RM 294 Optimization Course Project II

Portfolio Stock Selection and Weighting with Stochastic Control and Optimization

Group 26 Team Members:

Sahil Arora, J.T. Flume, Bret Jaco, Tanupriya Mehra

PROBLEM DESCRIPTION:

We are tasked with creating an efficient index fund that tracks a broad market without matching all the underlying assets and weights of the market index. The goal is to design a program that takes fewer positions, which in practice would save on the rebalancing costs from trading. Using an integer program we want to determine the best stock choices that are representative of the overall market index for a given number of stocks to be selected. Then we want to select the optimal weights of these stocks that minimize the difference between the portfolio returns and the returns of the index being tracked. Accomplishing this would allow us to design portfolios that best mirror the market index movement for a given number of stocks. For this, we will be using the NASDAQ-100 index.

APPROACH/METHOD:

In this section we will discuss the methodology used to solve the above problem using the following steps.

To start with, we calculated daily stock returns by applying the formula, $(P_1 - P_0) / P_0$, to each column of our 2019 stocks dataframe, which consisted of daily closing stock prices for the NASDAQ-100 index, as well as the closing prices for 100 underlying assets.

This can be seen in the code chunk below:

```
# Calculating the returns
def return_calc_function(x):
    return (x - x.shift(1))/x.shift(1)
stocks2019_returns = stocks2019_processed.apply(return_calc_function, axis = 0)
## Since no return will be calculated for day 1, Row 0 will be empty
## Dropping the NAN Rows
stocks2019_returns = stocks2019_returns.dropna(how = 'all')
print(stocks2019_returns.shape)
print(stocks2019_processed.shape)
stocks2019_returns.head()

## We can see that the first row has dropped
## Let's reset index so that we can use the general notation easily
stocks2019_returns = stocks2019_returns.reset_index().drop(['X'], axis = 1)
stocks2019_returns.head()
```

Output:

	NDX	ATVI	ADBE	AMD	ALXN	ALGN	GOOGL	GOOG	AMZN	AMGN	...	TCOM	ULTA	VRSN	VRSK
0	-0.033602	-0.035509	-0.039498	-0.094530	0.022030	-0.085791	-0.027696	-0.028484	-0.025242	-0.015216	...	-0.022834	-0.018591	-0.034989	-0.030557
1	0.044824	0.039903	0.048632	0.114370	0.057779	0.010445	0.051294	0.053786	0.050064	0.034184	...	0.058976	0.047954	0.044744	0.044147
2	0.010211	0.028196	0.013573	0.082632	0.018302	0.017192	-0.001994	-0.002167	0.034353	0.013457	...	0.022067	0.062620	0.016312	0.001000
3	0.009802	0.030309	0.014918	0.008751	0.006207	0.015954	0.008783	0.007385	0.016612	0.012824	...	0.010281	0.018450	0.036460	0.008902
4	0.007454	0.017210	0.011819	-0.026988	0.012430	0.038196	-0.003427	-0.001505	0.001714	-0.001196	...	0.023745	0.018804	-0.008157	0.003781

Next, we created a similarity matrix that displayed the correlation between the returns of every stock pair. This will help in the stock selection process for using stocks that are most similar and representative of the index we are tracking.

```
## Calculating the similarity matrix
## Let's start by dropping the NDX from the dataset because, we dont need it in the
## fund similarity
similarity = stocks2019_returns.drop(['NDX'], axis  =1).corr()
## The similarity matrix should be 100 X 100
print("The shape of similarity matirx is : {}".format(similarity.shape))
similarity.head()
```

Output:

	ATVI	ADBE	AMD	ALXN	ALGN	GOOGL	GOOG	AMZN	AMGN	ADI	...	TCOM	ULTA	VRSN	VRSK	VRTX
ATVI	1.000000	0.399939	0.365376	0.223162	0.216280	0.433097	0.426777	0.467076	0.203956	0.329355	...	0.322906	0.128241	0.464850	0.316549	0.259679
ADBE	0.399939	1.000000	0.452848	0.368928	0.363370	0.552125	0.540404	0.598237	0.291978	0.473815	...	0.360392	0.201151	0.711339	0.541243	0.402171
AMD	0.365376	0.452848	1.000000	0.301831	0.344252	0.418861	0.417254	0.549302	0.151452	0.503733	...	0.332776	0.210623	0.498342	0.330900	0.272983
ALXN	0.223162	0.368928	0.301831	1.000000	0.332433	0.315993	0.307698	0.363170	0.342022	0.317040	...	0.257143	0.408936	0.350581	0.191489	0.522423
ALGN	0.216280	0.363370	0.344252	0.332433	1.000000	0.248747	0.250316	0.399281	0.264599	0.328280	...	0.175957	0.128559	0.360886	0.251855	0.334978

