Stochastic Control and Optimization
Project 2 – Integer Programming

Submitted by Team 12: Emily McCullough (emm3863), Lydia Wang (lw28423), Ramya Desineedi (rd32895), Yashpreet Kaur(yk874)

Introduction

As a group of Financial analysts at a Mutual Fund organisation our objective of this project is to choose a portfolio, called as "index fund" (component stocks to include in the fund and their weights) that mirrors the movements of the broad market population or a market index such as NASDAQ-100. When we put together the portfolio for our index tracking mutual fund, the goal will be to closely replicate the fluctuations of a market index.

Dataset

We are using 2019 daily prices of the index as well as the component stocks of the NASDAQ-100 for portfolio construction tasks and then analyse the performance on the 2020 data.

Methodology Overview

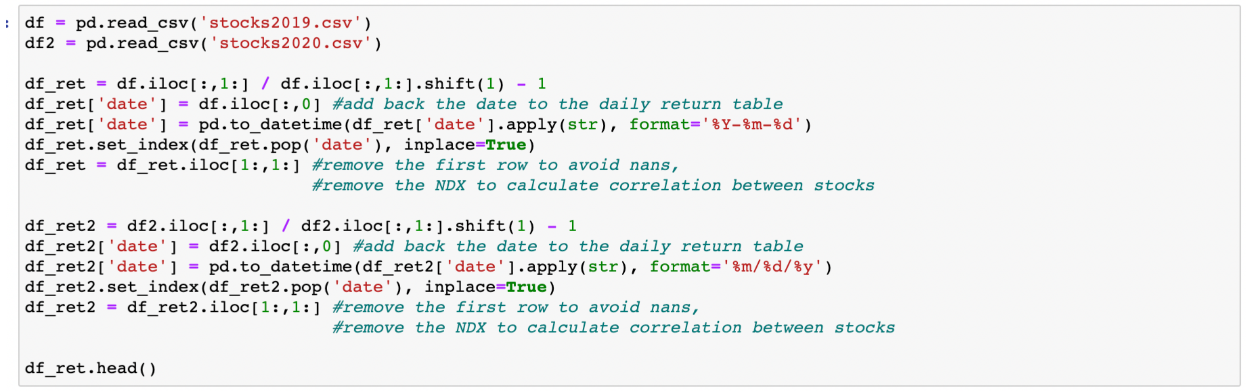
Brief summary of methodology for constructing Index fund:

1. **Stock Selection:** As a first step, we will formulate an ***integer program*** that picks exactly m out of n stocks for our portfolio. This integer program will take as input a similarity/correlation matrix (ρ), with individual elements of this matrix ρij, representing similarity between stock i and j.)
2. **Calculating Portfolio Weights:** Next, we will solve a ***linear program*** to decide how many of each chosen stock to buy for our portfolio (weightage of each stock in our fund)
3. **Performance evaluation:** Finally, we will evaluate how well our index fund performs in comparison to the NASDAQ-100 index. We will also test the performance for several values of m.

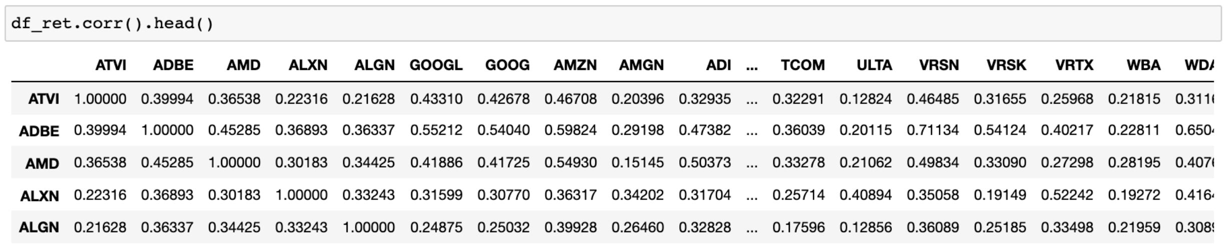
Calculating Returns of Stocks and Correlation Matrix

As a first step of index fund portfolio construction, we begin by computing the returns and correlation matrix, which will be used in our linear program to determine the best m stocks to invest in. Next, the correlation matrix was created using the stock returns, where we calculated the % difference in value for each time interval.

Returns and correlation matrix are calculated using the below code snippet in python:



```python
df = pd.read_csv('stocks2019.csv')
df2 = pd.read_csv('stocks2020.csv')

df_ret = df.iloc[:,1:] / df.iloc[:,1:].shift(1) - 1
df_ret['date'] = df.iloc[:,0] #add back the date to the daily return table
df_ret['date'] = pd.to_datetime(df_ret['date'].apply(str), format='%Y-%m-%d')
df_ret.set_index(df_ret.pop('date'), inplace=True)
df_ret = df_ret.iloc[1:,1:] #remove the first row to avoid nans,
                            #remove the NDX to calculate correlation between stocks

df_ret2 = df2.iloc[:,1:] / df2.iloc[:,1:].shift(1) - 1
df_ret2['date'] = df2.iloc[:,0] #add back the date to the daily return table
df_ret2['date'] = pd.to_datetime(df_ret2['date'].apply(str), format='%m/%d/%y')
df_ret2.set_index(df_ret2.pop('date'), inplace=True)
df_ret2 = df_ret2.iloc[1:,1:] #remove the first row to avoid nans,
                              #remove the NDX to calculate correlation between stocks

df_ret.head()
```



```python
df_ret.corr().head()
```

