06-Exam3

Table of contents

1 Modeling Task

Short-term asset return is a challenging quantity to predict.

The objective of this task is to produce a model to predict positive moves (up trend) using a **Support Vector Machine** (SVM) machine learning model . The proposed solution should be comprehensive with detailed feature engineering and model architecture.

Before we start, let's remind ourselves of one of the main assumption in our quantitative finance journey: financial assets follow a Brownian motion; hence over a longer-ish period of time, we would expect our model to be predict half the time up and half the time down.

1.1 SVM as a classifier.

Support Vector Machine is a machine learning algorithm that aims to find the hyperplane that maximizes the margins that separate *'hardly'* or *'softly'* the data into different classes. In this exercise, we are dealing with a binary classification (up or down).
The hyperplane can be modelled with

$$\theta^T \cdot X + \theta_0 = 0$$

where

- θ is the parameters vector of the plane that maximize the margins that separate the output variables. θ_T is the transpose of that vector.
- θ_0 is a scalar vector. How far is the hyperplane from the origin.

In the case of *hard-margin*, we want to maximize the margin width

$$\max_{\theta,\theta_0} \frac{2}{|\theta|}$$

such that there are no missclassifcation at all. This might require to use non-linear hyperplane (*radial*, *polynomial* or *sigmoid*) or to go a higher dimension (*the Kernel Trick*) to ensure that data are then exactly separable. Which is the same as finding

$$\min_{\theta,\theta_0} \sum_{i=1}^{N} \frac{1}{2}|\theta|^2$$

In the case of *soft-margin* where we have to weight in (with regularization) the cost of missclassification against wider margins; the loss function becomes:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} max\left(1 - y^{(i)} \cdot (\theta^T X^{(i)} + \theta_0), 0\right) + \lambda|\theta|^2$$

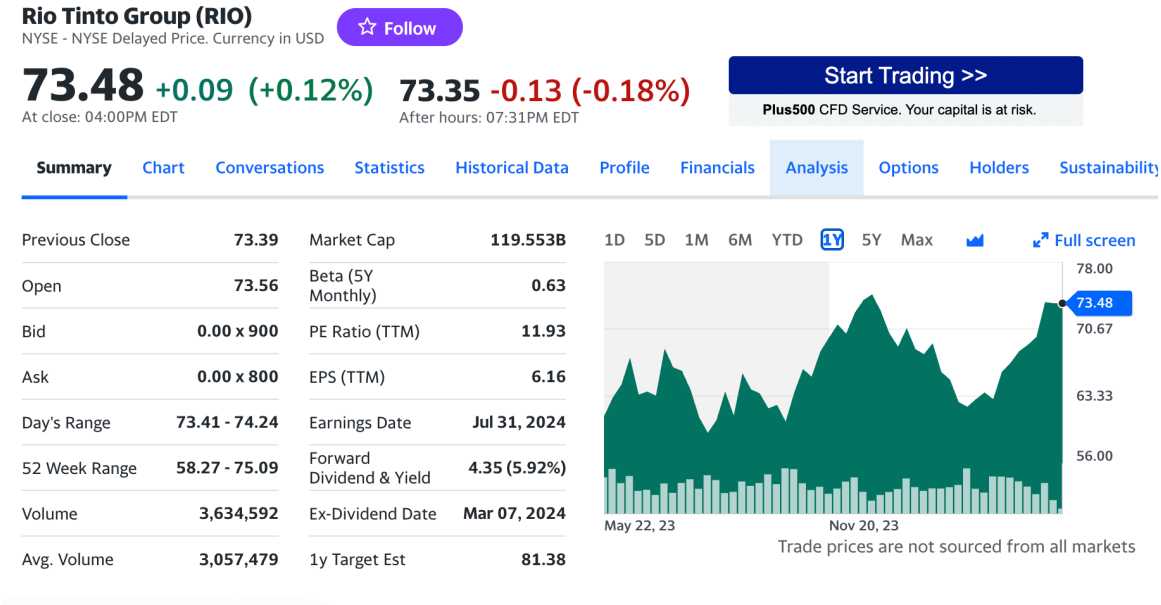
- N is the number of observation in our dataset

- $y^{(i)}$ is the binary target variable, the output
- $X^{(i)}$ is the feature vector, the inputs
- $max\left(1 - y^{(i)} \cdot (\theta^T X^{(i)} + \theta_0), 0\right)$ is the loss function
- $\lambda|\theta|^2$ is the regularization term

1.2 Choice of ticker

To complete the task, we will use a mining stock **Rio Tinto Group** with ticker 'RIO'.

