Add the volatility predictions as a new column to the training dataset
transformed_df$egarch_volatility <- egarch_volatility

# Step 2: Update Robust Regression Models with EGARCH Volatility
# Fit robust regression model
robust_model_egarch <- rlm(log_Close ~ . + egarch_volatility, data = transform

# Predict on the test data with added volatility feature
transformed_test_df$egarch_volatility <- sigma(egarch_fit_student)[1:nrow(trans
robust_preds_egarch <- predict(robust_model_egarch, newdata = transformed_test_

# Calculate MSPE for robust model with EGARCH volatility
robust_mspe_egarch <- mean((test_y - robust_preds_egarch)^2)
cat("EGARCH-Enhanced Huber Regression MSPE:", robust_mspe_egarch, "\n")

# Step 3: Update QR and LTS Models with EGARCH Volatility

# Fit Quantile Regression (QR) model with EGARCH volatility at tau = 0.5 (media
qr_model_egarch <- rq(log_Close ~ . + egarch_volatility, data = transformed_df

# Predict on the test data for QR model
qr_preds_egarch <- predict(qr_model_egarch, newdata = transformed_test_df)

# Calculate MSPE for QR model with EGARCH volatility
qr_mspe_egarch <- mean((test_y - qr_preds_egarch)^2)
cat("EGARCH-Enhanced Quantile Regression MSPE:", qr_mspe_egarch, "\n")


# Fit Least Trimmed Squares (LTS) model with EGARCH volatility
lts_model_egarch <- lqs(log_Close ~ . + egarch_volatility, data = transformed_
```

```
# Predict on the test data for LTS model
lts_preds_egarch <- predict(lts_model_egarch, newdata = transformed_test_df)

# Calculate MSPE for LTS model with EGARCH volatility
lts_mspe_egarch <- mean((test_y - lts_preds_egarch)^2)
cat("EGARCH-Enhanced Least Trimmed Squares Regression MSPE:", lts_mspe_egarch,


# Step 3: Update Regularized Regression Models with EGARCH Volatility
# Convert predictors and response using model.matrix including EGARCH volatili
x_egarch <- model.matrix(log_Close ~ . + egarch_volatility, transformed_df)[, -
y <- transformed_df$log_Close  # Response variable

# Fit Ridge, Lasso, and ElasticNet models with EGARCH volatility
ridge_model_egarch <- cv.glmnet(x_egarch, y, alpha = 0)
lasso_model_egarch <- cv.glmnet(x_egarch, y, alpha = 1)
elasticnet_model_egarch <- cv.glmnet(x_egarch, y, alpha = 0.5)

# Predict on test data
x_test_egarch <- model.matrix(log_Close ~ . + egarch_volatility, transformed_te
ridge_preds_egarch <- predict(ridge_model_egarch, newx = x_test_egarch)
lasso_preds_egarch <- predict(lasso_model_egarch, newx = x_test_egarch)
elasticnet_preds_egarch <- predict(elasticnet_model_egarch, newx = x_test_egarc

# Calculate MSPE for each regularized regression model with EGARCH volatility
ridge_mspe_egarch <- mean((test_y - ridge_preds_egarch)^2)
lasso_mspe_egarch <- mean((test_y - lasso_preds_egarch)^2)
elasticnet_mspe_egarch <- mean((test_y - elasticnet_preds_egarch)^2)

cat("EGARCH-Enhanced Ridge Regression MSPE:", ridge_mspe_egarch, "\n")
cat("EGARCH-Enhanced Lasso Regression MSPE:", lasso_mspe_egarch, "\n")
cat("EGARCH-Enhanced ElasticNet Regression MSPE:", elasticnet_mspe_egarch, "\n"

# Step 4: Update VIF Model and Full Model to include EGARCH Volatility
vif_model_egarch <- lm(log_Close ~ log_Volume + cube_volume_adi + cube_volume_c
full_model_egarch <- lm(log_Close ~ . + egarch_volatility, data = transformed_c

transformed_test_df$egarch_volatility <- sigma(egarch_fit_student)[1:nrow(trans
vif_preds_egarch <- predict(vif_model_egarch, newdata = transformed_test_df)
vif_mspe_egarch <- mean((test_y - vif_preds_egarch)^2)

full_preds_egarch <- predict(full_model_egarch, newdata = transformed_test_df)
full_mspe_egarch <- mean((test_y - full_preds_egarch)^2)
cat("EGARCH-Enhanced VIF Model MSPE:", vif_mspe_egarch, "\n")
cat("EGARCH-Enhanced Full Model MSPE:", full_mspe_egarch, "\n")
''')
```

In [ ]:

