Dauphine | PSL
UNIVERSITÉ PARIS

Master 272

Economics and Financial Engineering

Multi-Asset Portfolio Manager

by Mariano BENJAMIN
Rayan BIBILONI
Clément HUBERT

Professor Thomas VINCENTI

March 30, 2025

Abstract

This documentation provides an overview of the Multi-Asset Portfolio Manager application, a financial software solution designed for portfolio creation, optimization, and performance analysis across multiple asset classes. The application implements various portfolio optimization strategies based on modern portfolio theory, handles data collection and management from financial markets, visualizes portfolio performance metrics, and provides an intuitive graphical user interface for investment professionals. This document outlines the project's structure, technical implementation details, theoretical foundations of the optimization strategies, and user interaction workflows. The Multi-Asset Portfolio Manager enables investment professionals to make data-driven decisions, optimize portfolio allocations according to different risk profiles, and analyze performance across various market conditions.

Table of contents

Introduction

The Multi-Asset Portfolio Manager addresses a fundamental challenge in investment management: how to optimally allocate capital across various assets to achieve specific financial objectives. Modern portfolio management requires sophisticated tools that can process large amounts of market data, implement complex optimization algorithms, and provide intuitive visualizations to support decision-making. The importance of such tools has grown as financial markets have become increasingly complex and interconnected, with investors seeking exposure to multiple asset classes to achieve diversification benefits.

The application is built upon the foundation of Modern Portfolio Theory (MPT) introduced by Harry Markowitz in 1952, which emphasizes the importance of considering the risk-return relationship in portfolio construction. MPT suggests that by combining assets with different correlation patterns, investors can construct portfolios that maximize expected returns for a given level of risk. This project extends these concepts by implementing various strategies that address different investor needs, from risk minimization to yield maximization.

The portfolio optimization approaches used in this application include minimum variance optimization, mean-variance optimization with turnover constraints, and multi-factor selection models. These methods were chosen for their strong theoretical foundation, empirical validation in academic literature, and practical relevance to investment management. Each strategy is designed to address specific investor objectives, such as risk minimization, transaction cost reduction, or income generation. The application uses historical market data to train these models and generate investment signals, which are then translated into portfolio allocations.

Project Structure

The Multi-Asset Portfolio Manager project follows a modular architecture that separates data management, portfolio optimization, visualization, and user interface components. This separation of concerns enhances maintainability, facilitates testing, and allows for future extensions.

```
multi_asset_portfolio_manager/
 gui/                    # GUI components (Tkinter-based)
    components/         # Individual UI components
       portfolio_creation.py    # Portfolio creation interface
       portfolio_construction.py # Portfolio construction interface
       data_management.py       # Data management interface
       portfolio_comparison.py  # Portfolio comparison interface
    app.py              # Main application entry point
 src/                    # Core functionality
    data_management/    # Data fetching and storage
       database_manager.py      # Database operations
       data_collector.py        # Market data collection
       data_processor.py        # Data preprocessing
    portfolio_optimization/ # Portfolio strategies and optimization
       strategies.py           # Strategy implementations
       backtester.py           # Backtesting framework
       portfolio_metrics.py    # Performance metrics calculation
    visualization/      # Visualization tools
        portfolio_visualizer.py  # Portfolio visualization
 outputs/                # Output files and database
    database/           # SQLite database files
 docs/                   # Documentation
    en/                 # English documentation
    fr/                 # French documentation
```

Each module is responsible for a specific aspect of the application's functionality.

Data Management

Database Manager Implementation

The data management component is built around the `DatabaseManager` class, which serves as the central hub for all database operations. This class implements a repository pattern to abstract database interactions from the rest of the application. The design choice of using SQLite as the database system was made for its simplicity, portability, and sufficient performance for the expected data volumes. The database schema consists of tables for Products, Managers, Clients, Portfolios, Deals, MarketData, and PerformanceMetrics, with appropriate foreign key relationships to maintain data integrity.

