Faculté d'Économie et de Gestion
Aix-Marseille Université

Research Master's thesis

Machine learning for forecasting stock returns

Aix-Marseille University 2022/2023 Anthony Makarewicz

Acknowledgment

I would like to express my deepest appreciation to all those who provided me the possibility to complete this dissertation.

First of all, I would like to express my sincerest gratitude to my supervisor, Mr. Laurent Sébastien from Aix-Marseille School of Economics, and his assistant, Mr. Hué Sullivan from Aix-Marseille University, who gave me the opportunity to do this dissertation in machine learning.

Moreover, I must express my very profound gratitude to my parents, for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them.

Finally, I am grateful to all my professors in Aix-Marseille University who provided valuable support and assistance.

Table of contents

Introduction

Financial markets represent a complex system in which various factors influence stock returns. These factors can range from macroeconomic indicators to companies' individual characteristics, all of them having an influence on stock returns. Predicting these returns is a very important task for many stakeholders. In fact, for investors and portfolio managers it helps in the portfolio optimization process, namely allocating assets so as to maximize the portfolio's excess return based on a risk tolerance level. Moreover, it can provide insightful information for financial risk management, as if a stock is forecasted to have a negative return, then the investor or portfolio manager can reduce the exposure to that stock. In addition, by building interpretable models that predict those returns, we can extract the model's coefficients associated with each factor to understand which one has the most impact on the returns. Therefore, regulators can impose limits on risk exposures for systemically important financial institutions (SIFI), in order to avoid extreme events that could negatively impact the health of the global financial system.

Nevertheless, predicting stock returns has been a challenging problem in finance for many decades. Fama and French (1994) were the first to propose the famous three-Factor model to explain the cross-sectional variation in stock returns. Carhart (1997) proposed to include a momentum premium to the FF model, and Fama and French (2015) proposed to add a profitability factor (RMW) as well as investment factor (CMA). Harvey, Liu, and Zhu (2015) argued that the proliferation of factors in the literature creates problems for researchers who seek to use them to predict stock returns. For example, with so many factors to choose from, it can be difficult to identify which ones are truly important and which ones are just statistical noise. Moreover, the large number of factors can lead to overfitting, which can result in models that perform well in-sample but poorly out-of-sample.

For that purpose, there has been a growing interest in applying machine learning techniques that enable to handle this huge amount of data efficiently. In fact, machine learning algorithms are designed to learn patterns and relationships in large and complex datasets, which makes them well-suited for analyzing financial data. Furthermore, machine learning has variable selection and dimensionality reduction methods to reduce the number of noisy predictors, thus reducing considerably computational time.

The first type of machine learning models is called parametric and is most of the time highly interpretable, but this often comes at the expense of a low predictive power. On the other hand, non-parametric models are sometimes less interpretable [1] but possess more predictive powers as they adapt their functional form from the data. However, with high flexibility comes great responsibility, as now we can end up with ‘black box” models that we do not understand anymore.

In particular, Gu et al. (2020) proposed a paper entitled *"Empirical asset pricing via machine learning"* that combines traditional asset pricing models with machine learning techniques to improve the accuracy of stock return predictions. The main contribution of Gu et al. (2020) is to use machine learning models to capture the nonlinear relationship between predictors and returns, that may be missed by traditional asset pricing models. They show that machine learning can help to identify new factors that are relevant for predicting stock returns, and that combining these factors with traditional asset pricing factors can improve the performance of asset pricing models.

In this paper, we try to replicate the results achieved by Gu and al. (2020) using a smaller dataset, and we hope at least to obtain results in the same scale. In fact, the R-Squared performance metric used to evaluate the model’s performance does not exceed 0.40%. Therefore, as we are using a smaller version which is approximately 300 times lower, we expect at least to achieve a positive R-Squared. Moreover, we explain in depth each data-preprocessing steps involved to go from the raw data to the filtered datasets so as to obtain unbiased results. In addition, we distinguish between parametric and non-parametric models and dive into the main characteristics of each of them. Then, we detail the methodology used to train models, obtain the prediction’s results, and we finish by presenting the results obtained in our application.

[1] In fact, this is not the case for all of them. For example, the Random Forest model provides feature importance plots that helps understanding the most important factors.