```
ro.r(''' # Helper function to apply smearing (bias correction) and calculate MS
calculate_mspe_with_smearing <- function(log_preds, log_true) {
    # Exponentiate the predictions and true values to get them on the original
    pred_original <- exp(log_preds)
    true_original <- exp(log_true)

    # Calculate residuals on the log scale
    residuals_log <- log_true - log_preds

    # Bias correction using the smearing estimator
    CF <- mean(exp(residuals_log))  # Correction factor for bias
    adjusted_preds_original <- pred_original * CF
```

```
    # Calculate MSPE on the original scale
    mspe_original <- mean((true_original - adjusted_preds_original)^2)

    return(mspe_original)
}

# Step 4: Apply smearing for all EGARCH-enhanced models

# Huber Regression with EGARCH Volatility (Original Scale)
robust_mspe_egarch_original <- calculate_mspe_with_smearing(robust_preds_egarcl
cat("EGARCH-Enhanced Huber Regression MSPE (Original Scale):", robust_mspe_ega

# Quantile Regression (QR) with EGARCH Volatility (Original Scale)
qr_mspe_egarch_original <- calculate_mspe_with_smearing(qr_preds_egarch, test_y
cat("EGARCH-Enhanced Quantile Regression MSPE (Original Scale):", qr_mspe_egar

# Least Trimmed Squares (LTS) with EGARCH Volatility (Original Scale)
lts_mspe_egarch_original <- calculate_mspe_with_smearing(lts_preds_egarch, test
cat("EGARCH-Enhanced Least Trimmed Squares Regression MSPE (Original Scale):",

# Ridge Regression with EGARCH Volatility (Original Scale)
ridge_mspe_egarch_original <- calculate_mspe_with_smearing(ridge_preds_egarch,
cat("EGARCH-Enhanced Ridge Regression MSPE (Original Scale):", ridge_mspe_egar

# Lasso Regression with EGARCH Volatility (Original Scale)
lasso_mspe_egarch_original <- calculate_mspe_with_smearing(lasso_preds_egarch,
cat("EGARCH-Enhanced Lasso Regression MSPE (Original Scale):", lasso_mspe_egar

# ElasticNet Regression with EGARCH Volatility (Original Scale)
elasticnet_mspe_egarch_original <- calculate_mspe_with_smearing(elasticnet_pre
cat("EGARCH-Enhanced ElasticNet Regression MSPE (Original Scale):", elasticnet_


vif_mspe_egarch_original <- calculate_mspe_with_smearing(vif_preds_egarch, test
cat("VIF Model with EGARCH Volatility and Smearing MSPE (Original Scale):", vi

# Step 6: Apply smearing bias correction for Full Model with EGARCH Volatility
full_model_mspe_egarch_original <- calculate_mspe_with_smearing(full_preds_ega
cat("Full Model with EGARCH Volatility and Smearing MSPE (Original Scale):", fu

