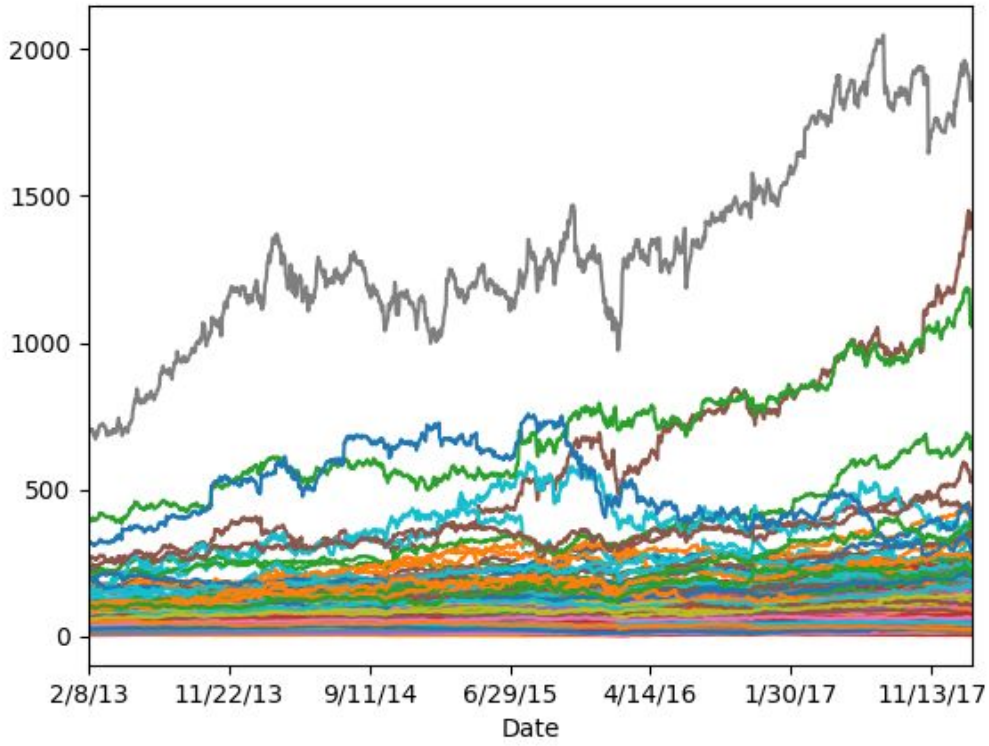
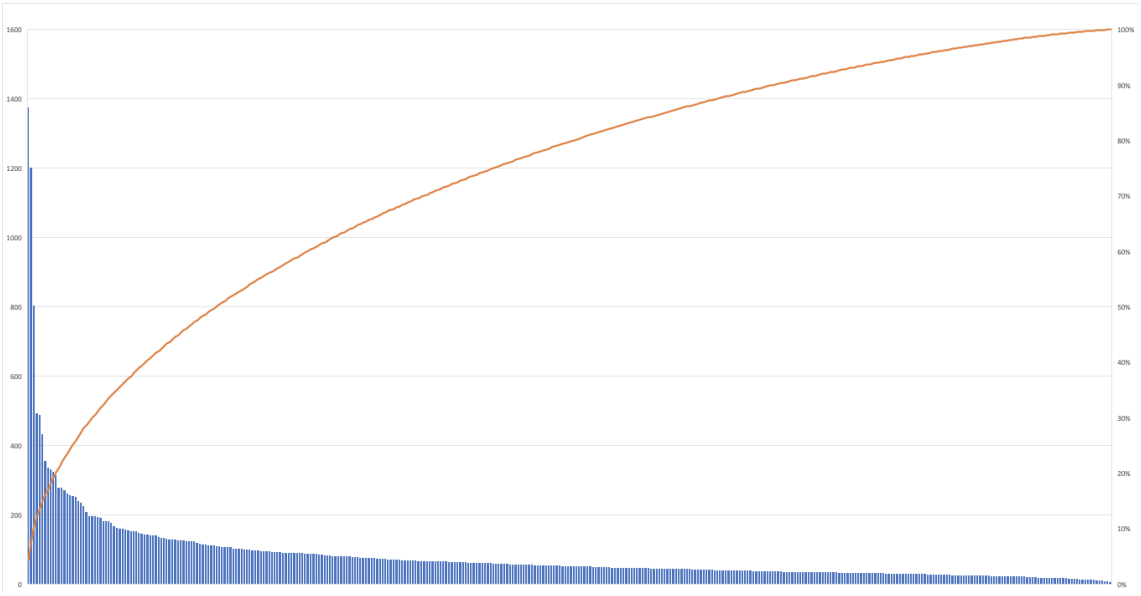
Spring 2020 IME Data Analysis Competition Report