- The `DatabaseManager` class uses the `sqlite3` Python module to create connections and execute SQL queries. The class follows the singleton pattern to ensure that only one instance

manages database connections throughout the application lifecycle.

- Table creation is handled using parameterized DDL statements, with transaction management to ensure atomic operations.

Data Collection

Market data collection is handled by the `DataCollector` class, which provides a unified interface for fetching financial data from external sources. The primary data source is Yahoo Finance, accessed through the yfinance library, which offers free access to historical price data for a wide range of financial instruments. This approach was chosen for its accessibility and sufficient data quality for demonstration purposes. In a production environment, this component could be extended to connect to premium data providers for more comprehensive and reliable data.

- The `DataCollector` class implements the adapter pattern to standardize data from different sources.

- The actual data retrieval uses a caching strategy with the database as a persistent cache.

- The `YahooFinanceProvider` implementation uses the `yfinance` package with error handling and rate limiting.

Data Processing

The `DataProcessor` class implements data preprocessing techniques necessary for portfolio optimization. This includes calculating returns, adjusting for corporate actions, handling missing values, and normalizing data for use in optimization algorithms. The processing pipeline uses pandas' powerful data manipulation capabilities to efficiently transform raw market data into formats suitable for portfolio optimization. The class employs exponentially weighted methods for calculating covariance matrices, giving more weight to recent observations, which better represents current market conditions.

- The `DataProcessor` uses numerical methods from `numpy` and time series functionality from ‘pandas‘ to perform calculations.

- For covariance matrix calculation, the implementation uses exponentially weighted methods.

- Missing data handling implements multiple imputation strategies such as Forward Filling, Backward Filling, Interpolation and finally Droping.

Data Storage and Retrieval Logic

Data persistence is achieved through SQLite, with a designed schema that balances normalization principles with query performance. Indexes are created on frequently queried columns to speed up data retrieval operations. The database manager implements prepared statements to prevent SQL injection and optimize query execution. The system follows a caching strat-

egy where frequently accessed data is stored in memory to reduce database calls, significantly improving performance during intensive operations like backtesting.

- The `DatabaseManager` implements caching mechanisms using Python dictionaries to store frequently accessed data.

- Query optimization is implemented using prepared statements and indexes.

- Bulk insert operations for performance optimization

Portfolio Optimization

Strategy Framework Design

The portfolio optimization component is structured around an abstract `PortfolioStrategy` base class that defines the interface for all strategy implementations. This design follows the strategy pattern, allowing different optimization algorithms to be interchanged without modifying the client code. Each concrete strategy must implement a `generate_signals` method that produces portfolio weights based on market data and current portfolio state, enabling a clean separation between the algorithm implementation and its application in portfolio construction.

Low Risk Strategy

The Low Risk strategy aims to minimize portfolio volatility while maintaining acceptable returns. It implements the Minimum Variance Portfolio approach by solving a quadratic optimization problem that minimizes the portfolio variance subject to constraints on weight allocation. Evidence from academic research demonstrates that minimum variance portfolios often achieve better risk-adjusted returns than cap-weighted indices, particularly during market downturns. This strategy is most suitable for conservative investors who prioritize capital preservation over high returns.

- The `LowRiskStrategy` class extends the base `PortfolioStrategy` and implements specific optimization logic using `scipy.optimize`.

- The strategy implementation includes detailed logging for debugging and diagnostics.

Low Turnover Strategy

The Low Turnover strategy extends the mean-variance optimization with an additional penalty term for portfolio changes, effectively balancing the trade-off between expected returns, risk, and turnover costs. This approach is based on research showing that excessive portfolio turnover can significantly erode returns through transaction costs and tax implications. The strategy's implementation uses a combined objective function that incorporates variance, expected return, and a quadratic penalty term for weight changes relative to the current portfolio.

