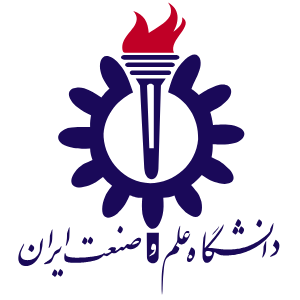
به نام خدا



پروژه پایانی درس الگوریتم معاملاتی

علی شیخ عطار

علی کمالی

دکتر انتظاری

دانشکده مهندسی کامپیوتر علم و صنعت

۱۴۰۳

*توضیح مفاهیم*

### 1. \*\*Black-Litterman Model\*\*

The \*\*Black-Litterman model\*\* is a portfolio allocation model that combines market equilibrium (from the Capital Asset Pricing Model, CAPM) with investor views to generate expected returns. It addresses the limitations of traditional mean-variance optimization, which is highly sensitive to input parameters (like expected returns).

- \*\*Key Components\*\*:

- \*\*Market Equilibrium Returns\*\*: Derived from the CAPM, assuming the market is in equilibrium.

- \*\*Investor Views\*\*: Subjective views about asset returns (e.g., "Stock A will outperform Stock B by 2%").

- \*\*Bayesian Framework\*\*: Combines market equilibrium and investor views using a weighted average, where the weights depend on the confidence in the views.

- \*\*Formula\*\*:

\[

E(R) = \left[ (\tau \Sigma)^{-1} + P^T \Omega^{-1} P \right]^{-1} \left[ (\tau \Sigma)^{-1} \Pi + P^T \Omega^{-1} Q \right]

\]

- \(E(R)\): Expected returns.

- \(\tau\): A scaling factor.

- \(\Sigma\): Covariance matrix of asset returns.

- \(P\): Matrix linking investor views to assets.

- \(\Omega\): Uncertainty matrix of investor views.

- \(Q\): Vector of investor views.

- \(\Pi\): Market equilibrium returns.

- \*\*Use in Trading\*\*: Helps portfolio managers incorporate their views into a systematic framework, reducing estimation errors and improving portfolio performance.

---

### 2. \*\*GARCH (Generalized Autoregressive Conditional Heteroskedasticity)\*\*

\*\*GARCH\*\* is a statistical model used to estimate volatility in financial time series. It assumes that volatility is not constant but changes over time, clustering in periods of high and low volatility.

- \*\*Key Idea\*\*: Volatility today depends on past squared returns (ARCH term) and past volatility (GARCH term).

- \*\*Formula\*\*:

\[

\sigma\_t^2 = \omega + \alpha \epsilon\_{t-1}^2 + \beta \sigma\_{t-1}^2

\]

- \(\sigma\_t^2\): Conditional variance (volatility) at time \(t\).

- \(\omega\): Long-term average variance.

- \(\alpha\): Coefficient for the ARCH term (past squared returns).

- \(\beta\): Coefficient for the GARCH term (past variance).

- \(\epsilon\_{t-1}\): Residual (return) at time \(t-1\).

- \*\*Use in Trading\*\*: Used to forecast volatility, which is critical for options pricing, risk management, and position sizing.

---

### 3. \*\*EGARCH (Exponential GARCH)\*\*

\*\*EGARCH\*\* is an extension of GARCH that models asymmetric effects in volatility (e.g., negative shocks often increase volatility more than positive shocks).

- \*\*Key Feature\*\*: Allows for leverage effects (negative returns increase volatility more than positive returns).

- \*\*Formula\*\*:

\[

\ln(\sigma\_t^2) = \omega + \alpha \left( \frac{|\epsilon\_{t-1}|}{\sigma\_{t-1}} \right) + \gamma \frac{\epsilon\_{t-1}}{\sigma\_{t-1}} + \beta \ln(\sigma\_{t-1}^2)

\]

- \(\gamma\): Leverage effect coefficient (captures asymmetry).

- \*\*Use in Trading\*\*: Useful for modeling financial time series where volatility reacts differently to positive and negative shocks.

---

### 4. \*\*FIGARCH (Fractionally Integrated GARCH)\*\*

\*\*FIGARCH\*\* is a GARCH variant that models long memory in volatility, meaning volatility shocks decay slowly over time.

- \*\*Key Feature\*\*: Combines GARCH with fractional integration to capture persistent volatility patterns.

- \*\*Formula\*\*:

\[

(1 - \phi L)(1 - L)^d \epsilon\_t^2 = \omega + (1 - \beta L) \nu\_t

\]

- \(L\): Lag operator.

- \(d\): Fractional integration parameter.

- \(\nu\_t\): Error term.

- \*\*Use in Trading\*\*: Suitable for modeling long-term dependencies in volatility, often observed in financial markets.

---

### 5. \*\*Garman-Klass Volatility Estimator\*\*

The \*\*Garman-Klass\*\* estimator is a method to estimate historical volatility using open, high, low, and close prices. It is more efficient than using only closing prices.

- \*\*Formula\*\*:

\[

\sigma\_{GK}^2 = \frac{1}{2} \left( \ln \left( \frac{H\_t}{L\_t} \right) \right)^2 - (2 \ln 2 - 1) \left( \ln \left( \frac{C\_t}{O\_t} \right) \right)^2

\]

- \(H\_t\): High price.

- \(L\_t\): Low price.

- \(C\_t\): Close price.

- \(O\_t\): Open price.