As of today (22 May 2025), this is a screenshot of the stock on Yahoo Finance.



1.3 Loading Python packages, data & Initial EDA

We'll start by loading the various Python packages required for our ML task.

1.3.1 Loading Python packages

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import yfinance as yf
import mplfinance as mpl

from datetime import date
from dateutil.relativedelta import relativedelta
```

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import TimeSeriesSplit, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.svm import SVC

from sklearn.metrics import roc_curve, ConfusionMatrixDisplay,
 ↪  accuracy_score, auc
from sklearn.metrics import classification_report
```

Let's first retrieve the price of Rio Tinto and store it in a *Pandas* dataframe.

```
# downloading the data and storing it in a dataframe
rio = yf.download('RIO')
df = pd.DataFrame(rio)
```

1.3.2 Exploring the time-series

A quick look into the first 5 rows to check the structure. we'll also check to see if there are missing values.

```
# get to know df variables
print('The shape of dataframe is:', df.shape)
print(df.head())

# checking for missing values.
df.isnull().sum()
```

```
The shape of dataframe is: (8544, 6)
                Open      High       Low     Close  Adj Close  Volume
Date
1990-06-28  10.09375  10.09375   9.96875   9.96875   1.916216  176400
1990-06-29  10.03125  10.06250  10.00000  10.06250   1.934235   69200
1990-07-02  10.00000  10.03125  10.00000  10.03125   1.928229   62000
1990-07-03  10.03125  10.06250  10.03125  10.06250   1.934235   29600
1990-07-05   9.71875   9.71875   9.65625   9.68750   1.862152   31200


Open         0
High         0
Low          0
Close        0
Adj Close    0
```

```
Volume       0
dtype: int64
```

No missing values. That's an encouraging start.

A quick descriptive stat check on last 5 years.

```
df.loc['2019-01-01':].describe().round(2)
```

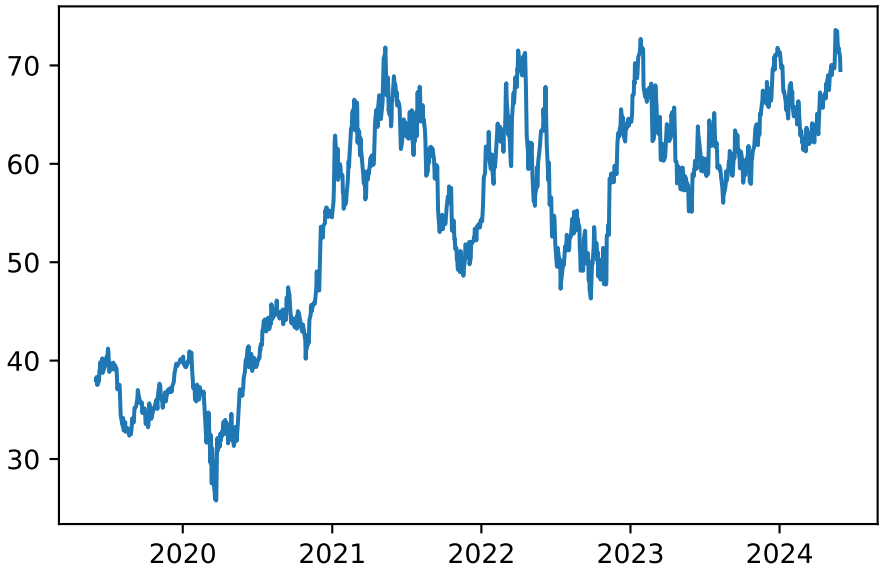
	Open	High	Low	Close	Adj Close	Volume
count	1361.00	1361.00	1361.00	1361.00	1361.00	1361.00
mean	65.08	65.64	64.51	65.11	52.63	2922285.82
std	10.47	10.54	10.38	10.50	12.28	1305290.07
min	36.10	37.21	35.35	36.42	25.78	378200.00
25%	58.22	58.73	57.69	58.29	39.81	2028300.00
50%	63.49	64.02	63.07	63.64	55.83	2710200.00
75%	71.68	72.22	71.03	71.80	63.00	3550000.00
max	95.23	95.97	93.65	94.65	73.61	10406600.00

