



Fitch Learning
25 Canada Square
London, E14 5LQ

t: +44 (0) 845 072 7620
f: +44 (0) 20 7496 8607
w: cqf.com

Certificate in Quantitative Finance
Final Project Declaration – January 2019 Program

I hereby declare that the project work entitled:

MARKET PREDICTION WITH MACHINE LEARNING

submitted to the Certificate in Quantitative Finance, provided by Fitch Learning is a record of an original work and contains only work completed by myself. Information from other sources has been duly referenced and acknowledged.

The project has not been submitted to any other organisation for the award of any certificate, degree or diploma.

I understand that failure to comply may result in disqualification from the program.

Delegate Signature:

Mattia Pennacchetti

Delegate Name (Printed):

MATTIA PENNACCHIETTI

Date:

13/07/2019

MARKET PREDICTION WITH MACHINE LEARNING

Certificate in Quantitative Finance *(January 2019 cohort)*
Final Project

Candidate: Pennacchietti Mattia

INDEX

Pag.

1	Introduction	5
2	Assets and data selection, features engineering and exploratory data analysis	
	2.1 Assets and data selection	7
	2.2 Features engineering	9
	2.3 Models for training and initial exploratory data analysis	13
	2.4 Models selection	15
3	Statistical learning algorithms for classification	
	3.1 Normalizing data	21
	3.2 Logistic regression	25
	3.3 Naïve Bayes	33
	3.4 Support Vector Machines	34
	3.5 K-nearest neighborhood	38
	3.6 Trees, Random forests and AdaBoost	40
	3.7 Artificial Neural Networks	45
4	Conclusions	48

LIST OF FIGURES

<u>N°</u>	<u>Figure Description</u>	<u>Pag.</u>
1	Volatilities / Classes pairplot for APPLE (10 day lag)	16
2	Volatilities / Classes pairplot for APPLE (30 day lag)	17
3	Fundamentals / Classes pairplot for APPLE (10 day lag)	18
4	Fundamentals / Classes pairplot for APPLE (30 day lag)	19
5	Features relevance for the assets universe	23
6	K-10 fold test score for a range of inverse regularization parameters	29
7	50/50 test score, with/without regularization	30
8	K10-fold test score, with/without regularization	30
9	Logistic classifier by feature with C = 100	32
10	SVM classifier with different levels of C [1,100]	36
11	50/50 test score, hard and soft margins	37
12	K10-fold test score, hard and soft margins	37
13	KNN classifier for different levels of neighbors [5,500]	39
14	Random Forest K10-fold test score for a range of base trees	43
15	50/50 test score for simple tree classifier, RF and AdaBoost	44
16	K10-fold score for simple tree classifier, RF and AdaBoost	44
17	Neural networks graphical representation	45
18	Neural Networks MLP K-10 fold test scores for a range of hidden layers	47

LIST OF TABLES

<u>N°</u>	<u>Figure Description</u>	<u>Pag.</u>
1	Assets selection	7
2	Data structure	8
3	Technical indicators	19
4	Volatility indicators	11
5	Returns to Classes mapping	13
6	Data features shift process	14
7	Relevant features for the assets	24
8	Logistic training set score, no regularization	26
9	Logistic test set scores, no regularization	28
10	Logistic test set scores, with regularization $C = 10$	28
11	Logistic bagging classifier K10-fold scores	31
12	Naïve-Bayes scores	33
13	SVM scores, hard margins – $C = 1e5$	35
14	SVM scores, soft margins – $C = 1$	35
15	K-nearest neighbors, $K = 5$ (default)	38
16	Simple Decision Tree training and test scores	41
17	Random Forests test scores	42
18	AdaBoost test scores	42
19	ANN, MLP with 100 hidden layers training and test scores	46
20	K10-fold scores for all algorithms	48

1. INTRODUCTION

Predicting stock market returns has always been and continues to be one of the most difficult tasks for buy-side and financial investors professionals. There is abundant literature on what is the best model, the best predictor which can accurately predict the stock market, but no definitive answer seems to emerge. Machine learning is nowadays considered to be a potential game changer in this regard. The reason why this is the case is to be found in the non-linearity which characterizes financial markets and its dynamics. Such non-linearity makes traditional prediction or equilibrium models such as the CAMP, APT obsolete and not applicable.

Techniques such as randomized decision trees or artificial neural networks have the potential to uncover the non-linear nature of financial returns hence offering better prediction results. These and other models are the subject of this report which aims to apply several classification machine learning techniques to an historical dataset of returns of various assets in order to test the predicting power of each.

The following report is composed of three main blocks:

- *Assets selection, features engineering and model selection via exploratory data analysis*

The first part of the report will cover the assets selection as well as the methodologies used for obtaining the various features to be later used in the statistical learning algorithms. It will seek to determine, for every asset, which features are more “relevant” in explaining returns.

This exploratory analysis will also be useful in choosing what model (among different returns lag) to use.

- *Statistical learning algorithms for classification*

In the third part of the report, we will employ a set of statistical learning tools traditionally used for multinomial classification, to the model chosen above.

Models will be run for each of the classification techniques for all the assets multiple times (by changing the various regularization, softness, other parameters) so as to have a broad set of results to compare and from which to draw meaningful conclusions. In addition to classification score metrics, visual results (either for parameters tuning or for a class decision boundary visualization) are provided.

- *Results comparison and conclusion*

In this final section, a comprehensive view of the results obtained so far is provided and conclusions are drawn on which estimator and for which asset provides the most reliable learning algorithm, given the training sample used in this exercise.

Note on code: *All the results provided in this report can be obtained by running the Python code files provided (running the script **main.py**). The project will refer to each module provided in the zip file (and imported in main.py) every time it is relevant in guiding the reader who is willing to reproduce this experiment. Please note it is assumed all the Python files provided as well as the excel data source (**data.xlsx**) are unzipped/moved to the user Python current working directory.*

The scripts in the zip file assume the user has installed the following packages: numpy, scipy, pandas, scikit learn.

Note on data: *Input data used in the exercise and provided in the excel raw data source ("**data.xlsx**") has been downloaded from Bloomberg.*

Note on output charts: *All charts have been produced via the code contained in the submission attachment. The bar charts are produced via excel using data from the code. The excel file with the source histograms is **OUTPUT SUMMARY DATA.xlsx**.*

2. ASSETS AND DATA SELECTION, FEATURES ENGINEERING AND EXPLORATORY DATA ANALYSIS

2.1 – Assets and data selection

The first step in this exercise is to choose what will be our assets universe: a list of traditional and alternative assets which together can offer a glimpse into more than just one market factor and hence offer a variety of results.

Given we will deal with typical equity features, as open and closing price, volume traded, fundamentals (whenever available), we will choose ETFs in the case of alternative assets, as these are equity-like products which are however a good proxy. In this way we will be able to model the data and engineer the data features homogenously all at once.

We will diversify our universe along two dimensions: a) from an asset class perspective, switching from equities into volatility, commodities, trading; b) from a geographical perspective (solely for equities), choosing companies from the three most relevant macro regions (US, Europe, Emerging Markets).

We summarize the list of assets in the table below together with a brief description, the proxy ETF in case of indices and the relative Bloomberg (BBG) ticker used for data retrieval.

TABLE 1 – Assets selection

Asset	Description	Proxy	Equity ticker
<i>Apple</i>	Apple corporation	N/A	APPLE US Equity
<i>Microsoft</i>	Microsoft Corporation	N/A	MSFT US Equity
<i>Total</i>	Total Corporation	N/A	FP FP Equity
<i>Tencent</i>	Tencent Corporation	N/A	700 HK Equity
<i>VIX</i>	CBOE Volatility Index	ProShares VIX	VIXY US Equity
<i>Gold</i>	Gold bullion	SPDR Gold Shares	GLD US Equity
<i>Credit</i>	Index of \$-IG Corporate Bonds	I-Shares ST Corp.B	IGSB US Equity

As mentioned in the introduction section, data is supplied by Bloomberg and is structured in the **data.xlsx** excel file as follows:

TABLE 2 – Data structure

Dates	PX_OPEN	PX_HIGH	PX_LOW	PX_VOLUM
.....

PE_RATIO	PX_TO_BOOK	DIVIDEND_INDICATED_YIELD
.....

The fields represented in the data structure above are Bloomberg fields and their meaning is as follows:

PX_OPEN: Price at which the security first traded on the current day.

PX_HIGH: Highest price the security reached during the current trading day.

PX_LOW: Lowest price the security reached during the current trading day.

PX_VOLUME: Total number of shares traded on a security on the current day.

PE_RATIO: Ratio of the price of a stock and the company's earnings per share.

PX_TO_BOOK_RATIO: Ratio of the stock price to the book value per share.

DIVIDEND_INDICATED_YIELD: Provides the most recently announced dividend amount, annualized based on the divided frequency, then divided by the market price.

For each asset listed in the asset table we retrieve daily values of each of the field listed above (with the exception of the last three fields which are only available for Apple, Microsoft, Total and Tencent) for the period: **24/7/2012 – 31/05/2019**.

This corresponds to 1724 data points and a time window of almost 7 years.

2.2 – Features engineering

The first step of the exercise is to calculate returns and extract from the dataset the features we will use in our algorithms later on. The routine used in main.py for this purpose is **features.py**.

The first calculation is that related to returns. A common choice for return calculation is that of log. returns. The daily log return for an asset is calculated as follows:

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

This is calculated at three different lags: 1 day, 10 day, 30 day. This will be useful later on in the initial data exploratory section where we will have a first visual insight into which type of return we would be more successful in predicting, given the available features and learning algorithms.

We now turn to the data features. We divide the features into three clusters: i) Technicals; ii) Volatilities; iii) Fundamentals (if available).

- *Technical Indicators*

The table and the formulas below give an overview of the various technical indicators used in our features script.

TABLE 3 – Technical indicators* (definitions from investopedia.com)

Technical	Description
<i>MACD</i>	Moving Average Convergence Divergence. Trend-following momentum indicator that shows relationship between two moving averages of a security's price.
<i>STOCHASTIC K</i>	Stochastic oscillator. Momentum indicator comparing a particular closing price of a security to a range of its prices over a certain period of time.
<i>RSI</i>	Relative strength index. Momentum indicator that measures the magnitude of recent price changes to evaluate overbought or oversold conditions in the price of a stock or other asset.
<i>CCI</i>	Commodity Channel Index is a momentum-based oscillator used to help determine when an asset is reaching a condition of being overbought / oversold

Calculation of the aforementioned indicators is as follows:

a) $MACD = 26 \text{ period EMA} - 12 \text{ period EMA}$

where EMA stands for Exponential Moving Average. This is calculated through the pandas Series method **ewm** appropriately selecting the attribute **span** to reflect the periods indicated in the formula above.

b) $STOCHASTIC\ K = \left(\frac{C - L14}{H14 - L14} \right) \times 100$

where:

C = the most recent closing price.

$L14$ = the lowest price traded of the 14 previous trading sessions.

$H14$ = the highest price traded during the same 14-day period.

c) $RSI_{step\ one} = 100 - \left[\frac{100}{1 + \frac{Average\ gain}{Average\ loss}} \right]$

where:

Average gain/loss = average percentage gain or loss during a certain period.

Average loss = the lowest price traded of the 14 previous trading sessions.

14-day is a common time window for this indicator hence this will be used in features.py as well.

d) $CCI = \frac{Typical\ Price - Moving\ Average}{0.15 \times Mean\ Deviation}$

where:

Typical Price = $\sum_{i=1}^P ((High + Low + Close) / 3)$ where p is the number of periods.

Moving Average = $\sum_{i=1}^P (Typical\ Price) / P$

Mean Deviation = $\sum_{i=1}^P (|Typical\ Price - Moving\ Average|) / P$

A common choice for P in this indicator is 20 days and so we'll use this value in our program.

- *Volatility Indicators*

The table and the formulas below give an overview of the various volatility indicators used in our features script.

TABLE 4 – Volatility indicators

Technical	Description
<i>Sample standard deviation</i>	Simple standard deviation calculated from a subset of the sample available.
<i>Exponential smoothing estimate</i>	Exponential smoothing estimate of volatility, which assigns growing importance (weight) to most recent returns.
<i>Drift-independent volatility¹ (DDI)</i>	An unbiased variance estimator based on multiple periods of open, close, high, and low prices. This estimator is independent of both drift motion and opening jumps.

$$a) \text{ SSD} = \frac{1}{n-1} \sum_{i=1}^n (r_i - \mu)$$

where:

μ = mean return for the period

$$a) \text{ Exp}(V_t) = \lambda * V_{t-1} + (1 - \lambda) * r_t^2$$

where:

λ = smoothing/decay parameter. In practice, and in our report as well, this is set to 0.94.