Surya Venugopal

Table of Contents

I began by making visualizations and trying to find the stocks that gave much higher returns than the rest. I first plotted all of the stocks vs. their prices, then created a pareto chart of the stocks vs. the amount you would make if you had bought one share on the first day. However, it was unreasonable to go about this problem by looking at these trends because of how many variables there are.





From there, I decided to solve an easier problem. I randomly selected 10 stocks from the database and reduced the number of days. I then realized that I was going about this problem from the wrong perspective entirely. I reread the prompt, and saw that I had an objective function, the earnings per stock times the number of shares I buy, as well as a set of constraints. This could be solved with linear programming. However, I did not know how to deal with an objective function that changed by the day. I decided I would first solve a case where I had to sell and buy on a particular day. I defined the problem and used solver to solve for the maximum earnings. I then realized how to solve the problem: I would need to iterate through all the possible combinations of dates and solve this optimization problem for each of them.

Analytical Method:

```
def pivot_maker(csvname):
  df = pd.read_csv(io.BytesIO(uploaded[csvname]))
  df['Date'] = pd.to_datetime(df['Date']).dt.date
  pivot = pd.pivot_table(df, values = 'Price', index = 'Date', columns = 'Code')
  return pivot

piv = pivot_maker('DAComp2Data.csv')
piv
```

First I had to clean the data in order to make it easier to work with. Using the python pandas library, I am able to create a pivot table that displays the dates as row indexes and the codes as column indexes with the prices listed below, rather than three columns with all the data. Here I also made the date indexes `datetime` objects so I can compare them easily.

```
def optimization(data, buy_row_index, sell_row_index, stock_names, details = False):
  prob = LpProblem('MAXRETURNS', LpMaximize)

  sellprice = dict(zip(stock_names, data.iloc[sell_row_index].values))
  buyprice = dict(zip(stock_names, data.iloc[buy_row_index].values))

  IntShares = LpVariable.dicts('IntShares', stock_names, lowBound=0, upBound=10, cat='Integer')
  BinTicker = LpVariable.dicts('BinTicker', stock_names, cat='Binary')
  prob += lpSum(IntShares[i] * (sellprice[i] - buyprice[i]) for i in stock_names)
```

