Mean-Variance vs. Mean-Absolute Deviation: A Performance Comparison of Portfolio Optimization Models

Justin Vaughn, Ryosuke Mega, Nicholas Chang

May 2025

1 Introduction

Portfolio optimization is a fundamental challenge in finance, seeking to balance the trade-off between investment return and financial risk. The foundational principle, often attributed to Modern Portfolio Theory (MPT), is that a rational investor will always choose the portfolio with the least risk for a given level of expected return.

The seminal work in MPT is Harry Markowitz's Mean-Variance (MV) model. While revolutionary in theory, the MV model has faced practical limitations. Its reliance on quadratic programming makes it computationally intensive for large-scale problems. Furthermore, it often produces highly diversified portfolios with minuscule weights assigned to many assets, creating significant management and transaction cost challenges.

In response to these difficulties, Hiroshi Konno and Hiroaki Yamazaki introduced the Mean-Absolute Deviation (MAD) model in 1991. This paper provides a direct, empirical comparison of these two foundational models. Our primary goal is to determine which model—the classic quadratic MV or the linear MAD—provides superior risk-adjusted returns when applied to real-world historical stock data under practical constraints. We hypothesize that the MAD model's different measure of risk will prove more robust and effective in an out-of-sample performance test.

2 Mathematical Models

Both optimization models aim to construct a portfolio by assigning a **weight**, w_j, to each asset j. This weight represents the percentage of total capital invested in that asset. The primary goal is to find the optimal vector of weights, $\vec{w}$, that best balances risk and return, based on historical data derived from our Julia script.

2.1 Mean-Variance (MV) Model

The MV model defines risk as the portfolio's **variance**. The objective is to find the weight vector $\vec{w}$ that minimizes this variance, subject to achieving a certain level of expected return. This is a quadratic programming problem.

2.1.1 MV Formulation and Terminology

The optimization problem, as implemented in our "mean variance optimization" function, is:

$$\begin{aligned} \text{Minimize} \quad & \sigma^2(\vec{w}) = \vec{w}^T Q \vec{w} \\ \text{Subject to} \quad & \vec{r}^T \vec{w} \geq \gamma \\ & \vec{w}^T \vec{1} = 1 \\ & \vec{w} \geq 0 \end{aligned}$$

The components in our Julia code correspond to this structure:

- $\vec{w}$ **(Weights)**: This is the vector 'x' in our code. It's the primary variable the model solves for. Each element w_j in the vector represents the percentage of our money allocated to stock j.
- **Objective: Minimize** $\vec{w}^T Q \vec{w}$ **(Portfolio Variance)**: This is the core of the MV model. The goal is to make the portfolio's returns as stable as possible.
  - Q: This is the **covariance matrix**, calculated in our code as 'Q = cov(returns)'. Think of it as a "risk map." The diagonal elements of Q measure the individual volatility of each stock (its variance). The off-diagonal elements measure how pairs of stocks move in relation to each other (their covariance). A positive covariance means two stocks tend to move in the same direction, while a negative covariance means they move in opposite directions. The model uses this entire matrix to find a mix of assets that minimizes total volatility, rewarding diversification among assets that do not move in perfect unison.
- **Constraint** $\vec{r}^T \vec{w} \geq \gamma$ **(Minimum Return)**: This constraint sets the investor's performance goal.
  - $\vec{r}$: This is the vector of mean daily returns, calculated as 'r =vec(Statistics.mean(returns; dims=1))'. It represents the historical average return for each stock in our training data.
  - γ: This is a parameter we can set, representing the minimum acceptable average return for the portfolio. If you set γ higher, you are telling the model "I am not interested in safe, low-return portfolios; find me something more aggressive." This forces the model to take on more risk to meet the higher return target.
- **Constraints** $\vec{w}^T \vec{1} = 1$ **and** $\vec{w} \geq 0$: These are standard portfolio constraints. The first ensures 100% of the capital is invested ('sum(x)

==

1'). The second prohibits short selling ('x *ge* 0').

