DAVIDOV_Exam_Three_Report

May 19, 2020

1 CQF Exam Three - Boyan Davidov

1.1 1. Introduction

In this report, we will implement several machine learning techniques and study their performance based on different parameters and features. For this purpose, I have chosen two stocks from the banking sector in Austria - Erste Group AG and Raiffeisen Bank AG. Both of them are included in Austrian Traded Index (ATX), with relatively similar size (market cap of around 5 and 7 bln. euros) and very liquid shares in Austria. We will start first by loading the modules needed.

[2]:
```
import pandas as pd
import numpy as np
from pylab import plt
plt.style.use('seaborn')
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

from scipy import stats
import statsmodels.api as sm

import statistics as st
from sklearn.model_selection import cross_val_score

import yfinance as yf
```

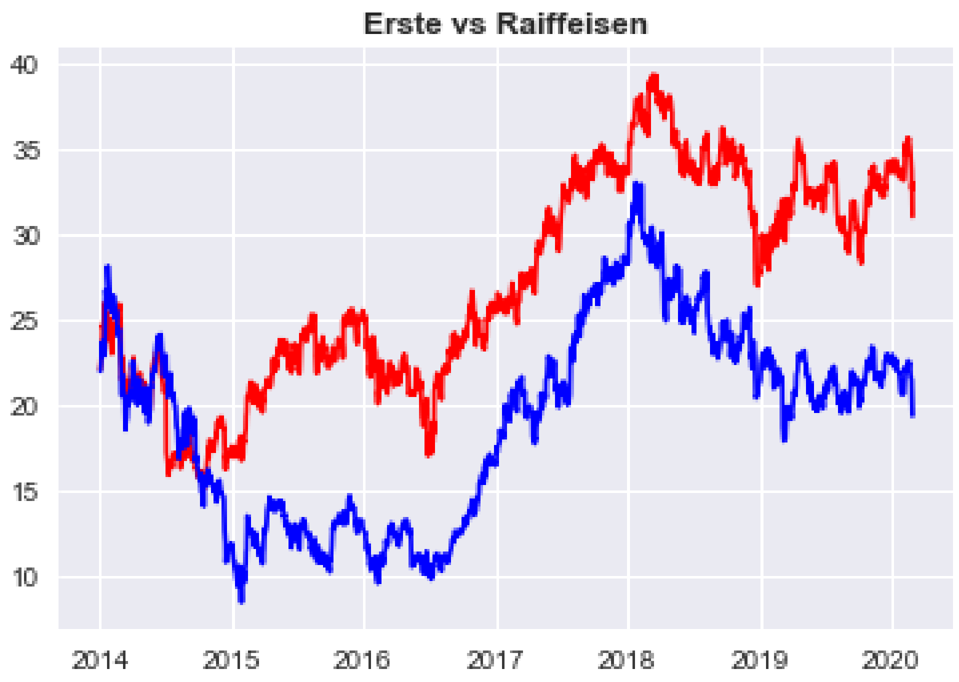
Next, we download timeseries. It will cover six years of daily prices data, 1531 observations. As the stocks are from the same index, we have equal lengths and same trading days in both timeseries.

[3]:
```
EBS_data = yf.download("EBS.VI", start="2014-01-01", end="2020-02-28") # Erste␣
 ↪Group AG
RBI_data = yf.download("RBI.VI", start="2014-01-01", end="2020-02-28") #␣
 ↪Raiffeisen Bank AG
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
```

Further on we will consider only the ‘Adj Close’. Let’s plot the data.

[4]:
```
EBS = EBS_data['Adj Close']
RBI = RBI_data['Adj Close']

plt.plot(EBS, 'r');
plt.plot(RBI, 'b');
plt.title("Erste vs Raiffeisen",fontweight="bold")
plt.show()
```



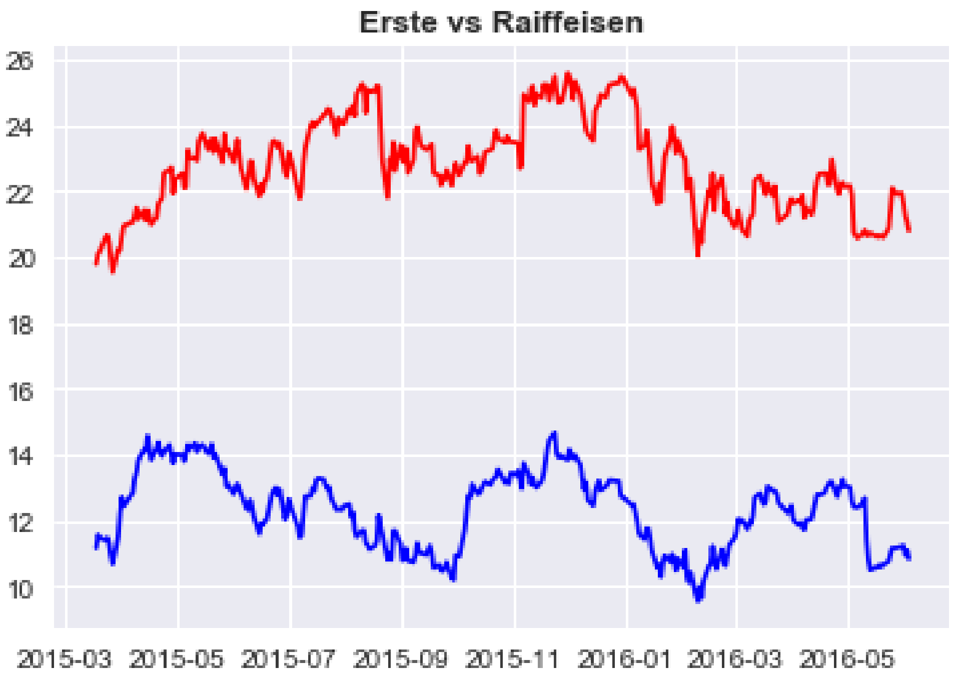
From the plot we can see that the stocks are well correlated. We can check that:

[6]:
```
np.corrcoef(EBS,RBI)
```

[6]:
```
array([[1.        , 0.77880169],
       [0.77880169, 1.        ]])
```

Albeit their similar size and the fact that they operate in the same sector, the correlation is not always so strong. One example is the years 2015-2016:

[5]:
```
plt.plot(EBS[300:600], 'r');
plt.plot(RBI[300:600], 'b');
plt.title("Erste vs Raiffeisen",fontweight="bold")
plt.show()

np.corrcoef(EBS[300:600],RBI[300:600])
```



```
[5]: array([[1.        , 0.3709669],
            [0.3709669, 1.        ]])
```

Since Raiffeisein Bank has a strong presence in Russia, stock has been positively correlated with the Russian ruble in turbulent times (strong influence from risk of sanctions, volatilve oil prices etc.). In such times, the stock of Erste Group historically seemed to be more stable, which explains some discrepancies in the positive correlation. Counting on mean reversion in long-term, the shares look interesting for analysis and pairs trading, which is the reason why we have chosen them.

1.1.1 1.1 Creating Features and Auxiliary Functions

Our first function computes lagged log returns. Here we create 5 lags of length 1D. We refer to the most recent daily return as "ret_0". In this report we have studied daily price predictions, however, with the function below one can easily adjust the window length (for instance to predict 1 week returns one can set window_length = 5).

```
[5]: ###########################################
     # Compute lagged log returnsa

     def getLaggedReturns(prices,lags = 5, window_length = 1):
         #df: pandas Dataframe type variable, and column 'Close' should be included
         #lags: how many lagged returns do you want
         #window_length: the window length we use to compute each return
         laggedReturns =pd.DataFrame(index = prices.index)
         laggedReturns['Adj Close'] = prices
```

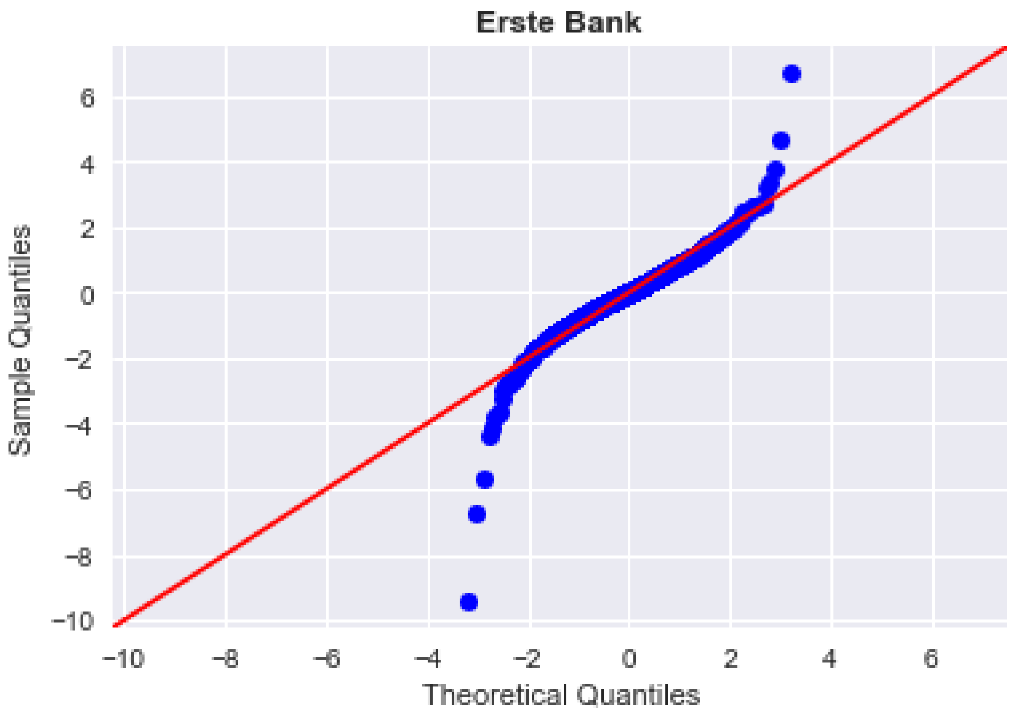
```
    laggedReturns['ret_0'] = np.log(laggedReturns['Adj Close']/
↪laggedReturns['Adj Close'].shift(window_length))


    for lag in range( 1,lags + 1 ):
        col = 'ret_%d'%lag
        laggedReturns[col] = laggedReturns['ret_0'].shift(lag)
    laggedReturns.dropna(inplace = True)

    return laggedReturns

EBS_laggedRet=getLaggedReturns(EBS)
RBI_laggedRet=getLaggedReturns(RBI)

EBS_laggedRet.tail(3)
```

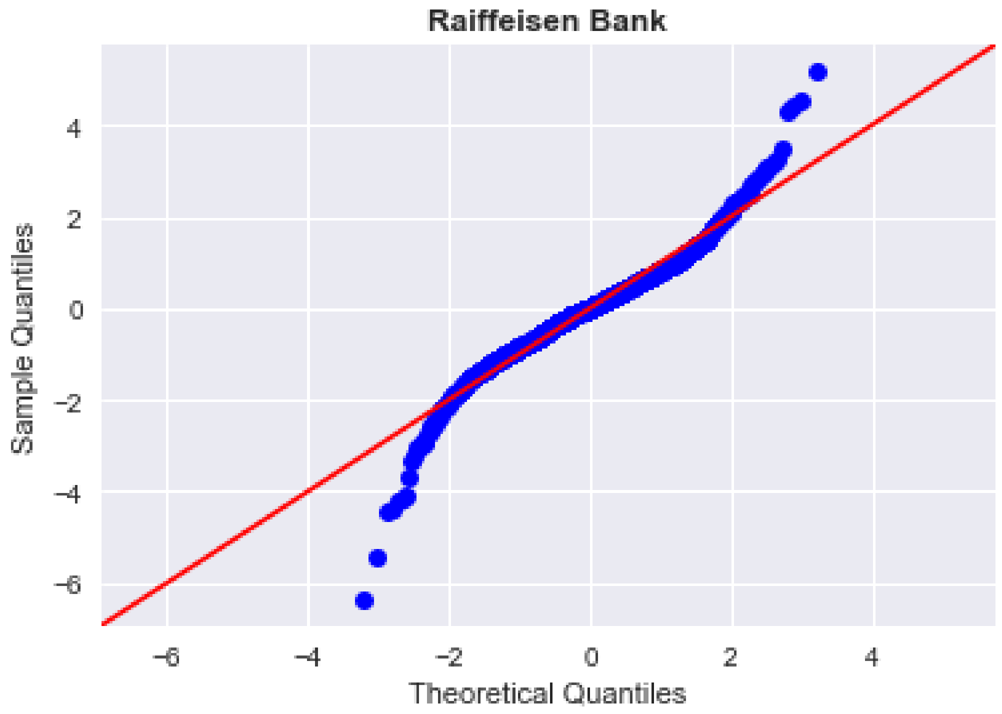
[5]:
```
            Adj Close     ret_0     ret_1     ret_2     ret_3     ret_4  \
Date
2020-02-25  32.599998 -0.031106 -0.027278 -0.006345 -0.010582 -0.001421
2020-02-26  32.959999  0.010982 -0.031106 -0.027278 -0.006345 -0.010582
2020-02-27  31.000000 -0.061307  0.010982 -0.031106 -0.027278 -0.006345

               ret_5
Date
2020-02-25 -0.010456
2020-02-26 -0.001421
2020-02-27 -0.010582
```

The qqplots below, show that the stock returns can exhibit fat tails.

[6]:
```
sm.qqplot(EBS_laggedRet['ret_0'],fit= True, line='45')
plt.title("Erste Bank",fontweight="bold")

sm.qqplot(RBI_laggedRet['ret_0'],fit= True, line='45')
plt.title("Raiffeisen Bank",fontweight="bold")
```

[6]:
```
Text(0.5, 1.0, 'Raiffeisen Bank')
```





Next, we create columns showing the sign of the respective lagged returns

```
[6]: ##############################################
     #Set of Features to predict the prices

     #starting point always lagged returns
     EBS_features=getLaggedReturns(EBS)
     RBI_features=getLaggedReturns(RBI)

     #shows whether return is negative or positive
     cols =['ret_0','ret_1','ret_2','ret_3','ret_4','ret_5']
     for idx,col in enumerate(cols):
             EBS_features['Sign'+str(idx)] = np.sign(EBS_features[col])

     EBS_features.head(3)
```

```
[6]:             Adj Close     ret_0     ret_1     ret_2     ret_3     ret_4  \
     Date
     2014-01-13  25.334984  0.031526  0.001959 -0.005157  0.043130  0.058728
     2014-01-14  24.972431 -0.014414  0.031526  0.001959 -0.005157  0.043130
     2014-01-15  25.955252  0.038602 -0.014414  0.031526  0.001959 -0.005157

                    ret_5  Sign0  Sign1  Sign2  Sign3  Sign4  Sign5
     Date
     2014-01-13  0.008480    1.0    1.0   -1.0    1.0    1.0    1.0
     2014-01-14  0.058728   -1.0    1.0    1.0   -1.0    1.0    1.0
     2014-01-15  0.043130    1.0   -1.0    1.0    1.0   -1.0    1.0
```

We compute and load momentum indicator $(P_t - P_{t-k})$, for 5D,7D,13D and 21D intervals.

```
[7]: #computes momentum for different time intervals
     def GetMomentum(targetDF,sourceDF,time_intervals=[5,7,13,21]):
         for time_interval in time_intervals:
             momentum = (sourceDF.shift(time_interval)['Adj Close']-sourceDF['Adj␣
      ↪Close'])/sourceDF['Adj Close']
             targetDF = pd.DataFrame(targetDF.iloc[time_interval:,:])
             targetDF['MOM'+str(time_interval)]=momentum

         targetDF = targetDF.dropna()
         return targetDF


     EBS_features=GetMomentum(EBS_features,EBS_features)
     RBI_features=GetMomentum(RBI_features,RBI_features)
```

Finally, we compute different intervals of Simple and Exponential Moving Averages, as well as setup for Moving Average Crossover strategies (i.e. shorter-term MA vs longer-term MA).

[8]:
```
#Compute EWMA for 5,7,13,21 days and make crossover strategies
EBS_features['EWMA5'] = pd.Series(EBS_features['Adj Close']).ewm(span = 5).
 ↪mean()
EBS_features['EWMA7'] = pd.Series(EBS_features['Adj Close']).ewm(span = 7).
 ↪mean()
EBS_features['EWMA13'] = pd.Series(EBS_features['Adj Close']).ewm(span = 13).
 ↪mean()
EBS_features['EWMA21'] = pd.Series(EBS_features['Adj Close']).ewm(span = 21).
 ↪mean()
EBS_features['EWMA5-7'] = EBS_features['EWMA5'] - EBS_features['EWMA7']
EBS_features['EWMA7-13'] = EBS_features['EWMA7'] - EBS_features['EWMA13']
EBS_features['EWMA13-21'] = EBS_features['EWMA13'] - EBS_features['EWMA21']

RBI_features['EWMA5'] = pd.Series(RBI_features['Adj Close']).ewm(span = 5).
 ↪mean()
RBI_features['EWMA7'] = pd.Series(RBI_features['Adj Close']).ewm(span = 7).
 ↪mean()
RBI_features['EWMA13'] = pd.Series(RBI_features['Adj Close']).ewm(span = 13).
 ↪mean()
RBI_features['EWMA21'] = pd.Series(RBI_features['Adj Close']).ewm(span = 21).
 ↪mean()
RBI_features['EWMA5-7'] = RBI_features['EWMA5'] - RBI_features['EWMA7']
RBI_features['EWMA7-13'] = RBI_features['EWMA7'] - RBI_features['EWMA13']
RBI_features['EWMA13-21'] = RBI_features['EWMA13'] - RBI_features['EWMA21']

#Compute Simple MA for 5,7,13,21 days and make crossover strategies
EBS_features['MA5'] = pd.Series(EBS_features['Adj Close']).rolling(5).mean()
EBS_features['MA7'] = pd.Series(EBS_features['Adj Close']).rolling(7).mean()
EBS_features['MA13'] = pd.Series(EBS_features['Adj Close']).rolling(13).mean()
EBS_features['MA21'] = pd.Series(EBS_features['Adj Close']).rolling(21).mean()
EBS_features['MA5-7'] = EBS_features['MA5'] - EBS_features['MA7']
EBS_features['MA7-13'] = EBS_features['MA7'] - EBS_features['MA13']
EBS_features['MA13-21'] = EBS_features['MA13'] - EBS_features['MA21']

RBI_features['MA5'] = pd.Series(RBI_features['Adj Close']).rolling(5).mean()
RBI_features['MA7'] = pd.Series(RBI_features['Adj Close']).rolling(7).mean()
RBI_features['MA13'] = pd.Series(RBI_features['Adj Close']).rolling(13).mean()
RBI_features['MA21'] = pd.Series(RBI_features['Adj Close']).rolling(21).mean()
RBI_features['MA5-7'] = RBI_features['MA5'] - RBI_features['MA7']
RBI_features['MA7-13'] = RBI_features['MA7'] - RBI_features['MA13']
RBI_features['MA13-21'] = RBI_features['MA13'] - RBI_features['MA21']
```

Our longest term feature will be 21D rolling standard deviation of returns.

[109]:
```
EBS_features['stdev21'] = pd.Series(EBS_features['Adj Close']).rolling(21).std()
RBI_features['stdev21'] = pd.Series(RBI_features['Adj Close']).rolling(21).std()
```

We will set a column to indicate up or down move from "ret_0" where returns >= 0 will be assigned 1, else 0.

```
[9]: EBS_features['I'] = EBS_features['ret_0']
     EBS_features['I'][EBS_features['I']>0] = 1
     EBS_features['I'][EBS_features['I']<=0] = -1

     RBI_features['I'] = RBI_features['ret_0']
     RBI_features['I'][RBI_features['I']>0] = 1
     RBI_features['I'][RBI_features['I']<=0] = -1

     EBS_features['Sign']= np.sign(EBS_features['ret_0'])
     EBS_features['Sign'][EBS_features['Sign']<0] =0

     RBI_features['Sign']= np.sign(RBI_features['ret_0'])
     RBI_features['Sign'][RBI_features['Sign']<0] =0
```

Let's see all features created

```
[25]: EBS_features.columns
```

```
[25]: Index(['Adj Close', 'ret_0', 'ret_1', 'ret_2', 'ret_3', 'ret_4', 'ret_5',
             'Sign0', 'Sign1', 'Sign2', 'Sign3', 'Sign4', 'Sign5', 'MOM5', 'MOM7',
             'MOM13', 'MOM21', 'stdev21', 'I', 'Sign', 'EWMA5', 'EWMA7', 'EWMA13',
             'EWMA21', 'EWMA5-7', 'EWMA7-13', 'EWMA13-21', 'MA5', 'MA7', 'MA13',
             'MA21', 'MA5-7', 'MA7-13', 'MA13-21'],
            dtype='object')
```

As a last step we clean the data by deleting the empty cells.

```
[10]: EBS_features=EBS_features.dropna()
      RBI_features=RBI_features.dropna()
```

1.1.2 1.2 Choosing between Features.

This is only a short section to describe our approach on choosing indicators. Clearly, some of the features we created so far supply the same information and our model will exhibit multicollinearity. This can be easily seen by running regressions and VIF (Variance Infaltion Factors). We start firstly with all features.

```
[18]: cols=['ret_1', 'ret_2', 'ret_3', 'ret_4', 'ret_5',
            'Sign1', 'Sign2', 'Sign3', 'Sign4', 'Sign5', 'MOM5', 'MOM7',
             'MOM13', 'MOM21', 'EWMA5', 'EWMA7', 'EWMA13', 'EWMA21', 'EWMA5-7',
             'EWMA7-13', 'EWMA13-21', 'MA5', 'MA7', 'MA13', 'MA21', 'MA5-7',
             'MA7-13', 'MA13-21', 'stdev21']
```

```
[19]: import statsmodels.api as sm
      from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
EBS_features_reg = EBS_features
sm_ols = sm.OLS(EBS_features_reg['ret_0'], EBS_features[cols]).fit()
print(sm_ols.rsquared)
```

```
0.9902456456929744
```

This seems like almost perfect match, however even Python warns of multicollinearity if run the whole model.summary(). There are variables which are highly correlated. Here we print only the first 5 coefficients.

```
[38]: variables = sm_ols.model.exog
      vif = [variance_inflation_factor(variables, i) for i in range(variables.
       ↪shape[1])]
      vif[1:5]
```

```
[38]: [5.429288417033856, 10.091441372879263, 20.888654844599134, 6.485081687605195]
```

It is clear that the signs and the lagged returns provide the same information. The features of same type are also highly correlated (eg. MA5 and MA7). We exclude one by one features and test for multicollinearity until we have VIF coefficients smaller than 5. As a benchamrk we will use the accuracy of a logit model with selected features on a in-sample and out-of-sample split, and we will disregard the r-squared from a simple regression. By excluding features, one can see that a smaller set could actually achieve higher accuracy than an extended set of features. The different sets tested can be seen in the code file provided with this report. We found that an optimal set is using only three indicators.

```
[20]: cols=['MOM5','EWMA5-7','ret_1']
      #selected features

      #let's check the linear regression summary
      lm = sm.OLS(EBS_features_reg['ret_0'], EBS_features[cols]).fit()
      print("R-squared :",lm.rsquared)
```

```
R-squared : 0.2596785973907262
```

Not a strong result but we will run more sophisticated models to improve that. We will implement learning algorithms on training set (in-sample) and validation set (out-of-sample). This is why for the moment we can disregard linear regression performance metrics. Important is that there is no multicollinearity.

```
[45]: variables = lm.model.exog
      vif = [variance_inflation_factor(variables, i) for i in range(variables.
       ↪shape[1])]
      vif
```

```
[45]: [3.152703731650922, 2.9819457865451553, 1.24695100045014]
```

Let’s check the accuracy of a fit/predict logit regression.

```
[21]: from sklearn import model_selection
      from sklearn import metrics

      EBS_features=EBS_features.dropna()
      X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
       ↪train_test_split(EBS_features[cols],
                                                                                 ␣
       ↪    EBS_features['I'] ,
                                                                                 ␣
       ↪    test_size=0.25, shuffle=True)
      from sklearn import linear_model
      logit_EBS = linear_model.LogisticRegression(C = 1e15,solver␣
       ↪='liblinear',penalty = 'l2')
      logit_EBS.fit(X_Train_EBS,Y_Train_EBS)
      Y_EBS_pred = logit_EBS.predict(X_Test_EBS)


      print("Accuracy with logit regression:",metrics.accuracy_score( Y_Test_EBS,␣
       ↪Y_EBS_pred))

      EBS_coef = pd.Series(logit_EBS.coef_.ravel(),index = cols)
      print("EBS Regression Model Coefficient Analysis")
      EBS_coef.sort_values()
```

```
Accuracy with logit regression: 0.684931506849315
EBS Regression Model Coefficient Analysis
```

```
[21]: MOM5       -35.138607
      ret_1      -25.924948
      EWMA5-7     -3.665535
      dtype: float64
```

Alternatively, one can use Principal Component Analysis narrowing down the features by for instance selecting eigenvalues larger than 1 (or the first components that explain 80% variance). This is however not in the scope of our report, we will not expand more in that and we will just refer to our optimal model that is created with the list of features: cols=['MOM5','EWMA5-7','ret_1']

1.2 Classifier A.1 Logistic Classiffier and Bayesian Classiffier

1.2.1 a) Penalised versions of logistic regression and impact on coefficients. Difference between L1 and L2 cost functions.

