Machine Learning

Lecture 2

We can now begin to set up training, validation and test data and train a decision tree on our data.

We will first set up training data consisting of a number of years of data, using the relative return as a way to make labels. We then try out the trained model on the validation set.

We aim to maximize the performance on the validation set.

When we are satisfied with the validation performance we run the model on the test set

Loading the Data Set (you need to put in the file where you have stored the data)

```
raw_data = pd.read_pickle(r'C:\Users\niels\OneDrive\Machine Learning 2022\Lecture 2\dataset.pkl')
```

```
raw_data.columns
```

[3]:
```
Index(['actq', 'apq', 'atq', 'ceqq', 'cheq', 'cogsq', 'csh12q', 'cshfdq',
       'cshiq', 'cshopq',
       ...
       'sector_code_815.0', 'sector_code_817.0', 'sector_code_822.0',
       'sector_code_823.0', 'sector_code_825.0', 'sector_code_830.0',
       'sector_code_835.0', 'sector_code_840.0', 'sector_code_845.0',
       'sector_code_850.0'],
      dtype='object', length=731)
```

```
raw_data = raw_data.drop([x  for x in raw_data.columns if 'fqtr' in x],axis=1)
```

Restricting to Companies with Market Cap > 1 Billion

```
data = raw_data[raw_data['market_cap'] > 1000.0]
```

```python
data['pred_rel_return']
```

```
5]: date        ticker
    2000-02-09  CSCO     -0.025923
                ROP       0.066175
    2000-02-10  CMOS      0.241345
    2000-02-11  DELL      0.306035
    2000-02-15  VAL       0.043852
                             ...
    2018-12-21  NKE            NaN
                SAFM           NaN
                SCHL           NaN
                WBA            NaN
    2018-12-24  KMX            NaN
    Name: pred_rel_return, Length: 111468, dtype: float64
```

The Total Number of Companies w/ Market Cap > 1 Billion that appear during our time horizon

```python
len(data.index.get_level_values(1).unique())
```

```
7]: 4076
```

Filling in Missing Values

```
data = data.copy()
data.replace([np.inf,-np.inf],np.nan,inplace=True)
data = data.fillna(method='ffill')
```

```
data = data.fillna(0)
```

We label a Data Point +1 if the difference between the return on the SPY and the return on the stock exceeds 1.0% during the earnings period, -1 if it is < -2.5% and 0 if it is between -1% and +1%. The function below turns the return differences into labels

```
def f(x):
    if x > 0.01:
        return 1
    elif x < -0.025:
        return -1
    else:
        return 0
```

Applying the function to the column of relative returns and making a column of labels

```
data = data.copy()
data['rel_performance'] = data['pred_rel_return'].apply(f)
```

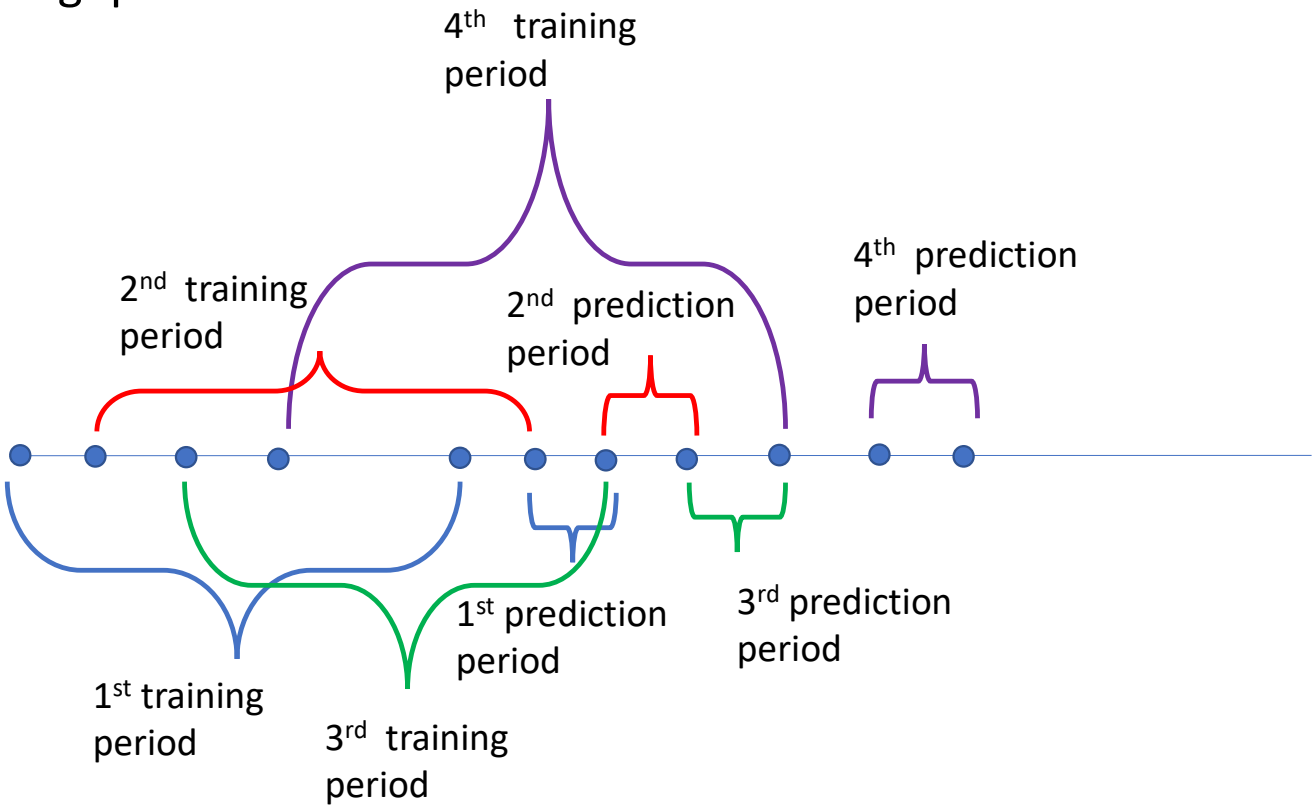
```python
data['rel_performance']
```

```
date        ticker
2000-02-09  CSCO     -1
            ROP       1
2000-02-10  CMOS      1
2000-02-11  DELL      1
2000-02-15  VAL       1
2000-02-16  AMAT      1
            ANF      -1
            DE        1
            EV        1
            NAV      -1
```

The strategy is going to work as follows:

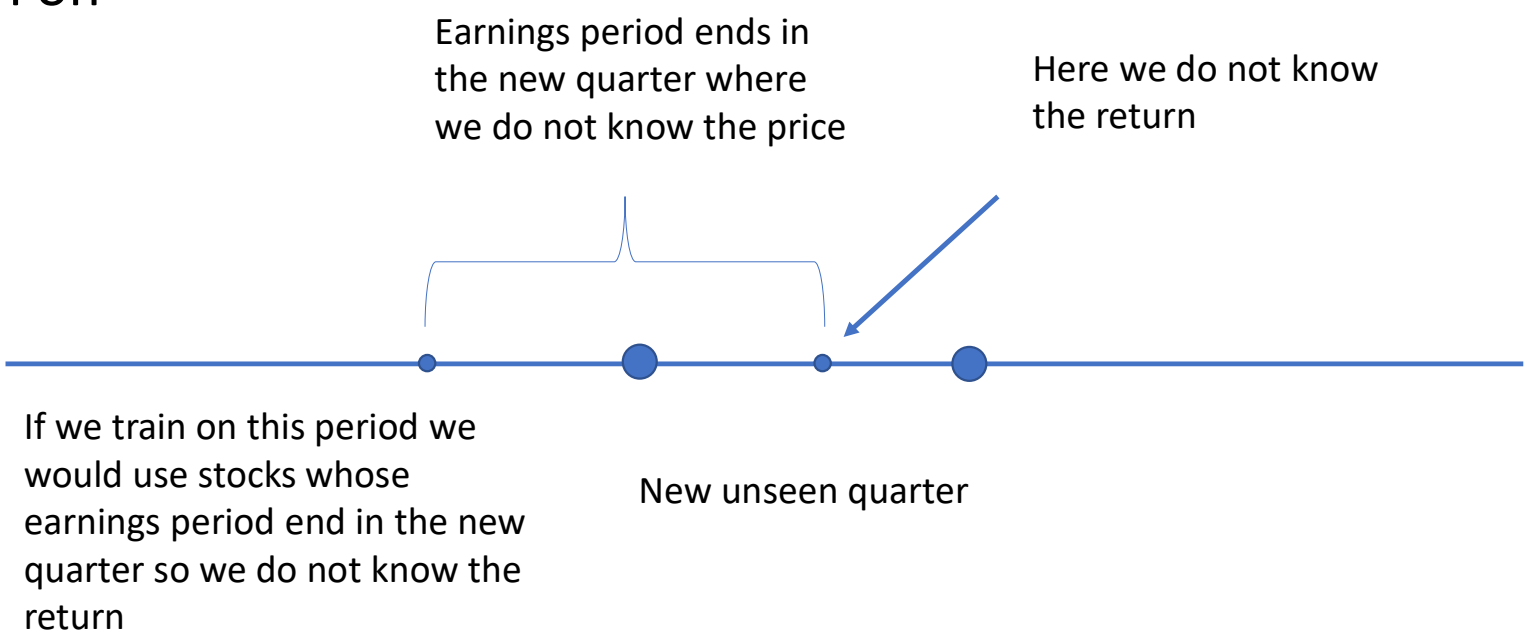
We train on a period of 3 years and then predict on the quarter starting 3 months after the end date of the training period

Then we move the 3 year period forward 1 quarter and train again and predict on the following quarter etc.



Why are we skipping a quarter?

This is because when we train we use the returns for a period that ends in the quarter immediately following the training period and so when we test on an unseen quarter this information would not be available to train on



During each prediction period we use the model that has been trained on the data in the preceding training period. During the prediction period, when a company releases quarterly earnings we use the trained model to tell us whether to buy, sell or not do anything.

We use a training period of 3 years = 12 quarters,

```
df_train = data.loc['2000-01-01':'2003-01-01']
```

The validation set and test set are the quarters starting 1 quarter after the end of the training periods

```
df_valid = data.loc['2003-04-01':'2003-07-01']
df_test = data.loc['2003-07-01':'2003-10-01']
```

Next we have the two prediction periods

```
df_valid = data.loc['2012-04-01':'2012-07-01']
df_test = data.loc['2018-07-01':'2018-10-01']
```

and the labels for the prediction periods

```
y_1 = df_1['rel_performance'].values
y_2 = df_2['rel_performance'].values
```

```
Counter(y_1)
```

```
5]: Counter({-1: 15542, 1: 15974, 0: 2888})
```

The data we train the model on, should not contain any features that use information not revealed before the end of the training period. This includes next_period_return, spy_next_period_return,rel_return,cum_return,spy_cum_return. We also leave out the ticker symbol and the date.

Next we delete the columns that are not needed for training

```
train = df_train.reset_index().drop(['ticker','date',
                                     'next_period_return',
                                     'spy_next_period_return',
                                     'rel_performance','pred_rel_return',
                                    'return', 'cum_ret', 'spy_cum_ret'],axis=1)

valid = df_valid.reset_index().drop(['ticker','date',
                                     'next_period_return',
                                     'spy_next_period_return',
                                     'rel_performance','pred_rel_return',
                                    'return', 'cum_ret', 'spy_cum_ret',
                                     ],axis=1)
test = df_test.reset_index().drop(['ticker','date',
                                     'next_period_return',
                                     'spy_next_period_return',
                                     'rel_performance','pred_rel_return',
                                    'return', 'cum_ret', 'spy_cum_ret',
                              ],axis=1)
```

We take out the actual earnings period stock returns in the training and test sets

```
train_stock_returns = df_train['next_period_return']
valid_stock_returns = df_valid['next_period_return']
test_stock_returns = df_test['next_period_return']
```

Next we want to normalize the data by centering i.e. subtracting the mean and making the standard deviation = 1. We can use sklearn's StandardScaler. But we only want to normalize the data that are floating point numbers (the other ate categorical variables encoded as 1-hot encoded vectors)

```
from sklearn.preprocessing import MinMaxScaler,StandardScaler
```

```
scaler = StandardScaler()
```

```
float_vars = [x for x in train.columns if data[x].dtype == 'float64']
```

```
len(float_vars)
```

179

```
train = train.copy()
valid = valid.copy()
test = test.copy()
```

```
train[float_vars] = scaler.fit_transform(train[float_vars])
valid[float_vars] = scaler.transform(valid[float_vars])
test[float_vars] = scaler.transform(test[float_vars])
```

We are now ready to start training our first model.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score,confusion_matrix
```

We instantiate a tree classifier and train it on the train data,

```
t_clf = DecisionTreeClassifier(min_samples_leaf=600,max_depth=8,random_state=123)
```

```
t_clf.fit(train_1,y_1)
```

```
.]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=8,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=600, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=123,
            splitter='best')
```

the number of features is

Number of Features

```
len(train.columns)
```

```
86]: 721
```

The large number of features is due to the one-hot-encoding of the categorical variables which associate to each category a new column consisting of only 0s and 1s.

