Marcellus Internship Report

- Amitava Roy
Dated : 10 July, 2023

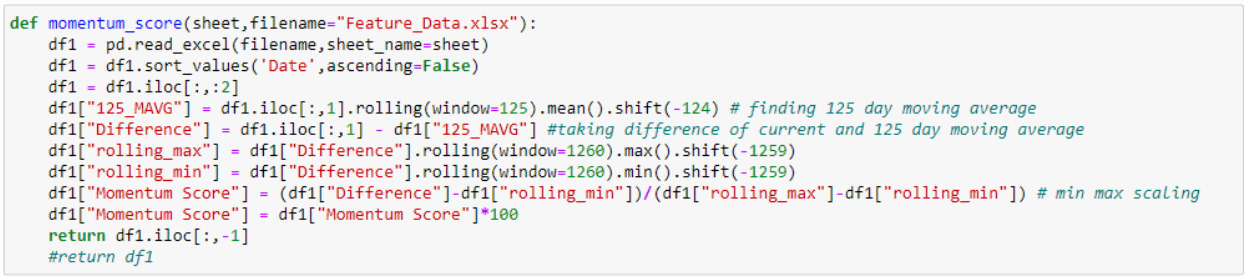
Aim of the Project:

The project aimed to develop a set of trading strategies on the S&P500 Index that can generate Buy, Hold and Sell Signals daily that can maximize CAGR and minimize drawdown. 18 years of daily data was used for the back testing of the project.

Features Used:

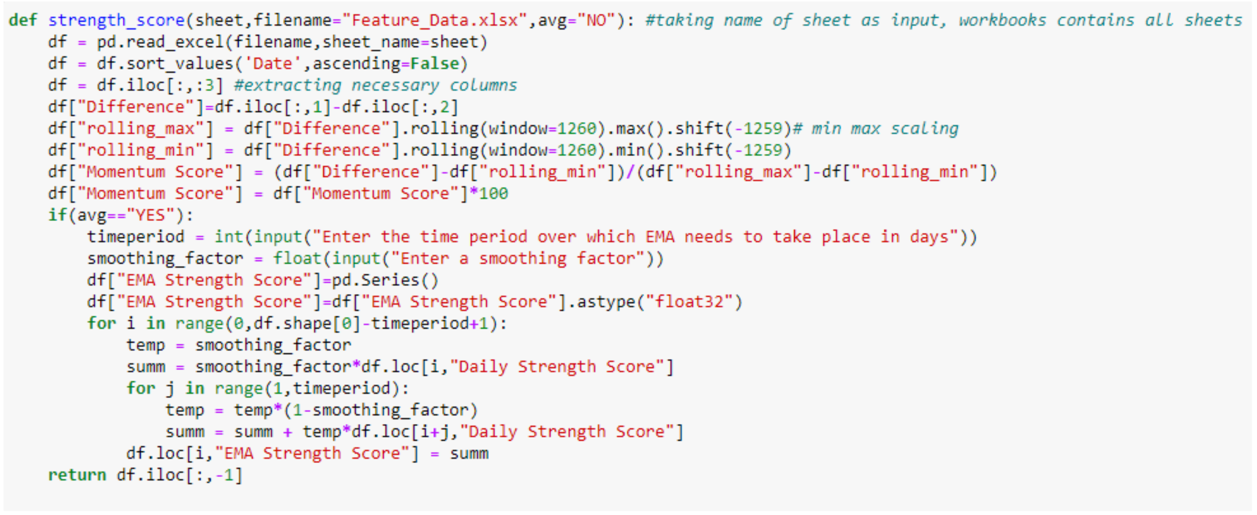
The project used 7 daily features which have been used by CNN for constructing their Fear and Greed Index namely:

- *Market Momentum*: Difference between SPXT and its 125-Day moving average



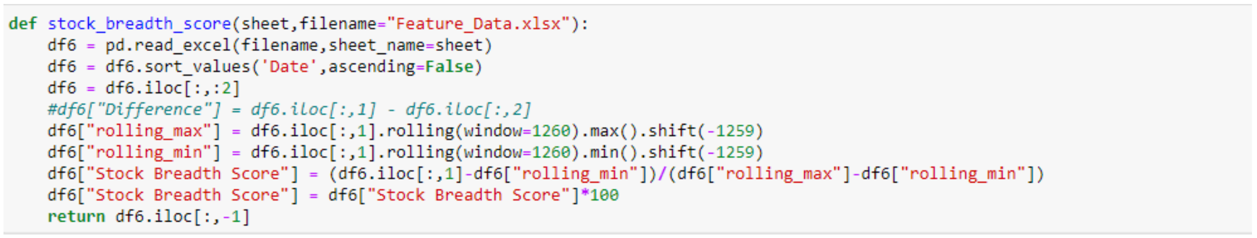
```python
def momentum_score(sheet,filename="Feature_Data.xlsx"):
    df1 = pd.read_excel(filename,sheet_name=sheet)
    df1 = df1.sort_values('Date',ascending=False)
    df1 = df1.iloc[:,:2]
    df1["125_MAVG"] = df1.iloc[:,1].rolling(window=125).mean().shift(-124) # finding 125 day moving average
    df1["Difference"] = df1.iloc[:,1] - df1["125_MAVG"] #taking difference of current and 125 day moving average
    df1["rolling_max"] = df1["Difference"].rolling(window=1260).max().shift(-1259)
    df1["rolling_min"] = df1["Difference"].rolling(window=1260).min().shift(-1259)
    df1["Momentum Score"] = (df1["Difference"]-df1["rolling_min"])/(df1["rolling_max"]-df1["rolling_min"]) # min max scaling
    df1["Momentum Score"] = df1["Momentum Score"]*100
    return df1.iloc[:,-1]
    #return df1
```