1.3.3 Vizualising the time-series

Initial line plot of RIO stock over the last 5 years

```
# Checking only over the last 5 years of data
start_date = date.today() + relativedelta(days = int(np.round(-5 * 365)))
end_date = date.today()

# first vizualization
plt.plot(df['Adj Close'].loc[start_date:end_date])
plt.show()
```



Using candlesticks on last 18 months of data

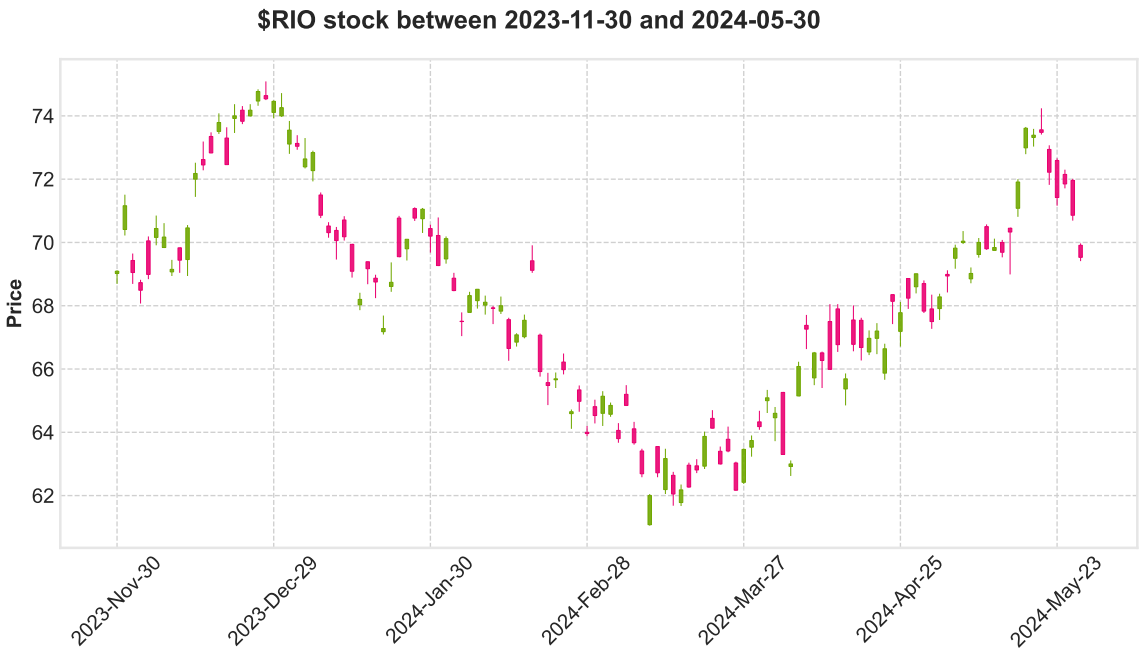
```
start_date = date.today() + relativedelta(days = int(np.round(-0.5 * 365)))

df_last18months = df.loc[start_date:end_date]

fig, axlist = mpl.plot(
  df_last18months,
  type = 'candle',
  style='binance',
  figratio = (15, 7),
  returnfig = True
)

fig = fig.suptitle(
  f"$RIO stock between {start_date} and {end_date}",
  y=.95,
  fontsize=15,
  weight = 'semibold',
  style = 'normal'
)

mpl.show()
```



1.4 Features Engineering

Before starting the feature engineering process, we need to clarify what is understood by **'predicting positive moves (up trend)'.** Since *"making sense of the instructions'* is part of the assessment, we are making the following assumptions.

- the "moves" are based on the closing prices
- we are only trading during official market hours. Making that assumption has some implication for trading as we would only know the closing price at the close of the trading session and hence technically we won't be able to trade that same day based on that information. We would assume we make the trade before the closing bell

- question to answer: during the trading session, would tomorrow's closing price be higher than today's closing price? Both are unknown at the moment of the trade.

With these assumptions in mind, we'll start the process of feature engineering, with the suggested features presented in the exam instructions.