We finally take the square root of the estimate in order to get a number in the same units as our returns.

$$c) \text{ DDI} = V_0 + k * V_c + (1 - k) * V_{RS}$$

where:

$$V_0 = \frac{1}{n-1} * \sum_{i=1}^n (o_i - o^{\text{mu}})^2$$

$$V_c = \frac{1}{n-1} * \sum_{i=1}^n (c_i - c^{\text{mu}})^2$$

¹ For more details please see Dennis Yang, Qiang Zhang: “Drift-Independent Volatility Estimation Based on High, Low, Open, and Close Prices”.

$$V_{RS} = \frac{1}{n} * \sum_{i=1}^n [u_i * (u_i - c_i) + d_i * (d_i - c_i)]$$

$$K = \frac{\alpha - 1}{\alpha + \frac{n+1}{n-1}}$$

As suggested in the paper, a sensible estimate for alpha to be used in the formula is 1.34. Also, the other following definition apply:

$o = \ln(O_1) - \ln(C_0)$, the normalized open .

$u = \ln(H_1) - \ln(O_1)$, the normalized high.

$d = \ln(L_1) - \ln(O_1)$, the normalized low.

$c = \ln(C_1) - \ln(O_1)$, the normalized close.

Where O_1 , H_1 , L_1 , C_0 , C_1 are directly taken from the raw data contained in excel. This information is indeed included in the fields ("PX_LAST", "PX_OPEN", "PX_HIGH", "PX_LOW").

- *Fundamental Indicators*

The last set of indicators used in this exercise are fundamental ones. These are metrics typically employed by "value" investors in order to spot so called "value" companies, i.e. companies whose shares are trading at a relative cheap price but which (fundamentally) are financially sound as well as profitable. Given these features are readily obtained by Bloomberg straightaway (as already explained in the data structure section), no further transformation is needed. More specifically, we will be using: i) Price-To-Earnings ratio; ii) Price-To-Book value; iii) Dividend yield. Given not all the assets in our universe are pure equities, these features won't be relevant for VIX, Gold and Credit.

2.3 – Models for training and initial exploratory data analysis

Now that we have calculated returns and the various features for all of the assets in our universe, the next step is create a model to be then used as training sample in the following section. Given this is a classification problem, we need to “transform” our output variable: returns, into “classes”. A common setting in this case is to create a binary classification where there are only two classes (positive returns vs. negative returns). Such a setting however is not particularly interesting as it wouldn’t be very informative. For example, when predicting returns for different purposes (as might be in the case of asset allocation, risk budgeting etc.) we are usually interested in worst-case scenarios of the asset, or best-case – we’re interested in various regimes of the market and the transition probabilities attached to them.

For the purposes of this exercise we have chosen a 5 classes model. Classes are determined as explained in the table below. The script associated with such calculation and labels creation is **models_for_training.py**.

TABLE 5 – Returns to Classes mapping

Class	Description	Calculation Threshold (X)
Class 1	Extremely positive return	Greater than 2
Class 2	Positive return	Between 2 (included) and 0.5
Class 3	Neutral	Between 0.5 (included) and -0.5
Class 4	Negative return	Between -0.5 (included) and -2
Class 5	Extremely negative return	Less than -2

In order to calculate the threshold, so as to determine which class the observation belongs to, we derive the ratio:

$$X = \frac{r_{t,t+n}}{\sqrt{n} * \sigma_{t+n}}$$

where:

$r_{t,t+n}$ = Log return for the period $[t, t+n]$ where n is our time lag

σ_{t+n} = Simple standard deviation calculated at time $[t+n]$ using (n) past daily returns


Given we have calculated three type of returns: i) daily; ii) 10-day returns; iii) 30-day returns, we have three models (i.e. three different sets of classified output variables for our complete set of features).

The last step in this model creation is to **shift** all the features variables for the number of days accounted for in the returns calculation.

The illustration below explains what the result of this procedure is.

TABLE 6 – Data features shift process

<u>PRICES</u>	<u>DAILY RETURNS</u>	<u>RETURN LAG M</u>	<u>VOLATILITY FEATURE</u>
Price day 1	n/a	n/a	n/a
Price day 2	Return day 2	n/a	n/a
....	n/a	n/a
Price day m	Return day m	Return [1 – m]	Feature (Vol) [1-m]
Price day m +1	Return day m +1	Return [2 – m+1]	Feature (Vol) [2-m+1]
Price day m +2	Return day m +2	Return [3 – m+2]	Feature (Vol) [3-m+2]
....
Price day m +n	Return day m +n	Return [m – n]	Feature (Vol) [m-n]



Considering the **Lag M** model, the data frame hence becomes:

<u>RETURN LAG M</u>	<u>VOLATILITY FEATURE</u>
Return [m – n]	Feature (Vol) [1-m]
Return [m+1 – n+1]	Feature (Vol) [2-m+1]
Return [m+2 – n+2]	Feature (Vol) [3-m+2]
....

The reason for this shifting is the following (here we talk about standard deviation, but a similar argument can be extended to the other data features as well): let's say we have used the **previous m day returns** to estimate the volatility prevailing today. What we're interested in, is using such estimate to predict what the return

might be over **next m** days.

Now we have a complete dataframe composed of the output variables “CLASSES” and our set of features. The next question is which dataframe (using which return lag) we should choose.

The initial visualization and data exploratory analysis will be a hint for which model to use in going forward with the analysis contained in this report.

2.4 – Models selection

The main purpose of this section is to offer a first glimpse into the relation between our features and the returns classes, calculated with returns at various lags. We will do so through a common graphical tool used in this context: the pairplot. The charts shown are generated via the routine **exploratory.py**.

As can be seen from the script `exploratory.py`, the main function (`pairplots()`) takes as input a model – which can be the dataframe we derived in the previous section – made of classes (calculated with returns at 1, 10, or 30 days lag) and the various features, plus a ticker (i.e. an asset) for which to conduct the exploratory data analysis. The function `pairplots()` helps visualize the correlation among the various features and the classes, via a simple pairplot. Seaborn is the package used in order to produce these. In the below we will show these graphical illustrations for **Apple**.

In particular the pairplot will highlight the volatilities, technical and fundamental features at 10 and 30 days lags (the other charts can be created by executing `main.py` and selecting the desired asset and feature set to visualize). This will build our intuition into which model to use in order to feed the various classification algorithms.

Figure 1 – Volatilities / Classes pairplot for APPLE (10 day lag)

Pairplot for: APPLE

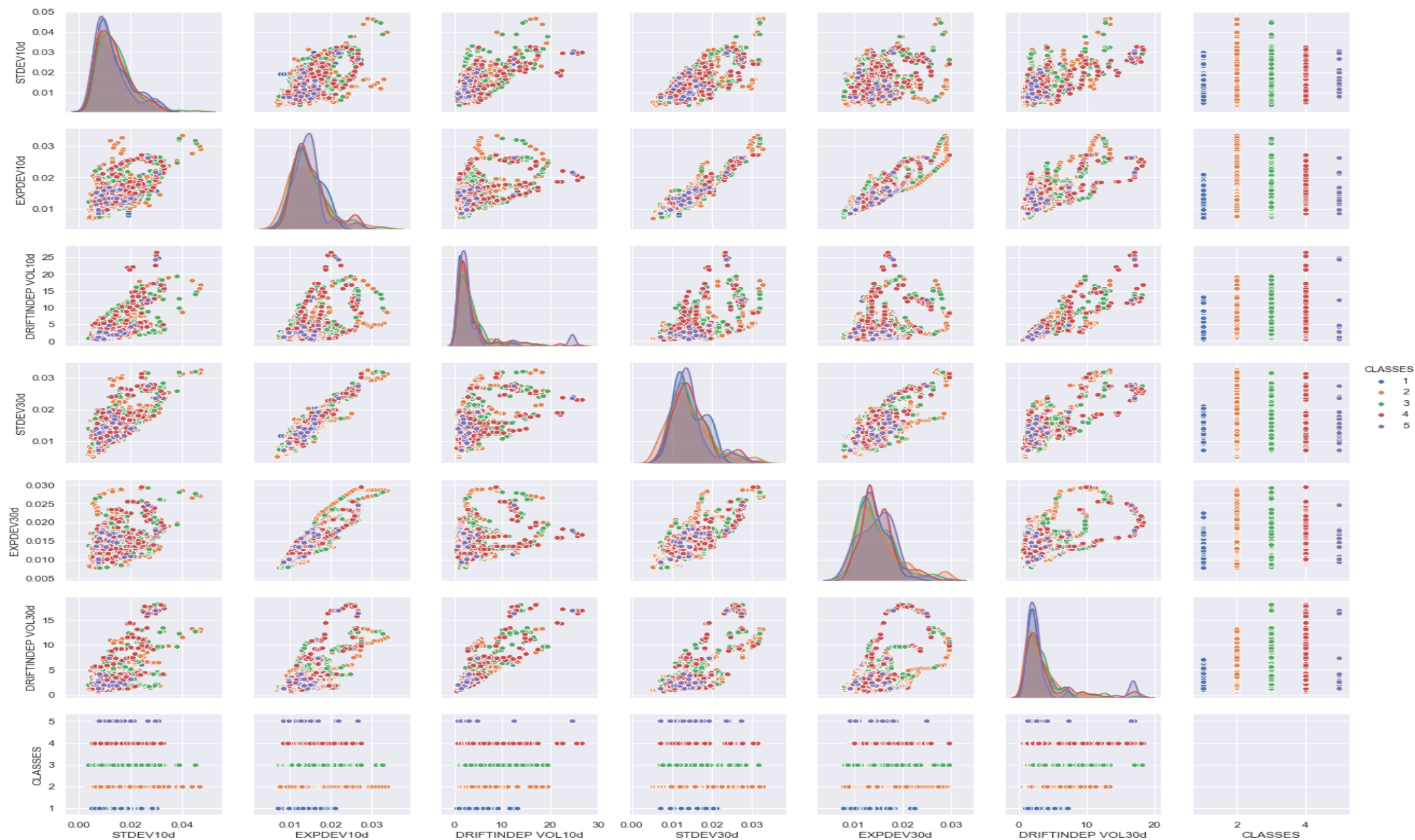


Figure 2 – Volatilities / Classes pairplot for APPLE (30 day lag)

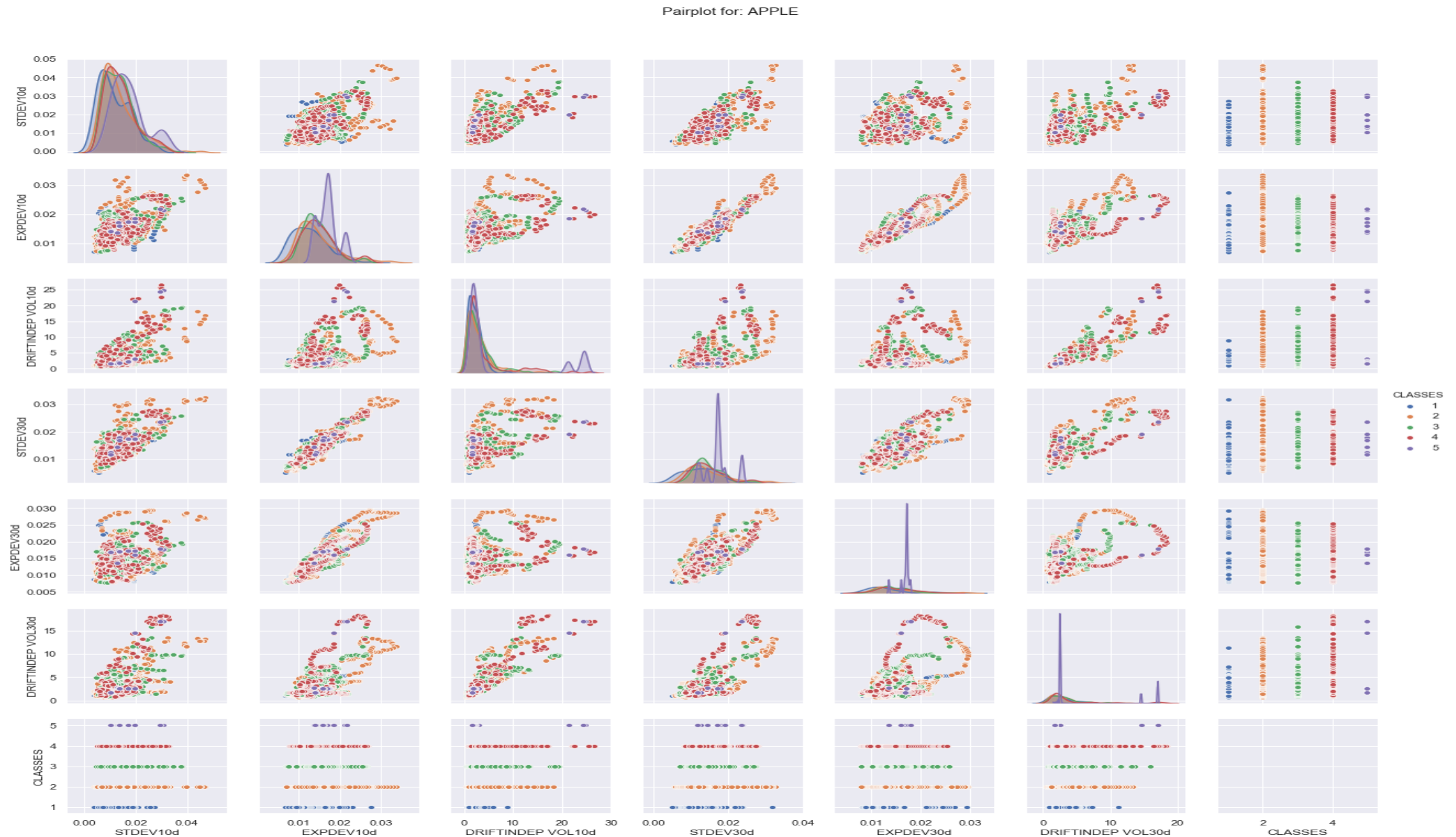


Figure 3 – Fundamentals / Classes pairplot for APPLE (10 day lag)

