University of Colorado Boulder

APPM 3310: Matrix Methods and Applications

Final Project Report

Index-tracking Portfolio Optimization Using Linear Algebra

Authors
Hyden Polikoff
Brian Mayers
Santiago Velasco

November 23, 2021

Abstract

This paper examines the algebra used to construct a financial portfolio and the simplification of that process achieved by using aspects of linear algebra like matrices and their inverses. The objective of the report is to invest in stocks, a basic financial security, and construct a portfolio of various stocks, representing it with linear algebra. Since all elements of a portfolio can be expressed in matrix notation, it will easily facilitate computations that will determine the optimal weights needed in order to reach an efficient portfolio that minimizes risk or given an expected level of return, minimize the overall risk. Whereas the model can reach good results, it is worth to note that there are some assumptions employed by the model that are not reflected in the real financial markets due to its stochastic nature.

Attribution

In this project, Hyden and Brian wrote the Python code used for the optimization problem as well as contributed to the mathematical concepts behind portfolio using linear algebra components. Santiago also helped formulate the mathematical formulation and assumptions behind portfolio theory, as well as providing an explanation and application of portfolio optimization to the real financial markets, taking into account the differences between real scenarios and fictitious mathematical scenarios such as the one presented in this paper. With all of the information, the members wrote this LaTeX document, contributing equally.

Introduction

Stocks, a certain type of equity, are a vital part of the global financial system. Simply put, a share of stock is the partial ownership of a company, and these shares are traded by the billions each day, amounting to almost 3 trillion trades in 2020. With an increasing democratization of finance and the rise of investing platforms available to the public, it is worthwhile examining stocks, the inherent risk that they carry, and how to minimize risk on an investment in various stocks.

In financial markets, risk and reward are a trade-off, so investors cannot expect a higher change in dollar value of the investment, also known as the return, in a security such as equities without bearing additional risk. In practice, market participants don't hold a single financial asset but rather a collection of them, named a portfolio. A portfolio, Π, will have a risk or variance σ^2, and a return, R. The risk of a portfolio, however, is not the sum of all the individual risk components, as some asset's gain may offset a loss in another asset, a concept that will be visualized using algebra later on.

When dealing with portfolios, investors want to get the best risk-return trade-off. This can be done by choosing a level of return and minimizing risk. Linear algebra can be used to create an optimization problem and solve for either of the aforementioned cases since a portfolio can be expressed as its various components: the return of each security, the expected return of the portfolio, component weights as a percentage of the portfolio, the variance of each security, and the covariance between securities. It will later be seen that all the vector components can be simplified using linear algebra, leading to easier computations on portfolios with a significant amount of securities.

Key Assumptions

The key assumptions of the model are the following:

- The constant expected return (CER) model. This model assumes that the returns of an asset are independent and identically distributed random variables with a normal distribution $N(\mu_i, \sigma_i^2)$. (See Appendix for further information)
  This normality of a stock's returns is not held in practice.
- The model also assumes that the variances of each asset's return as well as the covariances between assets are constant over time.

Mathematical Formulation

Portfolio Returns

Let there be a set of assets A, B, C. The return of each asset will be $R_i (i = A, B, C)$, a random variable under CER conditions. Each weight of the portfolio is represented as $x_i (i = A, B, C)$ and must fulfill the weight restriction $x_A + x_B + x_C = 1$. Thus, the return of a portfolio will be expressed as such:

$$R_{\Pi,x} = x_A R_A + x_B R_B + x_C R_C$$

The expected value of the random variable R will be expressed as:

$$E[R_{\Pi,x}] = \mu_{\Pi,x} = x_A \mu_A + x_B \mu_B + x_C \mu_C$$

Variance and Covariance

The variance of an asset is defined by σ_i^2 for any given asset i. In this model, the variance of an asset is measured by calculating the variance of expected returns for that given asset.

We then define the covariance between 2 securities as $cov[R_i, R_j] = \sigma_{ij}$. This co-variance measures the correlation between two assets' expected return.

The variance of the entire portfolio is denoted as:

$$\sigma^2_{\Pi,x}$$

Thus, since returns are normally distributed and the variance is σ^2, the variance of the portfolio return will be:

$$\sigma^2_{\Pi,x} = x_A^2\sigma_A^2 + x_B^2\sigma_B^2 + x_C^2\sigma_C^2 + 2x_Ax_B\sigma_{AB} + 2x_Ax_C\sigma_{AC} + 2x_Bx_C\sigma_{BC}$$

As we can see, the algebra will get significantly complicated as more assets are evaluated. Nonetheless, a mathematical tool that can be used to simplify this process are matrices, the essence of linear algebra, so we will express all portfolio components as such.