- *Stock Price Strength*: Difference between 52-week highs and 52-week lows



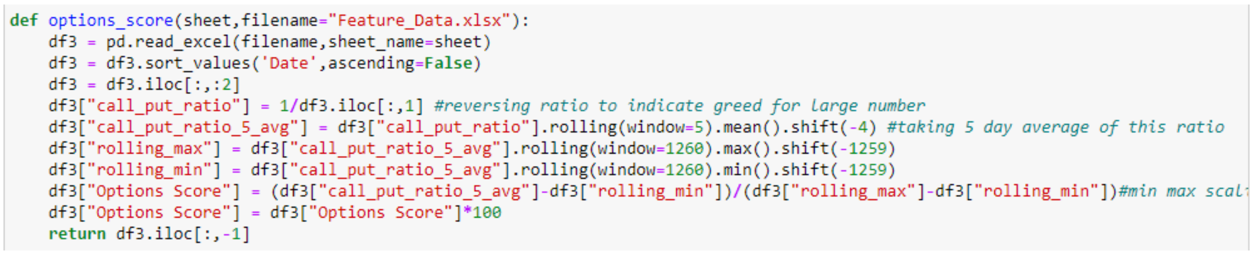
```python
def strength_score(sheet,filename="Feature_Data.xlsx",avg="NO"): #taking name of sheet as input, workbooks contains all sheets
    df = pd.read_excel(filename,sheet_name=sheet)
    df = df.sort_values('Date',ascending=False)
    df = df.iloc[:,:3] #extracting necessary columns
    df["Difference"]=df.iloc[:,1]-df.iloc[:,2]
    df["rolling_max"] = df["Difference"].rolling(window=1260).max().shift(-1259)# min max scaling
    df["rolling_min"] = df["Difference"].rolling(window=1260).min().shift(-1259)
    df["Momentum Score"] = (df["Difference"]-df["rolling_min"])/(df["rolling_max"]-df["rolling_min"])
    df["Momentum Score"] = df["Momentum Score"]*100
    if(avg=="YES"):
        timeperiod = int(input("Enter the time period over which EMA needs to take place in days"))
        smoothing_factor = float(input("Enter a smoothing factor"))
        df["EMA Strength Score"]=pd.Series()
        df["EMA Strength Score"]=df["EMA Strength Score"].astype("float32")
        for i in range(0,df.shape[0]-timeperiod+1):
            temp = smoothing_factor
            summ = smoothing_factor*df.loc[i,"Daily Strength Score"]
            for j in range(1,timeperiod):
                temp = temp*(1-smoothing_factor)
                summ = summ + temp*df.loc[i+j,"Daily Strength Score"]
            df.loc[i,"EMA Strength Score"] = summ
    return df.iloc[:,-1]
```

- *Stock Price Breadth*: McClellan Volume Summation Index



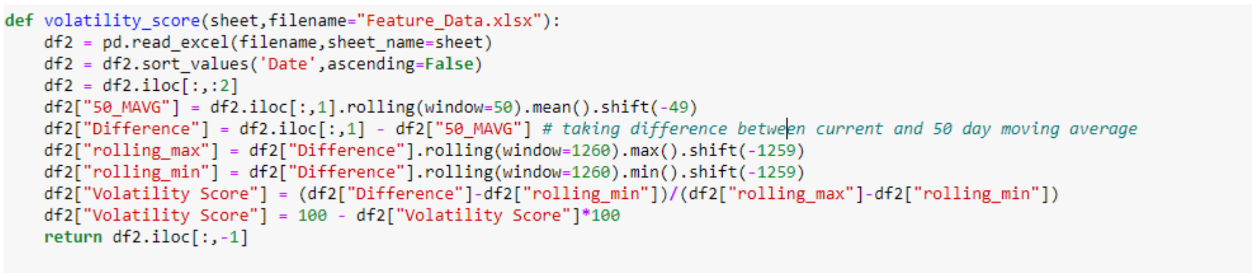
```python
def stock_breadth_score(sheet,filename="Feature_Data.xlsx"):
    df6 = pd.read_excel(filename,sheet_name=sheet)
    df6 = df6.sort_values('Date',ascending=False)
    df6 = df6.iloc[:,:2]
    #df6["Difference"] = df6.iloc[:,1] - df6.iloc[:,2]
    df6["rolling_max"] = df6.iloc[:,1].rolling(window=1260).max().shift(-1259)
    df6["rolling_min"] = df6.iloc[:,1].rolling(window=1260).min().shift(-1259)
    df6["Stock Breadth Score"] = (df6.iloc[:,1]-df6["rolling_min"])/(df6["rolling_max"]-df6["rolling_min"])
    df6["Stock Breadth Score"] = df6["Stock Breadth Score"]*100
    return df6.iloc[:,-1]
```

- *Put and Call Options*: 5-Day average of (Number of Call Options/Number of Put Options)



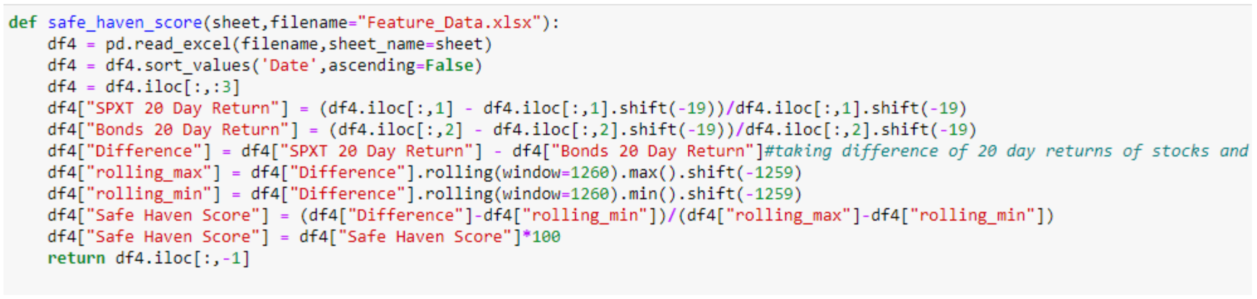
```python
def options_score(sheet,filename="Feature_Data.xlsx"):
    df3 = pd.read_excel(filename,sheet_name=sheet)
    df3 = df3.sort_values('Date',ascending=False)
    df3 = df3.iloc[:,:2]
    df3["call_put_ratio"] = 1/df3.iloc[:,1] #reversing ratio to indicate greed for large number
    df3["call_put_ratio_5_avg"] = df3["call_put_ratio"].rolling(window=5).mean().shift(-4) #taking 5 day average of this ratio
    df3["rolling_max"] = df3["call_put_ratio_5_avg"].rolling(window=1260).max().shift(-1259)
    df3["rolling_min"] = df3["call_put_ratio_5_avg"].rolling(window=1260).min().shift(-1259)
    df3["Options Score"] = (df3["call_put_ratio_5_avg"]-df3["rolling_min"])/(df3["rolling_max"]-df3["rolling_min"])#min max scal
    df3["Options Score"] = df3["Options Score"]*100
    return df3.iloc[:,-1]
```