Our first classifier is a logistic regression - supervised learning algorithm for binary classification where to y can be described only with two values 0 and 1. It can be regarded as an OLS Linear regression transformed into [0,1]. The following formula defines Logistic Regression Model:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + exp(-\theta^T x)} \tag{1}$$

$\theta^T x$) is in fact a linear regression prediction. We note that as $-\theta^{Tx} \to \infty$, then $h_\theta(x) \to 0$, i.e. our prediction will be further located in class labeled with 0, vice-versa:

as $-\theta^{Tx} \to -\infty$, then $h_\theta(x) \to 1$, i.e. our prediction will be further located in class labeled with 1. We can also summarize that if $h_\theta(x) > 0.5$,the probability that the sample is located in class 1 is larges, we simply assume y=1, whereas if $h_\theta(x) < 0.5$, we assume y=0:

$$y = \begin{cases} 1, & h_\theta(x) > 0.5 \\ 0, & h_\theta(x) < 0.5 \end{cases} \tag{2}$$

Writing the probability as:

$$p(y|x;\theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y} \tag{3}$$

.

In order to fit the parameters of the regression our objective is to maximise the log-likelihood function (assuming n training sets):

$$l(\theta) = \sum_{i=1}^{n} y^{(i)} logh(x^{(i)}) + (1 - y^{(i)}) log(1 - h(x^{(i)})) \tag{4}$$

In our first implementations we will always fit logistic regression by using only the first 5 lagged returns. This is indeed the simplest indicator and even when the results from the model are not good this can give us the opportunity to test whether different techniques can improve it. In this part will only measure the overall accuracy of the model and later (task B) we look at other indicators for model performance.

[83]:
```python
#import sklearn module for logistic regression
from sklearn import linear_model

#call the set of features to fit the classifier as "cols"
cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']



#LogisticRegression Type Object of Sklearn are created
lm_EBS = linear_model.LogisticRegression(C =␣
↪1e6,solver='liblinear',multi_class='ovr',penalty='l2')
lm_RBI = linear_model.LogisticRegression(C =␣
↪1e6,solver='liblinear',multi_class='ovr',penalty='l2')
```

```
EBS_features['Sign']= np.sign(EBS_features['ret_0'])
EBS_features['Sign'][EBS_features['Sign']<0] =0

RBI_features['Sign']= np.sign(RBI_features['ret_0'])
RBI_features['Sign'][RBI_features['Sign']<0] =0

###Fit our data
lm_EBS.fit(EBS_features[cols],EBS_features['Sign'])
lm_RBI.fit(RBI_features[cols], RBI_features['Sign'])

###Now we using our model to do prediction
EBS_features['Logit_Predict'] = lm_EBS.predict(EBS_features[cols])
RBI_features['Logit_Predict'] = lm_RBI.predict(RBI_features[cols])

EBS_features.head(5)

EBS_features['Logit_Predict'][EBS_features['Logit_Predict']==0] = -1
RBI_features['Logit_Predict'][RBI_features['Logit_Predict']==0] = -1

EBS_features['Logit_Returns'] = EBS_features['Logit_Predict'] *␣
 ↪EBS_features['ret_0']
RBI_features['Logit_Returns'] = RBI_features['Logit_Predict'] *␣
 ↪RBI_features['ret_0']

EBS_features[['Adj Close','ret_1','ret_2', 'ret_3',␣
 ↪'ret_4','ret_5','Sign','Logit_Predict','Logit_Returns']].head(3)
```

[83]:

```
            Adj Close     ret_1     ret_2     ret_3     ret_4     ret_5  Sign  \
Date
2014-04-15  24.115000 -0.016138 -0.003439  0.001212  0.003646 -0.025642   0.0
2014-04-16  24.514999 -0.006819 -0.016138 -0.003439  0.001212  0.003646   1.0
2014-04-17  24.600000  0.016451 -0.006819 -0.016138 -0.003439  0.001212   1.0

            Logit_Predict  Logit_Returns
Date
2014-04-15            1.0      -0.006819
2014-04-16            1.0       0.016451
2014-04-17           -1.0      -0.003461
```

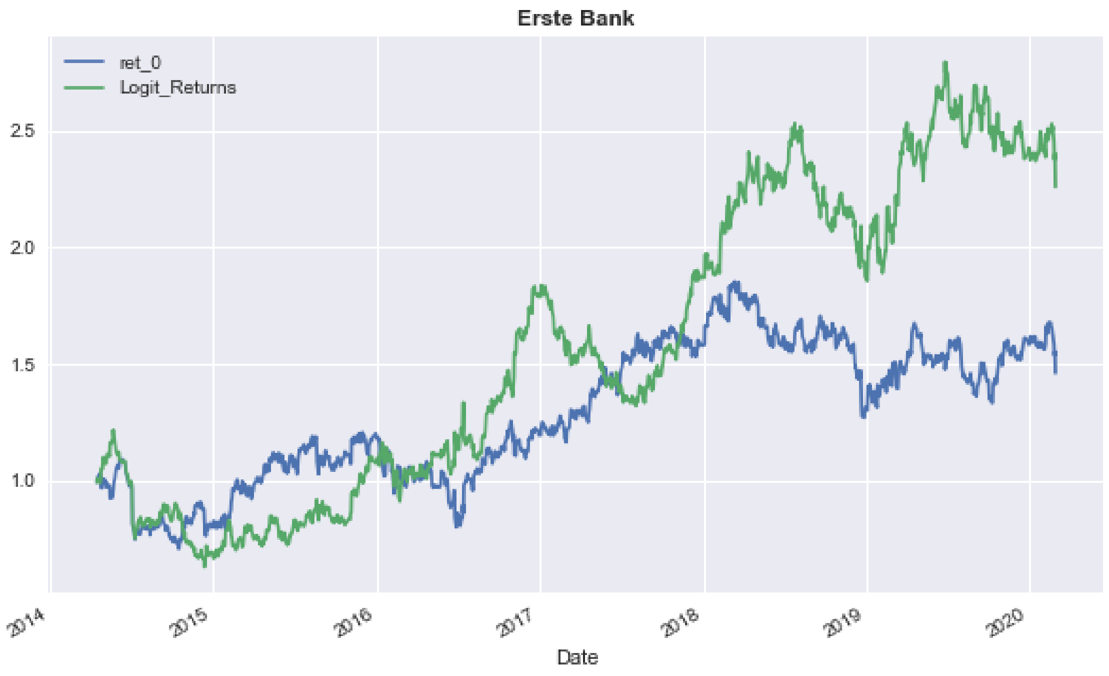
Now let us plot our Net Asset Value evolution that could have been obtained with the predicted values.

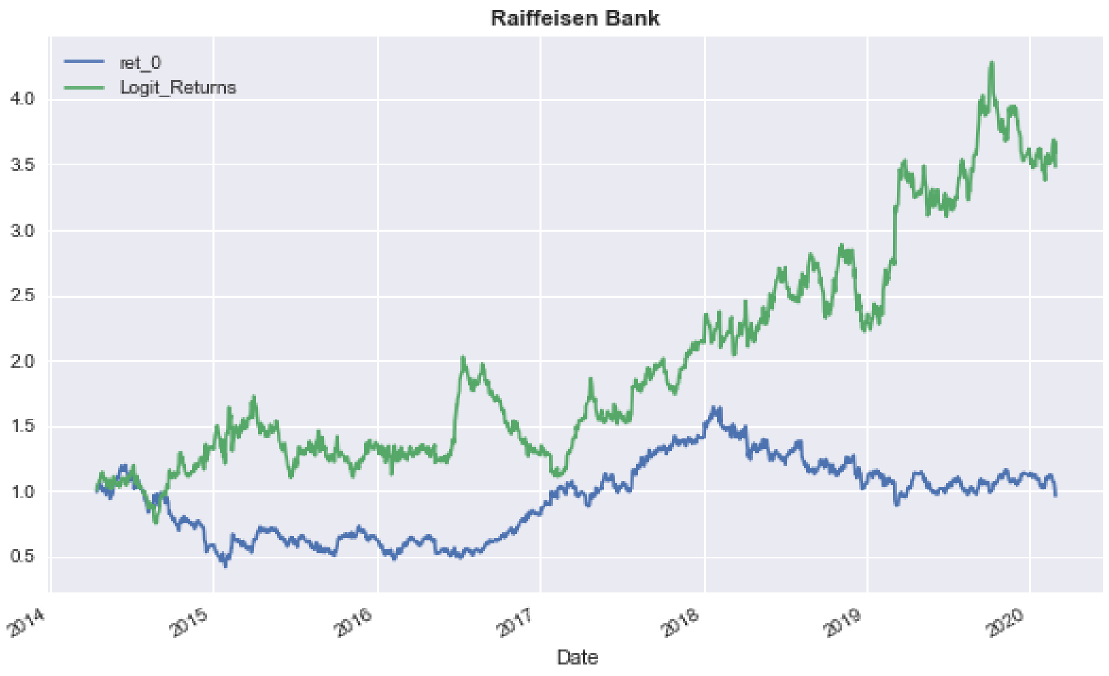
[12]:

```
#Compute the performance of the model
EBS_features[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
plt.title("Erste Bank",fontweight="bold")
```

```
RBI_features[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
plt.title("Raiffeisen Bank",fontweight="bold")
#Interesting observation - the model seems to match better the RBI stock
```

```
[12]: Text(0.5, 1.0, 'Raiffeisen Bank')
```





Visually, the plot leaves the impression that the direction has been most of the time correctly predicted. However, let's check the accuracy:

[43]:
```
from sklearn.model_selection import cross_val_score
crossval_EBS =␣
 ↪cross_val_score(lm_EBS,EBS_features[cols],EBS_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For EBS Logit Model : %.4f"%crossval_EBS.mean())
crossval_RBI =␣
 ↪cross_val_score(lm_RBI,RBI_features[cols],RBI_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For RBI Logit Model : %.4f"%crossval_RBI.mean())
```

```
Accuracy For EBS Logit Model : 0.4784
Accuracy For RBI Logit Model : 0.4949
```

This is certainly not sufficient. Admittedly, we can achieve better results with another set of feeatures. Let's check the coefficients.

[44]:
```
print('Erste Bank - Logit Regression Coefficients: ')
print(lm_EBS.coef_.ravel())
print('Raiffeisen Bank - Logit Regression Coefficients : ')
print(lm_RBI.coef_.ravel())
```

```
Erste Bank - Logit Regression Coefficients:
[-0.59770039 -1.63397823  0.09005414  0.07097229 -2.84028291]
```

```
Raiffeisen Bank - Logit Regression Coefficients :
[ 1.66466956 -1.97628537  0.44139752 -2.41130401 -1.27473537]
```

From the coefficients we can see that the stocks have different sensitivities to the lagged returns. For instance Raiffeisen is relatively more exposed to previous day return, Erste Group in turn on 5D return.

In this first attempt we implemented penalized L2 regression. With L2 we refer to the cost function or regularization technique (which allows to control the trade-off between variance and bias). There are namely two types:

L2: Ridge Regression: $l(\hat{\theta}) = \sum_{i=1}^{n} y^{(i)} logh(x^{(i)}) + (1 - y^{(i)}) log(1 - h(x^{(i)})) + \frac{\lambda}{2}\theta^T\theta$

L1: Lasso Regression: $l(\hat{\theta}) = \sum_{i=1}^{n} y^{(i)} logh(x^{(i)}) + (1 - y^{(i)}) log(1 - h(x^{(i)})) + \frac{\lambda}{2}\mathbf{1}^T|\theta|$

These two techniques have both a goal to maximize $l(\hat{\theta})$, while regularazing the coefficients and preventing from becoming too large.

Now let's see the results of the previous regression when applying L1 penalty.

[85]:
```
########################################
#Same regression but using L1 penalty
cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']
lm_EBS = linear_model.LogisticRegression(C =␣
 ↪1e6,solver='liblinear',multi_class='ovr',penalty='l1')
lm_RBI = linear_model.LogisticRegression(C =␣
 ↪1e6,solver='liblinear',multi_class='ovr',penalty='l1')

#create column "Sign" from "ret_0" where
#returns >= 0 will be assigned 1, else 0.

EBS_features['Sign']= np.sign(EBS_features['ret_0'])
EBS_features['Sign'][EBS_features['Sign']<0] =0

RBI_features['Sign']= np.sign(RBI_features['ret_0'])
RBI_features['Sign'][RBI_features['Sign']<0] =0

###Fit our data using Lagged ret_1, ret_2, ret_3, ret_4, ret_5 with Output Sign␣
 ↪+1 or -1
lm_EBS.fit(EBS_features[cols],EBS_features['Sign'])
lm_RBI.fit(RBI_features[cols], RBI_features['Sign'])

###Now we using our model to do prediction
EBS_features['Logit_Predict'] = lm_EBS.predict(EBS_features[cols])
RBI_features['Logit_Predict'] = lm_RBI.predict(RBI_features[cols])

EBS_features.head(5)

EBS_features['Logit_Predict'][EBS_features['Logit_Predict']==0] = -1
RBI_features['Logit_Predict'][RBI_features['Logit_Predict']==0] = -1
```

```
EBS_features['Logit_Returns'] = EBS_features['Logit_Predict'] *␣
 ↪EBS_features['ret_0']
RBI_features['Logit_Returns'] = RBI_features['Logit_Predict'] *␣
 ↪RBI_features['ret_0']

#visually not much of difference, lets make more sophisticated comparison
#let's check the coefficients
print('Erste Bank - Logit Regression Coefficients: ')
print(lm_EBS.coef_.ravel())
print('Raiffeisen Bank - Logit Regression Coefficients : ')
print(lm_RBI.coef_.ravel())

#In the same regression if we put C=0.1 the L1 will give only 0 coeff.
```

```
Erste Bank - Logit Regression Coefficients:
[-0.48856139 -1.69670951  0.23944774 -0.18401683 -2.37961238]
Raiffeisen Bank - Logit Regression Coefficients :
[ 1.66490046 -1.97705531  0.44201105 -2.41164084 -1.27349213]
```

[20]:
```
from sklearn.model_selection import cross_val_score
crossval_EBS =␣
 ↪cross_val_score(lm_EBS,EBS_features[cols],EBS_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For EBS Logit Model : %.4f"%crossval_EBS.mean())
crossval_RBI =␣
 ↪cross_val_score(lm_RBI,RBI_features[cols],RBI_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For RBI Logit Model : %.4f"%crossval_RBI.mean())
```

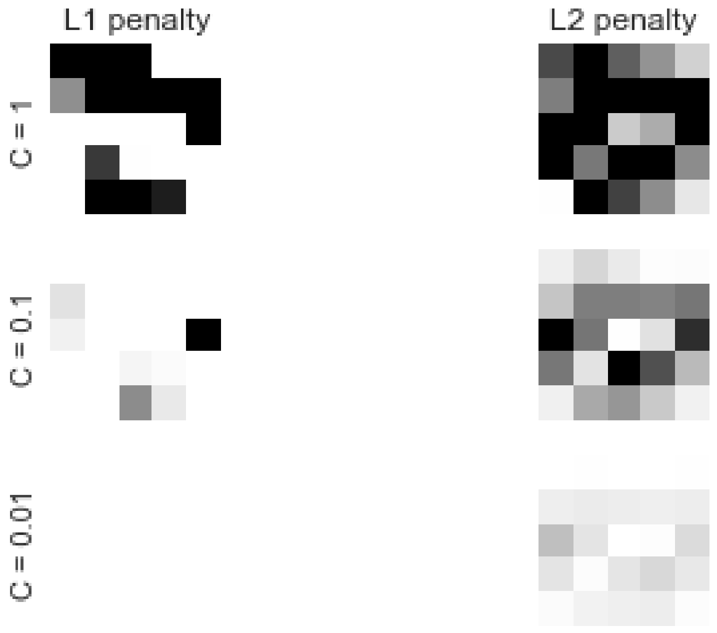
```
Accuracy For EBS Logit Model : 0.4764
Accuracy For RBI Logit Model : 0.4949
```

There is only slight difference in the coeffiecients. We have omitted the plots here but there was hardly any improvement as the it can be seen from the accuracy of the model. Important for the model is clearly also the freedom of the model set by the parameter C. Let us examine the influence of C but this time including more features to be able to compare differences.

[16]:
```
###########
#Comparison
#Different parameters C, L1,L2 penalties
cols=['ret_1', 'ret_2', 'ret_3', 'ret_4', 'ret_5',
      'Sign1', 'MOM5', 'MOM7',
       'MOM13', 'MOM21', 'EWMA5', 'EWMA7', 'EWMA13', 'EWMA21', 'EWMA5-7',
       'EWMA7-13', 'EWMA13-21', 'MA5', 'MA7', 'MA13', 'MA21', 'MA5-7',
       'MA7-13', 'MA13-21', 'stdev21']
```

```
fig, axes = plt.subplots(3, 2)

# Set regularization parameter
# C - smaller values specify stronger regularization.
for i, (C, axes_row) in enumerate(zip((1, 0.1, 0.01), axes)):
    # turn down tolerance for short training time
    clf_l1_LR = linear_model.LogisticRegression(C=C, penalty='l1', tol=0.01,␣
 ↪solver='liblinear')
    clf_l2_LR = linear_model.LogisticRegression(C=C, penalty='l2', tol=0.01,␣
 ↪solver='liblinear')

    # EBS_features=getLaggedReturns(EBS,lags=24)

    EBS_features['Sign']= np.sign(EBS_features['ret_0'])
    EBS_features['Sign'][EBS_features['Sign']<0] =0

    clf_l1_LR.fit(EBS_features[cols],EBS_features['Sign'])
    clf_l2_LR.fit(EBS_features[cols],EBS_features['Sign'])

    coef_l1_LR = clf_l1_LR.coef_.ravel()
    coef_l2_LR = clf_l2_LR.coef_.ravel()

    # coef_l1_LR contains zeros due to the
    # L1 sparsity inducing norm

    sparsity_l1_LR = np.mean(coef_l1_LR == 0) * 100
    sparsity_l2_LR = np.mean(coef_l2_LR == 0) * 100

    print("C=%.2f" % C)
    print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_l1_LR))
    print("{:<40} {:.2f}%".format("Sparsity with L2 penalty:", sparsity_l2_LR))
    print("{:<40} {:.2f}".format("Score with L1 penalty:",
                                 clf_l1_LR.
 ↪score(EBS_features[cols],EBS_features['Sign'])))
    print("{:<40} {:.2f}".format("Score with L2 penalty:",
                                 clf_l2_LR.
 ↪score(EBS_features[cols],EBS_features['Sign'])))

    if i == 0:
        axes_row[0].set_title("L1 penalty")
        axes_row[1].set_title("L2 penalty")

    for ax, coefs in zip(axes_row, [coef_l1_LR, coef_l2_LR]):
        ax.imshow(np.abs(coefs.reshape(5, 5)), interpolation='nearest',
                  cmap='binary', vmax=1, vmin=0)
        ax.set_xticks(())
        ax.set_yticks(())
    axes_row[0].set_ylabel('C = %s' % C)

plt.show()
```

```
C=1.00
Sparsity with L1 penalty:               12.00%
Sparsity with L2 penalty:               0.00%
Score with L1 penalty:                  0.85
Score with L2 penalty:                  0.72
C=0.10
Sparsity with L1 penalty:               72.00%
Sparsity with L2 penalty:               0.00%
Score with L1 penalty:                  0.65
Score with L2 penalty:                  0.68
C=0.01
Sparsity with L1 penalty:               92.00%
Sparsity with L2 penalty:               0.00%
Score with L1 penalty:                  0.50
Score with L2 penalty:                  0.62
```



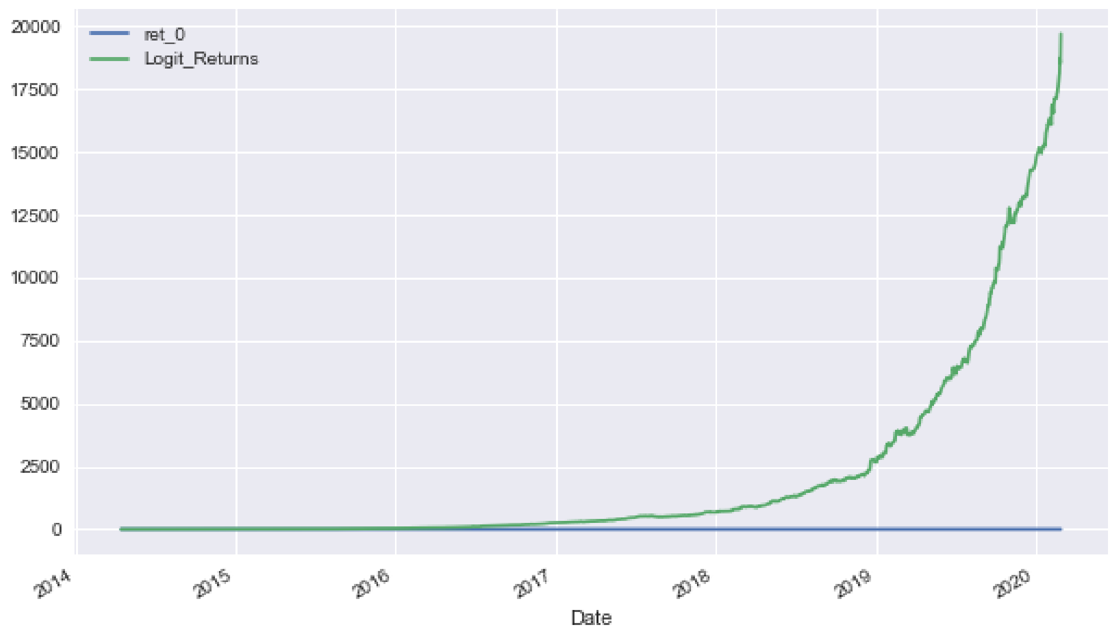
As parameter C sets the freedom of the model we observe, that higher values of C give more leeway to the model with L1 penalty,whereas, smaller values constrain it In the L1 case, decreaing C leads to sparser solutions (percentage of zero coeff.). We can also see that by less freedom of the model the Lasso (L1) regression performs better than the Ridge (L2). For smaller values of C, the latter performs better.

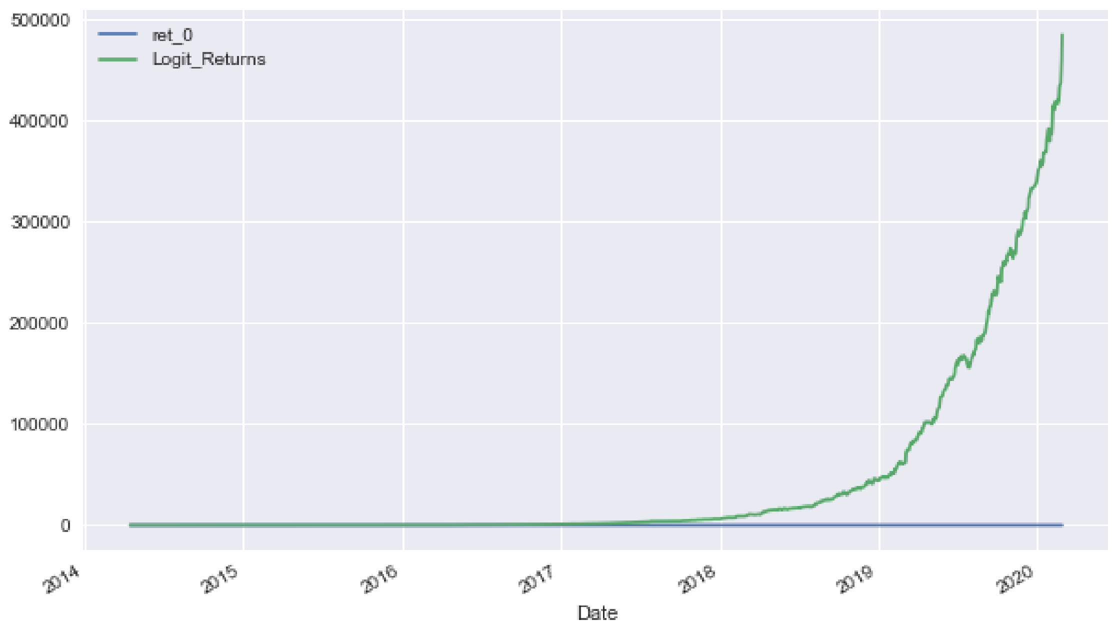
Next, we will examine a better set of features based on our discussion in the introductory section.