- We will create our binary classifier (*positive move*) based on the future asset price returns which we define as the log returns of the adjusted closing price.

$$Ret_t = log\left(\frac{P_{t+1}}{P_t}\right) = log(P_{t+1}) - log(P_t)$$

- As suggested in the list of possible *'basic'* features, we can use intraday moves as signal. As prices change overtime considerably (from a minimum of adjusted close of 25.7 to 73.6), we intuitively feel, the indicator might be more relevant if we scale it as a ratio

of the difference of the 2 prices over the closing price. And to avoid look ahead biais, we define the 2 features as

$$(O-C)_t = \frac{(O-C)_{t-1}}{C_{t-1}} \approx log\left(\frac{O_{t-1}}{C_{t-1}}\right)$$

$$(O-O)_t = \frac{(O_t-O_{t-1}}{O_t} \approx log\left(\frac{O_t}{O_{t-1}}\right)$$

$$(O-prevC)_t = \frac{(O_t-C_{t-1}}{O_t} \approx log\left(\frac{O_t}{C_{t-1}}\right)$$

and

$$(H-L)_t = \frac{(H-L)_{t-1}}{L_{t-1}} \approx log\left(\frac{H_{t-1}}{L_{t-1}}\right)$$

- Next, we used the lagged returns: $Ret_{t-1}, Ret_{t-2}, etc.$. Without much initial thought, we'll create 17 of these lagged returns.
- SMA defined as

$$\frac{1}{N}\sum_{i=1}^{N} P_{t-1}$$

- EMA defined as

$$EMA_t = EMA_{t-1} + \alpha \cdot [P_t - EMA_{t-1}]$$

To do so we'll use the pandas already built-in functionalities. We have checked in the documentation that they match the expected behavior for the exponential moving average Pandas Doc on EMW.

```
# getting returns. We shift it as it constitutes the base of what we have to
 ↪ predict.
df['ret_1d_target'] = np.log(df['Adj Close'].shift(-1) / df['Adj Close'])
df['ret_1d_shifted'] = (np.log(df['Adj Close']).diff(1)).shift(1)
df.head()

# creating variables based on intra day.
df['o-c'] = np.log(df.Open/df.Close).shift(1)   # Same day Open-Close
df['h-l'] = np.log(df.High/df.Low).shift(1)     # Same day High-Low
df['o-o'] = np.log(df.Open/df.Open.shift(1))    # Open-Previous day Open
df['o-prevC'] = np.log(df.Open/df.Close.shift(1)) # Open - Previous day Close

new_cols = {}     # avoid df fragmentation warnings

# creating our lag variables
for lags in range(1, 17, 1):
  new_cols['ret_1d_lag' + str(lags) + 'd'] = df.ret_1d_shifted.shift(lags)
  new_cols['o-c-lag' + str(lags) + 'd'] = df['o-c'].shift(lags)
  new_cols['o-o-lag' + str(lags) + 'd'] = df['o-o'].shift(lags)
  new_cols['h-l-lag' + str(lags) + 'd'] = df['h-l'].shift(lags)
  new_cols['o-prevC-lag' + str(lags) + 'd'] = df['o-prevC'].shift(lags)
```

```
# Creating momentum variables
for days in range(2, 61, 3):
  new_cols['ret_'+ str(days)] = df.ret_1d_shifted.rolling(days).sum()
  new_cols['sd_' + str(days)] = df.ret_1d_shifted.rolling(days).std()

# creating SMA percentage
for lags in range(2, 61, 4):
  new_cols['sma_' + str(lags) + 'd'] = df['Adj
 ↪  Close'].rolling(window=lags).mean()
  new_cols['sma_' + str(lags) + 'd'] = np.log(df['Adj Close'] /
 ↪  new_cols['sma_' + str(lags) + 'd'])
  new_cols['sma_' + str(lags) + 'd'] = new_cols['sma_' + str(lags) +
 ↪  'd'].shift(1)

# creating EMA percentage
for lags in range(3, 10, 2):
  new_cols['ema_' + str(lags) + 'd'] = df['Adj Close'].ewm(span = lags,
 ↪  adjust = False).mean()
  new_cols['ema_' + str(lags) + 'd'] = np.log(df['Adj Close'] /
 ↪  new_cols['ema_' + str(lags) + 'd'])
  new_cols['ema_' + str(lags) + 'd'] = new_cols['ema_' + str(lags) +
 ↪  'd'].shift(1)

df_new_cols = pd.DataFrame(new_cols)
df = pd.concat([df, df_new_cols], axis = 1)

print(df.shape)
```

```
(8544, 151)
```

1.5 Target Definition and model setup

Our initial thought to define the target variable is to use the median return over the last 5 years. We expect this to be a slightly positive value. The main advantage of using the median is that it ensures we have balance classes to predict.

