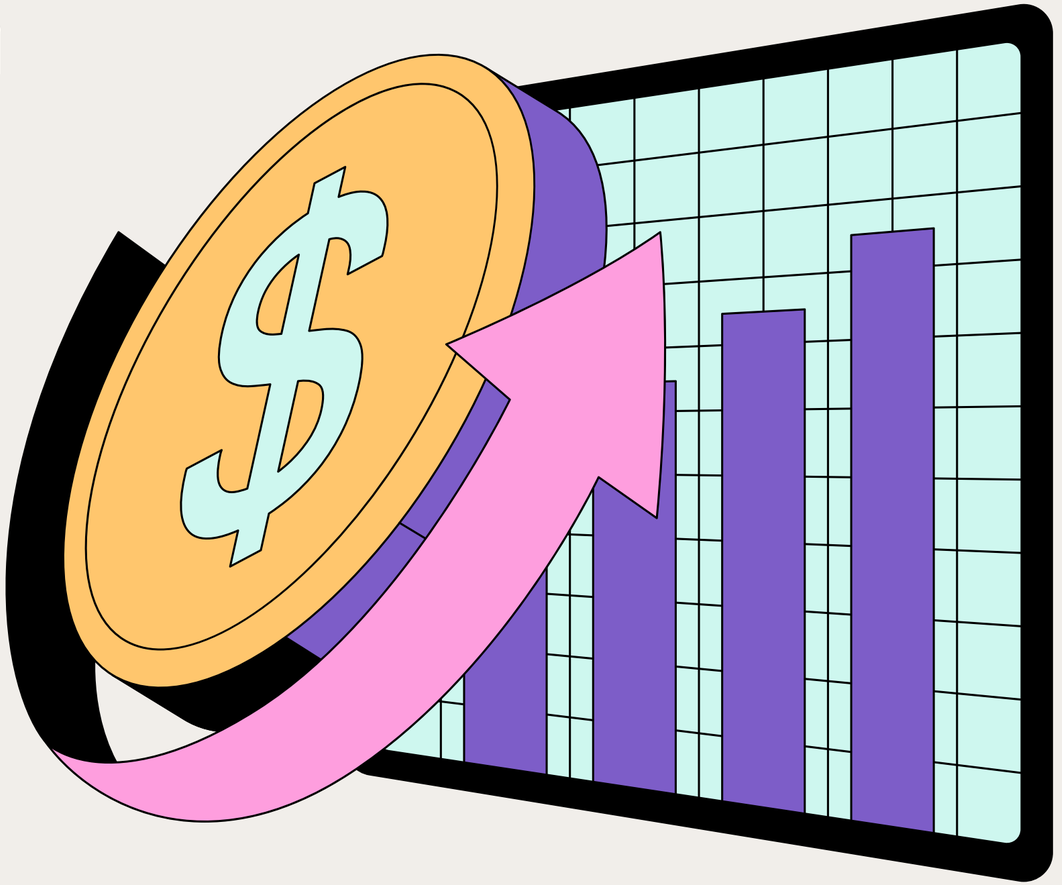
# Portfolio Optimisation using Dimensionality Reduction and Sparsification

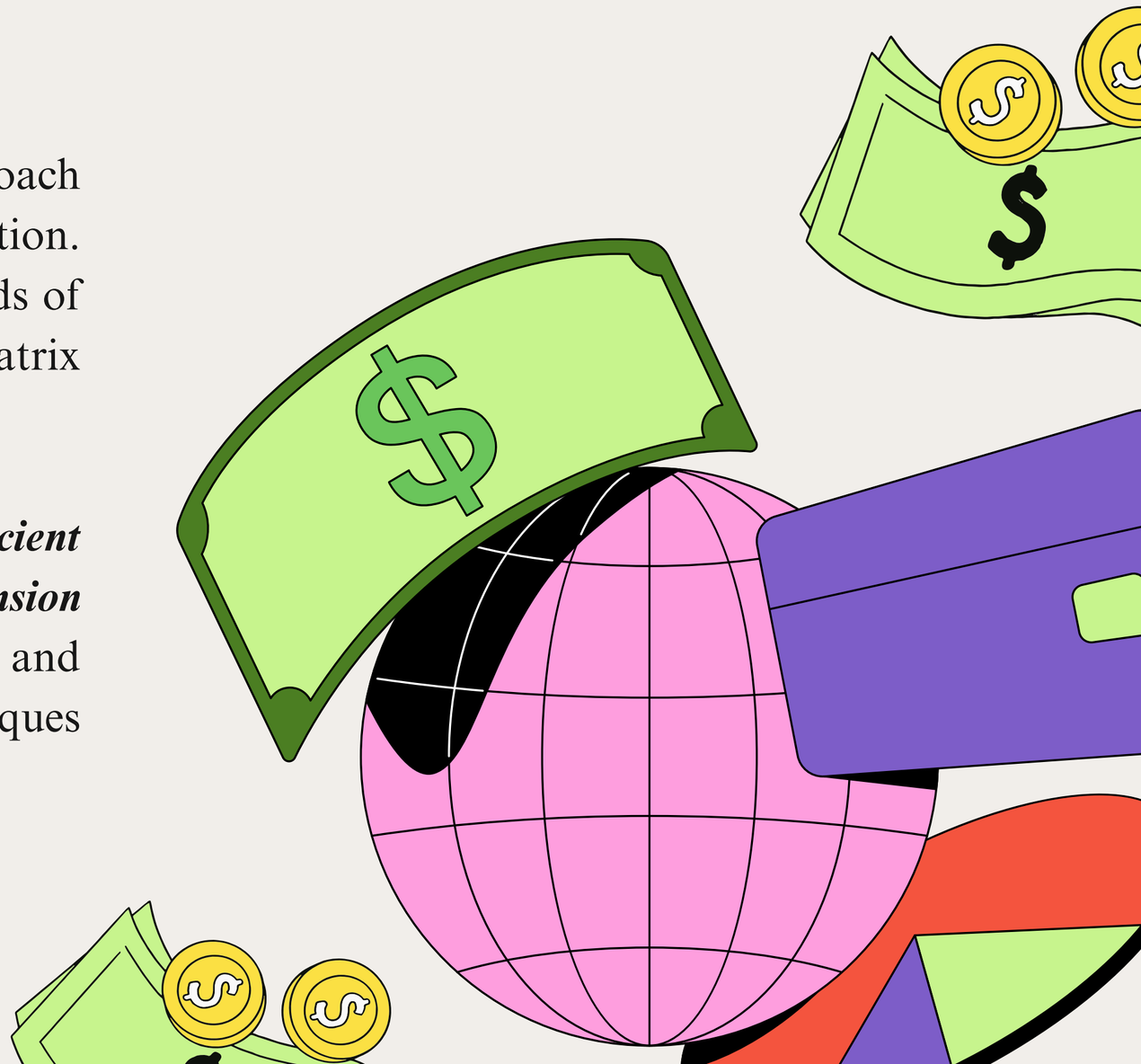
Presented by
Shruti Sharma (MDS202435)
Shuvodeep Dutta (MDS202436)
Srishti Lakhotia (MDS202437)



