Portfolio Risk and Optimization Engine

A Python-Based Tool for Portfolio Optimization

Dhruv Talati

Submitted as part of a project to develop a portfolio optimization tool using Python, with a focus on risk metrics and balanced allocation.

Contents

1 Introduction

The Portfolio Risk and Optimization Engine is a Python-based tool designed to assist investors in constructing an optimized portfolio while analyzing key risk metrics. Developed using libraries such as yfinance, pandas, numpy, scipy, matplotlib, and seaborn, this project fetches historical financial data, optimizes portfolio weights to maximize the Sharpe Ratio, computes risk metrics like Value at Risk (VaR) and Conditional Value at Risk (CVaR), and visualizes the results. The tool allows users to input their preferred financial instruments and investment amount, making it adaptable to individual preferences.

2 Goals and Objectives

The primary goals of this project are:

- To fetch historical financial data over a 5-year period using yfinance.
- To calculate returns, volatility, and correlations using pandas and numpy.
- To optimize portfolio weights to maximize the Sharpe Ratio, ensuring balanced allocation across all selected instruments, using scipy.optimize.
- To compute risk metrics, including Parametric VaR, Historical VaR, CVaR, Marginal VaR, and Incremental VaR.
- To visualize portfolio growth, optimal weights, and correlation heatmaps using matplotlib and seaborn.

3 Assumptions

The project relies on the following assumptions:

- Historical data over 5 years is a reliable predictor of future returns, volatility, and correlations.
- Returns are assumed to be normally distributed for Parametric VaR calculations, which estimates the maximum potential loss at a given confidence level based on statistical assumptions.
- The risk-free rate is constant at 2% annually.
- The covariance matrix accurately captures the relationships between financial instruments.
- Maximizing the Sharpe Ratio is the optimal objective for portfolio allocation.
- All capital is fully invested (weights sum to 1), with no cash holdings.
- Transaction costs and taxes are ignored for simplicity.
- Linear constraints in the optimization (weights between 0.01 and 1, summing to 1) lead to a global optimum.
- Historical VaR measures the potential loss at a given confidence level based on past data percentiles.
- CVaR, or Expected Shortfall, estimates the average loss in the worst-case scenarios beyond the VaR threshold.

- Marginal VaR indicates the change in portfolio VaR with a small increase in an instrument's weight, assuming other weights remain fixed.
- Incremental VaR measures the change in Historical VaR when an instrument's weight is increased by a small amount, with rebalancing of other weights.

4 Code Explanation

The following subsections break down the Python code in FinalProJ.py block by block, explaining the purpose and functionality of each section.

4.1 Importing Libraries

```
import yfinance as yf
import pandas as pd
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
```

This block imports the necessary Python libraries: yfinance for fetching financial data, pandas and numpy for data manipulation, scipy.optimize for optimization, matplotlib and seaborn for visualization, and datetime for date handling.

4.2 Setting Random Seed

```
np.random.seed(42)
```

This sets the random seed for NumPy to 42, ensuring reproducibility of results. The optimization process involves randomness (e.g., in initial guesses), and fixing the seed ensures consistent outputs across runs.

4.3 Input Principal and Tickers

```
principal = float(input("Enter your principal amount (e.g., 100000): "))
tickers = input("Enter ETF tickers separated by space (e.g., SPY NVDA AAPL
    XLF XLE XLV XLY XLC XLP XLK): ").split()
```

This block prompts the user to input their principal amount and a space-separated list of financial instrument tickers. The inputs are converted to a float and a list, respectively, allowing customization.

4.4 Fetching Historical Data

```
end_date = datetime(2025, 5, 18)
start_date = end_date - timedelta(days=5*365)
data = yf.download(tickers, start=start_date, end=end_date)['Adj Close']
```

This block defines a 5-year historical period and uses yfinance to download the adjusted closing prices for the specified instruments, storing them in a pandas DataFrame.

4.5 Calculating Returns and Covariance

```
returns = data.pct_change().dropna()
mean_returns = returns.mean() * 252
cov_matrix = returns.cov() * 252
num_assets = len(tickers)
```

This block computes daily percentage returns, drops missing values, and annualizes the mean returns and covariance matrix by multiplying by 252 (trading days in a year). The number of assets is stored for later use.

4.6 Defining Portfolio Performance

```
def portfolio_performance(weights, mean_returns, cov_matrix,
    risk_free_rate=0.02):
    portfolio_return = np.sum(mean_returns * weights) * 100
    portfolio_volatility = np.sqrt(np.dot(weights.T, np.dot(cov_matrix *
        100, weights)))
    sharpe_ratio = (portfolio_return - risk_free_rate * 100) /
        portfolio_volatility
    return portfolio_return, portfolio_volatility, sharpe_ratio
```

This function calculates the portfolio's expected return, volatility, and Sharpe Ratio given weights, annualized mean returns, and covariance matrix. The risk-free rate is set to 2%.

