Stock portfolia performance Dataset analysis

The dataset of performances of weighted scoring stock portfolios are obtained with mixture design from the US stock market historical database.

Dataset source (UCI Machine Learning Repository): https://archive.ics.uci.edu/ml/datasets/Stock+portfolio+performance
Dataset download link, direct link: stock portfolio performance data set.xlsx

Data Set Characteristics:	Multivariate	**Number of Instances:**	315	**Area:**	Business
Attribute Characteristics:	Real	**Number of Attributes:**	12	**Date Donated**	2016-04-22
Associated Tasks:	Regression	**Missing Values?**	N/A	**Number of Web Hits:**	82017

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/455d1a47-a00a-4e75-8677-51d893d3f2ed/stock_portfolio_performance_data_set.xlsx

If we look at excel file, we will see the following:

About Dataset – Stock Portfolio Perform

Dataset distribution (six sheets of excel file) –

- Sheet 1 – 4^{th} period (65 rows, 63 Ids or stocks and 19 columns).
- Sheet 2 – 3^{rd} period (65 rows, 63 Ids or stocks and 19 columns).
- Sheet 3 – 2^{nd} period (65 rows, 63 Ids or stocks and 19 columns).
- Sheet 4 – 1^{st} period (65 rows, 63 Ids or stocks and 19 columns).
- Sheet 5 – all period (65 rows, 63 Ids or stocks and 19 columns).
- Sheet 6 – Time frame (5 rows and 5 columns).

Data columns names (3 subsets – the weight of the stock-picking concept, the original investment performance indicator and the normalized investment performance indicator) (only in 4^{th}, 3^{rd}, 2^{nd}, 1^{st} and all period sheets) –

'ID', ' Large B/P ', ' Large ROE ', ' Large S/P ', ' Large Return Rate in the last quarter ', ' Large Market Value ', ' Small systematic Risk', 'Annual Return', 'Excess Return', 'Systematic Risk', 'Total Risk', 'Abs. Win Rate', 'Rel. Win Rate', 'Annual Return' (normalized), 'Excess Return' (normalized), 'Systematic Risk' (normalized), 'Total Risk' (normalized), 'Abs. Win Rate' (normalized), and 'Rel. Win Rate' (normalized).

Code info:

Used anaconda jupyter to do data analysis.

---

checking versions

checking versions

```
In [2]: !python --version   # Python version

        # Load Module ---
        import numpy as np, pandas as pd

        print('numpy version:',np.__version__)
        print('pandas version: ',pd.__version__)
```

```
Python 3.7.10
numpy version: 1.19.5
pandas version:  1.1.5
```

pandas,numpy warmup

```
In [3]: # Genrate one dimension data - Series
        one_d_data=np.random.rand(6) #  uniform distribution (in the range [0,1))
        print('one_d_data:', one_d_data)
        # pandas series
        pd_series=pd.Series(data=one_d_data,index=None,dtype=None,name=None,copy=False)
        pd_series
```

```
        one_d_data: [0.1292109  0.15204455 0.33103917 0.56314728 0.17482357 0.4902248 ]

Out[3]: 0    0.129211
        1    0.152045
        2    0.331039
        3    0.563147
        4    0.174824
        5    0.490225
        dtype: float64
```

we used numpy to generate random numbers and pandas to make it in series(as given in output)

Generating 2d random data frames

```
[4]: # Genrate two dimension data - dataframe (rows and columns)
two_d_data=np.random.randn(11,6) # normal distribution (random variable with a Gaussian distribution)
print('two_d_data:\n',two_d_data,sep='')
```

```
two_d_data:
[[ 0.03024512  0.50511375  1.24899931  0.20928989 -0.47490977 -0.89977549]
 [ 2.26637449 -0.35148803  0.25534478 -0.72811915 -0.69917574  1.53179337]
 [ 0.41398662  0.53985547  0.42538685  0.87461099  1.2956849  -1.44956502]
 [ 1.03998284  1.264679    0.89577547  1.45059898 -0.40823682 -0.55174932]
 [-1.00260081 -0.53435822  0.59312139  2.23561696 -0.19932813  0.14625376]
 [-0.99450457 -0.99283831 -0.68039512 -1.08610794  0.50637957 -0.10168937]
 [ 0.82489574  1.11154294  1.23895392 -0.40004446 -0.78221198 -0.19062221]
 [ 1.67141209  0.80761513 -0.23664303  0.7821994   1.15189517 -0.0620839 ]
 [ 0.88567916 -0.06809755 -1.46609073  1.13948543 -1.09704302 -1.21928878]
 [-1.71483322 -1.75610579 -0.55671873  1.27918576 -0.94455069  1.53517421]
 [ 0.69872964  1.49396406  0.14250355  0.67135143 -1.14379916  1.05199055]]
```

Then, i created dataframes using pandas

```
]: # pandas dataframe
pd_dataframe=pd.DataFrame(data=two_d_data,index=['a','b','c','d','e',1,2,3,4,5,6],columns=range(6),
                          dtype=None,copy=False)
pd_dataframe.head() # returns top five values
```

]:

	0	1	2	3	4	5
a	0.030245	0.505114	1.248999	0.209290	-0.474910	-0.899775
b	2.266374	-0.351488	0.255345	-0.728119	-0.699176	1.531793
c	0.413987	0.539855	0.425387	0.874611	1.295685	-1.449565
d	1.039983	1.264679	0.895775	1.450599	-0.408237	-0.551749
e	-1.002601	-0.534358	0.593121	2.235617	-0.199328	0.146254

here, head() gives the initial values of data.