```
from sklearn.tree import DecisionTreeClassifier,DecisionTreeRegressor
from sklearn.metrics import accuracy_score,confusion_matrix
```

```
t_clf = DecisionTreeClassifier(min_samples_leaf = 100,max_depth=15)
```

```
t_clf.fit(train,y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=15, min_samples_leaf=100)
```

Fraction of correct predictions

```
accuracy_score(y_train,t_clf.predict(train))
```

```
0.6111162404210138
```

```
accuracy_score(y_valid,t_clf.predict(valid))
```

```
0.5099567099567099
```

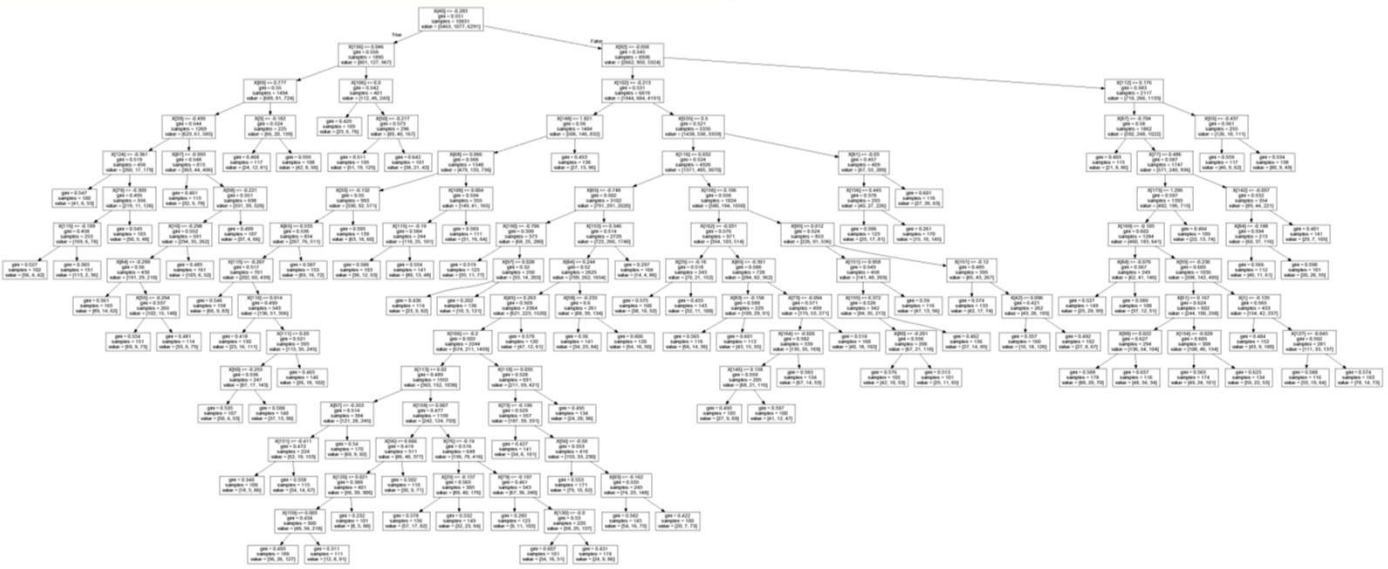
The large difference between the performance on the training set and the performance on the validation set indicates ‘over-fitting’

Visualizing the tree ¶

```python
import graphviz
from sklearn import tree
```

```python
dot_data = tree.export_graphviz(t_clf,out_file=None)
graph = graphviz.Source(dot_data)
graph.render('tree_10')
```

[4]: 'tree_10.pdf'



We randomly selected the hyperparameters max-depth and min-samples-leaf. The min-samples-leaf = n ensures that every leaf contains at least n samples. This is to lower over-fitting where the trained tree fits very well on the training set but does note perform well on datasets it has not been trained on i.e. it does not predict well.

The hyperparameters can be set arbitrarily but how do we find the ‘best’ i.e. those that maximize a certain measure, like the accuracy_score.

We shall use the package ‘Optuna’ to optimize hyperparameters. We don’t care so much about accuracy as we care about the profits we make by following the predictions of the model.

We estimate the profits computing the return on a portfolio, where we buy $1 worth of the stock if the prediction is +1 and -$1 (i.e. short $1 worth of the stock) if the prediction is -1 and doing nothing if the prediction is 0.

We can very simply compute this by multiplying the array of predictions with the array of the actual returns and summing them up

```
profit = (preds * val_rets).sum()
```

To use Optuna we first have to install it with `!pip install optuna` and import it

```
import optuna
from optuna.trial import Trial
```

We then have to make an objective function

```
def objective(trial:Trial,train=None,labels=None,val=None,val_labels=None,val_rets=None):

    t_min_samples_leaf = trial.suggest_int('min_samples_leaf',100,1200,step=100)
    t_max_depth = trial.suggest_int('max_depth',5,25,step=5)


    t_clf = DecisionTreeClassifier(min_samples_leaf = t_min_samples_leaf,max_depth=t_max_depth,random_state=123)
    t_clf.fit(train,labels)

    preds = t_clf.predict(val)
    profit = (preds * val_rets).sum()

    return profit

```

These are the various combinations of hyperparameters we will try

There are 240 different combinations so we might run 240 trials to find the combination that gives the best profit on the validation set. This is not necessary with optuna since it has advanced algorithms that selects promising paths that seems to lead to better values, rather then trying every combination. So instead we just run 100 trials

```
study = optuna.create_study(direction="maximize")
```

```
[I 2023-01-09 10:08:58,766] A new study created in memory with name: no-name-a2abfd1c-466b-4e98-b6e5-80ee35725f3d
```

```
from functools import partial
```

```
%%time
study.optimize(partial(objective,train=train,labels=y_train,val=valid,val_labels=y_valid,val_rets=valid_stock_r
```

```
ax_depth : 25}. Best is trial 16 with value: 4.020640000000023.
[I 2023-01-09 10:09:26,158] Trial 96 finished with value: -26.053565999999996 and parameters: {'min_samples_leaf':
'max_depth': 15}. Best is trial 16 with value: 4.020640000000023.
[I 2023-01-09 10:09:26,309] Trial 92 finished with value: -2.4917439999999758 and parameters: {'min_samples_leaf':
ax_depth': 25}. Best is trial 16 with value: 4.020640000000023.
[I 2023-01-09 10:09:26,349] Trial 93 finished with value: -2.4917439999999758 and parameters: {'min_samples_leaf':
ax_depth': 15}. Best is trial 16 with value: 4.020640000000023.
[I 2023-01-09 10:09:26,886] Trial 94 finished with value: -2.4917439999999758 and parameters: {'min_samples_leaf':
ax_depth': 20}. Best is trial 16 with value: 4.020640000000023.
```

We find the best combination of hyperparameters