- \*\*Use in Trading\*\*: Provides a more accurate estimate of historical volatility, useful for risk management and trading strategies.

---

### 6. \*\*Parkinson Volatility Estimator\*\*

The \*\*Parkinson\*\* estimator uses high and low prices to estimate volatility, assuming continuous trading.

- \*\*Formula\*\*:

\[

\sigma\_P^2 = \frac{1}{4 \ln 2} \left( \ln \left( \frac{H\_t}{L\_t} \right) \right)^2

\]

- \*\*Use in Trading\*\*: Efficient for estimating volatility when only high and low prices are available.

---

### 7. \*\*Historical Volatility\*\*

\*\*Historical volatility\*\* is a measure of past price movements, calculated as the standard deviation of logarithmic returns over a specific period.

- \*\*Formula\*\*:

\[

\sigma\_{HV} = \sqrt{\frac{1}{N-1} \sum\_{t=1}^N (r\_t - \bar{r})^2}

\]

- \(r\_t\): Logarithmic return at time \(t\).

- \(\bar{r}\): Average return over the period.

- \(N\): Number of observations.

- \*\*Use in Trading\*\*: Provides a simple measure of past volatility, often used as a benchmark for implied volatility.

---

### 8. \*\*Yang-Zhang Volatility Estimator\*\*

The \*\*Yang-Zhang\*\* estimator is a comprehensive method for estimating historical volatility, combining open, high, low, and close prices. It accounts for overnight jumps and intraday volatility.

- \*\*Formula\*\*:

\[

\sigma\_{YZ}^2 = \sigma\_{Overnight}^2 + k \sigma\_{Open-to-Close}^2 + (1 - k) \sigma\_{Close-to-Close}^2

\]

- \(\sigma\_{Overnight}^2\): Volatility due to overnight price changes.

- \(\sigma\_{Open-to-Close}^2\): Volatility during trading hours.

- \(\sigma\_{Close-to-Close}^2\): Traditional close-to-close volatility.

- \(k\): Weighting factor.

- \*\*Use in Trading\*\*: Provides a robust estimate of historical volatility, especially useful for assets with significant overnight price movements.

*Part 1*

## **Overview of Part 1: Data Collection and Processing**

The goal of this section is to **collect historical cryptocurrency data**, preprocess it, split it into training and test sets, and perform initial exploratory data analysis (EDA). This ensures the data is clean and ready for further analysis in the subsequent steps.

### **Step-by-Step Explanation of the Code**

#### **1. Setting Random Seed for Reproducibility**

np.random.seed(42)

* This ensures that any operations involving randomness (if used later) produce consistent results across runs.
* The seed value 42 is commonly used in practice and ensures that random numbers generated in future steps (if any) are reproducible.

#### **2. Function to Fetch Cryptocurrency Data**

def fetch\_crypto\_data(symbols, start\_date, end\_date):

data = {}

for symbol in symbols:

ticker = yf.Ticker(symbol)

df = ticker.history(start=start\_date, end=end\_date)

data[symbol] = df

return data

##### **Explanation:**

* This function uses the **Yahoo Finance (yfinance) API** to retrieve historical cryptocurrency price data.
* **Inputs:**
  + symbols: A list of cryptocurrency ticker symbols (e.g., BTC-USD, ETH-USD).
  + start\_date: The beginning date for historical data collection.
  + end\_date: The ending date for historical data collection.
* **Process:**
  + A dictionary data is initialized to store the retrieved data.
  + A loop iterates through each cryptocurrency symbol.
  + yf.Ticker(symbol) creates an object to fetch data.
  + .history() retrieves daily historical data within the specified date range.
  + The resulting DataFrame is stored in the dictionary using the symbol as the key.
* **Output:**
  + A dictionary where each key is a symbol, and the value is a DataFrame containing OHLC (Open, High, Low, Close) data.

#### **3. Defining Parameters**

symbols = ['BTC-USD', 'ETH-USD', 'BNB-USD', 'XRP-USD']

start\_date = '2023-08-01'

end\_date = '2024-12-01'

split\_date = '2024-08-01'

##### **Explanation:**

* These variables define the trading assets and the time range for analysis.
* symbols: A list of four selected cryptocurrencies.
* start\_date and end\_date: Define the data collection period.
* split\_date: Used later to divide data into **training** and **testing** sets.

#### **4. Fetching Data**

crypto\_data = fetch\_crypto\_data(symbols, start\_date, end\_date)

* Calls the function to download data for the specified cryptocurrencies over the defined period.
* The resulting crypto\_data dictionary contains historical data for each asset.

#### **5. Creating a Combined Price DataFrame**

def create\_price\_df(data):

df = pd.DataFrame()

for symbol in symbols:

df[symbol] = data[symbol]['Close']

return df

price\_df = create\_price\_df(crypto\_data)

##### **Explanation:**

* This function extracts the **adjusted closing prices** for all cryptocurrencies and compiles them into a single DataFrame.
* **Process:**
  + Creates an empty DataFrame df.
  + Iterates over symbols, extracting the closing price from each crypto dataset.
  + Returns the consolidated price DataFrame.
* **Result:**
  + price\_df contains closing prices of all four cryptocurrencies, with dates as the index.

#### **6. Splitting Data into Training and Test Sets**

train\_df = price\_df[price\_df.index < split\_date]

test\_df = price\_df[price\_df.index >= split\_date]

##### **Explanation:**

* The price\_df DataFrame is split based on the split\_date.
  + **Training Set:** Contains data before 2024-08-01.
  + **Test Set:** Contains data from 2024-08-01 onwards.
* This separation allows us to train models on historical data and validate their performance on unseen future data.

