01/03/2021
Bar Dahan

Final Project – Data Science for Finance 55759

1. Databases:
   1.1. Stocks in the Russell 1000 Index – A list of stocks that have been listed on Russell1000 between January 2007 and October 2020.
   1.2. Signal_from_bert_70 – Sentiment Analysis Scores and Data from 2007 to September 2020.
   1.3. Monthly_Data – Financial Data of Russell1000 stocks from January 1St 2007 to January 1st 2021.
   1.4. SMA – A file listing the most profitable Short and Long moving average periods for each stock (2007-2014).
2. Data Preparation:
   2.1. SMA
      2.1.1. Running a program we used on Exercise 4, picking up the best combination of Short and Long periods.
      2.1.2. The program runs on the "Train Period": From January 1st 2007 to December 31st 2004.
      2.1.3. Final output is a dataframe of 603 stocks and their respective most profitable Short and Long periods.
   2.2. Signal_from_bert_70
      2.2.1. Cleaning Ticker Name, leaving a "Short Ticker" column, similar to the ticker names given from yfinance.
      2.2.2. Filtering the dataframe by the 2005 stocks I chose before.
      2.2.3. Clearing dates, inserting "Short Date" column that contain the regular YYYY-MM-DD format.
      2.2.4. Creating a "Year_Month" column, to attribute the sentiment to one month **after** it was analyzed. This column is in YYYY-MM format.
   2.3. Monthly_Data
      2.3.1. For each ticker, I downloaded data from 2007 to 2020.
      2.3.2. Out of 2005 initial tickers, yfinance provided data for **1,437 tickers**.
      2.3.3. I created a similar "Year_Month" column, to be able to merge this dataframe with the Sentiment dataframe.
      2.3.4. I created "SMA_Short" and "SMA_Long" columns, each with the most profitable moving average (which I took from SMA dataframe).
         2.3.4.1. For stocks that don't exist within the SMA dataframe: the average Short and Long periods of the SMA dataframe.
      2.3.5. I created "UpperBand" and "LowerBand", by a 20-day period and a multiplier of 2, according to the Bollinger Bands method taught in class.
      2.3.6. I appended the data using a list and converted it to the dataframe "df1".
   2.4. Merged (or "mdf")
      2.4.1. This dataframe is a merge between the Monthly Data and the Sentiment dataframes.
      2.4.2. I merged it to the Monthly Data on Short Ticker and Year_Month.
      2.4.3. In that way, sentiments were assigned to the following month.
         2.4.3.1. For example: If I have a sentiment from September 14th, it'd be assigned to the following October 1st (Since it is irrelevant to Septmeber 1st).
3. Method
   3.1. Train Period: 01/01/2007 – 31/12/2014
   3.2. Test Period: 01/01/2015 – 31/12/2020
   3.3. I split the data to Train and Test sets according to the periods above.

3.4. To each set I added several variables:

3.4.1. Upper_i, Lower_i – indicators ranging from -1 to 0 and from 1 to 0 respectively, by the difference between the stock's Adj Close price to the upper/lower band.

3.4.2. Sma_i – an indicator that returns 1 when the Short MA is higher or equal to the Long MA, and -1 otherwise.

3.4.3. Rel_sa – a variable that reflects how is a sentiment score compared to the best sentiment score that was given to the stock up to this point.

3.4.4. Normalized Volume – To each month I calculated the mean and std Volume of all the stocks, and normalized each stock's volume.

3.5. The 5 variables listed above are the determining factors that will form the weighted score, according to which we'll rate the stocks and decide how to invest.

3.6. Thus, we have 6 unknowns: 5 coefficients to the determining factors, and the percentage of top stocks we wish to pick in each month.

3.7. This program receives values for the 6 unknowns from training the model on Train set.

3.8. Investment Method – **Long Only**:

3.8.1. In each month, we decide in the **end of the first day** which stocks to buy and sell. Stocks prices are the Adj Close prices known of each month's first day.