    ''')
```

Analysis of EGARCH-Enhanced Regression Methods

Overview

This analysis applied robust regression methods (Quantile Regression, Huber Regression, and Least Trimmed Squares), regularized regression methods (Ridge, Lasso, and Elastic Net), and EGARCH-enhanced versions of these models. The objective was to assess whether incorporating EGARCH volatility into the models would improve predictive performance by addressing volatility and outliers in financial data.

Results

EGARCH-Enhanced Regression Models

Model	MSPE (Log Scale)	MSPE (Original Scale)
EGARCH-Enhanced Huber Regression	4.948813e-05	1.097161
EGARCH-Enhanced Quantile Regression	5.062403e-05	1.083868
EGARCH-Enhanced Least Trimmed Squares Regression	0.0001883546	4.45806
EGARCH-Enhanced Ridge Regression	0.000681377	15.3242
EGARCH-Enhanced Lasso Regression	0.002135133	4.930664
EGARCH-Enhanced ElasticNet Regression	0.0008286128	8.247073
EGARCH-Enhanced VIF Model	0.1123632	4630.846
EGARCH-Enhanced Full Model	6.048988e-05	1.335532

Standard Robust and Regularized Regression Methods

Model	MSPE (Log Scale)	MSPE (Original Scale)
Quantile Regression	3.6021e-05	0.8046161
Huber Regression	3.789034e-05	0.8522115
Penalized Quantile Regression	7.939872e-05	1.758258
Least Trimmed Squares Regression	0.0007393806	15.15739

Regularized Regression Methods

Model	MSPE (Log Scale)	MSPE (Original Scale)
Ridge Regression	0.0006943541	16.37199
Lasso Regression	0.002135133	4.930664
Elastic Net Regression	0.002135133	4.930664
VIF Model	0.08338693	3076.539
Full Model	4.225889e-05	0.9543641

Interpretation of Results

General Observations

1. **Increased MSPE Across Most Models**: The EGARCH-enhanced models generally performed worse than their original versions, with higher MSPE on both the log and original scales. This can be attributed to the introduction of volatility features that added complexity to the models, which were not well-suited to handle the nonlinear nature of time-varying volatility.

2. **Robust Methods with EGARCH**: Both **Quantile Regression** and **Huber Regression** showed minimal increases in MSPE after incorporating EGARCH volatility. The results indicate that these models, which are inherently robust to outliers, were not significantly affected by volatility adjustments. However, their predictive accuracy did not substantially improve either, suggesting that the EGARCH volatility did not provide substantial new information.

3. **Improvement in Least Trimmed Squares (LTS)**: Interestingly, **Least Trimmed Squares Regression** showed improvement after introducing EGARCH volatility, reducing its MSPE from **15.15739 (Original Scale)** to **4.45806**. This improvement suggests that LTS, which minimizes the influence of extreme values, benefited from the EGARCH volatility's ability to account for large swings in the data, reducing the impact of outliers.

Regularized Regression Methods

1. **Ridge Regression with EGARCH**: **Ridge Regression** showed a slight improvement after incorporating EGARCH, with the MSPE improving on the log scale from **0.0006943541** to **0.000681377**. However, the improvement was not as significant on the original scale, indicating that while Ridge Regression handled multicollinearity effectively, it did not fully capitalize on the volatility captured by EGARCH.

2. **Lasso Regression**: **Lasso Regression** remained stable with no significant change after the EGARCH adjustment, maintaining an MSPE of **4.930664** on both scales. This shows that Lasso was largely unaffected by the volatility adjustment, as the sparsity enforced by the L1 penalty helped retain its predictive performance.

3. **Elastic Net Regression Got Worse**: The **Elastic Net Regression** model's performance deteriorated with EGARCH volatility, seeing an increase in MSPE from **4.930664** to **8.247073**. This suggests that the combination of L1 and L2 penalties could not balance the added complexity from the volatility, leading to a degradation in prediction accuracy.

4. **VIF Model and Full Model with EGARCH**: The **VIF Model** performed poorly even with EGARCH volatility, with an extremely high MSPE of **4630.846 (Original Scale)**. This likely resulted from the model’s inability to handle the complexity introduced by both multicollinearity and volatility, leading to poor predictions. The **Full Model**, though not as poor, also showed an increase in MSPE from **0.9543641 (Original Scale)** to **1.335532**, indicating a deterioration in performance when EGARCH volatility was introduced.

Mathematical and Theoretical Reasoning

1. **Nonlinear Complexity**: EGARCH models are specifically designed to capture time-varying volatility, which is a non-linear characteristic of financial data. However, the

regularized regression models (Ridge, Lasso, Elastic Net) are linear models, which means they struggle to incorporate and benefit from the volatility features introduced by EGARCH. This disconnect between the linear nature of the models and the nonlinear characteristics of the data likely led to increased MSPE.

2. **Smearing Bias Correction**: The application of smearing bias correction further exposed the limitations of these models in handling volatility. The correction for bias due to log transformation revealed that the models were consistently underperforming when predicting the original (untransformed) values of stock prices.

3. **Impact of EGARCH on Robust Methods**: While robust methods like Huber and Quantile Regression are designed to minimize the influence of extreme values, the introduction of EGARCH volatility did not provide significant new information to enhance their performance. The nature of these models already focuses on handling irregularities in the data, and thus the volatility adjustment did not substantially improve their predictions.

4. **Improvement in LTS**: The significant improvement in **Least Trimmed Squares Regression** after incorporating EGARCH volatility suggests that volatility features helped reduce the model's sensitivity to extreme outliers, making it more stable and reliable in handling the financial data's inherent volatility.

Conclusions

1. **EGARCH Volatility Did Not Enhance Regularized Regression**: Despite the complexity introduced by EGARCH volatility, the regularized methods (Lasso, Elastic Net, and Ridge) did not benefit substantially, with only marginal improvements in Ridge Regression. The primary reason lies in the disconnect between the non-linear nature of volatility and the linear structure of regularized regression models.