For the stock selection process, we formulated an integer program to choose the *m* stocks that best represent the NASDAQ-100 as a whole. Our objective function consisted of 10,100 binary decision variables with the first 100 y_j corresponding to which stocks from the index are present in our fund and the next 10,000 x_{ij} corresponding to which stock j in the index is the best representative of stock i. The objective was to maximize the similarity scores of the objective function. The coefficients for the x_{ij} variables in the objective function were the respective values calculated in our similarity matrix. This can be seen in our code as the matrix flattened out into a single array to use in our objective function.

For the constraint matrix, the first constraint we implemented on our program is that the number (or sum) of the binary y_j variables had to equal exactly *m*, the number of stocks we choose to select.

For the next set of constraints we made it so that each row of the x-variable matrix (if not flattened out) must only sum up to 1, so that we only choose a singular stock with the highest similarity for each row. Making a constraint for each row added 100 constraints to the constraint matrix. Lastly, we made it so that a stock can only have values of 1 in a column in the x-variable matrix if that stock is one of the selected stocks. Thus, each y-variable corresponds to a column representing a single stock. If that y-variable is 1 (as opposed to a 0) then that column can have values of 1 for the coefficient of the x-variables in that column. Otherwise, the coefficients to that column must be zero because that stock was not selected. This is represented in the following code chunk. This added another 100 constraints to the constraint matrix, as this had to be done for each y-variable and the set of corresponding x-variables of a given y-variable. This gave a total of 201 constraints.

```
def stock_selection(m):
    m = m
    cons_selection = len(similarity)
    obj = np.array([0]*cons_selection + list(similarity_array.flatten()))
    A = np.zeros((2*cons_selection+1 ,len(obj)))
    ## First constraint
    A[0][0:cons_selection]=1
    # ## next 100 constraint
    j=1
    for i in range(1,101):
        A[i][(j*cons_selection): ((j+1)*cons_selection)] = 1
        j=j+1

    # ## Last 100 constraints
    i = 0
    for constraint_number in range(101,201):
        A[constraint_number][i] = -100 ## - selection variable y
        for k in range(100):
            A[constraint_number][(k+1)*100 + i] = 1
        i = i+1

    ## RHS
    b = np.array([m]+[1]*cons_selection+[0]*cons_selection)
    ## Sense Array
    direction = np.array(['=']+ ['=']*cons_selection + ['<']*cons_selection)

    var_num = cons_selection + cons_selection*cons_selection

    tspMod = gp.Model()
    tspMod_x = tspMod.addMVar(var_num,vtype=['B']*var_num)
    tspMod_con = tspMod.addMConstrs(A, tspMod_x, direction,b)
    tspMod.setMObjective(None,obj,0,sense=gp.GRB.MAXIMIZE)

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

    stocks_selected = list(similarity.columns[np.where(tspMod_x.x[0:cons_selection] == 1)])
    return stocks_selected
```