[26]:
```
##############################################
#Let's use more sophisticated model
cols=['MOM5', 'EWMA5-7','ret_1']

lm_EBS = linear_model.LogisticRegression(C =␣
 ↪1e6,solver='liblinear',multi_class='ovr',penalty='l2')
lm_RBI = linear_model.LogisticRegression(C =␣
 ↪1e6,solver='liblinear',multi_class='ovr',penalty='l2')

EBS_features['Sign']= np.sign(EBS_features['ret_0'])
EBS_features['Sign'][EBS_features['Sign']<0] =0

RBI_features['Sign']= np.sign(RBI_features['ret_0'])
RBI_features['Sign'][RBI_features['Sign']<0] =0

###Fit our data
lm_EBS.fit(EBS_features[cols],EBS_features['Sign'])
lm_RBI.fit(RBI_features[cols], RBI_features['Sign'])

###Now we using our model to do prediction
EBS_features['Logit_Predict'] = lm_EBS.predict(EBS_features[cols])
RBI_features['Logit_Predict'] = lm_RBI.predict(RBI_features[cols])

EBS_features.head(5)

EBS_features['Logit_Predict'][EBS_features['Logit_Predict']==0] = -1
RBI_features['Logit_Predict'][RBI_features['Logit_Predict']==0] = -1

EBS_features['Logit_Returns'] = EBS_features['Logit_Predict'] *␣
 ↪EBS_features['ret_0']
RBI_features['Logit_Returns'] = RBI_features['Logit_Predict'] *␣
 ↪RBI_features['ret_0']

EBS_features.head(5)

#Compute the performance of the model
EBS_features[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
RBI_features[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));

#Amazing. This seems like perfect match



#let's check the coefficients
```

```python
print('Erste Bank - Logit Regression Coefficients: ')
print(lm_EBS.coef_.ravel())
print('Raiffeisen Bank - Logit Regression Coefficients : ')
print(lm_RBI.coef_.ravel())
```

```
Erste Bank - Logit Regression Coefficients:
[-35.65048551  -3.64696661 -25.65891935]
Raiffeisen Bank - Logit Regression Coefficients :
[-22.81230814  -2.93588206 -16.17437838]
```



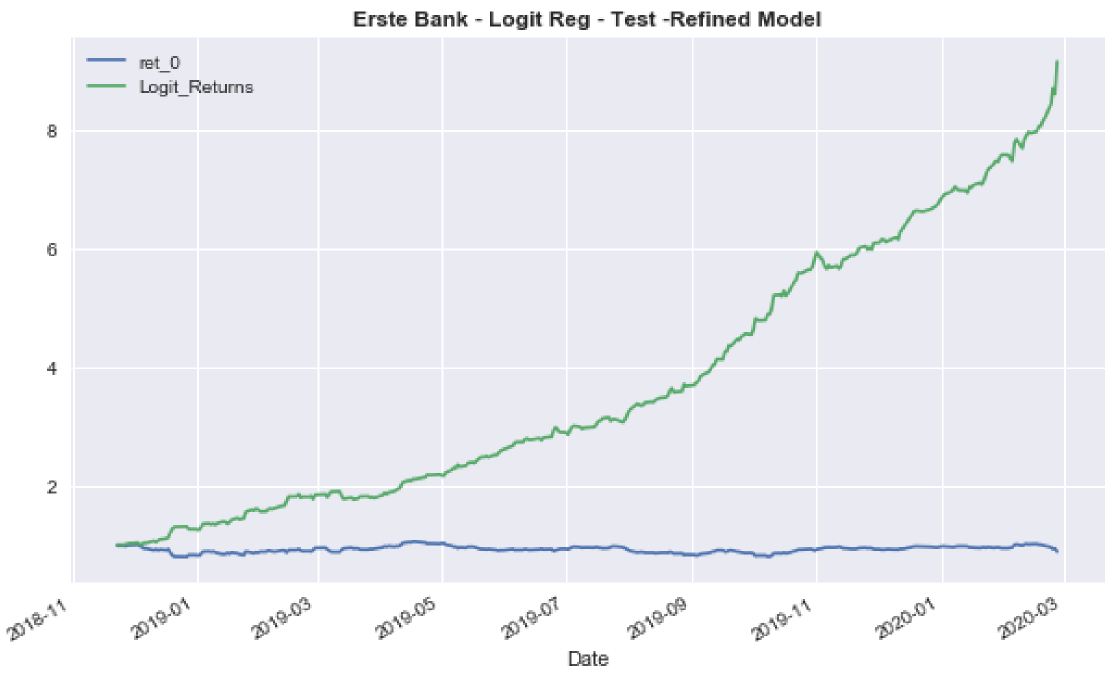


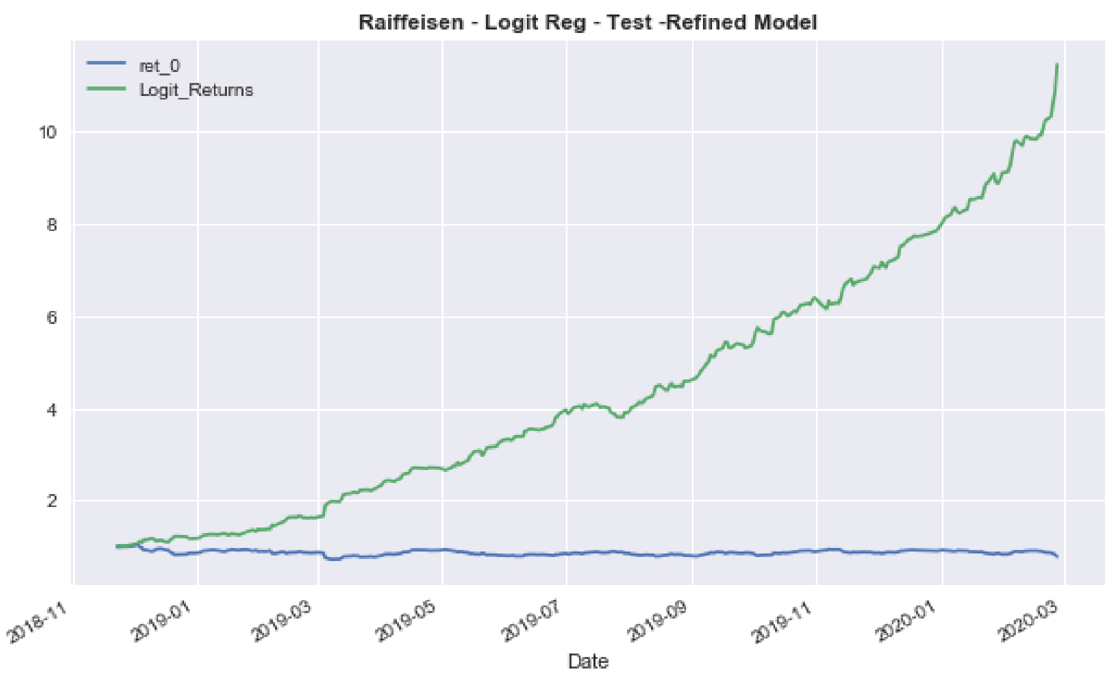
Based on the performance plot, one might think this is the perfect model. However this rather makes an impression of overfitting. What we can easily notice is also the high coefficients. To reduce the weight of the features one can simply increase C. The overfitting comes with the fact that we practically train and test the model on the same data. When backtesting a model, what counts is the predictions on a validation set or the out-of-sample data, rather than results on a training set of data (on which the model is fitted). Thus, in order tonfirm the robustness of the model we will split our data on train and test sets and check the results again.

[28]:
```
training_split = int(0.75*EBS.shape[0])

EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

#fit again
lm_EBS.fit(EBS_training[cols],EBS_training['Sign'])
lm_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['Logit_Predict'] = lm_EBS.predict(EBS_validate[cols])
RBI_validate['Logit_Predict'] = lm_RBI.predict(RBI_validate[cols])


EBS_validate['Logit_Predict'][EBS_validate['Logit_Predict']==0] = -1
RBI_validate['Logit_Predict'][RBI_validate['Logit_Predict']==0] = -1
```

```
EBS_validate['Logit_Returns']␣
↪=EBS_validate['ret_0']*EBS_validate['Logit_Predict']
RBI_validate['Logit_Returns']␣
↪=RBI_validate['ret_0']*RBI_validate['Logit_Predict']

EBS_validate.tail(3)

EBS_validate[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
↪plot(figsize=(10, 6));
plt.title("Erste Bank - Logit Reg - Test -Refined Model",fontweight="bold")
RBI_validate[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
↪plot(figsize=(10, 6));
plt.title("Raiffeisen - Logit Reg - Test -Refined Model",fontweight="bold")
```

[28]:
```
Text(0.5, 1.0, 'Raiffeisen - Logit Reg - Test -Refined Model')
```



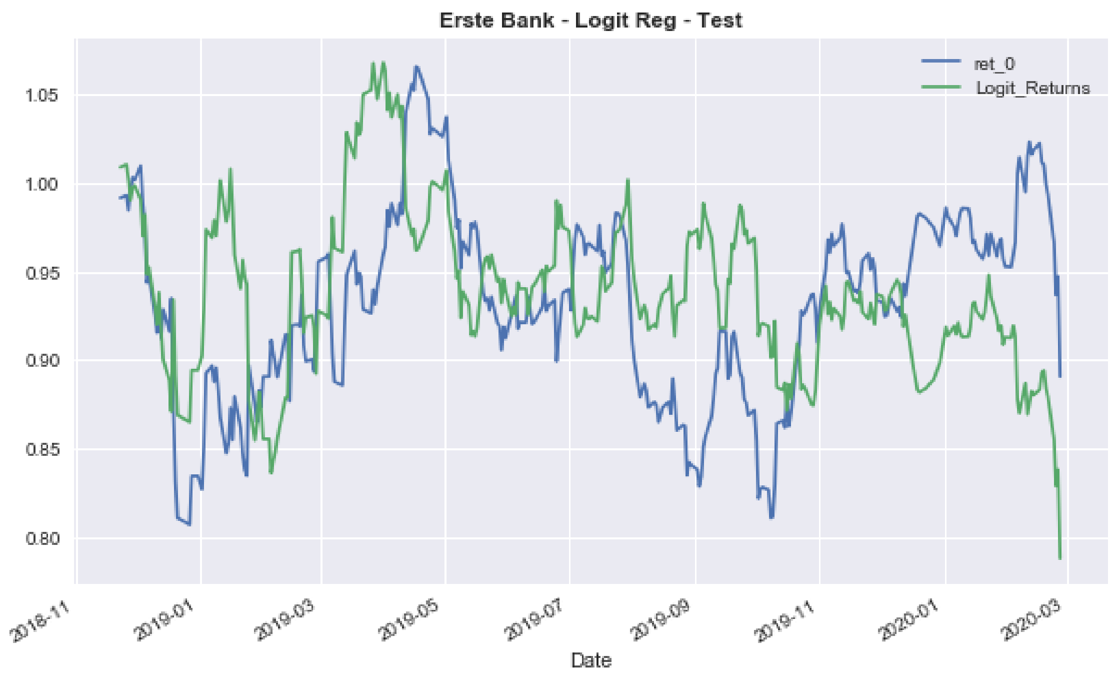


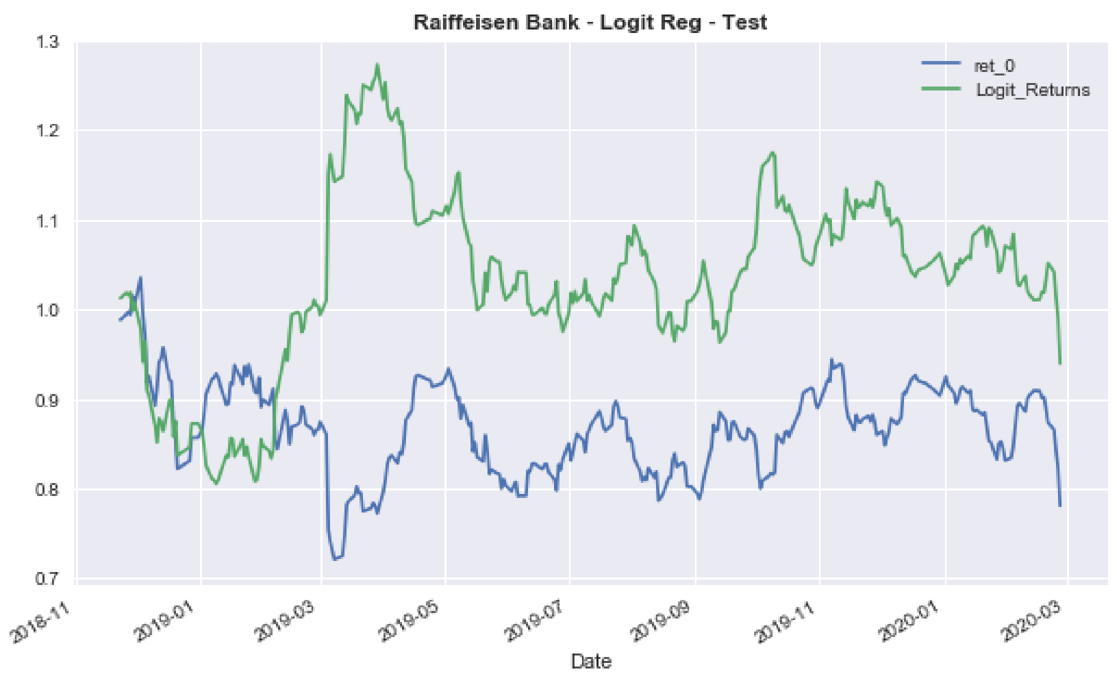
Now we can confirm indeed that the models would have performed well also in real-trading. For comparison, let's see the result of the old set of features on out-of-sample data.

[29]:
```
cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']
training_split = int(0.75*EBS.shape[0])

EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

#fit again
lm_EBS.fit(EBS_training[cols],EBS_training['Sign'])
lm_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['Logit_Predict'] = lm_EBS.predict(EBS_validate[cols])
RBI_validate['Logit_Predict'] = lm_RBI.predict(RBI_validate[cols])


EBS_validate['Logit_Predict'][EBS_validate['Logit_Predict']==0] = -1
RBI_validate['Logit_Predict'][RBI_validate['Logit_Predict']==0] = -1
```

```
EBS_validate['Logit_Returns']␣
↪=EBS_validate['ret_0']*EBS_validate['Logit_Predict']
RBI_validate['Logit_Returns']␣
↪=RBI_validate['ret_0']*RBI_validate['Logit_Predict']

EBS_validate.tail(3)

EBS_validate[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
↪plot(figsize=(10, 6));
plt.title("Erste Bank - Logit Reg - Test",fontweight="bold")
RBI_validate[['ret_0', 'Logit_Returns']].cumsum().apply(np.exp).
↪plot(figsize=(10, 6));
plt.title("Raiffeisen Bank - Logit Reg - Test",fontweight="bold")
```

[29]:
```
Text(0.5, 1.0, 'Raiffeisen Bank - Logit Reg - Test')
```





We can conclude that a model can give very good results on in-sample data or in “training mode”, however, the performance can fall drastically when tested on out-of-sample data or “real-trading” mode. Despite the initially good results, the overall impression is still that in out-of-sample the model behaves like random guessing. It could be noted that the performance of the model fell with the final part of the data set, where the stocks were in down-trend. Thus, it would be interesting to examine the power of predicting positive and negative returns. This will be made with confusion matrices in task B.1.

1.2.2 A.1 b) Demonstrate the use of sklearn.model selection for reshuffled samples and k-fold crossvalidation.

After having seen that the simple model based only on lagged returns fails in out-of-sample data, we will try to see in this section if we can improve it. For this purpose we will randomly sort our data (shuffle it) and split it in sections (creating KFold object). Thus, we will have K number of consecutive sections, and (k-1) subsets are used to train to the model. The remaining one (chosen with the random_state) is used to test our data. We will have k validation rounds and each time, one subset plays the role of validation set and the rest of data will be used to train the model.

[39]:
```
cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
#We first create a KFold object, and we define the number of sections to be 7
#Here shuffle= True means we will shuffle our data, or in other words, randomly␣
↪sort it, before our split
kfold = KFold(n_splits = 7,random_state=3,shuffle = True)
```

```
print(kfold)

###Now we get the number of splitting iterations in the cross-validator
kfold.get_n_splits(EBS_features[cols])
kfold.get_n_splits(RBI_features[cols])

#to cross-validate our EBS logit model
logit_EBS = linear_model.LogisticRegression(C=1e5,solver␣
 ↪='liblinear',multi_class='auto',penalty='l2')
logit_EBS.fit(EBS_features[cols],EBS_features['Sign'])
crossval_EBS =␣
 ↪cross_val_score(logit_EBS,EBS_features[cols],EBS_features['Sign'],cv=kfold,scoring␣
 ↪= 'accuracy')
print("Accuracy For EBS Logit Model with 5-Fold Cross Validation: %.
 ↪4f"%crossval_EBS.mean())

#to cross-validate our RBI logit model
logit_RBI = linear_model.LogisticRegression(C=1e5,solver␣
 ↪='liblinear',multi_class='auto',penalty='l2')
logit_RBI.fit(RBI_features[cols],RBI_features['Sign'])
crossval_RBI =␣
 ↪cross_val_score(logit_RBI,RBI_features[cols],RBI_features['Sign'],cv=kfold,scoring␣
 ↪= 'accuracy')
print("Accuracy For RBI Logit Model with 5-Fold Cross Validation: %.
 ↪4f"%crossval_RBI.mean())

#the model seems to gets improved with 5-Fold Cross Validation
```

```
KFold(n_splits=7, random_state=3, shuffle=True)
Accuracy For EBS Logit Model with 5-Fold Cross Validation: 0.4846
Accuracy For RBI Logit Model with 5-Fold Cross Validation: 0.4949
```

We can see that the model accuracy increased. Changing the number of splits and random section can help us further optimize the model. One should be aware that increasing the number of training sets can improve accuracy but increasing the trainings sets will decrease the nunmber of data observations in each set. In fact at some point increasing n_splits, will lead to lower model accuracy.

1.2.3 Bayesian Classiffier

We will implement the two most frequently applied Bayes models: Gaussian and Bernoulli. Noting that

$$y \sim Bernoulli(\phi) \tag{5}$$

we have:

$p(y) = \phi^y(1-\phi)^{1-y}$

$p(x|y=0) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0))$

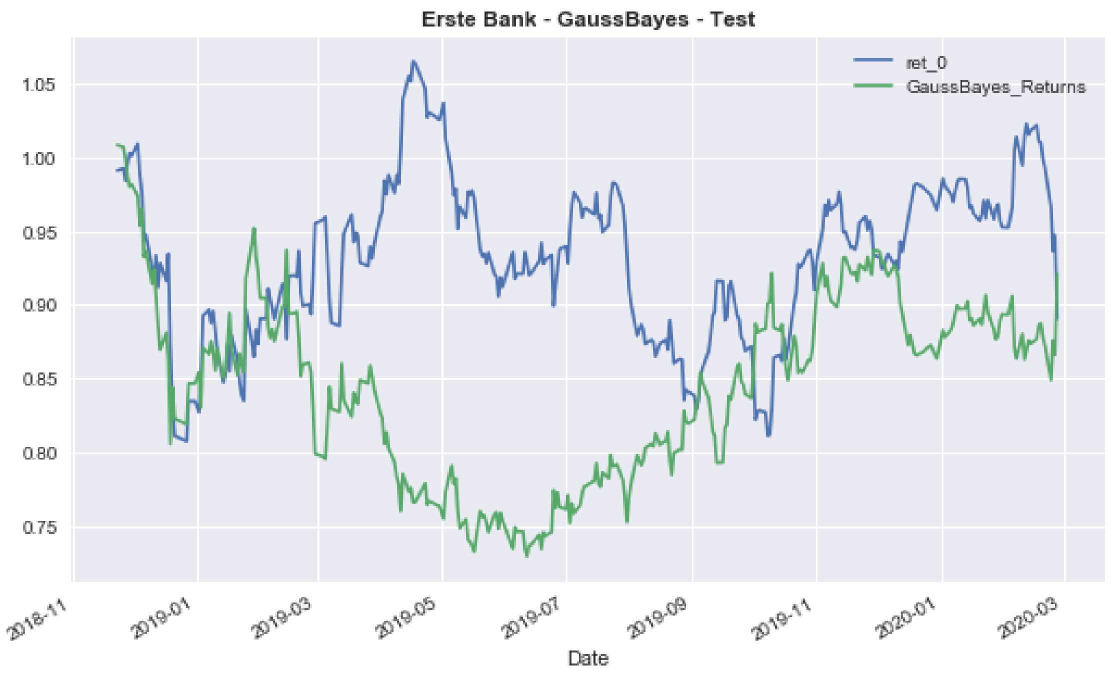
$p(x|y=1) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1))$

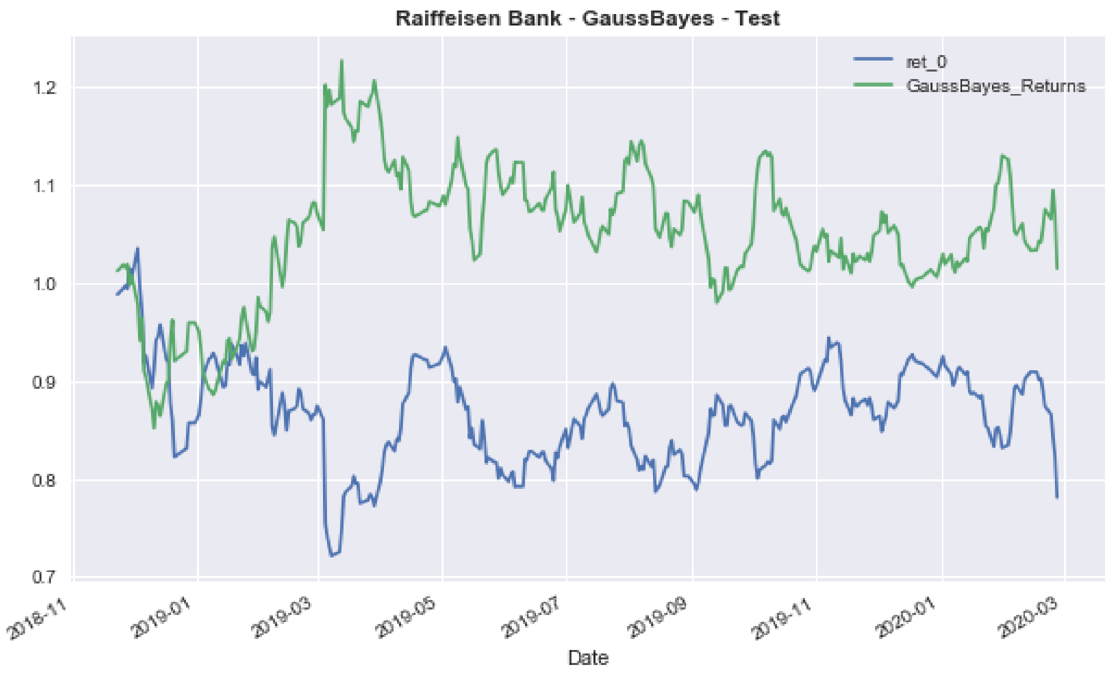
In the Bernoulli case our features' vector will be transformed to contain only two values 1 or 0, whereas in the Gaussian case we can consider only continous signals (the lagged returns comply with this).

To apply Gaussian Naive Bayesian Classifier we need to make sure whether our data follows Gaussian distribution.

[53]:
```
# Naive Bayes Classifier using GaussianNB
from sklearn.naive_bayes import GaussianNB

gaussianBayes_EBS = GaussianNB()
gaussianBayes_RBI = GaussianNB()

gaussianBayes_EBS.fit(EBS_training[cols],EBS_training['Sign'])
gaussianBayes_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['GaussBayes_Predict'] = gaussianBayes_EBS.
 ↪predict(EBS_validate[cols])
RBI_validate['GaussBayes_Predict'] = gaussianBayes_RBI.
 ↪predict(RBI_validate[cols])


EBS_validate['GaussBayes_Predict'][EBS_validate['GaussBayes_Predict']==0] = -1
RBI_validate['GaussBayes_Predict'][RBI_validate['GaussBayes_Predict']==0] = -1

EBS_validate['GaussBayes_Returns']␣
 ↪=EBS_validate['ret_0']*EBS_validate['GaussBayes_Predict']
RBI_validate['GaussBayes_Returns']␣
 ↪=RBI_validate['ret_0']*RBI_validate['GaussBayes_Predict']


EBS_validate[['ret_0', 'GaussBayes_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
plt.title("Erste Bank - GaussBayes - Test",fontweight="bold")
RBI_validate[['ret_0', 'GaussBayes_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
plt.title("Raiffeisen Bank - GaussBayes - Test",fontweight="bold")
# this is not better than the logit regression
```

[53]:
```
Text(0.5, 1.0, 'Raiffeisen Bank - GaussBayes - Test')
```





This is slightly better than the logit regression for the case of Raiffeisen Bank, but overall still not satisfactory performance.