- *Market Volatility*: Difference between VIX and its 50-Day Moving average



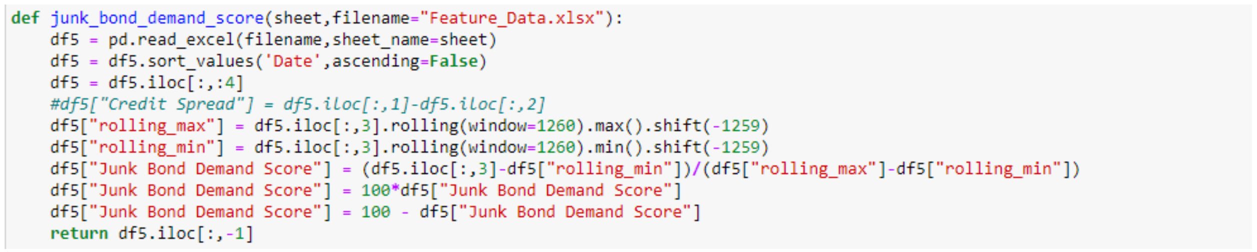
```python
def volatility_score(sheet,filename="Feature_Data.xlsx"):
    df2 = pd.read_excel(filename,sheet_name=sheet)
    df2 = df2.sort_values('Date',ascending=False)
    df2 = df2.iloc[:,:2]
    df2["50_MAVG"] = df2.iloc[:,1].rolling(window=50).mean().shift(-49)
    df2["Difference"] = df2.iloc[:,1] - df2["50_MAVG"] # taking difference between current and 50 day moving average
    df2["rolling_max"] = df2["Difference"].rolling(window=1260).max().shift(-1259)
    df2["rolling_min"] = df2["Difference"].rolling(window=1260).min().shift(-1259)
    df2["Volatility Score"] = (df2["Difference"]-df2["rolling_min"])/(df2["rolling_max"]-df2["rolling_min"])
    df2["Volatility Score"] = 100 - df2["Volatility Score"]*100
    return df2.iloc[:,-1]
```

- *Safe Haven Demand*: Difference in 20-Day Stock and Bond Returns



```python
def safe_haven_score(sheet,filename="Feature_Data.xlsx"):
    df4 = pd.read_excel(filename,sheet_name=sheet)
    df4 = df4.sort_values('Date',ascending=False)
    df4 = df4.iloc[:,:3]
    df4["SPXT 20 Day Return"] = (df4.iloc[:,1] - df4.iloc[:,1].shift(-19))/df4.iloc[:,1].shift(-19)
    df4["Bonds 20 Day Return"] = (df4.iloc[:,2] - df4.iloc[:,2].shift(-19))/df4.iloc[:,2].shift(-19)
    df4["Difference"] = df4["SPXT 20 Day Return"] - df4["Bonds 20 Day Return"]#taking difference of 20 day returns of stocks and
    df4["rolling_max"] = df4["Difference"].rolling(window=1260).max().shift(-1259)
    df4["rolling_min"] = df4["Difference"].rolling(window=1260).min().shift(-1259)
    df4["Safe Haven Score"] = (df4["Difference"]-df4["rolling_min"])/(df4["rolling_max"]-df4["rolling_min"])
    df4["Safe Haven Score"] = df4["Safe Haven Score"]*100
    return df4.iloc[:,-1]
```

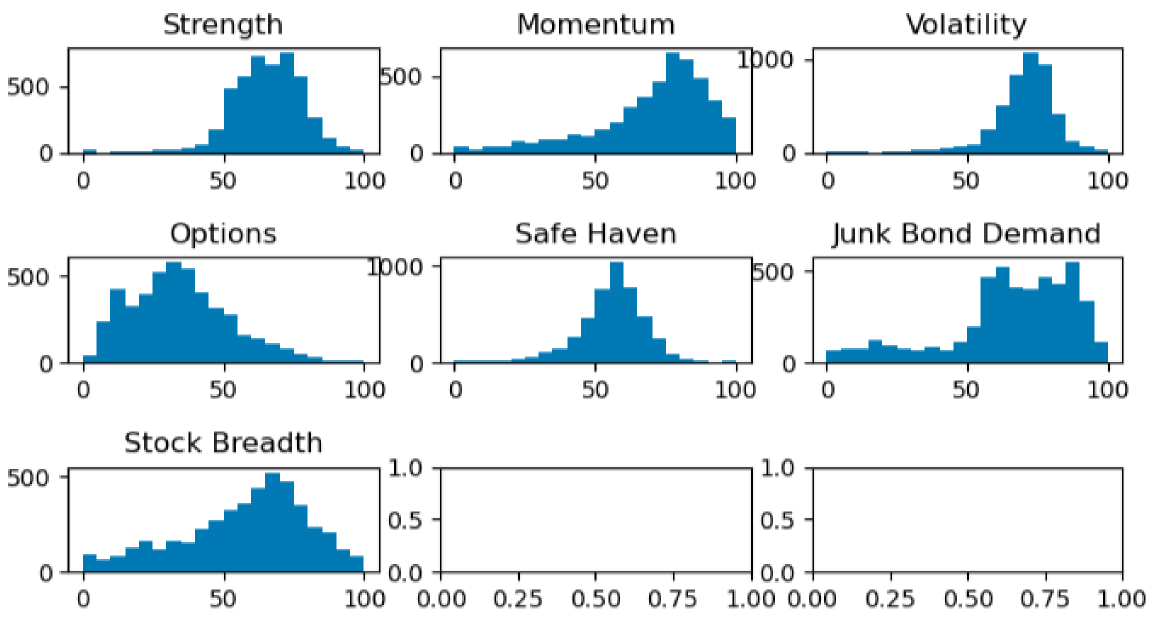
- *Junk Bond Demand*: Yield spread - junk bonds v investment grade



```python
def junk_bond_demand_score(sheet,filename="Feature_Data.xlsx"):
    df5 = pd.read_excel(filename,sheet_name=sheet)
    df5 = df5.sort_values('Date',ascending=False)
    df5 = df5.iloc[:,:4]
    #df5["Credit Spread"] = df5.iloc[:,1]-df5.iloc[:,2]
    df5["rolling_max"] = df5.iloc[:,3].rolling(window=1260).max().shift(-1259)
    df5["rolling_min"] = df5.iloc[:,3].rolling(window=1260).min().shift(-1259)
    df5["Junk Bond Demand Score"] = (df5.iloc[:,3]-df5["rolling_min"])/(df5["rolling_max"]-df5["rolling_min"])
    df5["Junk Bond Demand Score"] = 100*df5["Junk Bond Demand Score"]
    df5["Junk Bond Demand Score"] = 100 - df5["Junk Bond Demand Score"]
    return df5.iloc[:,-1]
```