Matrix Representations

The returns and the weights will be:

$$\mathbf{R} = \begin{pmatrix} R_A \\ R_B \\ R_C \end{pmatrix} \qquad \mathbf{x} = \begin{pmatrix} x_A \\ x_B \\ x_C \end{pmatrix}$$

The expected return, μ will be:

$$E[\mathbf{R}] = E\left[\begin{pmatrix} R_A \\ R_B \\ R_C \end{pmatrix}\right] = \begin{pmatrix} E[R_A] \\ E[R_B] \\ E[R_C] \end{pmatrix} = \begin{pmatrix} \mu_A \\ \mu_B \\ \mu_C \end{pmatrix} = \mu$$

And finally, the variance matrix will be:

$$var(\mathbf{R}) = \begin{bmatrix} var(R_A) & cov(R_A, R_B) & cov(R_A, R_C) \\ cov(R_B, R_A) & var(R_B) & cov(R_B, R_C) \\ cov(R_C, R_A) & cov(R_C, R_B) & var(R_C) \end{bmatrix} = \mathbf{\Sigma}$$

Therefore, the formulas are much simpler now. The return of the portfolio, R_Π will now be

$$R_{\Pi,x} = \mathbf{x^T R} = \begin{pmatrix} x_A & x_B & x_C \end{pmatrix} \cdot \begin{pmatrix} R_A \\ R_B \\ R_C \end{pmatrix} = x_AR_A + x_BR_B + x_CR_C$$

Likewise, the expected return of the portfolio is

$$E(\mathbf{x^T R}) = \mathbf{x^T} E[\mathbf{R}] = \mathbf{x^T}\mu = \begin{pmatrix} x_A & x_B & x_C \end{pmatrix} \cdot \begin{pmatrix} R_A \\ R_B \\ R_C \end{pmatrix} = \mu_A + \mu_B + \mu_C = \mu_{\Pi,x}$$

Finally, the variance of the portfolio can be elegantly expressed as

$$\sigma_\Pi^2 = \mathbf{x^T \Sigma x} = \begin{pmatrix} x_A & x_B & x_C \end{pmatrix} \cdot \begin{bmatrix} \sigma_A^2 & \sigma_{AB} & \sigma_{AC} \\ \sigma_{AB} & \sigma_B^2 & \sigma_{BC} \\ \sigma_{AC} & \sigma_{BC} & \sigma_C^2 \end{bmatrix} \begin{pmatrix} x_A \\ x_B \\ x_C \end{pmatrix}$$

$$= x_A^2\sigma_A^2 + x_B^2\sigma_B^2 + x_C^2\sigma_C^2 + 2x_Ax_B\sigma_{AB} + 2x_Ax_C\sigma_{AC} + 2x_Bx_C\sigma_{BC}$$

Note: The final portfolio variance formula represents a portfolio quite simply. A portfolio carries the variance, or the volatility of the assets it has: A, B, C. However, the covariance terms may reduce or increase risk. Say that asset A and asset C are inversely related, meaning that a decrease of the price of A will be met with an increase in the price of C. Therefore, the σ_{AC} will be negative, making the overall $2x_Ax_C\sigma_{AC}$ term negative. It is clear to see that the subtraction of the term will result in a lower variance of the portfolio. This makes sense, as the portfolio becomes less sensitive to changes by having inversely related assets, meaning the portfolio is less risky.

Given these matrices, we can now use Lagrangian multipliers to optimize the portfolio using a weight restriction.

Lagrange Multipliers

Lagrange multipliers are a way to solve constrained optimization problems. Say there exists a relationship between two or more inputs and some output. That is, $F(input_1, input_2, ...) = output$ for some function F. In this case, the inputs are the weights of two or more stocks an investor would like to invest in and the output will be the measure of the total variance (see definition above) which, as explained, will understand to be the risk. Simply solving this function for the minimum variance would yield some weighting that makes no sense since the model is operating under the assumption that the total capital of the portfolio is used. Thus, there must be a constraint to the inputs: the weights of the investments must sum to one $(x_A + x_B + x_C)$. This is the Lagrangian. Taking the example of 3 assets A, B, C and solving for the global minimum variance yields a weight vector of:

$$\mathbf{x} = \begin{pmatrix} 0.73 \\ 0.54 \\ 0.23 \end{pmatrix}$$

This is senseless, as the sum of the proportions will exceed 1 and one cannot invest more than 100% of a portfolio. Solving this function subject to the Lagrangian weight constraint solutions will look something more sensible like:

$$\mathbf{x} = \begin{pmatrix} 0.64 \\ 0.23 \\ 0.13 \end{pmatrix}$$

Let's take a look at how we might set up this problem. Say we have our weight vector

$$\mathbf{x} = (x_A + x_B + x_C)^T$$

where each variable represents the percent invested in said stock. We discussed earlier how $\mathbf{x}^T \Sigma \mathbf{x}$ represents the risk of our portfolio. Therefore our simple function that models risk given any arbitrary weights takes the form

$$F(\mathbf{x}) = \mathbf{x}^T \Sigma \mathbf{x}$$

Now let us constrain this such that the weights sum to one.

$$\mathbf{x} = (x_A + x_B + x_C)^T \mathbf{1} = 1$$

The Lagrangian—that is, the constrained function for variance of our portfolio—follows the form

$$L(\mathbf{x}, \lambda) = \mathbf{x}^T \Sigma \mathbf{x} + \lambda(\mathbf{x}\mathbf{1} - 1)$$