2. **Robust Methods Remain Strong Contenders**: Quantile Regression and Huber Regression maintained their strength even after introducing EGARCH volatility, but they did not show significant improvement. Least Trimmed Squares (LTS) saw the most notable improvement after EGARCH was introduced, highlighting its potential for handling volatility-driven data.

3. **Practical Implications**: The results suggest that while EGARCH is highly effective for volatility modeling, its integration into linear regression models may not yield substantial benefits unless the model structure itself is adapted to handle non-linearities. Financial models that directly address volatility through methods such as LTS or Ridge Regression are more suited to capture the characteristics of the data when combined with volatility features.

Practical Implications and Future Directions

The analysis demonstrated that the integration of **EGARCH volatility** into the regression framework did not yield significant improvements in most regularized methods. Despite the ability of the **EGARCH** model to capture volatility clustering, the linear nature of **Ridge**, **Lasso**, and **Elastic Net** models seems to have struggled to leverage the non-linear volatility insights provided by the EGARCH model.

However, there were **notable improvements** in **Ridge** and **Least Trimmed Squares** models, showcasing that **EGARCH-enhanced volatility** can enhance models that are more sensitive to outliers or multicollinearity. This insight suggests that future work should focus on models capable of handling non-linear relationships more effectively.

In Progress / To Be Completed

Residual Diagnostics and Error Distribution Analysis (In Progress)

- **Objective**: Evaluate the fit of residuals from all models to determine if further transformations or adjustments are needed.
- **Focus**: Explore **ARCH** effects in residuals, and conduct **Q-Q plots** to assess normality and the fit of residuals.

Non-Parametric and Machine Learning Methods (Next Step)

- **Objective**: Given the limited success of linear regression methods, I plan to implement **non-parametric and machine learning models** like **Random Forests** and **Gradient Boosting Machines (GBM)** to capture the non-linear effects present in financial data.
- **Reasoning**: Machine learning methods are better suited for capturing complex relationships and interactions between features that EGARCH volatility introduces but linear models fail to exploit fully.

Ensemble Methods for Model Combination (Planned)

- **Objective**: Combine the best performing models (e.g., **Quantile Regression** and **LTS**) with non-parametric methods to enhance predictive accuracy and stability.
- **Focus**: Utilize techniques like **stacking** and **blending** to leverage the strengths of various models. This approach can create a more balanced predictive strategy that incorporates linear, non-linear, and volatility-based insights.

Model Validation and Backtesting (In Progress)

- **Objective**: Rigorously backtest the models using financial metrics such as **Sharpe Ratio** and **Value at Risk (VaR)** to validate their performance in real-world trading conditions.
- **Techniques**: Employ **walk-forward validation** and **rolling window validation** to assess model performance over time, accounting for dynamic market conditions.

Advanced Optimization and Simulation Techniques (Planned)

- **Objective**: Optimize model parameters using advanced methods like **Simulated Annealing** and **Genetic Algorithms** to improve predictive performance.
- **Reasoning**: Current methods did not leverage optimization techniques for tuning parameters beyond grid search, and the non-linear nature of financial markets suggests that advanced optimization could significantly improve results.

Monte Carlo Simulations for Risk and Uncertainty Assessment (Planned)

- **Objective**: Incorporate **Monte Carlo simulations** to assess model uncertainty and stress-test predictions. This step will help quantify risk and improve decision-making strategies.
- **Focus**: Simulations will be run on the final models to assess their robustness across varying market conditions, particularly for extreme events (e.g., market crashes or volatility spikes).

Distribution Fitting and Simulation Methods (In Progress)

- **Objective**: Enhance stock price simulation by using **Kernel Density Estimation** and other non-parametric methods to fit and simulate price distributions more accurately.
- **Reasoning**: While EGARCH captured volatility well, there is a need to better simulate extreme price movements and outliers, which may be achieved through more flexible distribution-fitting techniques.

Takeaways and Next Steps

While the **EGARCH-enhanced models** did not universally improve performance across the board, they provided valuable insights into the complex volatility structures inherent in financial data. The **Ridge** and **LTS models** benefited from the volatility features, while **Elastic Net** and **VIF models** struggled to adapt. This mixed performance suggests that future work should focus on:

1. **Exploring non-linear and machine learning approaches**: Methods like **Random Forests**, **GBM**, and **Neural Networks** can capture complex interactions and non-

linearities better than linear regression models.

2. **Ensemble methods**: Combining the strengths of robust and machine learning models through stacking or blending will provide a more comprehensive predictive framework.

3. **Backtesting and validation**: Backtesting using rolling-window and walk-forward validation is crucial to ensuring robustness in live trading scenarios.

4. **Risk quantification**: Implementing **Monte Carlo simulations** will enhance risk management and provide deeper insights into the range of possible outcomes under varying market conditions.

The progress made so far has laid a strong foundation for further research into creating a highly effective quantitative trading strategy. By integrating these advanced methods and focusing on non-linearity, the next stage of this project will refine and enhance the overall predictive capability.