2.2 Mean-Absolute Deviation (MAD) Model

The MAD model's key innovation is its definition of risk. Instead of variance, which squares deviations from the mean, it uses the **mean of the absolute deviations**. This makes it less sensitive to the large, infrequent price swings often seen in financial markets.

2.2.1 MAD Formulation and Linearization

The MAD model minimizes the average daily "surprise" or deviation. While its mathematical definition includes an absolute value function, our 'MAD portfolio optimization' function implements it as a more efficient **linear program** using a standard linearization technique.

The formal optimization problem solved by our code is:

$$\begin{aligned}
\text{Minimize} \quad & \frac{1}{T}\sum_{t=1}^{T}(u_t + v_t) \\
\text{Subject to} \quad & \left(\sum_{j=1}^{n} r_{jt} w_j\right) - \left(\sum_{j=1}^{n} r_j w_j\right) = u_t - v_t \quad \forall t \in T \\
& \vec{r}^T \vec{w} \geq \gamma \\
& \vec{w}^T \vec{1} = 1 \\
& \vec{w} \geq 0, \quad u_t \geq 0, \quad v_t \geq 0
\end{aligned}$$

The components in our 'MAD portfolio optimization' function correspond to this structure:

- **Objective: Minimize** $\frac{1}{T}\sum(u_t + v_t)$: This is the core of the MAD model's risk measure.
  - u_t and v_t: These are non-negative auxiliary variables created in our code for each day ('@variable(m, u[m_TIME] *ge* 0)'). They are a mathematical trick to represent the absolute value. For any given day t, u_t captures the "upside surprise" (how much the portfolio performed *better* than its average), and v_t captures the "downside surprise" (how much it performed *worse*). Since only one can be non-zero at a time, their sum $(u_t + v_t)$ is exactly the absolute value of the deviation. The objective '@objective(m, Min, sum(u[t] + v[t] ...)' seeks to make the average of these daily surprises as small as possible.
- **Deviation Constraint**: This constraint, '@constraint(m, portfolio_returns[t...])', is the core of the linearization. It defines the relationship between the portfolio's actual return on a specific day ($\sum r_{jt} w_j$, or 'sum(daily_returns[t,i]...)' in the code) and its overall expected average return ($\sum r_j w_j$, or 'sum(mean_return[j]...)'). The difference is set equal to $u_t - v_t$.
- **Other Constraints**: The minimum return (γ, or 'm_R' in our code's parameters), budget, and no-short-selling constraints function identically to those in the MV model, ensuring a fair comparison between the two approaches.

3 Implementation

For our research, we selected stocks from SP 500 because its the best indicator of US equities and it is a diverse selection of commonly traded stock. Data was collected from the kaggle data set S&P 500 stock data by Cam Nugent. We used Julia code to implement the Mean Variance and Mean Absolute Deviation models. The Mean Absolute Deviation model was adapted to Julia from python code in *Hands-On Mathematical Optimization with Python.* The Julia code for the mean Variance Model was directly implemented from the jump.dev portfolio optimization example. Additional code was written to collect and process the data. All stocks with missing prices were deleted. There were initially 505 stocks and 1259 trading days, but after cleaning there were 470 stocks and 1259 trading days.