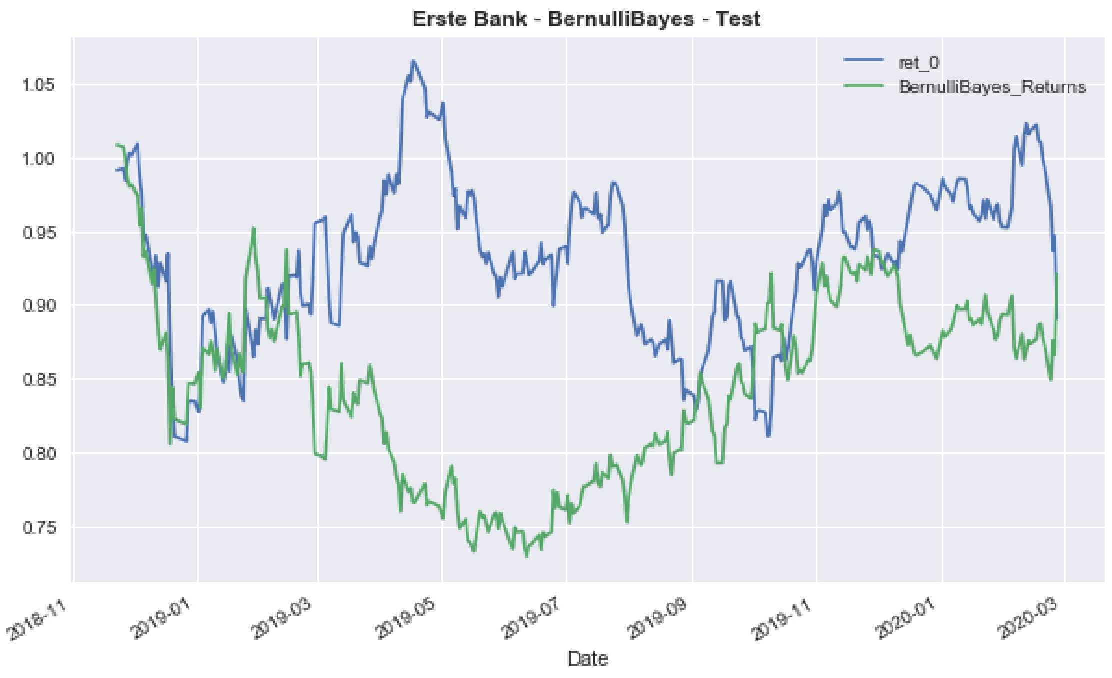
[54]:
```
from sklearn.model_selection import cross_val_score
crossval_EBS =␣
 ↪cross_val_score(gaussianBayes_EBS,EBS_features[cols],EBS_features['Sign'],scoring␣
 ↪= 'accuracy')
print("Accuracy For EBS Logit Model : %.4f"%crossval_EBS.mean())
crossval_RBI =␣
 ↪cross_val_score(gaussianBayes_RBI,RBI_features[cols],RBI_features['Sign'],scoring␣
 ↪= 'accuracy')
print("Accuracy For RBI Logit Model : %.4f"%crossval_RBI.mean())
```

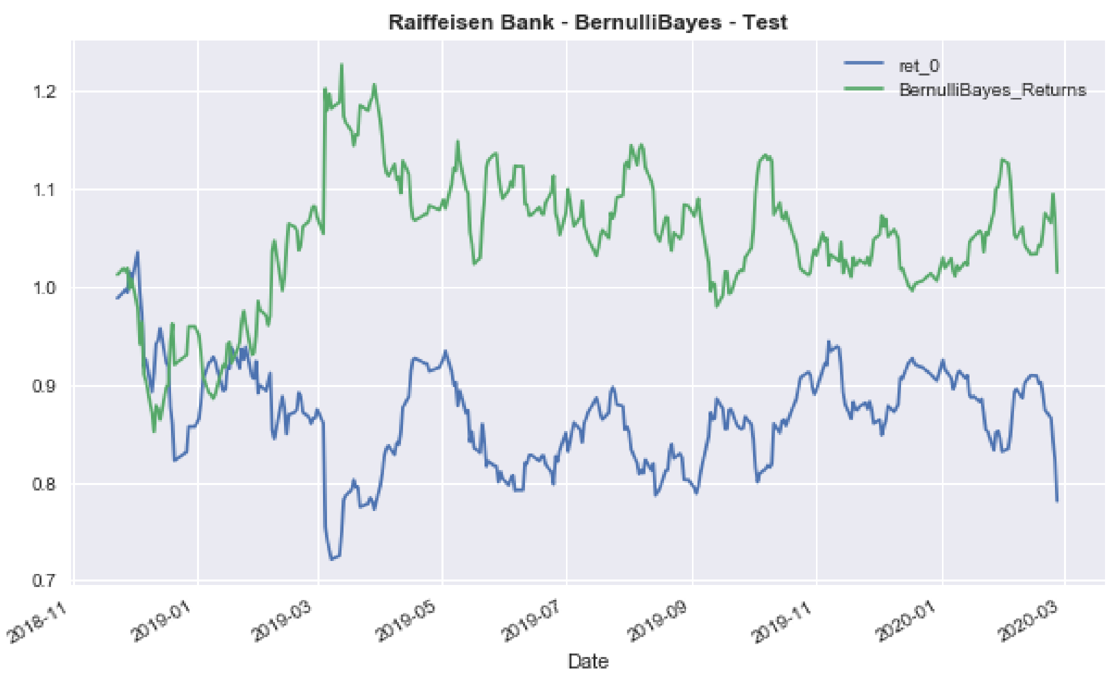
```
Accuracy For EBS Logit Model : 0.4757
Accuracy For RBI Logit Model : 0.5092
```

[55]:
```
# Naive Bayes Classifier using BernoulliNB
from sklearn.naive_bayes import BernoulliNB

bernulliBayes_EBS = BernoulliNB()
bernulliBayes_RBI = BernoulliNB()

#make a backup on features data
EBS_features_Binary=EBS_features
RBI_features_Binary=RBI_features

#make the relevant factors in binary form
for factor in cols:
       EBS_features_Binary[factor][EBS_features_Binary[factor]>0] = 1
       EBS_features_Binary[factor][EBS_features_Binary[factor]<=0] = 0
       RBI_features_Binary[factor][RBI_features_Binary[factor]>0] = 1
       RBI_features_Binary[factor][RBI_features_Binary[factor]<=0] = 0

training_split = int(0.75*EBS.shape[0])

EBS_training = EBS_features_Binary.iloc[0:training_split,:]
RBI_training = RBI_features_Binary.iloc[0:training_split,:]

EBS_validate = EBS_features_Binary.iloc[training_split:,:]
RBI_validate = RBI_features_Binary.iloc[training_split:,:]

bernulliBayes_EBS.fit(EBS_training[cols],EBS_training['Sign'])
bernulliBayes_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['BernulliBayes_Predict'] = bernulliBayes_EBS.
 ↪predict(EBS_validate[cols])
RBI_validate['BernulliBayes_Predict'] = bernulliBayes_RBI.
 ↪predict(RBI_validate[cols])
```

```
EBS_validate['BernulliBayes_Predict'][EBS_validate['BernulliBayes_Predict']==0]␣
 ↪= -1
RBI_validate['BernulliBayes_Predict'][RBI_validate['BernulliBayes_Predict']==0]␣
 ↪= -1

EBS_validate['BernulliBayes_Returns']␣
 ↪=EBS_validate['ret_0']*EBS_validate['BernulliBayes_Predict']
RBI_validate['BernulliBayes_Returns']␣
 ↪=RBI_validate['ret_0']*RBI_validate['BernulliBayes_Predict']


EBS_validate[['ret_0', 'BernulliBayes_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
plt.title("Erste Bank - BernulliBayes - Test",fontweight="bold")
RBI_validate[['ret_0', 'BernulliBayes_Returns']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
plt.title("Raiffeisen Bank - BernulliBayes - Test",fontweight="bold")
# this is better than Gaussian but not the logit regression
```

```
[55]: Text(0.5, 1.0, 'Raiffeisen Bank - BernulliBayes - Test')
```





[56]:
```
crossval_EBS =␣
 ↪cross_val_score(bernulliBayes_EBS,EBS_features[cols],EBS_features['Sign'],scoring␣
 ↪= 'accuracy')
print("Accuracy For EBS Logit Model : %.4f"%crossval_EBS.mean())
crossval_RBI =␣
 ↪cross_val_score(bernulliBayes_RBI,RBI_features[cols],RBI_features['Sign'],scoring␣
 ↪= 'accuracy')
print("Accuracy For RBI Logit Model : %.4f"%crossval_RBI.mean())
```

```
Accuracy For EBS Logit Model : 0.4757
Accuracy For RBI Logit Model : 0.5079
```

Albeit the same techniques (eg. k-fold crossvalidation) as for the Logistic Regression can be applied, we can conclude that the Bayesian Classifier with these features is still not sufficiently good model for modelling the market movement of our two stocks.

1.3 Classifier A.2 Support Vector Machines

1.3.1 a) Soft vs. hard margin, mathematical notations and impact on 2D relationships.

Suport vector machines is a popular supervised learning technique. As a binary classifier, the method divides data according to the position of the predicted point on the side relative to hyperplane. A large functional margin would represent clear and confident classification.

The dividing hyperplane is all points such that:

$$\theta^T x + \theta_0 = 0 \tag{6}$$

where θ is the vector perpendicular to the hyperplane and $\theta_0$is a scalar. Assume we have a function:

$$g(\theta^T x + \theta_0) \tag{7}$$

where $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ if $z \leq 0$, i.e $\theta^T x^{\pm} + \theta_0 = \pm 1$ or $\theta^T (x^+ - x^-) = 2$.In such way we can define binary classification based on our input. If we note as $x^{(n)}$,M-dimensional vector representing the n-th sample and $y^{(n)}$ as the class label (+1 or -1) we have after multiplication the cases $g(z) = 1$ and $g(z) = -1$:

$$y^{(n)}(\theta^T x^{(n)} + \theta_0) - 1 \geq 0 \tag{8}$$

Knowing that $\theta^T (x^+ - x^-) = 2$ if follows that:

$$margin\_width = \frac{2}{|\theta|} \tag{9}$$

Maximising the margin is equivalent to finding $\theta$and $\theta_0$to minimize the denominator $|\theta|$ or:

$$\min_{\theta,\theta_0} \frac{1}{2}|\theta|^2 \tag{10}$$

This optimization would be no issue if there is clear separation between groups of data obsevrations. If this is not the case, one has to accept that some points will be misclassified and the problem turns into minimizing loss function. Following the Machine Learning notes from Paul Wilmott, the problem reduces to choosing $\theta$and θ_0 to minimize:

$$J = \frac{1}{N}\sum_{1}^{N} max[1 - y^{(n)}(\theta^T x^{(n)} + \theta_0), 0] + \lambda|\theta|^2 \tag{11}$$

The first term is the loss function and the second term is the regularization. We can say that λ is a weighting between the margin size and whether or not data lies on the correct side of the margin.

Soft-margin SVMs minimize training error traded off against margin. In implementing this similarly to the logistic regression, the C parameter is a regularization term, which provides a way to control overfitting: as C becomes large, it is unattractive to not respect the data at the cost of reducing the geometric margin; when it is small, it is easy to account for some data points with the use of slack variables and to have a fat margin placed so it models the bulk of the data.

For a data whose features x(i) are 2-dimensional and can be separated by a straight line, or whose features then we can say that the data is linearly separable. The goal of linear support vector machine technique is to model such data.

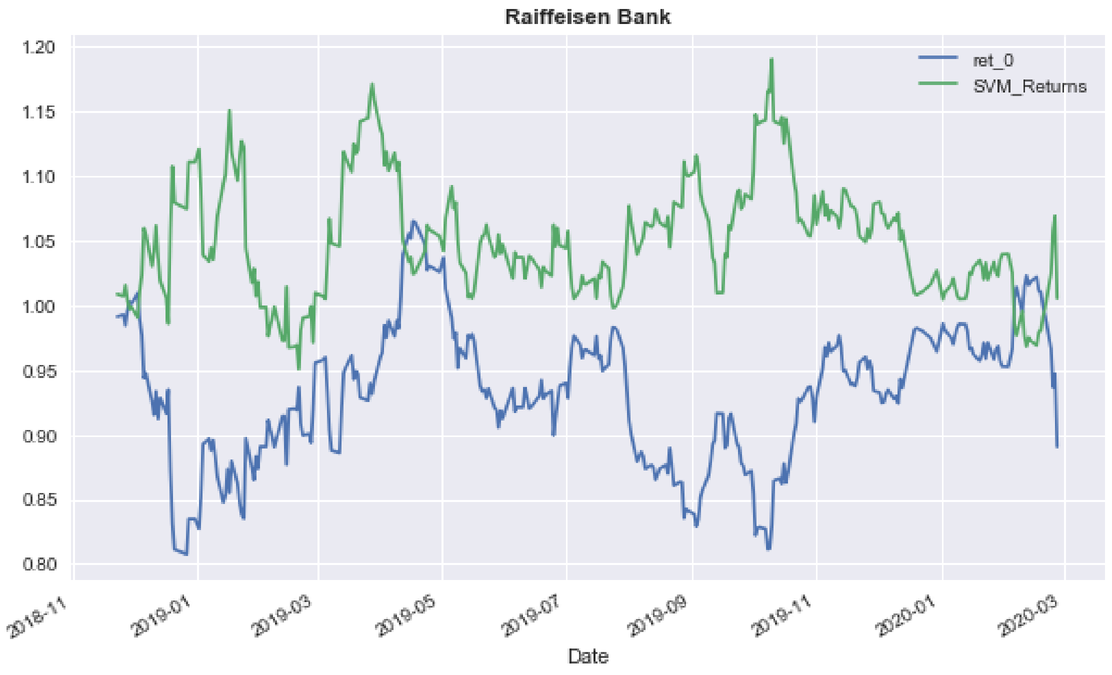
1.3.2 b) Momentum Feature vs Return t - 1, 2D visualization

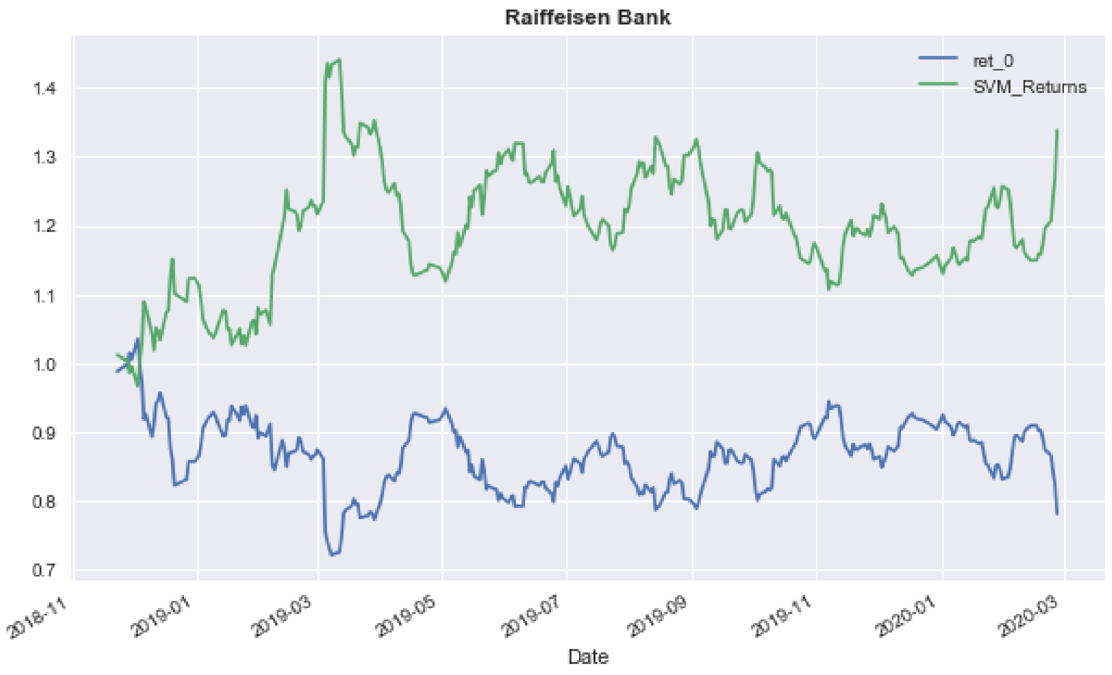
We start first with soft margins SVM models based on predictions with the first 5 lagged returns to see whether it has better performance than the logistic classifier.

[13]:
```
from sklearn.svm import LinearSVC,SVC
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

from sklearn import model_selection
```

[75]:
```
# # SVM Using only return with soft margins
# linear kernel

cols = [ 'ret_1','ret_2','ret_3','ret_4','ret_5' ]
training_split = int(0.75*EBS.shape[0])
EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

SVM_EBS =SVC(C=1e5,probability=True,kernel="linear")
SVM_RBI =SVC(C=1e5,probability=True, kernel="linear")
#fit again
SVM_EBS.fit(EBS_training[cols],EBS_training['Sign'])
SVM_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['SVM_Predict'] = SVM_EBS.predict(EBS_validate[cols])
RBI_validate['SVM_Predict'] = SVM_RBI.predict(RBI_validate[cols])


EBS_validate['SVM_Predict'][EBS_validate['SVM_Predict']==0] = -1
RBI_validate['SVM_Predict'][RBI_validate['SVM_Predict']==0] = -1

EBS_validate['SVM_Returns'] =EBS_validate['ret_0']*EBS_validate['SVM_Predict']
RBI_validate['SVM_Returns'] =RBI_validate['ret_0']*RBI_validate['SVM_Predict']

EBS_validate.tail(3)

EBS_validate[['ret_0', 'SVM_Returns']].cumsum().apply(np.exp).plot(figsize=(10,␣
 ↪6));
plt.title("Raiffeisen Bank",fontweight="bold")
RBI_validate[['ret_0', 'SVM_Returns']].cumsum().apply(np.exp).plot(figsize=(10,␣
 ↪6));
plt.title("Raiffeisen Bank",fontweight="bold")
```

```
[75]: Text(0.5, 1.0, 'Raiffeisen Bank')
```





The model performs indeed slighly better in comparison to the Logistic Regregression Classifier. It is worth mentioning that even from the plots one can notice that the model predicts better the down trend when compared to the logistic regression (eg. the end period around 2020-03).

[76]:
```
crossval_EBS =␣
 ↪cross_val_score(SVM_EBS,EBS_features[cols],EBS_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For EBS SVM Model : %.4f"%crossval_EBS.mean())
crossval_RBI =␣
 ↪cross_val_score(SVM_RBI,RBI_features[cols],RBI_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For RBI SVM Model : %.4f"%crossval_RBI.mean())
```

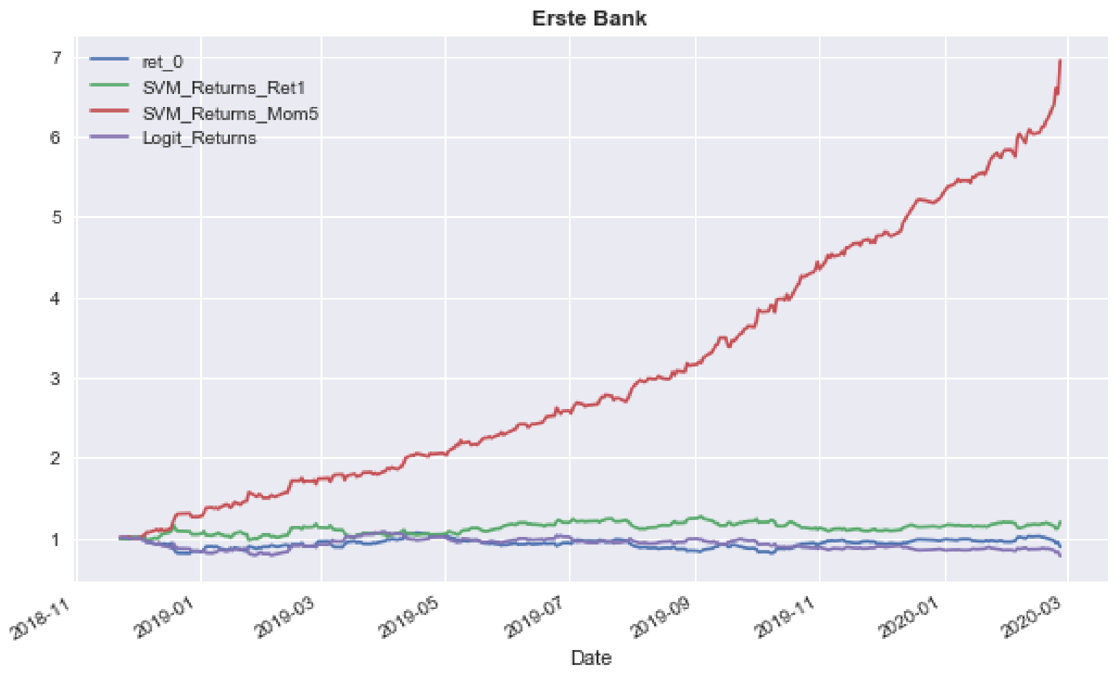
```
Accuracy For EBS SVM Model : 0.4825
Accuracy For RBI SVM Model : 0.5092
```

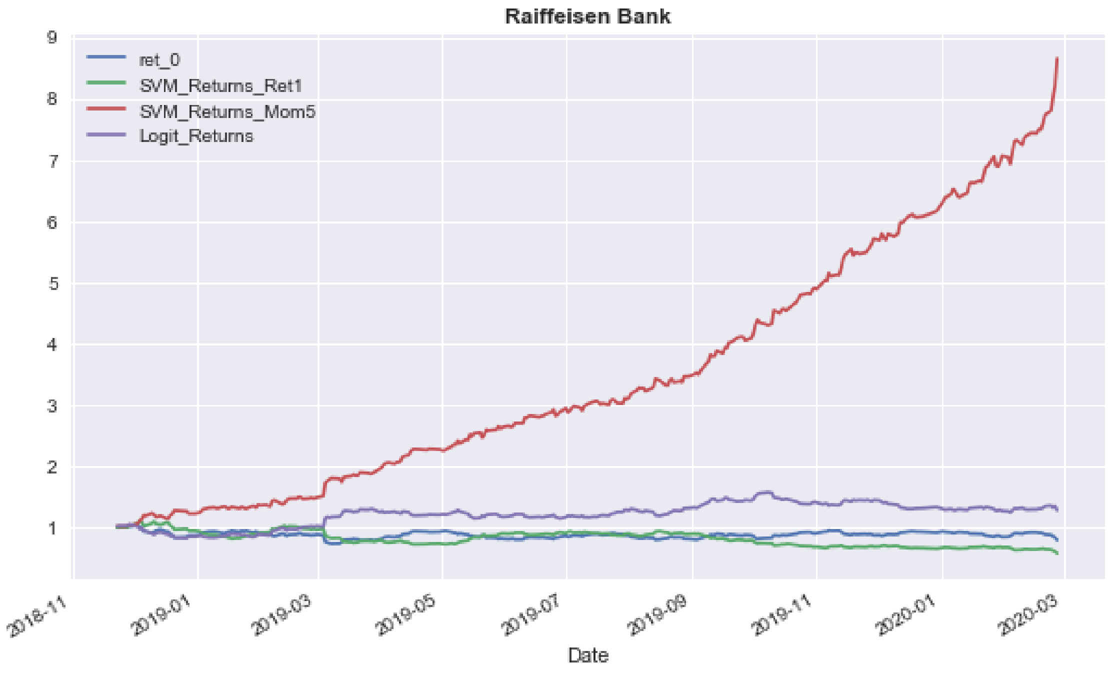
Now let us compare SVM models with soft based on predictions of previous day returns against momentum features.