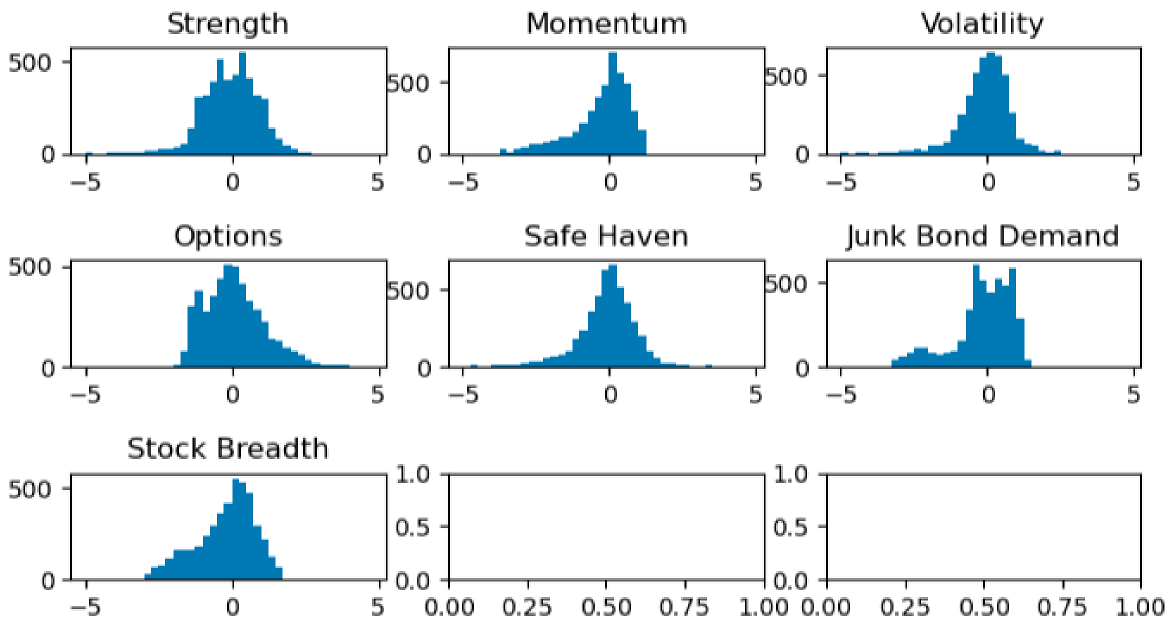
Note: *All these features have been normalized to a value between 0 and 100 using Min-Max Scaling. 5-year rolling minimum and maximum values were used for scaling the features. The data was extracted from Bloomberg Terminal (last 18 years data).Fear and Greed Index was calculated by taking an average of all these 7 features.*

Distributions of all the 7 features:

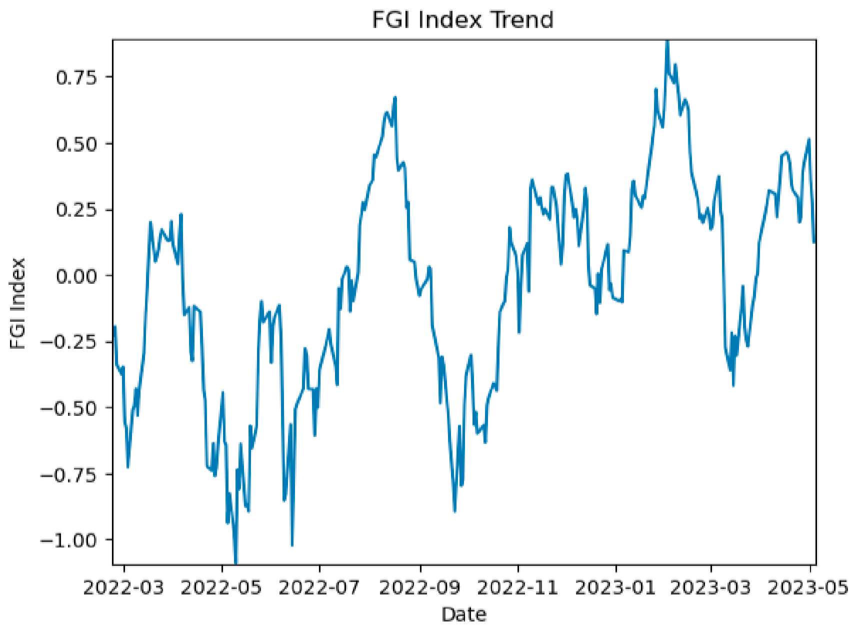


Note: *Since, all the features have different means and medians, the z-score of each feature was calculated for a better understanding*.

Distributions of Z-Scores:

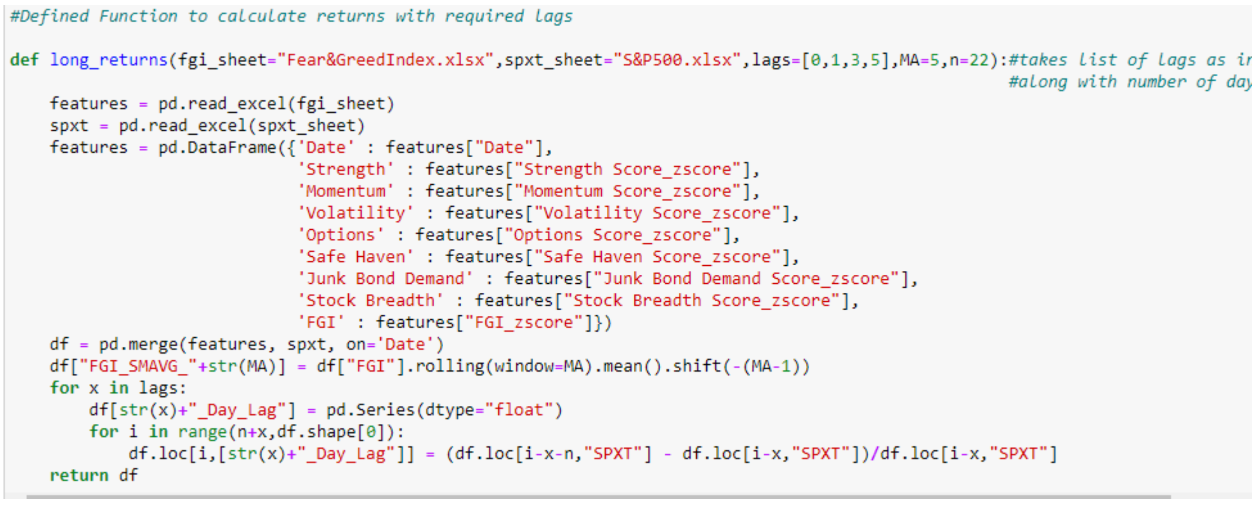


Plotting the Trend of FGI_zscore for the last 300 days:



Calculating Returns:

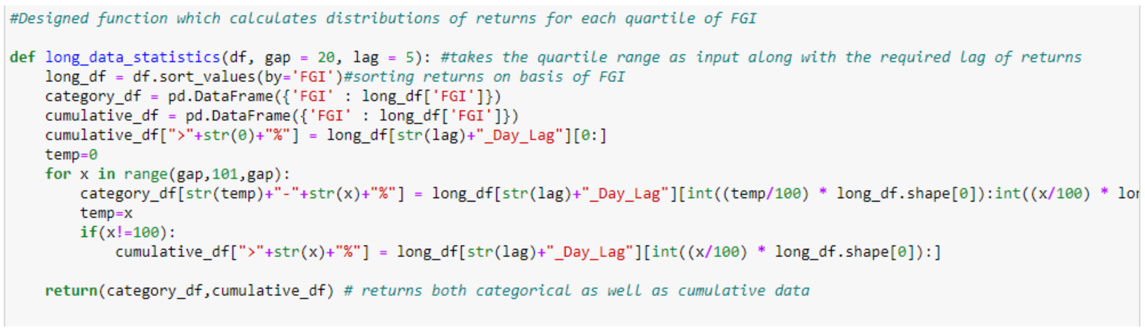
Designed a function that can calculate returns for any “n” number of days with a given list of lags.



```
#Defined Function to calculate returns with required lags

def long_returns(fgi_sheet="Fear&GreedIndex.xlsx",spxt_sheet="S&P500.xlsx",lags=[0,1,3,5],MA=5,n=22):#takes list of lags as i
                                                                                                     #along with number of day
    features = pd.read_excel(fgi_sheet)
    spxt = pd.read_excel(spxt_sheet)
    features = pd.DataFrame({'Date' : features["Date"],
                         'Strength' : features["Strength Score_zscore"],
                         'Momentum' : features["Momentum Score_zscore"],
                         'Volatility' : features["Volatility Score_zscore"],
                         'Options' : features["Options Score_zscore"],
                         'Safe Haven' : features["Safe Haven Score_zscore"],
                         'Junk Bond Demand' : features["Junk Bond Demand Score_zscore"],
                         'Stock Breadth' : features["Stock Breadth Score_zscore"],
                         'FGI' : features["FGI_zscore"]})
    df = pd.merge(features, spxt, on='Date')
    df["FGI_SMAVG_"+str(MA)] = df["FGI"].rolling(window=MA).mean().shift(-(MA-1))
    for x in lags:
        df[str(x)+"_Day_Lag"] = pd.Series(dtype="float")
        for i in range(n+x,df.shape[0]):
            df.loc[i,[str(x)+"_Day_Lag"]] = (df.loc[i-x-n,"SPXT"] - df.loc[i-x,"SPXT"])/df.loc[i-x,"SPXT"]
    return df
```

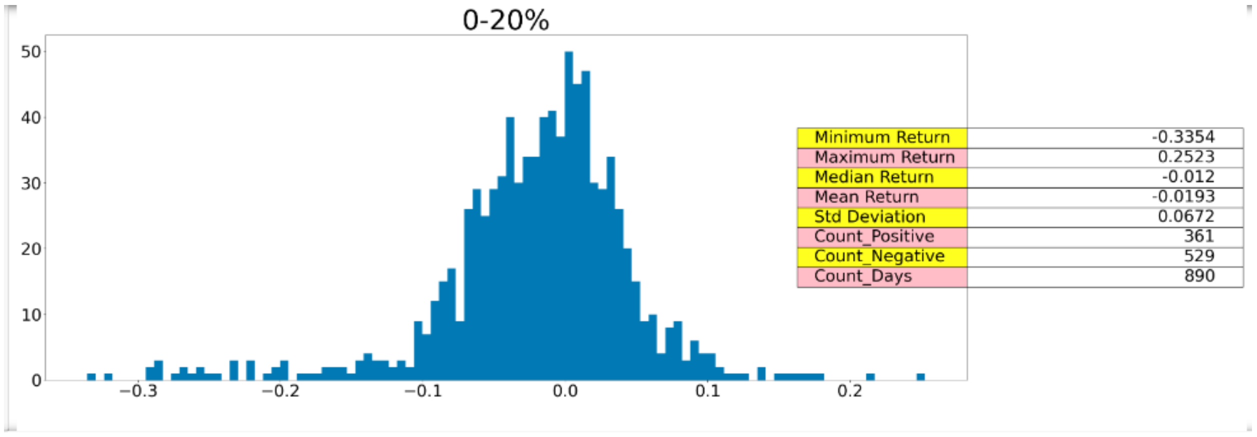
FGI Based Percentile Wise Analysis:

Designed a function to sort returns based on FGI scores categorically for each quartile. The function takes the required percentile range and the lag of the returns as input, and finds data both categorically and cumulatively.

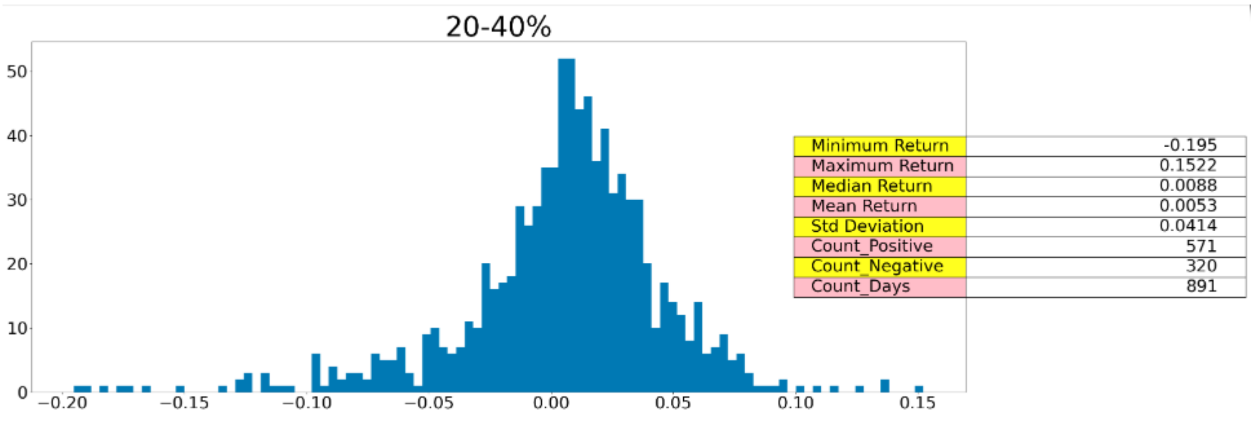


```python
#Designed function which calculates distributions of returns for each quartile of FGI

def long_data_statistics(df, gap = 20, lag = 5): #takes the quartile range as input along with the required lag of returns
    long_df = df.sort_values(by='FGI')#sorting returns on basis of FGI
    category_df = pd.DataFrame({'FGI' : long_df['FGI']})
    cumulative_df = pd.DataFrame({'FGI' : long_df['FGI']})
    cumulative_df[">"+str(0)+"%"] = long_df[str(lag)+"_Day_Lag"][0:]
    temp=0
    for x in range(gap,101,gap):
        category_df[str(temp)+"-"+str(x)+"%"] = long_df[str(lag)+"_Day_Lag"][int((temp/100) * long_df.shape[0]):int((x/100) * lon
        temp=x
        if(x!=100):
            cumulative_df[">"+str(x)+"%"] = long_df[str(lag)+"_Day_Lag"][int((x/100) * long_df.shape[0]):]

    return(category_df,cumulative_df) # returns both categorical as well as cumulative data
```