In order to solve this problem, we set up the first order conditions:

$$\begin{aligned} \mathbf{0} &= \frac{\partial L(\mathbf{x}, \lambda)}{\partial \mathbf{x}} = 2\Sigma \cdot \mathbf{x} + \lambda \cdot \mathbf{1} \\ \mathbf{0} &= \frac{\partial L(\mathbf{x}, \lambda)}{\partial \lambda} = \mathbf{x}^T \mathbf{1} - 1 \end{aligned} \tag{1}$$

In order to greatly simplify these calculations, we can represent these equations as a system using matrix notation, where Ax = b

$$\underbrace{\begin{pmatrix} 2\Sigma & \mathbf{1} \\ \mathbf{1}^T & 0 \end{pmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{pmatrix} \mathbf{x} \\ \lambda \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix}}_{\mathbf{b}}$$

As it can be seen, all that is left is to simply calculate an inverse in order to solve for the respective weightings of the stocks to form the minimum variance portfolio as $\mathrm{x} = \mathrm{A}^{-1} b$

Twice Constrained Lagrangian

Above, the task was to simply solve for the minimum risk. But what if an investor wanted to get a certain amount of return on their portfolio while still minimizing risk? Often when risk is absolutely minimized, it is done so at the cost of a reasonable expected return, meaning the investor will get no return as they are not experiencing risk. A much more sensible scenario is targeting a 10% return with securities $A, B,$ and C, minimize the risk. It turns out that this problem can easily be solved by adding a second constraint—or second Lagrangian.

With a second Lagrangian whose constraint requires that the portfolio return is constrained to $\mu_{p,x} = x^T \mu = \mu_{target}$, where μ_t is the target return, the following problem is solved:

$$\begin{aligned} &\text{minimize } \sigma^2_{\Pi,x} = \mathbf{x}^T \Sigma \mathbf{x} \\ &\quad \textbf{such that} \\ &\qquad \mu_{\Pi,x} = \mu_{target} \text{ (set the expected return to equal the target return)} \\ &\qquad \mathbf{x}^T \mathbf{1} = 1 \text{ (weights sum to one, our previous constraint)} \end{aligned} \tag{2}$$

The first order conditions that define the system are

$$\underbrace{\begin{pmatrix} 2\Sigma & \mu_{\Pi,x} & \mathbf{1} \\ \mu^T_{\Pi,x} & 0 & 0 \\ \mathbf{1}^T & 0 & 0 \end{pmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{pmatrix} \mathbf{x} \\ \lambda_1 \\ \lambda_2 \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} \mathbf{0} \\ \mu_{target} \\ 1 \end{pmatrix}}_{\mathbf{b}}$$

This can once again be solved easily with the use of linear algebra techniques.

Examples and Numerical Results

As previously mentioned, two scenarios will be explored:

1. Minimizing the overall risk of the portfolio
2. Minimizing the overall risk of the portfolio given a targeted level of return

Minimizing Risk

The first example deals with solving a portfolio that holds 3 assets picked arbitrarily, each of them stocks. Say a market participant decided to make an investment in Apple, General Electric, and Procter & Gamble, whose tickers in the market are AAPL, GE, PG, respectively. Below is a graph of their prices from 2015 - 2020, the time period for the data used to calculate variance, covariance and expected return.

