Problem Set 1: Key Concepts and Implementations

1 Retrieving Stock Prices from Yahoo Finance

We retrieve daily adjusted closing prices for five stocks—Apple (AAPL), Microsoft (MSFT), Tesla (TSLA), Amazon (AMZN), and Google (GOOGL)—from January 1, 2020, to December 31, 2024, using the `yfinance` library. Required libraries are: `yfinance`, `pandas`, `numpy`, `scipy.stats`, `matplotlib.pyplot`, `statsmodels.api`, and `scipy.optimize`.

1.1 Python Code

```
import yfinance as yf
import pandas as pd
import numpy as np
from scipy.stats import jarque_bera
import matplotlib.pyplot as plt
import statsmodels.api as sm
from scipy.optimize import minimize

# Define tickers and date range
tickers = ['AAPL', 'MSFT', 'TSLA', 'AMZN', 'GOOGL']
start_date = '2020-01-01'
end_date = '2024-12-31'

# Download adjusted closing prices
data = yf.download(tickers, start=start_date, end=end_date)['Adj
    Close']

# Calculate daily returns
returns = data.pct_change().dropna()

# Save to CSV for reproducibility
returns.to_csv('stock_returns.csv')
```

1.2 Code Explanation

- `tickers` defines the list of stock symbols, ensuring consistency with AAPL, MSFT, TSLA, AMZN, and GOOGL across all analyses.
- `yf.download(tickers, start=start_date, end=end_date)['Adj Close']` fetches adjusted closing prices from Yahoo Finance, accounting for dividends and splits. The result is a `pandas` DataFrame with dates as rows and stocks as columns.
- `pct_change()` computes daily returns as $(P_t - P_{t-1})/P_{t-1}$, where P_t is the price at time t, producing a DataFrame of percentage changes.
- `dropna()` removes rows with missing values (e.g., due to holidays), ensuring a clean dataset.

- `returns.to_csv('stock_returns.csv')` saves the returns to a CSV file, enabling reproducibility and verification of the input data.

1.3 Example

The code creates a DataFrame of daily returns for the five stocks, showing their price changes over the period. For instance, it calculates the percentage change in AAPL's adjusted closing price from one day to the next, reflecting its performance. For a volatile stock like TSLA, the returns may show larger fluctuations, while MSFT might exhibit steadier changes. This DataFrame, saved as a CSV, provides the foundational data for all subsequent analyses, enabling tests of statistical properties, portfolio construction, and performance evaluation.

IMPORTANT NOTE:

If this approach did not work, use an alternative!

1.4 Python Code

```
!pip install yahooquery
```

1.5 Python Code

```
from yahooquery import Ticker
import pandas as pd
import numpy as np
from scipy.stats import jarque_bera
import matplotlib.pyplot as plt
import statsmodels.api as sm
from scipy.optimize import minimize

tickers = ['AAPL', 'MSFT', 'TSLA', 'AMZN', 'GOOGL']
start_date = '2020-01-01'
end_date = '2024-12-31'

# Create ticker object (can handle multiple tickers)
t = Ticker(tickers)

# Get historical prices
hist = t.history(start=start_date, end=end_date)

# Reformat the data
# Multi-index to regular DataFrame: one column per ticker
adj_close = hist['adjclose'].unstack(level=0)
adj_close.index.name = 'Date'

# Preview and save
print(adj_close.head())
adj_close.to_csv("stock_adj_close_yahooquery.csv")

# Calculate daily returns
returns = adj_close.pct_change().dropna()
```

2 Jarque-Bera Test for Normality

The Jarque-Bera (JB) test assesses whether stock returns follow a normal distribution, a critical assumption in portfolio optimization.

2.1 Mathematical Formulation

The JB test statistic is:

$$\text{JB} = \frac{n}{6}\left(S^2 + \frac{(K-3)^2}{4}\right) \tag{1}$$

where:

- n: number of observations (daily returns).
- $S = \frac{\hat{\mu}_3}{\hat{\sigma}^3}$: skewness, with $\hat{\mu}_3 = \frac{1}{n}\sum_{i=1}^{n}(r_i - \bar{r})^3$ (third central moment) and $\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2}$ (standard deviation).
- $K = \frac{\hat{\mu}_4}{\hat{\sigma}^4}$: kurtosis, with $\hat{\mu}_4 = \frac{1}{n}\sum_{i=1}^{n}(r_i - \bar{r})^4$ (fourth central moment).