Portfolios with the top 75, 150, and 200 performing stocks from these dates were constructed. Instead of optimizing based on all the data, the earliest 60% of the data was chosen as training data, while the later 40% was used to test the effectiveness of each model. This meant training data was on the first 754 days, and the testing period was on the remaining 504 days. This was done because the time constraints meant we could not wait for a significant period of time to pass while observing each model's performance.

```
#Mean Absolute Deviation (MAD) vs Mean Variance optimization for stocks.
#  Tests different portfolio sizes in the first approximately 3 years
# then evaluates the performance on the remaining data.

# load packages
using CSV
using DataFrames
using JuMP
using GLPK
using Ipopt
using Statistics
using Plots

# Load and clean stock data from CSV
function load_stock_data(filepath)
    df = CSV.read(filepath, DataFrame)

    stocks = unique(df.Name)
    dates = sort(unique(df.date))

    println("Found ", length(stocks), " stocks and ", length(dates), " trading days")

    # Create close price matrix: rows=dates, columns=stocks
    price_matrix = Matrix{Float64}(undef, length(dates), length(stocks))

    for (stock_idx, stock) in enumerate(stocks)
        stock_data = filter(row -> row.Name == stock, df)
        stock_data = sort(stock_data, :date)

        for (date_idx, date) in enumerate(dates)
            matching_row = filter(row -> row.date == date, stock_data)
            price_matrix[date_idx, stock_idx] = isempty(matching_row) ? NaN : matching_row[1, :close]
        end
    end

    # Get rid of columns (stocks) with any NaN values
    valid_stocks_mask = [!any(isnan.(price_matrix[:, j])) for j in 1:size(price_matrix, 2)]

    # Filtered stocks and price matrix
    price_matrix = price_matrix[:, valid_stocks_mask]
    stocks = stocks[valid_stocks_mask]

    println("After cleaning: ", length(stocks), " stocks and ", length(dates), " trading days")
```

```
    return price_matrix, stocks, dates
end

# Calculate daily returns from price matrix
function calc_returns(prices)
    n_dates, n_stocks = size(prices)
    returns = Matrix{Float64}(undef, n_dates-1, n_stocks)

    for j in 1:n_stocks
        for i in 1:(n_dates-1)
            returns[i, j] = (prices[i+1, j] - prices[i, j]) / prices[i, j]
        end
    end

    return returns
end

# Select the top performing stocks by mean return
function select_top_stocks(returns, stocks, n_top)
    mean_returns = vec(mean(returns, dims=1))
    top_idx = sortperm(mean_returns, rev=true)[1:min(n_top, length(mean_returns))]

    return returns[:, top_idx], stocks[top_idx], mean_returns[top_idx]
end

# MAD portfolio optimization
function optimize_mad(returns, mean_returns)
    n_assets = size(returns, 2)
    n_periods = size(returns, 1)

    model = Model(GLPK.Optimizer)
    set_silent(model)

    # Decision variables
    @variable(model, w[1:n_assets] >= 0)     # portfolio weights (non-negative)
    @variable(model, u[1:n_periods] >= 0)    # positive deviations
    @variable(model, v[1:n_periods] >= 0)    # negative deviations

    # Minimize mean absolute deviation
    @objective(model, Min, sum(u[t] + v[t] for t in 1:n_periods) / n_periods)

    # Portfolio return deviations
```

```
    @constraint(model, deviations[t in 1:n_periods],
        sum(returns[t, i] * w[i] for i in 1:n_assets) ==
        sum(w[i] * mean_returns[i] for i in 1:n_assets) + u[t] - v[t])

    # Portfolio constraints
    @constraint(model, sum(w) == 1)          # weights sum to 1

    optimize!(model)

    return termination_status(model) == MOI.OPTIMAL ? [value(w[i]) for i in 1:n_assets] : nothing
end

# Mean Variance (Markowitz) optimization
function optimize_markowitz(returns, mean_returns; target_return=nothing)
    n_assets = size(returns, 2)
    cov_matrix = cov(returns)

    model = Model(Ipopt.Optimizer)
    set_silent(model)

    @variable(model, w[1:n_assets] >= 0)     # non-negative weights
    @objective(model, Min, w' * cov_matrix * w)     # minimize variance
    @constraint(model, sum(w) == 1)                 # weights sum to 1

    if target_return !== nothing
        @constraint(model, mean_returns' * w >= target_return)
    end

    optimize!(model)

    weights = termination_status(model) in [MOI.OPTIMAL, MOI.LOCALLY_SOLVED] ?
              [value(w[i]) for i in 1:n_assets] : nothing

    return weights, cov_matrix