# Introduction

The *Markowitz Mean-Variance model* is a classic approach for balancing risk and return in portfolio optimization. However, when applied to large portfolios with hundreds of assets, its reliance on a large, dense covariance matrix becomes a major computational bottleneck.

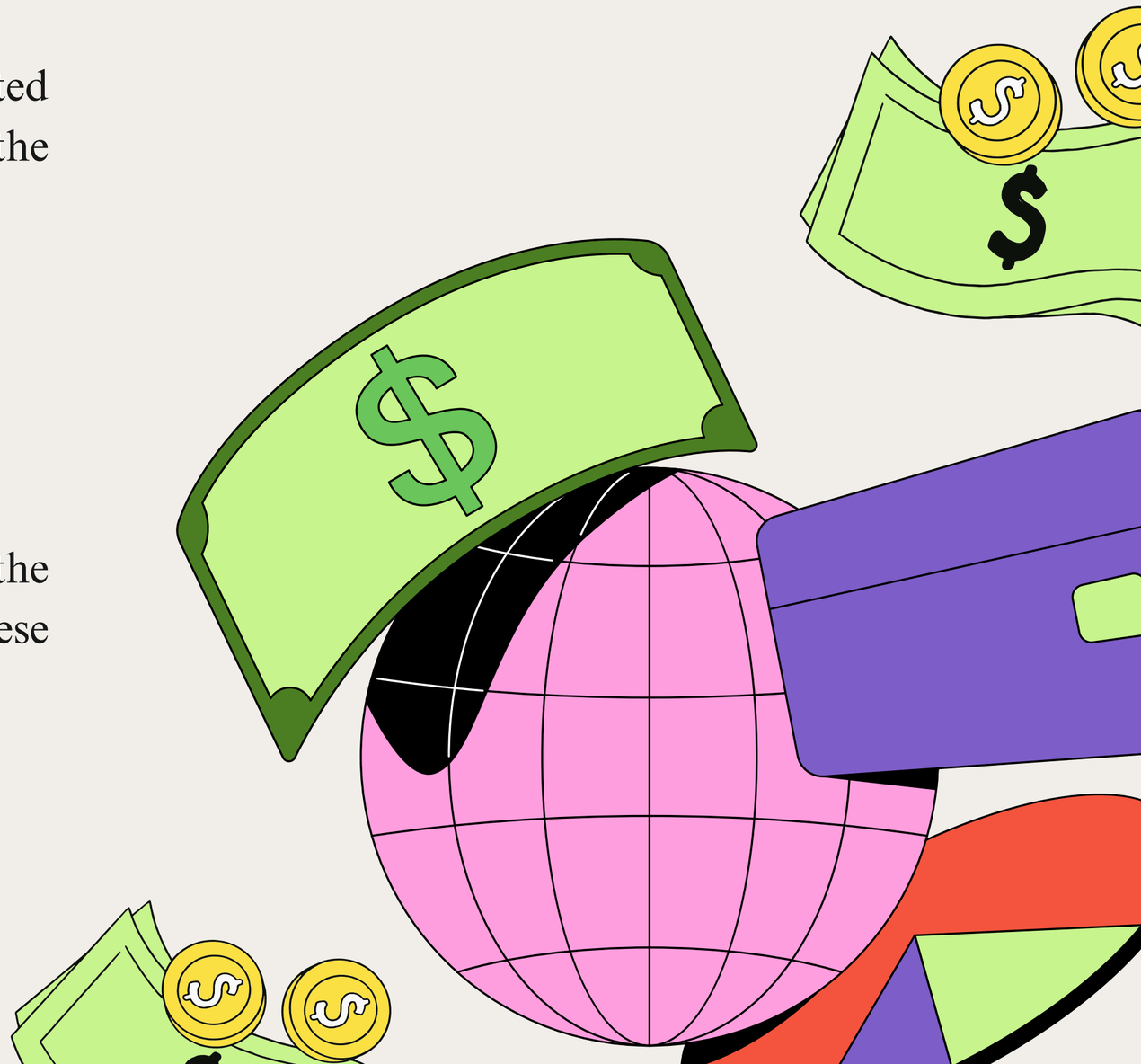
Our report is inspired by the research conducted in ***Efficient Solution of Portfolio Optimization Problems via Dimension Reduction and Sparsification*** by *Cassidy K. Buhler* and *Hande Y. Benson*, which explores linear algebra techniques to address this issue.