```
study.best_params
```

```
{'min_samples_leaf': 300, 'max_depth': 5}
```

And now we can instantiate and train a tree with these hyperparameters

Instantiating the classifier with hyper-parameters

```
t_clf = DecisionTreeClassifier(**study.best_params,random_state=123)
```

Now we can fit the tree with the training data

```
t_clf.fit(train,y_train)
```

5]:

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=15, min_samples_leaf=300, random_state=123)
```

Now the performance on the training and validation sets are much closer so the overfitting has been greatly reduced

and see how well it fits the training data

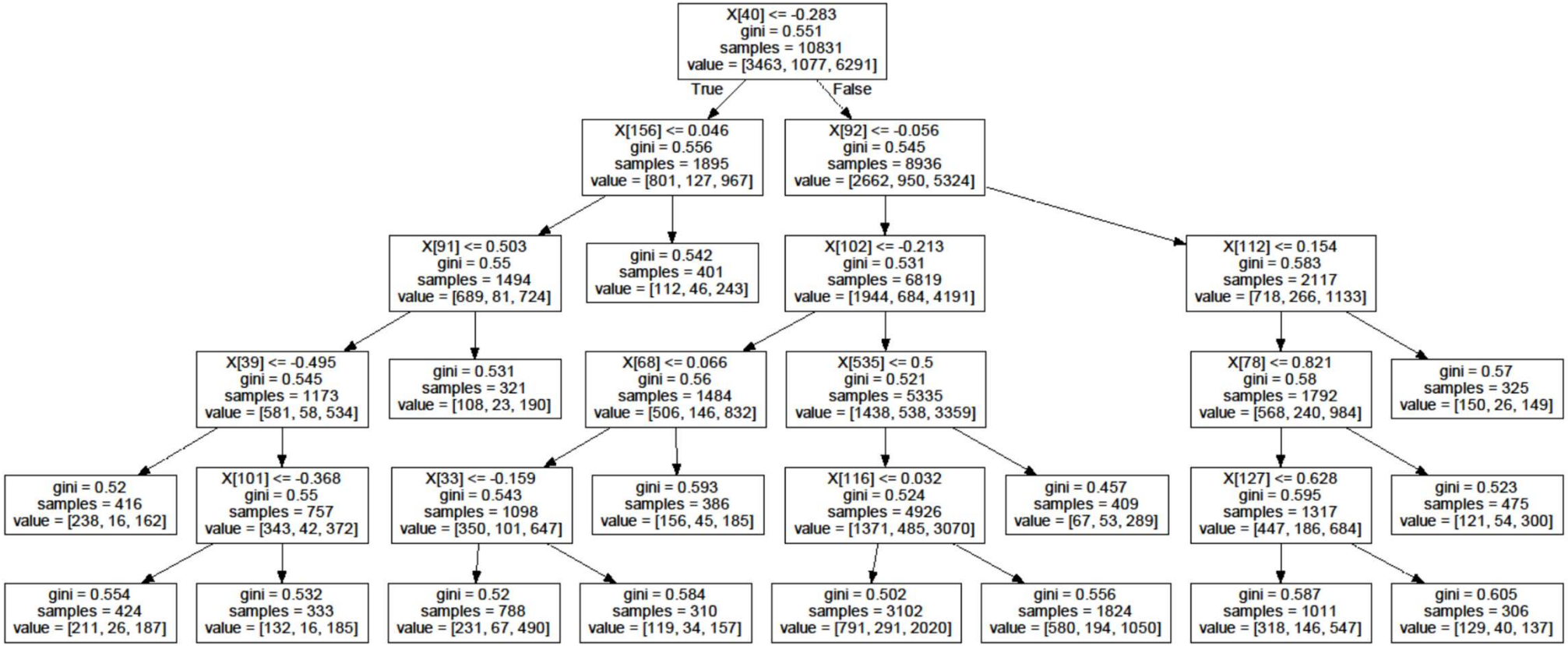
```
t_clf.score(train,y_train)
```

```
[6]: 0.5901578801588034
```

and predicts the validation data ¶

```
t_clf.score(valid,y_valid)
```

```
[7]: 0.5844155844155844
```


X[40] <= -0.283
gini = 0.551
samples = 10831
value = [3463, 1077, 6291]
True
False
X[156] <= 0.046
gini = 0.556
samples = 1895
value = [801, 127, 967]
X[92] <= -0.056
gini = 0.545
samples = 8936
value = [2662, 950, 5324]
X[91] <= 0.503
gini = 0.55
samples = 1494
value = [689, 81, 724]
gini = 0.542
samples = 401
value = [112, 46, 243]
X[102] <= -0.213
gini = 0.531
samples = 6819
value = [1944, 684, 4191]
X[112] <= 0.154
gini = 0.583
samples = 2117
value = [718, 266, 1133]
X[39] <= -0.495
gini = 0.545
samples = 1173
value = [581, 58, 534]
gini = 0.531
samples = 321
value = [108, 23, 190]
X[68] <= 0.066
gini = 0.56
samples = 1484
value = [506, 146, 832]
X[535] <= 0.5
gini = 0.521
samples = 5335
value = [1438, 538, 3359]
X[78] <= 0.821
gini = 0.58
samples = 1792
value = [568, 240, 984]
gini = 0.57
samples = 325
value = [150, 26, 149]
gini = 0.52
samples = 416
value = [238, 16, 162]
X[101] <= -0.368
gini = 0.55
samples = 757
value = [343, 42, 372]
X[33] <= -0.159
gini = 0.543
samples = 1098
value = [350, 101, 647]
gini = 0.593
samples = 386
value = [156, 45, 185]
X[116] <= 0.032
gini = 0.524
samples = 4926
value = [1371, 485, 3070]
gini = 0.457
samples = 409
value = [67, 53, 289]
X[127] <= 0.628
gini = 0.595
samples = 1317
value = [447, 186, 684]
gini = 0.523
samples = 475
value = [121, 54, 300]
gini = 0.554
samples = 424
value = [211, 26, 187]
gini = 0.532
samples = 333
value = [132, 16, 185]
gini = 0.52
samples = 788
value = [231, 67, 490]
gini = 0.584
samples = 310
value = [119, 34, 157]
gini = 0.502
samples = 3102
value = [791, 291, 2020]
gini = 0.556
samples = 1824
value = [580, 194, 1050]
gini = 0.587
samples = 1011
value = [318, 146, 547]
gini = 0.605
samples = 306
value = [129, 40, 137]

First we are going to reduce the number of features.