In [9]:
```
# Info
pd_dataframe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 11 entries, a to 6
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       11 non-null     float64
 1   1       11 non-null     float64
 2   2       11 non-null     float64
 3   3       11 non-null     float64
 4   4       11 non-null     float64
 5   5       11 non-null     float64
dtypes: float64(6)
memory usage: 616.0+ bytes
```

now we move on to get info where we can see data has non null values and datatypes is float.

In [13]:
```
# Stats
pd_dataframe.describe()
```

Out[13]:

	0	1	2	3	4	5
count	11.000000	11.000000	11.000000	11.000000	11.000000	11.000000
mean	0.374488	0.183626	0.169113	0.584370	-0.254118	-0.019051
std	1.205375	1.018732	0.847312	1.001085	0.865352	1.028576
min	-1.714833	-1.756106	-1.466091	-1.086108	-1.143799	-1.449565
25%	-0.482130	-0.442923	-0.396681	-0.095377	-0.863381	-0.725762
50%	0.698730	0.505114	0.255345	0.782199	-0.474910	-0.101689
75%	0.962831	0.959579	0.744448	1.209336	0.153526	0.599122
max	2.266374	1.493964	1.248999	2.235617	1.295685	1.535174

Then we check the stats of datasets where we can see the mean,25% ,50%, & 75% of datasets

```
In [15]: # Null values
         pd_dataframe.isna().sum()

Out[15]: 0    0
         1    0
         2    0
         3    0
         4    0
         5    0
         dtype: int64
```

then we check null values on how many null values do we have.

Lets move to our main dataset.

Dataset download link: stock portfolio performance data set.xlsx

```
[41]: #!wget https://archive.ics.uci.edu/ml/machine-learning-databases/00390/stock%20portfolio%20performance%

      # Read dataset
      data_file_link_xlsx='https://archive.ics.uci.edu/ml/machine-learning-databases/00390/stock%20portfolio%
      data=pd.read_excel(data_file_link_xlsx,sheet_name='4th period',header=1)
```

reading the data

```
In [45]: data.rename(index=None,columns=column_dict,axis=None,copy=True,inplace=True)
         data.tail()
```

Out[45]:

	id	bp_large	roe_large	sp_large	ror_large_last_quarter	mv_large	systematic_risk_small	annual_return	e:
58	59	0.200	0.200	0.200	0.000	0.200	0.200	0.034682	
59	60	0.200	0.200	0.000	0.200	0.200	0.200	0.033733	
60	61	0.200	0.000	0.200	0.200	0.200	0.200	0.044852	
61	62	0.000	0.200	0.200	0.200	0.200	0.200	0.040456	
62	63	0.167	0.167	0.167	0.167	0.167	0.167	0.057510	

rename the columns in dataset

```
In [51]: # stats - Transpose
         data.describe().T
```

Out[51]:

	count	mean	std	min	25%	50%	75%	max
id	63.0	32.000000	18.330303	1.000000	16.500000	32.000000	47.500000	63.000000
bp_large	63.0	0.166619	0.199304	0.000000	0.000000	0.167000	0.291500	1.000000
roe_large	63.0	0.166619	0.199304	0.000000	0.000000	0.167000	0.291500	1.000000
sp_large	63.0	0.166619	0.199304	0.000000	0.000000	0.167000	0.291500	1.000000
ror_large_last_quarter	63.0	0.166619	0.199304	0.000000	0.000000	0.167000	0.291500	1.000000
mv_large	63.0	0.166619	0.199304	0.000000	0.000000	0.167000	0.291500	1.000000
systematic_risk_small	63.0	0.166619	0.199304	0.000000	0.000000	0.167000	0.291500	1.000000
annual_return	63.0	0.040384	0.028337	-0.053382	0.021405	0.042629	0.061776	0.098369
excess_return	63.0	0.010196	0.007972	-0.014856	0.004378	0.010413	0.015840	0.026548

we can transpose the dataset using .t where the rows and columns will interchange.

2.0.2 sort

Lets move into sorting where we sort our datasets.

```
In [52]: # sort columns by names
         data.reindex(sorted(data.columns,reverse=False),axis='columns').head(3) # or axis=0; inplace -> X
         # pandas.DataFrame.sort_index -> https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_ind
```

Out[52]:

	abs_win_rate	abs_win_rate_norm	annual_return	annual_return_norm	bp_large	excess_return	excess_return_r
0	0.60	0.68	0.019516	0.488229	1.0	0.013399	0.
1	0.55	0.56	0.023829	0.505279	0.0	0.006410	0.
2	0.55	0.56	0.080282	0.728484	0.0	0.026548	0.

we sort the columns using names I.E using capital letters a,b,c,d...

```
]: # sort by column
data.sort_values(['bp_large','roe_large'],axis=0,ascending=True,inplace=False,kind='quicksort',na_posit
# for more look - https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html
```

	id	bp_large	roe_large	sp_large	ror_large_last_quarter	mv_large	systematic_risk_small	annual_return	e:
31	32	0.333	0.333	0.0	0.0	0.0	0.333	0.016851	
7	8	0.500	0.000	0.5	0.0	0.0	0.000	0.061700	
9	10	0.500	0.000	0.0	0.5	0.0	0.000	0.068515	
12	13	0.500	0.000	0.0	0.0	0.5	0.000	0.025587	
16	17	0.500	0.000	0.0	0.0	0.0	0.500	-0.023439	
6	7	0.500	0.500	0.0	0.0	0.0	0.000	0.061851	
0	1	1.000	0.000	0.0	0.0	0.0	0.000	0.019516	

next we sort using columns , where we specify the columns names and sorting algo. Here its quicksort.