S^2 measures asymmetry, and $(K-3)^2/4$ indicates excess kurtosis (normal distribution: $K = 3$). Under the null hypothesis of normality, JB follows a χ^2 distribution with 2 degrees of freedom. A p-value < 0.05 rejects normality.

2.2 Python Implementation

```python
# Jarque-Bera test for each stock
jb_results = {}
for ticker in tickers:
    jb_stat, p_value = jarque_bera(returns[ticker])
    jb_results[ticker] = {'JB Statistic': jb_stat, 'P-value': p_value}

# Display results
jb_df = pd.DataFrame(jb_results).T
print(jb_df)
```

2.3 Code Explanation

- `jb_results = {}` initializes a dictionary to store test results for each stock.
- The loop iterates over `tickers`, applying `jarque_bera(returns[ticker])` to each stock's returns (e.g., `returns['AAPL']`), a Series of daily returns.
- `jarque_bera` computes S, K, and the JB statistic using $n \approx 1258$, returning the statistic and p-value from the χ^2 distribution.
- Results are stored as `{'JB Statistic': jb_stat, 'P-value': p_value}` per ticker.
- `pd.DataFrame(jb_results).T` creates a DataFrame with stocks as rows and JB Statistic, P-value as columns. `print(jb_df)` outputs the table.

2.4 Example

The code applies the JB test to the daily returns of each stock, calculating skewness and kurtosis to determine if their distributions deviate from normality. For a stock like TSLA, the test may detect high kurtosis due to frequent large price swings, indicating fat tails. For a stable stock like MSFT, the returns may show less extreme behavior, but still deviate from normality due to market events. The output DataFrame shows the JB statistic and p-value for each stock, helping assess whether normality assumptions hold for portfolio models, directly reflecting the code's analysis of the returns' statistical properties.

3 Investment Opportunity Set

The investment opportunity set represents all possible risk-return combinations from portfolios of the five stocks.

3.1 Mathematical Formulation

For a portfolio with weights $\mathbf{w} = [w_1, \ldots, w_5]$, the expected return and variance are:

$$\mu_p = \mathbf{w}^T \boldsymbol{\mu} = \sum_{i=1}^{5} w_i \mu_i \tag{2}$$

$$\sigma_p^2 = \mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} = \sum_{i=1}^{5} \sum_{j=1}^{5} w_i w_j \sigma_{ij} \tag{3}$$

where:

- $\boldsymbol{\mu}$: expected returns, $\mu_i = \frac{1}{T} \sum_{t=1}^{T} r_{i,t}$.
- $\boldsymbol{\Sigma}$: covariance matrix, $\sigma_{ij} = \frac{1}{T-1} \sum_{t=1}^{T} (r_{i,t} - \mu_i)(r_{j,t} - \mu_j)$.

The opportunity set is the set of (σ_p, μ_p) for all $\mathbf{w}$ with $\sum w_i = 1$, visualized as a scatter plot.

3.2 Python Implementation

```python
# Calculate mean returns and covariance matrix
mean_returns = returns.mean() * 252  # Annualized
cov_matrix = returns.cov() * 252

# Simulate random portfolios
num_portfolios = 10000
results = np.zeros((3, num_portfolios))
for i in range(num_portfolios):
    weights = np.random.random(len(tickers))
    weights /= np.sum(weights)
    portfolio_return = np.sum(mean_returns * weights)
    portfolio_std = np.sqrt(np.dot(weights.T, np.dot(cov_matrix,
        weights)))
    results[0, i] = portfolio_return
    results[1, i] = portfolio_std
    results[2, i] = portfolio_return / portfolio_std  # Sharpe ratio

# Plot opportunity set
plt.scatter(results[1, :], results[0, :], c=results[2, :],
    cmap='viridis')
```

```python
plt.colorbar(label='Sharpe Ratio')
plt.xlabel('Volatility')
plt.ylabel('Expected Return')
plt.title('Investment Opportunity Set')
plt.savefig('opportunity_set.png')
```

3.3 Code Explanation

- `returns.mean() * 252` annualizes mean returns (e.g., daily 0.0005 becomes 12.6% annually). `returns.cov() * 252` annualizes the covariance matrix.
- `num_portfolios = 10000` specifies the number of random portfolios.
- `np.random.random(len(tickers))` generates five random weights, normalized by `weights /= np.sum(weights)` to sum to 1.
- `portfolio_return = np.sum(mean_returns * weights)` computes μ_p as a weighted sum of stock returns.
- `np.dot(weights.T, np.dot(cov_matrix, weights))` calculates σ_p^2, and `np.sqrt(...)` gives σ_p.
- `results[2, i] = portfolio_return / portfolio_std` computes the Sharpe ratio (risk-free rate assumed 0 for simplicity).
- `plt.scatter` plots volatility vs. return, colored by Sharpe ratio using `viridis`. `plt.savefig` saves the plot.