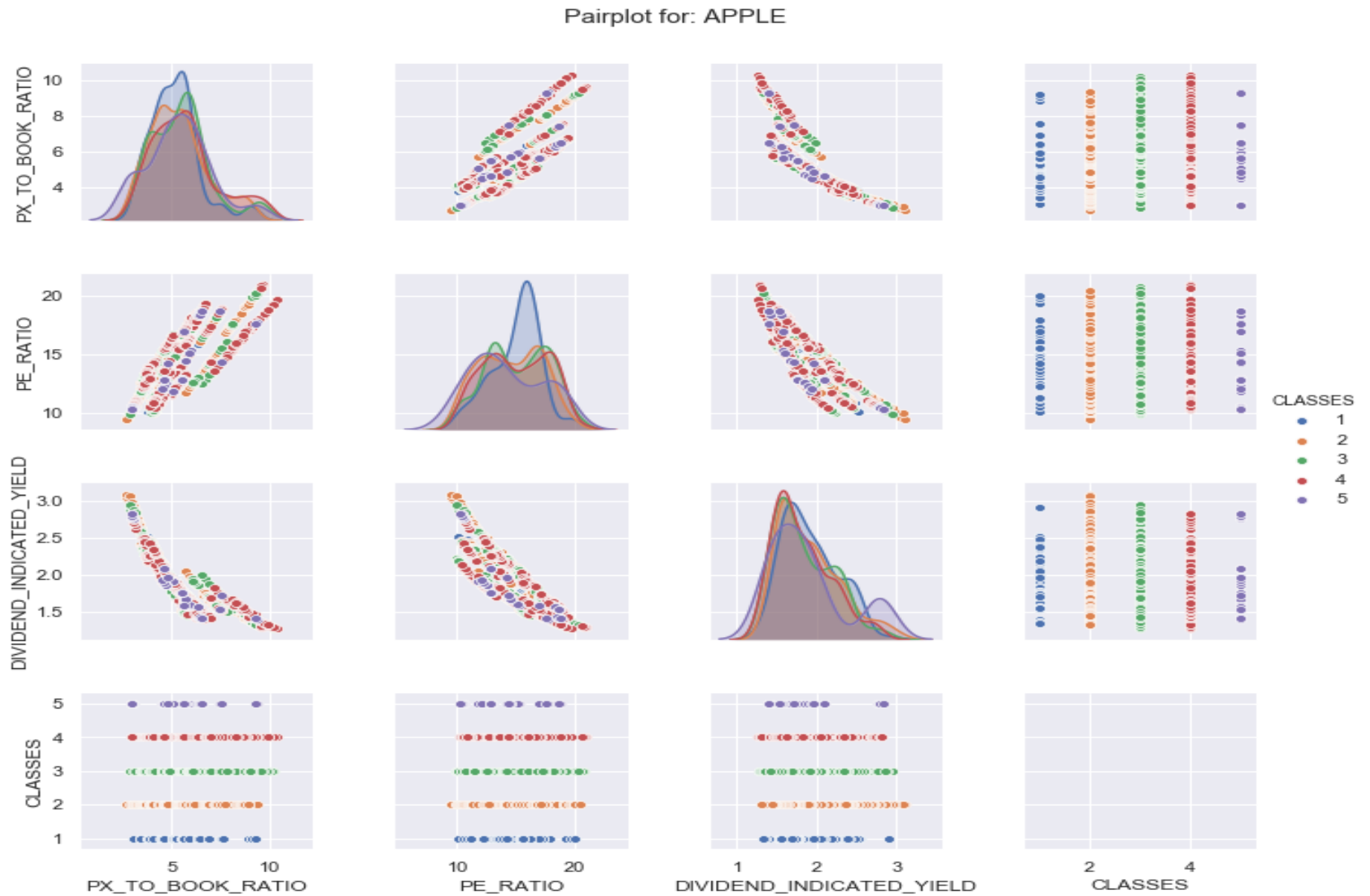
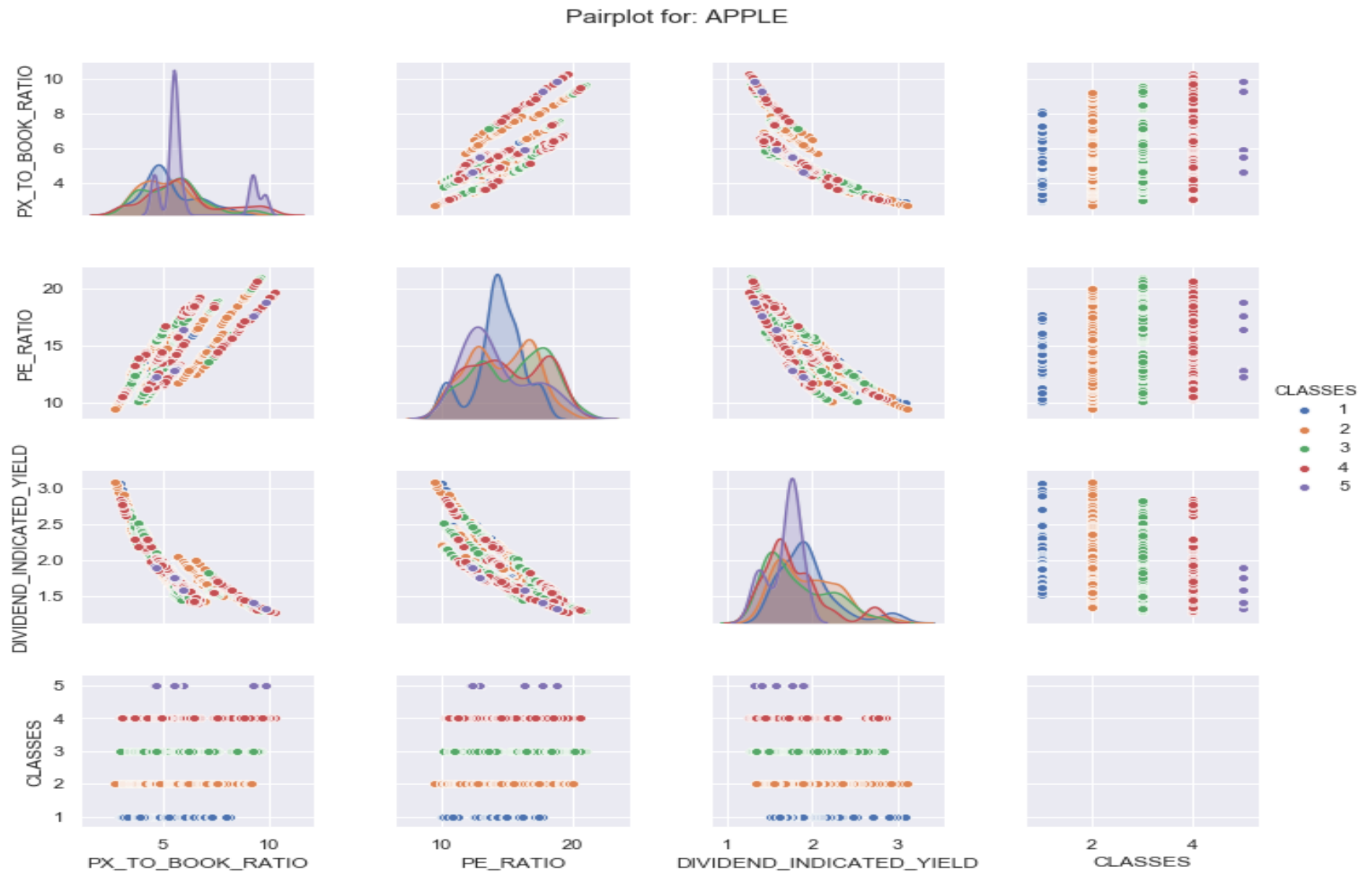


Figure 4 – Fundamentals / Classes pairplot for APPLE (30 day lag)



The pairplots above of Apple are self-explanatory. The model with the classes calculated over 30 day-lag returns seem to be more powerful as the features seem to capture much better the intra-class variation of the available observations. In order to see this, we must focus our attention to the last subplots row where we have a scatter plot of the classes vs. the features. Both volatilities but especially fundamentals, seem to be much more powerful when having to predict 30 day return vs. 10 days.

In the analysis we will illustrate going forward we will therefore use this model. One last remark: even if we had not had a visual inspection of our features vs. classes it could be sensible to choose 30 day return lag as appropriate model to analyze if our purpose was a tactical asset allocation exercise. In such contexts, portfolio/managers investors need to form their views/predictions not on the day to day return but on the performance with a monthly horizon.

3. STATISTICAL LEARNING ALGORITHMS FOR CLASSIFICATION

3.1 – Normalizing data

Before jumping into applying various classification techniques, we need two further refinements to our data:

- I. Normalization
- II. Feature selection

In data science, normalization is a technique often applied as part of data preparation and its primary goal is to change value of our predictor variables to a common scale, without distorting differences in the range of values. However, this is not always needed: indeed, when the features all share a common range, then there is no need for such normalization. In our case it is quite clear how that is not the case: we are aggregating volatility data, technical indicators, fundamental indicators. All these three clusters of predictors substantially differ in the range of values they can take. Hence, we normalize. The routine for doing so is shown in **model_prepare.py** (*function normalize*).

For every column, every feature, we apply the following transformation:

$$\text{Norm. } X_{i,j} = \frac{X_{i,j} - \mu_j}{\sigma_j}$$

where:

i = observation n.i

j = feature j

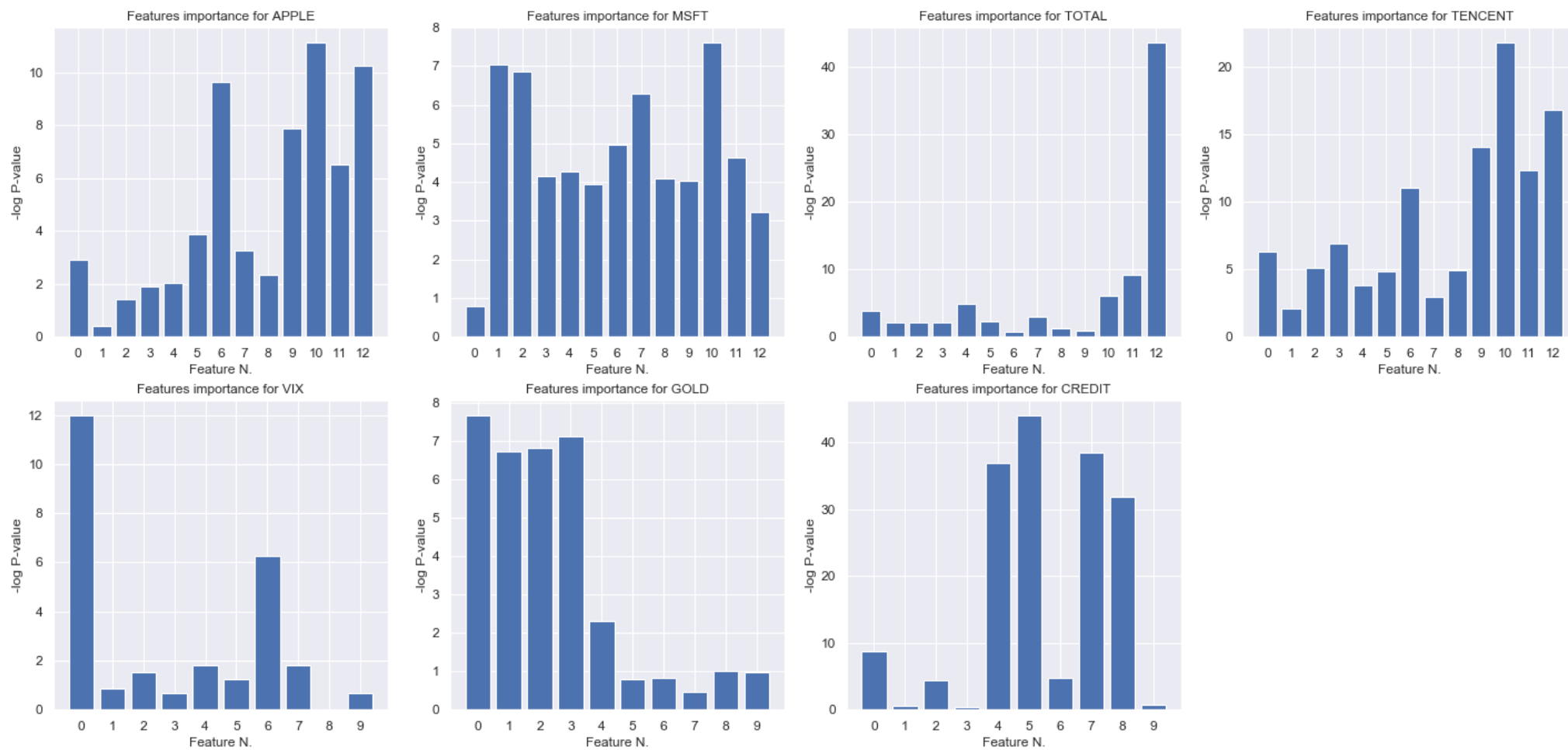
The second preparation technique is the aforementioned features reduction. This simply means avoiding to include in the data unnecessary predictors (by unnecessary we mean, not “relevant” or not “significant” according to a determined criteria of significance in a statistical learning setting). Before explaining how we do that, a quick note on why this is an important step of the process. Adding features is equivalent to adding dimensions. While more dimensions can help capture some of the **observed** variability in the response (thereby reducing the bias of the training