	ATVI	ADBE	AMD	ALXN	ALGN	GOOGL	GOOG	AMZN	AMGN	ADI	...	TCOM	ULTA	VRSN	VRSK	VRTX	WBA	WD
ATVI	1.00000	0.39994	0.36538	0.22316	0.21628	0.43310	0.42678	0.46708	0.20396	0.32935	...	0.32291	0.12824	0.46485	0.31655	0.25968	0.21815	0.311
ADBE	0.39994	1.00000	0.45285	0.36893	0.36337	0.55212	0.54040	0.59824	0.29198	0.47382	...	0.36039	0.20115	0.71134	0.54124	0.40217	0.22811	0.650
AMD	0.36538	0.45285	1.00000	0.30183	0.34425	0.41886	0.41725	0.54930	0.15145	0.50373	...	0.33278	0.21062	0.49834	0.33090	0.27298	0.28195	0.407
ALXN	0.22316	0.36893	0.30183	1.00000	0.33243	0.31599	0.30770	0.36317	0.34202	0.31704	...	0.25714	0.40894	0.35058	0.19149	0.52242	0.19272	0.416
ALGN	0.21628	0.36337	0.34425	0.33243	1.00000	0.24875	0.25032	0.39928	0.26460	0.32828	...	0.17596	0.12856	0.36089	0.25185	0.33498	0.21959	0.308

Stock Selection:

Decision Variables:

$$\max_{x,y} \sum_{i=1}^{n} \sum_{j=1}^{n} \rho_{ij} x_{ij}$$

$$s.t. \sum_{j=1}^{n} y_j = m.$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad for\ i = 1,2,\dots,n$$

$$x_{ij} \le y_j \qquad for\ i,j = 1,2,\dots,n$$

$$x_{ij}, y_j \in \{0,1\}$$

- The binary decision variables for stock selection, yj indicates which stocks j from the index are present in the fund (yj = 1 if j is selected in the fund, 0 otherwise).
- For each stock in the index, i = 1, . . . , n, the binary decision variable xij indicates which stock j in the index is the best representative of stock i (xij = 1 if stock j in index is the most similar stock i, 0 otherwise).

- Count of Decision variables: Xij, yj is 100*100 + 100 = 10,100
- Our goal in constructing portfolio for fund is to make stocks in index and their representatives in the fund as similar as possible.

Objective matrix:

Objective of the model is to maximizes the similarity between the n stocks and their representatives in the fund.

Constraint matrix:

We have three constraints in the ***integer program***:

- The first constraint selects exactly m stocks to be held in the fund (1 row)
- The second constraint imposes that each stock i has exactly one representative stock j in the index (n rows)
- The third constraint guarantees that stock i is best represented by stock j only if j is in the fund (n*n rows)

**n is the dataset's total number of stocks*

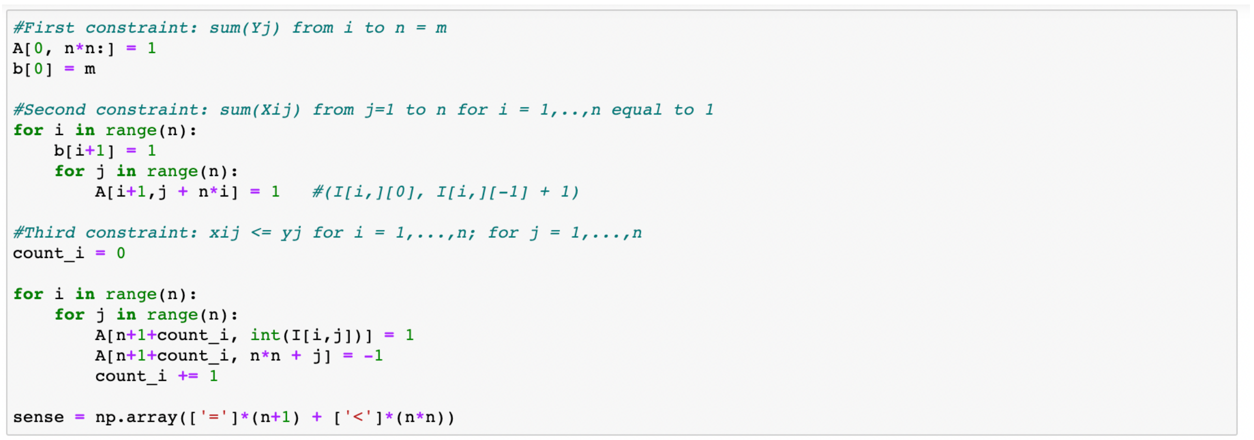
For m=5, the best stocks to be included in portfolio are: Liberty Global(LBTYK), Maxim Integrated Products (MXIM), Microsoft (MSFT), Vertex Pharmaceuticals (VRTX), Xcel Energy (XEL)

Below is the code snippet used for stock selection which uses Integer programming:

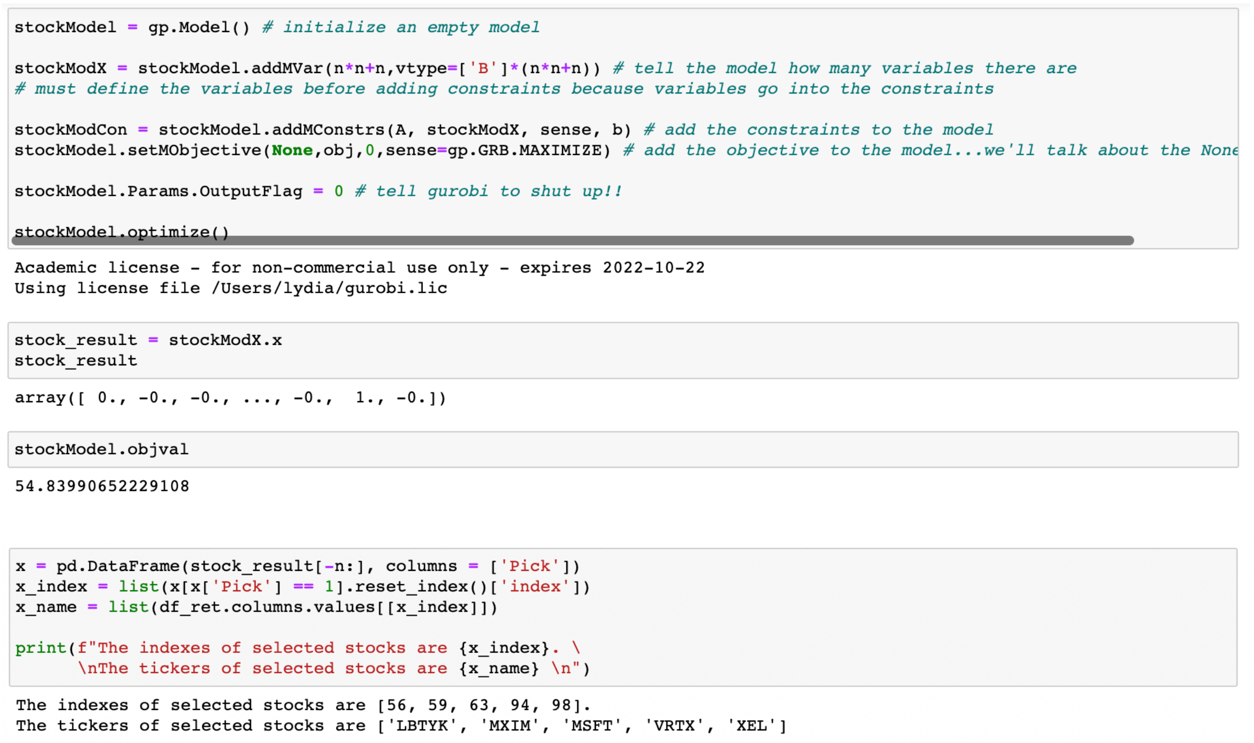
```
n = df_ret.shape[1] #number of stocks - 100
m = 5 #number of selected stocks
# n_variables = n*n (Xij)+n Yj

list_1 = list(df_ret.corr().unstack().values)
list_2 = [0]*n
obj_value = list_1 + list_2
obj = np.array(obj_value)
# obj[0,:n*n] = list(df_ret.corr().unstack().values)

obj
```

```
array([1.        , 0.39993857, 0.36537639, ..., 0.        , 0.        ,
       0.        ])
```

```
#To construct A matrix, we have 100*100 + 100 + 1 constrains and 100*100 + 100 variables
A = np.zeros((n*n + n + 1, n*n + n))

#To better represent the index ij and mapping, we construct an index matrix
I = np.ones((n,n))
for i in range(n):
    I[i] = list(range(n*i, n*(i+1)))

#For matrix b, its dimension would be (row of A) * 1
b = np.zeros((n*n + n + 1,1))
```



```
#First constraint: sum(Yj) from i to n = m
A[0, n*n:] = 1
b[0] = m

#Second constraint: sum(Xij) from j=1 to n for i = 1,..,n equal to 1
for i in range(n):
    b[i+1] = 1
    for j in range(n):
        A[i+1,j + n*i] = 1   #(I[i,j][0], I[i,j][-1] + 1)

#Third constraint: xij <= yj for i = 1,...,n; for j = 1,...,n
count_i = 0

for i in range(n):
    for j in range(n):
        A[n+1+count_i, int(I[i,j])] = 1
        A[n+1+count_i, n*n + j] = -1
        count_i += 1

sense = np.array(['=']*(n+1) + ['<']*(n*n))
```



```
stockModel = gp.Model() # initialize an empty model

stockModX = stockModel.addMVar(n*n+n,vtype=['B']*(n*n+n)) # tell the model how many variables there are
# must define the variables before adding constraints because variables go into the constraints

stockModCon = stockModel.addMConstrs(A, stockModX, sense, b) # add the constraints to the model
stockModel.setMObjective(None,obj,0,sense=gp.GRB.MAXIMIZE) # add the objective to the model...we'll talk about the None

stockModel.Params.OutputFlag = 0 # tell gurobi to shut up!!

stockModel.optimize()
```

```
Academic license - for non-commercial use only - expires 2022-10-22
Using license file /Users/lydia/gurobi.lic
```

```
stock_result = stockModX.x
stock_result
```

```
array([ 0., -0., -0., ..., -0.,  1., -0.])
```

```
stockModel.objval
```

```
54.83990652229108
```

```
x = pd.DataFrame(stock_result[-n:], columns = ['Pick'])
x_index = list(x[x['Pick'] == 1].reset_index()['index'])
x_name = list(df_ret.columns.values[[x_index]])

print(f"The indexes of selected stocks are {x_index}. \
      \nThe tickers of selected stocks are {x_name} \n")
```

```
The indexes of selected stocks are [56, 59, 63, 94, 98].
The tickers of selected stocks are ['LBTYK', 'MXIM', 'MSFT', 'VRTX', 'XEL']
```

Calculating Portfolio Weights :

To get the portfolio weights we will try to match the returns of the index as closely as possible. We would try to minimize the sum of absolute differences between the return of index and the portfolio to match the index returns closely daily.

To convert the non-linear weights program to a linear one, we use below methodology to reformulate this optimization problem as a linear program.