[18]:
```
SVM_EBS =SVC(C=1e5,probability=True,kernel="linear")
SVM_RBI =SVC(C=1e5,probability=True, kernel="linear")

cols1 = [ 'ret_1']
cols2 = [ 'MOM5' ]
```

[86]:
```
###########################################
# SVM comparing Ret1 and Momentum with soft margins

training_split = int(0.75*EBS.shape[0])
EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

#fit again
SVM_EBS.fit(EBS_training[cols1],EBS_training['I'])
SVM_RBI.fit(RBI_training[cols1],RBI_training['I'])

SVM_EBS.fit(EBS_training[cols2],EBS_training['I'])
SVM_RBI.fit(RBI_training[cols2],RBI_training['I'])

#predict using the validation set of data
EBS_validate['SVM_Pred_Ret1'] = SVM_EBS.predict(EBS_validate[cols1])
RBI_validate['SVM_Pred_Ret1'] = SVM_RBI.predict(RBI_validate[cols1])

EBS_validate['SVM_Pred_Mom5'] = SVM_EBS.predict(EBS_validate[cols2])
RBI_validate['SVM_Pred_Mom5'] = SVM_RBI.predict(RBI_validate[cols2])
```

```
EBS_validate['SVM_Pred_Ret1'][EBS_validate['SVM_Pred_Ret1']==0] = -1
RBI_validate['SVM_Pred_Ret1'][RBI_validate['SVM_Pred_Ret1']==0] = -1


EBS_validate['SVM_Pred_Mom5'][EBS_validate['SVM_Pred_Mom5']==0] = -1
RBI_validate['SVM_Pred_Mom5'][RBI_validate['SVM_Pred_Mom5']==0] = -1


EBS_validate['SVM_Returns_Ret1']␣
 ↪=EBS_validate['ret_0']*EBS_validate['SVM_Pred_Ret1']
RBI_validate['SVM_Returns_Ret1']␣
 ↪=RBI_validate['ret_0']*RBI_validate['SVM_Pred_Ret1']

EBS_validate['SVM_Returns_Mom5']␣
 ↪=EBS_validate['ret_0']*EBS_validate['SVM_Pred_Mom5']
RBI_validate['SVM_Returns_Mom5']␣
 ↪=RBI_validate['ret_0']*RBI_validate['SVM_Pred_Mom5']

#Now include in the comparison also the old logit model
EBS_validate[['ret_0', 'SVM_Returns_Ret1', 'SVM_Returns_Mom5','Logit_Returns']].
 ↪cumsum().apply(np.exp).plot(figsize=(10, 6));
plt.title("Erste Bank",fontweight="bold")
RBI_validate[['ret_0', 'SVM_Returns_Ret1', 'SVM_Returns_Mom5','Logit_Returns']].
 ↪cumsum().apply(np.exp).plot(figsize=(10, 6));
plt.title("Raiffeisen Bank",fontweight="bold")
plt.show()
```





[78]:
```
crossval_EBS =␣
 ↪cross_val_score(SVM_EBS,EBS_features[cols],EBS_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For EBS SVM Model : %.4f"%crossval_EBS.mean())
crossval_RBI =␣
 ↪cross_val_score(SVM_RBI,RBI_features[cols],RBI_features['Sign'],scoring =␣
 ↪'accuracy')
print("Accuracy For RBI SVM Model : %.4f"%crossval_RBI.mean())
```

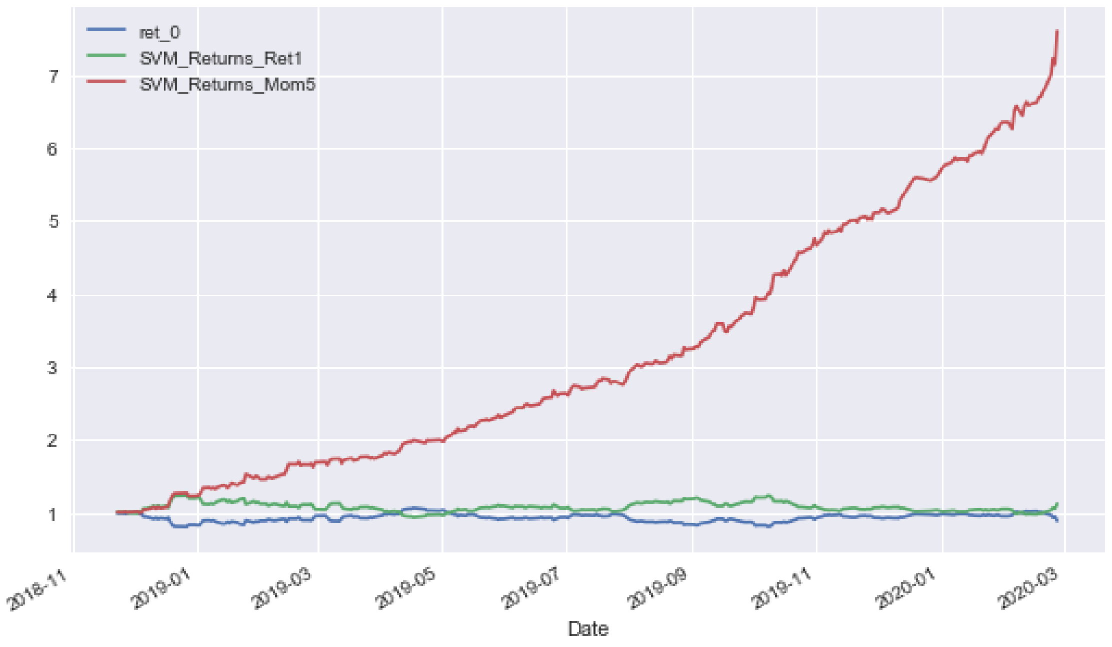
```
Accuracy For EBS SVM Model : 0.4825
Accuracy For RBI SVM Model : 0.5092
```

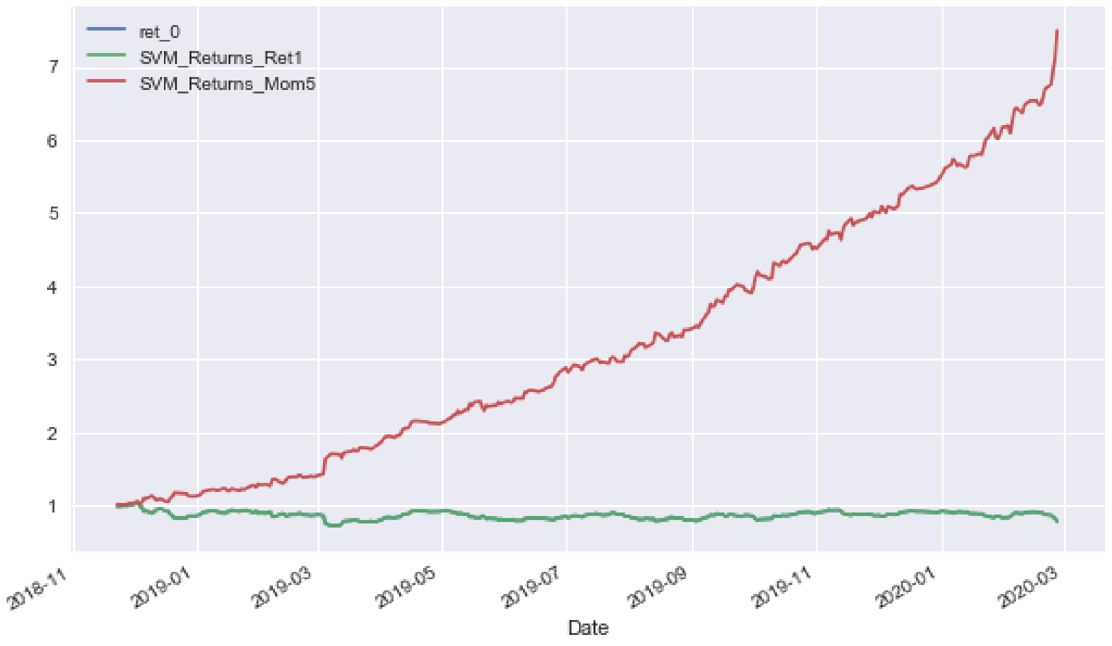
Again, the model using Momentum feature clearly outperforms all others. Using only one lag return in the case of Raiffeisen was not enough to outperform logistic regression. Let us see if setting hard margins would have an effect on the predictions.

[20]:
```
SVM_EBS_hard =SVC(C=1,probability=True, kernel="linear")
SVM_RBI_hard =SVC(C=1,probability=True, kernel="linear")
```

[64]:
```
###########################################
# SVM comparing Ret1 and Momentum with hard margins

cols1 = [ 'ret_1' ]
```

```
cols2 = [ 'MOM5' ]
training_split = int(0.75*EBS.shape[0])
EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

#fit again
SVM_EBS_hard.fit(EBS_training[cols1],EBS_training['I'])
SVM_RBI_hard.fit(RBI_training[cols1],RBI_training['I'])

SVM_EBS_hard.fit(EBS_training[cols2],EBS_training['I'])
SVM_RBI_hard.fit(RBI_training[cols2],RBI_training['I'])

#predict using the validation set of data
EBS_validate['SVM_Pred_Ret1'] = SVM_EBS_hard.predict(EBS_validate[cols1])
RBI_validate['SVM_Pred_Ret1'] = SVM_RBI_hard.predict(RBI_validate[cols1])

EBS_validate['SVM_Pred_Mom5'] = SVM_EBS_hard.predict(EBS_validate[cols2])
RBI_validate['SVM_Pred_Mom5'] = SVM_RBI_hard.predict(RBI_validate[cols2])


EBS_validate['SVM_Pred_Ret1'][EBS_validate['SVM_Pred_Ret1']==0] = -1
RBI_validate['SVM_Pred_Ret1'][RBI_validate['SVM_Pred_Ret1']==0] = -1


EBS_validate['SVM_Pred_Mom5'][EBS_validate['SVM_Pred_Mom5']==0] = -1
RBI_validate['SVM_Pred_Mom5'][RBI_validate['SVM_Pred_Mom5']==0] = -1


EBS_validate['SVM_Returns_Ret1']␣
 ↪=EBS_validate['ret_0']*EBS_validate['SVM_Pred_Ret1']
RBI_validate['SVM_Returns_Ret1']␣
 ↪=RBI_validate['ret_0']*RBI_validate['SVM_Pred_Ret1']

EBS_validate['SVM_Returns_Mom5']␣
 ↪=EBS_validate['ret_0']*EBS_validate['SVM_Pred_Mom5']
RBI_validate['SVM_Returns_Mom5']␣
 ↪=RBI_validate['ret_0']*RBI_validate['SVM_Pred_Mom5']


EBS_validate[['ret_0', 'SVM_Returns_Ret1', 'SVM_Returns_Mom5']].cumsum().
 ↪apply(np.exp).plot(figsize=(10, 6));
RBI_validate[['ret_0', 'SVM_Returns_Ret1', 'SVM_Returns_Mom5']].cumsum().
 ↪apply(np.exp).plot(figsize=(10, 6));
```





[27]:
```
crossval_EBS =␣
 ↪cross_val_score(SVM_EBS_hard,EBS_features[cols1],EBS_features['Sign'],scoring␣
 ↪= 'accuracy')
print("Accuracy For EBS SVM Hard Margins Model : %.4f"%crossval_EBS.mean())
```

```
crossval_RBI =␣
↪cross_val_score(SVM_RBI_hard,RBI_features[cols1],RBI_features['Sign'],scoring␣
↪= 'accuracy')
print("Accuracy For RBI SVM Hard Margins Model : %.4f"%crossval_RBI.mean())
```

```
Accuracy For EBS SVM Hard Margins Model : 0.5024
Accuracy For RBI SVM Hard Margins Model : 0.5045
```

The model performed with higher margins slightly better for the case of Erste Group and slightly worse for Raiffeisein when compared to models using softer margins. The difference is though not big. To examine better the effect of the margins let us plot the data.

1.3.3 2D visualisation (up/down points in different colour).

Let us inspect a model which makes predictions based on combined signals of previous day returns and 5D Momentum. We will look at the effect on the support vectors measured by standard deviation and then plot the decision boundaries. We start again with soft margins.

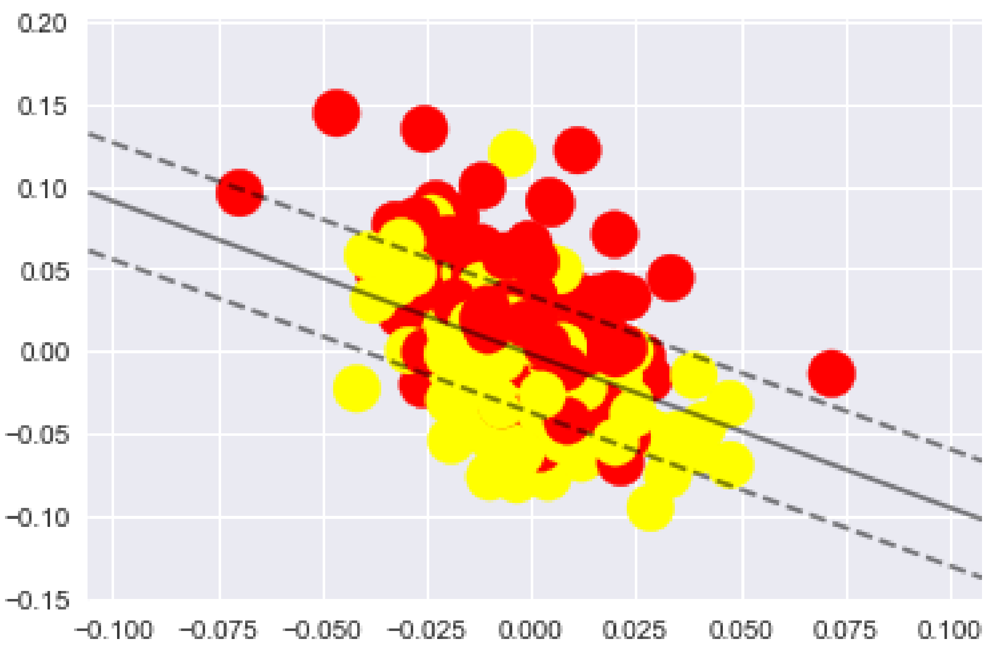
[60]:
```
###########################################
# graph for SVM

cols = ['ret_1','MOM5']
training_split = int(0.75*EBS.shape[0])
EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

SVM_EBS =SVC(C=1e5,probability=True,kernel='linear')
SVM_RBI =SVC(C=1e5,probability=True,kernel='linear')
#fit again
SVM_EBS.fit(EBS_training[cols],EBS_training['Sign'])
SVM_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['SVM_Predict'] = SVM_EBS.predict(EBS_validate[cols])
RBI_validate['SVM_Predict'] = SVM_RBI.predict(RBI_validate[cols])


EBS_validate['SVM_Predict'][EBS_validate['SVM_Predict']==0] = -1
RBI_validate['SVM_Predict'][RBI_validate['SVM_Predict']==0] = -1

EBS_validate['SVM_Returns'] =EBS_validate['ret_0']*EBS_validate['SVM_Predict']
RBI_validate['SVM_Returns'] =RBI_validate['ret_0']*RBI_validate['SVM_Predict']

EBS_validate.tail(3)
```

```
#print the support vectors
print('         Support Vectors for EBS')
df = pd.DataFrame(SVM_EBS.support_vectors_)
df.columns = cols
df.head(3)
df.std()
```

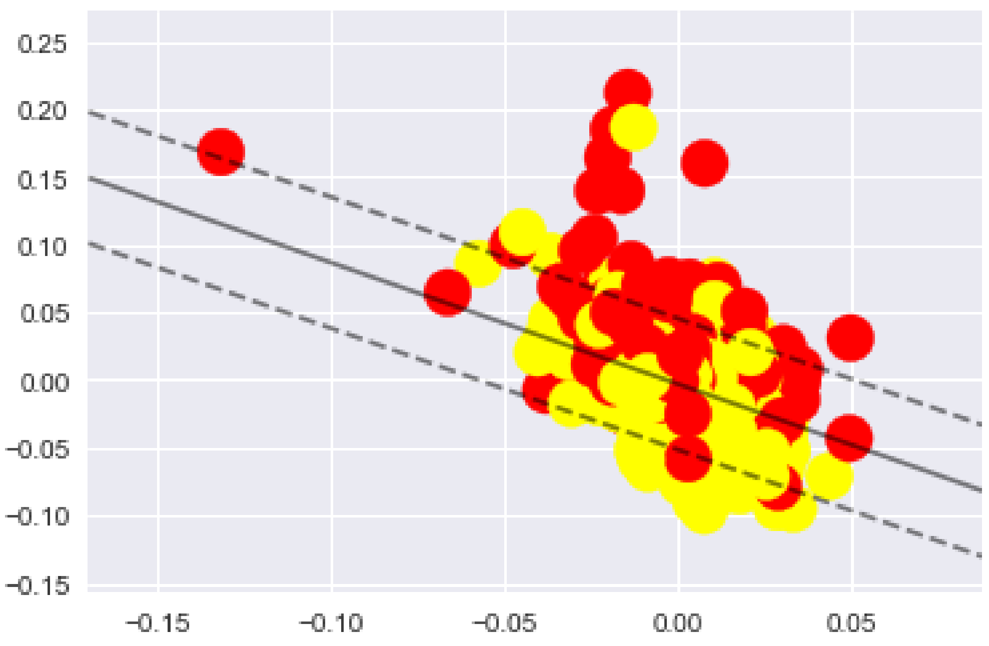
```
         Support Vectors for EBS
```

```
[60]: ret_1    0.017962
      MOM5     0.029260
      dtype: float64
```

```
[66]: def plot_svc_decision_function(model, ax=None, plot_support=True):
          """Plot the decision function for a 2D SVC"""
          if ax is None:
              ax = plt.gca()
          xlim = ax.get_xlim()
          ylim = ax.get_ylim()

          # create grid to evaluate model
          x = np.linspace(xlim[0], xlim[1], 30)
          y = np.linspace(ylim[0], ylim[1], 30)
          Y, X = np.meshgrid(y, x)
          xy = np.vstack([X.ravel(), Y.ravel()]).T
          P = model.decision_function(xy).reshape(X.shape)

          # plot decision boundary and margins
          ax.contour(X, Y, P, colors='k',
                     levels=[-1, 0, 1], alpha=0.5,
                     linestyles=['--', '-', '--'])

          # plot support vectors
          if plot_support:
              ax.scatter(model.support_vectors_[:, 0],
                         model.support_vectors_[:, 1],
                         s=300, linewidth=1, facecolors='none');
          ax.set_xlim(xlim)
          ax.set_ylim(ylim)

      plt.scatter(EBS_validate['ret_1'], EBS_validate['MOM5'],␣
       ↪c=EBS_validate['Sign'], s=len(EBS_validate["ret_1"]), cmap='autumn')
      plot_svc_decision_function(SVM_EBS);
```



[68]:
```
plt.scatter(RBI_validate['ret_1'], RBI_validate['MOM5'],␣
 ↪c=RBI_validate['Sign'], s=len(RBI_validate["ret_1"]), cmap='autumn')
plot_svc_decision_function(SVM_RBI);
```



As we can see there is no perfect separation between the two groups of data observations since some points appear to be on the wrong side of the hyperplane. If these “misclassiefied” points are not too far from the hyperplane then increasing the margins could improve the model.

Next, we apply hard margins.

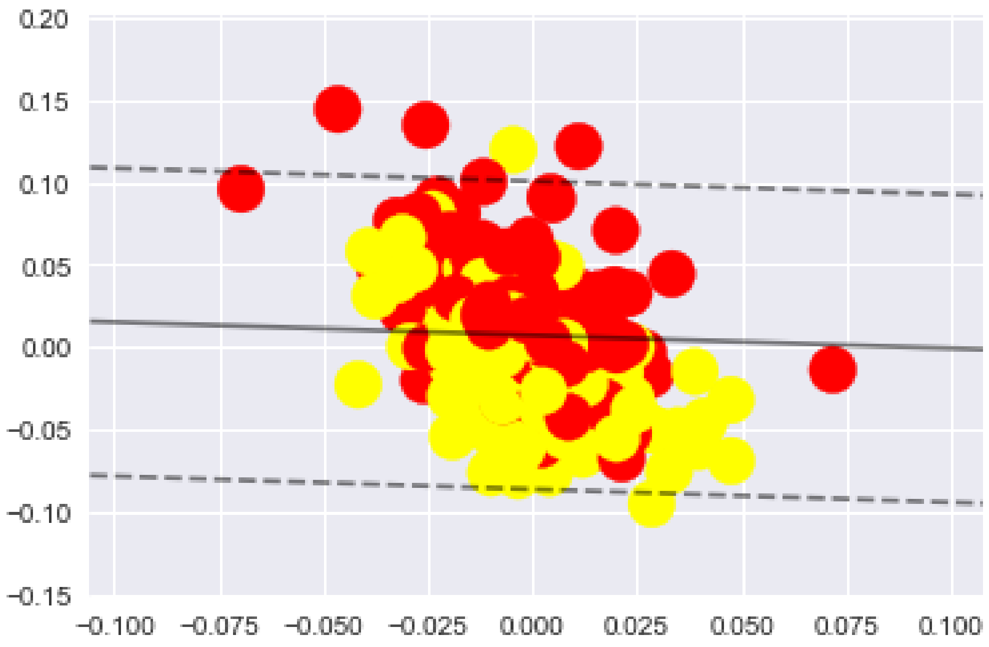
```
[69]: cols = ['ret_1','MOM5']
training_split = int(0.75*EBS.shape[0])
EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

SVM_EBS =SVC(C=1,probability=True,kernel='linear')
SVM_RBI =SVC(C=1,probability=True,kernel='linear')
#fit again
SVM_EBS.fit(EBS_training[cols],EBS_training['Sign'])
SVM_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['SVM_Predict'] = SVM_EBS.predict(EBS_validate[cols])
RBI_validate['SVM_Predict'] = SVM_RBI.predict(RBI_validate[cols])


EBS_validate['SVM_Predict'][EBS_validate['SVM_Predict']==0] = -1
RBI_validate['SVM_Predict'][RBI_validate['SVM_Predict']==0] = -1

EBS_validate['SVM_Returns'] =EBS_validate['ret_0']*EBS_validate['SVM_Predict']
RBI_validate['SVM_Returns'] =RBI_validate['ret_0']*RBI_validate['SVM_Predict']

EBS_validate.tail(3)

#print the support vectors
print('         Support Vectors for EBS')
df = pd.DataFrame(SVM_EBS.support_vectors_)
df.columns = cols
df.std()
```

```
         Support Vectors for EBS
```

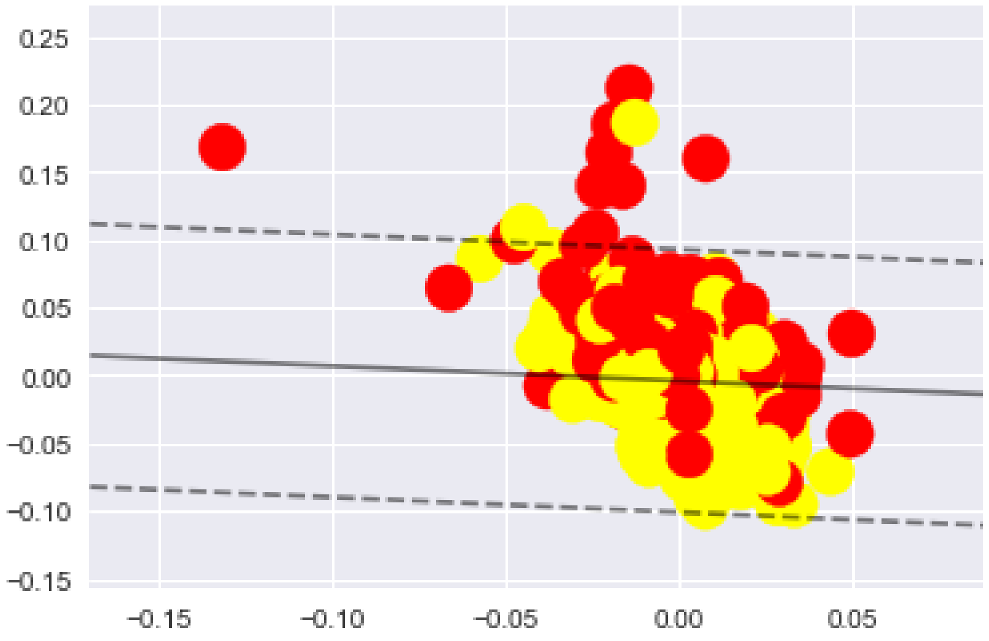
```
[69]: ret_1    0.017229
MOM5     0.035086
dtype: float64
```

We can see that the standard deviation increased slightly, however increasing further the margins should in general decrease the deviation.

[70]:
```
  plt.scatter(EBS_validate['ret_1'], EBS_validate['MOM5'],␣
↪c=EBS_validate['Sign'], s=len(EBS_validate["ret_1"]), cmap='autumn')
plot_svc_decision_function(SVM_EBS);
```



[71]:
```
plt.scatter(RBI_validate['ret_1'], RBI_validate['MOM5'],␣
↪c=RBI_validate['Sign'], s=len(RBI_validate["ret_1"]), cmap='autumn')
plot_svc_decision_function(SVM_RBI);
```

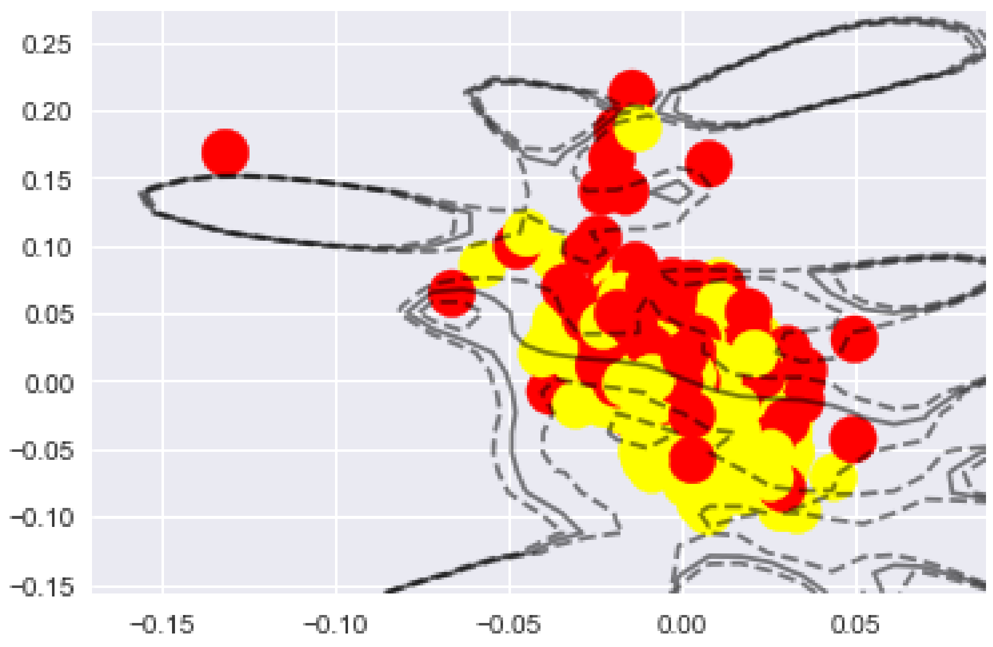


In the case of Erste Group and even more strongly in the case of Raiffeisen Bank there are some clear outliers. By increasing C we are left with less data observations considered and the outliers are hence overpresented. The higher the value of the C multiplier, the more SVM will be sensitive to noise.