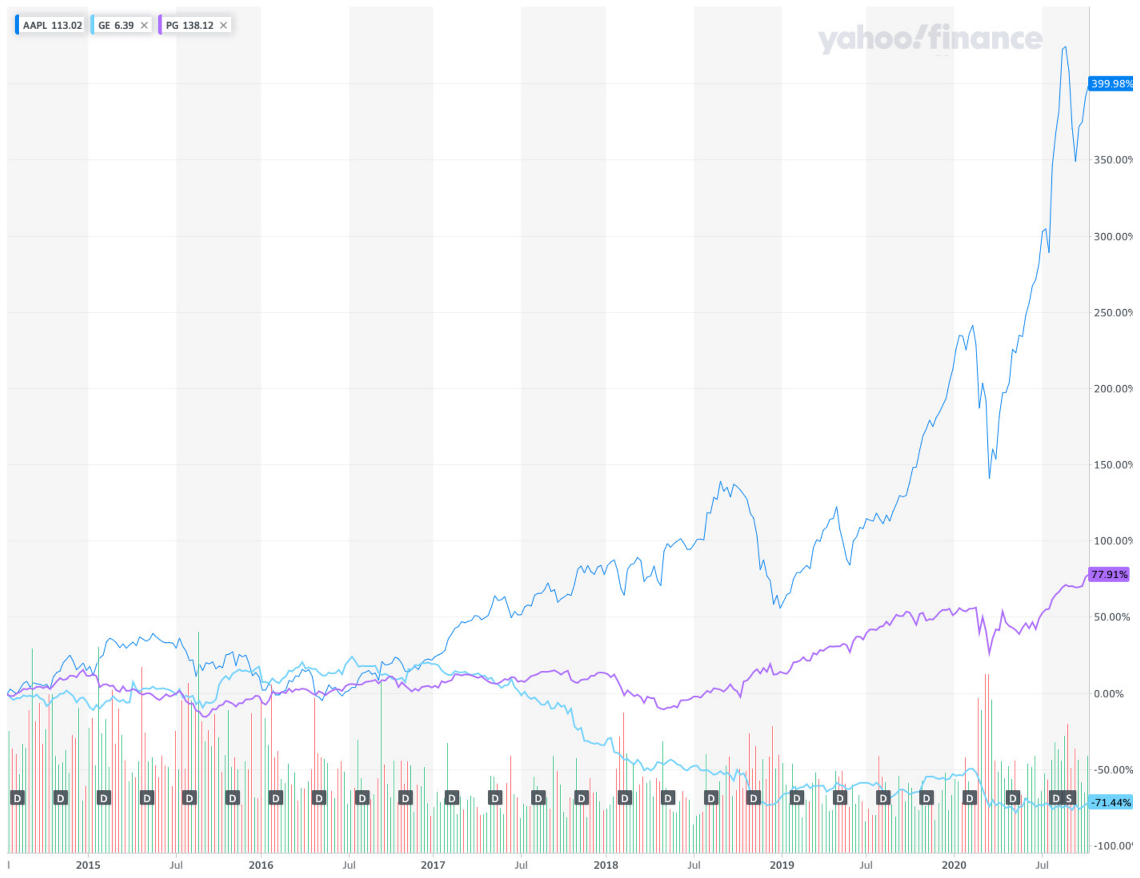


Figure 1: Historical Prices

Analysis of 3 Assets

Apple generated large returns while General Electrical lost a significant portion of its value. This inverse relation is important when crafting a portfolio. The third asset - Procter and Gamble - generated a more reasonable rate of return.

In order to compute our minimum variance portfolio, various Python functions were implemented to construct and then subsequently solve the linear systems Ax=b formulated above.[1] Computing the minimum variance portfolio, an optimal weighting of 9% for Apple, 77% for General Electric and 14% for Proctor and Gamble will be the result. That is,

$$\mathbf{x} = (0.09,\ 0.77,\ 0.12)^T$$

This yields an expected annual return of 3.37% for the portfolio assuming the CER model.

Having this in mind, one can generalize it to a larger basket of securities. Say an investor

[1]The entire script can be found in the appendix.

has an interest in 10 or 20 different stocks and wants to get an optimal weighting; this model can be easily adapted to solve for the proportions. In fact, the Python implementation only requires modifying the tickers of which stocks one may wish to invest into.

It is important to note that once one reaches a certain level of diversification, the risk plateaus. This means that investing in 100 versus 200 different stocks will do little to change the risk or return assuming a somewhat equal weighting, as the marginal decrease in risk may not be worth the additional purchase of securities. The S&P 500, which is an index comprised of 500 stocks, would benefit little from this sort of optimization give the plateauing effect past a certain level of diversification. In the three asset example, GE and AAPL contribute to diversification in the portfolio, as the investor will be less exposed to losses of one stock by an offset of the other stock's gain.

Note: Diversification, the process of allocating capital in a way that reduces the exposure to any one particular asset or risk, can be achieved through investing in a variety of assets, which reduces the overall volatility of the portfolio. Thus, with an increasing number of securities in a portfolio, a same level of return can be achieved with less exposure to risk.

Returning to the example of 3 assets of Apple, General Electric, and Procter and Gamble, an efficiency frontier can be obtained. The below figure plots Risk (Deviation of returns) *vs.* Expected Return. This is a graphical representation of the classical risk-return trade-off an investor must make. The red dot on the left side of the graph represents the portfolio with the lowest variance/ risk:

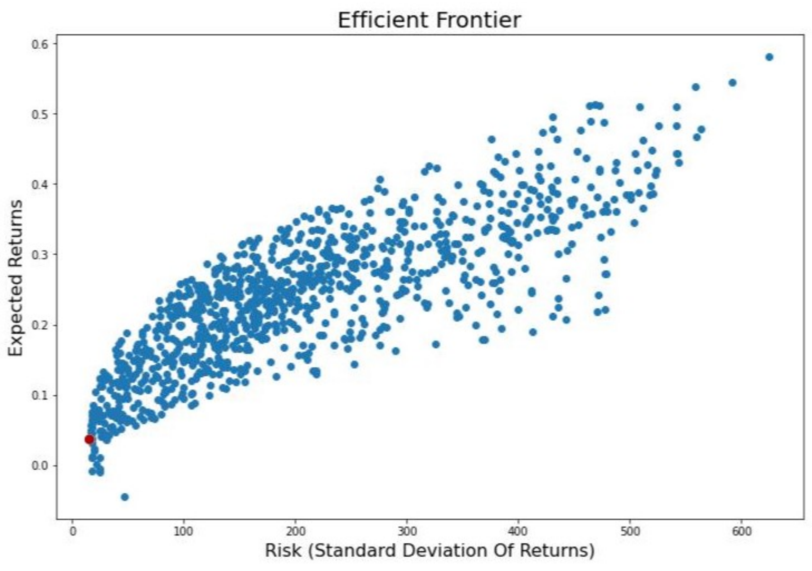


Figure 2: Efficient Frontier

Minimizing Risk with a Target Return

Now suppose the investor wants to optimize a portfolio around a target risk, or target return. In order to optimize a portfolio around a target return, an additional constraint is needed in model. As previously mentioned, the model is subject to the sum of the weights of assets being one: $x_B + x_B + x_C = 1$. By construction, this model has the potential to yield higher returns than the previous model while taking on some additional risk. For this example, seven securities will be evaluated from 2015 to 2020.