set), it often leads to a decreased test error (i.e. the mean squared error or other prediction accuracy score calculated on a test set). This is the result of the phenomenon known as “*over-fitting*” or the “curse of *high dimensionality*”. It is therefore important, that among our predictors, we choose the ones we really need, i.e. whose “significance” is high enough to justify a possible over fitting. Where this “high enough” parameter stands, is an open question.

In our python files, **model_prepare.py** contains the function used for this purpose (this is called *feature_select*). The script uses the **Select Percentile** and **f_classif** functions, from the **sklearn.feature_selection** library. *Select Percentile* allows us to select features according to a percentile of the highest scores. It returns scores and p-values according to a regression/classification function it applies on the two arrays (X,y). In our case, given we are looking at a classification problem we use **f_classif** which simply compute the ANOVA F-value for the set of regressors provided (and tested sequentially) and the classes variable Y. The below chart shows the p-values (inverted with Log 10 to have positive numbers) for all of our assets

Figure 5 – Features relevance for the assets universe (computed as $\text{Feat.Score} = -\log_{10} p\text{-value}$)

FEATURES RELEVANCE



Remembering that our features are: [0: MACD, 1: STOCHK, 2: RSI, 3: CCI, 4: STDEV10d, 5: EXPDEV10d, 6: DRIFTINDEP VOL10d, 7: STDEV30d, 8: EXPDEV30d, 9: DRIFTINDEP VOL30d, 10: PX_TO_BOOK_RATIO, 11: PE_RATIO, 12: DIVIDEND_INDICATED_YIELD].

The chart above reveals different patterns and different relevance for the various assets although we can immediately spot a common trend. Equities seem to privilege fundamental indicators; Credit seem to rely mostly on standard and exponential estimate of volatility, while VIX and GOLD seem to be best explained via momentum indicators. Finally, via our **prepare** function (also this one contained in *model_prepare.py*) we “slice” for each asset , the respective dataframe we keep in our dictionary, so that it contains only the features (the columns) whose p-value, calculated via the ANOVA f-test described above, is relevant at a 5% threshold.

After applying such threshold, the features associated with every asset are represented in the below table (green if relevant, red if not, N.A. if not available):

TABLE 7 – Relevant features for the assets

Feature	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
MACD							
STOCHK							
RSI							
CCI							
ST. Dev (10d)							
EXP. Dev (10d)							
DRIFT Indep. (10d)							
ST. Dev (30d)							
EXP. Dev (30d)							
DRIFT Indep. (30d)							
P/B RATIO					N.A.	N.A.	N.A.
PE RATIO					N.A.	N.A.	N.A.
DIV YLD					N.A.	N.A.	N.A.

After having normalized the data and operated a first dimensionality reduction we can now feed the model to a different set of classifiers / algorithms whose results will be explored sequentially. For each algorithm we will illustrate training and test scores, limitations, extensions pipelines (via bagging or boosting methodologies). Finally, we will produce an output table summarizing the various results in order to draw some conclusions on which method might be more suitable for the purposes of this project, i.e. predicting market returns via machine learning techniques.

3.2 – Logistic regression

Logistic regression is one of the most common linear models used for classification (known usually also as logit regression or log-linear classifier). In its basic form, logistic regression is used for binary classification problems (i.e. the output variable has only two labels, that we can express as a 0 and 1). In such a basic setting, the function used for minimization purposes is the **Sigmoid function**;

$$F(\theta, x) = \frac{1}{1+e^{-\theta^T x}} \quad \text{and associated cost function:}$$

$$J(\theta) = -\frac{1}{m} \sum_m \left[y^{(i)} \log(h(\theta)(x(i))) + (1 - y^{(i)}) \log(1 - h(\theta)(x(i))) \right]$$

Our case unfortunately is more complex since we are interested in multinomial prediction. We will then be using the **Softmax** function. Let's suppose y can take k classes. Then we have:

$$F(h(\theta, x)) = \begin{bmatrix} P(y=1|h(x)) \\ P(y=2|h(x)) \\ P(y=3|h(x)) \\ \dots \\ P(y=k|h(x)) \end{bmatrix} = \frac{1}{\sum_{j=1}^k e^{(\theta(j)^T x)}} * \begin{bmatrix} e^{(\theta(1)^T x)} \\ e^{(\theta(2)^T x)} \\ e^{(\theta(3)^T x)} \\ \dots \\ e^{(\theta(k)^T x)} \end{bmatrix}$$

with cost function:

$$J(\theta) = -\sum_{i=1}^m (1 - y^{(i)}) * \log(1 - h_{\theta}(x^{(i)})) + (y^{(i)}) * \log(h_{\theta}(x^{(i)}))$$

In order to implement this through the python sklearn library, we use the **linear_model** section of sklearn and in particular the **LogisticRegression** sub-package. In the code package provided with this project, this is implemented in *Logistic.py*.

In order to make our Logistic Regressor work in a multi class environment, the parameter “**multi_class**” is set to “**multinomial**”. The other key parameter we need to set before running this function in python is **C**. This represents the inverse of the regularization strength. Smaller values specify stronger regularization. This can be seen from the below formula where C regulates the strength we gives to the “unregularized” cost function. (The below formula uses **L2** regularization).

$$\min_{w,c} \frac{1}{2} h_{\theta}^T h + C \sum_{i=1}^m (1 - y^{(i)}) * \log(1 - h_{\theta}(x^{(i)})) + (y^{(i)}) * \log(h_{\theta}(x^{(i)}))$$

We start by setting the parameter equal to **1e5**, equivalent to the most flexible model possible with no regularization parameter. We have a first run of the classifier and calculate the prediction accuracy, by using the sklearn function **score**. This function returns a percentage number, i.e. the fraction of cases in which the algorithm correctly predicts the class/label of the output variable. Results are in the below table.

TABLE 8 – Logistic training set score, no regularization

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Score on training set</i>	48.23%	56.09%	49.61%	61.82%	45.04%	43.91%	51.16%

We see from the results in the table above that predictions over the training set do not go much further than 50%, except for Tencent in which case the score is c. 62%. Testing the prediction accuracy of a classifier solely using the training set is not really indicative as in real life we will likely have new observations which do not

necessarily resemble those in the training set. And it is on those observations that we need to test the prediction power of our trained algorithm. In order to carry out such exercise we will use two approaches: i) naïve train / test split; ii) K-fold cross validation test. Applying iteratively these procedures and see how the scores change will represent our standard pipeline which we will be using for all of the classifiers examined.

- Naïve train / test split

In this case we simply split the initial training set into two slices, according to a pre-determined split percentage. In our case we choose 50%; this means that we will divide our data into two sets, each representing 50% of the total observations. We will first train our model on the initial 50% and subsequently, test that very same model on the remaining 50% of the data. In our code we do this via the function **cross_validation** contained in **model_results.py**.

- K-fold cross validation test

K-fold is a cross validation test done via a resampling procedure. Such procedure takes a single parameter, k. K refers to the number of groups that the data sample will be split into. For each of the k groups, the group is taken as a test data set with the remaining k-1 groups being merged and used for training. We calculate the score and discard the model. We repeat these procedure for each of the k groups, and once we have all the k scores, we average them out. In the code this is implemented via the **k_fold_cross_val** function, still within **model_results.py**

The below table reports the score we obtain via the two in-sample test validation techniques (still with no regularization).

TABLE 9 – Logistic test set scores, no regularization

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>50/50 Naïve split</i>	44.15%	50.66%	49.93%	59.84%	44.15%	44.02%	49.33%
<i>K10-fold cross split</i>	46.91%	53.76%	48.17%	59.69%	44.44%	42.91%	50.11%

So far, all the results have been obtained with the regularization inverse parameter equal to $1e5$ (non-existent regularization). We now try to set this constant **C** to a lower number so as to introduce regularization in our model and see whether the test scores (both the naïve 50/50 and the k-fold 10 cross splits) experience any improvement).

In order to do so, we calculate the K10-fold test score for a range of models, for each asset. This collection of models are indexed by a single parameter (our constant C, the regularization inverse parameter). We then plot, asset by asset, the test score curve against the constant and see whether there is some convergence or some stabilization from which we can choose our C.

Once we do this exercise, it seem that $C = 10$ is a threshold point after which, no substantial improvement in the test score is obtained by subsequent models (with the exception of Apple and Total, for which increasing the parameter leads first to a decrease in the test score which then normalizes). We therefore set $C = 10$ and repeat the pipeline shown above (**1 – Model fit / 2 – 50/50 test / 3 – K10-fold test**).

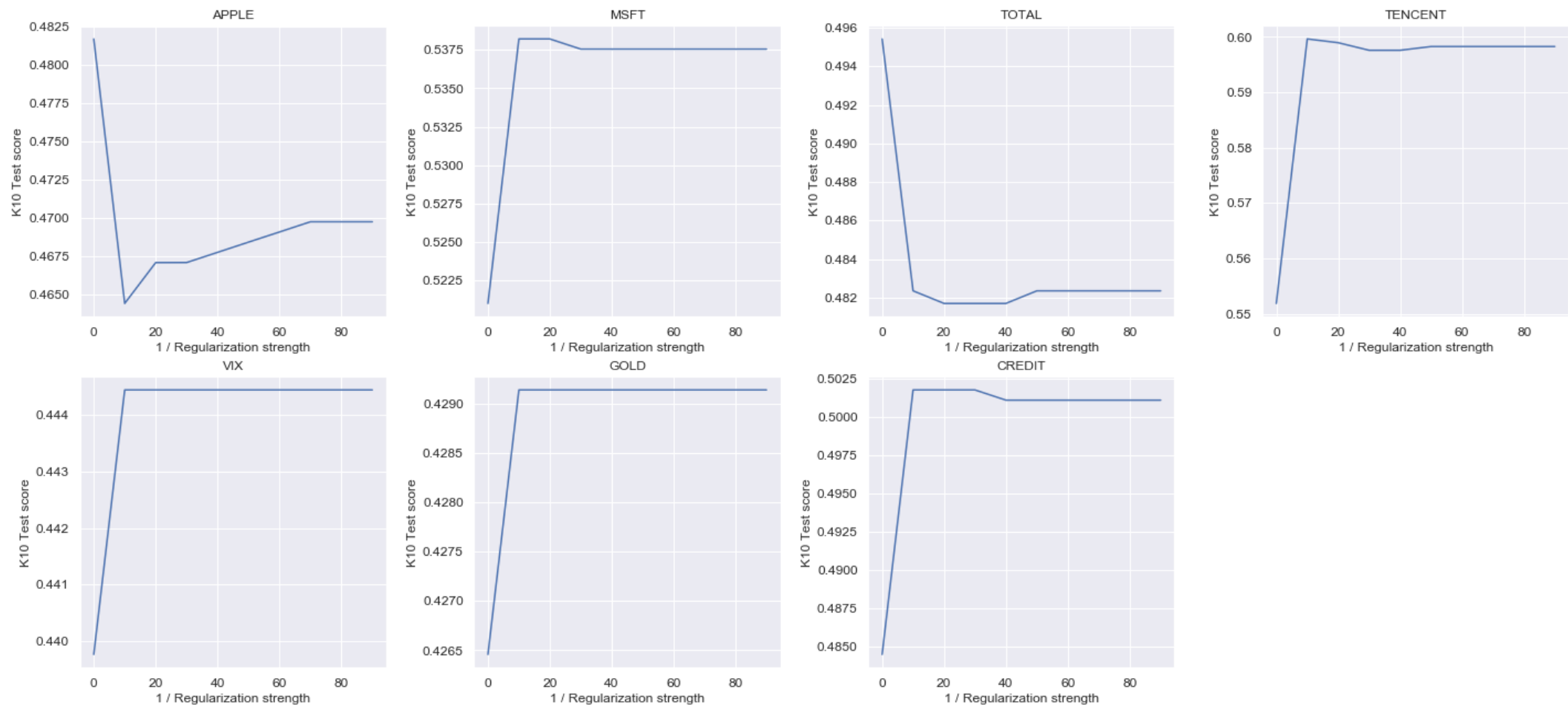
Results are in the below table.