Let absolute differences between the return of index and the portfolio be zi.

$$zi = |qt - \sum_{i=1}^{m} wirit|$$

$$\min_{w} \sum_{t=1}^{T} \left| q_t - \sum_{i=1}^{m} w_i r_{it} \right|$$

$$s.t. \sum_{i=1}^{m} w_i = 1$$

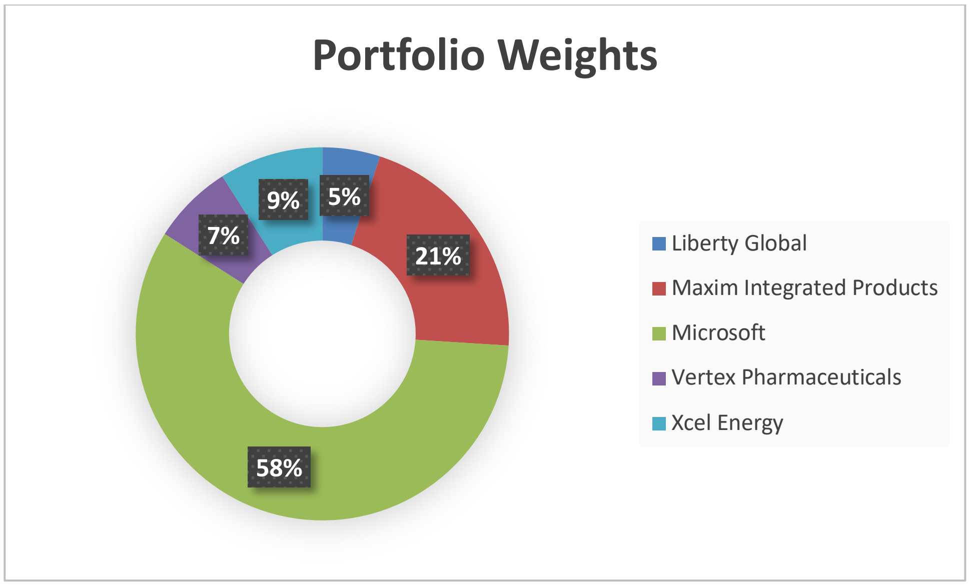
$$w_i \geq 0.$$

Objective: To minimize $z_1 + z_2 + \ldots\ldots + z_t$

Constraints:

1. z_i >= (q_t - sum of w_i * r_{it}) and zi >= - (q_t - sum of w_i * r_{it}) which corresponds to 2*t rows, t = number of days)
2. Sum of weights of all stocks in fund adds up to 1 (1 row)

Weightages of best stocks selected are shown in the below pie chart: (when m = 5)



Below is the code snippet used for calculating portfolio weights which uses Linear programming:

```
#Obj to minimize z_1 + z_2 + …… + z_T,
#fist m (5) variables are wi, the next t variables are zi, in total there are m+t variables
obj2 = np.array([0]*m + [1]*t)

#To construct A matrix, we have 1 + 2t constrains and m+t variables
A2 = np.zeros((1+2*t, m+t))
b2 = np.zeros((1+2*t,1))
sense2 = np.array(['=']*1 + ['>']*(2*t))

# constraint 1, all weights sum up to 1. 1 row in total
A2[0, :m] = 1
b2[0] = 1

# constraint 2, each z_i is corresponded to two constraints, convert absolute value
# 2*t rows in total
A2[1:1+t,m:] = np.diag([1]*t)
A2[1+t:,m:] = np.diag([1]*t)
A2[1:1+t,:m] = x_return
A2[1+t:,:m] = -x_return
b2[1:1+t] = ndx_ret
b2[1+t:] = -ndx_ret
```

```
stockModel2 = gp.Model() # initialize an empty model

stockModX2 = stockModel2.addMVar(m+t) # tell the model how many variables there are

stockModCon2 = stockModel2.addMConstrs(A2, stockModX2, sense2, b2) # add the constraints to the model
stockModel2.setMObjective(None,obj2,0,sense=gp.GRB.MINIMIZE) # add the objective to the model...we'll talk about the No

stockModel2.Params.OutputFlag = 0 # tell gurobi to shut up!!
stockModel2.optimize()
```

```
print(f"The minimum sum of absolute tracking return difference is {stockModel2.objval:.5f}.")
for i in range(m):
    print(f"The weight for {x_name[i]} is {stock_result2[i]:.5f}")
```

```
The minimum sum of absolute tracking return difference is 0.78918.
The weight for LBTYK is 0.04886
The weight for MXIM is 0.21039
The weight for MSFT is 0.58035
The weight for VRTX is 0.07119
The weight for XEL is 0.08921
```

Performance evaluation using 2020 data:

```
ndx2 = df2[['NDX']]

ndx_ret2 = ndx2 / ndx2.shift(1) - 1
ndx_ret2 = ndx_ret2.iloc[1:,:]

x_return2 = df_ret2.iloc[:, x_index]
x_return_weighted_2 = x_return2 * stock_result2
x_return_weighted_2['Index_Return_Predicted'] = x_return_weighted_2.sum(axis=1)
x_return_weighted_2['Index_Return'] = list(ndx_ret2['NDX'])
x_return_weighted_2['Abs_Difference'] = abs(x_return_weighted_2['Index_Return'] - \
                                            x_return_weighted_2['Index_Return_Predicted'])
x_return_weighted_2['Index'] = list(df2['NDX'][1:])
x_return_weighted_2['Index_Predicted'] = (x_return_weighted_2['Index_Return_Predicted'] + 1).cumprod() * df2['NDX'][0]

x_return_weighted_2.head()
```

