Portfolio Optimisation Project

Group members:

Wen Wen - 31514081
Stephanie Wainaina - 32558937
Maximilian Luckie - 31483615
Yuheng Shang - 30490324

Table of Contents

Executive summary

Project objectives:

We have information on 1199 stocks for a 26-year period (July 9, 1993 to July 31, 2019). The task at hand is to choose the composition of the portfolio in a way that maximises return and minimises risk. We must forecast future trends in stock prices by ranking portfolios and considering three different types of portfolio weights to complete forecasts of investment profitability.

Project process:

Our goal is to find the best investment portfolio that maximises returns while minimising risk, so during the course of the project we mainly carried out the following steps.

- Perform preliminary data processing
- Risk-reward calculations for stocks
- Combining risk and return, choose a portfolio with high returns and low risk by permuting stocks
- Examining Portfolio Efficient Frontiers
- Calculate relative weights, absolute weights and dynamic weights
- Use LSTM models and dynamic weights to predict future trends and returns for this portfolio

Final result:

After processing the data, we decided to select stocks in the range of 2000-2019 for the initial screening. After retaining 248 stocks, we performed risk and return calculations and selected high return and low risk portfolios through a portfolio of 5 A's of stocks. Once the above steps were completed, we calculated the relative, absolute and dynamic weights. And using the above findings, a model using LSTM was built for subsequent forecasting. We performed two forecasts, the first for the accuracy of the prediction model, for the last 180 days of the dataset, and the forecast method was to forecast the last 180 days and then compare it with the original data. A second forecast is for the next 40 days, and the return obtained for the last 40 days is 1802.68, giving a monthly growth rate of 1%.

Introduction

➢ Background

The stock market is an ever-evolving enigma with massive profits and loses to be found by thousands of investors everyday. To be able to predict stock prices

accurately and consistently would see one become the richest person in the world. This is because of the extreme difficulty and unpredictability in the stock market.

When investing in the stock market, an individual will buy a certain number of shares of a company they wish to invest in, and when they pick multiple companies this becomes their portfolio of stocks. Having a portfolio of stocks rather than just one individual stock allows risk to be minimised. We have been tasked with finding a method to arrange a portfolio of stocks that will guarantee a positive return. Minimising risk is also an upside. The S&P500 is an index containing America's 500 largest public companies. The S&P500 accounts for around 80% of the market value of the United States's equity market. We will be arranging our portfolio from companies that have been in the S&P 500 over the last 26 years.

➢ Dataset

The dataset we were given contains 1200 rows and 9459 columns. Each column represents a public company and each row represents a single day. The entries contain the daily adjusted closing price of each stock. Since this is data from the S&P500, each row only contains 500 entries and the other 700 odd companies were not part of the S&P500 at that point in time. In this case, the corresponding entry was filled with a NaN value.

➢ Goals, Problems and Solutions

This project's goal is to choose the portfolio's composition in a way that maximises return and minimises risk. This can be done simply by using the sharpe ratio or by other more advanced methods. Such as, by ranking the portfolios and taking into account three different types of portfolio weights—relative, absolute, and dynamic weights—we can choose a portfolio of stocks based on the stocks in the S&P 500 index over the previous 16 years and check how the efficient frontiers look like. An LSTM model is included to look into the stock's potential future trend in order to complete the forecast of the investment's profitability.

➢ Contributions

- **Stephanie Wainaina**
  - ☐ Primarily in charge of:
    - Data Wrangling and EDA of the efficient frontier
    - Visualisations of the optimal portfolios
    - The efficient frontier
- **Maximilian Luckie**
  - ☐ Primarily in charge of:
    - EDA
    - Sharpe ratio
    - Analysis with ARIMA model
- **Yuheng Shang**
  - ☐ Primarily in charge of :

- Executive summary
- Introduction Background
- Image comparison
- Calculate the increase
- Calculate the final profit

- **Wen Wen**
  - ☐ Primarily in charge of:
    - Initial stock price screening and calculation
    - Ranking portfolios to select the highest return and lowest risk portfolios
    - Calculating relative weights, absolute weights and dynamic weights
    - Dividing the current data into training, validation and test sets for the LSTM model to predict the future returns of the portfolios

Data Quality

EDA

The dataframe that was provided displayed the closing prices of stocks that appeared in the S&P500 for any amount of time from 1993-2019. The data was given in daily intervals, with a date column that was very simple to convert into datetime format. The dataframe contained 1200 columns and about 9500 rows, which translates to meaning that over the 26 year period (9500 days), 1200 companies were at some point part of the S&P500.

The fact that this dataframe only contained values for companies in the S&P500 meant that even though there were 1200 columns, every single row only 500 values, and the rest were filled with NaN. This resulted in differing numbers of values in every column as shown by the count value in the below diagram, which describes our initial dataframe.

	Date	0111145D US Equity	0202445Q US Equity	0203524D US Equity	0226226D US Equity	0376152D US Equity	0440296D US Equity	0544749D US Equity	0574018D US Equity	0598884D US Equity
count	9.459000e+03	6954.000000	6222.000000	6954.000000	5856.000000	5124.000000	1098.000000	4758.000000	5490.000000	2196.000000
mean	2.006199e+07	27.322987	46.999521	15.022384	23.477752	17.827976	98.748765	24.584672	23.179905	35.747307
std	7.484374e+04	11.726762	20.481534	6.365943	18.761716	12.514460	41.710473	9.956974	17.120526	14.427557
min	1.993091e+07	10.129900	13.131700	3.651000	4.201000	3.250000	50.171300	11.052100	6.122500	17.400200
25%	2.000023e+07	19.460600	31.485100	9.846950	8.771025	8.625000	69.896900	18.989050	11.814300	24.456400
50%	2.006082e+07	24.131900	44.780100	13.846650	16.249250	12.362500	77.398900	22.667800	17.085100	28.849850
75%	2.013021e+07	34.034800	62.828125	20.308700	34.030700	24.922500	135.685700	25.792350	31.275300	48.180600
max	2.019073e+07	57.253600	106.950000	38.791800	78.298300	51.470000	173.500000	70.470000	62.484400	69.562500

8 rows × 1200 columns

To bring some consistency to the dataframe, any company that spent time out of the S&P 500 was removed. This was extremely simple as any column containing a NaN value was just pulled from the dataframe. This reduced the dataframe from 1200 columns to about 390 columns. This means that for 26 years 390 of the 500 spots in the S&P500 were taken up by the same companies, and the other 810 company's moved in and out of the remaining 110

spots. Removing these 810 companies gives many advantages to subsequent calculations and models.The evidence shows that when a company is removed from the S&P500, their share price performs poorly. At best, it will hold its value. The company that replaced it in the index is almost always performing materially better at the time. So since any company that moves out of the index is at best holding its share price value, it holds that these stocks are going to be bad choices for portfolios. They will also have a bad sharpe ratio due to their unlikeliness to have a positive return. Another reason this was advantageous is because the stocks that move in and out of the S&P500 are more likely to have higher volatility, making them a bad choice for a portfolio and decreasing their sharpe ratio. The final reason for this is because it reduced the computing power required to perform any subsequent calculations on the whole dataset, especially when the removed columns were unlikely to be of any great use.

Another change that was made to the dataframe was the initial stock price. The starting price of all the stocks was different, as each price reflected the cost of one share in the respective company. So for every value in each column, the value was divided by the row one value. This meant that every stock started at a value of one, and the value in the last row of the dataframe very easily displayed its performance over the 26 year period. So the final dataframe is shown below on the left with the described dataframe shown on the right.