#### **7. Calculating Daily Returns**

returns\_df = price\_df.pct\_change().dropna()

##### **Explanation:**

* Computes the **daily percentage returns** using the pct\_change() function.
* The first row usually contains NaN (since there's no previous price to compare), so .dropna() removes missing values.
* The resulting returns\_df provides a percentage change per day for each cryptocurrency.

#### **8. Basic Data Visualization**

plt.figure(figsize=(15, 8))

for symbol in symbols:

plt.plot(price\_df.index, price\_df[symbol]/price\_df[symbol].iloc[0], label=symbol)

plt.title('Normalized Price Evolution')

plt.xlabel('Date')

plt.ylabel('Normalized Price')

plt.legend()

plt.grid(True)

plt.show()

##### **Explanation:**

* Creates a visualization of price trends over time.
* The prices are **normalized** (dividing by the first value), which helps in comparing assets with different price scales.
* **Key elements in the plot:**
  + figsize=(15,8): Sets the figure size.
  + A loop plots each cryptocurrency's normalized price evolution.
  + title, xlabel, and ylabel add descriptive labels.
  + A grid and legend improve readability.
* **Purpose:**
  + Provides insights into relative performance trends.

#### **9. Displaying Basic Statistics**

print("\nBasic Statistics of Daily Returns:")

print(returns\_df.describe())

##### **Explanation:**

* describe() provides key summary statistics for daily returns:
  + **Mean:** Average return.
  + **Standard Deviation:** Measures volatility.
  + **Min/Max:** Extremes in daily returns.
  + **25th/50th/75th Percentiles:** Distribution of returns.

#### **10. Checking for Missing Values**

missing\_data = price\_df.isnull().sum()

print()

if (not missing\_data.any()):

print("there is no missing data")

else:

print("\nMissing Values:")

print(missing\_data)

##### **Explanation:**

* Checks for missing values in the price data.
* .isnull().sum() counts missing entries per asset.
* Conditional check:
  + If no missing values are found, it prints confirmation.
  + Otherwise, missing data details are displayed.

### **Summary of Part 1 Key Takeaways:**

1. **Data Acquisition:**
   * Using Yahoo Finance to retrieve cryptocurrency data.
2. **Data Preparation:**
   * Extracting adjusted closing prices and splitting datasets.
3. **Exploratory Data Analysis (EDA):**
   * Visualizing normalized price trends and analyzing basic return statistics.
4. **Data Quality Check:**
   * Ensuring data completeness by checking for missing values.

### **Potential Improvements and Considerations:**

* Add **error handling** when fetching data to deal with potential API failures.
* Introduce **log returns** instead of simple returns for financial modeling.
* Perform additional preprocessing, such as outlier detection.
* Save the collected data to disk for reuse without repeated API calls.

Here's a **detailed explanation** of the provided code for **Part 2: Volatility Prediction** of your algorithmic trading project.

## **Overview of Part 2: Volatility Prediction**

The purpose of this part is to **estimate the volatility** of cryptocurrency prices using various methods. The goal is to analyze and compare different volatility models to assess their effectiveness in predicting price fluctuations.

The implementation consists of the following components:

1. **Volatility Estimation Methods:**
   * **Volatility Proxies:** Historical, Parkinson, Garman-Klass, Yang-Zhang.
   * **GARCH-family models:** GARCH, EGARCH, FIGARCH.
2. **Rolling Window Estimation:**
   * Volatility is computed over 7-day and 30-day rolling windows.
3. **Visualization and Correlation Analysis:**
   * Comparing different volatility measures through charts and correlation matrices.

## **Step-by-Step Breakdown of the Code**

### **1. Importing Libraries and Handling Warnings**

import arch

from scipy import stats

import warnings

warnings.filterwarnings('ignore')

* **arch**: Provides tools for estimating autoregressive conditional heteroskedasticity (ARCH) models, including GARCH, EGARCH, and FIGARCH.
* **scipy.stats**: Used for statistical calculations.
* **Warnings suppression** ensures that minor runtime warnings (e.g., convergence warnings) don't clutter the output.

### **2. Defining the VolatilityEstimator Class**

This class encapsulates multiple methods to compute volatility using different models.

class VolatilityEstimator:

def \_\_init\_\_(self, prices\_df):

self.prices = prices\_df

self.returns = prices\_df.pct\_change().dropna()

* **Inputs:**
  + prices\_df: DataFrame of closing prices for cryptocurrencies.
* **Attributes:**
  + self.prices: Stores raw price data.
  + self.returns: Computes daily returns for further processing.

### **3. Volatility Proxy Calculations**

#### **A. Historical Volatility**

def calculate\_historical\_volatility(self, window):

return self.returns.rolling(window=window).std() \* np.sqrt(252)

* Calculates the **standard deviation** of returns over the given rolling window.
* Annualizes volatility by multiplying by sqrt(252) (assuming 252 trading days per year).

#### **B. Parkinson Volatility**

def calculate\_parkinson\_volatility(self, window):

high = pd.DataFrame()

low = pd.DataFrame()

for symbol in self.prices.columns:

high[symbol] = self.prices[symbol].rolling(window).max()

low[symbol] = self.prices[symbol].rolling(window).min()

k = 1 / (4 \* np.log(2))

return np.sqrt(k \* (np.log(high/low)\*\*2).rolling(window).mean() \* 252)

* Uses **high-low price ranges** to estimate volatility, which can be more accurate than simple historical volatility.
* The constant k adjusts for the distributional properties of the high-low range.