In this sense, we define the label as

$$y_i = \begin{cases} 1 & \text{if } x_i \geq \text{median(return)} \\ 0 & \text{otherwise} \end{cases}$$

1.5.1 Final removing of NaN values and target definition

```
# slicing the dataframe to get all data after 2019 (close to last 5
 ↪ years)
df_new = df.loc['2019-01-01':].copy()
df_new.shape

# Step 2: Calculate the median of the targetcolumn in the filtered
 ↪ data
median_return = df_new['ret_1d_shifted'].dropna().median()
print(f"Median Return: {np.round(median_return, 6)}")

# Step 3: Create the 'target' column based on the median return
df_new['target'] = np.where(df_new['ret_1d_target'] > median_return,
 ↪ 1, 0)
df_new.shape
df_new['target'].value_counts(normalize = True).round(4) * 100

df_new.dropna(inplace = True)

# Step 4: Remove all unwanted columns and target variables
df3 = df_new.drop(['Open', 'High', 'Low', 'Close', 'Adj Close',
 ↪ 'Volume', 'target',
                'ret_1d_target'], axis = 1).copy()
# checking again size of data frame
print(df3.shape)
print(df_new.shape)

# setting up our inputs and output dataframe
X = df3.values
y = df_new['target'].values
```

```
Median Return: 0.001443
(1360, 144)
(1360, 152)
```

1.5.2 Splitting the data

Creating the training and testing set. Note that the training set will be use for cross-validation as well. Testing set will only be used twice. In our initial model and once we have chosen our final model.

```
# put shuffle to False as we are dealing with a timeseries where order
 ↪  of data matter.
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size =
 ↪  0.2, shuffle = False)

print('Shape of x_train:', x_train.shape)
print('Shape of y_train:', y_train.shape)
print('Shape of x_test:', x_test.shape)
print('Shape of y_test:', y_test.shape)
print(f"Size of training set is {len(x_train)} and size of testing set
 ↪  is {len(x_test)}")
```

```
Shape of x_train: (1088, 144)
Shape of y_train: (1088,)
Shape of x_test: (272, 144)
Shape of y_test: (272,)
Size of training set is 1088 and size of testing set is 272
```

Using 80-20 for training and testing leaves a bit over a year of data (272 trading days) for testing.

1.5.3 Checking for correlations

Although some ML algorithms are more robust to highly correlated variables (and SVM in theory is robust to correlated variables); it is generally good practice avoiding to bring in highly correlated variables. There are several reasons for this:

- redundancy (we prefer parsimonious model, simpler is better principle),
- model interpretability (when several features are strongly correlated it becomes difficult to know which one affect the output variable),
- model generalization. When features are strongly correlated, model can perform well on training set but generalize poorly in testing set.
- 'curse of dimensionality'. More variables than necessary degrade the model performances.
- stability of feature importance. With highly correlated feature, a small change in the input data can lead to significant changes in the variable importances.

It is expected that many of our features are highly correlated (especially with the SMA and EMA). this is because there is auto-correlation in the prices. There are $72X72$ correlated pairs or 2556 pairs if we remove all self-correlated and the duplicate ones.

Let's for instance look at the 20 most correlated (positive or negative) variables.

```
corr_matrix = df3.corr()

# unstack the get the list of all pairs and put it in dataframe
corr_pairs = corr_matrix.unstack()
corr_pairs_df = pd.DataFrame(corr_pairs,
 ↪  columns=['Correlation']).reset_index()

# Rename the columns for clarity
corr_pairs_df.columns = ['feature_1', 'feature_2', 'Correlation']

# Remove self-correlations
corr_pairs_df = corr_pairs_df[corr_pairs_df['feature_1'] !=
 ↪  corr_pairs_df['feature_2']]

# Drop duplicate pairs
corr_pairs_df['abs_corr'] = corr_pairs_df['Correlation'].abs()
corr_pairs_df = corr_pairs_df.sort_values(by='abs_corr',
 ↪  ascending=False).drop_duplicates(subset=['Correlation']).drop(columns=['abs_corr'])

# Sort the pairs by the absolute value of the correlation
sorted_corr_pairs = corr_pairs_df.sort_values(by='Correlation',
 ↪  ascending=False)

# Select the top 20 pairs
top_10_corr_pairs = sorted_corr_pairs.head(10)

print(f"There are
 ↪  {len(corr_pairs_df[corr_pairs_df['Correlation'].abs() > 0.8])}
 ↪  highly correlated pairs (above 0.8) from the
 ↪  {len(sorted_corr_pairs)} possible combination.")
```

```
There are 428 highly correlated pairs (above 0.8) from the 10296 possible combination.
```