	Date	AA	AAPL	ABMD	ABT
0	1993-09-07	1.000000	1.000000	1.000000	1.000000
1	1993-09-08	0.981164	1.019020	1.166667	0.980955
2	1993-09-09	0.988018	0.990428	1.166667	1.000000
3	1993-09-10	0.982881	1.000000	1.100000	1.000000
4	1993-09-11	0.982881	1.000000	1.100000	1.000000
...	...	...	...	...	...
9454	2019-07-27	1.697179	258.254600	72.709333	26.868551
9455	2019-07-28	1.697179	258.254600	72.709333	26.868551
9456	2019-07-29	1.696445	260.666335	73.581333	27.113506
9457	2019-07-30	1.690575	259.547489	74.461333	27.040020
9458	2019-07-31	1.690575	259.547489	74.461333	27.040020

9459 rows × 391 columns

	Unnamed: 0	AA	AAPL	ABMD	ABT
count	9459.000000	9459.000000	9459.000000	9459.000000	9459.000000
mean	4729.000000	2.769101	49.676865	11.269869	6.800398
std	2730.722432	1.302955	68.985606	21.450949	5.109668
min	0.000000	0.790696	0.507335	0.566667	0.980955
25%	2364.500000	1.646788	1.536487	1.950133	3.650540
50%	4729.000000	2.387497	11.619219	3.397333	5.072354
75%	7093.500000	3.954045	84.319368	6.246667	9.112098
max	9458.000000	6.758829	285.171681	119.933333	27.171683

8 rows × 391 columns

So the final values in each column of the dataframe reflect each company's performance relative to their share price in 1993. For example, the column titled AAPL, which is the company Apple, if you bought $1,000 of their stock in 1993, it would now be worth roughly $260,000.

Sharpe Ratio

The Sharpe ratio is a quantitative measurement of a stock's investability. It divides a stock's returns by its risk, giving a risk vs reward ratio that is a major indication of whether a stock will be a good choice for a portfolio. The numerator of the quotient finds the difference between the annual return of a stock and the risk-free rate. The denominator is the standard deviation of the company's stock price.

The risk-free rate of return acts as a benchmark, giving a theoretical return that your stock must outperform to achieve a positive return. This is because the risk-free rate of return can be achieved with an investment that contains almost no risk, whereas any investment in the stock market does not. Since the S&P500 was the index being focused on, the US treasury bond rate was used for the risk-free rate. Over the last 30 years this worked out to be about a 3% return per annum.

When calculating the standard deviation of a stock, the increments used must be considered. If daily stock prices are used, the residuals that are summed when calculating standard deviation become overwhelmingly large, and the sharpe ratio for these stocks becomes extremely small, meaning it is hard to find any stock to invest in. If annual stock prices are used, the converse happens. Even though this would be ideal for finding the annual return of each stock, the standard deviation would be far too small, making the sharpe ratio extremely big. The best increments to use for sharpe ratio calculations are monthly, and that is what was used on our data. This was achieved by taking the above dataframe, and only keeping every 30th row, one for each month. Next, the dataframe was sliced to obtain a 10 year period. Calculating sharpe ratios over the whole 26 year period is inefficient and what was happening in a company 25 years ago should not influence a decision to invest in the present day. Once the dataframe contained just 10 years of data, the sharpe ratios were calculated as follows.

```
returns = []
stds = []
for i in range(2,392):
  m_returns = []
  for j in range(119):
    m_returns.append((((df_mi.iloc[j+1][i] - df_mi.iloc[j][i])/df_mi.iloc[j][i])-0.002683)*100)
  returns.append(sum(m_returns)/len(m_returns)*12)
  stds.append(stdev(m_returns))
```

```
sharpe = [0,0]
for i in range(390):
  sharpe.append(returns[i]/stds[i]) # creates sharpe ratios for all stocks in your chosen period
```

m_returns is a list of monthly returns for a column. From each monthly return the risk free rate was subtracted, which is the annual risk free rate divided by 12. This worked out to about 0.26%. Then, the average monthly return was found, multiplied by 12 to give an average annual return and appended to the returns list. The standard deviation was calculated from the monthly returns list of each stock. From here, the sharpe ratios were calculated and this list became a new row in the dataframe. The dataframe would then be sorted by sharpe ratio to show which stocks gave the best returns.

From here, the following two years of data were examined. This allowed for a scoring metric to see how well stocks perform if you choose them just based on their sharpe ratio. The returns of the stocks over the next two years were then added to a dataframe with the best stocks sorted by sharpe ratio. The dataframe looked as shown (right).
This process was repeated for three different 10 year periods over the whole set of data. This allowed for

	Stock	Sharpe Ratio	Future Returns
0	ROST	3.287439	1.931053
1	CHD	3.256847	1.457355
2	AGN	2.967472	0.662984
3	NI	2.965457	1.130694
4	REGN	2.888019	0.648589
...	...	...	...
95	BA	1.636462	1.616799
96	LOW	1.635363	1.349038
97	WM	1.634751	1.599772
98	TMO	1.618831	1.670535
99	XLNX	1.597327	2.106723

different training circumstances, some during good years for financial markets, and others during global financial crises. However as the results (below) show, all three instances displayed positive returns that easily beat the risk free rate. The reason for selecting the best 3, 5 and 10 stocks is because the more stocks you choose in a portfolio, the lower your risk will be, but if you have too many, your returns just end up reflecting the returns of the whole market. The rule of thumb in stockbroking is that once you have more than about 8 stocks in a portfolio, you will struggle to outperform the market.

Testing range	3 stocks	5 stocks	10 stocks
90's	$1,190	$1,130	$1,230
00's	$1,460	$1,280	$1,220
10's	$1,350	$1,160	$1,130
avg	$1,333	$1,190	$1,193
avg annual growth	15.5%	10.91%	10.92%

Each of these figures is the result of investing $1000 into a portfolio containing either 3, 5 or 10 stocks and valuing that investment in 2 years time. Even the worst performing instance, which was the 5 stock portfolio from the 90's and the 10 stock portfolio from the 10's both achieved an average annual return of over 6%, which was nearly twice as good as the risk free rate.

Permutation of stock

Data Wrangling for permutation

```
value =list(data.iloc[0])[1:]  # filter the stock which the start price not located within IQR (25%-75%)
                               # remove too expansive or cheap
q3, q1 = np.percentile(value, [75 ,25]) #  get iqr
name  = data.keys() # get stock name
droplst=[]
for i in range(1, len(name)):
    if data.loc[0, name[i]]<q1 or data.loc[0,name[i]]>q3: # if price outside iqr then drop
        droplst.append(name[i])
data = data.drop( droplst,axis=1)
```

YRCW US Equity
144439.5121
143691.1208
143691.1208
145187.9033
145187.9033

When it came to the permutation of stock, we deleted the stocks that were bought at a very high/low price(delete data in iqr 0-25 and 75-100). Using YRCW US Equity as an example, let's say we were going to invest $10,000, because the unit price of this stock was too high for us to afford even one share, so I kept only 50% of the stocks at the appropriate price.

```
start =pd.DataFrame( data.groupby(['variable']).first()['value']) # get start price for each stock
```

```
end = pd.DataFrame( data.groupby(['variable']).last()['value']) # get end price for each stock
```

The main computation of the data is performed once the stocks have been cleaned up. To begin, I locate the opening and closing values of each retained stock and add them as two columns of data to the new table.

Risk calculation

```
def variance1(data):

    # Number of observations
    n = len(data)
    # Mean of the data
    mean = sum(data) / n

   # Square deviations
    deviations = [(x - mean) ** 2 for x in data]
   # Variance
    variance = sum(deviations) / n
    return variance
```

```
variance =[]
for i in stock_names:
    temp = []
    col = list(data[i])
    for j in range(1,len(col)):
        temp.append((col[j]-col[j-1])/col[j-1])
    variance.append(variance1(temp))
#     variance
```

After getting the opening and closing values, I calculated the variance, which is a statistical measure of how the numbers in a data set are distributed. More specifically, variance calculates how far off each number in the set is from both the mean and from each other. Both analysts and traders use it to assess market volatility and safety. My research led me to the conclusion that investors utilise variation to gauge the risk an investment entails and its likelihood of success. So, in this case, I compute the variance of the daily growth rate using the variance formula. According to my research, the higher the volatility indicates the greater the risk, while the smaller the variation indicates more steady development, the bigger the variance indicates higher volatility. For this project, I have thus regarded the variance as risk.