TABLE 10 – Logistic test set scores, with regularization $C = 10$

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>50/50 Naïve split</i>	47.61%	53.99%	48.37%	60.65%	42.55%	44.95%	48.53%
<i>K10-fold cross split</i>	46.44%	53.82%	48.23%	59.96%	44.44%	42.91%	50.17%

Figure 6 – K10-fold test score for a range of inverse regularization parameters

LOGISTIC K10 FOLD CROSS SCORES FOR DIFFERENT REGULARIZATION PARAMETERS



In order to get more of a visual impression from the table we plot the difference in the 50/50 test and K10 fold test for all the asset pre and post regularization.

Figure 7 – 50/50 test score, with/without regularization

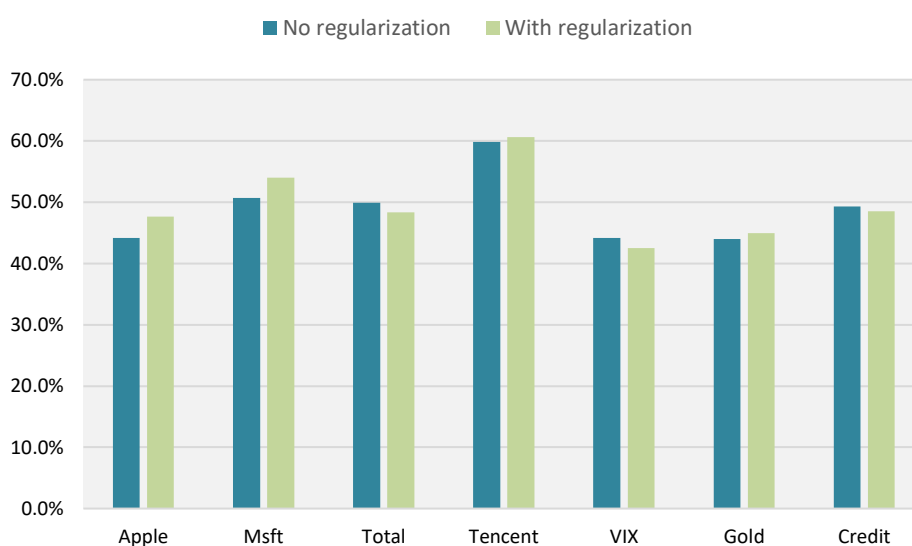
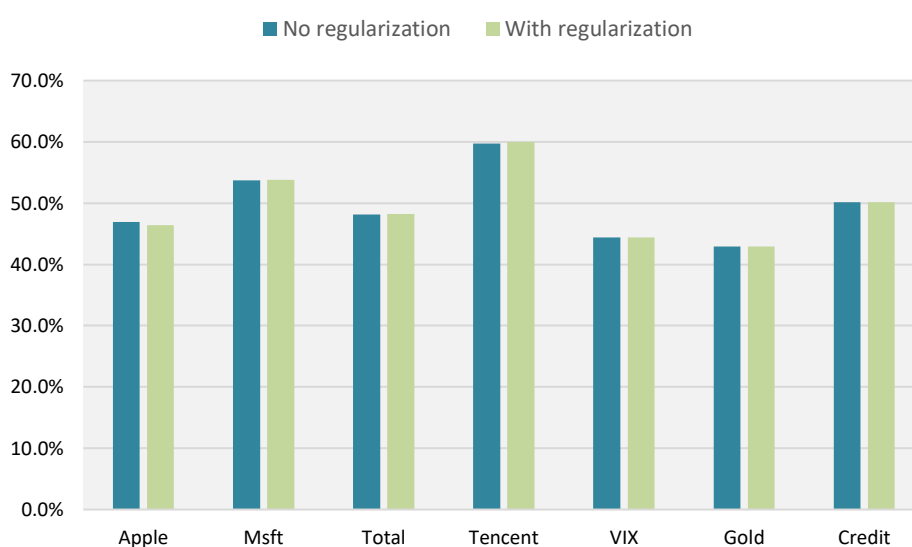


Figure 8 – K10-fold test score, with/without regularization



From the above charts we see that regularization might indeed help when applying our model to “unseen” data avoiding the common issue to flexible models known as over-fitting. In our case it seems regularization helps especially when testing our model with the naïve 50/50 test split, whilst it is less obvious (albeit present) in the K-fold method.

While the K-fold represents a substantial improvement over the naïve 50/50 test split methodology, it generally does not work well (in a classification context) when we have a binary response variable which is greatly biased towards one of the value (this is the case of so-called imbalanced samples). In order to address such issue, we can use “*averaging techniques*” between which the **Bagging Classifier**, available in the sklearn library. A bagging classifier fits a base classifier (in our case the logistic one) each on random subsets of the original datasets and then aggregate their individual predictions by averaging to form a final prediction. In addition to be potentially effective against unbalanced samples, this technique is particularly used as a way to reduce the variance of black-box estimator (such as decision trees by introducing randomization into its construction and then making an ensemble out of it).

The routine we used is contained in **model_result.py** (the function name is simply **bagging(..)**). We apply such ensemble estimator both to the regularized and unregularized models and calculate the K10 fold scores for each:

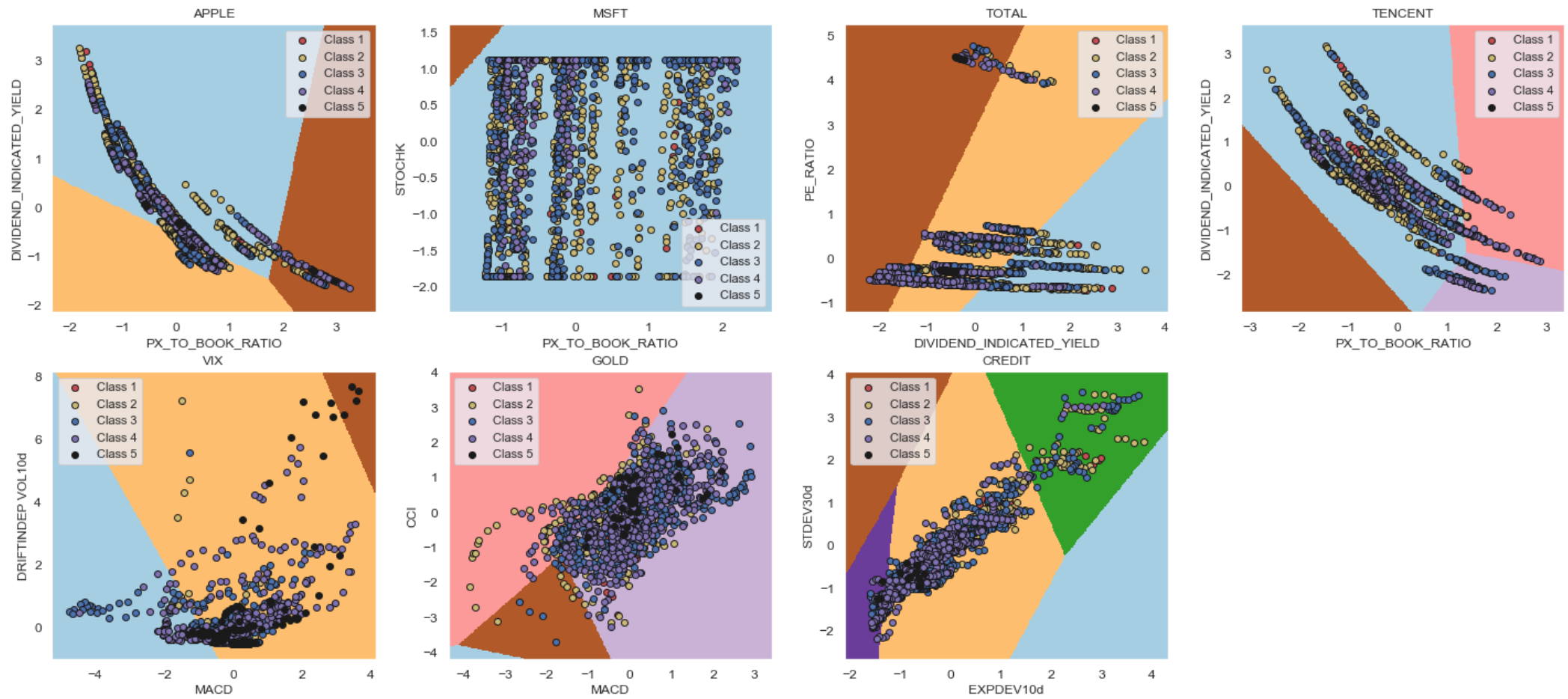
TABLE 11 – Logistic bagging classifier K10-fold scores

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
$C = 1e5$	46.57%	53.95%	48.23%	59.41%	43.64%	42.58%	50.37%
$C = 10$	45.83%	52.95%	47.97%	59.90%	44.51%	43.18%	49.05%

Finally, we show (for a value of $C = 100$) the decision boundaries, observations, and classification regions of the logistic classifier to the most relevant pairs of features (see section above on features selection) for each of our asset

Figure 9 – Logistic classifier by feature with C = 100

LOGISTIC CLASSIFIER BY FEATURE WITH REGULARIZATION PARAMETER: 100



3.3 – Naïve Bayes

The second algorithm we look at is the Naïve Bayes (NB) classifier, another supervising-learning technique that can be used for classification. The basic premise of this method is the Bayes Theorem, which (in the presence of perfect conditions where distribution probabilities of the data generating process is known) lets us calculate the probability of new data being in each class. Starting from the Bayes Theorem, using the conditional independence assumption of the covariates it is easy to derive the following formula (**m** predictors, **k** classes):

$$P(C_k|x) = P(C_k) * \prod_{m=1}^M P(x_m | C_k)$$

Given a new observation, we would need to calculate the above probabilities. X will be assigned to class K for which the above is maximized, a simple majority rule.

Whilst NB classifier can work quite well in many real-world situations, such as document classification, spam filtering, it does come with over-simplified assumptions. Whereas those assumptions do not hold true, the model risks to be of little practical use.

In our implementation via sklearn, the routine is within **Naïve_Bayes.py** we use the **GaussianNB** classifier. This classifier assumes the features to be distributed according to a Gaussian distribution. Below is the results table with the score obtained on the training set, on the 50/50 test split, and on the K-fold test.

TABLE 12 – Naïve-Bayes scores

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Score on training set</i>	36.99%	46.71%	40.00%	51.02%	39.79%	38.26%	32.47%
<i>50/50 Naïve split</i>	36.83%	44.95%	38.82%	50.82%	36.70%	36.97%	35.11%
<i>K10-fold cross split</i>	35.26%	46.11%	38.76%	49.93%	39.59%	37.86%	31.99%

As evidenced from the table above we have substantially lower scores (with respect to those obtained with the logistic model) across all our assets. This should not come as a surprise given the strong (and unrealistic) assumption of Gaussian distributed regressors.

3.4 – Support Vector Machines (SVM)

The third classification algorithm we explore is Support Vector Machines (SVM). SVM is a supervised-learning classification technique which, given a set of classified features (represented by vectors in n dimensions) divides the data via a n -dimensional hyperplane. Depending on which side of the hyperplane each point lies, this will be classified accordingly. An hyperplane is simply a subspace whose dimension is one less than that of its ambient space. This only means that if our features vectors is two dimensional, the hyperplane will be represented by a line. If our feature's dimension is three, then the hyperplane will be a plane and so on.