To gain practical insights into the approaches presented in the research paper, we also look at the results of the implementation the proposed methods:

- dimension reduction using neural networks,
- a linear programming-based formulation, and
- covariance matrix sparsification.

The goal of this implementation was to evaluate the computational efficiency and accuracy of these techniques when applied to a financial dataset.



# What is Portfolio Optimization?

**Goal:** Find the best way to invest your money across different assets (like stocks) to balance two things:

- Expected return: How much you hope to earn.
- Risk: How much the returns might go up and down (volatility).

**Markowitz Model:** The classic way to do this is called the Markowitz mean-variance model. It uses:

- Expected returns for each asset.
- Covariance matrix: A big table showing how each pair of assets moves together (if they go up and down together, or in opposite directions).
- Efficient Frontier: For every level of risk, there's a "best" portfolio. The set of all these best portfolios is called the efficient frontier.

# The Markowitz Model

Let $p_{t,j}$ represent the (known) closing price for stock $j = 1, \ldots, N$ on day $t = 1, \ldots, (T-1)$. The return $x_{t,j}$ for stock $j = 1, \ldots, N$ on day $t = 2, \ldots, (T-1)$ is calculated as

$$x_{t,j} = \frac{p_{t,j} - p_{t-1,t}}{p_{t-1,j}}. \tag{1}$$

For portfolio weights $w \in \mathcal{R}^N$, the portfolio return at time $t = 2, \ldots, (T-1)$ is computed by

$$R_t = \sum_{j=1}^{N} w_j x_{t,j}. \tag{2}$$

The portfolio return on day $T$, $\mathbf{R}_T$, is a random variable with

$$\mathbb{E}[\mathbf{R}_T] = \mu^T w, \qquad \mathbb{V}[\mathbf{R}_T] = w^T \Sigma w \tag{3}$$

where $\Sigma = \mathrm{cov}(X)$ and $\mu_j = \mathbb{E}[x_{T,j}],\ j = 1, ..., N$.

The Markowitz model is formulated as:

$$\begin{aligned} \max_{w} \quad & \mu^T w - \lambda w^T \Sigma w \\ \text{s.t.} \quad & e^T w = 1 \\ & w \geq 0 \end{aligned} \tag{4}$$

where $\lambda$ is the risk aversion parameter, varying which, we can obtain the *efficient frontier*.

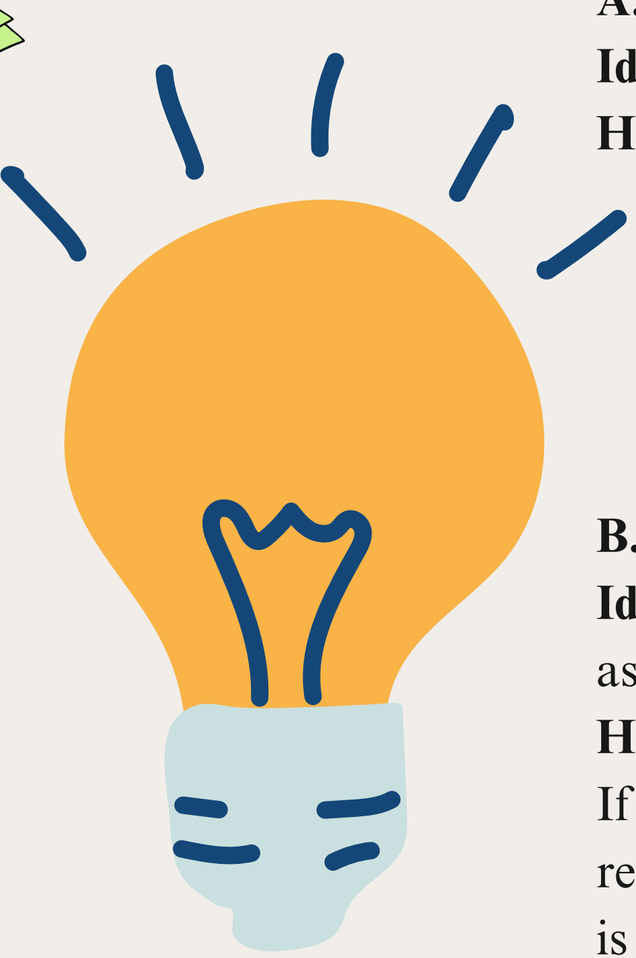
# **The problem :** Why is this hard for big portfolios?