Here the variance is calculated in the same way as in mathematics: sum the data and divide by the number of data to get the mean, then subtract the mean from the value of each data in the data set and square it to get the deviations, and finally sum the deviations and divide by the number of data to get the variance

```
new_data['increasing_rate']= (new_data['end_price']-new_data['start_price'])/new_data['start_price']
```

The increasing rate was then determined by using the end price less the start price and dividing it by the start price. I utilise the upside as the rate of return in the project since a stock with a larger upside after investing would provide a bigger return.

New data frame containing important information

```
new_data
```

	stock	start_price	end_price	variance	increasing_rate
0	ABMD US Equity	18.3750	279.23	7991.838467	14.196190
1	ABT US Equity	10.2149	88.31	272.769806	7.645214
2	ADBE US Equity	16.6977	303.64	4118.666038	17.184540
3	ADSK US Equity	8.1434	160.29	1387.192527	18.683425
4	AEE US Equity	12.5805	76.21	204.157956	5.057788
...	...	...	...	...	...
243	WOR US Equity	8.9766	40.87	166.739993	3.552949
244	WY US Equity	14.9738	26.09	56.065659	0.742377
245	X US Equity	25.8487	15.71	687.329233	-0.392232
246	XEL US Equity	8.4109	60.59	172.776754	6.203748
247	XOM US Equity	24.5770	75.35	428.596925	2.065875

248 rows × 5 columns

```
new_data = new_data[new_data['increasing_rate']>0]
```

```
new_data
```

	stock	start_price	end_price	variance	increasing_rate
0	ABMD US Equity	18.3750	279.23	0.000951	14.196190
1	ABT US Equity	10.2149	88.31	0.000151	7.645214
2	ADBE US Equity	16.6977	303.64	0.000521	17.184540
3	ADSK US Equity	8.1434	160.29	0.000476	18.683425
4	AEE US Equity	12.5805	76.21	0.000117	5.057788
...	...	...	...	...	...
242	WMB US Equity	13.1106	25.02	0.000897	0.908379
243	WOR US Equity	8.9766	40.87	0.000471	3.552949
244	WY US Equity	14.9738	26.09	0.000278	0.742377
246	XEL US Equity	8.4109	60.59	0.000191	6.203748
247	XOM US Equity	24.5770	75.35	0.000157	2.065875

225 rows × 5 columns

248 stocks are now kept, and the start price, end price, variance (risk), and increasing rate (return) are all added to a new table for simple follow-up study.

In contrast, a negative increasing rate indicates that the stock is in a losing position. As selecting the most profitable portfolio was our primary objective, I retained only companies with an upside greater than zero. After a final screening of the data, 225 stocks were retained in the end.

Permutation

```
import tqdm
iters =1000000
max_in = 0
permu = None
weight_best =None
for k in [5,7,9,11]:
    for i in tqdm.tqdm(range(iters)):
        per = new_data.sample(n=k).reset_index()
        weight= list(softmax(normalize(per['variance'])))
        sum_increase = sum([per.loc[i,'increasing_rate']*weight[i] for i in range(per.shape[0])])
        if sum_increase > max_in:
            max_in = sum_increase
            permu=per
            weight_best=weight
```

```
100%|██████████| 1000000/1000000 [10:43<00:00, 1552.88it/s]
100%|██████████| 1000000/1000000 [10:55<00:00, 1524.98it/s]
100%|██████████| 1000000/1000000 [11:14<00:00, 1482.42it/s]
100%|██████████| 1000000/1000000 [11:03<00:00, 1506.07it/s]
```

The first for loop limits the number of stocks to be bought to 5, 7, 9, and 11. The reason for starting with 5 is that if the number of stocks in the portfolio is too small, one stock will easily dominate. Then a sample of each portfolio is taken from a group of randomly chosen portfolios. At first, the weight is set to the default value, it will later be changed.

We know the variance of each stock, calculate the weight from the variance, and then calculate the growth rate of the portfolio by scaling the weight to the growth rate of the portfolio, which then results in sum_increase combining risk and return information. If this sector is hit hard, the rest of the stocks in the same sector will fall at the same time. The

sum_increase is used as the final optimization target, and if a higher sum_increase is found, the portfolio and weights are updated.

Portfolio containing 5 stocks

	index	stock	start_price	end_price	variance	increasing_rate
0	232	URI US Equity	17.1250	129.95	0.000715	6.588321
1	196	SBAC US Equity	18.7500	248.43	0.001046	12.249600
2	77	EHC US Equity	24.0621	64.83	0.001086	1.694279
3	35	BLK US Equity	12.3490	478.41	0.000324	37.740789
4	143	MGM US Equity	12.2273	30.55	0.000770	1.498507

The final best portfolio is composed of 5 stocks: 'URI US Equity','SBAC US Equity','EHC US Equity','BLK US Equity','MGM US Equity'

Efficient Frontier

A portfolio is said to be efficient if there is no other portfolio that offers higher returns for a lower or equal amount of risk.It is worth noting, there is no specific solution for this however, there is a cluster of solutions known as the efficient frontier. According to the Markowitz Portfolio theory, the efficient frontier is defined as a set of investment portfolios that are expected to provide the highest returns at a given level of risk.When creating the efficient frontier using random portfolios, we looked at several algorithms/simulations namely, Monte Carlo simulation, The Mathematical Optimization algorithm and the RiskFolio-Library. Since all the algorithms had relatively similar results, we used them for comparison and contrast of the portfolios that maximise returns for the risk assumed. RiskFolio-Lib was used in asset allocation as well as portfolio optimization. This is due to its ability in constructing investment portfolios based on mathematically complex models with low effort.We generated 25,000 random portfolios with a risk free rate of 2.5% in 252 trading days as it is assumed that the analysis is being done in 2020 .

Data Wrangling for Efficient Frontier

Following the completion of exploratory data analysis, the returns and statistics necessary to run all the aforementioned algorithms and simulations were computed. We got the daily returns for each stock using the pct_change() method. It was the preferred method as they have material of being asset-additive which in turn is needed to compute the portfolio returns. Subsequently, the daily returns were annualised using the .mean() method.

```
[ ]  mean_returns = returns_portfolio.mean() * 252
     covar = returns_portfolio.cov() * 252
     correlation = returns_portfolio.corr() * 252
     rf = 0
```

```
returns = data1.pct_change()
mean_returns = returns.mean()
cov_matrix = returns.cov()
num_portfolios = 25000
risk_free_rate = 0.0250
```

The risk factor in returns_portfolio is set to 0 because it will use Monte Carlo simulation whereas data1 will apply the Mathematical Optimization Algorithm. Thus, using a risk-free rate of 2.50% in the 52-week treasury bill rate of 2020.