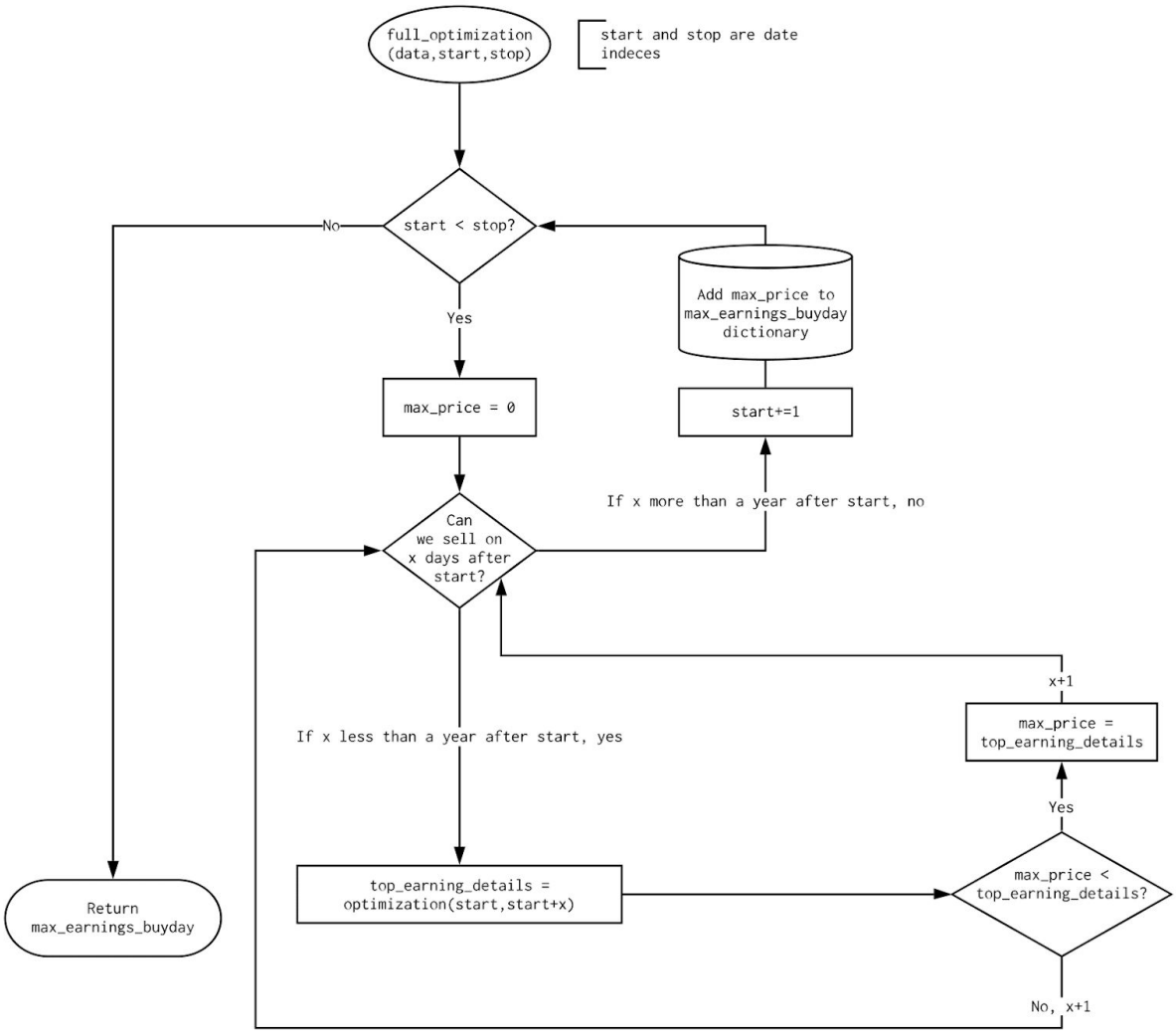
Next I defined the optimization function where you pass in the pivot table, the index of the buy row, sell row and the names of each stock. I then created two dictionary objects which contain the prices indexed by their name, which is used to create the objective function for the specific iteration. For each stock, I defined the number of shares as integer decision variables and constrained it to a maximum of 10 shares, and defined a binary decision variable for whether I buy the particular stock or not.

```
#Subject To:
prob += lpSum([IntShares[i] * buyprice[i] for i in stock_names]) <= 1500, 'Budget (1500)'
prob += lpSum([BinTicker[i] for i in stock_names]) >= 5, 'Min Tickers (5)'
for i in stock_names:
  prob += (IntShares[i] - ((10 * BinTicker[i]))) <= 0
  prob += (IntShares[i] - (.5 * BinTicker[i])) >= 0

prob.solve()

if details == True:
  details = []
  for v in prob.variables():
    if v.varValue > 0:
        details.append({v.name: v.varValue})
  details.append({'Maximum Earnings': value(prob.objective)})
  return details

print(str(data.index[sell_row_index]), value(prob.objective))
return (str(data.index[sell_row_index]), value(prob.objective))
```