The Markowitz Model is simple and has many applications even outside the realm of investing. Making the solution more efficient to reach can be beneficial across fields.

In addition, advancements in machine learning and other data science techniques have improved the ability to more accurately calculate $\mu$ and $\Sigma$.

The model unfortunately suffers from an efficiency problem due to $\Sigma$: it is large and dense. When you have hundreds of assets, the covariance matrix becomes huge and dense (lots of numbers, most not zero). Calculating the best portfolio gets slow and uses a lot of computer power. But, in practice, only a small number of assets end up in the best portfolios.

# **Solution :** Make the Problem Smaller and Simpler



The paper suggests two main ways to make the calculations faster:

**A. Dimension Reduction**

**Idea:** Predict which assets are likely to be in the best portfolios, and only focus on those.

**How?**

- Machine Learning (Neural Networks): Use a type of AI (LSTM networks) to look at past data and guess which assets will be chosen.
- Linear Programming (LP): Solve a simpler version of the problem to see which assets are likely to be picked.

**B. Sparsification**

**Idea**: Make the covariance matrix mostly zeros by ignoring weak relationships between assets.

**How?**

If two assets don't move together much (their correlation is close to zero), set their relationship to zero in the matrix. This makes the matrix "sparse" (mostly zeros), which is much faster for computers to handle. Make sure the new matrix still has the mathematical properties needed for the optimization to work.

# Method 1 : Reduction using LSTMs

The LSTM network architecture is notably well-suited for financial data, and any time series data due to its capacity to “hold memory”.

Here, instead of directly predicting stock prices or returns, the network classifies each stock into one of two groups:

- Class 0 for stocks that are never selected in any optimal portfolio across various risk preferences, and
- Class 1 otherwise.

This classification enables us to exclude unselected stocks and build a reduced covariance matrix.

The training process includes first splitting the historical daily returns matrix into train and test splits. Using the training set, we then classify the stocks into the two classes. The model is then tested

**Our LSTM learns the patterns using 5 layers.**

1. **Input Layer:** Accepts the input sequence $X_{train}$
2. **LSTM Layer:**
    - 150 hidden nodes
    - Hyperbolic tangent function for state activation
    - Sigmoid function for gate activation
    - Input and recurrent weights initialized with Glorot and Orthogonal strategies
3. **Fully Connected Layer:**
    - Takes in the output from the previous layer
    - Reshapes the data to prepare for the classification
4. **Softmax Layer**: Converts scores into class probabilities
5. **Weighted Cross-Entropy Output:**
    - Computes the loss between the predicted class probabilities (from the Softmax layer) and the true labels.
    - Accounts for class imbalance using weights.

# Method 2: Reformulating Markowitz as Linear Programming

The Markowitz model, which, uses a quadratic objective function because it directly incorporates variance (risk) through the covariance matrix (Σ). This creates a mathematically complex problem that becomes computationally expensive with many assets.

The key innovation in the paper is reformulating this quadratic problem into a linear programming (LP) problem by:

1. **Replacing quadratic risk with absolute deviation:**
   - Instead of using variance ($w^T \Sigma w$), which is quadratic, the approach uses absolute deviations ($\|A^T w\|_1$)
   - These deviations measure how much each portfolio return differs from its mean
   - The matrix A contains centered returns (each return minus the average return)
2. **Introducing auxiliary variables to handle absolute values:**
   - Absolute values aren't naturally linear, so the formulation introduces variables $v_1$, $v_2$, etc.
   - These v variables represent the magnitude of deviations without caring about direction (positive/negative)
   - The constraints $-v \leq A^T w \leq v$ ensure v captures the absolute values

We introduce a new penalty parameter: $\gamma \geq 0$

The covariance matrix can be written as,

$$\Sigma = A^T A, \text{ where } A = X - \bar{X}, \bar{X}_{ij} = \frac{1}{T-1} \sum_{t=1}^{T-1} X_{t,j}.$$

Matrix A represents the deviation of the actual return from its expectation. The covariance term in the Markowitz model measures the magnitude of this deviation in a quadratic sense.

To form the LP, we instead measure it in an absolute sense:

$$\begin{aligned} \max_w \quad & \gamma \mu^T w - \|A^T w\|_1 \\ \text{s.t.} \quad & e^T w = 1 \\ & w \geq 0. \end{aligned}$$

We introduce an auxiliary variable *v* to complete the reformulation:

**Advantages Over Markowitz**

1. Speed: Solves in microseconds vs. milliseconds for quadratic problems.
2. Scalability: Handles 10,000+ assets efficiently.
3. Stability: Less sensitive to outliers in returns data.
4. Interpretability: Absolute deviation is easier to understand than variance.

$$\begin{aligned} \max_w \quad & \gamma \mu^T w - \frac{1}{T} \sum_{i=1}^{T} v_i \\ \text{s.t.} \quad & -v \leq \mathbf{A}^T w \leq v \\ & e^T w = 1 \\ & w, v \geq 0. \end{aligned}$$

# Method 3: Sparsification by Correlation

$$\boldsymbol{\Sigma} = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} & 0 & \Sigma_{14} \\ \Sigma_{21} & \Sigma_{22} & 0 & 0 \\ 0 & 0 & \Sigma_{33} & \Sigma_{34} \\ \Sigma_{41} & 0 & \Sigma_{43} & \Sigma_{44} \end{bmatrix}$$

We replace the covariance matrix in our quadratic formulation with a **sparse matrix** obtained by replacing entries corresponding to small correlations with zero and follow it with partial completion to ensure PSD-ness.