The Monte Carlo Simulation

While applying this simulation,we set out to mimic a set of random portfolios in order to visualise the risk-return profiles of our given set of assets. Consequently, I defined 2 functions that take inputs of asset weights and output the expected portfolio return and standard deviation.

```
[74] #  daily returns = returns_portfolio
     # Function for computing portfolio return
     def portfolio_returns(weights):
       port = (np.sum(returns_portfolio.mean() * weights)) * 252
       return port

[75] # returns_portfolio

[76] #  Standard Deviation of the portfolio returns
     # Function for computing standard deviation of portfolio returns
     def portfolio_sd(weights):
       sdv = np.sqrt(np.transpose(weights) @ (returns_portfolio.cov() * 252) @ weights)
       return sdv
```

A for loop was then created to simulate the random vectors of asset weights by processing the expected portfolio and standard deviation for each combination of weights. This action occurs randomly. It Is worth noting that the Monte Carlo simulation is computationally intensive due to the size of the dataset hence, making us settle for mathematical optimization.

```
# instantiate empty array containers for returns and sd
list_portfolio_returns = []
list_portfolio_sd = []
p_weights = [] # Define an empty array for asset weights

# For loop to simulate 2000 random weight vectors (numpy array objects)
for p in range(10):
  # Return random floats in the half-open interval [0.0, 1.0)
  weights = np.array(np.random.random(299))
  # Normalize to unity
  # The /= operator divides the array by the sum of the array and rebinds "weights" to the new object
  weights /= np.sum(weights)
  p_weights.append(weights)

  # Lists are mutable so growing will not be memory inefficient
  list_portfolio_returns.append(portfolio_returns(weights))
  list_portfolio_sd.append(portfolio_sd(weights)) # do append and calculation separate

  # Convert list to numpy arrays
  port_returns = np.array(object = list_portfolio_returns)
  port_sd = np.array(object = list_portfolio_sd)

  sharpe_ratio = port_returns/port_sd
```

The Mathematical Optimization Algorithm

This algorithm uses optimization functions such as the SciPy library to find the optimal weights mathematically. SciPy's in-built optimization algorithm creates an optimizer that

attempts to minimise the negative Sharpe ratio. In turn, this helps calculate the weight allocation for the portfolio. The code below illustrates how to do it using python.

```
def neg_sharpe_ratio(weights, mean_returns, cov_matrix, risk_free_rate):
    p_var, p_ret = portfolio_annualised_performance(weights, mean_returns, cov_matrix)
    return -(p_ret - risk_free_rate) / p_var

def max_sharpe_ratio(mean_returns, cov_matrix, risk_free_rate):
    num_assets = len(mean_returns)
    args = (mean_returns, cov_matrix, risk_free_rate)
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
    bound = (0.0,1.0)
    bounds = tuple(bound for asset in range(num_assets))
    result = sco.minimize(neg_sharpe_ratio, num_assets*[1./num_assets,], args=args,
                          method='SLSQP', bounds=bounds, constraints=constraints)
    return result
```

Due to the lack of the maximise function in SciPy, the function passes something that is minimised hence the neg_sharp_ratio. SciPy's minimise function can be used to generate portfolios that will also plot the efficient frontier as seen in the next page.

```
def display_calculated_ef_with_random(mean_returns, cov_matrix, num_portfolios, risk_free_rate):
    results, _ = random_portfolios(num_portfolios,mean_returns, cov_matrix, risk_free_rate)

    max_sharpe = max_sharpe_ratio(mean_returns, cov_matrix, risk_free_rate)
    sdp, rp = portfolio_annualised_performance(max_sharpe['x'], mean_returns, cov_matrix)
    max_sharpe_allocation = pd.DataFrame(max_sharpe.x,index=data1.columns,columns=['allocation'])
    max_sharpe_allocation.allocation = [round(i*100,2)for i in max_sharpe_allocation.allocation]
    max_sharpe_allocation = max_sharpe_allocation.T

    min_vol = min_variance(mean_returns, cov_matrix)
    sdp_min, rp_min = portfolio_annualised_performance(min_vol['x'], mean_returns, cov_matrix)
    min_vol_allocation = pd.DataFrame(min_vol.x,index=data1.columns,columns=['allocation'])
    min_vol_allocation.allocation = [round(i*100,2)for i in min_vol_allocation.allocation]
    min_vol_allocation = min_vol_allocation.T
```

We have achieved relatively similar results with the randomly generated portfolios; however, this is simulated. The slight difference is that the Scipy's "optimise" function was not allocated any budget for MGM Us Equity on maximum Sharpe ratio portfolio, while one we chose from the randomly generated samples has 1.84% of allocation for the MGM Group.

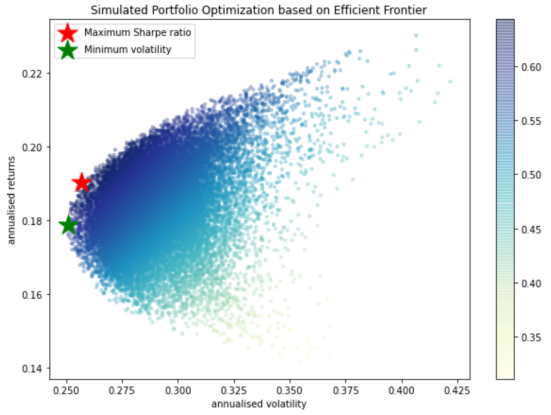
```
display_simulated_ef_with_random(mean_returns, cov_matrix, num_portfolios, risk_free_rate)
```

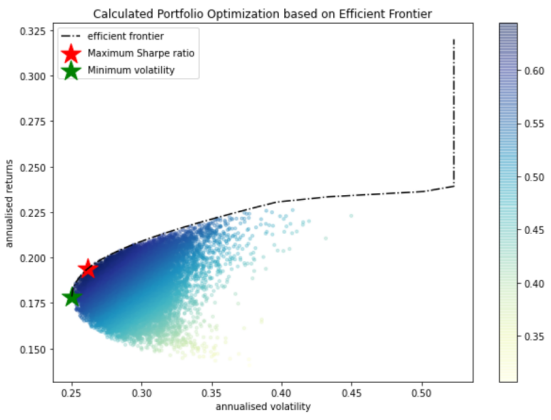
```
--------------------------------------------------------------------------------
Maximum Sharpe Ratio Portfolio Allocation

Annualised Return: 0.19
Annualised Volatility: 0.26


            URI US Equity  SBAC US Equity  EHC US Equity  BLK US Equity  \
allocation            8.9           16.54          20.01          52.71

            MGM US Equity
allocation           1.84
--------------------------------------------------------------------------------
```

```
display_calculated_ef_with_random(mean_returns, cov_matrix, num_portfolios, risk_free_rate)
```

```
--------------------------------------------------------------------------------
Maximum Sharpe Ratio Portfolio Allocation

Annualised Return: 0.19
Annualised Volatility: 0.26


            URI US Equity  SBAC US Equity  EHC US Equity  BLK US Equity  \
allocation           6.83           18.81          22.22          52.14

            MGM US Equity
allocation            0.0
--------------------------------------------------------------------------------
```

The mathematical algorithm(calculated) also achieved a different visual from the simulated frontier as seen below. It is assumed that this is due to the lack of a risk free rate in the simulated frontier.





RiskFolio-Lib

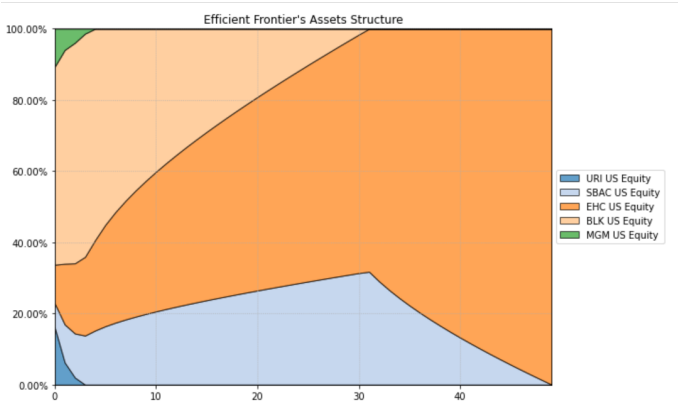
Riskfolio-Lib is a library for making portfolio optimization and quantitative strategic asset allocation in Python. Its objective is to build investment portfolios based on mathematically complex models with low effort. It is known to be built on top of CVXPY and closely integrated with pandas data structures. It allows us to calculate the optimum portfolios using 4 functions: the maximum return portfolio, maximum risk adjusted return ratio portfolio, minimum risk portfolio and maximum utility portfolio. The code below estimates the mean variance portfolio by calculating the portfolio that maximises the Sharpe ratio. This in turn enables us to get the visuals in terms of asset structure and the Sharpe mean variance.