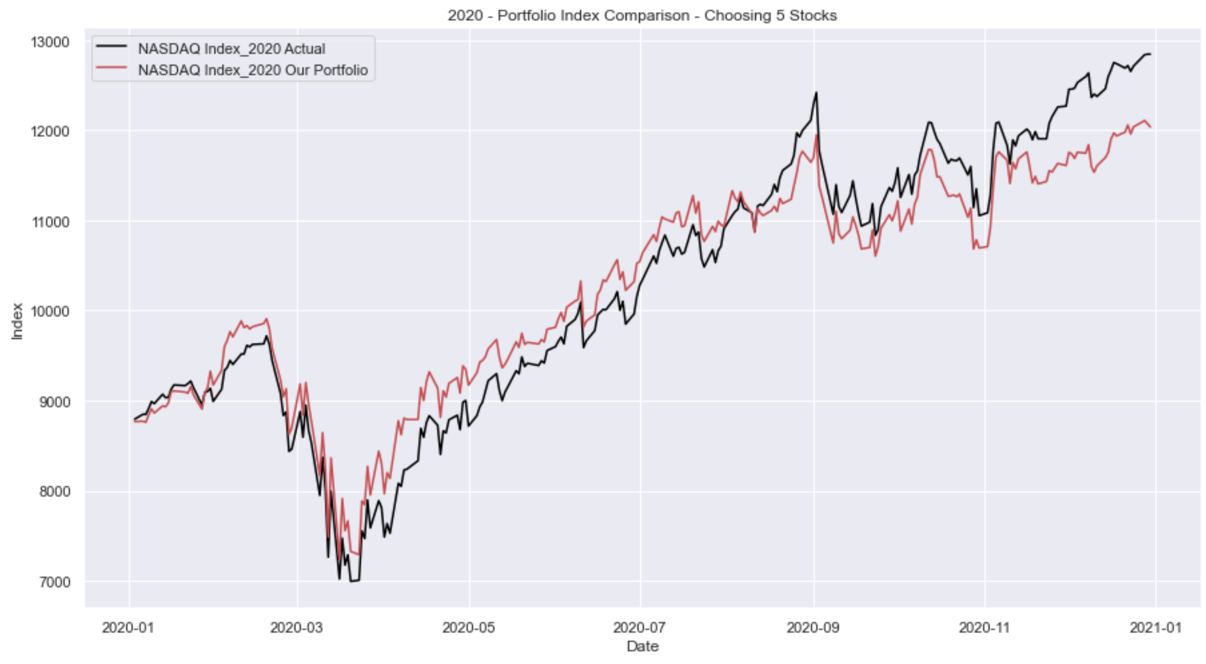
	LBTYK	MXIM	MSFT	VRTX	XEL	Index_Return_Predicted	Index_Return	Abs_Difference	Index	Index_Predicted
date										
2020-01-03	-0.00079	-0.00364	-0.00723	-0.00048	0.00043	-0.01171	-0.00883	0.00288	8793.90039	8768.36516
2020-01-06	0.00074	-0.00381	0.00150	0.00198	-0.00013	0.00028	0.00621	0.00593	8848.51953	8770.79591
2020-01-07	-0.00039	0.00475	-0.00529	-0.00008	-0.00019	-0.00119	-0.00023	0.00095	8846.45019	8760.37251
2020-01-08	-0.00212	-0.00007	0.00924	0.00232	-0.00009	0.00929	0.00745	0.00184	8912.37012	8841.74633
2020-01-09	-0.00119	0.00154	0.00725	-0.00026	0.00020	0.00754	0.00867	0.00113	8989.62988	8908.40109

```
print(f"The sum of absolute tracking return difference for 2020 is {x_return_weighted_2.sum(axis=0)[7]:.5f}.")
```

```
The sum of absolute tracking return difference for 2020 is 1.11244.
```

Portfolio Performance in 2020:

Fund constructed with 2019 data is closely matching with NASDAQ-100 index in 2020. The sum of absolute tracking return difference for 2020 is 1.11. Till August 2020 our portfolio was outperforming Index and from September onwards, fund's returns are lesser than the actual index. However, the overall changes and patterns are similar when comparing our portfolio and the NASDAQ index performance in 2020.



Stock Selection With Incremental M Values:

Next, we tested which stocks were optimal to select with M values between 10 and 100, with 10 stock increments. To do so, we created a function to replicate the ***integer program*** formulated above. The only change is in the first constraint, which will select M stocks based on the user input of M.

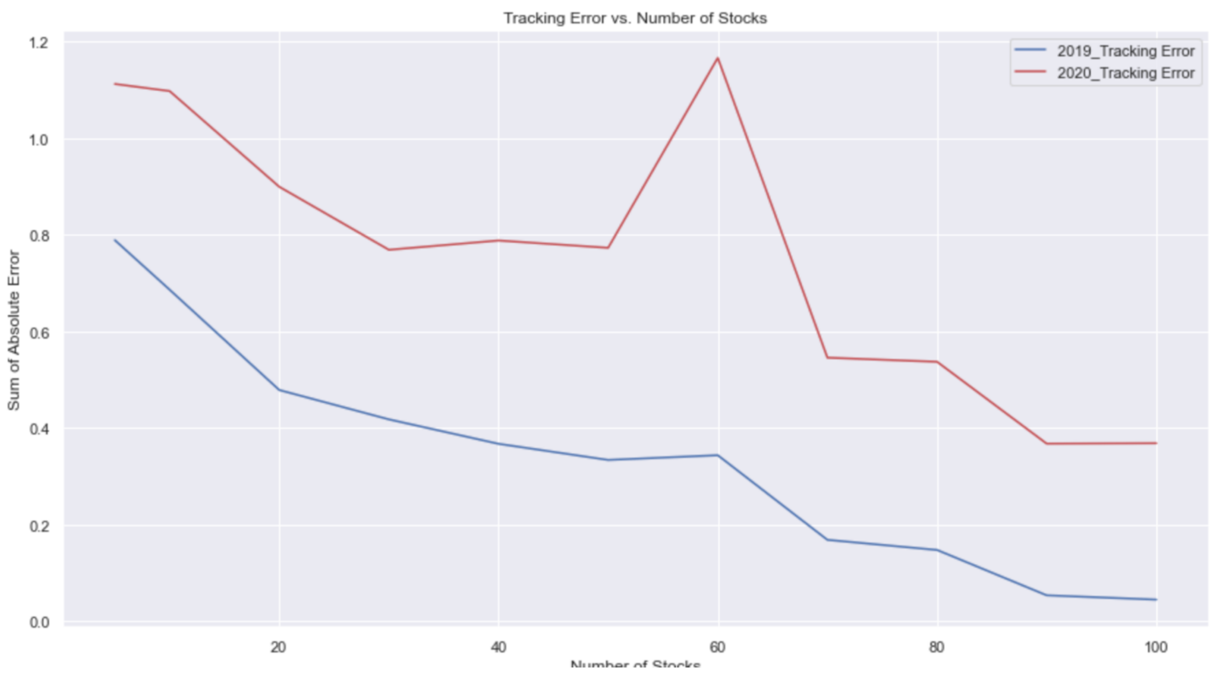
This function then solves a ***linear program*** to determine the optimal weights of each stock so that it can return the 2019 and 2020 tracking error for evaluation of its performance.