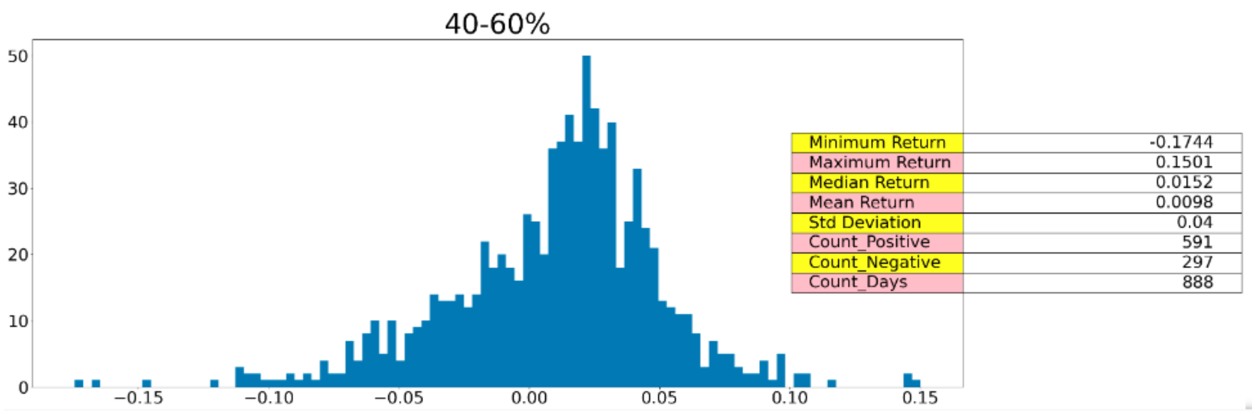
Categorical Distributions:



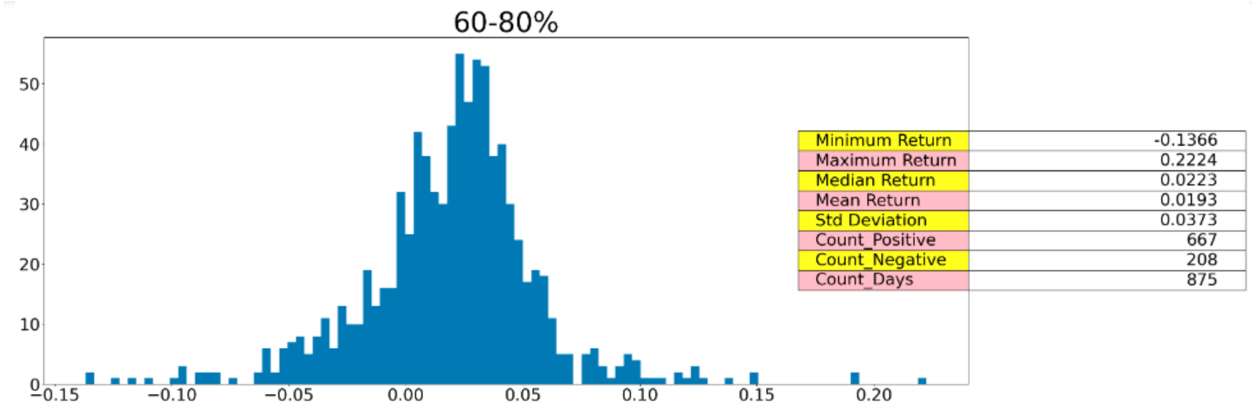
Minimum Return	-0.3354
Maximum Return	0.2523
Median Return	-0.012
Mean Return	-0.0193
Std Deviation	0.0672
Count_Positive	361
Count_Negative	529
Count_Days	890



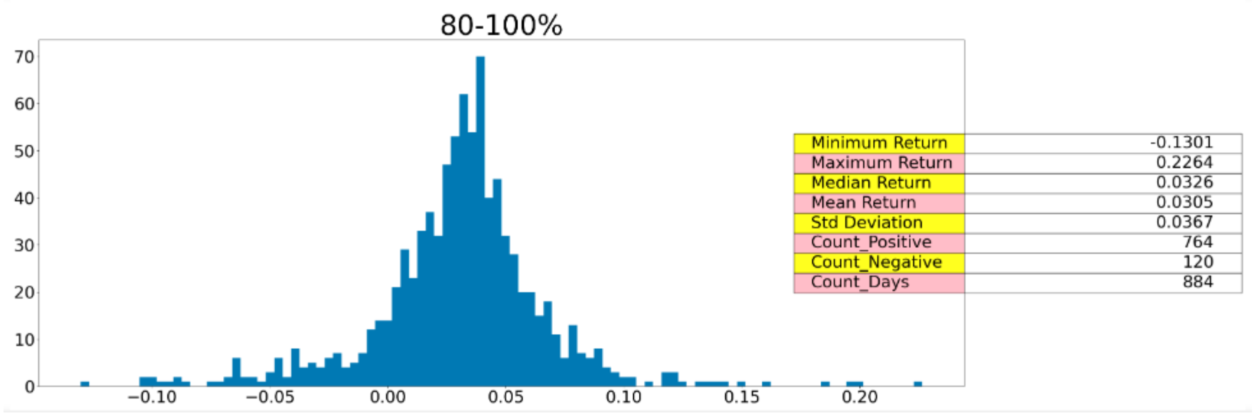
Minimum Return	-0.195
Maximum Return	0.1522
Median Return	0.0088
Mean Return	0.0053
Std Deviation	0.0414
Count_Positive	571
Count_Negative	320
Count_Days	891



Minimum Return	-0.1744
Maximum Return	0.1501
Median Return	0.0152
Mean Return	0.0098
Std Deviation	0.04
Count_Positive	591
Count_Negative	297
Count_Days	888



Minimum Return	-0.1366
Maximum Return	0.2224
Median Return	0.0223
Mean Return	0.0193
Std Deviation	0.0373
Count_Positive	667
Count_Negative	208
Count_Days	875



Minimum Return	-0.1301
Maximum Return	0.2264
Median Return	0.0326
Mean Return	0.0305
Std Deviation	0.0367
Count_Positive	764
Count_Negative	120
Count_Days	884

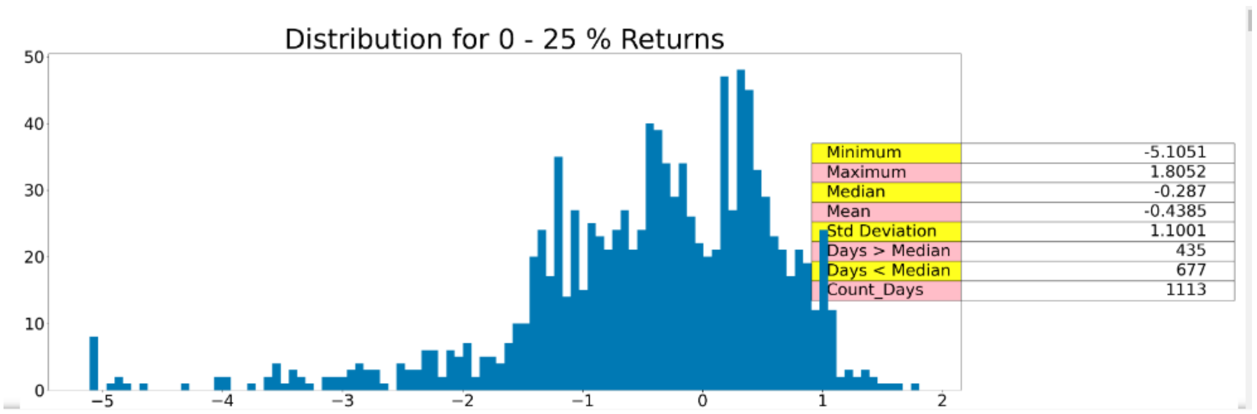
Note: *Created similar plots for cumulative data as well*