Ticker:	STX	HON	NVDA	BRK-B	CAT	TSLA
μ_i	.213	.306	1.57	.1478	.321	1.45
σ_i^2	104.19	976.12	18033.32	939.52	1035.81	12654.55

Table 1: Expected return and variances for 5 stocks

The portfolio will be constrained to a $30\%(\mu_{target} = -0.3)$ expected return. The process used to calculate the weights of this portfolio is based on translating much of the mathematical formulation into a Python program. Below is some pseudo-code that captures the most important parts of the program:

```
#Function to calculate the optimal weights
import numpy as np
def portfolio_optimization(list_stock_prices, desired_return):

    sigma = build_sigma(list_stock_prices) #builds variance/covariance matrix
    mew = expected_returns(list_stock_prices) #expected return vector

    #matrix formed given the FOCs of the Lagrangian
    A = FOC_matrix(constraints, sigma, mew)

    #solve Ax=b to get weight vector
    A_inv = np.linalg.inv(A)
    soln = np.matmul(A_inv, b)

    return soln
```

For parameters, the function above takes in a list of previous stock prices for multiple stocks as well as the desired return that the investor seeks from the portfolio. The solution is set up by calling functions that construct the Σ matrix along with the μ vector. [2] After, the first order conditions of the model are translated into matrix form. Finally, the system is solved by using the *numpy* library on Python. Here are the resulting weights of the portfolio:

[2]These functions can be seen in the appendix.

Ticker:	STX	HON	NVDA	BRK-B	CAT	TSLA
x_i	.875	-.416	-.14	.815	-.209	.075

Table 2: Optimal weights of the stocks in the portfolio

It is important to note that although the sum of all the weights is equal to one, not all the given weights fall between 0 and 1. The intuition of the negative weight is that an investor should short the stock. Essentially, shorting a stock, labeled S, is taking a position in which the investor believes the stock's price will decline over a certain period. The investor does this expecting to make a profit by first selling a share, receiving P_s in order to later buy the share back at a lower price. When investors start targeting higher returns such as 30%, the use of more sophisticated financial vehicles such as derivatives becomes more prevalent.

More advanced financial instruments, such as derivatives, are out of the scope of this paper. Nonetheless, the model successfully calculates the desired weights given our target return, giving a good estimate of the real-world optimal values.

Discussion and Conclusion

This optimization model of a portfolio provides a mathematical framework to create an optimal portfolio. As shown by the numerical examples, the weights of a portfolio would create a scenario where variance is minimized given a return. Nonetheless, it is imperative to take the irrationality of investors in the financial markets into account; the returns may not be as attractive in real scenarios, not to mention the plethora of unfortunate global and domestic events that may negatively affect the overall market, as was the case with the COVID-19 pandemic.

However, properly managing financial risk, this portfolio optimization model can be a helpful tool in the world of finance to a certain extent. Since the 1990s, a financial vehicle known as an Exchange Traded Fund (ETF) has been gaining popularity in the markets. ETFs provide access to passive investing, and there are various types of ETFs that can cater to any type of investor, each which can specialize in a stock index like the Dow Jones Industrial Average or a more niche product.

This framework of portfolio optimization can lead to the creation of a fund that tracks a given set of stocks, providing the efficient portfolio and investing using the weights of each stock in the portfolio. This could act as a way for a fund to attract investors who want to invest in a low-risk portfolio but do not have the capital to achieve the exact correct weights. Thus, the investor will invest their money in a larger optimized portfolio, receiving that percentage gain on their invested amount (when excluding management fees paid to fund managers).

The optimization model used in this paper takes big assumptions and ignores the true "randomness" of financial markets; however, it still provides a good reference of optimal weight distribution in a portfolio comprised of various securities.

References

Ghysels, Eric, et al. “There Is a Risk-Return Trade-off after All.” Journal of Financial Economics, vol. 76, no. 3, June 2005, pp. 509–548., doi:https://doi.org/10.1016/j.jfineco.2004.03.008.

“Historical Market Volume Data.” Cboe Global Markets, 2020, www.cboe.com/us/equities/market_statistics/historical_market_volume/.

Markowitz, H. (1959). Portfolio Selection: Efficient Diversification of Investments, Wiley, New York

Segal, Troy. Diversification. Investopedia, 21 Apr. 2021, www.investopedia.com/terms/d/diversification.asp.