Lets come to ml part

3.0.12 data normalization

```
In [72]: # import function
from sklearn.preprocessing import normalize
# doc -> https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html#skle
# axis -> # 0:column; 1:rows
# l1 -> value/sum(respective row or column) --> norm_value -> sum axis wise
# l2 -> value/sqrt(sum of square of each element wise of respective row or column))  --> norm_value -
# 'max' -> value/max(respective row or column) --> norm_value -> maximum value of axis
```

Lets normalize the data.

Note: Machine learning algorithms tend to perform better or converge faster when the different features (variables) are on a smaller scale. Therefore it is common practice to normalize the data before training machine learning models on it.

```
# Method 1 - direct
data_small_norm_matrix,_=normalize(data_small[data_small.columns[1:]].values,norm='l2',axis=0,copy=True
data_small_norm=pd.DataFrame(data=data_small_norm_matrix,index=None,columns=data_small.columns[1:],
                             dtype=None,copy=True)
#data_small_norm['id']=data_small.id
data_small_norm.head()
```

	roe_large	sp_large	ror_large_last_quarter	mv_large	systematic_risk_small
0	0.000000	0.000000	0.000000	0.000000	0.0
1	0.487267	0.000000	0.000000	0.000000	0.0
2	0.000000	0.487267	0.000000	0.000000	0.0
3	0.000000	0.000000	0.487267	0.000000	0.0
4	0.000000	0.000000	0.000000	0.487267	0.0

we normalize the dataset here and get the data which we want as an input, you can see in output of the code - these are columns which are inputs

```
In [63]: # data x
         x=data_small_norm.to_numpy()
```

Now we set our x after normalizing the inputs value.

```
In [64]: # label y
         y=data.annual_return_norm.to_numpy()
```

and set y as our outputs.

```
In [65]: from sklearn.model_selection import train_test_split

         x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.1)

In [66]: from sklearn.linear_model import LinearRegression

         reg=LinearRegression()

In [67]: reg.fit(x_train,y_train)

Out[67]: LinearRegression()
```

now we use regression model to fit the test data (we use linear regression here)

```
In [68]: y_pred=reg.predict(x_test)

In [69]: y_pred,y_test

Out[69]: (array([0.6354012 , 0.55441832, 0.53990086, 0.82834902, 0.5622464 ,
                 0.52041136, 0.58125288]),
          array([0.59792744, 0.59355751, 0.49273836, 0.72848434, 0.59234009,
                 0.54443801, 0.60937582]))
```

we do predict the output now.

```
                 0.54443801, 0.60937582]))

In [70]: sum((y_pred-y_test)**2)

Out[70]: 0.017407222699613572

In [ ]:
```

Now we calculate total mean squared error .

note: we have put all input datas as

roe_large	sp_large	ror_large_last_quarter	mv_large	systematic_risk_small

and output as annual return.

```
y_pred,y_test

(array([0.6354012 , 0.55441832, 0.53990086, 0.82834902, 0.5622464 ,
        0.52041136, 0.58125288]),
 array([0.59792744, 0.59355751, 0.49273836, 0.72848434, 0.59234009,
        0.54443801, 0.60937582]))
```

as you can see what our model predicated and what is the actual number

Now we will go for DAY@2

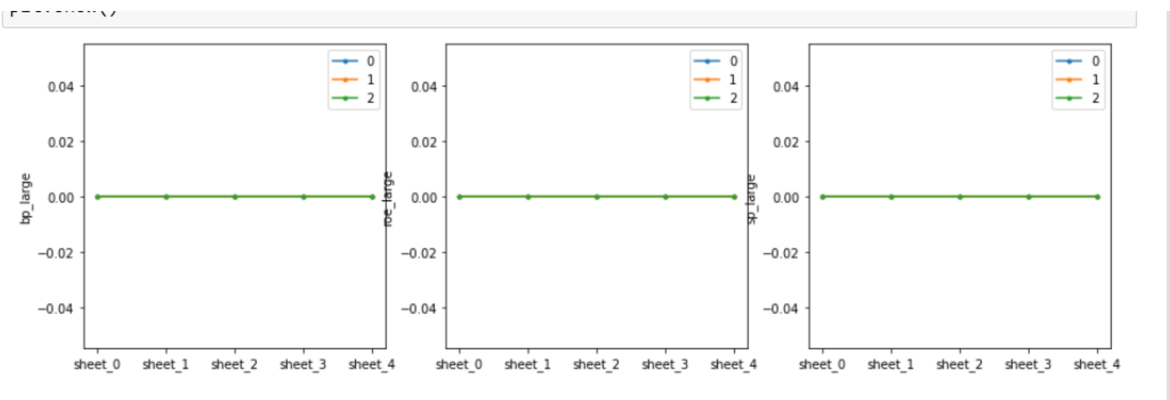
1.0.1 long term

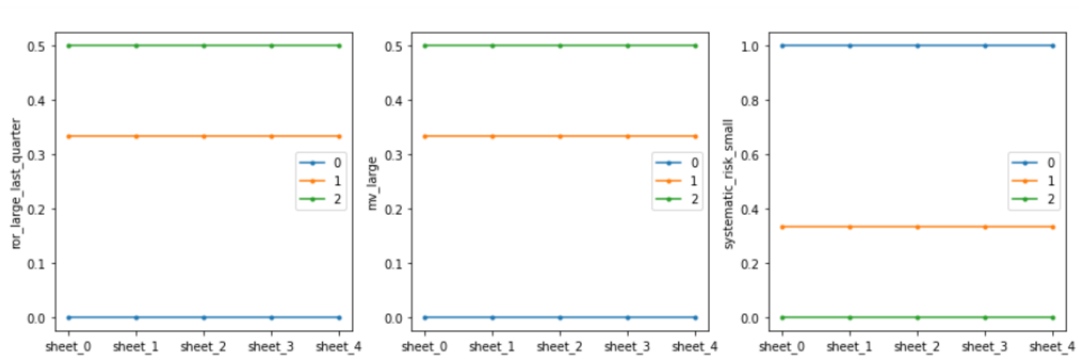
In [19]:
```
# See down trends - top three from bottom
ltm_worse_three_idx=data3d.loc['sheet_4'].sort_values('annual_return',ascending=True).index[:3]
ltm_worse_three=data3d.loc['sheet_4'].sort_values('annual_return',ascending=True).id.values[:3]
print('worse_three',ltm_worse_three)

# See down trends - top three from top
ltm_best_three_idx=data3d.loc['sheet_4'].sort_values('annual_return',ascending=False).index[:3]
ltm_best_three=data3d.loc['sheet_4'].sort_values('annual_return',ascending=False).id.values[:3]
print('best_three',ltm_best_three)

data3d.loc[mId['sheet_4',ltm_worse_three_idx],data3d.columns[:10]]
```

```
worse_three [ 6 41 16]
best_three [ 7 22 42]
```