1. Data and problem formulation

1.1. Unvectorized formulation

During this section we present the problem formulation along with the description of the data used. In its most general form, the problem is to predict the equity risk premium for a particular stock at a particular date using a functional form of predictors variables. For that purpose, we dispose of a panel data of 30000 stocks from 1957 to 2021, which extends the time index by 4 years since Gu and al. (2020) used this dataset up to 2016. The general problem formulation for one stock at a particular date can be expressed as follows:

$$r_{t+1,i}{}^{E} = f\left(x_{t,i}\,;\,.\right) + \varepsilon_{t+1,i} \ , \tag{1.1}$$

where

$$\begin{aligned} f :\ & \mathbb{R}^n \longmapsto \mathbb{R} \\ & x_{t,i} \longmapsto f\left(x_{t,i}\right) \end{aligned}$$

$r_{t+1,i}{}^{E} = r_{t+1,i} - r_{t+1,f}$ is the stock i excess return at time $t+1$ over the risk-free interest rate $r_{f,t+1}$ at time $t+1$, $x_{t,i} \in \mathbb{R}^n$ is a vector of predictor variables associated with the stock i at time t, and $\varepsilon_{t+1,i}$ is an unpredictable component of the stock s return i at time $t+1$ with mean $\mu = 0$ and standard deviation $\sigma_\epsilon = 1$

The time frequency is monthly, and we dispose of T observations $t = 1, \dots, T$ in the time-series dimension, and N_t stocks $i = 1, \dots, N_t$ for each t in the cross-sectional dimension. In fact, the number of stocks is different for each month as some companies have gone public since many years, whereas some have just arrived a few years ago. One important thing to notice in (1.1), is the use of the lagged regressors which prevents from the "look ahead bias". As a matter of that, this bias is ubiquitous when working with time series data, as the use of data which is not available at the time the prediction is made could lead to an unrealistic prediction.

Moreover, the functional form $f(\,.\,;\,.)$ is very general in the sense that it can be a parametric form of the predictors, like for instance the Ordinary Least Squares (OLS) model, or it can be non-parametric forms like the ones constructed by tree-based models or neural network models. On the one hand, parametric models take this form $f(\,.\,;\theta)$, where θ is a set of parameters. In contrast, for non-parametric models this set $(\,;\,.)$ can be anything the model chose based on the data but without the econometrician's intervention. For instance, for decision trees and ensemble tree-based models, it can be the name of the regressor along with the value chosen for the split, which is definitely not specified by econometrician. Additionally, this functional form is assumed to be neither dependent on i nor t, as it enables to specify the same model for all stocks across all dates.

Nevertheless, using the same general model for all stocks across all dates may lead to poor out-of-sample estimates for individual stocks. This is why predictions are typically performed at the portfolio-level so as to obtain more robust estimates, which is more aligned with typical investment practices. Furthermore, specifying the same model from 1957 to 2021 may lead to terrible estimates, as financial markets exhibit strong regimen changes. Consequently, neglecting this parameter could be very deleterious for our models. For that purpose, using the same methodology than Gu and al. (2020), we re-estimate each model every year using an expanding rolling window so as to improve the out-of-sample performance.

Lastly, the error term $\varepsilon_{t+1,i}$ of equation (1.1) is related to unpredictable events that appear for stock i at time $t+1$. For instance, it can be natural catastrophes like earthquakes that damage factories of manufacturing companies, or even changes of regulations that force oil companies to reduce their oil production. Nevertheless, these events are most of time completely unpredictable and thus cannot be modeled econometrically. These unpredictable events impact significantly stock returns, which often lead to a very low out-of-sample performance, as the model was not trained to predict those extreme events.

1.2. Data construction

The set of all predictor variables, $x_{t,i}$, is composed of three components, each one is assumed to have a predictive capacity on stock returns.

Firstly, we have individual stock characteristics which are stacked in the vector denoted $c_{t,i}$, which is a $(n_c, 1)$ vector where in this paper $n_c = 94$. These individual characteristics are themselves composed of two types of data. At first, there are accounting data such as: earnings-per-share (eps), dividend-price ratio (dp), and price-earning ratio (per), which are essentially updated at least quarterly and are not very important to explain monthly equity risk premiums. As a matter of fact, as we are using monthly data there are a lot of missing values. Hence, either deleting them or using data imputation technique would still impact negatively our dataset. Nonetheless, according to Gu and al. (2020) for longer time periods such as annual, accounting variables are more relevant and can explain more equity premiums, as it is aligned with fundamental analysis. For that reason, for shorter time periods such as monthly, market data which constitute the second component of our individual stock characteristics, $c_{t,i}$, are the most important predictors. Specifically, they include momentum variables such as: short-term reversal ($mom1$), industry moment ($idmom$); volatility variables such as: return volatility ($retvol$), beta ($beta$) or beta squared ($betasq$); and liquidity variables such as: market volume, bid-ask spread and so on.