end

# Evaluate portfolio performance on test data
function backtest_portfolio(weights, test_returns)
    portfolio_returns = test_returns * weights

    # Calculate performance metrics
    returns_plus_one = 1 .+ portfolio_returns
    total_return = prod(returns_plus_one) - 1
```

```
    annual_return = (1 + total_return)^(252/length(portfolio_returns)) - 1
    volatility = std(portfolio_returns) * sqrt(252)
    sharpe = annual_return / volatility

    # Calculate max drawdown
    cumulative = cumprod(returns_plus_one)
    running_max = accumulate(max, cumulative)
    drawdowns = (cumulative .- running_max) ./ running_max
    max_dd = minimum(drawdowns)

    return annual_return, volatility, sharpe, max_dd
end
```

```
# Create pie chart for portfolio weights
function create_portfolio_pie_chart(weights, stock_names, method_name, n_stocks)
    # Only show stocks with meaningful weights (>= 1%)
    min_weight = 0.01
    significant_indices = findall(w -> w >= min_weight, weights)

    # Prepare data for plotting
    plot_weights = weights[significant_indices]
    plot_labels = stock_names[significant_indices]

    # Group small weights into "Others"
    other_weight = sum(weights[weights .< min_weight])

    if other_weight > 0.001
        plot_weights = vcat(plot_weights, other_weight)
        plot_labels = vcat(plot_labels, "Others")
    end

    # Convert to percentages for display
    plot_percentages = plot_weights * 100

    # Create the pie chart
    pie_chart = pie(plot_labels, plot_percentages,
                    title = string(method_name, " Portfolio (Top ", string(n_stocks), " stocks)"),
                    legend=:outertopright,
                    size=(800, 600))

    return pie_chart
end

# Main analysis function with pie charts
function run_analysis(filepath; create_charts=true)
    println("=== Portfolio Optimization Analysis ===")
    println("Comparing MAD vs Mean Variance optimization")

    # Load data
    prices, all_stocks, dates = load_stock_data(filepath)
    returns = calc_returns(prices)

    # Split into training (60%) and testing (40%)
    n_train = Int(floor(size(returns, 1) * 0.6))
    train_returns = returns[1:n_train, :]
    test_returns = returns[(n_train+1):end, :]
```

```
println("Training: ", n_train, " days, Testing: ", size(test_returns, 1), " days")

# Test different portfolio sizes
portfolio_sizes = [75, 150, 200]

println("\nResults Summary:")
println("Portfolio | Method    | Return  | Risk    | Sharpe  | Max DD ")
println("-" ^ 58)

for n_stocks in portfolio_sizes
    # Select top performers
    selected_returns, selected_stocks, mean_returns = select_top_stocks(
        train_returns, all_stocks, n_stocks)

    # Get corresponding test returns for selected stocks
    stock_indices = [findfirst(x -> x == stock, all_stocks) for stock in selected_stocks]
    test_selected = test_returns[:, stock_indices]

    # MAD optimization
    mad_weights = optimize_mad(selected_returns, mean_returns)
    if mad_weights !== nothing
        ret, vol, sharpe, dd = backtest_portfolio(mad_weights, test_selected)
        @printf("Top %-4d | %-8s | %6.2f%% | %6.2f%% | %6.3f | %6.2f%%\n",
                n_stocks, "MAD", ret*100, vol*100, sharpe, dd*100)

        # Create pie chart for MAD
        if create_charts
            mad_chart = create_portfolio_pie_chart(mad_weights, selected_stocks, "MAD", n_stocks)
            display(mad_chart)
        end
    end

    # Mean Variance optimization
    mv_weights, cov_mat = optimize_markowitz(selected_returns, mean_returns)
    if mv_weights !== nothing
        ret, vol, sharpe, dd = backtest_portfolio(mv_weights, test_selected)
        @printf("Top %-4d | %-8s | %6.2f%% | %6.2f%% | %6.3f | %6.2f%%\n",
                n_stocks, "Markowitz", ret*100, vol*100, sharpe, dd*100)

        # Create pie chart for Markowitz
        if create_charts
            mv_chart = create_portfolio_pie_chart(mv_weights, selected_stocks, "Mean-Variance", n_s
```

```
                display(mv_chart)
            end
        end

        println()
    end
end

# Run analysis
println("Ready to analyze portfolio optimization strategies")
println("Usage: run_analysis(\"your_stock_data.csv\") - with pie charts")
println("Usage: run_analysis(\"your_stock_data.csv\", create_charts=false) - without charts")
println("This compares MAD and Mean Variance optimization on different portfolio sizes")
```