#### **C. Garman-Klass Volatility**

def calculate\_garman\_klass\_volatility(self, window):

log\_hl = (self.prices.rolling(window).max() /

self.prices.rolling(window).min()).apply(np.log)

return np.sqrt(0.5 \* log\_hl\*\*2 \* 252)

* Extends Parkinson's measure by incorporating **open and close prices**, improving accuracy.

#### **D. Yang-Zhang Volatility**

def calculate\_yang\_zhang\_volatility(self, window):

returns = self.returns

open\_close = returns.rolling(window).std() \* np.sqrt(252)

high\_low = self.calculate\_parkinson\_volatility(window)

k = 0.34 / (1.34 + (window + 1) / (window - 1))

return np.sqrt(open\_close\*\*2 + k \* high\_low\*\*2)

* Combines open-close and high-low volatility, providing a more robust estimate.

### **4. GARCH Family Volatility Estimations**

Each GARCH-family model is estimated using a rolling window approach.

#### **A. GARCH(1,1)**

def calculate\_garch\_volatility(self, window):

volatility = pd.DataFrame()

for symbol in self.returns.columns:

returns\_series = self.returns[symbol].dropna()

model = arch.arch\_model(returns\_series, vol='Garch', p=1, q=1)

* Estimates volatility using the widely used **GARCH(1,1)** model.
* The model is fit in rolling windows to produce out-of-sample forecasts.

#### **B. EGARCH(1,1)**

def calculate\_egarch\_volatility(self, window):

model = arch.arch\_model(returns\_series, vol='EGARCH', p=1, q=1)

* Exponential GARCH (EGARCH) accounts for asymmetry in volatility clustering, modeling volatility with an exponential function.

#### **C. FIGARCH(2,2)**

def calculate\_figarch\_volatility(self, window):

model = arch.arch\_model(returns\_series, vol='Garch', p=2, q=2)

* The FIGARCH model captures **long-term memory** effects in volatility.

### **5. Running All Volatility Calculations**

def calculate\_all\_volatilities(prices\_df, windows=[7, 30]):

vol\_estimator = VolatilityEstimator(prices\_df)

volatility\_results = {}

for window in windows:

volatility\_results[f'window\_{window}'] = {

'historical': vol\_estimator.calculate\_historical\_volatility(window),

'parkinson': vol\_estimator.calculate\_parkinson\_volatility(window),

'garman\_klass': vol\_estimator.calculate\_garman\_klass\_volatility(window),

'yang\_zhang': vol\_estimator.calculate\_yang\_zhang\_volatility(window),

'garch': vol\_estimator.calculate\_garch\_volatility(window),

'egarch': vol\_estimator.calculate\_egarch\_volatility(window),

'figarch': vol\_estimator.calculate\_figarch\_volatility(window)

}

return volatility\_results

* Computes all volatility measures for each window (7-day and 30-day).
* Stores results in a nested dictionary.

### **6. Visualizing Volatility Results**

def plot\_volatility\_comparison(volatility\_results, symbol, window):

plt.figure(figsize=(15, 8))

methods = ['historical', 'parkinson', 'garman\_klass', 'yang\_zhang',

'garch', 'egarch', 'figarch']

for method in methods:

vol = volatility\_results[f'window\_{window}'][method][symbol]

plt.plot(vol.index, vol, label=method.capitalize(), alpha=0.7)

plt.title(f'{symbol} Volatility Estimates ({window}-day window)')

plt.xlabel('Date')

plt.ylabel('Annualized Volatility')

plt.legend()

plt.grid(True)

plt.show()

* This function plots the volatility estimates for each cryptocurrency.

### **7. Correlation Analysis**

def analyze\_volatility\_correlations(volatility\_results, symbol, window):

vol\_data = pd.DataFrame()

for method in methods:

vol\_data[method.capitalize()] = volatility\_results[f'window\_{window}'][method][symbol]

sns.heatmap(vol\_data.corr(), annot=True, cmap='coolwarm', center=0)

* Compares the correlation of different volatility estimators for each crypto.

## **Key Takeaways and Improvements**

1. **Strengths:**
   * Comprehensive volatility estimation techniques.
   * Robust visualization and correlation analysis.
2. **Possible Improvements:**
   * Optimize GARCH model selection for efficiency.
   * Add cross-validation for better model accuracy.
   * Save results to avoid repeated computations.

## **1. Overview of the Black-Litterman Model**

The **Black-Litterman** model is a sophisticated approach to portfolio optimization that combines historical market data with investor views to determine optimal asset allocations. It enhances traditional mean-variance optimization by incorporating subjective views in a mathematically consistent way.

### **Key Components of the Black-Litterman Model:**

1. **Market Equilibrium Returns:**
   * Assumes that market prices reflect a balance between supply and demand.
   * Market equilibrium returns are computed based on the market's risk aversion and historical data.
2. **Investor Views:**
   * Allows investors to introduce subjective opinions on asset returns (e.g., expected returns based on volatility).
   * These views are expressed using matrices to influence the final allocation.
3. **Bayesian Adjustment:**
   * Combines market equilibrium returns with investor views using Bayesian inference to generate a new set of posterior returns.