Conclusion: soft margins would allow more data points and as data is not that well segregated and there is mixing over, soft margins would be a better option. Ideally for such data, non linear kernels could achieve better classification and 3D view can allow for better visual inspection.

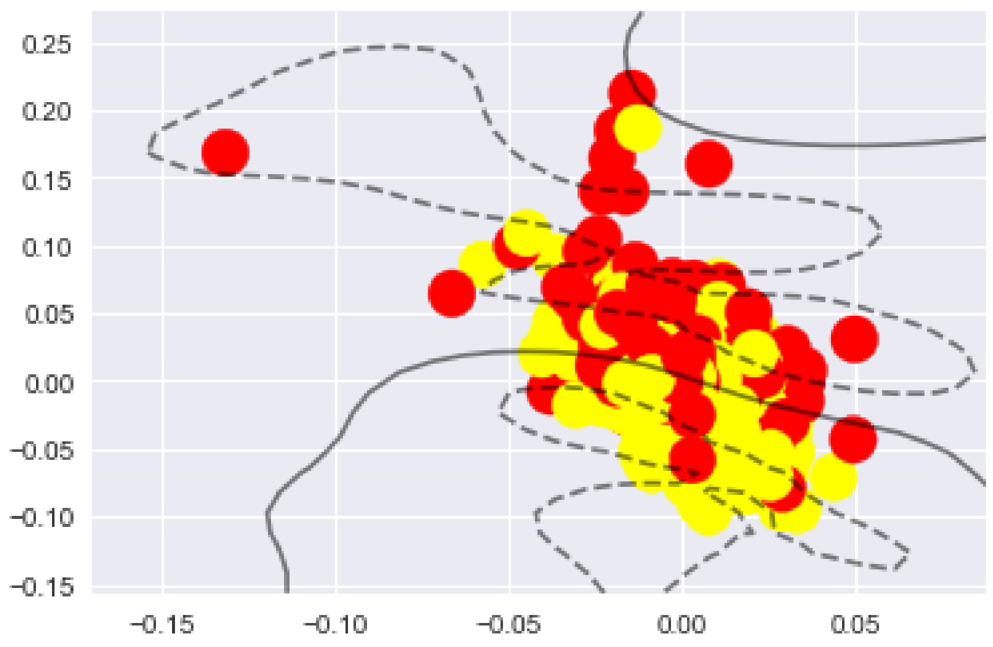
Finally, we plot the results of the model applying non-linear kernel.

```
[72]: # graph for SVM with soft margins and non-linear kernel

cols = ['ret_1','MOM5']
training_split = int(0.75*EBS.shape[0])
RBI_training = RBI_features.iloc[0:training_split,:]

RBI_validate = RBI_features.iloc[training_split:,:]

SVM_RBI =SVC(C=1e5,probability=True)
#fit again
SVM_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
RBI_validate['SVM_Predict'] = SVM_RBI.predict(RBI_validate[cols])

RBI_validate['SVM_Predict'][RBI_validate['SVM_Predict']==0] = -1

RBI_validate['SVM_Returns'] =RBI_validate['ret_0']*RBI_validate['SVM_Predict']

plt.scatter(RBI_validate['ret_1'], RBI_validate['MOM5'],␣
 ↪c=RBI_validate['Sign'], s=len(RBI_validate["ret_1"]), cmap='autumn')
plot_svc_decision_function(SVM_RBI);
```



Nonlinear kernel seem to be more flexible, however one should be aware also of the risk of overfittin especially when using soft margins. From the plot we can see that the model attempts to consider every single point. The increased comlpexity of the model has been felt also on the system time needed for the computer to run it. Now we plot the same model but with larger margins.

[73]:
```
cols = ['ret_1','MOM5']
training_split = int(0.75*EBS.shape[0])
RBI_training = RBI_features.iloc[0:training_split,:]

RBI_validate = RBI_features.iloc[training_split:,:]

SVM_RBI =SVC(C=1,probability=True)
#fit again
SVM_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
RBI_validate['SVM_Predict'] = SVM_RBI.predict(RBI_validate[cols])

RBI_validate['SVM_Predict'][RBI_validate['SVM_Predict']==0] = -1

RBI_validate['SVM_Returns'] =RBI_validate['ret_0']*RBI_validate['SVM_Predict']

plt.scatter(RBI_validate['ret_1'], RBI_validate['MOM5'],␣
 ↪c=RBI_validate['Sign'], s=len(RBI_validate["ret_1"]), cmap='autumn')
plot_svc_decision_function(SVM_RBI);
```



The harder margins allow us to spot easily the groups and boundaries identified by the computer.

Non-linear kernel is per se more flexible and cases of no clear separation of groups can in fact achieve better results than when trying to fit linear hyperplane.

1.4 A.3 Classifer K-Nearest Neighbours

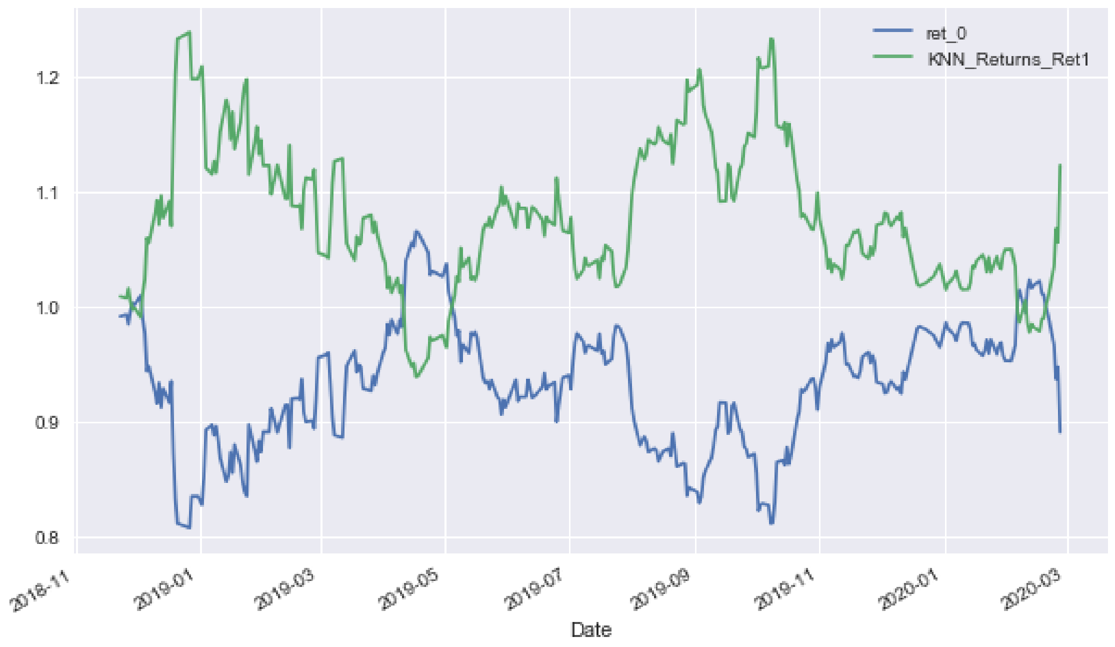
K-nearest neighbours (or KNN) is a distance-based algorithm where data is classified based on proximity to K-Neighbors. The principle applied by the K-nearest Neighbours learning approach is that objects that are alike are more likely to have properties that are alike.

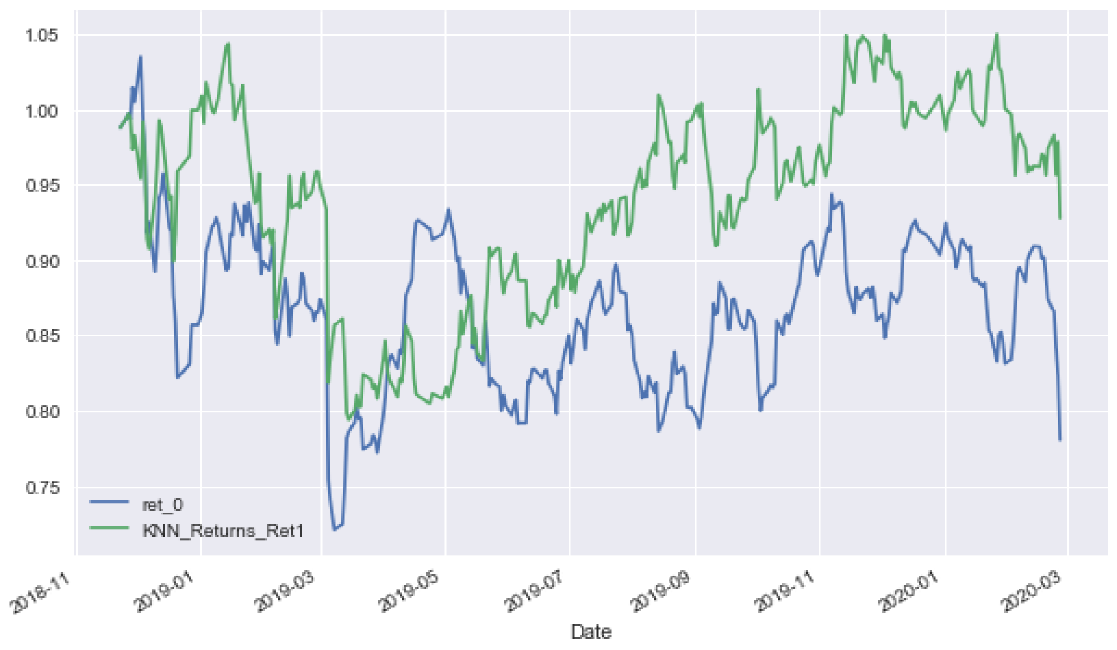
1.4.1 a) scale the features, eg, StandardScaler from sklearn.preprocessing.

This principle used to classify data is by placing it in the category with which it is most similar, or “nearest” neighbors.Some of features of the data we use are not at the same scale (for instance EWMA crossover strategy vs Momemnutm or standard deviations). Thus it might make sense to scale the features first. We will then implement the KNN classifier testing it again with our set of lagged returns like we did so far for the other classifiers.

[89]:
```python
from sklearn.preprocessing import StandardScaler
EBS_features[cols] = StandardScaler().fit_transform(EBS_features[cols])

from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)

cols1 = [ 'ret_1','ret_2','ret_3','ret_4','ret_5' ]
#EBS
```

```
knn.fit(EBS_training[cols1],EBS_training['I'])

EBS_validate['KNN_Pred_Ret1'] = knn.predict(EBS_validate[cols1])

EBS_validate['KNN_Pred_Ret1'][EBS_validate['KNN_Pred_Ret1']==0] = -1

EBS_validate['KNN_Returns_Ret1']␣
 ↪=EBS_validate['ret_0']*EBS_validate['KNN_Pred_Ret1']

EBS_validate[['ret_0', 'KNN_Returns_Ret1']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));

#RBI
knn.fit(RBI_training[cols1],RBI_training['I'])

RBI_validate['KNN_Pred_Ret1'] = knn.predict(RBI_validate[cols1])

RBI_validate['KNN_Pred_Ret1'][RBI_validate['KNN_Pred_Ret1']==0] = -1

RBI_validate['KNN_Returns_Ret1']␣
 ↪=RBI_validate['ret_0']*RBI_validate['KNN_Pred_Ret1']

RBI_validate[['ret_0', 'KNN_Returns_Ret1']].cumsum().apply(np.exp).
 ↪plot(figsize=(10, 6));
```





The model with this set of features is in fact worse than the logistic regression and SVM. We should find an optimal number of "neighbours" in order to improve the model.

1.4.2 b) Report on sensible values,comparsion etc.

We will test a set of features of Momentum and Return t-1 signals.

Firstly, instead of repeating the previous procedure we make use of the model_selection function and testing the model with different number of neighbours k

[103]:
```
from sklearn import model_selection
from sklearn import metrics

EBS_features['I'] = EBS_features['ret_0']
EBS_features['I'][EBS_features['I']>0] = 1
EBS_features['I'][EBS_features['I']<=0] = -1

cols=['MOM5','ret_1']

#plot_logit_ROC(default_ind,Y_test,X_test,logit,Y_response,X_features)
X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
↪train_test_split(EBS_features[cols],                                         ␣
↪    EBS_features['I'] ,                                                      ␣
↪    test_size=0.25, shuffle=True)
```

```
#k=5, number of neighbours
 ↪
knn5 = KNeighborsClassifier(n_neighbors=5)

knn5.fit(X_Train_EBS,Y_Train_EBS)
Y_EBS_P = knn5.predict(X_Test_EBS)

print("Accuracy with k=5:",metrics.accuracy_score( Y_Test_EBS, Y_EBS_P))

#k=7, number of neighbours
knn7 = KNeighborsClassifier(n_neighbors=7)

knn7.fit(X_Train_EBS,Y_Train_EBS)
Y_EBS_P = knn7.predict(X_Test_EBS)

print("Accuracy with k=7:",metrics.accuracy_score( Y_Test_EBS, Y_EBS_P))

#generally,increased number of neighbors leads to more accuracy
#but not always

#k=13, number of neighbours
knn13 = KNeighborsClassifier(n_neighbors=13)

knn13.fit(X_Train_EBS,Y_Train_EBS)
Y_EBS_P = knn13.predict(X_Test_EBS)

print("Accuracy with k=13:",metrics.accuracy_score( Y_Test_EBS, Y_EBS_P))
#accuracy fell in fact
```

```
Accuracy with k=5: 0.5561643835616439
Accuracy with k=7: 0.6
Accuracy with k=13: 0.5698630136986301
```

To find the exact point by which increasing k doesn't improve accuracy we run the following code:

[104]:
```
#let's find with a loop the optimal number for k
for k in range(3,20):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_Train_EBS,Y_Train_EBS)
    Y_EBS_P = knn.predict(X_Test_EBS)
    print("Accuracy with k=:{} is".format(k),metrics.accuracy_score(␣
 ↪Y_Test_EBS, Y_EBS_P))
```

```
Accuracy with k=:3 is 0.5561643835616439
Accuracy with k=:4 is 0.5835616438356165
Accuracy with k=:5 is 0.5561643835616439
Accuracy with k=:6 is 0.5917808219178082
Accuracy with k=:7 is 0.6
Accuracy with k=:8 is 0.5972602739726027
Accuracy with k=:9 is 0.6027397260273972
Accuracy with k=:10 is 0.6027397260273972
Accuracy with k=:11 is 0.5972602739726027
Accuracy with k=:12 is 0.6027397260273972
Accuracy with k=:13 is 0.5698630136986301
Accuracy with k=:14 is 0.589041095890411
Accuracy with k=:15 is 0.5698630136986301
Accuracy with k=:16 is 0.5808219178082191
Accuracy with k=:17 is 0.5726027397260274
Accuracy with k=:18 is 0.5972602739726027
Accuracy with k=:19 is 0.5452054794520548
```

As we can see there in the beginning increasing k leads to higher accuracy, however for k>12 the accuracy starts to diminish.

As a next step, we will examine the influence of the distance metric. The distance metric helps algorithms to recognize similarities between the contents. It uses distance function which provides a relationship metric between each elements in the dataset.

The distance can be calculated by the following formula:

$$(\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p} \tag{12}$$

with p = 1, Manhattan Distance
p = 2, Euclidean Distance

If the Manhattan Distance resembles grid search and the Euclidean is simply distance in a plane, Mahalanobis Distance is used for calculating the distance between two data points in a multivariate space. Mahalanobis Distance considers how many standard deviations away certain point is from the mean of the respective distribution. The strength of using mahalanobis distance is, it takes covariance in account which helps in measuring similarity between two different data objects.

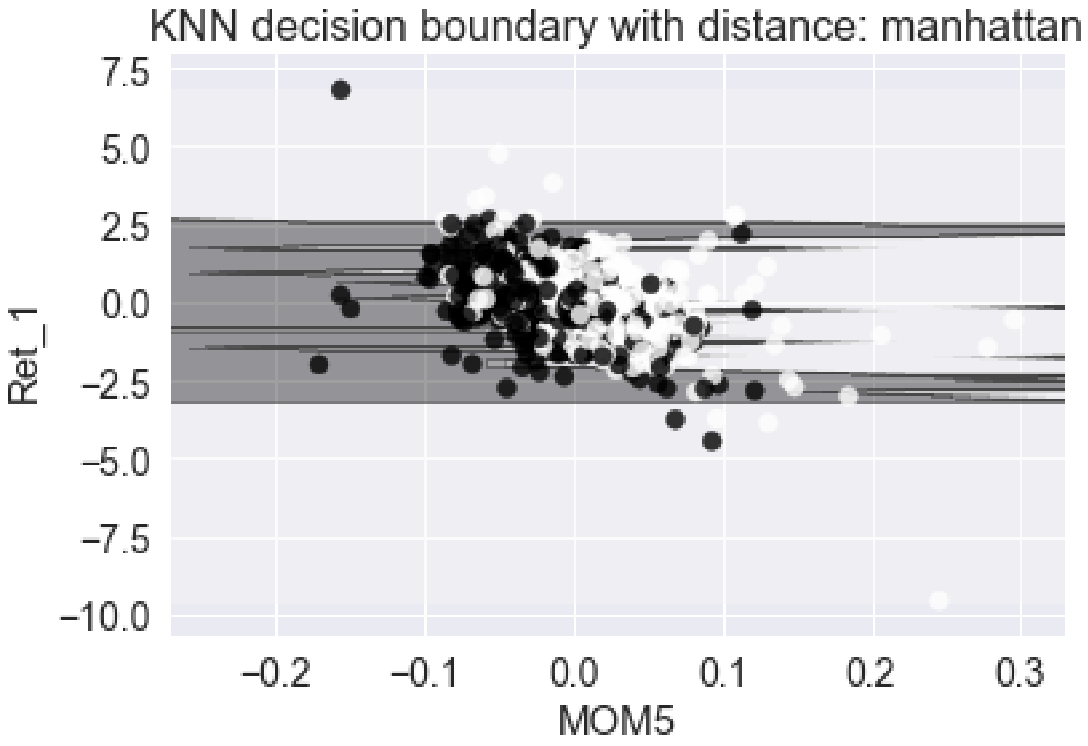
1.4.3 c) Plot decision boundaries and describe "lazy" classification

[100]:
```
def plot_decision_boundaries(X, y, model_class, **model_params):
    """
    Function to plot the decision boundaries of a classification model.
    This uses just the first two columns of the data for fitting
    the model as we need to find the predicted value for every point in
    scatter plot.
    Arguments:
            X: Feature data as a NumPy-type array.
            y: Label data as a NumPy-type array.
            model_class: A Scikit-learn ML estimator class
            e.g. GaussianNB (imported from sklearn.naive_bayes) or
            LogisticRegression (imported from sklearn.linear_model)
            **model_params: Model parameters to be passed on to the ML estimator

    """
    try:
        X = np.array(X)
        y = np.array(y).flatten()
    except:
        print("Coercing input data to NumPy arrays failed")
    # Reduces to the first two columns of data
    reduced_data = X[:, :2]
    # Instantiate the model object
    model = model_class(**model_params)
    # Fits the model with the reduced data
    model.fit(reduced_data, y)

    # Step size of the mesh. Decrease to increase the quality of the VQ.
    h = .02     # point in the mesh [x_min, m_max]x[y_min, y_max].

    # Plot the decision boundary. For that, we will assign a color to each
    x_min, x_max = reduced_data[:, 0].min() - 0.1, reduced_data[:, 0].max() + 0.
↪1
    y_min, y_max = reduced_data[:, 1].min() - 0.1, reduced_data[:, 1].max() + 0.
↪1
    # Meshgrid creation
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Obtain labels for each point in mesh using the model.
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() -0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))

    # Predictions to obtain the classification results
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Plotting
    plt.contourf(xx, yy, Z, alpha=0.4)
    plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.8)
    plt.xlabel("MOM5",fontsize=15)
    plt.ylabel("Ret_1",fontsize=15)
    plt.xticks(fontsize=14)
    plt.yticks(fontsize=14)
    return plt
#This function is referenced to towardsdatascience.com
```

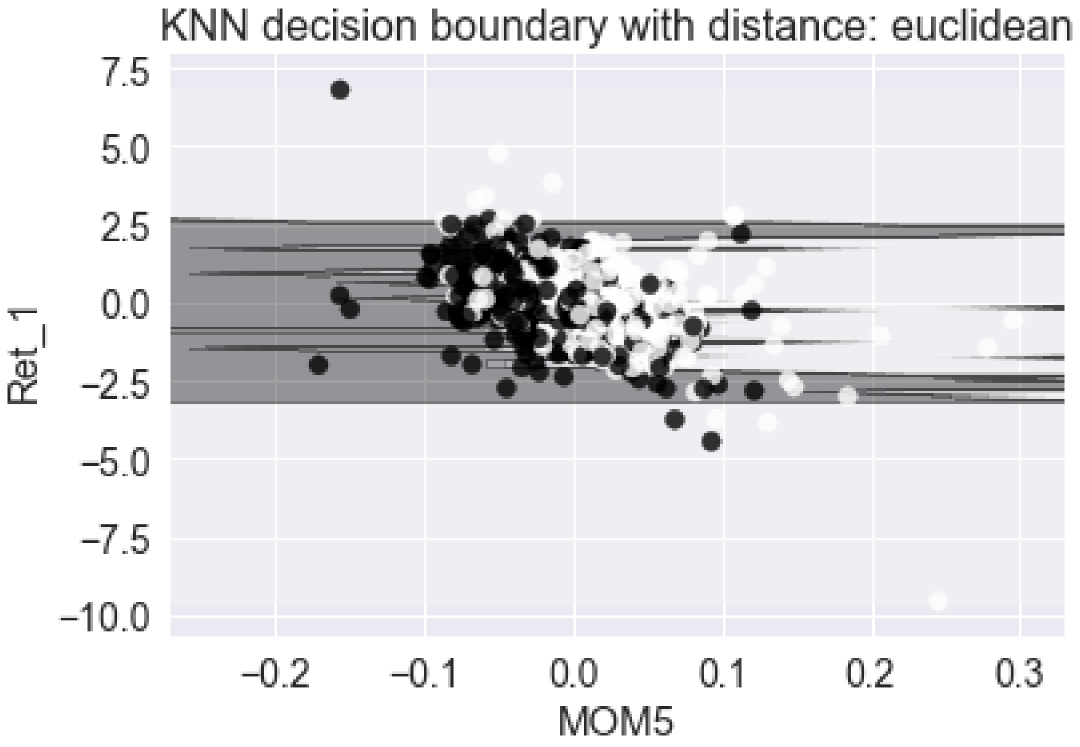
Now let's compare accuracy with different distance metrics and plot the decision boundaries.

```
[113]: for dist in ["manhattan", "euclidean", "mahalanobis"]:
           if (dist=="mahalanobis"):
               knn = KNeighborsClassifier(n_neighbors=5, metric=dist,␣
        ↪metric_params={'V': np.cov(X_Train_EBS, rowvar=False)})
           else:
               knn = KNeighborsClassifier(n_neighbors=5, metric=dist)
           knn.fit(X_Train_EBS,Y_Train_EBS)
           Y_EBS_P = knn.predict(X_Test_EBS)
           print("Accuracy with {} distance metric is".format(dist),metrics.
        ↪accuracy_score( Y_Test_EBS, Y_EBS_P))
           plt.figure()
           plt.title("KNN decision boundary with distance: {}".
        ↪format(dist),fontsize=16)
           if (dist=="mahalanobis"):
               ␣
        ↪plot_decision_boundaries(X_Train_EBS,Y_Train_EBS,KNeighborsClassifier,n_neighbors=5,metric=
        ↪ np.cov(X_Train_EBS, rowvar=False)})
           else:
               ␣
        ↪plot_decision_boundaries(X_Train_EBS,Y_Train_EBS,KNeighborsClassifier,n_neighbors=5)
           plt.show()
```

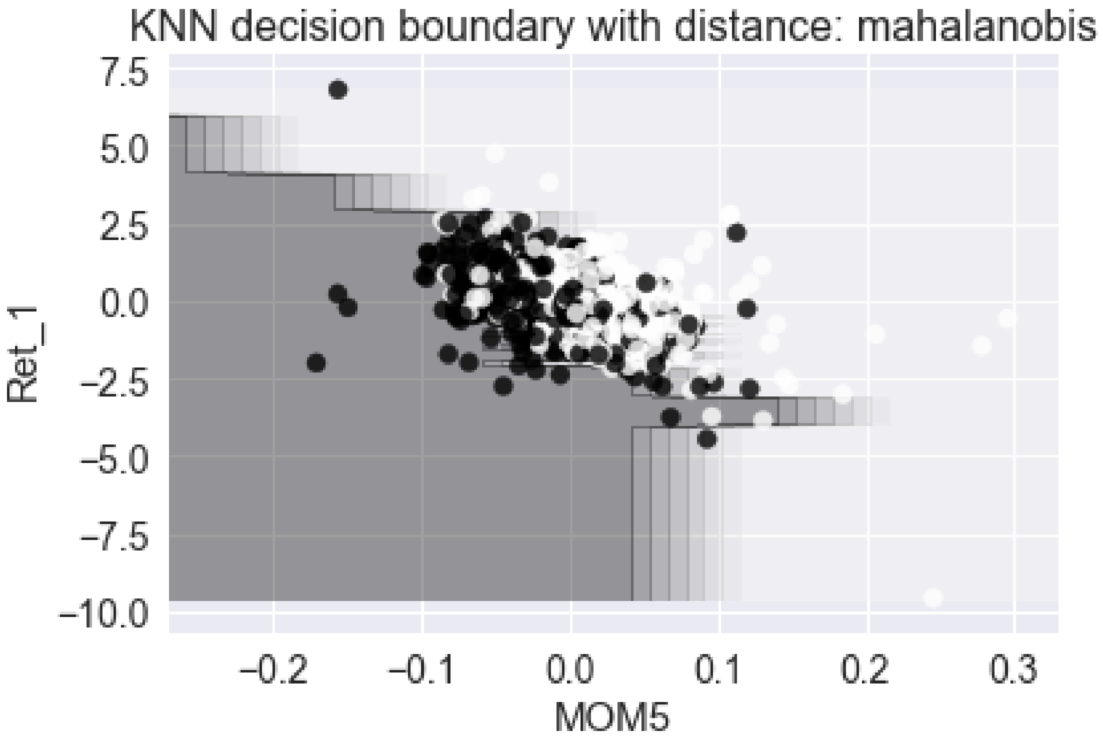
```
Accuracy with manhattan distance metric is 0.5808219178082191
```



```
Accuracy with euclidean distance metric is 0.5561643835616439
```



```
Accuracy with mahalanobis distance metric is 0.6383561643835617
```



The mahalanobis distance was used by the most accurate model. The main difference comes how outliers are considered. The models with manhattan and euclidean distance metrics seem to have considered outlier points which are far from the bulk of points of their class. The model with mahalanobis metric have ignored thodse point and in such way considered more homogeneous data.