Secondly, the next component of our predictor variables set, $x_{t,i}$, is interaction effects of our stock-level characteristics, $c_{t,i}$, with 8 macro-economic variables, m_t , which are presented in Welch and Goyal (2008). For instance, these macroeconomic variables include such as: the default yield spread (dfy), which is the difference between BBB and AAA corporate bonds; FED's treasury bill rates (tbl); and global market variables which are computed based on the S&P 500 index. The interaction effects are constructed mathematically using the Kronecker product, which creates a new matrix where each element of the first matrix is multiplied scalarly with the second one. Formally, if A is an (m, n) and B is a (p, q) matrix, namely

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m-1} & a_{1,m} \\ a_{2,1} & \ddots & . & . & a_{2,m} \\ \vdots & . & \ddots & . & \vdots \\ a_{n-1,1} & . & . & \ddots & a_{n-1,m} \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m-1} & a_{n,m} \end{pmatrix}, \ B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,q-1} & b_{1,q} \\ b_{2,1} & \ddots & . & . & b_{2,q} \\ \vdots & . & \ddots & . & \vdots \\ b_{p-1} & . & . & \ddots & b_{p-1,q} \\ b_{p,1} & b_{p,2} & \cdots & b & b_{p,q} \end{pmatrix}$$

Then: $A \otimes B \equiv \begin{pmatrix} a_{1,1}\, B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \cdots & a_{n,m}B \end{pmatrix}$ gives a (mp, nq) matrix.

Therefore, in our application, as the individual characteristics, $c_{t,i}$, form a $(n_c, 1)$ vector where $n_c = 94$, and that the macro-economic predictors , m_t , form a $(n_m, 1)$ vector, where $n_m = 8$, namely:

$$c_{t,i} = \begin{pmatrix} c_{t,i,1} \\ c_{t,i,2} \\ \vdots \\ c_{t,i,n_c-1} \\ c_{t,i,n_c} \end{pmatrix} = \begin{pmatrix} c_{t,i,1} \\ c_{t,i,2} \\ \vdots \\ c_{t,i,93} \\ c_{t,i,94} \end{pmatrix}, \ m_t = \begin{pmatrix} m_{t,1} \\ m_{t,2} \\ \vdots \\ m_{t,n_m-1} \\ m_{t,n_m} \end{pmatrix} = \begin{pmatrix} m_{t,1} \\ m_{t,2} \\ \vdots \\ m_{t,7} \\ m_{t,8} \end{pmatrix} \quad (1.2)$$

Then $c_{t,i} \otimes m_t$ is a $(n_c n_m, 1)$ vector, so a $(752,\ 1)$ vector, which formally is expressed as:

$$z_{t,i} = c_{t,i} \otimes m_t \equiv \begin{pmatrix} c_{t,i,1}\, m_t \\ c_{t,i,2}\, m_t \\ \vdots \\ c_{t,i,n_c-1}\, m_t \\ c_{t,i,n_c}\, m_t \end{pmatrix} = \begin{pmatrix} c_{t,i,1}\, m_t \\ c_{t,i,2}\, m_t \\ \vdots \\ c_{t,i,93}\, m_t \\ c_{t,i,94}\, m_t \end{pmatrix} \tag{1.3}$$

Finally, the last predictors are industry sector dummy variables which are represented according to the Sector Industrial Classification (SIC) code that encompasses 74 sectors which we denote by s_j. For simplicity, we assume that the SIC code goes from 1 to 74, thus s_j denotes the j-th SIC code[2]. Additionally, we denote $s_{t,i}$, the SIC code of the stock i at time t as the score can change over time for some stocks. Indeed, some companies may expand their activities to conquer new sectors, or merge with companies to diversify their activities. Therefore, considering the SIC code as a variable that changes over time may yield better results. This assumption is particularly important when dealing with missing data, as we can simply perform cross-sectional imputation by replacing missing values across all dates for each stock by the most frequent SIC code observed on non-missing values. For the modelling process, it is better to specify $s_{t,i}$ as a $(n_s, 1)$ vector of dummy variables where $n_s = 74$, so that the j-th element equals to 1 if the stock i at time t has the SIC code s_j. This modelling procedure is called One-Hot Encoding and will be explained further in the data-preprocessing section. Furthermore, we use the indicator variable that we denote $I(\,.\,)$, which equals 1 if the condition is true and 0 otherwise, and is expressed as follows:

$$\mathbb{1}(x)_{\{x \in \mathcal{A}\}} \equiv \begin{cases} 1, & x \in \mathcal{A} \\ 0, & x \notin \mathcal{A} \end{cases}$$