**Why?**

- Buhler and Benson claim that the PSD covariance matrix is likely to be diagonally dominated. The risk of an asset is then usually much more important in deciding its weight in the optimal portfolio than how it relates to other assets.
- Large covariance entries indicate strong, persistent correlations likely to continue in the future. Small correlations can be assumed insignificant and replaced with zeros, resulting in a sparse covariance matrix.

**Challenge?**

- We need to determine an appropriate definition of a small correlation, and
- The resulting sparse covariance matrix needs to be positive semidefinite.

**Why is maintaining PSD important?**

Markowitz model is a convex optimisation problem. To ensure it remains convex which is essential for finding the global optimum, we need to maintain PSD.

Let $\theta$ be the dense $N \times N$ correlation matrix, $\tau$ be a threshold value (with $0 \leq \tau < 1$) obtained from the unique values of $\theta$, and $\hat{\Sigma}(\tau)$ be the sparse $N \times N$ covariance matrix defined for threshold $\tau$. We propose the following simple scheme to sparsify $\Sigma$:

$$\hat{\Sigma}_{ij}(\tau) = \begin{cases} 0 & -\tau \leq \theta_{ij} \leq \tau \\ \Sigma_{ij} & \text{otherwise} \end{cases}, \text{ for all } i, j = 1, \ldots, N$$

**Algorithm 1** Partial Matrix-Completion for Threshold $\tau$

1: set $\hat{\Sigma}(\tau)$ according to Equation (7).
2: $\hat{n} \leftarrow$ column(s) where there exists an off-diagonal non-zero entry in $\hat{\Sigma}$
3: **for** $i \in \hat{n}$ and $j \in \hat{n}$ **do**
4: $\quad \hat{\Sigma}_{ij}(\tau) = \Sigma_{ij}$
5: **end for**

Essentially, we set covariances to 0 for which corresponding absolute correlations are below a threshold and then partially fill the matrix to maintain definiteness.

It can be shown that the algorithm proposed by Buhler and Benson, will always render a positive semidefinite matrix, thereby ensuring that this sparsification always yields a convex quadratic programming problem.

**Python Code:**

```python
def sparsify_cov(returns, threshold=0.3):

    corr = returns.corr() # create a correlation matrix
    sparse_corr = corr.copy() # save a copy of it
    sparse_corr[np.abs(sparse_corr) < threshold] = 0 # values less than the threshold are replaced with 0 to create sparse matrix
    np.fill_diagonal(sparse_corr.values, 1) # diagonal values filled with 1 cause correlation with itself is 1

    # convert our correlation matrix to a covariance matrix using Cov(i,j)=Corr(i,j)×std(i)×std(j)
    std = returns.std()
    sparse_cov = pd.DataFrame(np.outer(std, std) * sparse_corr,
                              index=returns.columns, columns=returns.columns)

    # Ensure positive definiteness
    eigval, eigvec = linalg.eigh(sparse_cov)
    eigval[eigval < 1e-8] = 1e-8 # replacing very small or negative eigen values with small positive values
    sparse_cov_pd = pd.DataFrame(eigvec @ np.diag(eigval) @ eigvec.T, # reconstruct the matrix with modified eigenvalues
                                 index=returns.columns, columns=returns.columns)

    return sparse_cov_pd, sparse_corr
```

The sparsification method improved runtimes without reduction in size because reducing non zero elements is much more efficient than simply making the matrix smaller in dimension. This is so because the algorithm skips over the zeroes saving memory and computation times.

# Implementation

To gain practical insights into the approaches presented in the research paper, let’s look at the results of the implementation presented in the paper as well as our implementation of the proposed methods.