Clearly not all of the 721 features appear in the tree. Those features that do not appear have no influence on the performance and so we can eliminate them.

We can compute how important a given feature is in the tree by looking at the nodes in the tree where the split is on the feature and look at how much the split reduces the Gini index. The sum of these is the feature importance.

We compute the importance of each feature

```python
def tree_feat_importance(m, df):
    return pd.DataFrame({'cols':df.columns, 'feat_imp':m.feature_importances_}
                        ).sort_values('feat_imp', ascending=False)

def plot_fi(fi): return fi.plot('cols', 'feat_imp', 'barh', figsize=(12,7), legend=False)
```

```python
fi = tree_feat_importance(t_clf,train)
```

```python
fi
```

40	oiadpq	0.246802
156	fcf_yield	0.105154
92	market_cap	0.103065
102	oancfy_q	0.074014
116	cfmq	0.072963
...	...	...
250	sic_2673	0.000000
251	sic_2711	0.000000
252	sic_2721	0.000000
253	sic_2731	0.000000
720	sector_code_850.0	0.000000

721 rows × 2 columns

And eliminate all those that have feature importance = 0, meaning that they don't appear in the tree at all

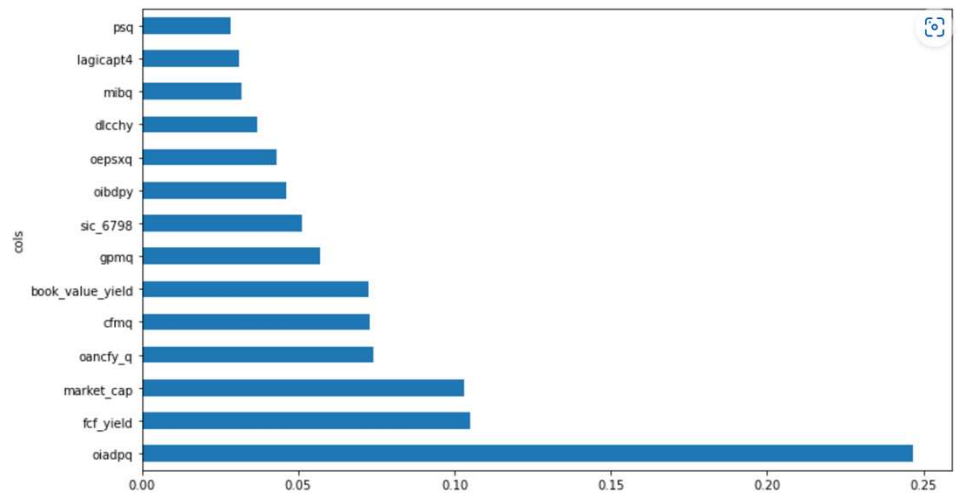
```
features = fi[(fi['feat_imp'] > 0.00)]
```

```
features
```

	cols	feat_imp
40	oiadpq	0.246802
156	fcf_yield	0.105154
92	market_cap	0.103065
102	oancfy_q	0.074014
116	cfmq	0.072963
91	book_value_yield	0.072320
112	gpmq	0.056935
535	sic_6798	0.051326
78	oibdpy	0.046291
39	oepsxq	0.043120
68	dlcchy	0.036894
33	mibq	0.031844
127	lagicapt4	0.030897
101	psq	0.028373

Feature importance spectrum

```
plot_fi(features);
```



We are left with a much smaller set of features

```
cols = features['cols'].values
```

```
len(cols)
```

```
93]: 14
```

We cut down the training and validation data sets to only include the relevant features and retrain the tree on the reduced data set

Remark that we have to re-normalize the new data set

```
train_red = pd.DataFrame(data = scaler.fit_transform(train[cols].values),columns = cols)
valid_red = pd.DataFrame(data = scaler.transform(valid[cols].values),columns = cols)
test_red = pd.DataFrame(data = scaler.transform(test[cols].values),columns = cols)
```

```
t_clf.fit(train_red,y_train)
```

6]:

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=5, min_samples_leaf=300, random_state=123)
```

```
t_clf.score(train_red,y_train)
```

7]: 0.5901578801588034

```
t_clf.score(valid_red,y_valid)
```

8]: 0.5844155844155844

The tree is precisely the same

A subject that is becoming increasingly important is the notion of ‘Explainable’ or ‘Interpretable’ Machine Learning models.

Instead of a model being a ‘black-box’ that just makes predictions, we want to be able to figure out how it arrives at the predictions and how much weight it puts on each feature. We also want to know how changing the value of a feature effects the prediction.

For instance for a simple linear regression model

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_n X_n$$

the size of the coefficients (the β‘s) indicate how each feature effects the output

A way to explain the effect of is to use ‘Shapley’ values. The definition of Shapley values is rather complicated but in essence it computes the effect on a adding a feature to any subset of the dataset and also taking into account the order in which the features are added.

The Shapley values are computed for each feature and for each symbol in the validation set

The Shapley values are computed on a trained module. We need to have a model that computes the profit for each symbol in the validation set using the prediction of the model. This requires that we include the actual returns in the model so we add the returns column to the validation set

```
valid_1 = valid_red.copy()

valid_1['rets'] = df_valid['next_period_return'].values
```

```
import shap
```

```
def model(features):
    tree_features = features[features.columns[:-1].values]

    pred = t_clf.predict(tree_features)

    ret = pred * features[features.columns[-1]]

    return ret
```

The predictions of the model does not use the actual returns so we remove that column when we use the tree to predict values

This is the array of the predicted return of each symbol