4 Results and Discussion

The analysis was performed on portfolios of 75, 150, and 200 stocks. The models were optimized on the training data, and the resulting optimal portfolios were then evaluated over the subsequent two-year testing period. The performance summary is presented below.

Table 1: Performance Summary (Testing Period)

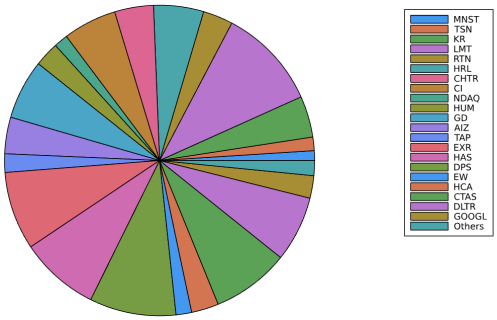
Portfolio Size	Method	Ann. Return	Volatility	Sharpe Ratio	Max DD
Top 75	MAD	**20.36%**	9.49%	**2.144**	**-8.27%**
	Mean-Var	17.29%	9.56%	1.808	-8.84%
Top 150	MAD	**14.51%**	9.19%	**1.578**	-8.96%
	Mean-Var	13.80%	**8.95%**	1.541	**-8.61%**
Top 200	MAD	**17.75%**	8.89%	**1.996**	**-7.96%**
	Mean-Var	15.21%	**8.88%**	1.712	-8.32%

The results from our backtest are remarkably consistent and clear. In all three scenarios (75, 150, and 200 stocks), the **Mean-Absolute Deviation (MAD) model generated portfolios with a higher Annualized Return and a superior risk-adjusted return (Sharpe Ratio)** compared to the traditional Mean-Variance (MV) model.

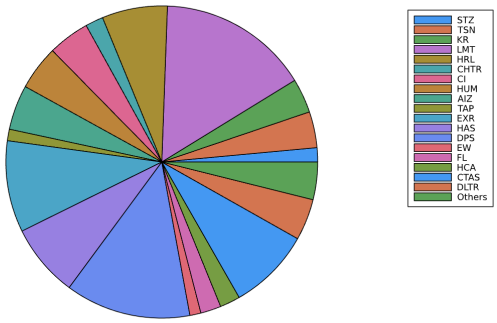
For instance, in the 75-stock portfolio, the MAD model achieved an annualized return of 20.36% and a Sharpe Ratio of 2.144, significantly outperforming the MV model's 17.29% return and 1.808 Sharpe Ratio. This pattern holds across all portfolio sizes. While the MV model did produce portfolios with marginally lower volatility in the 150 and 200-stock cases, the higher returns from the MAD portfolios more than compensated for this, leading to a better overall risk-reward profile as measured by the Sharpe Ratio. The MAD model also consistently resulted in a lower or comparable maximum drawdown, suggesting better capital preservation during downturns.

These findings strongly support the hypothesis that the MAD model is a more robust optimization technique for real-world financial data, which often deviates from the assumption of normality. The MV model's use of squared deviations likely over-penalizes extreme events (fat tails) common in stock returns, leading to overly conservative and ultimately lower-performing portfolios. In contrast, the MAD model's linear treatment of deviations appears to produce more practical and effective asset allocations.