Why is the classification called "lazy" The K-nearest neighbours is considered lazy because no abstraction occurs, i.e. the abstraction and generalization processes are not part of it. In contrast to other learning techniques where raw input is summarized into a model (equations, decision trees, clustering, if then rules), a lazy learner is not really learning anything. Instead, it is only storing the training data, which takes very little time. Classification, however, is very slow. This is unlike most classifiers in which training takes a long time, but classification is very fast.

1.5 B. Prediction Quality and Bias

1.6 B.1 Investigation using confusion matrix & ROC curve

So far to compare our models and to evaluate how good they are we have applied cross-validation and checked accuracy. In this part we will include two other metrics:

-AUC Validation, or Area under ROC curve validation.
-Confusion Matrix Validation.

In AUC Validation, ROC plots the relationship between True Positive Rate (TPR) and False Positive Rate(FPR), where

$$TRP = \frac{TP}{TP + FN} \tag{13}$$

$$FPR = \frac{FP}{FP + TN} \tag{14}$$

TP and FP stand for true and false positives resp., whereas TN and FN for true and false negatives. The idea behind TP is that if the actual outcome should be positive and our predicted outcome is also positive, then we can say we have true positive. Analogically are defined the other terms. If a model classifies data in right order, the TP Rate will grow faster than FN Rate, thus the area under ROC curve will be also larger and growing in faster pace to 1.

In contrast to the AUC validation which has its focus detecting True Positive Rate and False Positive Rates, confusion matrix focuses more on True Positive Number and True Negative Number.

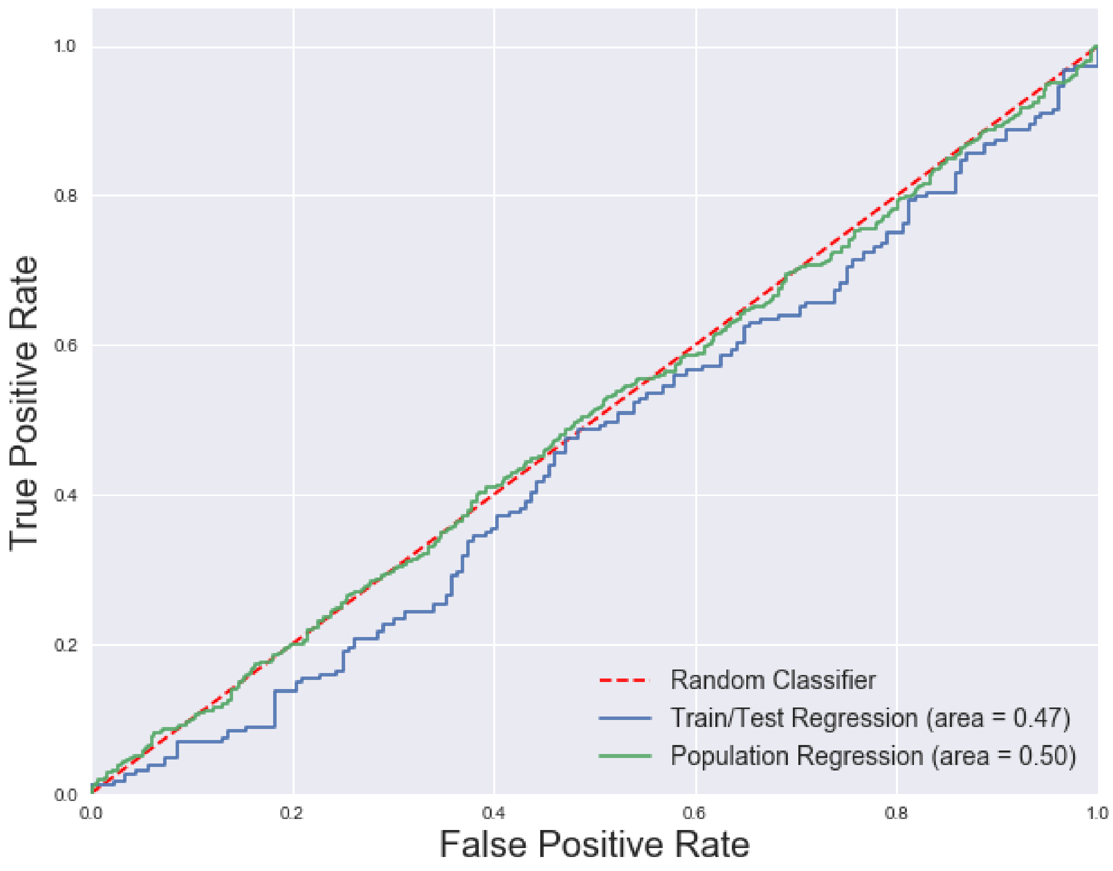
Now let us apply these validation techniques on our classifiers.

1.6.1 Logistic Regression

Firstly, we will define our ROC plotting function.

[11]:
```
#module we use for auc
from sklearn.metrics import roc_curve,roc_auc_score,auc
#module we will use for confusion matrix analysis
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import StratifiedKFold



#we first define a function to plot ROC/AUC
def plot_logit_ROC(default_ind,Y_test,X_test,logit,Y_response,X_features):
    #logit: We pass a defined LogisticRegression Object here
    #Y-Response: Y Population
    #X_Features: X Population
    #Y_test: testing set Y
    #X_test: testing set X

    # (1) Calculate AUC on testing set
    logit_roc_aucT = roc_auc_score(Y_test, logit.predict(X_test))
    # (2) calculate False Positive Rate and True Positive Rate on test set
    fprT,tprT,thresholdsT = roc_curve(Y_test,logit.predict_proba(X_test)[:
 ↪,1],pos_label = default_ind)

    #(3)Calculate AUC on population testing set
    logit_roc_aucP = roc_auc_score(Y_response,logit.predict(X_features))
    #(4)calculate False Positive Rate and True Positive Rate on Population
    fprP,tprP,threasholdsP = roc_curve(Y_response,logit.
 ↪predict_proba(X_features)[:,1],pos_label=default_ind)

    fig,ax = plt.subplots(figsize=(10,8))

    #Plot diagnoal line
    ax.plot([0,1],[0,1],'r--',label = "Random Classifier")

    #Plot ROC for predictions on testing set
    ax.plot(fprT,tprT,label = 'Train/Test Regression (area = %0.
 ↪2f)'%logit_roc_aucT)

    #Plot ROC curve for the prediction on full set
    ax.plot(fprP,tprP,label = 'Population Regression (area = %0.
 ↪2f)'%logit_roc_aucP)

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate', fontsize=20)
    plt.ylabel('True Positive Rate', fontsize=20)
    plt.legend(loc="lower right", fontsize=14)
    plt.show()

    return ax
```

[116]:
```
cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']

X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
 ↪train_test_split(EBS_features[cols],
                                                                                   ␣
 ↪    EBS_features['I'] ,
                                                                                   ␣
 ↪    test_size=0.25, shuffle=True)

lm_EBS = linear_model.LogisticRegression(C = 1e15,solver ='liblinear',penalty =␣
 ↪'l2')
lm_EBS.fit(X_Train_EBS,Y_Train_EBS)


plot_logit_ROC(1,Y_Test_EBS,X_Test_EBS,lm_EBS,EBS_features['I'],EBS_features[cols])
```



```
[116]: <matplotlib.axes._subplots.AxesSubplot at 0x54222c8>
```

A good model is expected to output Train/Test Regression area at least higher than 0.50, otherwise we can say that a random classifier would have been better than our model. The logistic regression based on lagged returns is not robust model according to the AUC validation. The confusion matrix can give us more precise idea why the model fails.

```
[118]: #print confusion matrix
Y_EBS_pred = lm_EBS.predict(X_Test_EBS)
confusion_matrix_EBS = confusion_matrix(Y_Test_EBS,Y_EBS_pred)
print("Confusion Matrix - Logistic Regression")
print(confusion_matrix_EBS)
#this is clearly bad model, a random model could have been better
```

```
Confusion Matrix
[[131  45]
 [152  37]]
```

The confusion matrix contains four elements. The upper-left element is the number of True Positive Samples, which is 131, and the right-upper element is the number of False Negative Samples, which is 45. So, the actual postive sample number should be 131 + 45 = 176, and our model catches

131/176 actual positive sample correctly, which is not that bad actually. The problem is when the model handles negative sample. The lower-left element is the number of False Positive Samples, which is 152, and the right-lower element is the number of True Negative Samples, which is 37. So, the actual negative sample number should be 152 +37 = 189, and our model predicts 37/189 actual negative samples correctly, which is undoubtedly poor result and it has made the overall model performance to deteriorate significantly.

Last but not least, we have added Precision, Recall and F1 Score to the Accuracy Score that we measured so far. The calculation of these socres are summarized below:

$$Accuracy = \frac{TP + TN}{Total} \tag{15}$$

$$Precision = \frac{TP}{TP + FP} \tag{16}$$

$$Recall = \frac{TP}{TP + FN} \tag{17}$$

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{18}$$

Accuracy is defined as the number of true positives and true negatives divided by the number of true positives, true negatives, false positives, and false negatives. Previously, we have used the accuracy as model benchmark. However, to fully evaluate the effectiveness of a model, one must examine both precision and recall. Precision shows what proportion of positive identifications was actually correct, whereas Recall shows what proportion of actual positives was identified correctly.

Since precision and recall are often in tension (improving precision typically reduces recall and vice versa), we will compute also F-1 score, which is a harmonic mean of precision and recall. For problems where both precision and recall are important, one can select a model which maximizes this F-1 score (we will see example in task B.2). Let us compute the metrics now:

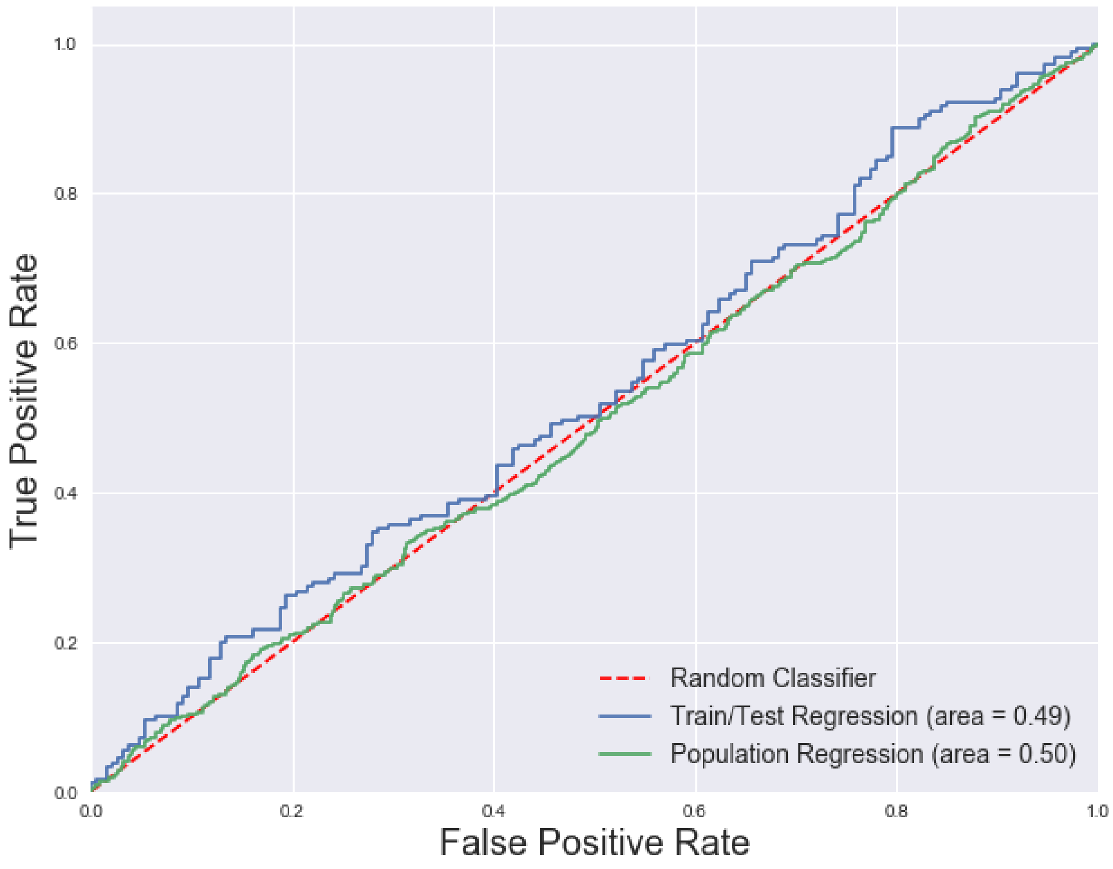
[119]:
```
from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_pred)))
print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_pred)))
print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_pred)))
print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_pred)))
```

```
Accuracy Score : 0.4602739726027397
Precision Score : 0.45121951219512196
Recall Score : 0.19576719576719576
F1 Score : 0.27306273062730624
```

As we could expect based on the confusion matrix, the recall score of this model is very low. We can recall that the SVM model (at least by observing the plot) seemed to be better in predicting down trends. Let us repeat this model validation approach also for SVM.

1.6.2 Support Vector Machines

```
[133]: cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']
       EBS_features['I'][EBS_features['I']<=0] = -1



       X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
        ↪train_test_split(EBS_features[cols],
                                                                                 ␣
        ↪    EBS_features['I'] ,
                                                                                 ␣
        ↪    test_size=0.25, shuffle=True)

       SVM_EBS =SVC(C=1,probability=True,kernel="linear")
       SVM_EBS.fit(X_Train_EBS,Y_Train_EBS)

       plot_logit_ROC(1,Y_Test_EBS,X_Test_EBS,SVM_EBS,EBS_features['I'],EBS_features[cols])
```



```
[133]: <matplotlib.axes._subplots.AxesSubplot at 0xe76db48>
```

THe SVM model still cannot get us a robust result. The AUC curve is near to the random model.

[134]:
```
#print confusion matrix
Y_EBS_P = SVM_EBS.predict(X_Test_EBS)
confusion_matrix_EBS = confusion_matrix(Y_Test_EBS,Y_EBS_P)
print("Confusion Matrix - SVM Model")
print(confusion_matrix_EBS)
```

```
Confusion Matrix - SVM Model
[[94 92]
 [93 86]]
```

Even though the result seem to be 50/50, this is certainly more balanced model than the logistic regression (i.e. we would expect to perform equally good in down and up trends)

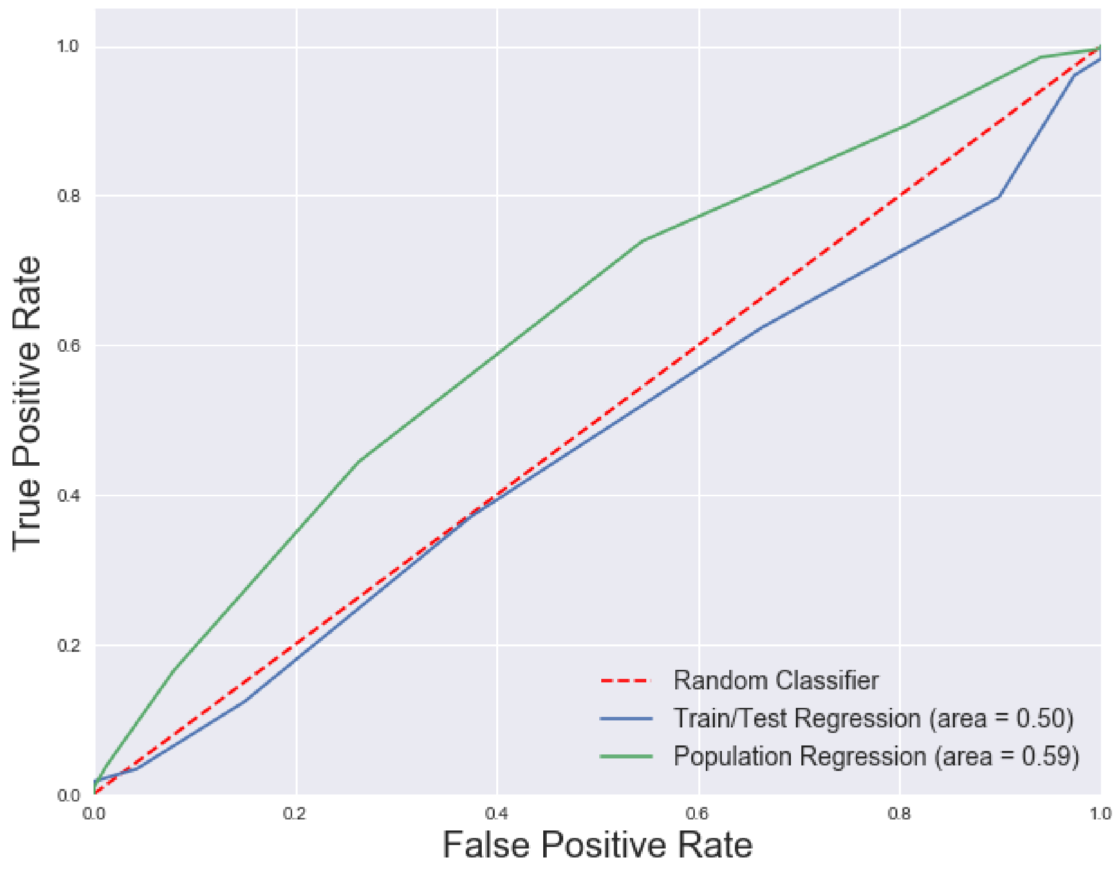
[135]:
```
from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_P)))
print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_P)))
print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_P)))
print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_P)))
```

```
Accuracy Score : 0.4931506849315068
Precision Score : 0.48314606741573035
Recall Score : 0.48044692737430167
F1 Score : 0.48179271708683474
```

Almost equal precision and recall scores, which confirms our guess. LAst but not least, we evaluate the K-nearest Neighbours classifier.

1.6.3 K-nearest Neighbours

[29]:
```
from sklearn import model_selection
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
cols = ['ret_1','ret_2', 'ret_3', 'ret_4', 'ret_5']

X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
 ↪train_test_split(EBS_features[cols],
                                                                          ␣
 ↪    EBS_features['I'] ,
                                                                          ␣
 ↪    test_size=0.25, shuffle=True)

knn8 = KNeighborsClassifier(n_neighbors=8)

knn8.fit(X_Train_EBS,Y_Train_EBS)

plot_logit_ROC(1,Y_Test_EBS,X_Test_EBS,knn8,EBS_features['I'],EBS_features[cols])
```



[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xc275f08>
```

Although the KNN algorithm performed better on the population set, it has similar result to the SVM and Logistic Regression classifiers in out-of-sample data. The model is not much different than random classifier.

[30]:
```
#print confusion matrix
Y_EBS_P = knn8.predict(X_Test_EBS)
confusion_matrix_EBS = confusion_matrix(Y_Test_EBS,Y_EBS_P)
print("Confusion Matrix - KNN Model")
print(confusion_matrix_EBS)
```

```
Confusion Matrix - KNN Model
[[117  70]
 [112  66]]
```

The KNN classifier results can be placed between the Logistic Regression and the SVM model.

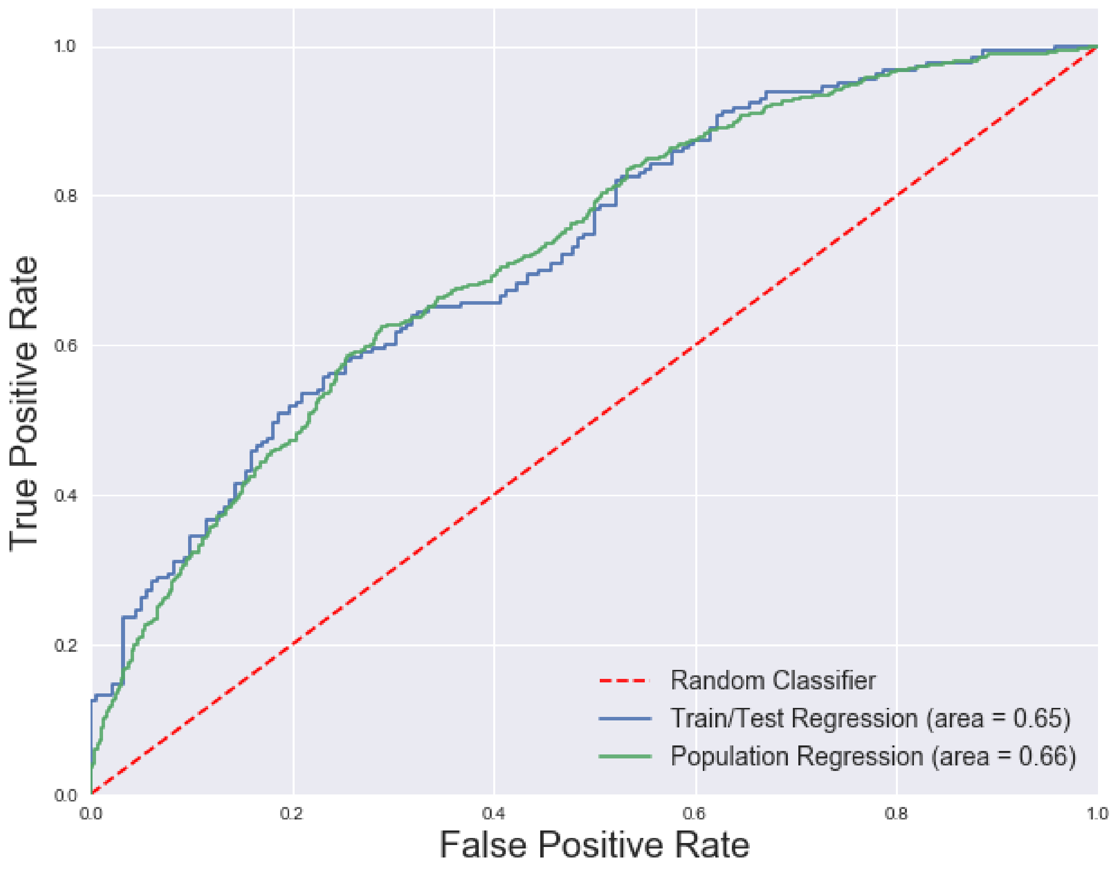
[19]:
```
from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_P)))
print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_P)))
print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_P)))
print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_P)))
```

```
Accuracy Score : 0.5013698630136987
Precision Score : 0.4846153846153846
Recall Score : 0.3539325842696629
F1 Score : 0.4090909090909091
```

Overall, we can say that none of the three algorithms cannot give us a significant potential to achieve excess returns if only lagged returns are used to train the models. Our winning probability is usually near to the random classifier. In the next section we will test a model with our selected features.

1.6.4 Logistic Regression - Momentum, EWMA Crossover & Previous day return signals

```
[140]: cols=['MOM5', 'EWMA5-7','ret_1']

X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
 ↪train_test_split(EBS_features[cols],
                                                                          ␣
 ↪    EBS_features['I'] ,
                                                                          ␣
 ↪    test_size=0.25, shuffle=True)

lm_EBS = linear_model.LogisticRegression(C = 1e15,solver ='liblinear',penalty =␣
 ↪'l2')
lm_EBS.fit(X_Train_EBS,Y_Train_EBS)


plot_logit_ROC(1,Y_Test_EBS,X_Test_EBS,lm_EBS,EBS_features['I'],EBS_features[cols])
```



[140]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xcbba7c8>
```

Indeed, AUC analysis does support our model. The Regression curves are far from the random classifier, which is exactly what we want.