3.4 Example

The code generates a scatter plot of 10,000 portfolios, each with different weights for the five stocks. It calculates the expected return of each portfolio by weighting the stocks' average returns (e.g., higher weight on TSLA increases return but also risk due to its volatility). The volatility reflects the portfolio's risk, influenced by stocks like TSLA (high variance) or MSFT (lower variance), and their correlations. The plot visualizes the trade-off: portfolios with higher expected returns typically have higher volatility, and those with higher Sharpe ratios (colored yellow) are closer to the efficient frontier, directly showing how the code maps the risk-return landscape.

4 Efficient Frontiers

The efficient frontier comprises portfolios maximizing return for given risk, comparing unconstrained (allowing short-selling) and constrained (no short-selling) cases.

4.1 Mathematical Formulation

The unconstrained frontier solves:

$$\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{\Sigma}\mathbf{w} \quad \text{s.t.} \quad \mathbf{w}^T\boldsymbol{\mu} = \mu_p, \quad \mathbf{w}^T\mathbf{1} = 1 \tag{4}$$

The constrained frontier adds:

$$w_i \geq 0 \quad \forall i \tag{5}$$

Minimizing variance achieves the lowest risk for target return μ_p. Unconstrained allows short-selling ($w_i < 0$), reducing risk via negative correlations, while constrained is practical for no-short-selling investors.

4.2 Python Implementation

```python
def portfolio_volatility(weights, cov_matrix):
    return np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))

def efficient_frontier(mean_returns, cov_matrix, constrained=False):
    n = len(mean_returns)
    returns_range = np.linspace(mean_returns.min(),
        mean_returns.max(), 50)
    efficient_portfolios = []

    for ret in returns_range:
        constraints = [{'type': 'eq', 'fun': lambda w: np.sum(w) - 1},
                       {'type': 'eq', 'fun': lambda w:
                           np.sum(mean_returns * w) - ret}]
        bounds = [(0, 1)] * n if constrained else [(-1, 1)] * n
        result = minimize(portfolio_volatility, np.ones(n)/n,
            args=(cov_matrix,),
                          method='SLSQP', bounds=bounds,
                              constraints=constraints)
        efficient_portfolios.append(result['fun'])

    return returns_range, efficient_portfolios

# Compute frontiers
returns_unc, vols_unc = efficient_frontier(mean_returns, cov_matrix,
    False)
returns_con, vols_con = efficient_frontier(mean_returns, cov_matrix,
    True)

# Plot
plt.plot(vols_unc, returns_unc, 'b-', label='Unconstrained')
plt.plot(vols_con, returns_con, 'r--', label='Constrained')
plt.xlabel('Volatility')
plt.ylabel('Expected Return')
plt.title('Efficient Frontiers')
plt.legend()
plt.savefig('efficient_frontier.png')
```

4.3 Code Explanation

- `portfolio_volatility` computes $\sigma_p = \sqrt{\mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w}}$ using matrix operations.
- `efficient_frontier` generates 50 target returns between `mean_returns.min()` and `max()`.
- Constraints enforce $\sum w_i = 1$ and $\mathbf{w}^T \boldsymbol{\mu} = \mu_p$. `bounds` are $[0, 1]$ (constrained) or $[-1, 1]$ (unconstrained).
- `minimize` uses SLSQP, starting with equal weights, to find minimum volatility for each target return.
- `returns_unc, vols_unc` and `returns_con, vols_con` store frontier points, plotted with blue solid (unconstrained) and red dashed (constrained) lines.