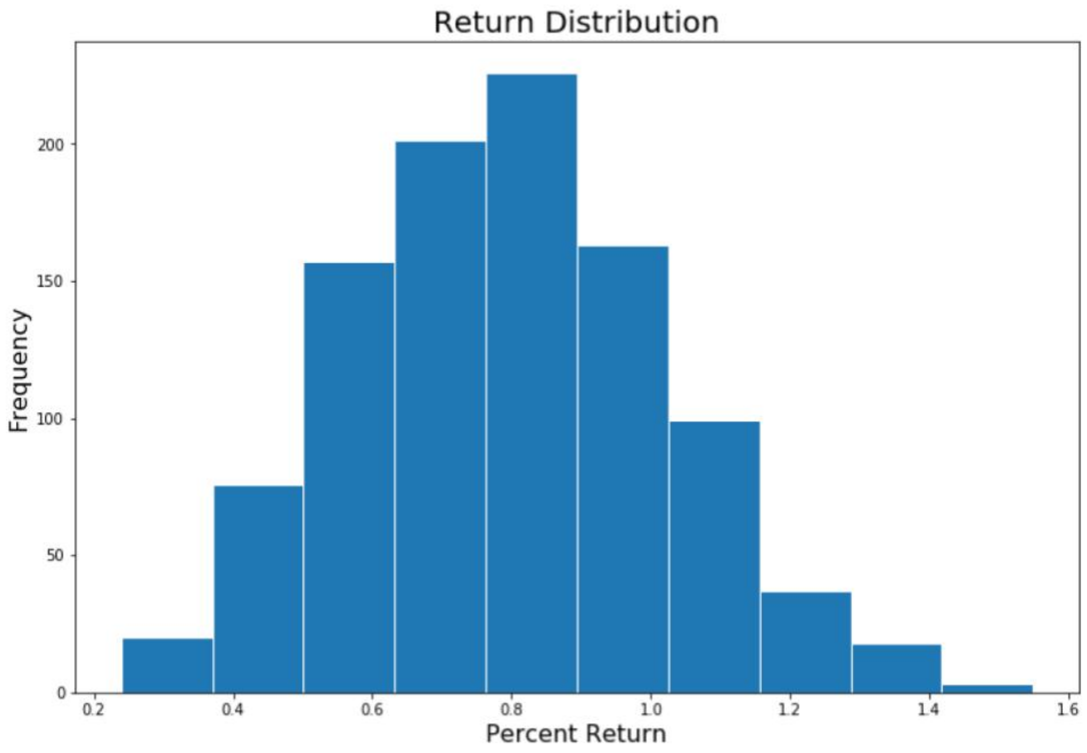
Simpson, Stephen D. “A Brief History Of Exchange-Traded Funds.” Investopedia, 5 Feb. 2021, www.investopedia.com/articles/exchangetradedfunds/12/brief-history-exchange-traded-funds.asp.

Zivot, Eric. University of Washington, 2000, The Constant Expected Return Model, http://research.economics.unsw.edu.au/jmorley/econ487/ch3.pdf

Appendix

Return Distribution

Normal distribution of portfolio returns:
The graph below shows the normal distribution for the returns of a given stock. Recall that our return for a given portfolio is defined as $R_{\Pi,x} \sim N(\mu_{\Pi,x}, \sigma_{\Pi,x})$. This graph displays this normal distribution generated from 1000 random samples of weight vectors for the given portfolio.


Return Distribution
Frequency
Percent Return
0
50
100
150
200
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

Code

Matrix_project

April 28, 2021

1 Functions

[ ]:
```python
# import needed packages
import pandas as pd
import yfinance as yf
import datetime
import numpy as np
import matplotlib.pyplot as plt
import statistics as st
```

[ ]:
```python
# function to fetch adj close data
# takes in list of symbols
# returns list of data frames in order of symbols
# Date | Adj Close
def get_prices(symbol_list):
  # create empty dataframe and list to store data frames
  df = pd.DataFrame()
  data = []

  # iterate over each symbol
  for i in symbol_list:

    # print the symbol which is being downloaded
    print( str(symbol_list.index(i)) + str(' : ') + i, sep=',', end=',',␣
↪flush=True)

    try:
      # download the stock price into a list
      stock = []
      stock = yf.download(i, start=start, end=end, progress=False)

      # append the individual stock prices
      if len(stock) == 0:
        None
      else:
        stock['Name'] = i
        # put data in a data frame and drop unnecessary info
        df = stock
        df.drop(columns=['Open', 'Low', 'High', 'Close'], inplace=True)

        # add the dataframe to the end of the list
        data.append(df)
    except Exception:
      None

  # return the list of data
  return data
```

[ ]:
```
# takes in a list of  data frames of date | adj close
# returns a list of values for the variances
def stock_variance(list_df_prices):
  # calculate the variances for the stocks
  stock_variances = []
  for i in range(0, len(list_df_prices)):

    df = pd.DataFrame(list_df_prices[i]['Adj Close'])

    # Calculate the variance and append to list
    var = np.var(df['Adj Close'])
    stock_variances.append(var)

  return stock_variances
```

[ ]:
```
# takes in a list of  data frames of date | adj close
# returns the covarance between i and j
def stock_covariance(list_df_prices, i, j):

  l = len(list_df_prices)

  # make sure i and j are within range and not the same
  if i < 0 or i >= l or j < 0 or j >= l:
    print("Indices out of range")
    return -1
  if i == j:
    print("i == j")
    return -1

  # setup a numpy array for covariance calculations
  arr1 = np.array(pd.DataFrame(list_df_prices[i]['Adj Close'])).T
  arr2 = np.array(pd.DataFrame(list_df_prices[j]['Adj Close'])).T

  cov = np.cov(arr1, arr2)[0][1]

  return cov
```