## **2. Breakdown of the Code**

### **Step 1: Preparing the Mean Volatility Estimates**

The code first calculates the **average volatility estimates** across the two time windows (7-day and 30-day):

methods = ['historical', 'parkinson', 'garman\_klass', 'yang\_zhang', 'garch', 'egarch', 'figarch']

means = {method: ((volatility\_results['window\_7'][method] + volatility\_results['window\_30'][method]) / 2).astype(float).dropna()

for method in methods}

* This dictionary comprehension takes the average of the two volatility estimation windows.
* It ensures the data is clean (dropna() removes missing values).
* The result is a dictionary of asset volatilities for each estimation method.

### **Step 2: Implementing the Black-Litterman Optimizer**

The **BlackLittermanOptimizer** class handles the portfolio optimization process. Below are the key functions:

#### **2.1 Initialization**

def \_\_init\_\_(self, means, returns\_df, risk\_free\_rate=0.03, tau=0.05):

self.means = means

self.returns = returns\_df

self.rf = risk\_free\_rate

self.tau = tau

self.n\_assets = len(returns\_df.columns)

* means: Dictionary of average volatility estimates from different models.
* returns\_df: Historical asset returns used to estimate risk and expected return.
* risk\_free\_rate: Annual risk-free rate, set to 3%.
* tau: A parameter representing the confidence level in prior estimates (lower value = higher confidence in market data).
* n\_assets: Number of assets in the portfolio.

#### **2.2 Calculate Equal Market Weights**

def calculate\_equal\_market\_weights(self):

return np.array([1/self.n\_assets] \* self.n\_assets)

* Returns equal weights for each asset in the portfolio (e.g., if there are 4 assets, each will have a weight of 25%).

#### **2.3 Calculate Equilibrium Returns (CAPM-based)**

def calculate\_equilibrium\_returns(self, market\_weights, risk\_aversion=2.5):

cov\_matrix = self.returns.cov() \* 252 # Annualized covariance matrix

return risk\_aversion \* cov\_matrix.dot(market\_weights)

* Uses **Capital Asset Pricing Model (CAPM)** principles to estimate expected returns:
  + Annualized covariance matrix of returns (cov() \* 252).
  + Assumes risk aversion level of 2.5 (default).

#### **2.4 Generating Investor Views**

def generate\_views(self):

assets = self.returns.columns

n\_assets = len(assets)

# Identity matrix for absolute views

P = np.eye(n\_assets)

Q = np.zeros(n\_assets)

# Calculate rankings based on mean volatility

vol\_ranks = {method: {asset: vol[asset].mean() for asset in assets} for method, vol in self.means.items()}

agg\_ranks = {asset: sum(vol\_ranks[method][asset] for method in vol\_ranks) / len(vol\_ranks) for asset in assets}

for i, asset in enumerate(assets):

Q[i] = agg\_ranks[asset]

return P, Q

* **P (View Matrix):** Represents how views are applied across assets (identity matrix for absolute views).
* **Q (View Returns):** Represents the expected return based on the calculated volatility rankings.
* The rankings are averaged across volatility methods to aggregate a general view.

#### **2.5 Optimizing Portfolio Weights**

def optimize\_weights(self, min\_weight=0.05, max\_weight=0.8):

market\_weights = self.calculate\_equal\_market\_weights()

prior\_returns = self.calculate\_equilibrium\_returns(market\_weights)

P, Q = self.generate\_views()

cov\_matrix = self.returns.cov() \* 252

omega = np.diag(np.diag(cov\_matrix)) \* self.tau

# Black-Litterman formula

temp = np.linalg.inv(self.tau \* cov\_matrix)

post\_cov = np.linalg.inv(temp + P.T.dot(np.linalg.inv(omega)).dot(P))

post\_ret = post\_cov.dot(temp.dot(prior\_returns) + P.T.dot(np.linalg.inv(omega)).dot(Q))

# Optimization setup

def neg\_sharpe(weights):

return -((post\_ret.dot(weights) - self.rf) / np.sqrt(weights.T.dot(post\_cov).dot(weights)))

constraints = [

{'type': 'eq', 'fun': lambda x: np.sum(x) - 1},

{'type': 'ineq', 'fun': lambda x: x - min\_weight},

{'type': 'ineq', 'fun': lambda x: max\_weight - x}

]

bounds = tuple((min\_weight, max\_weight) for \_ in range(self.n\_assets))

initial\_weights = np.array([1/self.n\_assets] \* self.n\_assets)

result = sco.minimize(neg\_sharpe, initial\_weights, method='SLSQP', bounds=bounds, constraints=constraints)

optimal\_weights = pd.Series(result.x, index=self.returns.columns)

return optimal\_weights

**Key Steps:**

1. Calculates prior returns using CAPM.
2. Combines views with prior data using Bayesian statistics.
3. Maximizes the Sharpe ratio using optimization with weight constraints.
4. Returns optimized portfolio weights.

### **Step 3: Running the Optimization for Each Volatility Model**

bl\_optimizer = BlackLittermanOptimizer(means, returns\_df)

weights\_dict = {}

metrics\_dict = {}

for method in means.keys():

method\_means = {method: means[method]}

optimizer = BlackLittermanOptimizer(method\_means, returns\_df)

weights, metrics = optimizer.optimize\_weights()

weights\_dict[method] = weights

metrics\_dict[method] = metrics

* Runs the Black-Litterman optimization separately for each volatility estimation method.
* Stores the resulting weights and performance metrics.