Returns Based Percentile Wise Analysis of Every Feature:

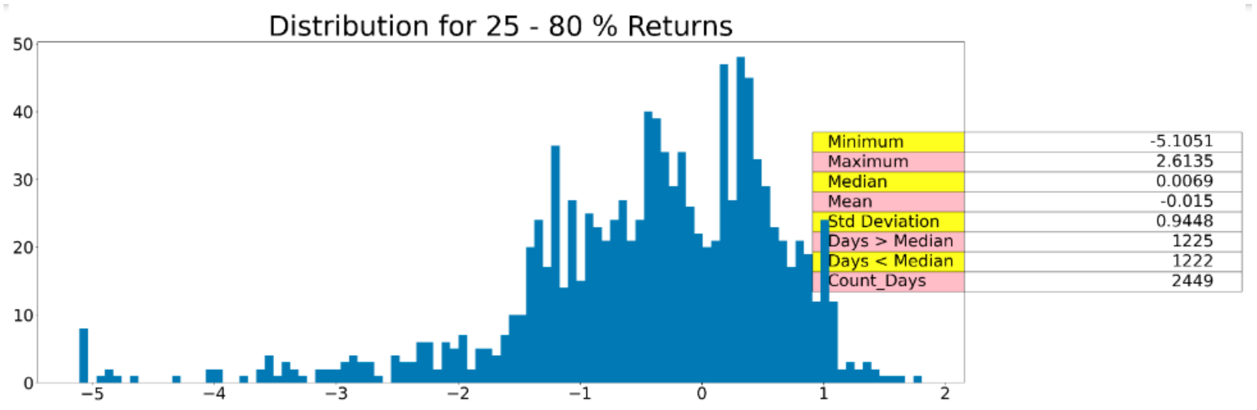
Conduced a similar analysis like before for each feature based on 3 cohort of returns i.e. low returns, medium returns, and high returns.

Designed a function that can sort the scores of each feature based on returns and plot them on a histogram for each cohort of returns and display some statistics about that data like minimum, maximum, median, mean, std deviation, days where the value of feature was more than its overall median, and lesser than its overall median.

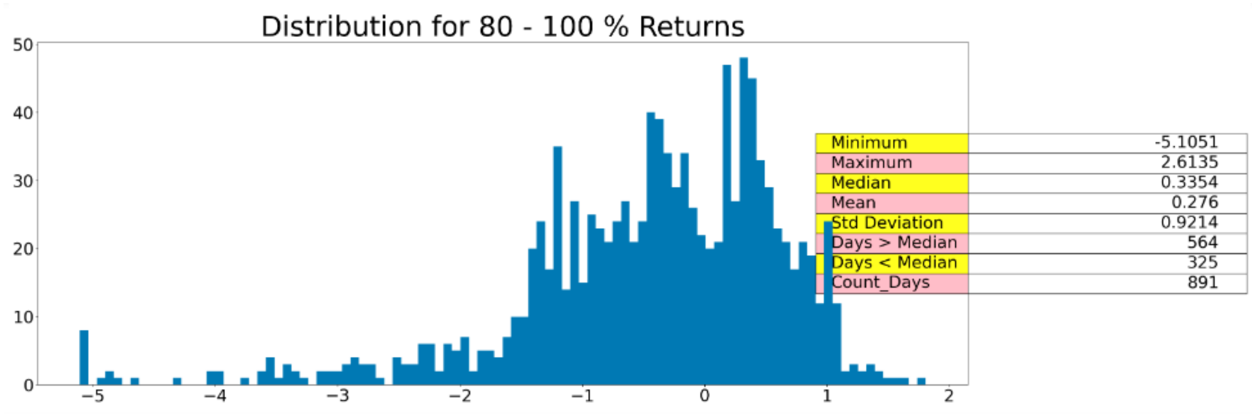
Example of the Analysis (Strength Score):



Minimum	-5.1051
Maximum	1.8052
Median	-0.287
Mean	-0.4385
Std Deviation	1.1001
Days > Median	435
Days < Median	677
Count_Days	1113



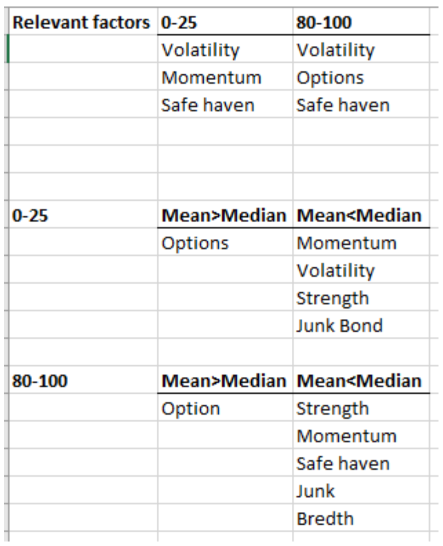
Minimum	-5.1051
Maximum	2.6135
Median	0.0069
Mean	-0.015
Std Deviation	0.9448
Days > Median	1225
Days < Median	1222
Count_Days	2449



Minimum	-5.1051
Maximum	2.6135
Median	0.3354
Mean	0.276
Std Deviation	0.9214
Days > Median	564
Days < Median	325
Count_Days	891

Note: *Similarly, this was done for all the other features as well*

Insights Drawn from the Analysis:

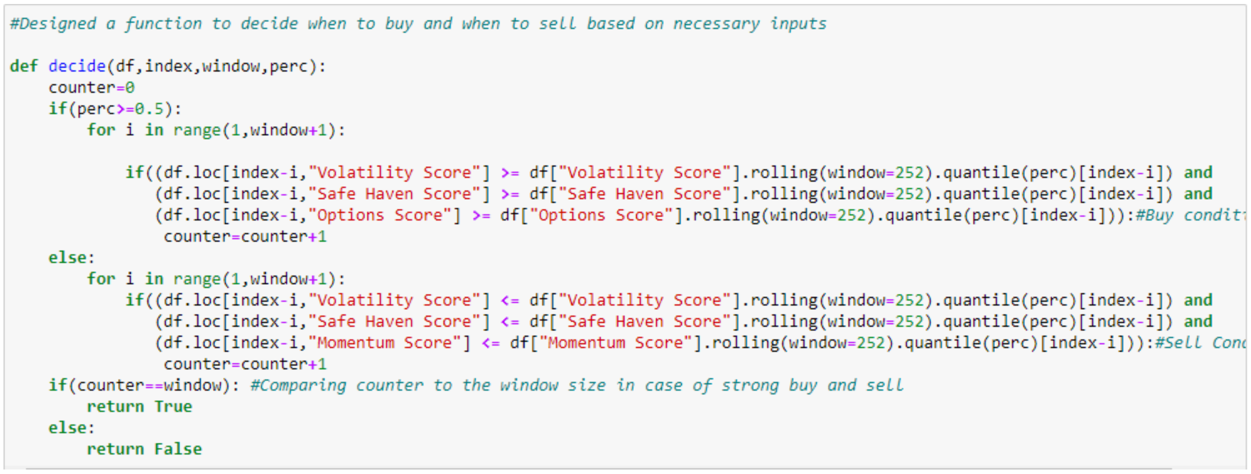


Relevant factors	0-25	80-100
	Volatility	Volatility
	Momentum	Options
	Safe haven	Safe haven
0-25	**Mean>Median**	**Mean<Median**
	Options	Momentum
		Volatility
		Strength
		Junk Bond
80-100	**Mean>Median**	**Mean<Median**
	Option	Strength
		Momentum
		Safe haven
		Junk
		Bredth

Note: *Therefore, Volatility and Safe Haven turned out to be leading indicators for high returns as well as low returns. Momentum was found to be a good indicator for low returns and Options Score seemed to be a good indicator for high returns. Based on these insights, trading strategies were built.*

Designing functions for building trading strategies:

Designed the "*decide*" function which takes the decision to buy or sell. It takes as input the current index at which the decision needs to be taken and the window size i.e. the number of days prior to the current index whose signal needs to be looked at.



```
#Designed a function to decide when to buy and when to sell based on necessary inputs

def decide(df,index,window,perc):
    counter=0
    if(perc>=0.5):
        for i in range(1,window+1):

            if((df.loc[index-i,"Volatility Score"] >= df["Volatility Score"].rolling(window=252).quantile(perc)[index-i]) and
               (df.loc[index-i,"Safe Haven Score"] >= df["Safe Haven Score"].rolling(window=252).quantile(perc)[index-i]) and
               (df.loc[index-i,"Options Score"] >= df["Options Score"].rolling(window=252).quantile(perc)[index-i])):#Buy condit
                counter=counter+1
    else:
        for i in range(1,window+1):
            if((df.loc[index-i,"Volatility Score"] <= df["Volatility Score"].rolling(window=252).quantile(perc)[index-i]) and
               (df.loc[index-i,"Safe Haven Score"] <= df["Safe Haven Score"].rolling(window=252).quantile(perc)[index-i]) and
               (df.loc[index-i,"Momentum Score"] <= df["Momentum Score"].rolling(window=252).quantile(perc)[index-i])):#Sell Con
                counter=counter+1
    if(counter==window): #Comparing counter to the window size in case of strong buy and sell
        return True
    else:
        return False
```

Designed the "*position_function*" which returns the current position at every index i.e. 1 whenever we are invested in the market and 0 otherwise.

```python
def position_function(df):

    position = []

    for i in range(len(df)):
        position.append(0)


    for i in range(len(df)):
        if df["Trading_Signal"][i] == 1:#signal = 1 for BUY
            position[i] = 1              #position changed from 0 to 1
        elif df["Trading_Signal"][i] == -1:#signal = -1 for sell
            position[i] = 0                #position changed from 1 to 0
        else:
            if i==0:
                position[i] = 0
            else:
                position[i] = position[i-1]  #If no trading signal, position is same as before

    return position
```

Designed the “*returns_function*” which calculates the return associated with every transaction and shows it next to every to every selling price i.e. every time we are selling and squaring off our position.

```python
def returns_function(df):

    returns = [0]*len(df)

    temp = 0

    for i in range(1,len(df)):
        if(df["Position"][i]==1 and df["Position"][i-1]==0):
            temp = df["SPXT"][i] #storing buy price
        elif(df["Position"][i]==0 and df["Position"][i-1]==1):
            returns[i] = (df["SPXT"][i] - temp)/temp #calculating return of transaction when sell signal encountered
            temp = 0#resetting buy price to 0

    return returns
```

Designed the “*assets_function*” which calculates the current assets under management for every index. It continuously changes whenever we are invested in the market and remains constant when we are not.

```python
def assets_function(df,principal):

    df["Asset"] = pd.Series()
    df["Asset"][0] = principal#initial asset = principal amount

    n_stocks = 0
    curr_asset = principal
    for i in range(1,len(df)):
        if(df["Position"][i]==1 and df["Position"][i-1]==0):
            n_stocks = curr_asset/df["SPXT"][i] #number of stocks bought is current assets divided by price of spxt that day
            df["Asset"][i] = n_stocks * df["SPXT"][i]
        elif(df["Position"][i]==0 and df["Position"][i-1]==1):
            curr_asset = n_stocks * df["SPXT"][i]#current asset becomes stocks sold multiplied by price of spxt that day
            n_stocks = 0
            df["Asset"][i] = curr_asset
        else:
            if df["Position"][i]==1:
                df["Asset"][i] = n_stocks * df["SPXT"][i]#when invested in market, current asset is stocks sold multiplied by pr
            else:
                df["Asset"][i] = df["Asset"][i-1]#when not invested in market, current asset is same as before

    return df["Asset"]
```

Building Trading Strategies:

Based on above insights, 4 different strategies were built:

Strategy 1: Simple Buy and Sell signal, no fixed holding period

Strategy 2: Strong Buy and Sell signal (2 consecutive days), no fixed holding period

Strategy 3: Strong Buy (2 consecutive days) and Simple Sell signal, no fixed holding period

Strategy 4: Strong Buy (2 consecutive days) and Simple Sell signal, fixed holding period of 60 days

Results:

Strategy 1: CAGR = 14.56% , Drawdown = -42.55% , No. of Transactions = 31

Strategy 2: CAGR = 15.25% , Drawdown = -31.90% , No. of Transactions = 27

Strategy 3: CAGR = 15.25% , Drawdown = -31.90% , No. of Transactions = 27

Strategy 4: CAGR = 9.27% , Drawdown = 29.56% , No. of Transactions = 33

Following this, designed a function that can display a menu to choose from these 4 strategies, which will then execute the chosen strategy and display the CAGR, Drawdown, No. of transactions and current signal, and return 18 years of backtesting data as a dataframe.

Apart from this, also designed appending functions that can append daily data for each feature extracted from Bloomberg into the existing database.