[ ]:
```python
# takes in a list of  data frames of date | adj close
# returns a list of values for the expected returns as decimals, not percents
def expected_return(list_df_prices):
  exp_returns = []
  for i in range(0, len(data)):

    # calculate the expected the average daily returns
    df = pd.DataFrame(data[i]['Adj Close']).pct_change()
    m = df['Adj Close'].mean()

    # convert to annual expected returns
    # annual = (dialy_return + 1)^365 - 1
    annual = ((m + 1) ** 365) - 1
    exp_returns.append(annual)

  # return the list
  return exp_returns
```

[ ]:
```python
# function to build Sigma matrix
# returns a 2D numpy array
def build_sigma(list_df_prices):
  # store size of data and create blank array
  l = len(list_df_prices)
  sigma = np.zeros((l, l))

  s_vars = stock_variance(list_df_prices)
  # iterate through
  for i in range(0, l):
    for j in range(0, l):

      # if on the diagonal, input the variance
      if j == i:
        sigma[i][j] = s_vars[i]

      # if not on the diagonal, insert the covariance
      else:
        sigma[i][j] = stock_covariance(list_df_prices, i, j)
  return sigma
```

[ ]:
```python
# function to return optimal portfolio weights for absolute minimum variance
# returns list of weightings as decimals length of number of assets
def solve_min_variance(list_df_prices):

  # preallocate the matrix A
  l = len(list_df_prices)
  size = l + 1
  A = np.ones((size, size))
  # build the Sigma matrix
  sigma = build_sigma(list_df_prices)

  # Build the A matrix
  for i in range(0, l):
    for j in range(0, l):
      A[i][j] = 2 * sigma[i][j]
  A[l][l] = 0

  # build b matrix
  b = np.zeros((size, 1))
  b[l][0] = 1

  # compute A^-1*b for solutions
  A_inv = np.linalg.inv(A)
  soln = np.matmul(A_inv, b)

  # Extract solutions
  ans = soln[:l]
  print(ans)

  # perform check on weightings add to 1
  sum = 0
  for i in range(0, l):
    sum += float(soln[i])
  c = round(sum, 5)
  assert(c == 1)

  return ans
```

[ ]:
```
#function finds weights of stocks based subject to a constraint of expected␣
 ↪return
def solve_mean_variance(list_df_prices, expected_return_constraint):
    num_stocks = len(list_df_prices)
    matrix_size = num_stocks+2
    A = np.zeros((matrix_size, matrix_size))

    #compute sigma matrix
    sigma = build_sigma(list_df_prices)
    #find expected return of the stocks
    mew = expected_return(list_df_prices)

    #construct matrix of FOCs
    for i in range(0, num_stocks):
        for j in range(0, num_stocks):
          A[i][j] = 2 * sigma[i][j]
    for i in range(num_stocks,num_stocks+1):
        for j in range(0,num_stocks):
            A[i][j] = mew[j]

    for i in range(num_stocks,num_stocks+1):
        for j in range(0,num_stocks):
            A[j][i] = mew[j]

    for i in range(num_stocks+1,num_stocks+2):
        for j in range(0,num_stocks):
           A[i][j] = 1

    for i in range(num_stocks+1,num_stocks+2):
        for j in range(0,num_stocks):
           A[j][i] = 1

    #construct b vector
    b = np.zeros((matrix_size, 1))
    for i in range(0, matrix_size):
        if i == matrix_size-num_stocks+1:
            b[i] = expected_return_constraint
    b[matrix_size-1][0] = 1
    #print(b)
    #solve

    A_inv = np.linalg.lstsq(A, b, rcond=None)[0]
    soln = A_inv #np.matmul(A_inv, b)
    #A_inv = np.linalg.inv(A)
    #soln = np.matmul(A_inv, b)

    ans = soln[:num_stocks]
    print(ans)


    return ans
```

1.1 Driver Cell

[ ]:
```
# min variance

# Specify Start and end time for data
start = datetime.datetime(2015,1,1)
end = datetime.datetime(2020,12,31)

# Specify tickers for desired portfolio
Symbols = ['AAPL', 'GE', 'PG']
```

```
# -----------------
# get the data and build sigma
data = get_prices(Symbols)

# print(data)
A = solve_min_variance(data)

# print the weightings
for i in range(0, len(Symbols)):
  r = float(A[i] * 100)
  p = round(r, 0)
  print("Weight for {}   is {}%".format(Symbols[i], p))

print()
ex_returns = expected_return(data)
for i in range(0, len(Symbols)):
  print("Expected return of {} is {}%".format(Symbols[i],␣
 ↪round(ex_returns[i]*100), 4))

print()
prtflo_return = round(float(np.matmul(A.T, ex_returns)), 4);
print("portfolio expected return = {}%".format(prtflo_return*100))
```

[ ]:
```
# Test Mean-Var
Symbols = ['AAPL', 'GE', 'PG']


data = get_prices(Symbols) #get the prices
print()
res = solve_mean_variance(data, .3) #getting linalg error, not happening␣
 ↪locally used LS instead


prtflo_return = np.matmul(res.T, ex_returns)
print(prtflo_return)
```