The code is a combination of the first two parts which have been combined into a function.

Below is table of the resulting errors at each value of M:

	Number of Stocks	2019_Tracking Error	2020_Tracking Error
0	5	0.78918	1.11244
1	10	0.68653	1.09771
2	20	0.47884	0.89960
3	30	0.41801	0.76911
4	40	0.36744	0.78833
5	50	0.33401	0.77322
6	60	0.34379	1.16644
7	70	0.16859	0.54574
8	80	0.14768	0.53732
9	90	0.05378	0.36779
10	100	0.04491	0.36868

This shows that higher m selections do better overall. To better visualize this fluctuation, below is a graph of the data:



We wouldn't say that performance is monotonously growing as m increases because there is some fluctuation, but the best performance is when 100 stocks are chosen, the highest we could choose. When we compare the performance of in and out of sample data (or the two years), we can observe that they are nearly identical. The model's best performance in 2019 and 2020 was when m was 100.

This is moderately intriguing because there are 100 stocks to choose from, so the top performing model isn't weeding out any stocks as we would expect. However, choosing all stocks in the index into our fund is not the best solution since we need to limit our portfolio components (m) to an economically efficient level to prevent excessive rebalancing expenses.

To determine the exact number of component stocks to include in our portfolio, we need to use the performance metric to see which m may offer us with the smallest difference in returns compared to the market index while maintaining a reasonable level of rebalancing costs. In this situation, when m = 30, the out-of-sample data performs best in the range of 5 to 60, hence we should recommend 30 component stocks. The worst number of stocks to choose are between 40 and 60, as well as 5 to 20.

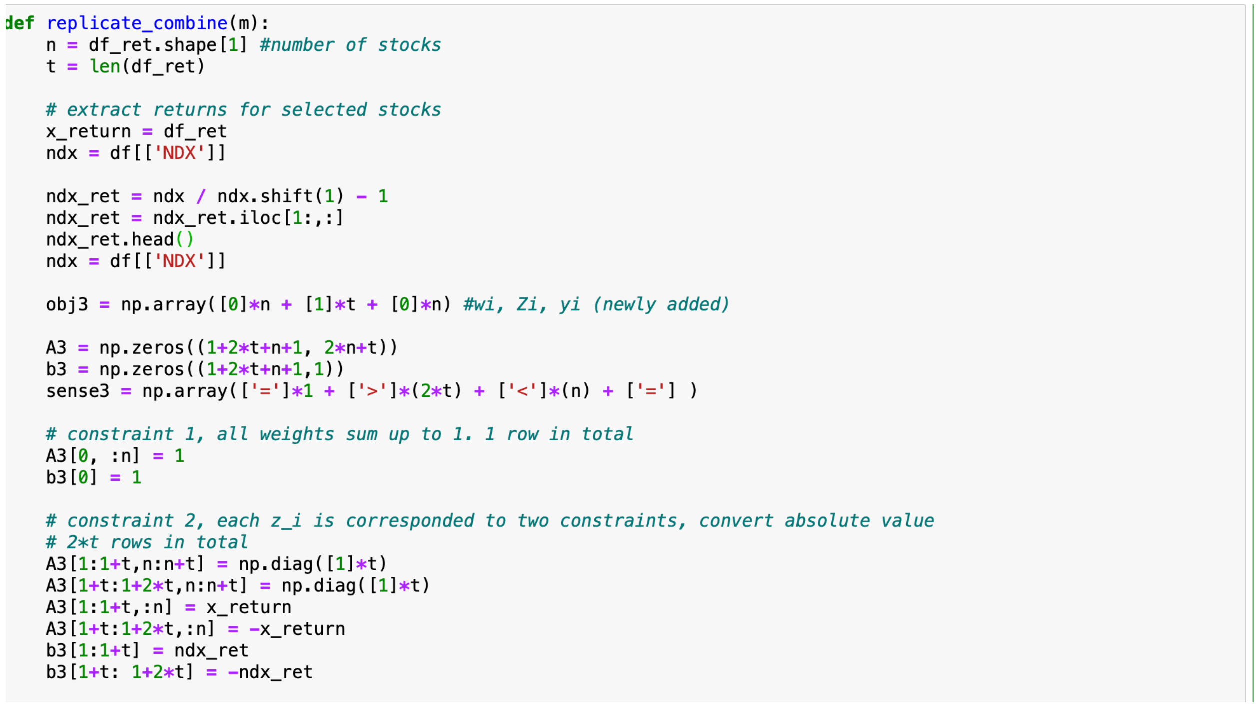
It is also interesting to note the spike in the 2020 tracking error with 60 stocks chosen. This could be because of overfitting of the data to the 2019 index or the inclusion of a significant stock that drastically fluctuated in 2020 without more stocks to offset the performance.