```
explainer = shap.explainers.Permutation(model,valid_1)
```

```
shap_values = explainer(valid_1,max_evals=2000)
```

```
Permutation explainer: 1156it [03:51,  4.73it/s]
```

```
shap_values
```

```
.values =
array([[-2.79437672e-03, -2.11024219e-03, -6.48359531e-04,
         0.00000000e+00, -2.52859125e-03,  1.43178815e-01],
       [-3.85787734e-03,  1.24468750e-04,  8.30313266e-03,
         0.00000000e+00, -4.78539656e-03, -5.37845084e-02],
       [-2.61572594e-03, -3.03927703e-03,  5.32072922e-03,
         0.00000000e+00, -2.94981266e-03,  1.67141716e-02],
       ...,
       [ 3.62267656e-03, -1.27358562e-03,  1.63853172e-03
```

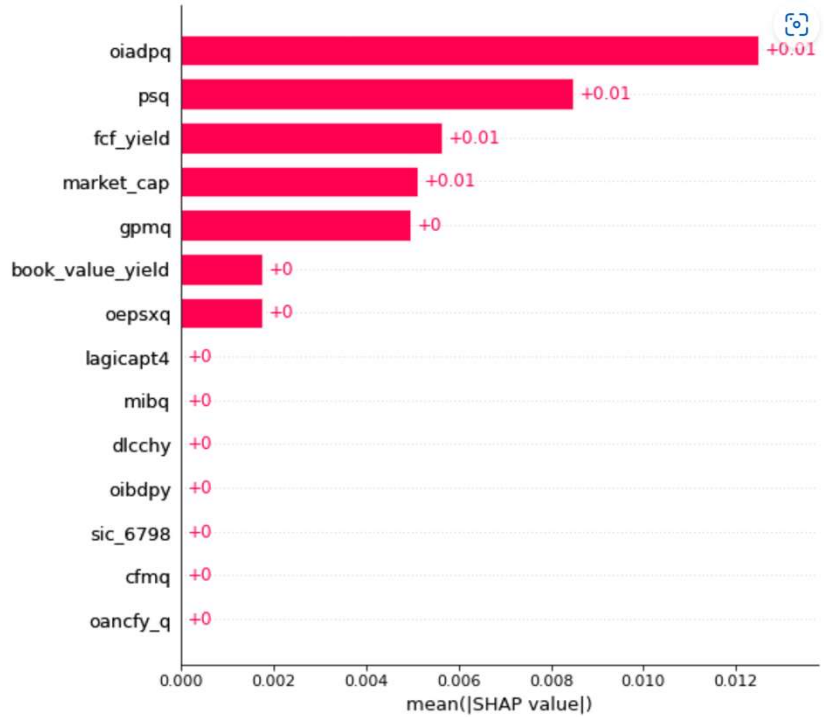
```
shap_values.values.shape
```

```
(1155, 15)
```

There is a Shapley value for each feature and each symbol

The ‘shap’ package contains a lot of tools to visualize the Shapley values. We can graph the average value of the absolute values of the Shapley values.

```
shap.plots.bar(shap_values[:,:-1],max_display=30)
```



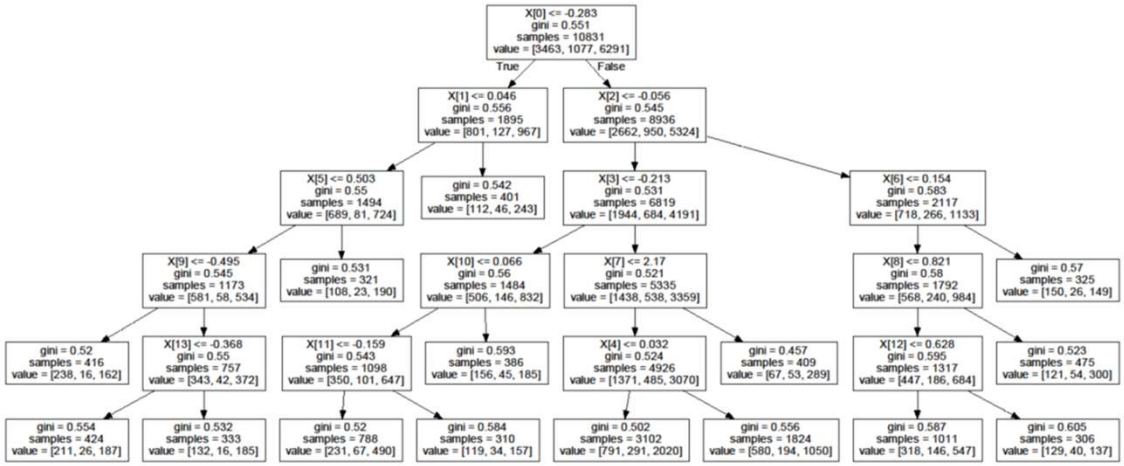
This shows that only the first 7 features have an effect and so we can further cut down our feature set.

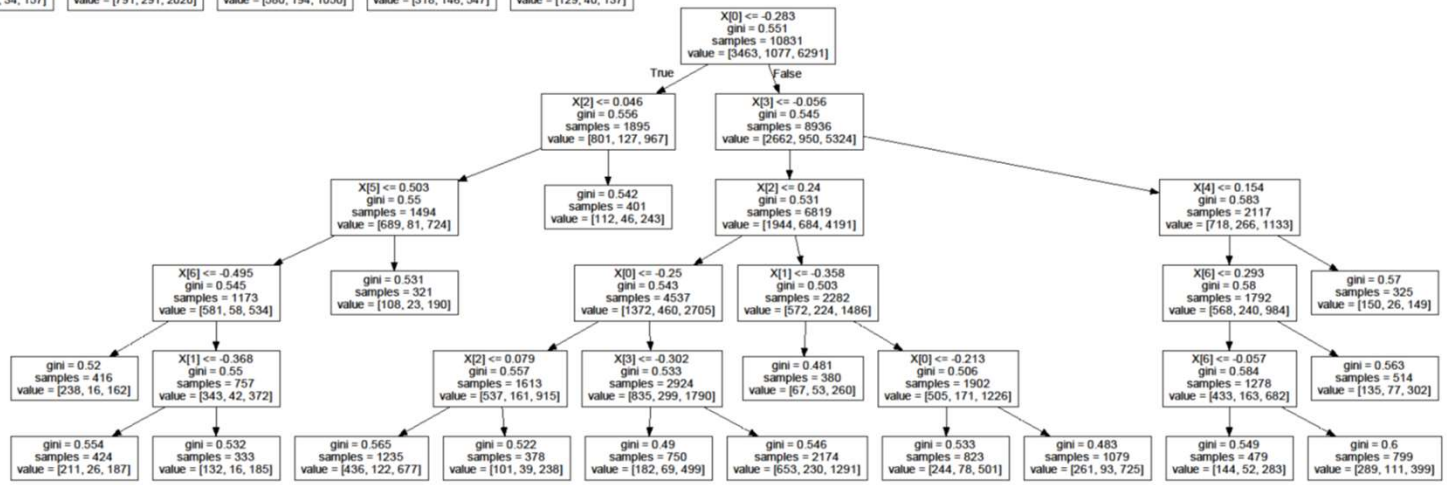
So we can make a new tree and train it only on these features