4.4 Example

The code identifies the minimum-risk portfolios for a range of target returns by optimizing weights for the five stocks. In the unconstrained case, it may assign negative weights to stocks like TSLA to reduce risk by exploiting correlations (e.g., if TSLA moves opposite to AAPL). In the constrained case, it only uses positive weights, potentially increasing risk for the same return. The plot shows the unconstrained frontier below the constrained one, illustrating how the code's optimization finds superior risk-return combinations when short-selling is allowed, directly highlighting the concept of efficiency in portfolio selection.

5 Tangency Portfolio and Optimal Complete Portfolio

The tangency portfolio maximizes the Sharpe ratio, and the optimal complete portfolio combines it with a risk-free asset.

5.1 Mathematical Formulation

The Sharpe ratio is:

$$\text{Sharpe Ratio} = \frac{\mathbf{w}^T \boldsymbol{\mu} - r_f}{\sqrt{\mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w}}} \tag{6}$$

The tangency portfolio maximizes this, subject to $\mathbf{w}^T \mathbf{1} = 1$. The optimal complete portfolio allocates:

$$w_m = \frac{\mu_m - r_f}{A \sigma_m^2} \tag{7}$$

where $\mu_m = \mathbf{w}_m^T \boldsymbol{\mu}$, $\sigma_m = \sqrt{\mathbf{w}_m^T \boldsymbol{\Sigma} \mathbf{w}_m}$, A is risk aversion, and $r_f = 0.01$.

5.2 Python Implementation

```python
def neg_sharpe_ratio(weights, mean_returns, cov_matrix, rf=0.01):
    p_ret = np.sum(mean_returns * weights)
    p_vol = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
    return -(p_ret - rf) / p_vol

# Find tangency portfolio
constraints = [{'type': 'eq', 'fun': lambda w: np.sum(w) - 1}]
bounds = [(-1, 1)] * len(tickers)
result = minimize(neg_sharpe_ratio, np.ones(len(tickers))/len(tickers),
                  args=(mean_returns, cov_matrix, 0.01),
                     method='SLSQP',
                  bounds=bounds, constraints=constraints)
tangency_weights = result['x']

# Optimal complete portfolio (A=3)
A = 3
mu_m = np.sum(mean_returns * tangency_weights)
sigma_m = np.sqrt(np.dot(tangency_weights.T, np.dot(cov_matrix,
    tangency_weights)))
w_m = (mu_m - 0.01) / (A * sigma_m**2)
print(f"Tangency Weights: {tangency_weights}")
print(f"Optimal Weight in Tangency Portfolio: {w_m}")
```

5.3 Code Explanation

- `neg_sharpe_ratio` computes $\mu_p =$ `np.sum(mean_returns * weights)`, $\sigma_p =$ `np.sqrt(...)`, and returns the negative Sharpe ratio for minimization.
- `minimize` uses SLSQP to maximize the Sharpe ratio, with $\sum w_i = 1$ and bounds $[-1, 1]$.

- `tangency_weights = result['x']` stores the optimized weights.
- `mu_m` and `sigma_m` are computed for the tangency portfolio, and `w_m` is calculated for $A = 3$.
- `print` displays the results.

5.4 Example

The code optimizes weights to maximize the Sharpe ratio, balancing the portfolio's excess return over the risk-free rate against its volatility. For stocks like TSLA, it may assign a small or negative weight to temper risk, while favoring stable stocks like AAPL or MSFT. The resulting `tangency_weights` define a portfolio with the highest risk-adjusted return. The `w_m` calculation determines how much to invest in this portfolio versus the risk-free asset, reflecting risk aversion. This process, driven by the code, illustrates how to construct an optimal risky portfolio and tailor it to an investor's preferences.

6 Decomposing Portfolio Performance

Portfolio return and risk are decomposed into systematic and idiosyncratic components using the CAPM.

6.1 Mathematical Formulation

CAPM for stock i:

$$R_{i,t} - r_f = \alpha_i + \beta_i(R_{m,t} - r_f) + \epsilon_{i,t} \tag{8}$$

Portfolio:

$$\mu_p = r_f + \beta_p(\mu_m - r_f) + \alpha_p \tag{9}$$

$$\sigma_p^2 = \beta_p^2\sigma_m^2 + \sigma_{\epsilon_p}^2 \tag{10}$$

where $\beta_p = \sum w_i\beta_i$, $\alpha_p = \sum w_i\alpha_i$, and $\sigma_{\epsilon_p}^2 = \sigma_p^2 - \beta_p^2\sigma_m^2$.