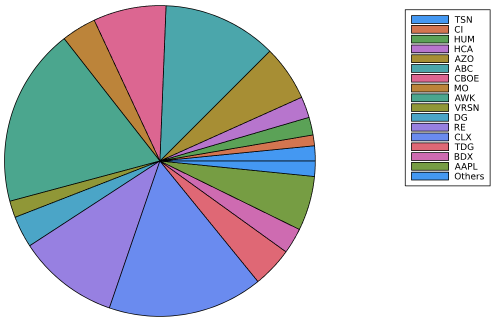
MAD Portfolio (Top 75 stocks)



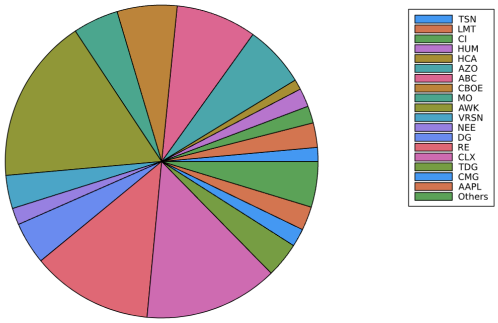
Mean-Variance Portfolio (Top 75 stocks)



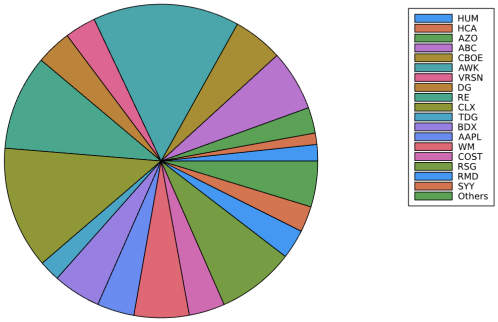
MAD Portfolio (Top 150 stocks)



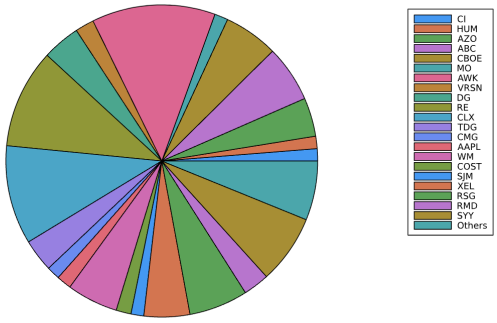
Mean-Variance Portfolio (Top 150 stocks)



MAD Portfolio (Top 200 stocks)



Mean-Variance Portfolio (Top 200 stocks)



5 Conclusion

This project set out to compare the classical Mean-Variance portfolio optimization model with the more modern Mean-Absolute Deviation model. By optimizing portfolios on three years of historical S&P 500 data and testing their performance on the subsequent two years, we found compelling evidence of the MAD model's superiority.

Across portfolios of 75, 150, and 200 stocks, the MAD model consistently delivered higher total returns and higher risk-adjusted returns (Sharpe Ratio) than the Mean-Variance model. We conclude that this is likely because the MAD model's risk measure is more robust to the non-normal, "fat-tailed" nature of real-world stock returns. By not squaring deviations, it is less sensitive to extreme outliers and produces portfolios that perform better in out-of-sample testing. Our results affirm that while Markowitz's Mean-Variance model is a cornerstone of financial theory, the Mean-Absolute Deviation model represents a more practical and effective tool for modern portfolio optimization.

6 Bibliography

Postek, K., Zocca, A., Gromicho, J. a. S., and Kantor, J. C. (2024). *Hands-On Mathematical Optimization with Python*. O'Reilly Media.

Konno, H., and Yamazaki, H. (1991). Mean-Absolute Deviation Portfolio Optimization and Its Applications to Tokyo Stock Market. *Management Science*, 37(5), 519-531.

"Example: Portfolio Optimization · Jump." · JuMP, jump.dev/JuMP.jl/stable/tutorials/nonlinear/portfolio/. Accessed 9 June 2025.

Postek, K., Zocca, A., Gromicho, J. a. S., & Kantor, J. C. (2024). Hands-On Mathematical Optimization with Python. Cambridge University Press. (25-28)

Nugent, Cam. "S&P 500 Stock Data." Kaggle, 10 Feb. 2018, www.kaggle.com/datasets/camnugent/sandp500/data.