MIP:

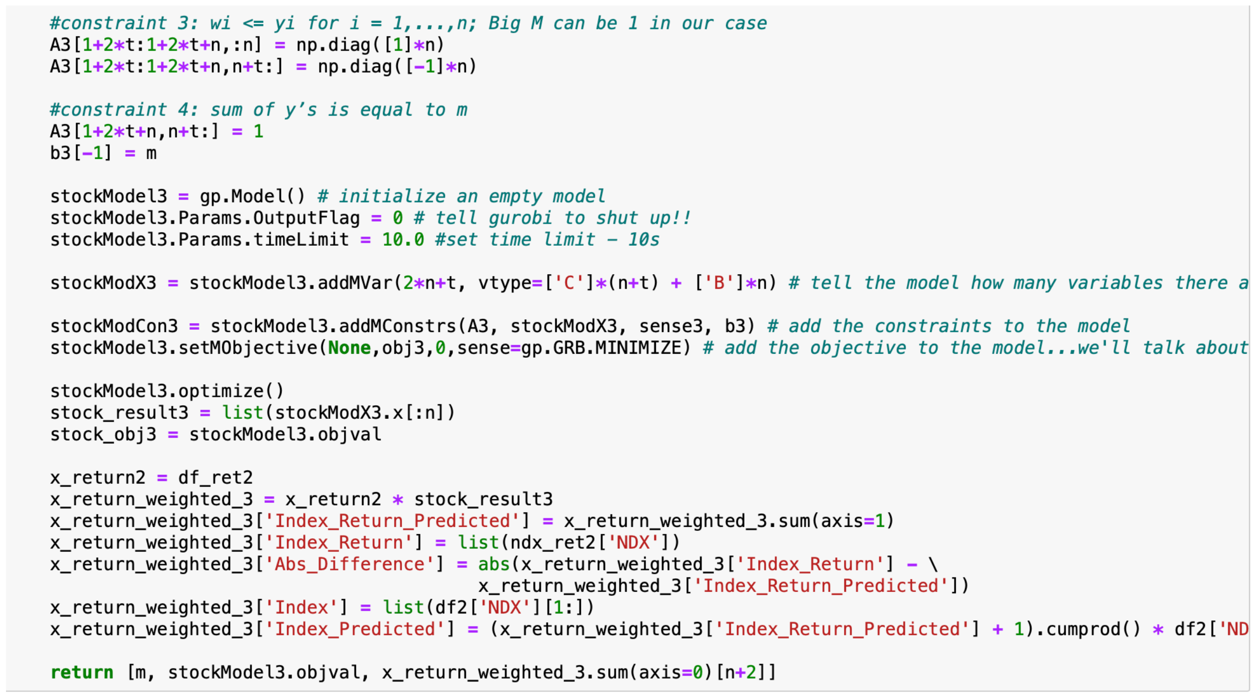
The implementation of "Big M" is the key difference in constructing the MIP. We add more limitations to the program by inserting binary and integer decision variables, which force the program to only allow weight variables with meaningful values when an on/off indication binary variable is equal to 1. This ensures the program's outputs are valid, and it also includes a range of variable types.

Essentially, if a binary variable, which stock, is chosen then an optimal amount is chosen for coordinating weight variables. In the big M constraint, $w_i \leq M * y_i$. Each weight variable wi must be less than or equal to 1, while the binary variable yi is 0 or 1. We chose M to be 1 because that is the highest value that (wi / yi) could be.

To do this, we created a function similar to the one in the previous question to test the tracking errors for different numbers of stocks selected.