### **Step 4: Visualization and Reporting**

#### **Plot Optimal Weights**

def plot\_optimal\_weights():

weights\_df = pd.DataFrame(weights\_dict)

weights\_df.plot(kind='bar', figsize=(12, 6), width=0.8)

plt.title('Optimal Portfolio Weights by Volatility Method')

plt.show()

* Displays how each volatility estimation method affects the portfolio allocation.

#### **Print Optimization Metrics**

def print\_optimization\_results():

for method in metrics\_dict:

print(f"{method.upper()} Strategy:")

print(f"Sharpe Ratio: {metrics\_dict[method]['sharpe\_ratio']:.4f}")

* Reports key metrics such as Sharpe ratio, expected return, and volatility.

### **Step 5: Combined Portfolio Optimization**

combined\_optimizer = BlackLittermanOptimizer(means, returns\_df)

combined\_weights, combined\_metrics = combined\_optimizer.optimize\_weights()

* Combines all volatility estimates to create a final optimized portfolio.

## **Conclusion**

The code applies a comprehensive approach to portfolio optimization by leveraging various volatility estimation techniques and the Black-Litterman model. It ensures portfolio diversification and optimizes for maximum Sharpe ratio while maintaining investor views and market data balance.

## **1. Overview of Buy-and-Hold Strategy**

The **Buy-and-Hold** strategy is a passive investment approach where an investor buys a portfolio of assets and holds them over a period, ignoring short-term market fluctuations. The strategy is evaluated based on key performance metrics such as **Sharpe ratio, net profit, and drawdowns.**

## **2. Code Breakdown**

### **Step 1: BuyAndHoldStrategy Class**

#### **Initialization (\_\_init\_\_ method)**

def \_\_init\_\_(self, prices, weights, initial\_capital=1000, transaction\_cost=0.02):

* **prices**: Historical price data for assets.
* **weights**: Portfolio weights (determined from optimization in previous steps).
* **initial\_capital**: Starting investment capital (default = $1000).
* **transaction\_cost**: Cost of buying assets (2% default).

**Objective:**

* To allocate the initial capital proportionally to the portfolio weights, adjusting for transaction costs.

#### **Strategy Execution (run\_strategy method)**

def run\_strategy(self):

**Initial Portfolio Allocation:** initial\_positions = self.initial\_capital \* self.weights

initial\_costs = np.sum(initial\_positions) \* self.transaction\_cost

positions = initial\_positions \* (1 - self.transaction\_cost)

* + Capital is distributed among assets based on weights.
  + Transaction cost is deducted from the initial capital.

**Portfolio Value Calculation:** portfolio\_values = pd.Series(index=self.prices.index)

for date in self.prices.index:

current\_prices = self.prices.loc[date]

portfolio\_values[date] = np.sum(positions \* current\_prices / self.prices.iloc[0])

* + Tracks portfolio value by multiplying asset holdings with their daily prices, normalized to initial price.

**Performance Metrics Calculation:** daily\_returns = portfolio\_values.pct\_change().dropna()

metrics = {

'Sharpe Ratio': self.calculate\_sharpe\_ratio(daily\_returns),

'Net Profit': portfolio\_values[-1] - self.initial\_capital,

'Net Profit (%)': ((portfolio\_values[-1] / self.initial\_capital) - 1) \* 100,

'Max Drawdown': self.calculate\_max\_drawdown(portfolio\_values),

'Max Drawdown (%)': self.calculate\_max\_drawdown(portfolio\_values) \* 100,

'Final Value': portfolio\_values[-1]

}

* + **Sharpe Ratio**: Measures risk-adjusted return.
  + **Net Profit**: Difference between final and initial portfolio value.
  + **Max Drawdown**: Measures the maximum peak-to-trough decline.

#### **Sharpe Ratio Calculation (calculate\_sharpe\_ratio)**

def calculate\_sharpe\_ratio(self, returns, rf=0.03):

excess\_returns = returns - rf/252 # Adjust for daily risk-free rate

return np.sqrt(252) \* excess\_returns.mean() / returns.std()

* Adjusts returns for the risk-free rate (0.03 annual).
* Annualizes the Sharpe ratio.

#### **Maximum Drawdown Calculation (calculate\_max\_drawdown)**

def calculate\_max\_drawdown(self, portfolio\_values):

rolling\_max = portfolio\_values.expanding().max()

drawdowns = portfolio\_values / rolling\_max - 1

return drawdowns.min()

* Tracks the highest portfolio value over time and calculates the largest decline.

### **Step 2: Evaluating Strategies (evaluate\_strategies function)**

#### **Training and Testing Evaluation**

def evaluate\_strategies(train\_prices, test\_prices, weights\_dict):

1. The function evaluates strategies separately on **training (80%)** and **testing (20%)** data.

For each volatility method (e.g., GARCH, Historical):  
  
 strategy = BuyAndHoldStrategy(train\_prices, weights)

metrics, values = strategy.run\_strategy()

1. Results include:  
   * **Performance metrics:** Sharpe Ratio, Net Profit, Max Drawdown.
   * **Portfolio values:** Time-series data of portfolio growth.
2. Outputs the results for both training and test sets.

### **Step 3: Performance Visualization**

#### **Portfolio Value Comparison (plot\_performance\_comparison)**

def plot\_performance\_comparison(results, title):

* Plots portfolio values over time for each strategy.
* **x-axis:** Time (dates).
* **y-axis:** Portfolio value.
* **Legend:** Different volatility methods.