For the weight calculation function we constructed an objective function consisting of 250 y-variables that represented each time period (one for each day), as well as a variable for each stock selected (so an additional m number of variables). The objective function was designed to minimize the absolute value of the difference between the index returns and the summation of the weighted stock returns. Because we are using an absolute value in our objective function, which is a nonlinear function, we have to code this in a special way by adding two constraints for each time period to utilize our constraint matrix. One constraint will have our variables summed greater than or equal to 0, while the other constraint will be the negative of the first constraint. When using a particular time period data for the selected stocks we assigned a coefficient of 1 to the y-variable corresponding to that time period. Otherwise, the y-variable had a coefficient of 0, as we were not using that time period data. This was done for a total of 500 constraints (2 times the number of time periods). Finally, for the last constraint we made the sum of the weights have to be equal to 1 for a total of 501 constraints in this next constraint matrix. This can be seen in the following code chunk.

```python
def weight_calculation(m,stocks_selected):
    ## Total variables = # time entries -1 (len return) & m variables for weights
    len_y_vars = len(stocks2019_returns) ## We have 250 timeperiods
    obj_wt = np.array([1]*len_y_vars + [0]*m) ## 250 Ys and m variables for weights
    #print(len(obj_wt))
    q = stocks2019_returns.loc[:,'NDX'].to_numpy()
    r_matrix = stocks2019_returns.loc[:,stocks_selected].to_numpy()
    ## Defining and filling up the constraint matrix
    A_weight = np.zeros((len_y_vars*2+1,len(obj_wt)))

    ## Greater than constraints
    A_weight[0:len_y_vars,0:len_y_vars] = np.identity(len_y_vars)
    A_weight[0:len_y_vars, len_y_vars:len_y_vars+m] = r_matrix

    ## Less than constraints
    A_weight[len_y_vars:len(A_weight)-1,0:len_y_vars] = np.identity(len_y_vars)
    A_weight[len_y_vars:len(A_weight)-1, len_y_vars:len_y_vars+m] = r_matrix*-1

    ## Weight Constraint
    A_weight[-1,len_y_vars:len_y_vars+m] = 1

    # ## RHS
    b_weight = np.append(np.append(q , -1*q),np.array([1]))

    ## Sense Aqrray
    direction_weight = np.array(['>']*len_y_vars*2 + ['=']*1)

    ## Optimization
    var_num = len(obj_wt)
    WeightMod = gp.Model()
    WeightMod_x = WeightMod.addMVar(var_num,vtype=['C']*var_num)
    WeightMod_con = WeightMod.addMConstrs(A_weight, WeightMod_x, direction_weight,b_weight)
    WeightMod.setMObjective(None,obj_wt,0,sense=gp.GRB.MINIMIZE)

    WeightMod.Params.OutputFlag = 0 # tell gurobi to shut up!!
    WeightMod.optimize()
    Index_Fund_Weights = WeightMod_x.x[-1*m:]

    return(Index_Fund_Weights)

```

SOLUTIONS:

Question 1:

Once we had defined functions for obtaining m stocks for creating the portfolio and their respective weights, we also defined functions to evaluate the performance of the stocks by calculating their returns and their total absolute deviation from the returns of the NASDAQ fund.

```
def performanceEvaluation(returns_matrix, stocks_selected,Index_Fund_Weights):
    weighted_returns = np.array(returns_matrix.loc[:,stocks_selected]) * Index_Fund_Weights
    weighted_returns = weighted_returns.sum(axis = 1)
    ## Calculating the absolute deviations from the NASDAQ
    TotalAbsoluteDeviation = np.sum(np.abs(returns_matrix.iloc[:,0].to_numpy().flatten() - weighted_returns))
    return TotalAbsoluteDeviation

def weighted_returns(returns_matrix, stocks_selected,Index_Fund_Weights):
    weighted_returns = np.array(returns_matrix.loc[:,stocks_selected]) * Index_Fund_Weights
    weighted_returns = weighted_returns.sum(axis = 1)
    return weighted_returns
```

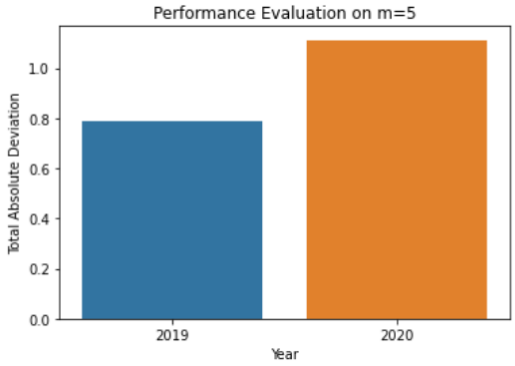
Using these functions we calculated the specified metrics for both 2019 and 2020.

Question 2:

Using all the functions defined above, we took m = 5 and obtained the 5 best stocks to include in our portfolio and their corresponding weights.