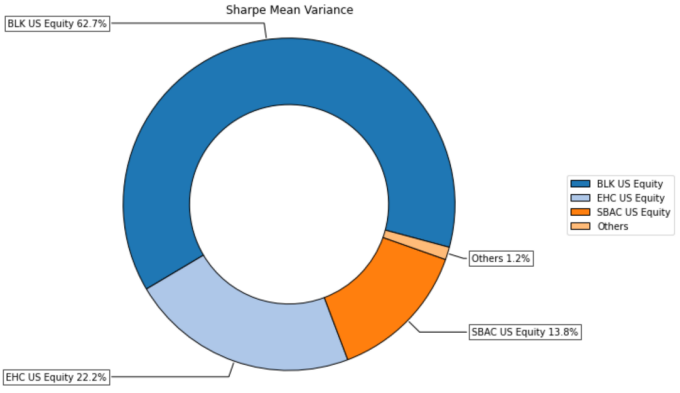
```
# Building the portfolio object

port = rp.Portfolio(returns=ye)

# calculating optimal portfolio
# Select method and estimate input parameters:

methmu = 'hist' #Method to estimate expected returns based on historical data
methcov= 'hist' #Method to estimate covariance matrix based on historical data

port.assets_stats(methmu=methmu,methcov=methcov, d=0.94)

#  Estimate the optimal portfolio
model = 'Classic' #Could be Classic(historical), BL OR FM
rm = 'MV'#Variance
obj = 'Sharpe' #Objective function could be MinRisk, MaxRet, Utility or Sharpe Ratio
hist = True
rf =  0 #rf=risk free rate
l= 0 # only useful when obj is 'Utility'

w = port.optimization(model=model, rm=rm, obj=obj, rf=rf, l=l, hist=hist)
yu = w.T
yu
```

]:

	URI US Equity	SBAC US Equity	EHC US Equity	BLK US Equity	MGM US Equity
weights	3.269278e-08	0.138498	0.22234	0.626875	0.012287

Data Visualisation of the efficient frontiers.





BLK US Equity takes up the most in the portfolio composition and Sharpe Mean Variance as it is the most stable stock in the portfolio. Hence it is intuitive to allocate most of the budget to it. While EHC Us Equity has the highest risk/reward ratio, thus making it have the second most allocation in the portfolio. The MGM and SBAC US Equities have least allocation in both visuals as they have the lowest returns out of the 5 in the portfolio.

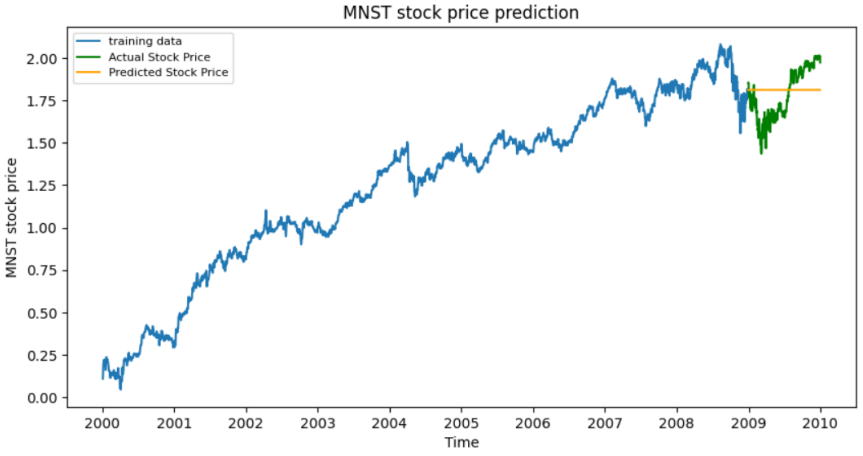
ARIMA model

An Auto-Regressive Integrated Moving Average (ARIMA) model is a machine learning model that uses past data to forecast and extrapolate in order to give users a likely idea of where their data is heading. ARIMA models are used in many scenarios, but can be most productive at forecasting sales and determining future demand for food to help supply chains cope. ARIMA can be broken down into three parts, AR, I and MA. Each of these three parts are associated with a variable that you must enter into the function to perform a forecast. The three parameters p, d and q, correspond to each part of the word ARIMA previously highlighted. The p represents the number of previous points that the model considers when making forecasting its next value, the d represents the difference in the nonseasonal observations and the q represents the size of the moving average window.

```
model_autoARIMA = auto_arima(train_data, start_p=0, start_q=0,
                      test='adf',       # use adftest to find optimal 'd'
                      max_p=3, max_q=3,
                      m=1,
                      d=None,           # let model determine 'd'
                      seasonal=False,   # No Seasonality
                      start_P=0,
                      D=0,
                      trace=True,
                      error_action='ignore',
                      suppress_warnings=True,
                      stepwise=True)
print(model_autoARIMA.summary())
model_autoARIMA.plot_diagnostics(figsize=(15,8))
plt.show()
```

```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=-18562.120, Time=1.01 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=-18601.478, Time=1.46 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=-18601.143, Time=2.35 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=-18559.796, Time=0.22 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=-18599.490, Time=3.43 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=-18599.489, Time=5.44 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=-18597.489, Time=3.29 sec
 ARIMA(1,1,0)(0,0,0)[0]             : AIC=-18598.117, Time=0.85 sec

Best model:  ARIMA(1,1,0)(0,0,0)[0] intercept
Total fit time: 18.113 seconds
```

The findings were that ARIMA models were not very effective at predicting stock prices. The parameters for the ARIMA model were tested both by hand and by using an autoARIMA function to optimise p, d and q (shown left). However, the ARIMA model was still unable to accurately predict stock prices. This is largely because stock prices past performance is not a good indication of future performance, and the model cannot analyse risk and outside factors and use them to help its forecast. It simply uses a stocks past performance to predict its future performance. Instances like a global financial crisis also really hurt the models accuracy, in both training and testing. All results with the ARIMA model looked like the graph shown (below) where it really didn't predict much movement in the stock price, more just an upward or downward trend if anything.



Weight Calculation

Relative Weight

```
import itertools
pp = []
for i in tqdm.tqdm(itertools.permutations([i/100 for i in range(100) ], len(weight_best))):
    if sum(i)==1 and min(i)>0.1 and max(i)<0.8:
        pp.append(i)
```

```
9034502400it [36:21, 4141451.59it/s]
```

```
max_inc= max_in
new_weight= None
for i in range(len(pp)):
    sum_in = sum([pp[i][j]*permu.loc[j,'increasing_rate'] for j in range(len(pp[i]))])
    if sum_in>max_inc:
        max_inc=sum_in
        new_weight= pp[i]
if not new_weight:
    new_weight = weight_best
```

In calculating the relative weights, I have taken a random sample and set a threshold for the weights. This threshold states that there can't be a weight less than 0.1 or greater than 0.8 for each of the 5 stocks. The purpose of this is to not allow one stock to dominate, otherwise there would be no difference between a portfolio of 5 stocks and an investment in only one stock. A further condition is that the investment weights of the five stocks must add up to 1. This is until I achieve a greater return than that achieved using the softmax and normalise methods, while maintaining the newly determined weights. The result is the following graph

```
new_weight
```

```
[0.038222488966547355,
 0.016752049221367738,
 0.015675265052862097,
 0.8976166938774541,
 0.03173350288176866]
```

Absolute Weight

```
money =10000
static_weight = [money*new_weight[i]//permu.loc[i,'end_price'] for i in range(len(new_weight))]
```

```
for i in range(len(new_weight)):
    print('Stock Name: ', permu.loc[i,'stock'],end=' ')
    print('Buying Number: ', static_weight[i])
```

```
Stock Name:  URI US Equity Buying Number:  2.0
Stock Name:  SBAC US Equity Buying Number:  0.0
Stock Name:  EHC US Equity Buying Number:  2.0
Stock Name:  BLK US Equity Buying Number:  18.0
Stock Name:  MGM US Equity Buying Number:  10.0
```

Relative weights give a percentage result, but weights in equity investing are not feasible because in the equity market, whole shares must be purchased and there are no percentages. In order to be able to calculate the number of shares to be purchased per stock, I have calculated a new weighting called the absolute weighting.