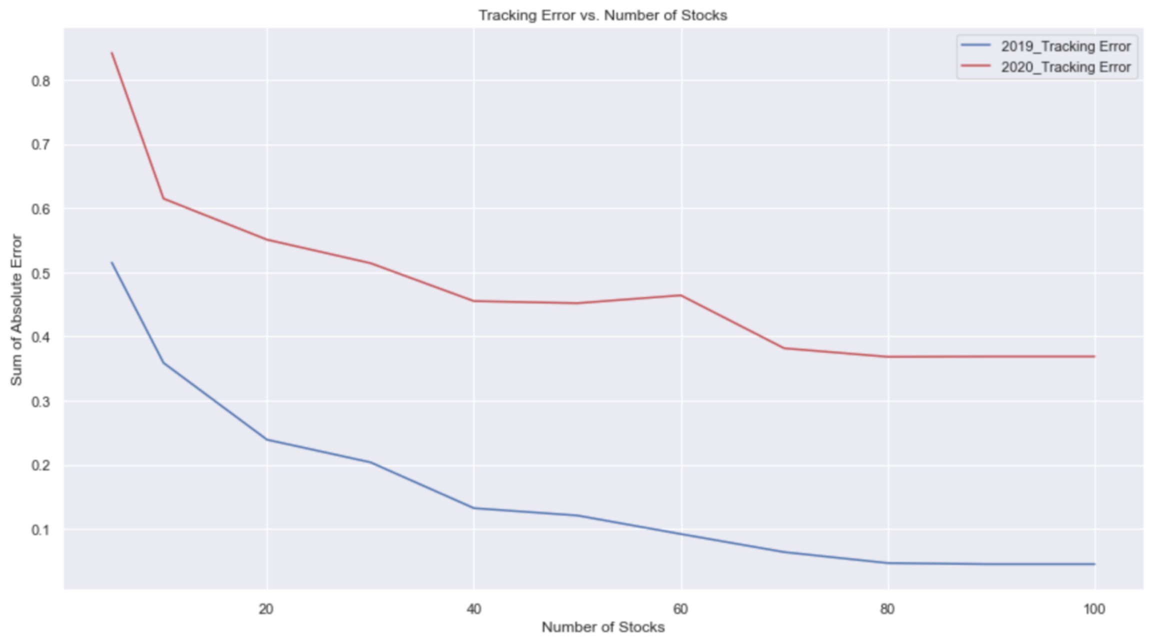
```
def replicate_combine(m):
    n = df_ret.shape[1] #number of stocks
    t = len(df_ret)

    # extract returns for selected stocks
    x_return = df_ret
    ndx = df[['NDX']]

    ndx_ret = ndx / ndx.shift(1) - 1
    ndx_ret = ndx_ret.iloc[1:,:]
    ndx_ret.head()
    ndx = df[['NDX']]

    obj3 = np.array([0]*n + [1]*t + [0]*n) #wi, Zi, yi (newly added)

    A3 = np.zeros((1+2*t+n+1, 2*n+t))
    b3 = np.zeros((1+2*t+n+1,1))
    sense3 = np.array(['=']*1 + ['>']*(2*t) + ['<']*(n) + ['='] )

    # constraint 1, all weights sum up to 1. 1 row in total
    A3[0, :n] = 1
    b3[0] = 1

    # constraint 2, each z_i is corresponded to two constraints, convert absolute value
    # 2*t rows in total
    A3[1:1+t,n:n+t] = np.diag([1]*t)
    A3[1+t:1+2*t,n:n+t] = np.diag([1]*t)
    A3[1:1+t,:n] = x_return
    A3[1+t:1+2*t,:n] = -x_return
    b3[1:1+t] = ndx_ret
    b3[1+t: 1+2*t] = -ndx_ret
```



```
    #constraint 3: wi <= yi for i = 1,...,n; Big M can be 1 in our case
    A3[1+2*t:1+2*t+n,:n] = np.diag([1]*n)
    A3[1+2*t:1+2*t+n,n+t:] = np.diag([-1]*n)

    #constraint 4: sum of y’s is equal to m
    A3[1+2*t+n,n+t:] = 1
    b3[-1] = m

    stockModel3 = gp.Model() # initialize an empty model
    stockModel3.Params.OutputFlag = 0 # tell gurobi to shut up!!
    stockModel3.Params.timeLimit = 10.0 #set time limit - 10s

    stockModX3 = stockModel3.addMVar(2*n+t, vtype=['C']*(n+t) + ['B']*n) # tell the model how many variables there a

    stockModCon3 = stockModel3.addMConstrs(A3, stockModX3, sense3, b3) # add the constraints to the model
    stockModel3.setMObjective(None,obj3,0,sense=gp.GRB.MINIMIZE) # add the objective to the model...we'll talk about

    stockModel3.optimize()
    stock_result3 = list(stockModX3.x[:n])
    stock_obj3 = stockModel3.objval

    x_return2 = df_ret2
    x_return_weighted_3 = x_return2 * stock_result3
    x_return_weighted_3['Index_Return_Predicted'] = x_return_weighted_3.sum(axis=1)
    x_return_weighted_3['Index_Return'] = list(ndx_ret2['NDX'])
    x_return_weighted_3['Abs_Difference'] = abs(x_return_weighted_3['Index_Return'] - \
                                            x_return_weighted_3['Index_Return_Predicted'])
    x_return_weighted_3['Index'] = list(df2['NDX'][1:])
    x_return_weighted_3['Index_Predicted'] = (x_return_weighted_3['Index_Return_Predicted'] + 1).cumprod() * df2['ND

    return [m, stockModel3.objval, x_return_weighted_3.sum(axis=0)[n+2]]
```

This produces the below table and graph:

	Number of Stocks	2019_Tracking Error	2020_Tracking Error
0	5	0.51999	0.85656
1	10	0.41244	0.62149
2	20	0.27938	0.55715
3	30	0.22089	0.54764
4	40	0.12779	0.44433
5	50	0.14836	0.55619
6	60	0.08253	0.39229
7	70	0.04913	0.36239
8	80	0.04664	0.35827
9	90	0.04491	0.36868
10	100	0.04491	0.36868



We set the time limit for each loop to one hour, and the function stopped and discovered the optimal solution when m=80, 90, and 100 within the time limit. Perhaps we might conclude that the answer for m=10~70 is not ideal because it should take longer to obtain the optimal result, but we just ran for an hour. Besides, no significant changes were found between the findings obtained between 10 seconds and 1 hour for each loop after our testing.

We can see that the first strategy for selecting stocks and weights is marginally worse than this approach. Between 70 and 100 stocks, there is not much improvement in the tracking error. Thus, we conclude that 60 stocks should be selected using the second approach.

Conclusion and Recommendations:

As we have mentioned above, choosing all stocks in the index into our fund (in this example m=100) is not the best solution since we need to limit our portfolio components (m) to an economically efficient level to prevent excessive rebalancing expenses. Increasing stock selection will exponentially increase our trading cost in both explicit and implicit expenses including: bid-ask spread, transaction cost, labour cost, etc.

To determine the exact number of component stocks to include in our portfolio, we need to use the performance metric to see which m may offer us with the smallest difference in returns compared to the market index while maintaining a reasonable level of rebalancing costs. In this situation, when m = 60 in the second method, the out-of-sample data performs relatively good, hence we should recommend 60 component stocks utilizing the second stock selection method.

By selecting this specific number of stocks (60) and algorithm (second approach), we can achieve sum of absolute differences as low as 0.08 in 2019 and 0.39 in 2020. The error is up a bit from the best data of the first method (when m=100), but the transaction costs have been reduced substantially. Thus this proposal is our best choice after a comprehensive evaluation.