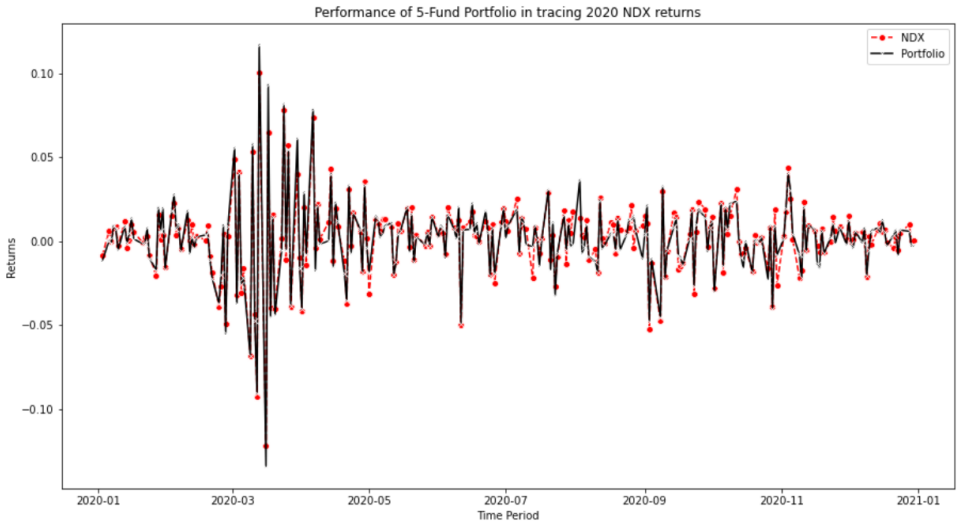
The best stock to include in the portfolio, with the highest weight of 0.580 is MSFT. The others include MXIM, XEL, VRTX, LBTYK with weights 0.210, 0.089, 0.071, and 0.049 respectively. These numbers are obtained using the 2019 data. The next important question to answer was: How well does this portfolio of 5 stocks track the index fund in 2020?

To answer this, we calculated the total absolute deviation of the returns of the portfolio with the returns of NASDAQ in both 2019 and 2020.



From the above graph, the Total Absolute Deviation from NASDAQ in 2019 is 0.789 and in 2020 is 1.112.

Based on the portfolio made using 2019 stocks data, in 2020, the Total Absolute Deviation of the portfolio from NASDAQ is 0.323 units higher. This implies that the portfolio returns in 2020 are more widely dispersed from the actual value of the NASDAQ returns. This problem most likely arises because a portfolio of just 5 stocks is not large enough to accurately track the index fund. This idea is backed up by the fact that even in 2019, the deviations from NASDAQ are relatively high. Therefore, increasing the number of stocks in the portfolio is more likely to closely mimic the returns of NASDAQ-100.

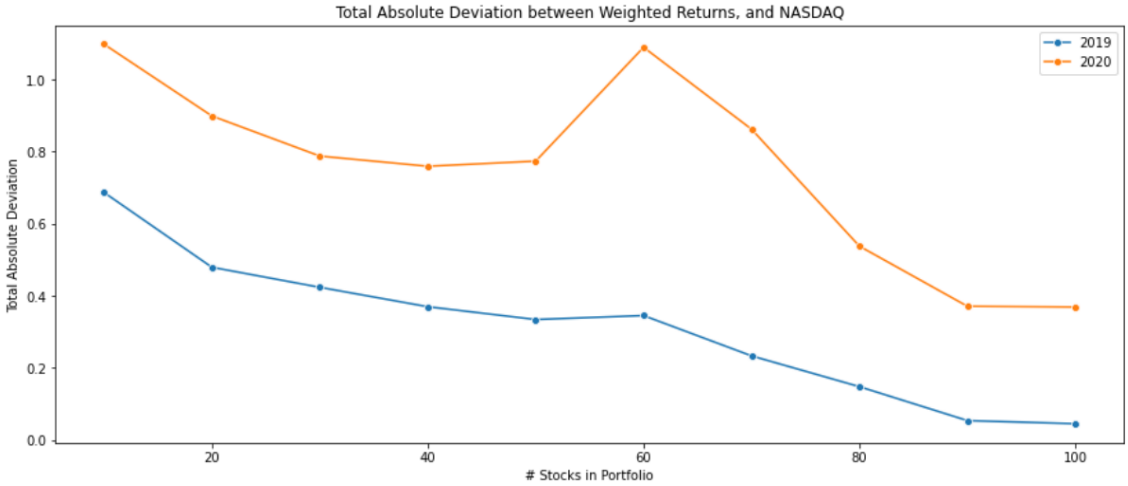


For a closer look at the month on month tracking of the index fund, we made the above time series plot.