SVM therefore seeks to find the “best” hyperplane i.e. the one which most accurately divides and classifies the data. In a 2D setting we can think about a line dividing two sets of points. The algorithm simply tries to find the line which maximizes the margins from the line itself and the boundary of the two sets of points. The cases that define the margins are called “support vectors”. In SVM we can refer to hard or soft margins. When talking about hard margins, we refer to there being a clear boundary between the various classes. In reality, this is almost never feasible since we will likely have observations stretching into the “wrong” region. Here we then talk about “soft margins”. Assuming n dimension and a binary classification problem this reduces to a minimization problem (which is equivalent to maximizing the margins) as follow:

$$\text{margin width} = \frac{2}{|\theta|}$$

$\text{Min}_{\theta, \theta_0} \frac{1}{2} |\theta|^2$ where θ is the vector of the coefficients associated with the n features.

The sklearn library implements support vector machines with different kernels. Given the time complexity of non-linear kernels grows with quadratic speed with the dimension of the sample, we choose the **LinearSVC** function (*SVM.py* is the main routine for this classifier).

As we did before for the logistic regression, we will need to decide the value of the constant **C** which in the SVM case, refers to the “softness” of the hyperplane margins (a high value will be representative of hard margin while a low value will indicate soft margins). For the time being we consider hard margins and set $C = 1e5$. We first have a run of the classifier following the usual pipeline (*training set score, 50/50 test split score, K-fold score*) and we get the below output:

TABLE 13 – SVM scores, hard margins – $C = 1e5$

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Score on training set</i>	33.47%	45.04%	36.67%	42.55%	31.54%	38.79%	38.79%
<i>50/50 Naïve split</i>	26.73%	34.04%	43.01%	34.43%	37.10%	26.73%	38.83%
<i>K10-fold cross split</i>	34.06%	41.99%	37.25%	43.09%	36.40%	31.87%	37.80%

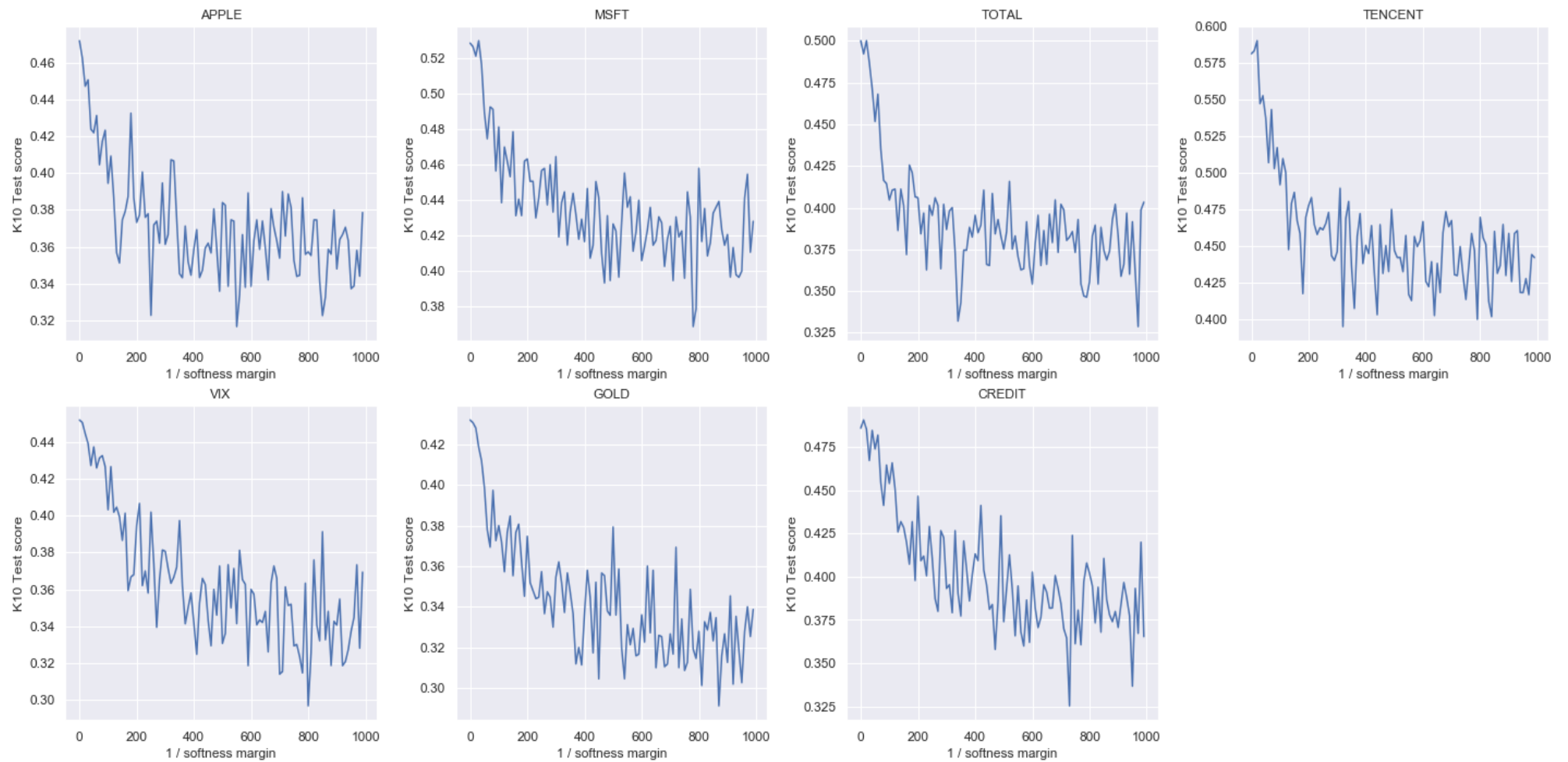
The SVM algorithm with a hard margin setting has a rather poor score, in some instances even lower than the Gaussssian Naïve-Bayes. We now try to relax the margin parameter and see whether we obtain an improvement. In order to do this we plot different K10-fold scores for a range of softness parameter and see how the test score behaves. From the chart below (Figure 12) there seems to be a clear downward trend. Increasing **C**, (therefore reducing the softness) leads to significantly lower test scores. We then set $C = 1$ and have the below scores:

TABLE 14 – SVM scores , soft margins – $C = 1$

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Score on training set</i>	46.31%	54.82%	50.52%	59.84%	45.64%	43.45%	50.23%
<i>50/50 Naïve split</i>	44.55%	52.79%	50.33%	57.38%	45.88%	40.43%	48.14%
<i>K10-fold cross split</i>	45.57%	53.36%	48.82%	57.99%	45.38%	43.11%	49.31%

Figure 10 – SVM classifier with different levels of C [1,100]

SVM K10 FOLD CROSS SCORES FOR DIFFERENT SOFTNESS MARGINS



For ease of comparison we plot in the histograms below the improvement once we switch from hard margin to soft margins.

Figure 11 – 50/50 test score, hard and soft margins

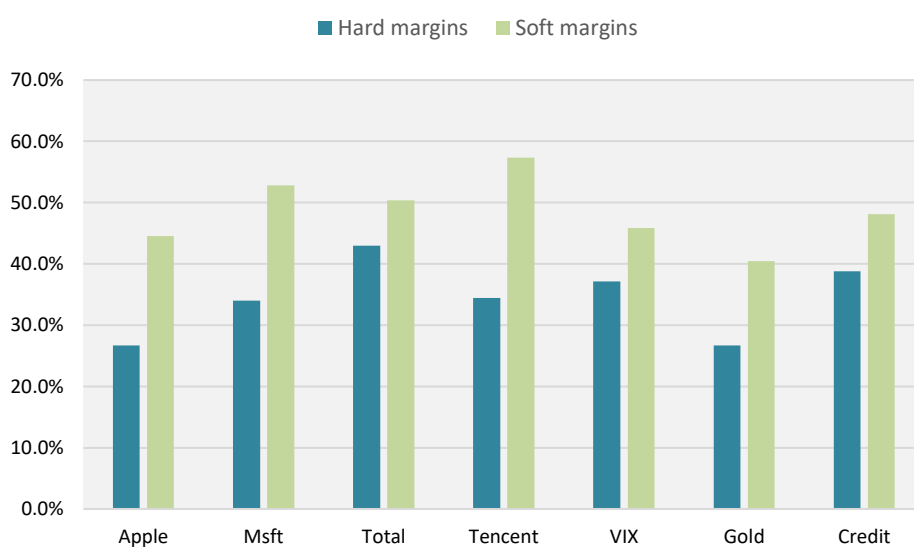
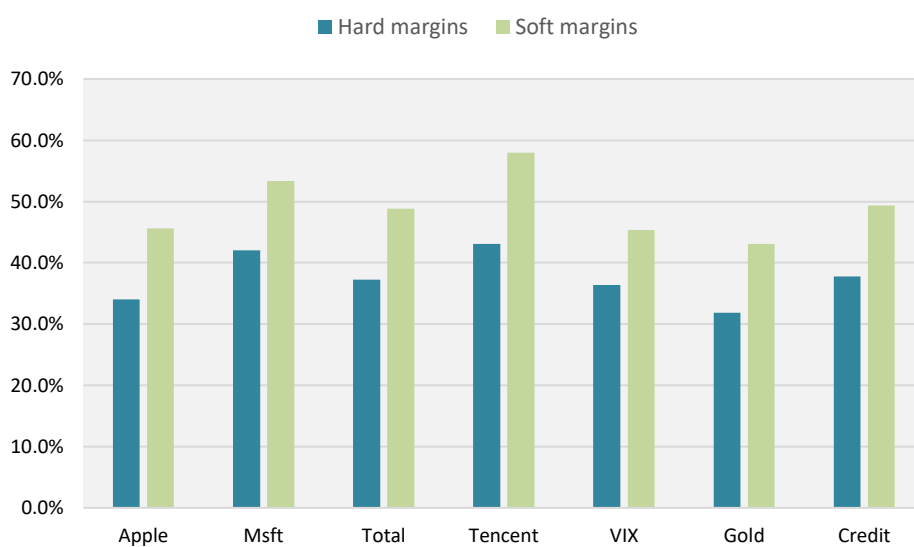


Figure 12 – K10-fold test score, hard and soft margins



3.5 – K-Nearest Neighbors (KNN)

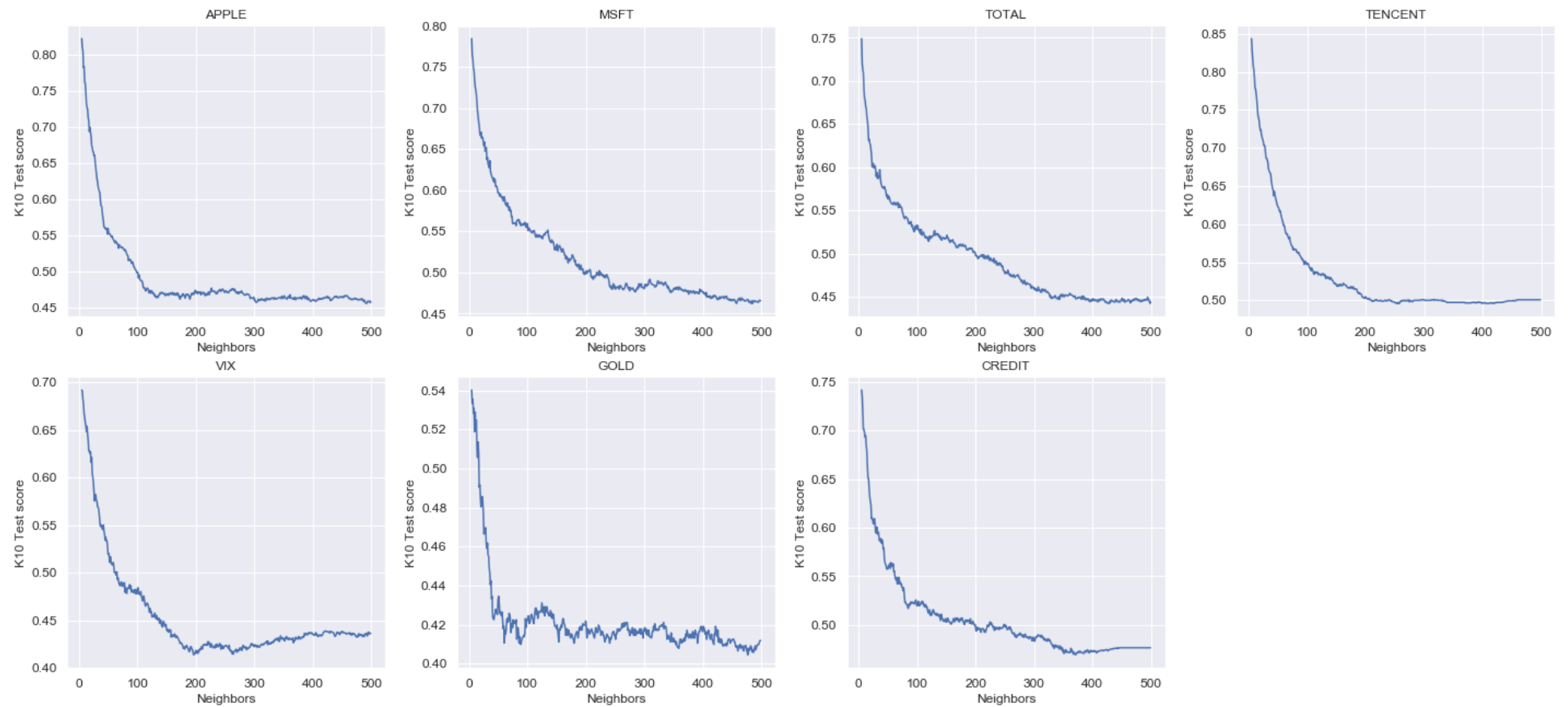
The algorithms tested so far do not seem to predict reliably market returns, with the logistic and SVM classifier yielding at best around a 50/60% prediction accuracy score. That is far from being a satisfactory result. We therefore continue exploring other types of models, such as the K-Nearest Neighbors (used for classification, KNN). KNN is another supervised-learning technique taking as input classified data represented by a vector of features. Once we have such classified features, classifying a new data point becomes trivial. We simply measure distances (according to whatever definition of vector distance is supplied to the algorithm) between the new data point and the nearest K of our already-classified data and thus decide to which class our new data point belongs to by a simple majority voting. In practice, the algorithm has simply two steps: the first is to normalize / scale all the features so that they are of comparable sizes, (we have already done this when we have normalized the entire dataset); after that, we simply compute distances of a new data point from all of the others (which have already been labelled). From those K points the classification with the greatest number of appearances determines the classification of our new data point. In the sklearn library we use the package **neighbors** and in particular the function **KNeighborsClassifier**. This function (used in *KNN.py*) uses the standard Euclidean definition of distance. As for the number of neighbors, this is set initially to 5, as that is the default parameter. We decide to keep 5 as this is the parameter which leads to the highest test score (this can be visually appreciated in the subplots below).