```
t_clf1 = DecisionTreeClassifier(**study.best_params,random_state=123)
```

```
t_clf1.fit(train[shap_cols],y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=5, min_samples_leaf=300, random_state=123)
```


X[0] <= -0.283
gini = 0.551
samples = 10831
value = [3463, 1077, 6291]
True
False
X[1] <= 0.046
gini = 0.556
samples = 1895
value = [801, 127, 967]
X[2] <= -0.056
gini = 0.545
samples = 8936
value = [2662, 950, 5324]
X[5] <= 0.503
gini = 0.55
samples = 1494
value = [689, 81, 724]
gini = 0.542
samples = 401
value = [112, 46, 243]
X[3] <= -0.213
gini = 0.531
samples = 6819
value = [1944, 684, 4191]
X[6] <= 0.154
gini = 0.583
samples = 2117
value = [718, 266, 1133]
X[9] <= -0.495
gini = 0.545
samples = 1173
value = [581, 58, 534]
gini = 0.531
samples = 321
value = [108, 23, 190]
X[10] <= 0.066
gini = 0.56
samples = 1484
value = [506, 146, 832]
X[7] <= 2.17
gini = 0.521
samples = 5335
value = [1438, 538, 3359]
X[8] <= 0.821
gini = 0.58
samples = 1792
value = [568, 240, 984]
gini = 0.57
samples = 325
value = [150, 26, 149]
gini = 0.52
samples = 416
value = [238, 16, 162]
X[13] <= -0.368
gini = 0.55
samples = 757
value = [343, 42, 372]
X[11] <= -0.159
gini = 0.543
samples = 1098
value = [350, 101, 647]
gini = 0.593
samples = 386
value = [156, 45, 185]
X[4] <= 0.032
gini = 0.524
samples = 4926
value = [1371, 485, 3070]
gini = 0.457
samples = 409
value = [67, 53, 289]
X[12] <= 0.628
gini = 0.595
samples = 1317
value = [447, 186, 684]
gini = 0.523
samples = 475
value = [121, 54, 300]
gini = 0.554
samples = 424
value = [211, 26, 187]
gini = 0.532
samples = 333
value = [132, 16, 185]
gini = 0.52
samples = 788
value = [231, 67, 490]
gini = 0.584
samples = 310
value = [119, 34, 157]
gini = 0.502
samples = 3102
value = [791, 291, 2020]
gini = 0.556
samples = 1824
value = [580, 194, 1050]
gini = 0.587
samples = 1011
value = [318, 146, 547]
gini = 0.605
samples = 306
value = [129, 40, 137]


X[0] <= -0.283
gini = 0.551
samples = 10831
value = [3463, 1077, 6291]
True
False
X[2] <= 0.046
gini = 0.556
samples = 1895
value = [801, 127, 967]
X[3] <= -0.056
gini = 0.545
samples = 8936
value = [2662, 950, 5324]
X[5] <= 0.503
gini = 0.55
samples = 1494
value = [689, 81, 724]
gini = 0.542
samples = 401
value = [112, 46, 243]
X[2] <= 0.24
gini = 0.531
samples = 6819
value = [1944, 684, 4191]
X[4] <= 0.154
gini = 0.583
samples = 2117
value = [718, 266, 1133]
X[6] <= -0.495
gini = 0.545
samples = 1173
value = [581, 58, 534]
gini = 0.531
samples = 321
value = [108, 23, 190]
X[0] <= -0.25
gini = 0.543
samples = 4537
value = [1372, 460, 2705]
X[1] <= -0.358
gini = 0.503
samples = 2282
value = [572, 224, 1486]
X[6] <= 0.293
gini = 0.58
samples = 1792
value = [568, 240, 984]
gini = 0.57
samples = 325
value = [150, 26, 149]
gini = 0.52
samples = 416
value = [238, 16, 162]
X[1] <= -0.368
gini = 0.55
samples = 757
value = [343, 42, 372]
X[2] <= 0.079
gini = 0.557
samples = 1613
value = [537, 161, 915]
X[3] <= -0.302
gini = 0.533
samples = 2924
value = [835, 299, 1790]
gini = 0.481
samples = 380
value = [67, 53, 260]
X[0] <= -0.213
gini = 0.506
samples = 1902
value = [505, 171, 1226]
X[6] <= -0.057
gini = 0.584
samples = 1278
value = [433, 163, 682]
gini = 0.563
samples = 514
value = [135, 77, 302]
gini = 0.554
samples = 424
value = [211, 26, 187]
gini = 0.532
samples = 333
value = [132, 16, 185]
gini = 0.565
samples = 1235
value = [436, 122, 677]
gini = 0.522
samples = 378
value = [101, 39, 238]
gini = 0.49
samples = 750
value = [182, 69, 499]
gini = 0.546
samples = 2174
value = [653, 230, 1291]
gini = 0.533
samples = 823
value = [244, 78, 501]
gini = 0.483
samples = 1079
value = [261, 93, 725]
gini = 0.549
samples = 479
value = [144, 52, 283]
gini = 0.6
samples = 799
value = [289, 111, 399]

How well does it do on the test set?

```
test_set = test_red[shap_cols]
pred_test = t_clf1.predict(test_set)
# pred_test_avr = pred_test_avr/np.abs(pred_test_avr).sum()
(pred_test * df_test['next_period_return']).sum()
```

```
440]: 84.19119100000003
```

If we were 100% correct on the test set we would get this profit

```
(y_test * df_test['pred_rel_return']).sum()
```

```
51]: 113.34048000000001
```

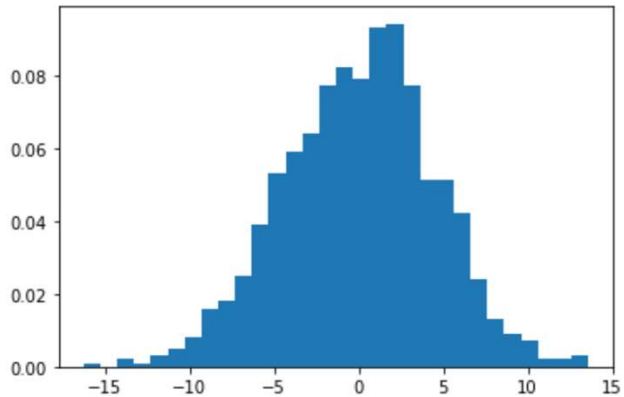
A totally random strategy where we randomly decide whether to buy or sell

```
m = len(df_test['next_period_return'])
```

```
random_predictions = []
for _ in range(1000):
    pred_random = np.random.choice([-1,0,1],m)
    random_predictions.append((pred_random * df_test['next_period_return']).sum())

```