Here we see long term stocks which are good to buy or add in portfolio

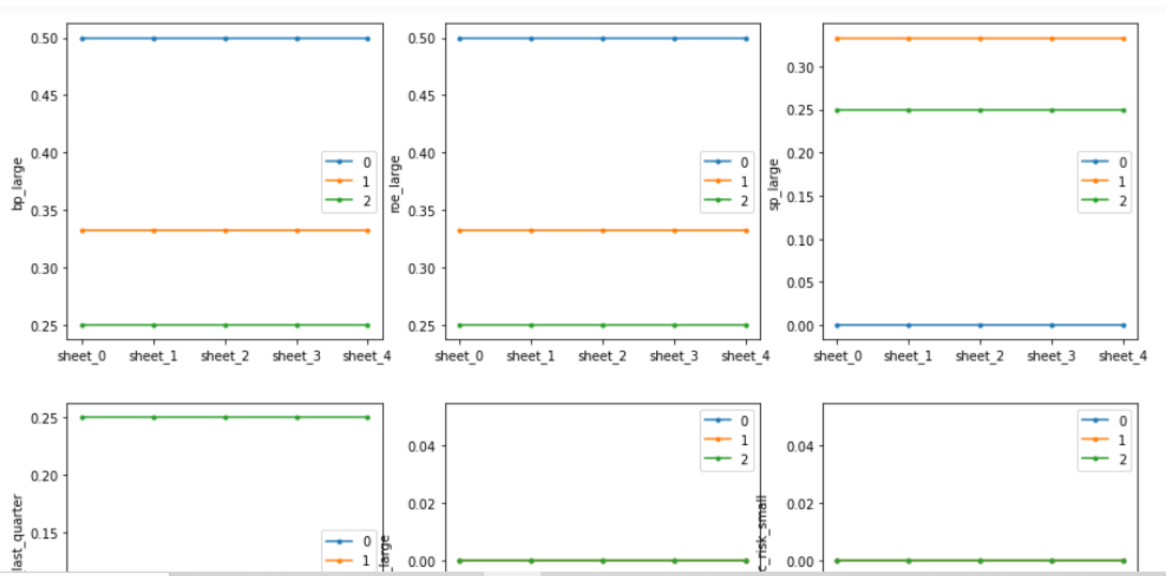
In [6]:
```
# plot 20 years data for worse three
plt.figure(figsize=(15,10))

# plot trends
# x - axis
x_axis=['sheet_'+str(x) for x in range(total_sheets-1)]

for n_plot,column in enumerate(data3d.columns[1:7],1):
  plt.subplot(2,3,n_plot)
  # y - axis
  y_axis_all=[data3d.loc[pd.IndexSlice[:,ltm_worse_three_idx[y]],column].values for y in range(3)]
  # plot
  for n,y_axis in enumerate(y_axis_all,0):
    plt.plot(x_axis,y_axis,label=str(n),marker='.')
  plt.legend(),plt.ylabel(column)

plt.show()
```





here in graph we can see best 3 and worst 3 stocks.

```
# plot 20 years data for best three
plt.figure(figsize=(15,10))

# plot trends
# x - axis
x_axis=['sheet_'+str(x) for x in range(total_sheets-1)]
#
for n_plot,column in enumerate(data3d.columns[1:7],1):
  plt.subplot(2,3,n_plot)
  # y - axis
  y_axis_all=[data3d.loc[pd.IndexSlice[:,ltm_best_three_idx[y]],column].values for y in range(3)]
  # plot
  for n,y_axis in enumerate(y_axis_all,0):
    plt.plot(x_axis,y_axis,label=str(n),marker='.')
  plt.legend(),plt.ylabel(column)

plt.show()
```



Similary short term investments

short term

In [8]:
```
# See down trends - top three from bottom
stm_worse_three_idx=data3d.sort_values('annual_return',ascending=True).index[:3]
stm_worse_three=data3d.sort_values('annual_return',ascending=True).id.values[:3]
print('worse_three',stm_worse_three)

# See down trends - top three from top
stm_best_three_idx=data3d.sort_values('annual_return',ascending=False).index[:3]
stm_best_three=data3d.sort_values('annual_return',ascending=False).id.values[:3]
print('best_three',stm_best_three)

data3d.loc[mId[stm_worse_three_idx],data3d.columns[:10]]
```

```
worse_three [ 6 17 16]
best_three [22  7  3]
```

1.0.3 sort term, minumum risk

n [10]:
```
data3d.sort_values(['annual_return','systemtic_risk_actual'],ascending=False).head()
```

ut[10]:

		id	bp_large	roe_large	sp_large	ror_large_last_quarter	mv_large	systematic_risk_small	annual_re
sheet_1	21	22	0.333	0.333	0.333	0.0	0.0	0.0	0.:
	6	7	0.500	0.500	0.000	0.0	0.0	0.0	0.2
	2	3	0.000	0.000	1.000	0.0	0.0	0.0	0.2
sheet_3	6	7	0.500	0.500	0.000	0.0	0.0	0.0	0.2
	21	22	0.333	0.333	0.333	0.0	0.0	0.0	0.2

Lets take a look at short term and minimum risks.