TABLE 15 – K-nearest neighbors, K = 5 (default)

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
Score on training set	91.08%	87.62%	85.69%	89.62%	81.44%	71.06%	85.16%
50/50 Naïve split	74.73%	70.88%	65.75%	78.01%	67.29%	50.67%	67.42%
K10-fold cross split	82.23%	78.44%	74.90%	84.36%	69.20%	54.03%	74.18%

Figure 13 – KNN classifier for different levels of neighbors [5,500]

KNN K10 FOLD CROSS SCORES FOR DIFFERENT NUMBER OF NEIGHBORS



3.6 – Trees, Random Forests and AdaBoost

In this section, we will explore decision trees for classification as well as two extensions which build on the concept of creating ensemble algorithms by averaging several models altogether (Random Forests (RF) and AdaBoost (AB), respectively).

Decision Tree (DT) are a non-parametric supervised learning method which can be used for both classification and regressions. The objective here is to create a model which is able to predict the value of a target variable by constructing the best possible tree structure of questions and answers so that when getting a new item to classify, this can be done quickly and accurately simply by looking at the structure of the tree.

There are two parameters when building out a tree: the order in which features of the data are looked at, and what the conditions (the questions we ask at each node of the tree) are. The key here is to look at the so called “information gain” we have as a result of choosing a particular feature, asking a particular question.

Decision Trees have several advantages: they are simple to understand and to interpret, requires little data preparation and generally perform well. However, one common problem with decision trees, especially when we choose a sizeable depth, is to create over-complex tree structure that are not able to generalize the data well. This can easily lead to over fitting and instability: small variations in the data might result in a completely different tree being generated. As we will see this can be mitigated by introducing ensemble methodologies (random forests, bagging regressor, adaboost). We will see via our cross-validation methodology how the scores of such ensembles outperform simple decision trees. Let's start with the simple decision tree.

In sklearn the function we need is **DecisionTreeClassifier** from the **tree** package. The code of this part of the project is contained in **Trees.py**. This classifier is able to handle both single and multi-class classification. The only parameter we need to specify is **max_depth**. This indicates the maximum depth of the tree. We set this equal to 10.

We now follow the same pipeline used in previous model (**1 – Model fit / 2 – 50/50 test / 3 – K10-fold test**) and we get the following results:

TABLE 16 – Simple Decision Tree training and test scores

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Score on training set</i>	93.35%	90.62%	85.62%	95.15%	80.44%	73.25%	91.55%
<i>50/50 Naïve split</i>	71.01%	69.02%	65.49%	75.27%	59.71%	44.55%	66.76%
<i>K10-fold cross split</i>	78.31%	73.19%	69.61%	76.85%	64.34%	49.50%	66.00%

The scores we see in the above table are substantially better than the results we had so far, reaching almost 80% in the case of Apple on the K-fold test split. However, for almost every asset we observe a 15-20% drop in accuracy when we switch from the training set to the test set (the K-fold yields slightly better results than the naïve split). This is due to the aforementioned issue of over-complexity and over-fitting, when using one single decision tree. The answer to mitigate such concerns resides in **Ensemble Methods**. As already discussed when we introduced the logistic classifier, ensemble methods are useful in combining prediction of several base estimators in order to improve generalizability and robustness over a single estimator, thereby reducing the over-fitting concern. When dealing with ensembles we usually distinguish between bagging and boosting methodologies. In bagging (also called averaging methods – we already explored this in the logistic setting and saw that the K-fold score calculated above a bagging estimator slightly improved) several estimators are built independently and then their predictions are averaged. In boosting techniques, base estimators are built sequentially in order to reduce the bias of the combined estimator.

We will explore whether the above-mentioned methodologies yield better results by calculating their naïve 50/50 and K-fold score. In particular, we will focus on the bagging technique known as Random Forests (RF) and the boosting technique known as AdaBoost (AB).

RF is a meta-estimator that fits a number of decision trees classifiers on various

sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Sample size is always kept constant at the original sample size but with replacement set as true. AB is a meta-estimator that starts by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of the incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. In sklearn we will use the functions **RandomForestClassifier** and **AdaBoostClassifier** respectively. Both are methods are part of the **ensemble** package. The code is in **trees.py**. For both estimators the key parameter we need to insert in the function is **n_estimators**, i.e. the number of base trees we need to feed the algorithm. In our case we choose the value 100, i.e. 100 trees. The reason for choosing 100 comes from the convergence subplots illustrated in Figure 16 where we plot the K10-fold scores of the RF algorithm varying the number of trees from a minimum of 10 to a maximum of 100. We can see from the figure that after 100, the increase in the test score is less obvious with some cases where it actually decreases. Therefore, we choose 100 and go ahead in calculating the usual two test scores.

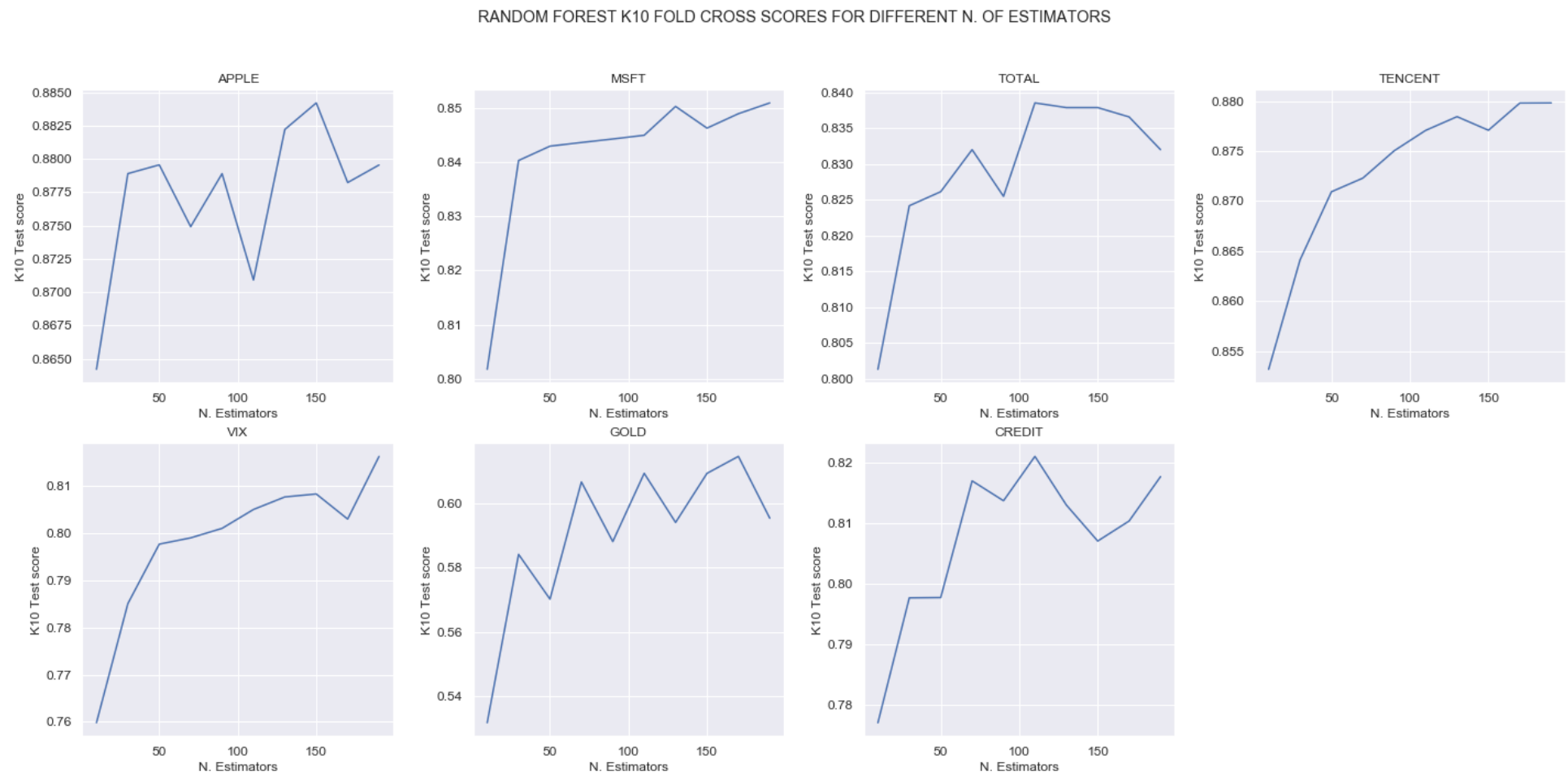
TABLE 17 – Random Forests test scores

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>50/50 Naïve split</i>	83.11%	79.52%%	74.51%	82.79%	70.48%	54.79%	73.40%
<i>K10-fold cross split</i>	85.03%	82.90%	78.56%	84.70%	73.52%	53.82%	77.52%

TABLE 18 – AdaBoost test scores

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>50/50 Naïve split</i>	80.72%	80.98%	77.25%	84.84%	73.27%	54.26%	73.67%
<i>K10-fold cross split</i>	88.22%	84.63%	82.88%	87.58%	80.57%	60.35%	80.64%

Figure 14 – Random Forest K10-fold test score for a range of base trees



As done for previous algorithms we now summarize via bar-charts the information contained in the previous tables so that we get a visual feel for the improvement obtained once we adjust a high variance estimator such as a basic decision tree via the two ensemble methods shown above.