Then I added the constraints. I defined the budget constraint as the buying price times the number of shares I opt to buy `<= $1500` and the minimum tickers constraint as the sum of the binary variables `<= 5`. I then wrote a for loop which creates a constraint for each pair of binary and integer variables, which prevents situations I don't want to occur. The first line prevents opting to not buy the stock `(BinTicker[i] = 0)` but still buying shares `(IntShares[i] > 0)`, while the second line prevents opting to buy the stock `(BinTicker[i] = 1)` but not buying any shares `(IntShares[i] = 0)`. Once I have determined the solution at the very end, I can set `details = True` which will return the exact Stock portfolio to enter into the Investment Outline. For now, the function returns the sell date passed in as well as the maximized value for that day.

```
def full_optimization(data, start, stop):
  max_earnings_buyday = {}
  labels = data.columns

  #Larger loop that iterates through all buy dates
  for buyidx in range(start, stop):
    buydate = data.index[buyidx]
    buy_prices = data.iloc[buyidx].values

    max_price = 0
    sell_day = ""

    #Smaller loop that iterates through all possible sell dates given the buy date
    for sellidx in range(buyidx + 1, piv.shape[0]):

      selldate = data.index[sellidx]

      less_than_a_year = relativedelta(selldate, buydate).years
      exactly_one_year = relativedelta(selldate, buydate).years + relativedelta(selldate, buydate).days
      if less_than_a_year < 1 or exactly_one year == 1:
        top_earning_details = optimization(data, buyidx, sellidx, labels)

        if max_price < top_earning_details[1]:
          max_price = top_earning_details[1]
          sell_day = top_earning_details[0]
      else:
        break

    max_earnings_buyday[str(data.index[buyidx]) + " TO " + sell_day] = max_price

  return max_earnings_buyday
```

Here I defined a function that will iterate through every possible combination of dates. I passed in the pivot table, and the indexes of the range I want to search through. I first instantiated `max_earnings_buyday`, which is a dictionary that will store the maximum earnings for each buy date. The outer loop iterates through all the buy dates passed in and instantiates placeholders meant to update when I calculate a sell date with a higher maximum earning than one I have found before. The inner loop iterates through the dates after the buy date. I can now write the final constraint, that the sell date must be within a year of the buy date. I check that using `relativedelta`, where `.years` will return 0 if the sell date is within a year, exclusive. A second condition checks the special case where the buy and sell date are exactly a year apart. If one of these is met, I run the first optimization function with the buy date and sell date. If I find a max earnings larger than anything I have seen for this buy date, I update our placeholders with the new maximum earnings and sell day. Once I have looped through all the possible sell dates for the buy date, I can store the best sell date and max earnings of our particular buy date in the dictionary `max_earnings_buyday,` which the function returns after looping through all the buy dates.