Setting a total investment amount is necessary before doing Absolute Weight, and given that I have already eliminated the pricey stocks, I believe a $10,000 investment fund to be an acceptable level. The sum is divided by the rounded-down closing price of each stock in accordance with the previously determined percentage of Relative Weight. Since the equities may only be purchased in entire multiples, as was previously stated, I've opted to round down here because there may not be enough money to round up all of the stocks. The calculation's output is the absolute weight.

Dynamic Weight

Dynamic weighting was done by comparing the difference between the start and end prices (range), variance (risk), and increasing rate (rate) of these five stocks over a 19-year period and the last 100 days of the 19-year period.

```
def dynamic_weight(data,stock,weight):
    whole_ranges=[]
    whole_risk=[]
    whole_rate=[]
    recent_ranges= []
    recent_risk=[]
    recent_rate=[]
    whole_ranges,whole_rate,whole_risk = overall(data,stock)
    recent_ranges,recent_rate,recent_risk = overall(data.tail(100).reset_index(),stock)
```

whole_xxx represents the data over a 19-year period, and recent_xxx represents the base data for the last 100 days.

```
    ranges_change_rate= []
    risk_change_rate=[]
    rate_chage = []
    for i in range(len(whole_ranges)):
        ranges_change_rate.append(( recent_ranges[i]- whole_ranges[i])/whole_ranges[i])
        risk_change_rate.append(( recent_risk[i]- whole_risk[i])/whole_risk[i])
        rate_chage.append(( recent_rate[i]- whole_rate[i])/whole_rate[i])
    joint = []
    for i in range(len(whole_ranges)):
        joint.append(ranges_change_rate[i]*risk_change_rate[i]*rate_chage[i])
```

In the case of getting the last 100 days of range, risk, rate and the rate of change of range, risk, rate during the 19 years, multiply these rates together to get a probability, which represents the assumption that the percentage of decline in range during the 100 days and 19 years will affect the percentage of purchases, so our weights are also affected by range, risk, rate, Joint is the result of combining these several characteristic values, for example, the result of joint is a decline of 10 percent, then the original weight should also decline by 10 percent.

```
    #modify weigth
    max_idex = weight.index(max(weight))
    if max_idex >0.7:
        weight[max_idex] *= 0.8

    res = weight[max_idex]*0.2/(len(stock)-2)
    for i in range(len(weight)):
        if i != max_idex:
            weight[i]+=res

    for i in range(len(weight)):
        weight[i]=weight[i]*(1+joint[i])
    return (softmax(normalize(weight)))
```

The final step in calculating the dynamic weights is to modify the weights, here limiting the situation where one weight dominates. This is because if a stock has a weight of 0.9 and the remaining 4 stocks share the remaining 0.1 equally, then the permutation of the stock selection done before would be meaningless, the risk would be high and the difference with a straightforward selection of one stock would not be great. So when the weighting is greater than 0.7, I keep 80% of the original weighting of this and then divide the original 20% equally between the remaining 4 stocks so that it does not lead to too much risk.

Finally, I modified the weights by multiplying the original weight by (1+joint) using the values of the joints I had made before to get the new weights. However, after multiplication, the weights of the 5 stocks must not add up to 1, so in the end we use normalise and softmax to map the weights to the (0,1) interval and add up to 1.

The final value of the dynamic weights is:

```
dynamic_weight(data,stock,weight)
```

```
array([0.16579363, 0.15012032, 0.14542474, 0.38242879, 0.15623252])
```

Prediction Model using LSTM

```
money=10000
weight =[0.16579363, 0.15012032, 0.14542474, 0.38242879, 0.15623252]
stock_count = [money*weight[k]//data.loc[0,stock[k+1]] for k in range(len(weight))]
data['weight_sum']=0
for i in range(data.shape[0]):
    data.loc[i,'weight_sum'] = sum([stock_count[j]*data.loc[i,stock[j+1]] for j in range(len(weight))])
```

Assuming that the stocks were bought on the first day of the selected data set, that is January 1st-2000, now we need to use the value of the dynamic weights to calculate how much to buy for each stock, so multiply the investment capital by the weight, and get the amount spent on each stock and divide it by the price of the stock, which is equal to the number of stocks bought, so the stock_count here is the number of stocks bought for each stock

Weight_sum means that after we buy a stock according to the dynamic weight, we need to calculate the amount spent on this weight by multiplying the stock_count (the number of

stocks bought) by the price of each stock, and then summing the amount spent on the 5 stocks, stock[j+1] because in the table of data the first column is the date.

```
df=  data[['Date','weight_sum']]
```

```
df
```

	Date	weight_sum
0	20000101	9956.4341
1	20000102	9956.4341
2	20000103	9601.5087
3	20000104	9417.0209
4	20000105	9300.3805
...	...	...
7146	20190726	186872.1800
7147	20190727	186872.1800
7148	20190728	186872.1800
7149	20190729	187055.3200
7150	20190730	187947.9400

7151 rows × 2 columns

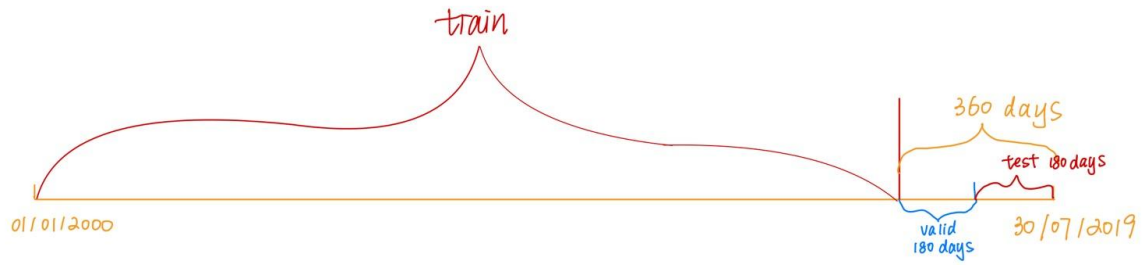
The first row of this subdivision represents our invested capital of $10,000, and on the first day, January 1st - 2000, $9,956,4341 was invested, and from the second row onward represents the change in investment amount for each day thereafter, assuming that the first day is weighted according to dynamic weights and the amount invested is $9,956,4341.

```
from  sklearn.preprocessing  import  MinMaxScaler
scaler  =  MinMaxScaler(feature_range=(0,1))
scaled_data  =  scaler.fit_transform(x.reshape(-1,1))
```

In this step we will use the LSTM model, which is a neural network, so the data must be normalised before being put into the data, so the MinMaxScaler() function is introduced here, and scaled_data is the last data to be put into the model.
Because our prediction method uses the previous day's data to predict the next day, it will result in an extra day of data, so one needs to be deleted.

```
x_train,y_train  =  x[:len(x)-360],y[:len(y)-360]
x_valid,y_valid  =  x[len(x)-360:len(x)-180],y[len(y)-360:len(y)-180]
x_test,y_test  =  x[len(x)-180:],y[len(y)-180:]
date_train,date_valid,date_test=  date[:len(date)-360],date[len(date)-360:len(date)-180],date[len(date)-180:]
```