Now we will do the predication for long term investment model.

1.0.4 predection - long term

In [11]:
```
data3d.columns
```

Out[11]:
```
Index(['id', 'bp_large', 'roe_large', 'sp_large', 'ror_large_last_quarter',
       'mv_large', 'systematic_risk_small', 'annual_return', 'excess_return',
       'systemtic_risk_actual', 'total_risk', 'abs_win_rate',
       'relative_win_rate', 'annual_return_norm', 'excess_return_norm',
       'systemtic_risk_actual_norm', 'total_risk_norm', 'abs_win_rate_norm',
       'relative_win_rate_norm'],
      dtype='object')
```

here we will take some inputs, say first seven columns as an inputs

1.0.4 predection - long term

```
In [11]: data3d.columns

Out[11]: Index(['id', 'bp_large', 'roe_large', 'sp_large', 'ror_large_last_quarter',
               'mv_large', 'systematic_risk_small', 'annual_return', 'excess_return',
               'systemtic_risk_actual', 'total_risk', 'abs_win_rate',
               'relative_win_rate', 'annual_return_norm', 'excess_return_norm',
               'systemtic_risk_actual_norm', 'total_risk_norm', 'abs_win_rate_norm',
               'relative_win_rate_norm'],
              dtype='object')
```

1.0.5 get data

```
In [12]: # data (input) ['bp_large','roe_large','sp_large','ror_large_last_quarter','mv_large','systematic_risk_
         in_names=['bp_large','roe_large','sp_large','ror_large_last_quarter','mv_large','systematic_risk_small'
         X=data3d.loc[mId['sheet_4',:],data3d.columns[1:7]].values  # sort term - X=data3d.loc[mId['sheet_0':'sh

         # labels (to predict) ['annual_return','excess_return','total_risk']
         outs_name=['annual_return','excess_return','total_risk']
         y=data3d.loc[mId['sheet_4',:],['annual_return','excess_return','total_risk']].values # sort term - y=da

         X.shape,y.shape
         #output--6 rows given in in_names

Out[12]: ((63, 6), (63, 3))
```

in get data we took the output as aanual return,excess return and total risk. and we redicated number of rows and columns for input and output for the same.

```
In [13]: # import function
         from sklearn.preprocessing import normalize
         # doc -> https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html#sklear

         X_norm,norms_of_x=normalize(X,norm='l1',axis=0,copy=True,return_norm=True)
         y_norm,norms_of_y=normalize(y,norm='l1',axis=0,copy=True,return_norm=True)

         # split
         from sklearn.model_selection import train_test_split
         # Source: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.ht
         nX_train,nX_test,ny_train,ny_test=train_test_split(X_norm,y_norm,test_size=0.10,random_state=7)

         # split - un-norm data
         X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.10,random_state=7)

         # shape
         nX_train.shape,nX_test.shape,ny_train.shape,ny_test.shape
```

we split the data ,normalize it

(here n stands for normalized data)

1.0.7.1 multiple outputs

In [14]:
```
# linear model for multiple outputs - norm data
from sklearn.linear_model import LinearRegression
# Source: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

# make model for multiple outputs
nliner_multi=LinearRegression()
# to know more - https://github.com/scikit-learn/scikit-learn/blob/15a949460/sklearn/linear_model/_base
# train model
nliner_multi.fit(nX_train,ny_train)
# print coef_
print('coef_ of shape:',nliner_multi.coef_.shape,': value -\n',nliner_multi.coef_)
# print intercept_
print('intercept_ of shape:',nliner_multi.intercept_.shape,': value -\n',nliner_multi.intercept_)#6*3 =
```

NOW we will check the multiple outputs and single output model.

here in multiple output model nlinear_multi we gave as name where n is normalizied

```
coef_ of shape: (3, 6) : value -
 [[ 2.202696    2.23229594  2.24975041  2.15741387  2.13026079  2.12529582]
 [ 5.91621291  5.97400974  6.010513    5.78611649  5.73625632  5.75602156]
 [-0.31908885 -0.37532361 -0.3165433  -0.32982325 -0.39759677 -0.36089127]]
intercept_ of shape: (3,) : value -
 [-0.19191142 -0.54251533  0.04914567]
```

here is the output where we used linear regression in multiple outputs so our equation will be y=m1x1+m2x2+m3x3. so we will get 18(6*3) outputs and 3 y value.

1.0.7.2 single outputs

In [15]:
```
# make model for single outputs - norm data
nliner_single=LinearRegression()

# train model
nliner_single.fit(nX_train,ny_train[:,0])
# print coef_
print('coef_ of shape:',nliner_single.coef_.shape,': value -\n',nliner_single.coef_)
# print intercept_
print('intercept_ of shape:',nliner_single.intercept_.shape,': value -\n',nliner_single.intercept_)


# make model for single outputs - un-norm data
liner_single=LinearRegression()
# train model
liner_single.fit(X_train,y_train[:,0])
```

```
coef_ of shape: (6,) : value -
 [2.202696   2.23229594 2.24975041 2.15741387 2.13026079 2.12529582]
intercept_ of shape: () : value -
 -0.19191141657389413
```

```
5]: LinearRegression()
```