3.9. Training the model on Train Set

3.9.1. First Part

```
def train_model(train,s_sum):
    pd.set_option('mode.chained_assignment', None)
    df = train
    profit = 0
    n_list=[]
    current = s_sum
    portfolio={}
    t_coefs=[]
    first_month = list(df['Year_Month'].unique())[0]
    last_month = list(df['Year_Month'].unique())[-1]
    for n in tqdm(np.arange(0.01, 0.31, 0.01), position = 0, leave = True):
        for month in list(df['Year_Month'].unique()):
            is_month = df['Year_Month']==month
            month_df = df[is_month]
            coefs = train_coefs(month_df)
            t_coefs.append(coefs)
            month_df = weighted_score(month_df,coefs)
            top_n = round(n*len(list(month_df['Short_Ticker'])))
            top_df = month_df.nlargest(top_n, 'Weighted_Score')
            top_df['Percent'] = [
                float(x/(top_df['Weighted_Score'].sum())) for x in top_df['Weighted_Score']]
```

The first part of the train_model function consists of the following steps:

1. Setting profit to 0 (initially).
2. Creating a list of n's (%'s) and coefficients
3. Creating the Portfolio dictionary – in which we'll manage the stocks we have; Tickers as keys and amount of stocks as value.
4. Setting first and last months of the dataframe – we define them for knowing how to handle the first month (in which we only buy) and the last month(in which we only sell)
5. For each n between 1%-30% we run a training simulation.
6. For each month in the dataframe, we filter by month.

7. Then we use the train_coefs function to get the coefficients for that month.
8. We then proceed to use weighted_score function to calculate the weighted score for each stock.
9. Then we pick the top n stocks in each month, and we split the investment between them by the column 'Percent' which is basically the ratio between the stock's weighted score and the sum of weighted scores in that month.

3.9.2. Train_coefs function

```
def train_coefs(train):
    coefs=[]
    df= train
    SA = scipy.stats.pearsonr(df['Monthly_Yield'].notna(),df['Relative_SA'].notna())[1]
    SMA = scipy.stats.pearsonr(df['Monthly_Yield'].notna(),df['SMA_Ind'].notna())[1]
    U_I = scipy.stats.pearsonr(df['Monthly_Yield'].notna(),df['Upper_Ind'].notna())[1]
    L_I = scipy.stats.pearsonr(df['Monthly_Yield'].notna(),df['Lower_Ind'].notna())[1]
    nVol = scipy.stats.pearsonr(
        df['Monthly_Yield'].notna(),df['Normalized_Volume'])[1]
    if not isinstance(SA, float):
        SA = np.nanmean([SMA,U_I,L_I,nVol])
    if not isinstance(SMA, float):
        SMA = np.nanmean([SA,U_I,L_I,nVol])
    if not isinstance(U_I, float):
        U_I = np.nanmean([SA,SMA,L_I,nVol])
    if not isinstance(L_I, float):
        L_I = np.nanmean([SA,U_I,SMA,nVol])
    if not isinstance(nVol, float):
        nVol = np.nanmean([SA,U_I,L_I,SMA])
    if [SA,SMA,U_I,L_I,nVol] == [np.nan,np.nan,np.nan,np.nan,np.nan]:
        coefs.extend([0.2,0.2,0.2,0.2,0.2])
    else:
        coefs.extend([SA,SMA,U_I,L_I,nVol])
    return coefs
```

This function receives the Monthly Dataframe, and then calculates the Pearson Correlation between the stocks' prices and the 5 determining factors – to each their own correlation score.
If one factor is not available – the score is set to be np.nan.
If all 5 factors are not available, a default set of 0.2's is given to allow purchasing (relevant for the first month the stock is introduced).

3.9.3. Weighted_Score function

```
def weighted_score(df,coefs):
    SA = coefs[0]
    SMA = coefs[1]
    U_I = coefs[2]
    L_I = coefs[3]
    nVol = coefs[4]
    df['Weighted_Score'] = [
        nVol*z if np.isnan(x) and np.isnan(y) and np.isnan(w)
        else (nVol*z + x*SA if np.isnan(y) and np.isnan(w)
        else (nVol*z + w*SMA if np.isnan(x) and np.isnan(y)
        else (nVol*z + y*U_I + v*L_I if np.isnan(x) and np.isnan(w)
        else (nVol*z + y*U_I + v*L_I + w*SMA if np.isnan(x)
        else (nVol*z + x*SA + w*SMA if np.isnan(y)
        else (nVol*z + x*SA + y*U_I + v*L_I if np.isnan(w)
        else nVol*z + x*SA + y*U_I + v*L_I + w*SMA))))))
        for v,w,x,y,z
        in zip(
            df['Lower_Ind'],df['SMA_Ind'],df['Relative_SA'],
            df['Upper_Ind'],df['Normalized_Volume'])]
    return df
```

This function receives the list of coefficients, and the Monthly Dataframe. Then it calculates the weighted score of each stock, removing factors that are not available (so the score won't be np.nan as well). The weighted score is the sum of the multiply of each factor with it's coefficient.