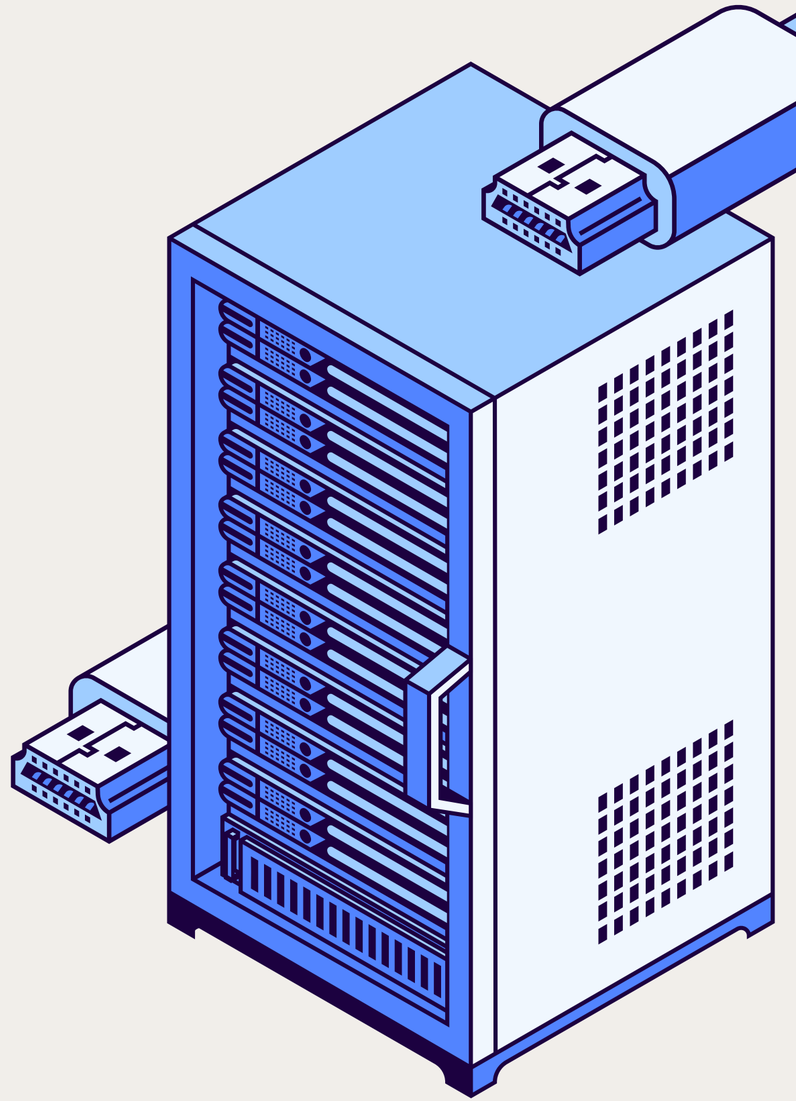
The goal of the implementation was to evaluate the computational efficiency and accuracy of these techniques when applied to a financial dataset.

| | **Benson and Buhler** | **Us** |
|---|---|---|
| Programming Language(s) used | Python, Matlab | Python |
| Optimization Solver | Gurobi Optimizer v10.0.02 | Gurobi Optimizer |
| Libraries used | Matlab Deep Learning Toolbox | Scikit-Learn |

# Data used

The financial data used in the paper was collected from Yahoo! Finance over the dates January 23rd 2012 to December 31st 2019 for 374 firms selected from the S&P500 index.

The returns were computed using the percentage change of the closing price.

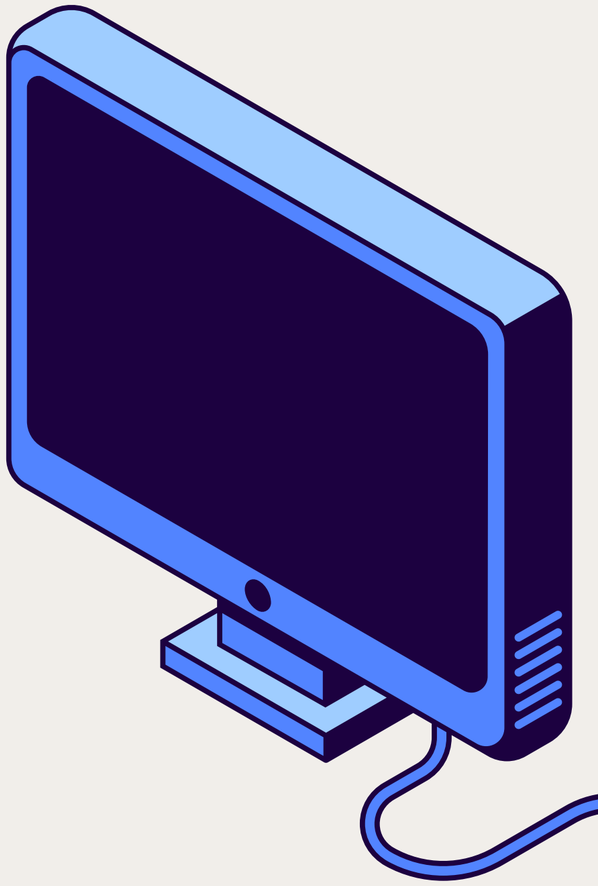


# Numerical Testing

For reduction by neural networks, Buhler and Benson partitioned the data into the first 1499 days for training and the subsequent 500 days for testing i.e. they chose a 75/25 split, arguing that it would give a large enough testing set such that the covariance matrix would not be rank deficient.

For every solution on the efficient frontier, they recorded the CPU time, total number of iterations, and CPU time per iteration (TPI). Given that CPU time can vary on each run for reasons external to the numerical testing, they computed the efficient frontier 20 times and recorded the mean for the CPU times.

In addition, each solution on the efficient frontier also gives an expected risk, expected return and actual return.



# LSTM and LP reduction results

| Metric | No Reduction | LSTM Reduction | LP Reduction |
|---|---|---|---|
| Matrix Size | 374×374 | 128×128 | 82×82 |
| Total Iterations | 168 | 71 | 95 |
| Avg CPU Time | 0.0183s | 0.0162 s | 0.0165 s |
| Avg TPI (CPU Time/it) | 0.0688s | 0.1439 s | 0.1097 s |
| Assets Selected | 65 | 43 | 30 |
| Expected Return | 0.00161 | 0.00145 | 0.00128 |
| Actual Return | 0.01490 | 0.00104 | 0.00379 |
| Expected Risk | 0.00029 | 0.00017 | 0.00017 |

We see that the reductions due to both LSTM and LP now require far less iterations for convergence. Although, on average the CPU time per iteration was smaller for the unreduced model, when considering overall time we see a significant reduction.