Here we can see the output is same from multiple out.

```
[16]: # error - mean absolute error
from sklearn.metrics import mean_absolute_error
# source: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html

print('error multi-outputs norm:',mean_absolute_error(ny_test,nliner_multi.predict(nX_test),multioutput
print('error single-outputs norm:',mean_absolute_error(ny_test[:,0],nliner_single.predict(nX_test))*nor
print('error single-outputs un-norm:',mean_absolute_error(y_test[:,0],liner_single.predict(X_test)))
print('absolute difference:',(ny_test[:,0]-nliner_single.predict(nX_test))*norms_of_y[0])
```

```
error multi-outputs norm: [0.01927095 0.005284   0.00775093]
error single-outputs norm: 0.019270951142947867
error single-outputs un-norm: 0.019270951142947516
absolute difference: [-0.02585069 -0.03771539  0.01296044 -0.02185498  0.01766048  0.00254028
 -0.01631441]
```

Now we will compare the mean error of both models(single vs multiple).we can see its same.

and we got absolute difference which tells difference of actual and predicated of each row.

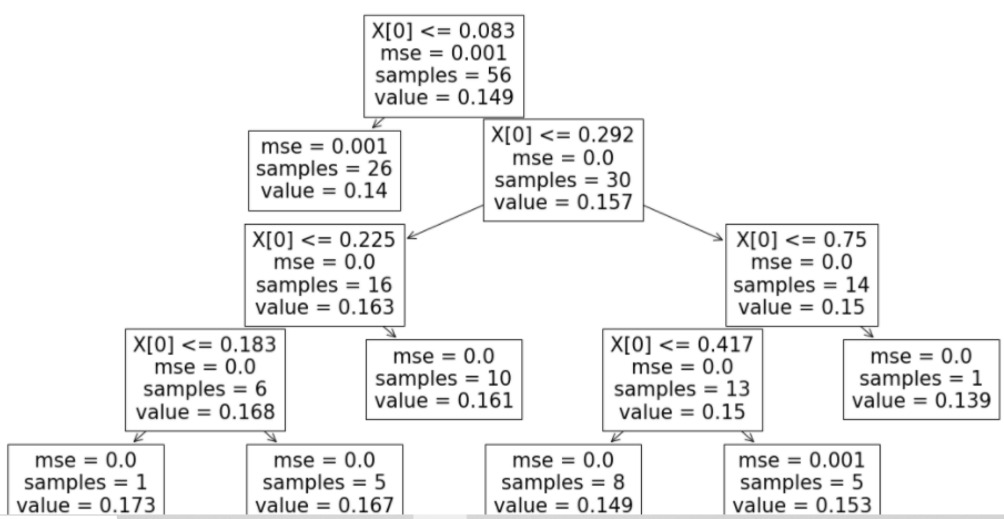
Now we go on trees, decision tree

```
]: # make model for single output
from sklearn.tree import DecisionTreeRegressor
# source: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

# calculations
# MSE = (sum_square_of_left / w_l) + (sum_square_of_right / w_r)
# FriedmanMSE = (w_r * total_left_sum - w_l * total_rigth_sum)**2 / (w_r * w_l)

# make model - un-norm
tree_single=DecisionTreeRegressor(criterion='friedman_mse',random_state=7)#friedman_mse- making model j
# train model
tree_single.fit(X_train,y_train[:,0])
print('error single-outputs un-norm:',mean_absolute_error(y_test[:,0],tree_single.predict(X_test)))

# make model - norm
ntree_single=DecisionTreeRegressor(criterion='friedman_mse',random_state=7)
# train model
ntree_single.fit(nX_train,ny_train[:,0])
print('error single-outputs norm',mean_absolute_error(ny_test[:,0],ntree_single.predict(nX_test))*norms
```

```
ntree_single=DecisionTreeRegressor(criterion='friedman_mse',random_state=7)
# train model
ntree_single.fit(nX_train,ny_train[:,0])
print('error single-outputs norm',mean_absolute_error(ny_test[:,0],ntree_single.predict(nX_test))*norms
```

```
error single-outputs un-norm: 0.011428571428571423
error single-outputs norm 0.009142857142857128
```

```
[19]: # make model for single output
from sklearn.tree import plot_tree

plt.figure(figsize=(20,10))
sample_tree_single=DecisionTreeRegressor(max_depth=7).fit(X_train[:,0].reshape(-1,1),y_train[:,0])
plot_tree(sample_tree_single)
plt.show()
```



comparing between tree and linear regression

1.0.10 tree vs linear

In [20]:
```
print('error linear single-outputs un-norm:',mean_absolute_error(y_test[:,0],liner_single.predict(X_tes
print('error tree single-outputs un-norm:',mean_absolute_error(y_test[:,0],tree_single.predict(X_test))
print('absolute difference linear:',y_test[:,0]-liner_single.predict(X_test))
print('absolute difference tree:',y_test[:,0]-tree_single.predict(X_test))
print('absolute difference linear-tree:',liner_single.predict(X_test)-tree_single.predict(X_test))
```

```
error linear single-outputs un-norm: 0.019270951142947516
error tree single-outputs un-norm: 0.011428571428571423
absolute difference linear: [-0.02585069 -0.03771539  0.01296044 -0.02185498  0.01766048  0.00254028
 -0.01631441]
absolute difference tree: [-0.003  0.003  0.019  0.008 -0.025  0.008 -0.014]
absolute difference linear-tree: [ 0.02285069  0.04071539  0.00603956  0.02985498 -0.04266048  0.0054
5972
  0.00231441]
```

here decision tree has less error so it is a good model for the same.