3.9.4. Second Part

```
        if month == first_month:
            current,portfolio = buy(top_df,portfolio,current)
        elif month == last_month:
            current,portfolio = sell(top_df,month_df,portfolio,current,'YES')
        else:
            current,portfolio = sell(top_df,month_df,portfolio,current)
            if current <=0:
                profit = -100
                del t_coefs[-1]
                break
            current,portfolio = buy(top_df,portfolio,current)
    profit = 100*(current - s_sum)/s_sum
    if profit >0:
        n_list.append(n)

final_n = np.median(n_list)
f_coefs = []
SA1 = np.nanmean([x[0] for x in t_coefs])
SMA1 =  np.nanmean([x[1] for x in t_coefs])
U_I1 =  np.nanmean([x[2] for x in t_coefs])
L_I1 =  np.nanmean([x[3] for x in t_coefs])
nVol1 =  np.nanmean([x[4] for x in t_coefs])
f_coefs.extend([SA1,SMA1,U_I1,L_I1,nVol1])
return final_n, f_coefs
```

In this part, if it is the first month – we only use the buy function. If it's the last month - we only use the sell function.

Else, we first use the sell function (in order to sell the stocks we wish to sell), and then we buy the new stocks we want with the money we got.

If after we sell the stocks out current is 0 – the simulation ends and we don't use the coefficients nor the % we ran in this try.

If in the end of the simulation our profit is positive, we add both coefficients and n to their respective lists.

Then, we take the **median n** and the **mean of each coefficient** and return two lists to use on the Test Set.

3.9.5. Buy Function

```
def buy(top_df,portfolio,current):
    base = current
    for ticker in list(top_df['Short_Ticker']):
        if ticker in list(portfolio.keys()):
            stocks = float(portfolio.get(ticker))
        else:
            stocks = 0
        invest = float(top_df.loc[top_df['Short_Ticker'] == ticker,\
                                  'Percent'].iloc[0])*base
        price = float(top_df.loc[top_df['Short_Ticker'] == ticker, 'Adj_Close'].iloc[0])*1.004
        stocks += float(invest/price)
        portfolio[ticker] = stocks
        current -= invest
    return current,portfolio
```

The Buy function receives the top_df (made of top n % of stocks in that month), the portfolio and the current amount of cash available.
The n for each stock we check if it exists in the portfolio already.
If it does – we set the amount of stocks as the current value.
If it doesn't- we set the amount of stocks as 0
Then we calculate how much of our current amount of money we wish to invest, according to the percentage we calculated before.
We then move on to the price- we take the Adj Close price known for the stock, and buy it with 40 basis points fee.
The amount of stocks added is the division of the invest variable and the price.
We set the amount of stocks as the new value in our portfolio.
We deduct the amount we invested from the current amount of money.
The function then returns the amount of money remained after we bought our stocks and the updated portfolio.

3.9.6. Sell function

```
def sell(top_df,month_df,portfolio,current,indicator='NO'):
    if indicator == 'NO':
        for ticker in list(portfolio.keys()):
            if ticker not in list(top_df['Short_Ticker']):
                if pd.isnull(portfolio.get(ticker)):
                    continue
                else:
                    try:
                        price = 0.996*float(month_df.loc[
                            month_df['Short_Ticker'] == ticker,'Adj_Close'].iloc[0])
                    except Exception:
                        continue
                    sell = float(portfolio.get(ticker))*price
                    current += sell
                    portfolio[ticker]=0
    else:
        for ticker in list(portfolio.keys()):
            try:
                price = 0.996*float(month_df.loc[
                            month_df['Short_Ticker'] == ticker,'Adj_Close'].iloc[0])
            except Exception:
                continue
            sell = float(portfolio.get(ticker))*price
            current += sell
            portfolio[ticker]=0
    return current,portfolio
```

This function receives the top_df, monthly df, portfolio, current and an indicator that tells if it is the last month.
If it isn't the last month: For each ticker in our portfolio, if the stock isn't amongst the stock we want to buy- we don't sell the stocks.
Else, we set the price according to the known Adj Close price of the current day, deducting a 40 basis points fee.

The amount of money gained is the multiply between the number of stocks and the price. We then add the amount gained to the current amount and set the number of stocks of that ticker to 0 in our portfolio.
If it is the last month: we simply sell each stock in the portfolio.
Please note that this model isn't taking into account OTC stocks. Such stocks are regarded as excess burden and are not sold, simply remain in the portfolio without ever sanctioning a sale. Main problem was that we took one month intervals, and the actual last price could've been anywhere in between.