[141]:
```
#print confusion matrix
Y_EBS_pred = lm_EBS.predict(X_Test_EBS)
confusion_matrix_EBS = confusion_matrix(Y_Test_EBS,Y_EBS_pred)
print("Confusion Matrix - Logit Reg. - Improved Set")
print(confusion_matrix_EBS)
#this is a very good model
```

```
Confusion Matrix - Logit Reg. - Improved Set
[[128  54]
 [ 73 110]]
```

The model is still stronger in predicting positive returns. In the next section we will see if we can improve negative returns prediction.

[142]:
```
from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_pred)))
print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_pred)))
print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_pred)))
print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_pred)))
```

```
Accuracy Score : 0.6520547945205479
Precision Score : 0.6707317073170732
Recall Score : 0.6010928961748634
F1 Score : 0.6340057636887609
```

The metrics confirm the robustness of the model.

1.6.5 B.2 Grid Search

Instead of following trial and error approach in finding the optimal parameters one can use the GridSearchCV function. To specify the grid values we will choose between l1 and l2 penalty for different regularization parameters C. Since one of the tasks is to recommend approach how to reduce misclassified negative returns we will set our Grid Search score to "recall".

In the previous section we have seen that our model tend to regard more samples to be positive, and this could be confirmed from the confusion matrices where the left side numbers are always higher than those in the right. We will see if we can improve on that.

```
[145]: cols=['ret_1', 'ret_2', 'ret_3', 'ret_4', 'ret_5',
             'Sign1', 'MOM5', 'MOM7',
              'MOM13', 'MOM21', 'EWMA5', 'EWMA7', 'EWMA13', 'EWMA21', 'EWMA5-7',
              'EWMA7-13', 'EWMA13-21', 'MA5', 'MA7', 'MA13', 'MA21', 'MA5-7',
              'MA7-13', 'MA13-21', 'stdev21']

       from sklearn import svm, datasets
       from sklearn.model_selection import GridSearchCV

       X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
        ↪train_test_split(EBS_features[cols],                                                ␣
        ↪    EBS_features['I'] ,                                                             ␣
        ↪    test_size=0.25, shuffle=True)



       ##############
       # Logit Model

       lm_EBS = linear_model.LogisticRegression()
       grid_values = {'penalty': ['l1', 'l2'],'C':[0.001,.009,0.01,.09,1,5,10,25]}
       grid_lm_EBS_acc = GridSearchCV(lm_EBS, param_grid = grid_values,scoring =␣
        ↪'recall')
       grid_lm_EBS_acc.fit(X_Train_EBS,Y_Train_EBS)
```

```
Y_EBS_pred = grid_lm_EBS_acc.predict(X_Test_EBS)


from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_pred)))
print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_pred)))
print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_pred)))
print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_pred)))

#see what parameters to use
print(grid_lm_EBS_acc.best_params_)
```

```
Accuracy Score : 0.6931506849315069
Precision Score : 0.7065868263473054
Recall Score : 0.6519337016574586
F1 Score : 0.67816091954023
{'C': 25, 'penalty': 'l2'}
```

The output of our gridsearch tells us that by using regularization param. C=25 and by implement l2 penalized regression we will achieve maximum recall score. Now let us apply the gridsearch function to the our selected features:

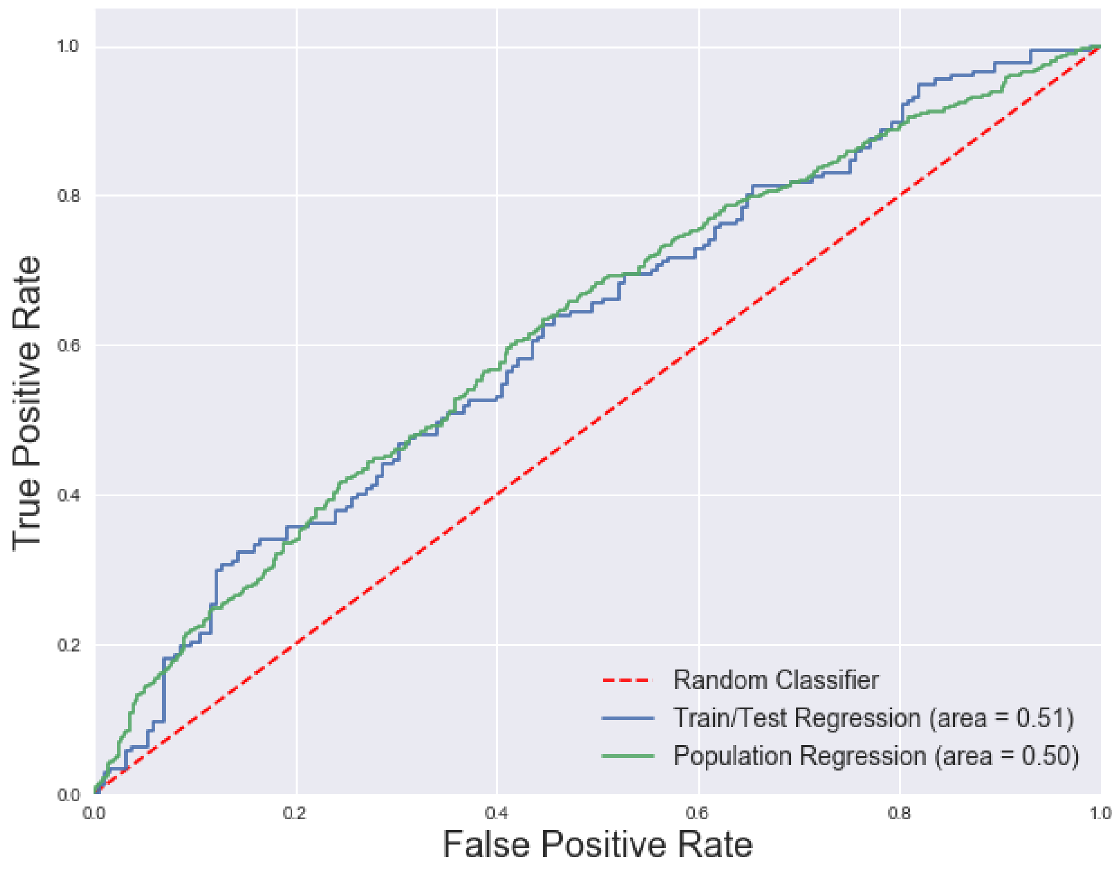
[157]:
```
from sklearn import svm, datasets
from sklearn.model_selection import GridSearchCV

cols=['MOM5', 'EWMA5-7', 'ret_1']

X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
 ↪train_test_split(EBS_features[cols],                                    ␣
 ↪    EBS_features['I'] ,                                                  ␣
 ↪    test_size=0.25, shuffle=True)



##############
# Logit Model

lm_EBS = linear_model.LogisticRegression()
grid_values = {'penalty': ['l1', 'l2'],'C':[0.001,.009,0.01,.09,1,5,10,25]}
grid_lm_EBS_acc = GridSearchCV(lm_EBS, param_grid = grid_values,scoring =␣
 ↪'recall')
grid_lm_EBS_acc.fit(X_Train_EBS,Y_Train_EBS)
Y_EBS_pred = grid_lm_EBS_acc.predict(X_Test_EBS)


from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
```

```
print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_pred)))
print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_pred)))
print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_pred)))
print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_pred)))

#see what parameters to use
print(grid_lm_EBS_acc.best_params_)
```

```
Accuracy Score : 0.4712328767123288
Precision Score : 0.4712328767123288
Recall Score : 1.0
F1 Score : 0.6405959031657356
{'C': 0.001, 'penalty': 'l2'}
```

The gridsearch tells us that we should use C=0.001 and again l2 penalty. It is remarkable that the gridsearch maximised the recall to 1. We will check results of the logistic regression with these parameters:

```
[161]: X_Train_EBS, X_Test_EBS, Y_Train_EBS, Y_Test_EBS = model_selection.
        ↪train_test_split(EBS_features[cols],
                                                                                ␣
        ↪    EBS_features['I'] ,
                                                                                ␣
        ↪    test_size=0.25, shuffle=True)

       lm_EBS = linear_model.LogisticRegression(C = 0.001,penalty = 'l2')
       lm_EBS.fit(X_Train_EBS,Y_Train_EBS)


       plot_logit_ROC(1,Y_Test_EBS,X_Test_EBS,lm_EBS,EBS_features['I'],EBS_features[cols])


       #print confusion matrix
       Y_EBS_pred = lm_EBS.predict(X_Test_EBS)
       confusion_matrix_EBS = confusion_matrix(Y_Test_EBS,Y_EBS_pred)
       print("Confusion Matrix")
       print(confusion_matrix_EBS)
       #this is clearly bad model, a random model could have been better

       from sklearn.metrics import accuracy_score,recall_score,precision_score,f1_score
       print('Accuracy Score : ' + str(accuracy_score(Y_Test_EBS,Y_EBS_pred)))
       print('Precision Score : ' + str(precision_score(Y_Test_EBS,Y_EBS_pred)))
       print('Recall Score : ' + str(recall_score(Y_Test_EBS,Y_EBS_pred)))
       print('F1 Score : ' + str(f1_score(Y_Test_EBS,Y_EBS_pred)))
```



```
Confusion Matrix
[[  2 186]
 [  0 177]]
Accuracy Score : 0.4904109589041096
Precision Score : 0.48760330578512395
Recall Score : 1.0
F1 Score : 0.6555555555555556
```

Now looking at the Confusion Matrix, the right side numbers are much higher than those in the left. So we have indeed found a model that regards more samples to be negative, but this phenomenon is brought to extreme. The model has failed almost completely in detecting true positives. Indeed, if we want to improve recall, the model will have to keep generating results which are not accurate, hence lowering the precision.

In conclusion, we can say that a good model should show balance in precision and recall scores. Going in extremes can make the model good in detecting perfectly one class but then failing completely in detecting another. If we want to keep the robustness of the model we should recommend maximising the F1 Score.

1.7 B.3 P&L Backtesting. transition probabilities. Kelly

Though we have seen already consistent price prediction paths with combined signals from Momentum, EWMA Crossover and previous day return, transition probability can shed more light upon our model. Transition probability can tell us how “sure” the model is in its prediction for an up or down movement.

[162]:
```
#this is our best set of features in logit regression
cols=['MOM5', 'EWMA5-7','EWMA7-13']
lm_EBS = linear_model.LogisticRegression(C =␣
↪1e6,solver='liblinear',multi_class='ovr',penalty='l2')
lm_RBI = linear_model.LogisticRegression(C =␣
↪1e6,solver='liblinear',multi_class='ovr',penalty='l2')

training_split = int(0.75*EBS.shape[0])

EBS_training = EBS_features.iloc[0:training_split,:]
RBI_training = RBI_features.iloc[0:training_split,:]

EBS_validate = EBS_features.iloc[training_split:,:]
RBI_validate = RBI_features.iloc[training_split:,:]

#fit again
lm_EBS.fit(EBS_training[cols],EBS_training['Sign'])
lm_RBI.fit(RBI_training[cols],RBI_training['Sign'])

#predict using the validation set of data
EBS_validate['Logit_Predict'] = lm_EBS.predict(EBS_validate[cols])
RBI_validate['Logit_Predict'] = lm_RBI.predict(RBI_validate[cols])

EBS_validate['Logit_Returns']␣
↪=EBS_validate['ret_0']*EBS_validate['Logit_Predict']
RBI_validate['Logit_Returns']␣
↪=RBI_validate['ret_0']*RBI_validate['Logit_Predict']



Pred_Probs_EBS=lm_EBS.predict_proba(EBS_validate[cols])
Pred_Probs_RBI=lm_RBI.predict_proba(RBI_validate[cols])
#1st column is probability of down move
#2nd column is probability of up move

EBS_validate['real_NAV'] = abs(EBS_validate['ret_0']).cumsum().apply(np.exp)
EBS_validate['predicted_NAV'] = EBS_validate['Logit_Returns'].cumsum().apply(np.
↪exp)
np.corrcoef(EBS_validate['real_NAV'],EBS_validate['predicted_NAV'])
```

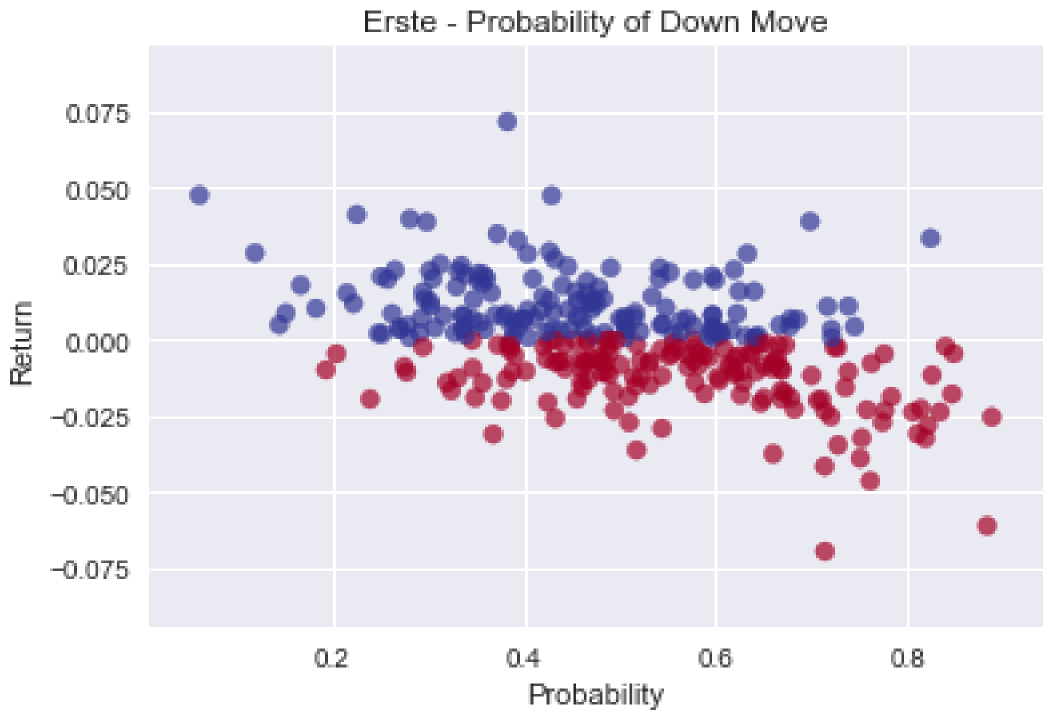
[162]:
```
array([[1.        , 0.95650053],
       [0.95650053, 1.        ]])
```

We have already plotted the hypothetical P&L of this model, here we only compute the correlation of the model returns with the absolute returns of Erste Group.

The transition probabilites are stored in vectors Pred_Probs. There are two columns, the 1st (the zero) column is the probability output of -1(negative return), whereas the 2nd column contains the probability of output of 1 (positive return). Most of transition probabilities are consistently higher than 0.5, which indicates very good results. As a next step, we will plot the down move (i.e. negative return) probabilities.

```
[163]: #defining an array of colors
       colors = ['r', 'b']
         #assigns a color to each data point
       plt.scatter(Pred_Probs_EBS[:,0], EBS_validate['ret_0'],c=EBS_validate['Sign'],␣
        ↪alpha=0.70, cmap='RdYlBu')
       plt.title("Erste - Probability of Down Move")
       plt.xlabel('Probability')
       plt.ylabel('Return')
```

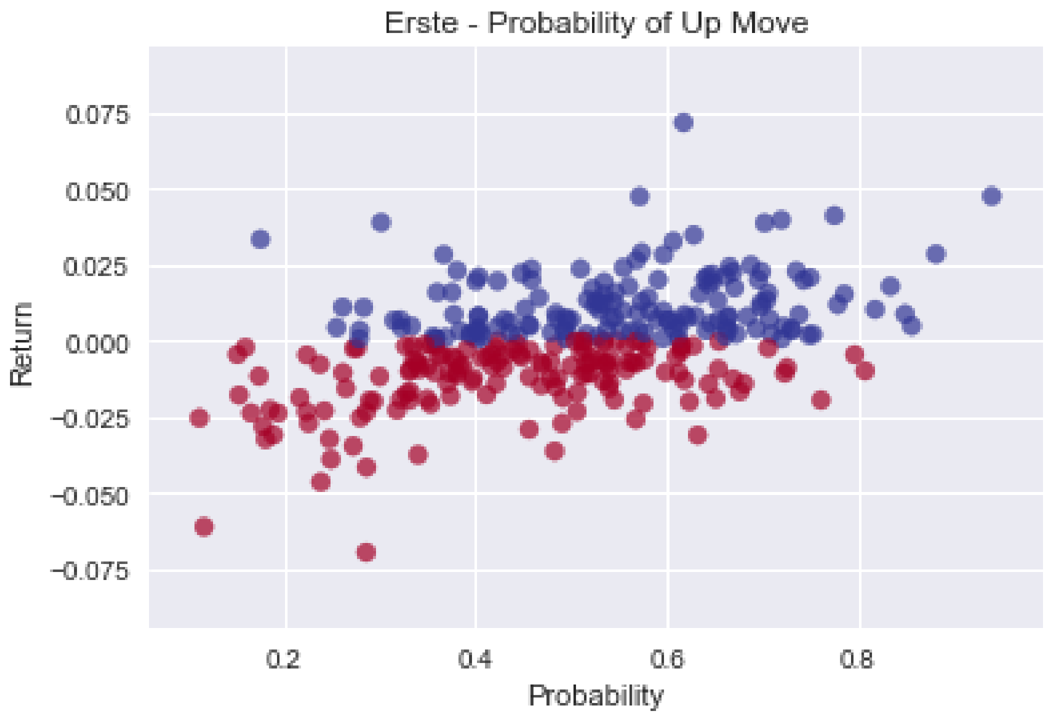
```
[163]: Text(0, 0.5, 'Return')
```



We can observe negative correlation - and for down movements this is exactly what we want, i.e. the higher the probability of prediction, the more negative the actual return comes out. Now let us plot also the positive return probability (i.e. up move).

```
[164]: plt.scatter(Pred_Probs_EBS[:,1], EBS_validate['ret_0'],c=EBS_validate['Sign'],␣
        ↪alpha=0.70, cmap='RdYlBu')
       plt.title("Erste - Probability of Up Move")
       plt.xlabel('Probability')
       plt.ylabel('Return')
```

```
[164]: Text(0, 0.5, 'Return')
```



We see strong positive correlation - this is exactly what we want, i.e. high probability for large positive returns

The transition probabilities provide another proof of the strength of the model.

Kelly Criterion Finally, let's introduce the Kelly criterion in our model. We can set our threshold probability at 0.55, i.e. position will be opened only if the model generates probability higher than 0.55

```
[165]: strategy_PNL=0
       PNL_history = []
       for i in range(len(Pred_Probs_EBS)):
           daily_PNL=0
           if (Pred_Probs_EBS[i][0]>0.55):
               daily_PNL=(2*Pred_Probs_EBS[i][0]-1)*(-EBS_validate['ret_0'][i])
           else:
```

```
        if (Pred_Probs_EBS[i][1]>0.55):
            daily_PNL=(2*Pred_Probs_EBS[i][1]-1)*(EBS_validate['ret_0'][i])
    strategy_PNL=strategy_PNL+daily_PNL
    PNL_history.append(daily_PNL)


strategy_PNL
```

[165]:
```
0.7864674169201707
```

This is very good result, however let us check the hypothetical result if we didn't use Kelly the return is

[168]:
```
EBS_validate['Logit_Returns'].cumsum()[-1]
```

[168]:
```
0.8824530021865854
```

So, it turns out that the Kelly criteterion curbed the profit. Though, the benefit of Kelly Criterion can be seen in a lower drawdown:

[170]:
```
min(PNL_history)
```

[170]:
```
-0.021569589379991067
```

[171]:
```
min(EBS_validate['Logit_Returns'])
```

[171]:
```
-0.031155108901866346
```

1.7.1 Conclusion and Summary of findings on the classifiers

Our observations show that there are no consistent relationship between future price movement with lagged returns among Erste Group and Raiffeisen Bank. However, a set of features including Momentum, EWMA Crossover strategy and previous day return can achieve much better performance. Thus, the choice of features should not be neglected and even the most sophisticated learning algorithm will not be able to improve our model if there is no consistent relationship between returns and feautures. We have made the following observation when implementing the classifiers:

Logistic Regression Increasing C can lead to better accuracy of L1 penalised regression over L2 penalised version and also sparser solutions. Smaller values of C (eg. less than 1) lead in general to better performance of the L2 version over L1. K-fold crossvalidation can improve the results, however one should be wary that using more data sets to train the model can lead to small number of observations in the sets.

Support Vector Machines The results on the margins were mixed. Hard margins gave improved results for Raiffeisen Bank which had some clear outliers in its data. Soft margins worked better for Erste Group, where data seemed to be more homogeneous. SVM model seemed to be better predictor of down movements than the logistic regression.

K-nearest Neighbours The algorithm is sensitive to the choice of k: increasing k could firstly improve accuracy but high value of k could reduce model's accuracy. The Mahalanobis distance metric appeared to be superior to the Euclidean and Manhattan distance metrics.

Performance Evaluation and Backtesting AUC Validation, Confusion Matrices and Transition probabilites can shed further light on model. In additon to accuracy, one should check also recall and precision scores. Gridsearch can improve results and find a model sensitive to particular class (eg. negative samples), however, one should be aware of the tug of war between precision and recall. To avoid extremes, one could aim to maximise F1 score.

1.8 Task B.4

The pdf of normal distribution is:

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} = (2\pi\sigma^2)^{-\frac{1}{2}} exp\{\frac{-(x-\mu)^2}{2\sigma^2}\}$$

To present it in exponential form we firstly take logs on both sides:

$$log(f(x|\mu,\sigma) = -\frac{1}{2} log(2\pi\sigma^2) - \frac{(x-\mu)^2}{2\sigma^2}$$

Then, take exponential:

$$f(x|\mu,\sigma) = exp\{-\frac{1}{2} log(2\pi\sigma^2) - \frac{(x-\mu)^2}{2\sigma^2}\}$$

Expand the square:

$$f(x|\mu,\sigma) = exp\{\frac{-x^2 + 2x\mu - \mu^2}{2\sigma^2} - \frac{1}{2} log(2\pi\sigma^2\}$$

To present $f(x|\mu)$ in the form of $exp\{a(x)b(\mu) + c(\mu) + d(x)\}$ we write as:

$$f(x|\mu,\sigma) = exp\{(\frac{x\mu}{\sigma^2}) + (-\frac{\mu^2}{2\sigma^2}) + (-\frac{x^2}{2\sigma^2}) - \frac{1}{2} log(2\pi\sigma^2)\}$$

We see that $a(x) = x$,$b(\mu) = \frac{\mu}{\sigma^2}$, $c(\mu) = -\frac{1}{2}(\frac{\mu}{\sigma})^2$ and $d(x) = -\frac{1}{2}(\frac{x}{\sigma})^2$

If we assume that the variance is known then we have:

$$f(x|\mu) = exp\{(x\mu) + (-\frac{\mu^2}{2}) + (-\frac{x^2}{2}) - \frac{1}{2} log(2\pi)\}$$

i.e. $a(x) = x$, $b(\mu) = \mu$, $c(\mu) = -\frac{1}{2}\mu^2$ and $d(x) = -\frac{1}{2}x^2$

Another way of writing the normal distribution in the exponential family is:

$$f(y|\theta, \phi) = exp\{\frac{y\theta - b(\theta)}{a(\phi)} + c(y; \phi)\}$$

where

$\theta = \mu; b(\theta) = \frac{1}{2}\theta^2; a(\phi) = \sigma^2$

One can check $E[x] = \frac{\partial}{\partial\theta}b(\theta) = \theta = \mu$ and $Var[x] = a(\phi)\frac{\partial^2}{\partial\theta^2}b(\theta) = \sigma^2$

1.9 References

CQF Lectures: Module 4

P. Wilmott - Machine Learning: An Applied Mathematics Introduction

Python Data Science Handbook: Essential Tools for Working with Data