6.2 Python Implementation

```
# 1. Download S&P 500 data using yahooquery
sp500 = Ticker('^GSPC')
hist = sp500.history(start=start_date, end=end_date).reset_index()
market = hist[hist['symbol'] ==
    '^GSPC'].set_index('date')['adjclose'].pct_change().dropna()

# 2. Align with returns DataFrame
market = market.reindex(returns.index).fillna(0)

# 3. CAPM regression for each stock
betas, alphas = [], []
for ticker in tickers:
    X = sm.add_constant(market)
    Y = returns[ticker] - 0.01 / 252  # daily excess return (risk-free
        = 1% annually)
    model = sm.OLS(Y, X).fit()
    betas.append(model.params.iloc[1])          # fixed: use iloc
        for positional access
    alphas.append(model.params.iloc[0] * 252)   # annualized alpha
        using intercept

```

```
# 4. Portfolio decomposition
portfolio_beta = np.sum(np.array(betas) * tangency_weights)
portfolio_alpha = np.sum(np.array(alphas) * tangency_weights)
systematic_var = portfolio_beta**2 * market.var() * 252

def portfolio_volatility(weights, cov_matrix):
    return np.sqrt(weights @ cov_matrix @ weights)

total_var = portfolio_volatility(tangency_weights, cov_matrix)**2
idiosyncratic_var = total_var - systematic_var

# 5. Output
print(f"Portfolio Beta: {portfolio_beta:.4f}")
print(f"Portfolio Alpha: {portfolio_alpha:.4f}")
print(f"Systematic Variance: {systematic_var:.6f}")
print(f"Idiosyncratic Variance: {idiosyncratic_var:.6f}")
```

6.3 Code Explanation

- `sp500 = Ticker('^ GSPC')`
  Creates a Ticker object for the S&P 500 index using `yahooquery`.

- `hist = sp500.history(start=start_date, end=end_date).reset_index()`
  Downloads historical S&P 500 index data between the specified dates and resets the index for ease of use.

- `market = hist[hist['symbol'] == '^GSPC'].set_index('date')['adjclose'].pct_change().dropna()`
  Filters the data to only include rows for the S&P 500, sets the date as index, selects adjusted closing prices, calculates daily returns:

$$R_m(t) = \frac{P_t - P_{t-1}}{P_{t-1}}$$

  and removes missing values.

- `market = market.reindex(returns.index).fillna(0)`
  Reindexes the market returns to match the stock returns index, filling missing values with 0.

- `betas, alphas = [], []`
  Initializes empty lists to store beta and alpha values for each stock.

- `for ticker in tickers:`
  Begins a loop over each stock ticker in the portfolio.

- `X = sm.add_constant(market)`
  Adds a constant column (value 1) to include an intercept in the regression:

$$X = \begin{bmatrix} 1 & R_m(1) \\ 1 & R_m(2) \\ \vdots & \vdots \end{bmatrix}$$

- `Y = returns[ticker] - 0.01 / 252`
  Computes daily excess return of the stock assuming a 1% annual risk-free rate:

$$R_i(t) - R_f = R_i(t) - \frac{0.01}{252}$$

- `model = sm.OLS(Y, X).fit()`
  Performs Ordinary Least Squares (OLS) regression to estimate:
  $$R_i(t) - R_f = \alpha + \beta(R_m(t) - R_f) + \epsilon_t$$
- `betas.append(model.params.iloc[1])`
  Extracts the estimated β, which measures sensitivity to the market:
  $$\beta = \frac{\mathrm{Cov}(R_i, R_m)}{\mathrm{Var}(R_m)}$$
- `alphas.append(model.params.iloc[0] * 252)`
  Extracts the intercept α, annualized by multiplying daily value by 252:
  $$\alpha_{\text{annual}} = \hat{\alpha}_{\text{daily}} \times 252$$
- `portfolio_beta = np.sum(np.array(betas) * tangency_weights)`
  Computes the portfolio's overall beta:
  $$\beta_p = \sum_i w_i \beta_i$$
- `portfolio_alpha = np.sum(np.array(alphas) * tangency_weights)`
  Computes the portfolio's overall alpha:
  $$\alpha_p = \sum_i w_i \alpha_i$$
- `systematic_var = portfolio_beta**2 * market.var() * 252`
  Calculates the systematic (market-driven) variance of the portfolio:
  $$\text{Systematic Var} = \beta_p^2 \cdot \mathrm{Var}(R_m) \cdot 252$$
- `def portfolio_volatility(weights, cov_matrix):`
  `return np.sqrt(weights @ cov_matrix @ weights)`
  Defines a function to compute portfolio volatility using the covariance matrix:
  $$\sigma_p = \sqrt{w^\top \Sigma w}$$
- `total_var = portfolio_volatility(tangency_weights, cov_matrix)**2`
  Computes total portfolio variance:
  $$\text{Total Var} = \sigma_p^2$$
- `idiosyncratic_var = total_var - systematic_var`
  Residual (idiosyncratic) variance is calculated as:
  $$\text{Idiosyncratic Var} = \text{Total Var} - \text{Systematic Var}$$
- `print(f"Portfolio Beta: {portfolio_beta:.4f}")`
  Displays the computed portfolio beta.
- `print(f"Portfolio Alpha: {portfolio_alpha:.4f}")`
  Displays the computed portfolio alpha.
- `print(f"Systematic Variance: {systematic_var:.6f}")`
  Displays the portfolio's systematic variance.
- `print(f"Idiosyncratic Variance: {idiosyncratic_var:.6f}")`
  Displays the portfolio's idiosyncratic variance.

6.4 Example

The code estimates each stock's sensitivity to the market (e.g., TSLA may have a higher β_i due to volatility, MSFT lower). Using `tangency_weights`, it calculates the portfolio's overall market exposure and excess return not explained by the market. The variance decomposition separates risk into market-driven (systematic) and stock-specific (idiosyncratic) components, with stocks like TSLA contributing more to idiosyncratic risk. This output, generated by the code, clarifies how much of the portfolio's performance stems from market movements versus individual stock characteristics, embodying the CAPM framework.

7 Expected vs. Realized Performance

Expected performance (historical) is compared to realized performance (holdout period).

7.1 Mathematical Formulation

Expected:

$$\mu_p = \mathbf{w}^T \boldsymbol{\mu}, \quad \sigma_p = \sqrt{\mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w}} \tag{11}$$

Realized (test period, T days):

$$\mu_p^{\text{real}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}^T \mathbf{r}_t \tag{12}$$

$$\sigma_p^{\text{real}} = \sqrt{\frac{1}{T-1} \sum_{t=1}^{T} (\mathbf{w}^T \mathbf{r}_t - \mu_p^{\text{real}})^2} \tag{13}$$

μ_p assumes historical stability; μ_p^{real} reflects actual outcomes.

7.2 Python Implementation

```python
from datetime import date

# Split data using datetime.date
train_returns = returns.loc[:date(2023, 12, 31)]
test_returns = returns.loc[date(2024, 1, 1):]

# Expected performance
mean_returns_train = train_returns.mean() * 252
cov_matrix_train = train_returns.cov() * 252
exp_return = np.sum(mean_returns_train * tangency_weights)
exp_vol = portfolio_volatility(tangency_weights, cov_matrix_train)

# Realized performance
realized_returns = np.sum(test_returns * tangency_weights, axis=1)
realized_return = realized_returns.mean() * 252
realized_vol = realized_returns.std() * np.sqrt(252)

print(f"Expected Return: {exp_return:.4f}, Realized Return: {realized_return:.4f}")
print(f"Expected Volatility: {exp_vol:.4f}, Realized Volatility: {realized_vol:.4f}")
```

7.3 Code Explanation

- `train_returns` (2020–2023) and `test_returns` (2024) split the data for estimation and testing.
- `mean_returns_train * 252` and `cov_matrix_train * 252` compute annualized statistics from training data.
- `exp_return` uses `tangency_weights`; `exp_vol` uses `portfolio_volatility`.
- `realized_returns` computes daily portfolio returns in 2024, with `mean() * 252` and `std() * np.sqrt(252)` annualizing return and volatility.

7.4 Example

The code uses `tangency_weights` to estimate the portfolio's expected return and volatility based on historical data (2020–2023), assuming past patterns persist. For 2024, it calculates the actual return and volatility by applying the same weights to new returns, capturing real market conditions. Differences arise if stocks like TSLA perform unexpectedly (e.g., due to market shifts). The output compares these metrics, showing how the code tests the reliability of historical estimates against actual outcomes, illustrating the challenge of forecasting portfolio performance.