The next step is to split the test set, the valid set and the training set. The language description will not be easy to understand, so we use images to express how we split. The following figure is how we split:




```
x_train=x_train.reshape(-1,1)
y_train=y_train.reshape(-1,1)
x_valid=x_valid.reshape(-1,1)
y_valid=y_valid.reshape(-1,1)
x_test=x_test.reshape(-1,1)
y_test=y_test.reshape(-1,1)
```

Now we have a one-dimensional list, but to put it into input_shape, we need to turn it into two-dimensional data.Then there is the modelling part.

```
opt = keras.optimizers.Adam(learning_rate=0.0001)
model.compile(optimizer=opt, loss='mean_squared_error')
model.fit(x_train, y_train, batch_size= 128, epochs=20,validation_data=(x_valid, y_valid))
```

```
Epoch 1/20
54/54 [==============================] - 6s 27ms/step - loss: 0.1023 - val_loss: 0.4096
Epoch 2/20
54/54 [==============================] - 1s 10ms/step - loss: 0.0649 - val_loss: 0.2544
Epoch 3/20
54/54 [==============================] - 1s 13ms/step - loss: 0.0381 - val_loss: 0.1297
Epoch 4/20
54/54 [==============================] - 1s 10ms/step - loss: 0.0273 - val_loss: 0.0810
Epoch 5/20
54/54 [==============================] - 1s 10ms/step - loss: 0.0199 - val_loss: 0.0517
Epoch 6/20
54/54 [==============================] - 1s 10ms/step - loss: 0.0117 - val_loss: 0.0248
Epoch 7/20
54/54 [==============================] - 1s 10ms/step - loss: 0.0047 - val_loss: 0.0059
Epoch 8/20
54/54 [==============================] - 1s 10ms/step - loss: 0.0011 - val_loss: 6.7322e-04
Epoch 9/20
54/54 [==============================] - 1s 10ms/step - loss: 1.8427e-04 - val_loss: 1.3604e-04
Epoch 10/20
54/54 [==============================] - 1s 10ms/step - loss: 8.3362e-05 - val_loss: 1.8794e-04
Epoch 11/20
54/54 [==============================] - 1s 10ms/step - loss: 7.7430e-05 - val_loss: 2.0530e-04
Epoch 12/20
54/54 [==============================] - 1s 13ms/step - loss: 7.6187e-05 - val_loss: 2.2066e-04
Epoch 13/20
54/54 [==============================] - 1s 21ms/step - loss: 7.5090e-05 - val_loss: 2.0543e-04
Epoch 14/20
54/54 [==============================] - 1s 13ms/step - loss: 7.3795e-05 - val_loss: 2.1220e-04
Epoch 15/20
54/54 [==============================] - 1s 16ms/step - loss: 7.2625e-05 - val_loss: 2.0904e-04
Epoch 16/20
54/54 [==============================] - 1s 10ms/step - loss: 7.1175e-05 - val_loss: 2.0164e-04
Epoch 17/20
54/54 [==============================] - 1s 10ms/step - loss: 6.9918e-05 - val_loss: 1.8824e-04
Epoch 18/20
54/54 [==============================] - 1s 10ms/step - loss: 6.8506e-05 - val_loss: 1.9677e-04
Epoch 19/20
54/54 [==============================] - 1s 10ms/step - loss: 6.7129e-05 - val_loss: 1.9644e-04
Epoch 20/20
54/54 [==============================] - 1s 10ms/step - loss: 6.5781e-05 - val_loss: 1.9726e-04
<keras.callbacks.History at 0x7f8d3081e290>
```

Here we can see that the loss is in a decreasing process, which means that the model is finding the optimal solution. From the results, we can see that both training and validation are in the state of always seeing smaller, so there is no overfitting of the model.

```
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)
```

predictions is all the predicted data, the first line is to put the tested data into prediction, because we have performed MinMaxScaler on the original data (the previous data are between 0-1), now we need to return the data to the original value , the second line scaler.inverse_transform is the process of returning the data to the original value

```
final_data
```

	Date	weight_sum	pred_weight_sum
6971	2019-02-01	161198.2681	161673.718750
6972	2019-02-02	161198.2681	162283.218750
6973	2019-02-03	161198.2681	162283.218750
6974	2019-02-04	161968.8144	162283.218750
6975	2019-02-05	161987.6112	163109.531250
...	...	...	...
7146	2019-07-26	186872.1800	188221.265625
7147	2019-07-27	186872.1800	190127.593750
7148	2019-07-28	186872.1800	190127.593750
7149	2019-07-29	187055.3200	190127.593750
7150	2019-07-30	187947.9400	190328.484375

180 rows × 3 columns

We then call the sum of the actual values of the 5 stocks in the portfolio weight_sum and the sum of the predicted values pred_weight_sum. so we now get the results for the actual and predicted prices of the portfolio from February 1, 2019 to July 30, 2019.

According to the existing data, we can calculate the growth rate of the forecast data, the formula is: where n is the year of the data used, so n=19

$$\text{start price} \cdot (1+x)^{n} = \text{end price}$$

The final result is that the growth rate is around: 17%

Predict one month later

```
def  minmaxscalar(x, _min, _max):
    return  (x-_min)/(_max-_min)
def  reverse_minmax(x, _min, _max):
    return  x*(_max-_min)+_min
```

The first step in making a forecast for the next 40 days is to define the generalisation function minmaxscalar() is forward, which serves to compress a value to a smaller range, and reverse_minmax is reverse, which serves to return the compressed value to its original size.

```
days=  30
min_weight  =  df['weight_sum'].min()
max_weight  =  df['weight_sum'].max()*(1+0.17)
```

Because of the external factors that can cause stock market volatility and the limitations of the model, we do not intend to forecast for too long. Also, in normal stock market forecasts are usually short time forecasts, so we intend to forecast data for the next 30 days. In max_weight we do rescale, the formula is:

$$\frac{X - X_{min}}{X_{max} - X_{min}}$$

However,after the prediction,there will be a value greater than the maximum value. At this time,the result of the formula will be greater than 1,when it is greater than 1,the data will increase sharply,which will magnify the prediction result many times,resulting in inconsistency with the original distribution.So in this step,we need to redefine max_weight. The way to define it is to calculate the growth rate before,assuming that the portfolio we selected will grow according to the previous growth rate,df['weight_sum'].max()*(1+0.17) restricts the value of max_weight to be less than 1, to prevent sudden growth, so that the distribution is in a stable state.

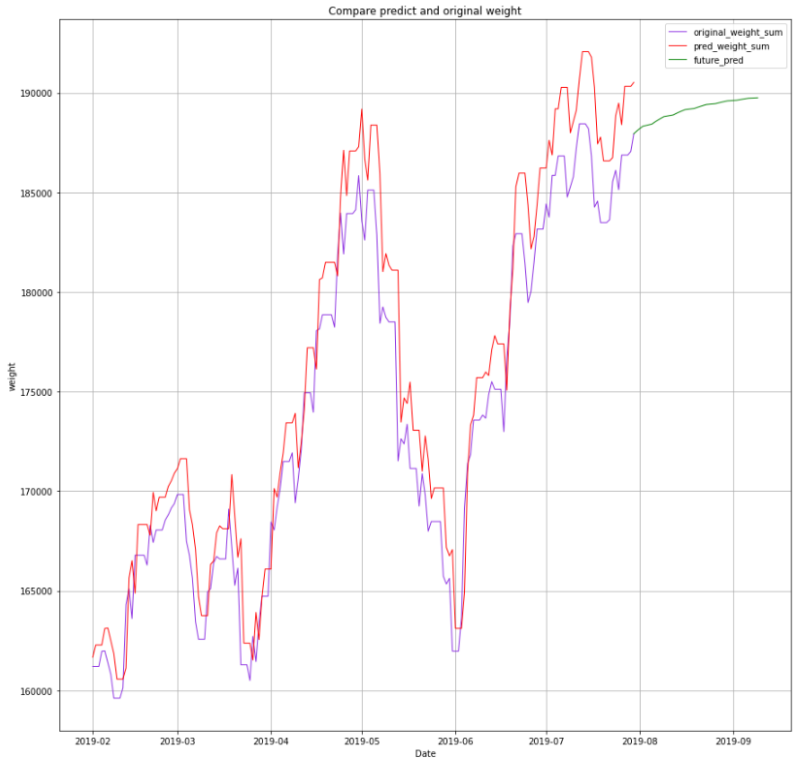
```
new_x,new_y  =[],[]
temp=[]
for  i  in  range(days):
    new_x.append(initial_sum[0][0])
    initial_sum[0][0]  =  minmaxscalar(initial_sum[0][0],min_weight,max_weight)
    pred  =  model.predict(initial_sum,verbose=False)
    pred  =  reverse_minmax(float((pred)),min_weight,  max_weight)

    new_y.append(pred)
    initial_sum[0][0]=pred
```