- The `'LowTurnoverStrategy'` implements a modified optimization objective that penalizes deviations from current portfolio weights.

High Yield Equity Strategy

The High Yield Equity strategy focuses on income generation through dividend-paying stocks. It employs a multi-factor approach that evaluates assets based on dividend yield, dividend growth rate, payout ratio sustainability, and historical volatility. This strategy addresses the needs of income-focused investors, particularly in low interest rate environments where traditional fixed income investments may not provide sufficient yield. The implementation uses a scoring system to rank and select assets, with adjustments to ensure adequate diversification and position size control.

- The `'HighYieldEquityStrategy'` implements a multi-factor model for stock selection.

Backtesting Framework

The `'PortfolioBacktester'` class provides a framework for evaluating strategy performance using historical data. It simulates portfolio construction, rebalancing, and performance measurement over specified time periods, generating metrics such as annualized return, volatility, Sharpe ratio, maximum drawdown, and turnover. The backtesting results have demonstrated that the Low Risk strategy achieves the highest Sharpe ratio (1.11), the Low Turnover strategy has the lowest annual turnover (14%), and the High Yield Equity strategy delivers the highest annualized return (12.3%) with corresponding higher risk metrics.

Visualization

Portfolio Visualizer

The visualization module centers around the `'PortfolioVisualizer'` class, which provides methods for generating visual representations of portfolio performance and characteristics. This component uses matplotlib and seaborn to create charts and graphs that help users interpret financial data intuitively. The visualizer implements various chart types, including time series for portfolio value, line charts for returns, bar charts for comparisons, and scatter plots for risk-return analysis.

GUI

Application Architecture

The graphical user interface is built using Tkinter, Python's standard GUI toolkit, following a component-based architecture. The main application window, defined in `'app.py'`, uses a notebook-style interface with tabs for different functionalities, keeping the interface organized and intuitive. Each tab contains specialized frames implemented as separate classes in the

`gui/components/` directory, promoting code modularity and maintainability. This design allows for independent development and testing of each component while ensuring consistent styling and behavior across the application.

The application architecture follows a modular design pattern with a main application class that manages high-level GUI structure.

Portfolio Creation Interface

The portfolio creation interface, implemented in `PortfolioCreationFrame`, provides a user-friendly form for defining new investment portfolios. It includes fields for portfolio name, client selection, strategy choice, and asset universe definition. The interface uses responsive form validation that provides immediate feedback on invalid inputs, improving user experience. When a strategy is selected, the available asset universes are automatically filtered to show only compatible options, demonstrating context-sensitive UI behavior that guides users toward valid configurations.

Data Management User Interface

The data management interface, implemented in `DataManagementFrame`, enables users to fetch, visualize, and manage market data. It presents a multi-panel layout with controls for selecting data sources, date ranges, and specific assets. The interface includes a data table for detailed inspection and interactive charts for visualization. Background threading is used for data-intensive operations to keep the UI responsive, with progress indicators to provide feedback during long-running tasks. It implements multi-threaded data operations.

Portfolio Construction Interface

The portfolio construction interface, implemented in `PortfolioConstructionFrame`, guides users through the process of building and optimizing portfolios. It organizes the workflow into logical steps: portfolio selection, parameter configuration, construction execution, and results visualization. The interface uses a state machine to manage the workflow progression, ensuring that each step is completed appropriately before proceeding. Real-time updates during the optimization process provide users with immediate feedback on the construction progress.

Portfolio Comparison Interface

The portfolio comparison interface, implemented in `PortfolioComparisonFrame`, allows users to analyze multiple portfolios side by side. It uses a dual-list selection box for choosing portfolios, date range selectors for defining the analysis period, and checkbox controls for selecting metrics to compare. The visualization area includes multiple tabs for different chart types, including time series, bar charts, risk-return plots, and summary tables. This component exemplifies interactive data exploration, with changes to selection parameters immediately reflected in the visualizations.