The dimension reduction improves the risk associated, but reduces the return in both expectation and actuality.

Both reduction and no reduction showed that actual and expected return differ, which is not uncommon since the Markowitz model has been documented to emphasize estimation error, especially in larger portfolios.

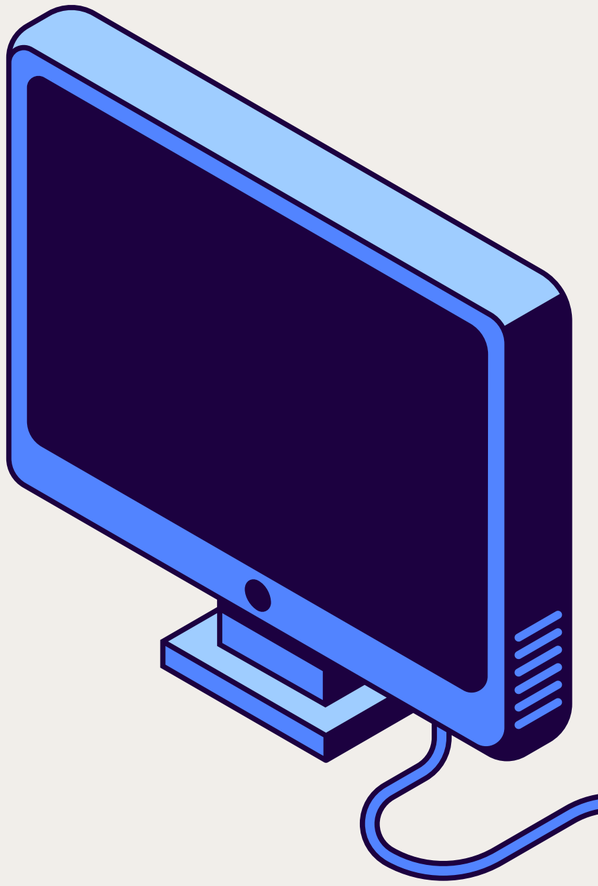
In the two reduction models, LP-based reduction provided better predictive accuracy (70.9%) compared to LSTM (61.2%).

# Sparsification Testing

For sparsification, the paper finds that for a low threshold ($0.4 \leq \tau \leq 0.7$), partial matrix completion as a means to re-establish positive semidefiniteness also requires a significant compromise on sparsity.

With $\tau > 0.8$, it resulted in the retention of only strong relationships in the risk measure while simultaneously promoting both sparsity and positive semidefiniteness.

For varying levels of sparsification, the Markowitz model was run.



# Sparsification results

| **Sparsity Level (# 0s induced /total # elements)** | **0% (Dense)** | **50% Sparse** | **90% Sparse** | **99% Sparse** |
|---|---|---|---|---|
| Avg Time per Iteration | 0.069 s | 0.033 s | 0.023 s | 0.019 s |
| Assets Selected | 65 | 150 | 286 | 356 |
| Expected Return | 0.0016 | 0.0018 | 0.0018 | 0.0018 |
| Actual Return | 0.0149 | 0.0182 | 0.0198 | 0.0208 |

Sparsification most notably decreased the run-time per iteration.

As the covariance matrix grows more sparse, the optimal solution invests in more assets.

With respect to the portfolio performance, the expected risk was similar to the dense portfolio and expected return increased as the sparsity increased. This might suggest that sparsification promoted a high diversification in portfolio in case where most stocks are strongly positively correlated.

Overall, sparsification yielded better optimizer results, while not compromising on either risk or return.

# Our Implementation

We have created synthetic Stock Data. ( Simulated 10 stocks behaving like real S&P 500 stocks over 3 years )