Improving trees

we will use gradientboosting technique and randomforest regressor

2]:
```
# make model for single output - Gradient Boosting
from sklearn.ensemble import GradientBoostingRegressor
# source: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.

# make model - un-norm
gboost_tree_single=GradientBoostingRegressor(learning_rate=0.1,criterion='friedman_mse')
# train model
gboost_tree_single.fit(X_train,y_train[:,0])
print('error Gradient Boosting Regressor-outputs un-norm:',mean_absolute_error(y_test[:,0],gboost_tree_
```

here gradient boosting allows multiple model where one model learns from previous and with learning rate given in code

```
# make model - un-norm
gboost_tree_single=GradientBoostingRegressor(learning_rate=0.1,criterion='friedman_mse')
# train model
gboost_tree_single.fit(X_train,y_train[:,0])
print('error Gradient Boosting Regressor-outputs un-norm:',mean_absolute_error(y_test[:,0],gboost_tree_

# make model for single output -
from sklearn.ensemble import RandomForestRegressor
# source: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

# make model - un-norm
forest_single=RandomForestRegressor(criterion='friedman_mse')
# train model
forest_single.fit(X_train,y_train[:,0])
print('error Forest Regressor-outputs un-norm:',mean_absolute_error(y_test[:,0],forest_single.predict(X
```

This is random forest regressor code

```
error Forest Regressor-outputs un-norm: 0.012475714285714297
```

```
3]: print('error Gradient Boosting Regressor-outputs un-norm:',mean_absolute_error(y_test[:,0],gboost_tree_
print('error tree single-outputs un-norm:',mean_absolute_error(y_test[:,0],tree_single.predict(X_test))
print('error linear single-outputs un-norm:',mean_absolute_error(y_test[:,0],liner_single.predict(X_tes
print('Gradient Boosting Regressor score:',gboost_tree_single.score(X_test,y_test[:,0]))
print('tree score:',tree_single.score(X_test,y_test[:,0]))
print('linear score:',liner_single.score(X_test,y_test[:,0]))
```

```
error Gradient Boosting Regressor-outputs un-norm: 0.007529135019854578
error tree single-outputs un-norm: 0.011428571428571423
error linear single-outputs un-norm: 0.019270951142947516
Gradient Boosting Regressor score: 0.9057719335358216
tree score: 0.811187390827477
linear score: 0.5278509157028655
```

checking the score of different models:

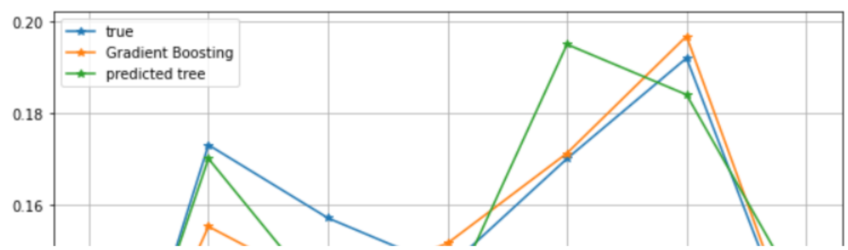
Score qualty 0<score<1

1- overfitting

0 - bad

the one which highest score wins

```
In [24]: # plot
plt.figure(figsize=(10,7))
plt.plot(range(y_test.shape[0]),y_test[:,0],marker='*',label='true')
plt.plot(range(y_test.shape[0]),gboost_tree_single.predict(X_test),marker='*',label='Gradient Boosting'
plt.plot(range(y_test.shape[0]),tree_single.predict(X_test),marker='*',label='predicted tree')
plt.grid(),plt.legend()
plt.show()
```



Here, the plots of the same.

Now we will go with the real datas

2 get me the real ---

source: https://datahub.io/core/s-and-p-500-companies-financials

In [25]:
```
# data source
data_link='https://datahub.io/core/s-and-p-500-companies-financials/r/constituents-financials.csv'

# load data
snp_500=pd.read_csv(data_link)
snp_500.head(3)
#EBITDA, or earnings before interest, taxes, depreciation, and amortization
```

In [26]:
```
snp_500.columns
```

Out[26]:
```
Index(['Symbol', 'Name', 'Sector', 'Price', 'Price/Earnings', 'Dividend Yield',
       'Earnings/Share', '52 Week Low', '52 Week High', 'Market Cap', 'EBITDA',
       'Price/Sales', 'Price/Book', 'SEC Filings'],
      dtype='object')
```

columns

click to scroll output; double click to hide

In [27]:
```
rename_dict_500={'Symbol':'symbol','Name':'name','Sector':'sector',
                 'Price':'price','Price/Earnings':'profit','Dividend Yield':'divident',
                 'Earnings/Share':'earning_share','52 Week Low':'low','52 Week High':'high',
                 'Market Cap':'market_cap','EBITDA':'ebitda','Price/Sales':'price_sales',
                 'Price/Book':'book_price','SEC Filings':'links'}

# rename columns
mdf_snp_500=snp_500.rename(columns=rename_dict_500)
# drop - name,links
mdf_snp_500.drop(columns=['name','links'],inplace=True)
mdf_snp_500.head(3)
```

Out[27]:

we are renaming columns.