```
plt.hist(random_predictions,bins=30,density=True);
```




```
np.mean(random_predictions)
```

5]: 0.3030609289999995

We will back test our strategy over the range 2001-01-01 to 2018-10-01

First we need the starting dates and the end dates for the training periods, each training period is 36 months and each training period is shifted one quarter from the previous

```
start_dates = [pd.to_datetime('2001-01-01') + pd.DateOffset(months = 3*i) for i in range(58)]
end_dates = [d + pd.DateOffset(months = 36) for d in start_dates]
```

```
training_frames = [data.loc[d:d+pd.DateOffset(months = 36)] for d in start_dates]
test_frames = [data.loc[d + pd.DateOffset(months=3):d+pd.DateOffset(months = 6)] for d in end_dates]
```

```
training_data = [d.reset_index() for d in training_frames]
```

```
test_data = [d.reset_index() for d in test_frames]
```

Then we can select the data frames for each training and validation period

```
training_frames = [data.loc[d:d+pd.DateOffset(months = 36)] for d in start_dates]
validation_frames = [data.loc[d + pd.DateOffset(months=3):d+pd.DateOffset(months = 6)] for d in end_dates]
```

The validation period corresponding to a training period starts 3 months after the end date of the training date and runs for 3 months

We get the training labels for each training set

```
training_labels = [d['rel_performance'].values for d in training_frames]
```

```
scalers = [StandardScaler() for _ in range(len(training_data))]

opt_training_data = [pd.DataFrame(scalers[i].fit_transform(training_frames[i][shap_cols].values),columns=shap_cols) for i
opt_test_data = [pd.DataFrame(scalers[i].transform(test_frames[i][shap_cols].values),columns=shap_cols) for i in range(le
```

Remark that we have to normalize the dataset for each training period

Now we can run through all the dates, train on each training_data and use the trained tree to predict the profit on each validation_data.

We also see how starting with $1 will perform over the 16 year period.

If we have $x we invest (long or short) we invest x/#securities in each stock. The profits from the strategy for each stock in period i is

```
preds = t_clf.predict(opt_test_data[i])
profit_i = (preds*test_frames[i]['next_period_return']).sum()
```

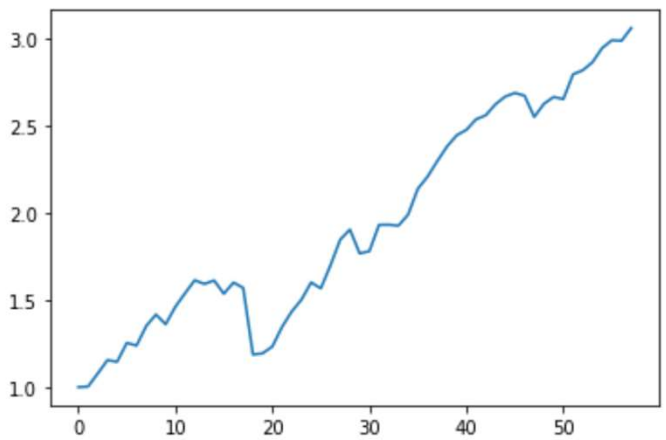
and so the total profit is (x/#securities) * profit_i.

```python
x = [1]
ret = []

for i in range(len(start_dates)-1):
        t_clf.fit(opt_training_data[i],training_labels[i])

        preds = t_clf.predict(opt_test_data[i])
        profit_i = (preds*test_frames[i]['next_period_return']).sum()
        ret.append(profit_i)
        num_names = len(opt_test_data[i])
        x.append(x[i] + (x[i]/num_names)*profit_i)
```

```python
plt.plot(x);
```



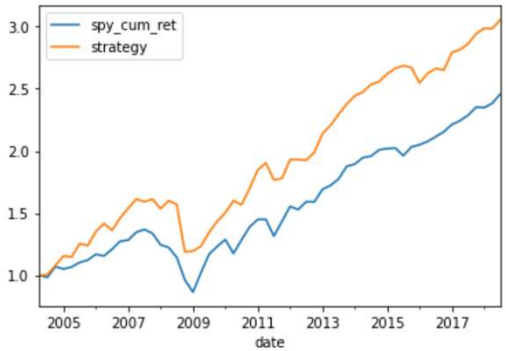
We will compare this to a $1 investment in SPY

We have the cumulative returns on SPY in the raw_data data frame and we only want one return for each date (for each date there will be as many copies of the cumulative SPY returns as there are companies reporting on that date)

Compared to a buy-and-hold of SPY

```
SPY = pd.read_pickle(r'C:\Users\niels\OneDrive\Machine Learning 2022\Lecture 2\SPY_cum_ret.pkl')
SPY = SPY.loc['2004-04-01':'2018-09-30']
SPY = SPY.resample('Q').ffill()
SPY['spy_cum_ret'] = (SPY['spy_cum_ret'] - SPY['spy_cum_ret'][0]+1)
SPY['strategy'] = x
```

```
SPY.plot();
```



The strategy handily beats the SPY but remark that we have not included transaction costs which might change the result significantly.

The Sharpe Ratios:

```
strategy_mean_ret = (SPY['strategy'] - 1).diff().mean()
strategy_std = (SPY['strategy'] - 1).diff().std()
print('Strategy Sharpe Ratio: ',strategy_mean_ret/strategy_std)
```

```
Strategy Sharpe Ratio:  0.43225473187964636
```

```
strategy_std
```

```
0.08352985589809588
```

```
spy_mean_ret = (SPY['spy_cum_ret'] - 1).diff().mean()
spy_std = (SPY['spy_cum_ret'] - 1).diff().std()
print('SPY Sharpe Ratio: ',spy_mean_ret/spy_std)
```

```
SPY Sharpe Ratio:  0.3825904447137517
```

```
print(spy_std)
```

```
0.06690829328830444
```

Total Returns

```
x[-1]
```

50]: 3.058052001515085

```
SPY['spy_cum_ret'][-1]
```

51]: 2.459113

Computing the (quarterly) α of the strategy

```
strategy_ret = (SPY['strategy'] - 1).diff().values[1:]
spy_ret = (SPY['spy_cum_ret'] - 1).diff().values[1:]
```

```
beta = (np.cov(spy_ret,strategy_ret)/np.var(spy_ret))[1,0]
beta
```

54]: 0.7217252621919491

```
residual_ret = strategy_ret - beta * spy_ret
IR = np.mean(residual_ret)/np.std(residual_ret)
IR
```

55]: 0.25871496302388025

```
alpha = np.mean(residual_ret)
alpha
```

56]: 0.017631110333726375