[ ]:
```
#Construct Efficent Fronteir Graph
def random_weights(data):
    val = np.random.random(len(data))
    val  = val / val.sum()
    return val



def graph_EfficentFronteir(price_df, min_SD):
    returns = expected_return(price_df)
    sigma = build_sigma(price_df)

    gain = []
    std = []
    weights = []
    for i in range(1000): #calculate values at random
        rand_weight = random_weights(price_df)
        weights.append(rand_weight)
        gain.append(np.dot(returns,rand_weight))
        std.append(np.dot(np.dot(rand_weight,sigma),rand_weight))

    plt.figure(figsize=(12,8))
    plt.scatter(std,gain) #plot returns vs standard deviation
    plt.title("Efficient Frontier",fontsize=20)
    plt.xlabel("Risk (Standard Deviation Of Returns)",fontsize=16)
    plt.ylabel("Expected Returns",fontsize=16)


Symbols = ['GE', 'PG','AAPL']


data = get_prices(Symbols) #get the prices
graph_EfficentFronteir(data,.2)
```

[ ]:
```
################################################################################
#                    CODE TO CONVERT COLAB NOTEBOOK TO PDF                     ␣
 ↪#
#This code converts an .ipynb file to a Tex based PDF and saves it in the␣
 ↪Colab#
#Notebook folder with the same filename.pdf                                    ␣
 ↪#
################################################################################

filename = 'Matrix_project.ipynb' # Ex. 'Coding_Packet.ipynb'

################################################################################
# Code
################################################################################
# Imports
from google.colab import drive; from datetime import datetime; import sys;␣
 ↪import random; import time
# To make Watermark distinctive
class color:
  BLUE = '\033[94m';GREEN = '\033[92m';BOLD = '\033[1m';UNDERLINE = '\033[4m';
 ↪END = '\033[0m';FAIL = '\033[91m';WARNING = '\033[93m'
now = datetime.now()
print(color.UNDERLINE,'Unique Watermark',color.END,'\U0001F512\n',color.END)
print(color.BOLD,color.BLUE,now.strftime("%d/%m/%Y %H:%M:%S")," ",str(random.
 ↪randrange(1000000, 9999999, 1)),color.END,'\n')
# Mount Drive First to give time for mounting to process
```

```
drive.mount('/content/gdrive', force_remount = True)
t1 = time.time()
# Install some dependences into the RUNTIME (is not local, needs to reinstall␣
 ↪every Factory runtime)
!pip install IPython >> outputsuppressed.txt
!pip install Latex >> outputsuppressed.txt
!pip install pandoc >> outputsuppressed.txt
!pip install nbconvert >> outputsuppressed.txt
!pip install jupyter >> outputsuppressed.txt
!apt-get install texlive-xetex texlive-fonts-recommended␣
 ↪texlive-generic-recommended >> outputsuppressed.txt
# Searches the Google drive directory for the filename and gives back it's
# location (This accounts for Wildcards and Spaces in the directory names).
# Uses jupyter and nbconvert to convert to a Tex file, then into a pdf
print('\nFinding file. This may take a minute or two depending on the size of␣
 ↪your drive...')
!IFS=$'\n' #Sets the reader to only break at newlines instead of spaces,␣
 ↪tabs,and newlines
loc = !find '/content/gdrive' -name "{filename}"
try:
  fileloc = loc[0]
except IndexError as error:
  print(color.BOLD,color.FAIL, "\nCould not find file in your Drive!\n" ,color.
 ↪END,color.WARNING,"- Make sure you input the correct filename\n","  - Make␣
 ↪sure the file is saved in the google drive you mounted\n\n",color.END)
  Er = str('Error: {}. {}, line: {}'.format(sys.exc_info()[0],sys.
 ↪exc_info()[1],sys.exc_info()[2].tb_lineno))
  f = open("ErrorLog.txt","a+"); f.write(Er); f.close()
  sys.tracebacklimit=0
  sys.exit("Please Try Again")
except Exception as exception:
  print(color.BOLD,color.WARNING, "Exception Occured, Please Check Log",color.
 ↪END)
  Er = str('Error: {}. {}, line: {}'.format(sys.exc_info()[0],sys.
 ↪exc_info()[1],sys.exc_info()[2].tb_lineno))
  f = open("ErrorLog.txt","a+");f.write(Er);f.close()
  sys.tracebacklimit=0
  sys.exit("Please Try Again")
# Convert the file
print('File Location: ',str(fileloc),'\n')
%autosave 5
!sleep 15s
!jupyter nbconvert --to pdf "{fileloc}" --log-level ERROR
# The PDF will be in the same folder as the original file
print(color.GREEN,"Conversion Complete!\nGreat Job and Have a Wonderful Day!"
      ,color.END,"\U0001F30C")
%autosave 120
t4 = time.time(); m = str(int((t4-t1)/60)); s = str(int((t4-t1)% 60))
print("Total Time: ",m," m  ",s," s")
############ Version Control:
# Last Update: 2020-10-18, 1459 - V 1.4.1 - Thomas Horning
# - Error Handling
# - Autosave added before conversion so that watermark is gaurenteed
# - Added timers for debugging
# Created by Thomas Horning
```

```
 Unique Watermark
```

```
  28/04/2021 04:39:17   5631653
```

```
Mounted at /content/gdrive
```

[ ]:

```
```