**We Compared:**

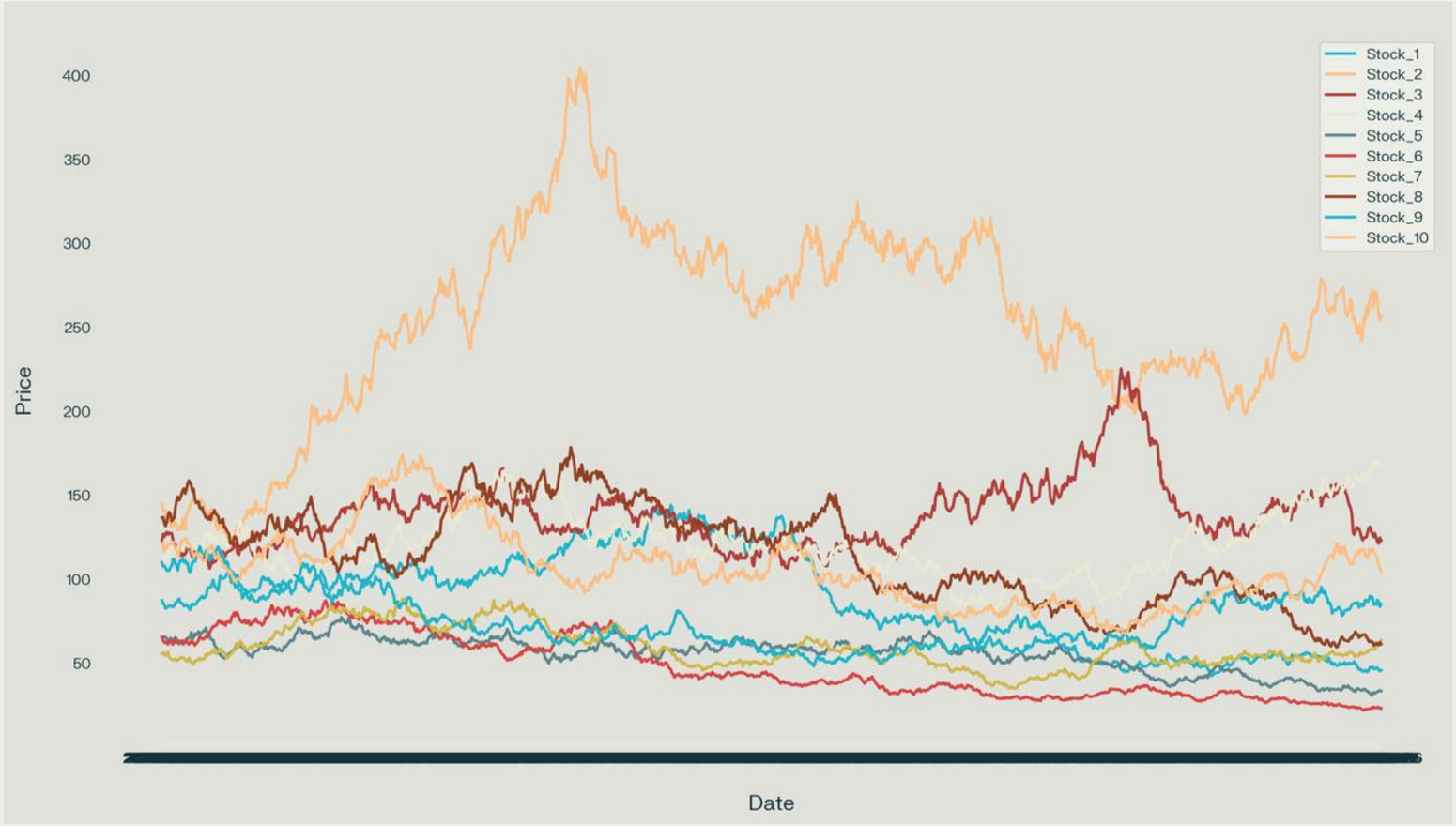
1. The original method (using all data).
2. The reduced methods (using only predicted important assets).
3. The sparse method (using a simplified covariance matrix).

**On the basis of:**

1. Speed: How fast the computer could solve the problem.
2. Portfolio quality: Did the new methods still find good portfolios (in terms of risk and return)?

**EDA of synthetic data we have taken**

- Dimension reduction (using neural networks or LP) made the problem much smaller and faster, with only a small loss in portfolio quality.
- Sparsification made each calculation step much faster, especially as the matrix got sparser.
- The LP method was better at predicting which assets would be in the best portfolios than the neural network.
- The new methods found portfolios with similar risk and return as the original, but much faster.



# Portfolio Allocation Comparison

When applied to the synthetic stock dataset, both models yield different optimal allocations:

| Stock | **Markowitz Model** | **LP Model** | **Difference** |
|---|---|---|---|
| **Stock_1** | 0.103 | 0.102969 | -0.000031 |
| **Stock_2** | 0.117 | 0.104827 | -0.012173 |
| **Stock_3** | 0.117 | 0.105155 | -0.011845 |
| **Stock_4** | 0.110 | 0.099270 | -0.010730 |
| **Stock_5** | 0.091 | 0.092877 | +0.001877 |
| **Stock_6** | 0.081 | 0.092450 | +0.011450 |
| **Stock_7** | 0.108 | 0.113510 | +0.005510 |
| **Stock_8** | 0.088 | 0.094658 | +0.006658 |
| **Stock_9** | 0.098 | 0.102633 | +0.004633 |
| **Stock_10** | 0.087 | 0.091653 | +0.004653 |

# Future Improvements

- One can experiment with other, more complex sparsification techniques in accompaniment with techniques to restore positive semidefiniteness, if lost, such as:

  H. Liu, L. Wang, T. Zhao, *Sparse Covariance Matrix Estimation With Eigenvalue Constraints* (2014)
- Hyperparameter Tuning in the Neural Network can be explored.
- We can utilize alternate Neural Network Structures/Transformer models for portfolio optimisation.
- Techniques utilised here can be further tested on other financial assets (and mixtures of assets) such as cryptocurrencies, stock derivatives, etc. to check for robustness across fields.

# Conclusion

In real-world investing, being able to quickly update and optimize portfolios is important, especially as the number of possible investments grows. These methods lets us handle much bigger problems in less time, without sacrificing much in terms of portfolio quality.

- We don’t need to consider every possible asset: Most won’t end up in the best portfolios.
- We can ignore weak relationships: This makes the math much faster.
- Machine learning and simpler math models can help: They can predict which assets matter most.
- The result: Faster, more efficient portfolio optimization for large numbers of assets, with little loss in quality.

# Thank You

Thank You For Your Attention