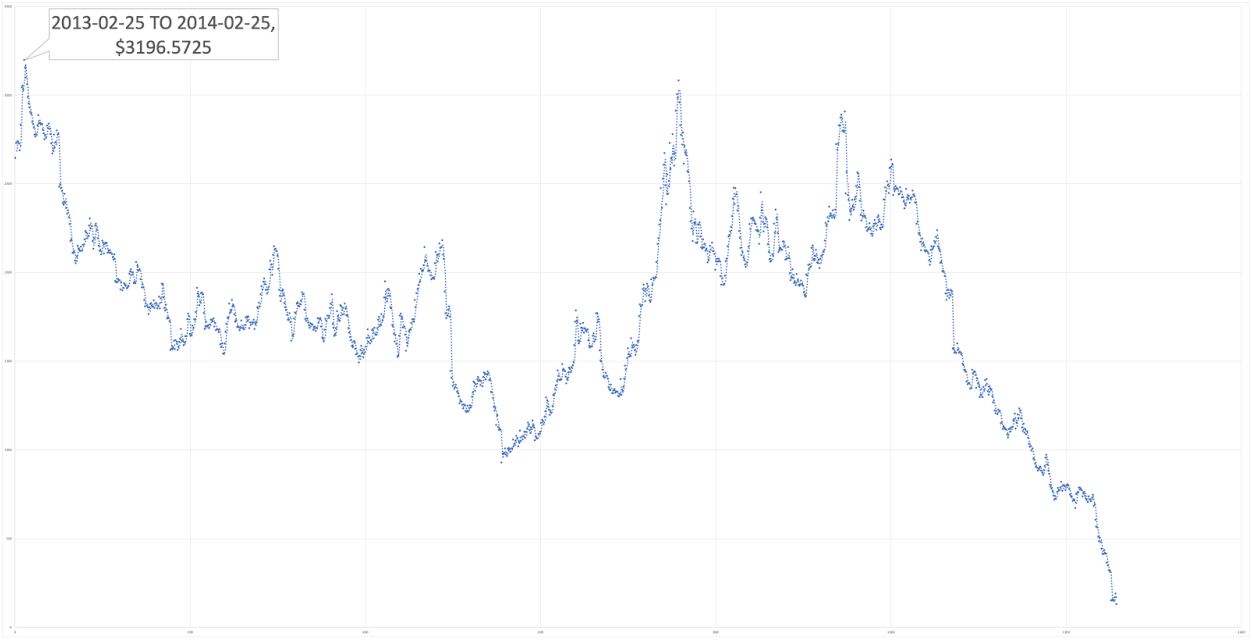


Managing Time Cost:

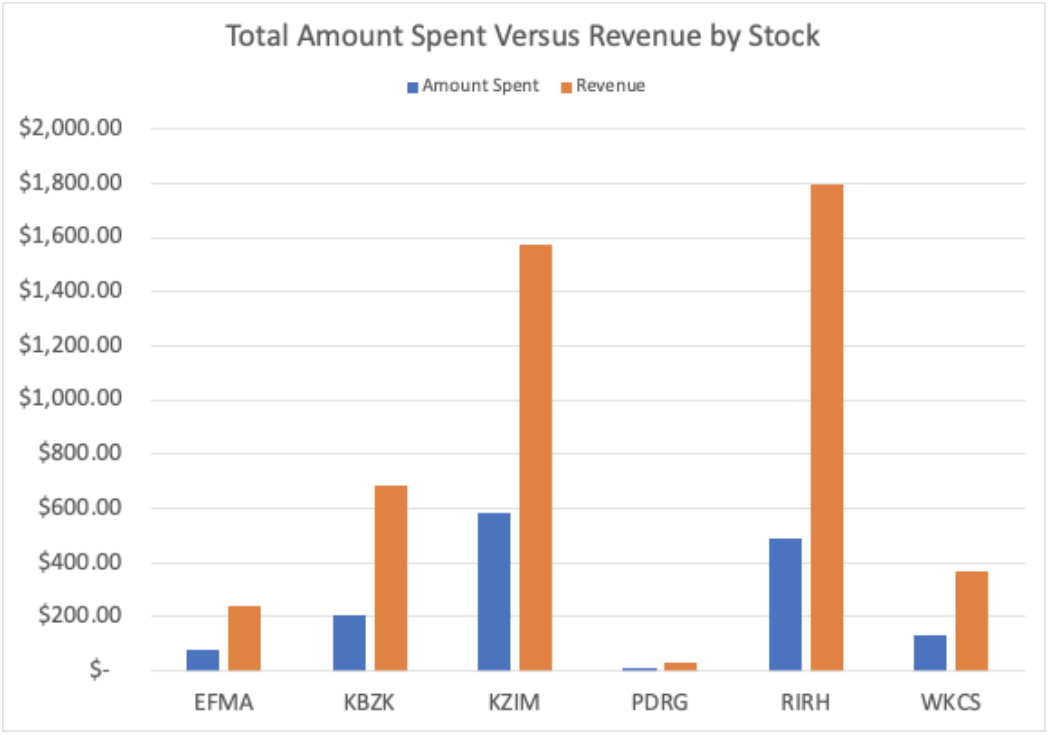
You can imagine that excel solver would not fare well doing even one iteration of this problem. For this reason, I used the PuLP linear programming package in python, whose `pulp.solve()` function computed one iteration of the optimization function per second. From there I used Google Colab which executes code on Google's cloud servers, allowing me to leverage Google's GPUs. This brought down the time it took to execute a single `pulp.solve()` to three tenths of a second. By dividing the data into four parts and running the optimization on four different Google Colab accounts in parallel, I bring the time down even further to 8 hundreths of a second.