From the above graph, we can see that the portfolio is relatively accurate in tracking the index fund in 2020. Especially when there are large fluctuations in the returns. For example, between the time periods 2020-03 and 2020-05. The biggest discrepancies are seen at 2020-5, between 2020-07 and 2020-08, and finally near 2020-11, when the black line (portfolio) is clearly not overlapping with the red line (NDX).

Question 3:

Our next step was to redo the above analysis with a higher number of stocks in our portfolio (with m = 10,20,....,90,100).



We can see that for both 2019 and 2020, the total absolute deviation is consistently decreasing till 50 stocks in the portfolio. This is exactly what we expected when we were evaluating results for the 5 stock portfolio, i.e. the portfolio will get better at tracking the fund as we increase the number of stocks in our portfolio.

At m = 60, we see diminishing returns starting to set. This change is much more evident in the 2020 graph (due to the spike at # Stocks in Portfolio = 60) . This is because the modeling was done using weights from 2019.

After 60 stocks, the deviation from the index fund starts to decrease again, and finally gets stable in 2020 at the number of stocks in the portfolio equal to 90 and 100. Therefore, 90 stocks is optimal for the portfolio as that has the least deviation from the index fund (since 100 stocks implies that all stocks in NASDAQ are also included in our portfolio).

Question 4:

The final section of our evaluation was to redo this problem with a different approach. Instead of formulating the problem as an integer programming followed by linear programming, we formulated it as a mixed-integer programming problem.

Our objective function consisted of 450 decision variables with the first 250 variables corresponding to the number of days for which the returns were calculated. Since we are optimizing over all weights in this approach, we used the next 100 variables as the weights corresponding to each stock and the last 100 variables as binary variables indicating whether the stock has been selected to be a part of the index fund or not.

```
len_y_vars = len(stocks2019_returns) ## We have 250 timeperiods
weight_variables = stocks2019_returns.shape[1]-1
stock_selector_binary_var = stocks2019_returns.shape[1]-1

obj_wt = np.array([1]*len_y_vars + [0]*weight_variables+
                  [0]*stock_selector_binary_var )
```

Our objective was to minimize the total absolute difference between returns on NASDAQ and weighted average returns for the stocks selected in the index fund.

For the constraint matrix, we formulated a total of 602 constraints.Similar to the weight calculation function, we formulated two constraints for each time period which contributed to the first 500 constraints. We used the 'big M' constraint to establish a relationship between binary indicator variables y_1, y_2, ...y_{100} and weight variables w_1, w_2,...w_{100}. We added the next 100 constraints to ensure that the weight variable w_i is 0 if y_i is 0 and is greater than 0 only if y_i is 1. This ensures that a weight is allocated to a stock only when it is selected to be a part of the index fund. Since our weight variables have to be less than or equal to 1, we used the least possible value of 'M' as 1.

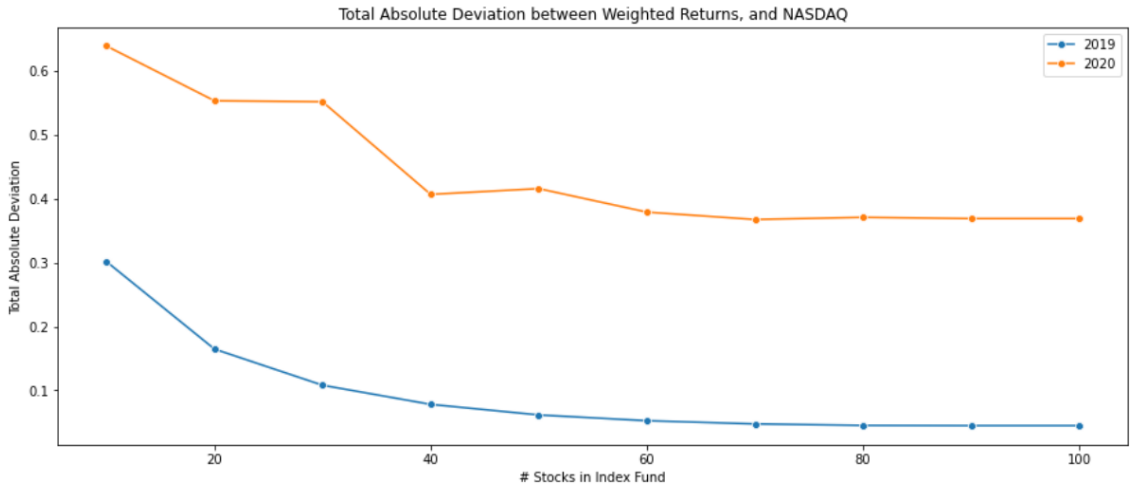
The 601^{th} constraint ensures that the sum of indicator variables, y_i, adds up to 'm', the number of stocks to select. The last constraint ensures that all the weights add up to 1.