4.7 Optimizing Portfolio

```
def neg_sharpe_ratio_with_penalty(weights, mean_returns, cov_matrix,
    risk_free_rate=0.02, penalty_factor=50):
    _, _, sharpe = portfolio_performance(weights, mean_returns,
        cov_matrix, risk_free_rate)
    weight_variance = np.var(weights)
    target_weight = 1.0 / num_assets
    deviation_penalty = np.sum((weights - target_weight) ** 2)
    return -sharpe + penalty_factor * (weight_variance + deviation_penalty)

min_weight = 0.01
constraints = (
    {'type': 'eq', 'fun': lambda x: np.sum(x) - 1},
    {'type': 'ineq', 'fun': lambda x: x - min_weight}
)
bounds = tuple((min_weight, 1) for _ in range(num_assets))
initial_guess = np.array([1. / num_assets] * num_assets)

opt_result = minimize(neg_sharpe_ratio_with_penalty, initial_guess,
    args=(mean_returns, cov_matrix),
                      method='SLSQP', bounds=bounds,
                          constraints=constraints, options={'maxiter':
                          2000})
opt_weights = opt_result.x
```

This block defines the optimization objective: maximizing the Sharpe Ratio while penalizing uneven weight distributions. Constraints ensure weights sum to 1 and each is at least 0.01. The SLSQP method optimizes the weights, starting from an equal-weight guess.

4.8 Calculating Portfolio Metrics

```
opt_return, opt_volatility, opt_sharpe =
    portfolio_performance(opt_weights, mean_returns, cov_matrix)
```

This block computes the portfolio's expected return, volatility, and Sharpe Ratio using the optimized weights.

4.9 Computing Risk Metrics

```
daily_portfolio_returns = returns @ opt_weights
portfolio_mean = daily_portfolio_returns.mean()
portfolio_std = daily_portfolio_returns.std()
confidence_level = 0.05
z_score = -1.645
parametric_var = -(portfolio_mean + z_score * portfolio_std) * principal
historical_var = -np.percentile(daily_portfolio_returns, confidence_level
    * 100) * principal
tail_losses = daily_portfolio_returns[daily_portfolio_returns <=
    np.percentile(daily_portfolio_returns, confidence_level * 100)]
cvar = -tail_losses.mean() * principal

def marginal_var(weights, cov_matrix, portfolio_std,
    confidence_level=0.05):
    z_score = -1.645
    marginal = z_score * (cov_matrix @ weights) / portfolio_std
    return marginal * principal

marginal_vars = marginal_var(opt_weights, cov_matrix / 252, portfolio_std)

incremental_vars = []
perturbation = 0.01
for i in range(num_assets):
    new_weights = opt_weights.copy()
    new_weights[i] += perturbation
    new_weights = new_weights / np.sum(new_weights)
    new_portfolio_returns = (returns @ new_weights)
    new_var = -np.percentile(new_portfolio_returns, confidence_level *
        100) * principal
    incremental_vars.append((new_var - historical_var) / perturbation)
```

This block calculates daily risk metrics: Parametric VaR (based on normal distribution assumptions), Historical VaR (based on past data percentiles), CVaR (average loss in the worst-case scenarios), Marginal VaR (risk contribution per instrument), and Incremental VaR (VaR change after rebalancing).

4.10 Portfolio Growth Simulation

```
cumulative_returns = (returns @ opt_weights).cumsum() * principal
cumulative_returns = cumulative_returns + principal
```

This block simulates the portfolio's growth over time by applying the optimized weights to historical returns, scaling by the principal, and adding the initial investment.

4.11 Visualizations

```
plt.figure(figsize=(10, 6))
plt.plot(cumulative_returns.index, cumulative_returns, label='Portfolio Value')
plt.title('Portfolio Growth Over Time (5 Years)')
plt.xlabel('Date')
plt.ylabel('Portfolio Value ($)')
plt.legend()
plt.grid(True)
plt.savefig('portfolio_growth.png')
plt.close()

plt.figure(figsize=(8, 6))
plt.bar(tickers, opt_weights)
plt.title('Optimal Portfolio Weights')
plt.xlabel('ETF')
plt.ylabel('Weight')
plt.savefig('portfolio_weights.png')
plt.close()

plt.figure(figsize=(8, 6))
sns.heatmap(returns.corr(), annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Heatmap of ETFs (5 Years)')
plt.savefig('correlation_heatmap.png')
plt.close()
```

This block generates three visualizations: a line plot of portfolio growth, a bar chart of optimal weights, and a correlation heatmap, saved as PNG files.

4.12 Printing Results

```
print(f"\nOptimal Weights: {dict(zip(tickers, opt_weights))}")
print(f"Expected Annual Return: {opt_return:.2f}%")
print(f"Annual Volatility: {opt_volatility:.2f}%")
print(f"Sharpe Ratio: {opt_sharpe:.2f}")
print(f"Parametric VaR (95%, 1-day): ${parametric_var:,.2f}")
print(f"HistoricalVaR(95%, 1-day): ${historical_var:,.2f}")
print(f"CVaR (95%, 1-day): ${cvar:,.2f}")
print(f"Marginal VaR: {dict(zip(tickers, marginal_vars))}")
print(f"Incremental VaR: {dict(zip(tickers, incremental_vars))}")
print("\nCheck your PyCharm project folder for the visualizations:
    portfolio_growth.png,portfolio_weights.png, and
    correlation_heatmap.png.")
```

This block prints the final results, including optimal weights, expected return, volatility, Sharpe Ratio, risk metrics, and a note about the saved visualizations.

5 Conclusion

The Portfolio Risk and Optimization Engine successfully achieved its goals, delivering a balanced portfolio with optimized weights, a positive expected annual return, low volatility, and a high Sharpe Ratio. The risk metrics provide a clear picture of potential losses, while the visualizations offer insights into portfolio growth, weight distribution, and instrument correlations, making this tool a valuable asset for investors. Future improvements could include dynamic penalty adjustments and additional analytical features.