The format of Initial_sum is [[ ]] because it is a fixed format in the input model. The data it saves is the value of the last day, and then the Initial_sum is normalised by minmaxscaler, and it is changed to the range of 0-1 before putting it into the model. After the prediction, the real value is returned through reverse_minmax, and the predicted result is saved in new_y. The last update of Initial_sum is to use the predicted value as input for loop operation.

```
from datetime import timedelta, date
def daterange(date1, date2):
        for n in range(int ((date2 - date1).days)+1):
                yield date1 + timedelta(n)

start_dt = date(2019,7,30)
end_dt = date(2019,12,31)

weekdays = [5,6]
day_list=[]
for dt in daterange(start_dt, end_dt):
        if dt.weekday() not in weekdays:
                day_list.append(dt.strftime("%Y-%m-%d"))
        if len(day_list)==days:
            break
new_df =pd.DataFrame( pd.to_datetime(day_list, format='%Y-%m-%d'))
new_df['pred_weight_sum']=new_x
new_df= new_df.rename(columns={0:'Date'})
```

Then set the start date of the forecast: July 30, 2019 The end date of the forecast: December 31, 2019, because in the original data, the data on Saturday and Sunday were the same as the closing on Friday, and the price did not start to fluctuate until Monday. , so only the data from Monday to Friday is used here, so that the forecast image is relatively flat, and there will be no ladder-shaped upward trend.

The final predicted image is as follows:



Origin_weight_sum is the image of the portfolio from February 1, 2019 to July 30, 2019
Pred_weight_sum is an image of the portfolio from February 1, 2019 to July 30, 2019
Future_pred is an image of the forecasted future 40 days from July 31, 2019 to September 9, 2019

Since we have real data until July 30, 2019, our Future_pred is directly connected to do future predictions when the real data is available

The data predicted for the next 40 days are as follows.

0	2019-07-30	187947.940000
1	2019-07-31	188081.080827
2	2019-08-01	188205.873614
3	2019-08-02	188322.878821
4	2019-08-05	188432.543275
5	2019-08-06	188535.356361
6	2019-08-07	188631.722352
7	2019-08-08	188722.109352
8	2019-08-09	188806.815249
9	2019-08-12	188886.223038
10	2019-08-13	188960.673160
11	2019-08-14	189030.484777
12	2019-08-15	189095.913221
13	2019-08-16	189157.277653
14	2019-08-19	189214.790851
15	2019-08-20	189268.729421
16	2019-08-21	189319.327418
17	2019-08-22	189366.755060
18	2019-08-23	189411.246402
19	2019-08-26	189452.950385
20	2019-08-27	189492.037231
21	2019-08-28	189528.719714
22	2019-08-29	189563.104223
23	2019-08-30	189595.318423
24	2019-09-02	189625.532533
25	2019-09-03	189653.874220
26	2019-09-04	189680.428593
27	2019-09-05	189705.344595
28	2019-09-06	189728.728615
29	2019-09-09	189750.623206

According to the data, we can calculate that the forecast return after 40 days is:1802.683206

Conclusion

Overall, we were able to investigate numerous ways in which the portfolio may behave by taking into account many elements in our optimisation, including the risk of the stock, the return, the price of the stock, the number of portfolios created, and even the set of limited dates. We were able to see, for instance, that the portfolios often performed better when five equities were pooled rather than seven or even more when we finished our strategy by employing permutation.

The financial crisis was included in the data we used to build the LSTM model, which can help to mitigate some of the stock market volatility it created. Finally, we looked into using models to predict the data. The ARIMA model struggled at predicting more than a trend in stock price and this was investigated and explained but not used for our calculations. The LTSM model was split into two parts. The first uses the previous 180 days of known data as the target for prediction. After prediction, we discovered that the predicted graph is very similar to the graphical trend of the actual data, indicating that our model is workable. The second part predicts the following 40 days, which results in a return of 1802.68, with a monthly return of 1%. However, LSTM is particularly well suited to handle the issue with time series, although the size of the data time span and LSTM's incapacity to retain the data for a lengthy period of time are also issues.

Appendix

Softmax and Normalise in permutation

```
def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    x= 1/x
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0) # only difference
def normalize(v):
    norm=np.linalg.norm(v)
    if norm==0:
        norm=np.finfo(v.dtype).eps
    return v/norm
```

```
weight= list(softmax(normalize(per['variance'])))
```

I chose the stocks using the permutation approach. The initial weights are calculated using two equations: softmax and normalise.

The purpose of softmax, also known as the normalised exponential function, is used to represent the outcomes of numerous classifications as probabilities. It translates the output of many neurons into the (0,1) range, which is consistent with the requirement that the weights total up to 1. But in this case, I changed a few things: x = 1/x is a new line that I inserted. This is due to the fact that it would not make sense to exclude the 1/x symbol to signify that the riskier a stock is, the more of it is purchased. The modification here indicates that the

risk affects how much stock is purchased. The likelihood of buying is higher the smaller the danger.

Normalisation is a crucial step, sometimes referred to as the normalisation function. It can successfully scale the variance's value to fall between 0 and 1. By way of illustration, we scale the first attribute of the training data from [-10,+10] to [-1, +1] using this approach, which only modifies the value but not the feature. This prevents the characteristics in the big value interval from outweighing the attributes in the small value interval in the first place, and it also enables softmax of the variance values.

Overall function in Dynamic weight

```
def overall(data,stock):
    ranges= []
    risk= []
    rate= []
    for i in range(1,len(stock)):
        ranges.append(max(data[stock[i]])-min(data[stock[i]]))
        rate.append((float( data[stock[1]].tail(1))-float(data[stock[1]].head(1)))/float(data[stock[i]].head(1)))
        temp_in =[]
        for j in range(1,data.shape[0]):
            temp_in.append((data.loc[j,stock[i]] - data.loc[j-1,stock[i]])/data.loc[j,stock[i]])
        risk.append(variance1(temp_in))
    # cut dominate
    return ranges, rate,risk
```

The overall function have parameters date and the name of the stock and is used to calculate the ranges (the difference between the start and end prices, representing how much money has been gained), the risk (representing the variance of the daily increasing rate) and the growth rate (how much a stock has grown over a period of time)

MinMaxScaler in LSTM Model

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(x.reshape(-1,1))
```

MinMaxScaler() can normalize any value to a certain interval.

It has the following prototype.

sklearn.preprocessing.MinMaxScaler(feature_range=(0, 1), copy=True)

where feature_range means the normalized range. copy defaults to True, which is the copy property, and defaults to True, which means that the original

data set is copied, so that the meta-array remains unchanged after the transformation.