Out[27]:

	symbol	sector	price	profit	divident	earning_share	low	high	market_cap	ebitda	price_sales
0	MMM	Industrials	222.89	24.31	2.332862	7.92	259.77	175.490	138721055226	9.048000e+09	4.390
1	AOS	Industrials	60.24	27.76	1.147959	1.70	68.39	48.925	10783419933	6.010000e+08	3.575
2	ABT	Health Care	56.27	22.51	1.908982	0.26	64.60	42.280	102121042306	5.744000e+09	3.740

#EBITDA, or earnings before interest, taxes, depreciation, and amortization

and drop null values

Lets do predication now

In [37]:
```python
# x-data - ['low','high','market_cap','ebitda','price_sales','book_price']
X_500=sample_data.loc[:,mdf_snp_500.columns[-6:-4]].values
# y-labels - ['price','profit']
y_500=sample_data.loc[:,['price','profit']].values

# normalize
X_500_norm,norms_of_x_500=normalize(X_500,norm='l2',axis=0,copy=True,return_norm=True)
y_500_norm,norms_of_y_500=normalize(y_500,norm='l2',axis=0,copy=True,return_norm=True)

# split data
X_train_500,X_test_500,y_train_500,y_test_500=train_test_split(X_500_norm,y_500_norm,test_size=0.10,rar

X_train_500.shape,X_test_500.shape,y_train_500.shape,y_test_500.shape
```

8]:
```python
# make model - un-norm
gboost_tree_500=GradientBoostingRegressor(learning_rate=0.05,criterion='mse',random_state=10)
# train model
gboost_tree_500.fit(X_train_500,y_train_500[:,1])
print('error Gradient Boosting Regressor-outputs un-norm:',
      mean_absolute_error(y_test_500[:,1],gboost_tree_500.predict(X_test_500),multioutput='raw_values')

# scoure https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html
from sklearn.metrics import mean_squared_error

print('error Gradient Boosting Regressor-outputs un-norm:',
      mean_squared_error(y_test_500[:,1],gboost_tree_500.predict(X_test_500),multioutput='raw_values')*
```

gradient boosting

:
```python
from sklearn.feature_selection import RFE

sample_tester_rfe=GradientBoostingRegressor()

f_selector=RFE(sample_tester_rfe,n_features_to_select=2,step=1)

f_selector.fit(X_500_4_cv,y_500_4_cv[:,1])
```

Check similar models on the same page.

Super model test- feature selecting models.

:
```python
# x-data - ['low','high','market_cap','ebitda','price_sales','book_price']
X_500=sample_data.loc[:,mdf_snp_500.columns[-6:]].values
# y-labels - ['price','profit']
y_500=sample_data.loc[:,['price','profit']].values

# normalize
X_500_4_cv,norms_of_x_500=normalize(X_500,norm='l2',axis=0,copy=True,return_norm=True)
y_500_4_cv,norms_of_y_500=normalize(y_500,norm='l2',axis=0,copy=True,return_norm=True)
```

here firstly we took inputs - X , outputs -Y and normalized the data.

```
from sklearn.model_selection import cross_validate,KFold,GridSearchCV
from sklearn.feature_selection import RFE


# creating a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 100)

# specify range of hyperparameters
hyper_params = [{'n_features_to_select': list(range(1, 7))}]


# Test classifier - GradientBoostingRegressor
test_clf_GradientBoostingRegressor=GradientBoostingRegressor(learning_rate=0.05,criterion='mse')
test_clf_GradientBoostingRegressor.fit(X_500_4_cv,y_500_4_cv[:,1])

# set up GridSearchCV()
model_f_select = GridSearchCV(estimator = RFE(test_clf_GradientBoostingRegressor),param_grid = hyper_pa
                              scoring= 'r2', cv = folds, verbose = 1, return_train_score=True)

# fit the model
model_f_select.fit(X_500_4_cv,y_500_4_cv[:,1])
```

then we took **cross_validate,KFold,GridSearchCV**

and we import ***RFE*** which is used for feature selecting

```
from sklearn.model_selection import cross_validate,KFold,GridSearchCV
from sklearn.feature_selection import RFE
```

Kfold breaks the datasets into trainsets and test sets-basically split our dataset into5 different ways.

```
# creating a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 100)

# specify range of hyperparameters
hyper_params = [{'n_features_to_select': list(range(1, 7))}]
```

now the hyperparameter does is take all features - say 1st it takes one then 2 and then 3.

```
# Test classifier - GradientBoostingRegressor
test_clf_GradientBoostingRegressor=GradientBoostingRegressor(learning_rate=0.05,criterion='mse')
test_clf_GradientBoostingRegressor.fit(X_500_4_cv,y_500_4_cv[:,1])

# set up GridSearchCV()
model_f_select = GridSearchCV(estimator = RFE(test_clf_GradientBoostingRegressor),param_grid = hyper_pa
                              scoring= 'r2', cv = folds, verbose = 1, return_train_score=True)

# fit the model
model_f_select.fit(X_500_4_cv,y_500_4_cv[:,1])
```

we will train our model here using gradientboostingregressor.

```
results_
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_features_to_select	params
0	0.248634	0.007160	0.000206	0.000412	1	{'n_features_to_select': 1}
1	0.214942	0.003096	0.001010	0.002021	2	{'n_features_to_select': 2}
2	0.183767	0.006097	0.000200	0.000399	3	{'n_features_to_select': 3}
3	0.149872	0.003750	0.001319	0.001679	4	{'n_features_to_select': 4}
4	0.103332	0.003746	0.001407	0.001871	5	{'n_features_to_select': 5}
5	0.050095	0.003012	0.002911	0.003834	6	{'n_features_to_select': 6}

in output we get different features .

note:

an_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_features_to_select	params	split0_t
0.248634	0.007160	0.000206	0.000412	1	{'n_features_to_select': 1}	
0.214942	0.003096	0.001010	0.002021	2	{'n_features_to_select': 2}	
0.183767	0.006097	0.000200	0.000399	3	{'n_features_to_select': 3}	
0.149872	0.003750	0.001319	0.001679	4	{'n_features_to_select': 4}	
0.103332	0.003746	0.001407	0.001871	5	{'n_features_to_select': 5}	
0.050095	0.003012	0.002911	0.003834	6	{'n_features_to_select': 6}	

x 21 columns

here we get 2 feature select- we will select from first 2 features not random feature.