There is clearly opportunities to reduce our amount of variables and/or reduce the correlation of our features.

	feature_1	feature_2	Correlation
125	ret_1d_shifted	sma_2d	0.999916
20154	sma_58d	sma_54d	0.998280
19866	sma_50d	sma_54d	0.998001
19864	sma_50d	sma_46d	0.997598
19576	sma_42d	sma_46d	0.997016
19431	sma_38d	sma_42d	0.996296

	feature_1	feature_2	Correlation
19429	sma_38d	sma_34d	0.995332
17692	sd_56	sd_59	0.994775
17688	sd_56	sd_53	0.994400

Now we have 2 options to build our base models. Either, we transform the data through dimensionality reduction (PCA) to remove correlation or we select only a few 'less' correlated features. In the next section, we will explore both cases.

1.6 Base model

1.6.1 Base model with PCA

The intuition behind using PCA for our base model is 2 folds.

- remove all form of correlation between data. This is because all the components are orthogonal to each other in the hyperspace.
- reduce the number of features (**parsimony principle**). As our dataframe contains highly correlated features, we can reduce the number of features and still keep a high amount of variance in our data.

After some trials, we decided to keep 95% of the variance in our data through the use of PCA.

```
#let's scale the data first
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
print(x_train_scaled.shape)

pca = PCA()      # apply PCA with all the components
x_train_pca = pca.fit_transform(x_train_scaled)
x_train_pca.shape

cumul_var_ratio = np.cumsum(pca.explained_variance_ratio_)
print(cumul_var_ratio[1:30])

# to keep 99% of the variance
num_comp = np.argmax(cumul_var_ratio > 0.95) + 1
print(f"{num_comp} components already explain 95% of the variance")
```

```
(1088, 144)
[0.37834386 0.42968914 0.45841187 0.48467234 0.5093519  0.53278288
```

```
 0.55369435 0.57437601 0.59469219 0.61489769 0.63469291 0.65415441
 0.67293189 0.69137148 0.70820194 0.72420138 0.73974373 0.75430529
 0.76717734 0.77996071 0.79140375 0.80250373 0.8131996  0.82292655
 0.832519   0.84180223 0.85070097 0.85951491 0.86819362]
44 components already explain 95% of the variance
```

44 components is a lot less than the initial 144 and we are not loosing that much information.

Now the number of components to use for a base model is itself a parameters that we can tune. Let's see what different number of components with different kernels provide us. To make sure that we are not fine tuning anything with the test set in mind and also knowing that with such high number of features, we will check results on the *cross-validation training set* using *time-series split*. Using a pipeline, we can now create our first base model using PCA.

```
pca_levels = [0.8, 0.85, 0.9, 0.95, 0.99]
kernel_type = ['linear', 'rbf', 'poly', 'sigmoid']
df_xval_score = pd.DataFrame(columns =
 ↪  ['pca_levels','kernel_type','xval_score'])
tscv = TimeSeriesSplit(n_splits = 10, gap = 1)


for level in pca_levels:
  for k_type in kernel_type:
    model_svm_base = Pipeline([
      ('std_scaler', StandardScaler()),
      ('minmax_scaler', MinMaxScaler()),
      ("pca", PCA(n_components = level, random_state = 42)),
      ('classifier', SVC(kernel = k_type, random_state = 42))
      ])

    score_xval = cross_val_score(estimator = model_svm_base,
                                 X = x_train, y = y_train, cv = tscv,
                                 scoring = 'accuracy',
                                 n_jobs = 3).mean()
    new_row = pd.DataFrame({'pca_levels': [level],
                            'kernel_type': [k_type],
                            'xval_score': [score_xval]})
    df_xval_score = pd.concat([df_xval_score, new_row],
 ↪  ignore_index=True)

print(df_xval_score)
```

First interesting observation is that the *rbf* kernel has not made the top of the list. Considering the non-linear nature of financial time-series, I would have intuitively thought it

would be a better kernel to use.