3.10. Testing the model:

3.10.1. After training the model, I received the following parameters:

3.10.1.1. **N**=10.5%

3.10.1.2. Coefficients =

Relative SA	0.3900565559161011
SMA Indicator	0.07427839882514041
Upper Band Indicator	0.03392599711677078
Lower Band Indicator	0.03392599711677078
Normalized Volume	0.43635819201105164

3.10.1.3. Then I ran the simulation function on the Test set, with the given parameters.

3.10.1.4. First Part

```
def simulator(df,n,coefs,s_sum):
    profit = 0
    current = s_sum
    portfolio={}
    profits_dict={}
    first_month = list(df['Year_Month'].unique())[0]
    last_month = list(df['Year_Month'].unique())[-1]
    months = list(df['Year_Month'].unique())
    for month in tqdm(months , position = 0, leave = True):
        is_month = df['Year_Month']==month
        month_df = df[is_month]
        month_df = weighted_score(month_df,coefs)
        top_n = round(n*len(list(month_df['Short_Ticker'])))
        top_df = month_df.nlargest(top_n, 'Weighted_Score')
        top_df['Percent'] = [
            float(
                x/(top_df['Weighted_Score'].notna().sum())) for x in top_df['Weighted_Score']]
```

The first part of the function is very similar to the train_model, only it doesn't calculate coefficients or runs several %'s, but using the ones we received from the training function.

3.10.1.5. Second Part

```
        if month == first_month:
            current,portfolio = buy(top_df,portfolio,current)
        elif month == last_month:
            current,portfolio = sell(top_df,month_df,portfolio,current,'YES')
            final_profit = round(100*(current - s_sum)/s_sum,3)
            profits_dict[(int(month[:4])-1)] = final_profit
        else:
            current,portfolio = sell(top_df,month_df,portfolio,current)
            if current <=0:
                print(f'Ran out of cash on {month[:4]}')
                return None
            if month[-2:]=='01':
                pfl_value = get_portfolio_value(month_df,portfolio)
                annual_profit = round(100*(current + pfl_value - s_sum)/s_sum,3)
                profits_dict[(int(month[:4])-1)] = annual_profit
            current,portfolio = buy(top_df,portfolio,current)

    return profits_dict
```

The second part of the function still looks quite similar to the train_model function, only that after the first year we calculate both free amount of money and stocks value to see the aggregate profit from the previos year.

To calculate the portfolio value we use the get_portfolio_value function.

If we run out of cash after selling the stocks, the simulation ends.

We save the yearly aggregate profit in a dict.

3.10.1.6. Get_Portfolio_Value function:

```
def get_portfolio_value(df,portfolio):
    portfolio_value = 0
    for ticker in list(portfolio.keys()):
        if portfolio.get(ticker)>0:
            try:
                ticker_price = float(df.loc[df['Short_Ticker'] == ticker, 'Adj_Close'].iloc[0]
            except Exception:
                ticker_price = 0
            try:
                ticker_value = float(ticker_price*portfolio.get(ticker))
            except Exception:
                ticker_value = 0
            portfolio_value += ticker_value
    return portfolio_value
```

This function receives the monthly df and the portfolio.
For each ticker in the portfolio, the function calculates the value of the stocks according to the Adj Close price known in the day of the calculation. We add the value of each value into a final sum (portfolio_value), which the function returns.

4. Results and Comparison to market indices

4.1. I ran the simulation function on the Test database (2015-2020).

4.2. Then, I compared the aggregate profit of each year with the performance of the following market indices:

4.2.1.S&P500
4.2.2.Russell1000
4.2.3.Rusell2000
4.2.4.DOW
4.2.5.NASDAQ

4.3. The results are as follows (Yearly Agg. Yield % since 01-01-2015):

My Program	RUI^	DJI^	IXIC^	RUT^	GSPC^	
4.481-	1.091-	2.233-	5.730	5.712-	0.727-	**2015**
14.940	8.502	10.882	13.663	12.653	8.739	**2016**
49.799	29.487	38.692	45.763	27.460	29.856	**2017**
37.173	20.963	30.884	40.102	11.942	21.757	**2018**
123.199	55.912	60.121	89.453	38.497	56.918	**2019**
264.996	85.331	71.724	172.131	63.930	82.431	**2020**

01/03/2021
Bar Dahan

5. What did not work out?
   5.1. In the process of creating my program, I tried to use machine learning and regressions.
   5.2. I tried to categorize the stocks by the monthly yield as profit(yield>0), loss(yield<0) and neutral(yield=0), and run a logistic regression on the types of sentiments. The algorithm could not classify the stocks, with the highest score being 57%.
   5.3. I also tried to train the model with linear regression as a tool to determine the factors' coefficients. Normalized Volume's coefficient wasn't significant at all so I decided to go with Pearson instead.
   5.4. Although I started the project with 2,005 tickers, yfinance provided data for just 1,437 stocks (71.67%).