Figure 15 – 50/50 test score for simple tree classifier, RF and AdaBoost

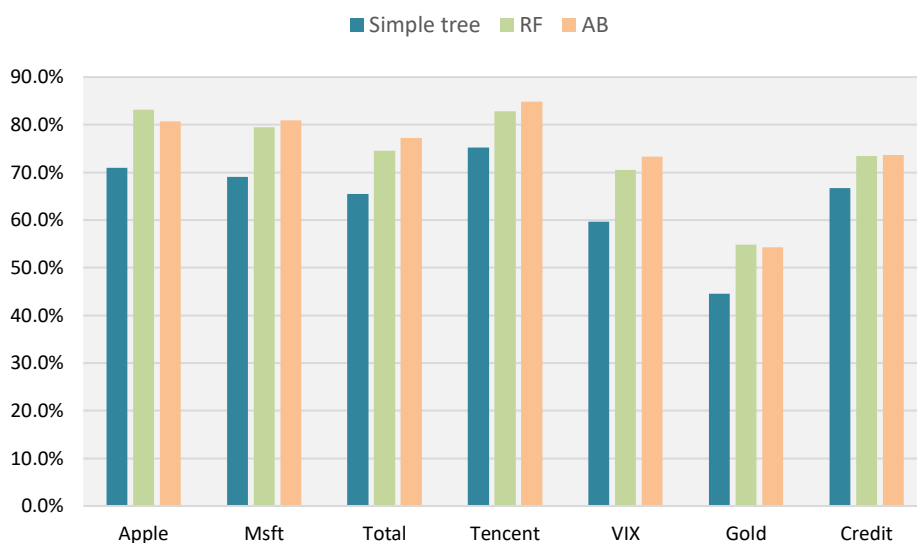
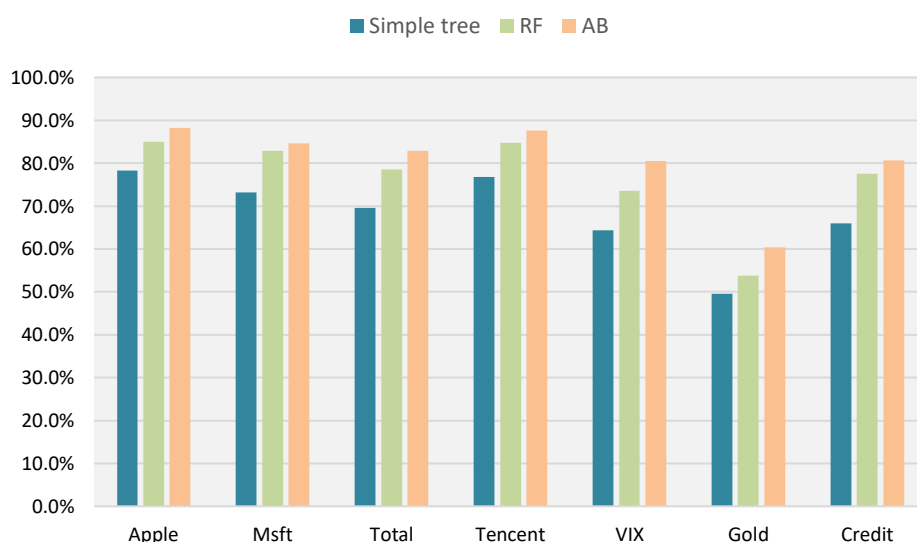


Figure 16 – K10-fold score for simple tree classifier, RF and AdaBoost



The charts confirm a remarkable improvement achieved by the ensemble methods.

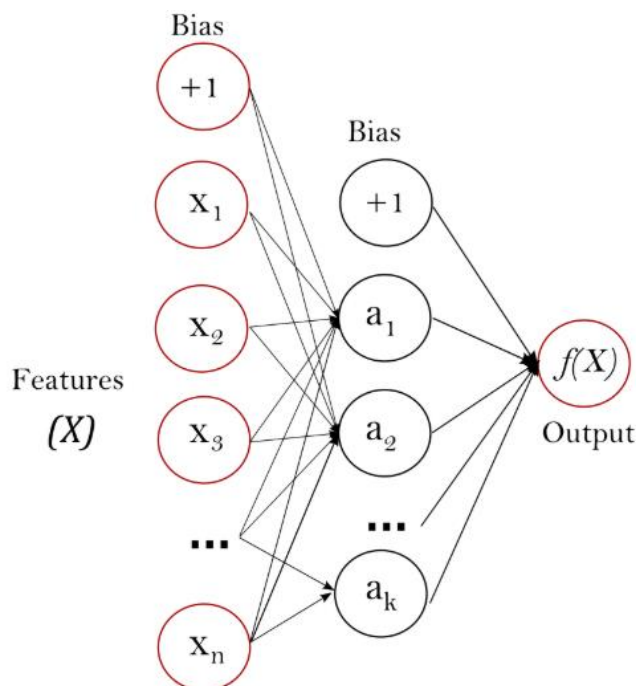
3.7 – Artificial Neural Networks (ANN)

The last statistical learning technique we need to explore before drawing some conclusions is the artificial neural networks technique (ANN). In particular, we will use the Multi-layer Perceptron (MLP) algorithm, as that is the one available in the **sklearn** package we have been using so far.

MLP is a supervised algorithm that learns a function $f(x): \mathbf{R}^m \rightarrow \mathbf{R}^o$ by training on a dataset, where m is the number of dimensions for the input and o is the number of dimension for the output. Such method can learn a non-linear function and use such for either classification or regression purposes.

Below is a representation (from the sklearn website of how such an algorithm works.

Figure 17 – Neural networks graphical representation (from *scikit-learn.org*)



Starting from a set of features (leftmost side of the diagram), each neuron in the layers between the input and the output (so-called hidden layers) “transforms” the

values of the features recursively via a weighted linear summation. Such summation is then fed into a non-linear activation function. Common activation functions used for such purpose include the sigmoid function or the hyperbolic tan function. On balance, ANN have the great advantage of being able to learn non-linear models. However, they suffer from the non-convex issue in the hidden layers, as well as being sensitive to feature scaling.

Here, given we are in a classification scenario, we will be using the sklearn function **MLPClassifier** from the **neural_network** package. We set up our MLPClassifier function by introducing regularization parameter alpha equal to 1, making sure we have a L2 penalty term in the function. We leave the activation function equal to the default value of '**relu**', which is a common choice in ANN applications.

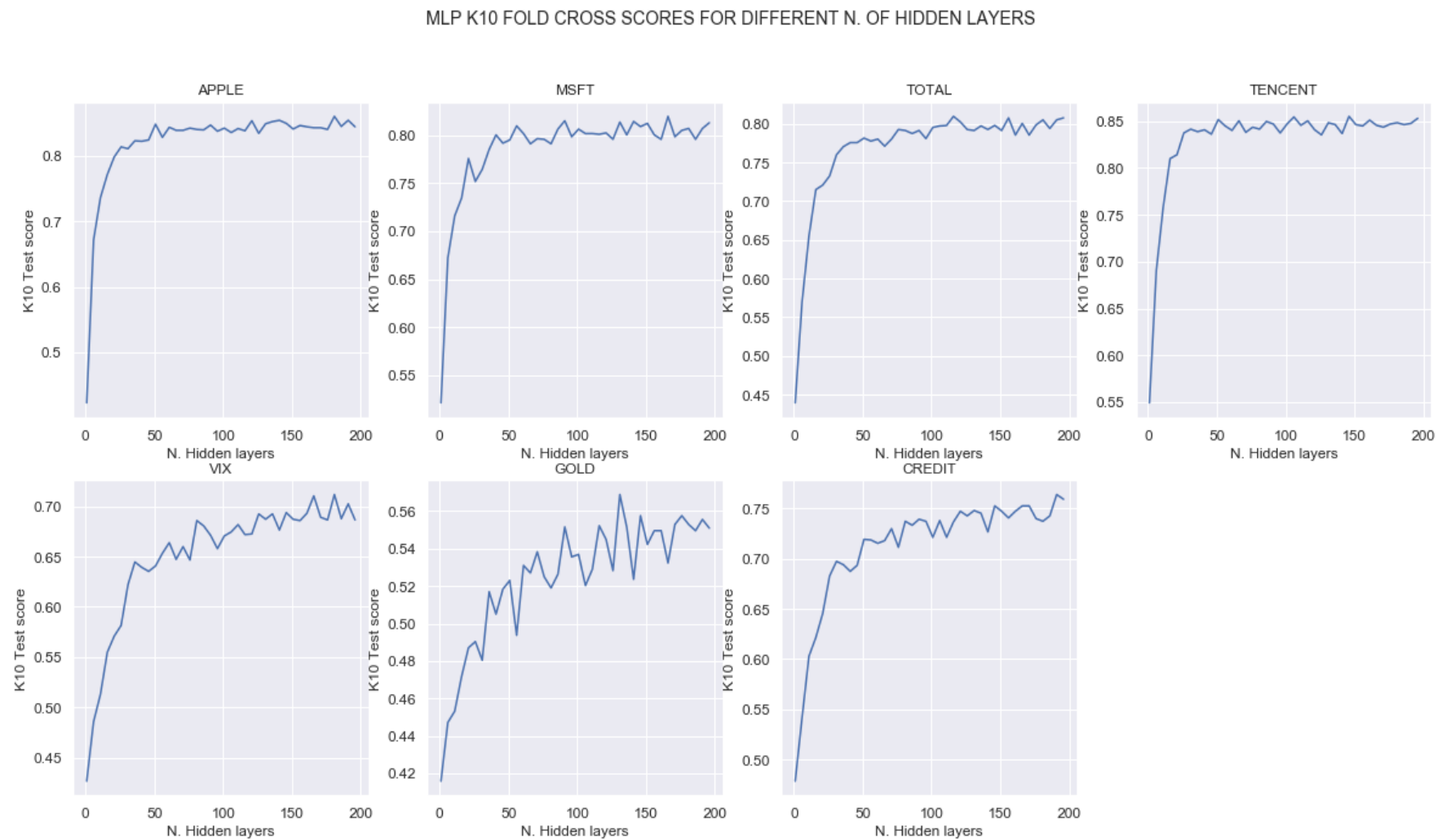
The other key parameter that needs to be set is **hidden_layers_size** i.e. the number of hidden layers in between the input features and the final output. The greater this number, the more complex, flexible (and potentially more prone to over-fitting) the resulting model. Our choice for this parameter is 100, i.e. we leave the default assigned by sklearn. Figure 15 reveals why 100 is a sensible choice in our case as well; indeed, we do not see a tangible improvement in the test scores after this threshold. As done for previous algorithms we follow the usual pipeline (1 – Model fit / 2 – 50/50 test / 3 – K10-fold test) and see what results we obtain.

TABLE 19 – ANN, MLP with 100 hidden layers training and test scores

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Score on training set</i>	95.81%	92.48%	92.55%	96.79%	78.31%	66.27%	85.16%
<i>50/50 Naïve split</i>	80.45%	77.92%	75.42%	82.79%	63.70%	51.46%	67.29%
<i>K10-fold cross split</i>	85.23%	80.78%	79.93%	85.86%	67.60%	53.90%	73.65%

From a first glance at the table results it looks like we have an improvement over logistic regression, KNN, SVM, simple decision trees. However, as we will better analyze in the conclusions, Random Forests and AdaBoost especially, seem to deliver superior results.

Figure 18 – Neural Networks MLP K10-fold test scores for a range of hidden layers



4 – CONCLUSIONS

In this report we have explored several different algorithms in order to test their ability to correctly classify returns of various asset classes, on a 30-day horizon, using a set of predictors as explained in the model sections of this paper. What we find is that progressing from simpler model into more complex, more flexible ones yield better results, underlining the non-linearity dimension which is inherent in financial markets and returns prediction.

However too much complexity and flexibility can also mean significantly loss of prediction accuracy when switching from training to test sets. Cross validation techniques such as the K-fold split gives us an insight into which model we should prefer. If we take this metric as our guide, then the best model across all of the assets seem to be ensemble decision trees: **AdaBoost** seems to be the one consistently delivering the best accuracy scores.

TABLE 20 – K10-fold scores for all algorithms

Score / Asset	<u>Apple</u>	<u>Msft</u>	<u>Total</u>	<u>Tencent</u>	<u>VIX</u>	<u>Gold</u>	<u>Credit</u>
<i>Logistic regression</i>	46.44%	53.82%	48.23%	59.96%	44.44%	42.91%	50.17%
<i>Naïve Bayes</i>	35.26%	46.11%	38.76%	49.93%	39.59%	37.86%	31.99%
<i>SVM</i>	45.57%	53.36%	48.82%	57.99%	45.38%	43.11%	49.31%
<i>K-Nearest Neighbors</i>	82.23%	78.44%	74.90%	84.36%	69.20%	54.03%	74.18%
<i>Simple Trees</i>	78.31%	73.19%	69.61%	76.85%	64.34%	49.50%	66.00%
<i>Random Forests</i>	85.03%	82.90%	78.56%	84.70%	73.52%	53.82%	77.52%
<i>AdaBoost</i>	88.22%	84.63%	82.88%	87.58%	80.57%	60.35%	80.64%
<i>Neural Networks</i>	85.23%	80.78%	79.93%	85.86%	67.60%	53.90%	73.65%

In the table above we have highlighted the best (in green), second best (yellow), and third best (grey) scores, alongside the poorest model (red). AdaBoost consistently outperforms all the other algorithms for all of the assets in our exercise.

BIBLIOGRAPHY

James, G., Witten, D., Hastie, T., Tibshirani, R. (2013). Introduction to Statistical Learning with applications in R. *Springer Texts in Statistics*.

Paul Wilmott., Et Al. (January 2019). Module 4: Data Science and Machine Learning. *Course in Quantitative Finance (CQF) slides*.

Hastie, T., Tibshirani, R., Et Al. (2009). The Elements of Statistical Learning. *Springer Series in Statistics*.

Deisenroth, M.P., Faisal, A. (Draft 2019). Mathematics for Machine Learning. *Cambridge University Press*.

Yhang, D., Zhang, Q. (2000). Drift-Independent Volatility Estimation Based on High, Low, Open and Close Prices. *Journal of Business*, 477.

WEB RESOURCES

Scikit-Learn. <https://scikit-learn.org>

Stanford (Department of Computer Science). <http://deeplearning.stanford.edu/>

Investopedia. <https://www.investopedia.com/>