The entire formulation is implemented within the MIP function that takes 'm' as the input

```
def MIP_Solver(m_input):
    len_y_vars = len(stocks2019_returns) ## We have 250 timeperiods
    weight_variables = stocks2019_returns.shape[1]-1
    stock_selector_binary_var = stocks2019_returns.shape[1]-1
    obj_wt = np.array([1]*len_y_vars + [0]*weight_variables+
                      [0]*stock_selector_binary_var )
    stock_returns_2019_all_funds = stocks2019_returns.drop(['NDX'], axis = 1).to_numpy()

    q = stocks2019_returns.loc[:,'NDX'].to_numpy()
    M = 1
    A_weight = np.zeros((2*len_y_vars + weight_variables+2 ,len(obj_wt)))
    ## First 250 constraints
    A_weight[0:len_y_vars,0:len_y_vars] = np.identity(len_y_vars)
    A_weight[0:len_y_vars,len_y_vars:len_y_vars+weight_variables] = stock_returns_2019_all_funds
    ## Next 250 constraints
    A_weight[len_y_vars:2*len_y_vars,0:len_y_vars] = np.identity(len_y_vars)
    A_weight[len_y_vars:2*len_y_vars,len_y_vars:len_y_vars+weight_variables] = stock_returns_201
    ## Next 100 constraints
    ## For Big M Constraints
    i = 0
    for row_number in range(2*len_y_vars, 2*len_y_vars+weight_variables):
        A_weight[row_number][len_y_vars+i]=1
        A_weight[row_number][len_y_vars+weight_variables+i]=-1*M
        i = i+1

    ## Last 1 constraint to make sure that indicator variables add up to #stocks to pick
    A_weight[-2][-1*stock_selector_binary_var:] = 1

    ## Last 1 constraint to make the weights as 1
    A_weight[-1][len_y_vars: len_y_vars+weight_variables] = 1


    m = m_input
    b_weight = np.array(list(q) + list(-1*q) + [0]*weight_variables + [m]*1 +[1]*1)
    ## Sense Aqrray
    direction_weight = np.array(['>']*len_y_vars*2 + ['<']*weight_variables +['=']*2)
```

```
## Optimization
var_num = len(obj_wt)
WeightMod = gp.Model()
WeightMod_x = WeightMod.addMVar(var_num,vtype=['C']*len_y_vars + ['C']*weight_variable
WeightMod_con = WeightMod.addMConstrs(A_weight, WeightMod_x, direction_weight,b_weigh
WeightMod.setMObjective(None,obj_wt,0,sense=gp.GRB.MINIMIZE)
WeightMod.Params.OutputFlag = 0 # tell gurobi to shut up!!
WeightMod.Params.timeLimit = gurobi_time_limit
WeightMod.optimize()

return WeightMod_x.x[len_y_vars: len_y_vars+weight_variables]
```

After formulating the problem, we ran the optimization to compare the performance of the portfolio with NASDAQ in both 2019 and 2020. We obtained the following graph as a result:



From the above graph, we can clearly see that the second method is much more accurate in tracking NASDAQ. Even at 10 stocks in the portfolio, the deviation in 2020 is near 0.65, whereas in the old method, it was over 1.

Additionally, diminishing returns are not apparent at any point in the graph. In fact, the graph starts to get stable earlier, at 60 stocks in the portfolio (both in 2019 and 2020). The final value of the deviation in 2020 is very similar to the value of deviation obtained in the previous method, but the optimal number of stocks is much less. In this method, the optimal number of stocks is 70 (because the deviation at 60 is slightly higher). The only drawback in this method is the computational time.

Recommendations:

Our final recommendation is to use the second method, i.e., mixed integer programming for stock selection. We should select 70 stocks to put into the index fund. In order to improve or build up on the current analysis, we can compare rebalancing costs with the cost of inaccurately predicting the 2020 NASDAQ returns to better evaluate the cutoff threshold for the number of funds.