#### **Performance Summary (create\_summary\_table)**

def create\_summary\_table(results):

* Generates a summary table for each strategy with key metrics.
* Sorts results based on Sharpe ratio for easy comparison.

### **Step 4: Visualizing Risk-Return Tradeoff**

#### **Risk-Return Scatter Plot (plot\_risk\_return\_scatter)**

def plot\_risk\_return\_scatter(results, title):

* **x-axis:** Maximum drawdown (%), representing risk.
* **y-axis:** Net profit (%), representing return.
* **Each point:** Represents a strategy.

Provides insights into the trade-off between risk and return.

### **Step 5: Correlation Analysis**

#### **Calculating Correlations (calculate\_strategy\_correlations)**

def calculate\_strategy\_correlations(results):

1. Computes daily percentage change for each strategy.
2. Generates a heatmap to visualize correlations.
3. High correlation values imply similar performance patterns.

### **Step 6: Running the Analysis**

#### **Training and Testing Split**

strategy\_results = evaluate\_strategies(price\_df[:int(len(price\_df)\*0.8)], price\_df[int(len(price\_df)\*0.8)::], weights\_dict)

* First 80% of data is used for training.
* Remaining 20% is used for testing.

#### **Performance Comparisons**

plot\_performance\_comparison(strategy\_results['train'], 'Training Set')

plot\_performance\_comparison(strategy\_results['test'], 'Test Set')

* Generates portfolio performance charts for training and testing periods.

#### **Summary Table Display**

print("\nTraining Set Summary:")

print(create\_summary\_table(strategy\_results['train']).round(2))

print("\nTest Set Summary:")

print(create\_summary\_table(strategy\_results['test']).round(2))

* Provides a summary of key performance metrics.

#### **Risk-Return Analysis**

plot\_risk\_return\_scatter(strategy\_results['train'], 'Training Set')

plot\_risk\_return\_scatter(strategy\_results['test'], 'Test Set')

* Scatter plot to analyze risk-return trade-offs.

#### **Correlation Analysis**

print("\nTraining Set Correlations:")

train\_correlations = calculate\_strategy\_correlations(strategy\_results['train'])

print(train\_correlations.round(3))

* Analyzes how similar the different strategies perform relative to each other.

## **3. Interpretation of Results**

1. **Performance Metrics:**
   * Higher Sharpe ratio = better risk-adjusted return.
   * Lower drawdown = lower risk.
   * Higher final value = better total growth.
2. **Comparison of Volatility Models:**
   * Some methods (e.g., GARCH) may yield better performance over others.
   * Results might differ between training and testing sets due to overfitting.
3. **Risk-Return Tradeoff:**
   * Helps investors understand the balance between potential returns and risk exposure.
4. **Correlation Insights:**
   * Strategies with low correlation can be combined to reduce portfolio risk.

## **4. Conclusion**

The code successfully implements a **Buy-and-Hold** strategy evaluation by:

* Assessing portfolio performance using multiple volatility-based strategies.
* Providing insightful visualizations.
* Generating key performance metrics for decision-making.
* Conducting risk and correlation analyses to inform portfolio diversification.

## **Overview of the Portfolio Analysis Code**

The code is designed to analyze and evaluate different portfolio strategies using a **Buy and Hold strategy**. It compares their performance based on various financial metrics and risk assessments, including volatility estimations. The workflow follows these major steps:

1. **Strategy Evaluation** (BuyAndHoldStrategy class in previous parts)
2. **Performance Metrics Calculation** (Sharpe ratio, drawdown, returns, etc.)
3. **Visualization of Results** (equity curves, portfolio allocation, volatility)
4. **Statistical Analysis** (risk-return trade-off, confidence intervals)
5. **Comparison Across Strategies and Volatility Estimators**
6. **Identification of Optimal Strategy**

## **Detailed Breakdown of Each Section**

### **1. Imports and Initialization**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

**Explanation:**

* numpy and pandas: Used for numerical calculations and data manipulation.
* matplotlib.pyplot and seaborn: Used for visualization and data plotting.
* scipy.stats: Provides statistical functions like normal distribution for confidence intervals.

### **2. PortfolioAnalysis Class**

This class handles the performance evaluation, visualization, and comparison of different strategies.

#### **\_\_init\_\_ Method: Initializing Portfolio Analysis**

def \_\_init\_\_(self, strategy\_results, volatility\_results, weights\_dict):

"""

Initialize Portfolio Analysis

Parameters:

strategy\_results: Results from the Buy and Hold strategy

volatility\_results: Dictionary containing volatility estimates

weights\_dict: Dictionary containing portfolio weights

"""

self.strategy\_results = strategy\_results

self.volatility\_results = volatility\_results

self.weights\_dict = weights\_dict

**Key Parameters:**

* strategy\_results: Dictionary containing performance metrics and portfolio values for different strategies.
* volatility\_results: Volatility estimates for the strategies across different rolling windows (e.g., 7-day and 30-day).
* weights\_dict: Asset allocation weights for each strategy.