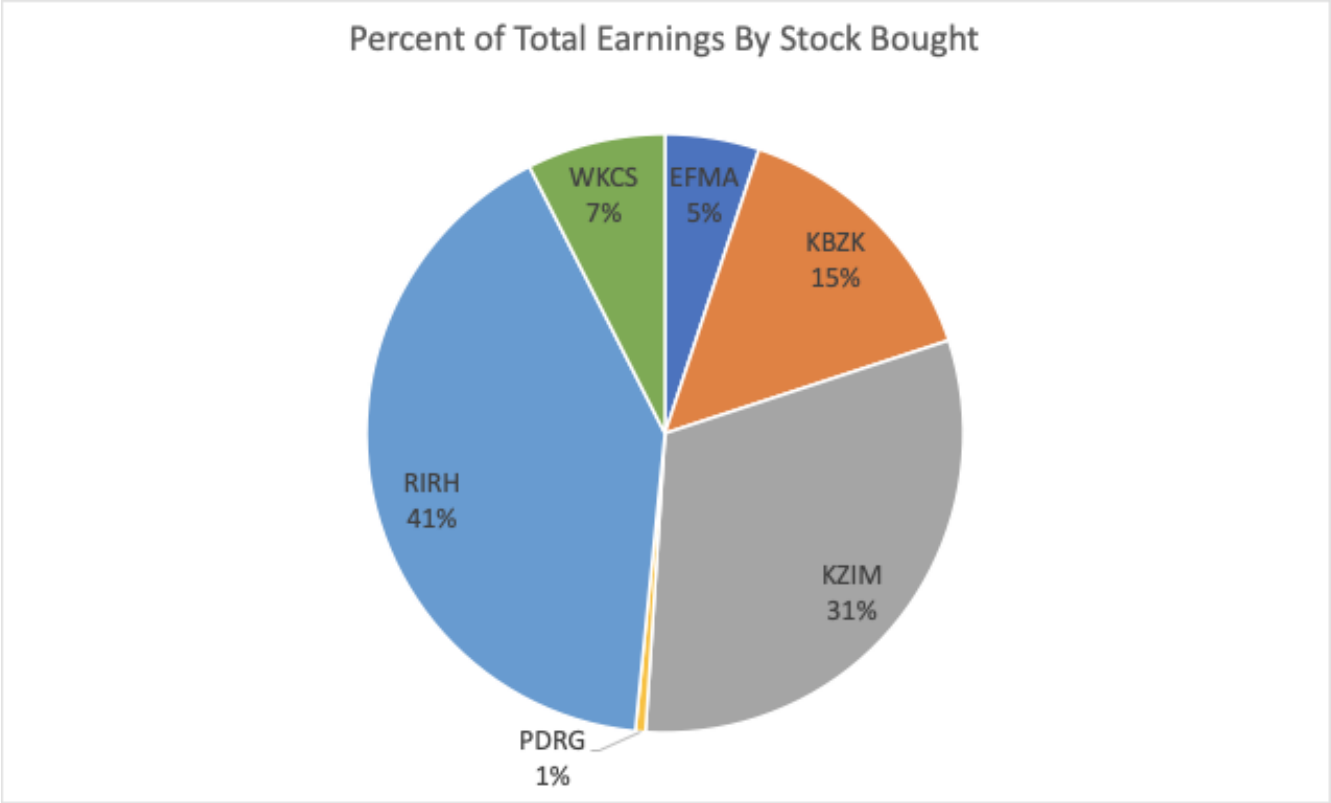
Findings:

After plotting the `max_earnings_buyday` dictionary, I get my maximum earnings and dates.



Running the first optimization function for this instance and setting `details = True,` I get the optimal stock portfolio given the constraints. I bought 10 shares of EFMA, KBZK, RIRH, and WKCS as well as 7 shares of KZIM and 1 share of PDRG.




Percent of Total Earnings By Stock Bought
WKCS
7%
EFMA
5%
KBZK
15%
RIRH
41%
KZIM
31%
PDRG
1%