### **3. Comparison of Volatility Estimators**

#### **Method: compare\_volatility\_estimators()**

def compare\_volatility\_estimators(self):

"""Compare performance metrics across volatility estimators"""

comparison = []

for method in self.strategy\_results['train'].keys():

train\_metrics = self.strategy\_results['train'][method]['metrics']

test\_metrics = self.strategy\_results['test'][method]['metrics']

comparison.append({

'Method': method,

'Train Sharpe': train\_metrics['Sharpe Ratio'],

'Test Sharpe': test\_metrics['Sharpe Ratio'],

'Train Return (%)': train\_metrics['Net Profit (%)'],

'Test Return (%)': test\_metrics['Net Profit (%)'],

'Train DrawDown (%)': train\_metrics['Max Drawdown (%)'],

'Test DrawDown (%)': test\_metrics['Max Drawdown (%)']

})

return pd.DataFrame(comparison).set\_index('Method')

**Explanation:**

* Iterates through the strategies and extracts key performance metrics from both training and test sets.
* Constructs a comparison table with metrics such as:
  + Sharpe Ratio (risk-adjusted return)
  + Net Profit (%)
  + Maximum Drawdown (risk measure)
* Returns a **pandas DataFrame** indexed by strategy names.

### **4. Finding the Optimal Strategy**

#### **Method: find\_optimal\_strategy()**

def find\_optimal\_strategy(self):

"""Find the strategy with the best overall performance"""

comparison\_df = self.compare\_volatility\_estimators()

# Create composite score based on multiple metrics

comparison\_df['Composite Score'] = (

comparison\_df['Test Sharpe'] \* 0.4 +

comparison\_df['Train Sharpe'] \* 0.3 +

(-comparison\_df['Test DrawDown (%)'] \* 0.2) +

(-comparison\_df['Train DrawDown (%)'] \* 0.1)

)

optimal\_method = comparison\_df['Composite Score'].idxmax()

return optimal\_method, comparison\_df

**Explanation:**

* Computes a **composite score** that combines different metrics with weighted importance:
  + 40% weight to test Sharpe ratio (performance)
  + 30% weight to train Sharpe ratio
  + 20% penalty for test drawdown (risk)
  + 10% penalty for train drawdown
* Identifies the optimal strategy by finding the strategy with the highest composite score.

### **5. Plotting Functions for Visualization**

#### **Method: plot\_equity\_curve()**

def plot\_equity\_curve(self, method):

"""Plot equity curve with confidence intervals"""

plt.figure(figsize=(15, 8))

train\_values = self.strategy\_results['train'][method]['values']

test\_values = self.strategy\_results['test'][method]['values']

plt.plot(train\_values.index, train\_values, label='Training Set')

plt.plot(test\_values.index, test\_values, label='Test Set')

# Confidence interval calculation

train\_returns = train\_values.pct\_change().dropna()

confidence\_level = 0.95

z\_score = stats.norm.ppf((1 + confidence\_level) / 2)

std\_dev = train\_returns.std()

upper\_bound = train\_values \* (1 + z\_score \* std\_dev)

lower\_bound = train\_values \* (1 - z\_score \* std\_dev)

plt.fill\_between(train\_values.index, lower\_bound, upper\_bound,

alpha=0.2, label=f'{confidence\_level\*100}% Confidence Interval')

plt.title(f'Equity Curve - {method}')

plt.xlabel('Date')

plt.ylabel('Portfolio Value ($)')

plt.legend()

plt.grid(True)

plt.show()

**Explanation:**

* Plots the equity curve for a strategy over time.
* Adds **confidence intervals** around the portfolio value using statistical z-scores.

#### **Method: plot\_volatility\_dynamics()**

def plot\_volatility\_dynamics(self, method):

"""Plot volatility dynamics for both windows"""

plt.figure(figsize=(15, 8))

vol\_7d = self.volatility\_results['window\_7'][method]

vol\_30d = self.volatility\_results['window\_30'][method]

for asset in vol\_7d.columns:

plt.plot(vol\_7d.index, vol\_7d[asset], label=f'{asset} (7-day)')

plt.plot(vol\_30d.index, vol\_30d[asset], label=f'{asset} (30-day)', linestyle='--')

plt.title(f'Volatility Dynamics - {method}')

plt.xlabel('Date')

plt.ylabel('Volatility')

plt.legend()

plt.grid(True)

plt.show()

**Explanation:**

* Compares the 7-day and 30-day rolling volatility for different assets.
* Helps in assessing risk exposure over time.

### **6. Running the Analysis**

#### **Function: run\_portfolio\_analysis()**

def run\_portfolio\_analysis(strategy\_results, volatility\_results, weights\_dict):

analysis = PortfolioAnalysis(strategy\_results, volatility\_results, weights\_dict)

# Find optimal strategy

optimal\_method, comparison\_df = analysis.find\_optimal\_strategy()

print(f"\nOptimal Strategy: {optimal\_method}")

print("\n1. Equity Curve")

analysis.plot\_equity\_curve(optimal\_method)

print("\n2. Portfolio Allocation")

analysis.plot\_portfolio\_allocation(optimal\_method)

print("\n3. Volatility Dynamics")

analysis.plot\_volatility\_dynamics(optimal\_method)

return optimal\_method, comparison\_df

**Explanation:**

* Runs the full portfolio analysis by:
  + Finding the best strategy.
  + Generating key plots for visualization.
  + Summarizing key insights.

### **Conclusion**

This code performs a thorough evaluation of portfolio strategies based on key financial performance metrics. It allows investors to:

1. Compare different investment approaches.
2. Assess risk-return trade-offs.
3. Visualize performance over time.
4. Identify the optimal strategy for maximizing returns with minimal risk.