Therefore, in our application we denote the indicator variable associated with the j-th SIC code, s_j, for the stock i at time t, $s_{t,i}$, as:

$$\mathbb{1}\left(s_{t,i}\right)_{\{s_{t,i} = s_j\}} \equiv \begin{cases} 1, & s_{t,i} = s_j \\ 0, & s_{t,i} \neq s_j \end{cases} \tag{1.4}$$

By notation abuse of $s_{t,i}$, we define $s_{t,i}$ as a $(n_s, 1)$ vector of dummy variables:

[2] In fact, the first SIC code is 7 and the last one is 97, so for the purpose of illustration it is better to represent them from 1 to 74.

$$s_{t,i} \equiv \begin{pmatrix} \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_1\}} \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_2\}} \\ \vdots \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_j\}} \\ \vdots \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_{n_s-1}\}} \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_{n_s}\}} \end{pmatrix} = \begin{pmatrix} \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_1\}} \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_2\}} \\ \vdots \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_j\}} \\ \vdots \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_{73}\}} \\ \mathbb{1}(s_{t,i})_{\{s_{t,i} = s_{74}\}} \end{pmatrix} \tag{1.5}$$

For instance, using (3.5), if the SIC code of stock i at time t is j , then:

$$s_{t,i} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

Ultimately, we can re-write our set of predictors denoted $x_{t,i}$ as the row-concatenated vector of $c_{t,i}$, $z_{t,i}$, and $s_{t,i}$. Formally, we write:

$$x_{t,i} = \begin{pmatrix} c_{t,i} \\ z_{t,i} \\ s_{t,i} \end{pmatrix} \equiv \begin{pmatrix} c_{t,i,1} \\ c_{t,i,2} \\ \vdots \\ c_{t,i,n_c} \\ z_{t,i,1} \\ z_{t,i,2} \\ \vdots \\ z_{t,i,n_z} \\ s_{t,i,1} \\ s_{t,i,2} \\ \vdots \\ s_{t,i,n_s} \end{pmatrix} = \begin{pmatrix} c_{t,i,1} \\ c_{t,i,2} \\ \vdots \\ c_{t,i,94} \\ z_{t,i,1} \\ z_{t,i,2} \\ \vdots \\ z_{t,i,752} \\ s_{t,i,1} \\ s_{t,i,2} \\ \vdots \\ s_{t,i,74} \end{pmatrix} \tag{1.6}$$

where $x_{t,i}$ is a $(n_c + n_z + n_s, 1)$ vector where $n_c = 94$, $n_z = 820$, and $n_s = 74$

1.3. Vectorized formulation

Now that we have specified each component of equation (3.1), we can re-write it using matrix notations for all the stocks across all dates. As we are using a panel data of different entities collected at different dates, we choose a multi-level index representation where the first level is the time t, and the second one is for the stock i. We prefer this representation as it provides a clearer view of the data, namely we conserve the temporal order in the time-series dimension, and we can observe the number of stocks available at each date. Specifically, for each time observation t there are N_t stocks, so that time t is replicated N_t times. Therefore, according to equation (3.1) the first-time observation is not observed at time t = 1, but rather at time t = 2 since the regressors are lagged one-period ahead. Formally, we re-write (3.1) as:

$$R^E = f(X\,;\,.) + \mathrm{E} \quad , \tag{1.7}$$

where

$$\begin{aligned} f \,:\, \mathbb{R}^{NT\times n} &\longmapsto \mathbb{R} \\ X &\longmapsto f(X) \end{aligned}$$

Then, we can re-write (3.7) in details to break down each component of (3.7) as follows:

$$\begin{pmatrix} \boldsymbol{r}_2{}^{\boldsymbol{E}} \\ \boldsymbol{r}_3{}^{\boldsymbol{E}} \\ \vdots \\ \boldsymbol{r}_{\boldsymbol{t-1}}{}^{\boldsymbol{E}} \\ \boldsymbol{r}_{\boldsymbol{t}}{}^{\boldsymbol{E}} \\ \boldsymbol{r}_{\boldsymbol{t+1}}{}^{\boldsymbol{E}} \\ \vdots \\ \boldsymbol{r}_{\boldsymbol{T-1}}{}^{\boldsymbol{E}} \\ \boldsymbol{r}_{\boldsymbol{T}}{}^{\boldsymbol{E}} \end{pmatrix} = \begin{pmatrix} f(x_{1,1}{}^T) \\ f(x_{1,2}{}^T) \\ \vdots \\ f(x_{1,N_1-1}{}^T) \\ f(x_{1,N_1}{}^T) \\ f(x_{2,1}{}^T) \\ f(x_{2,2}{}^T) \\ \vdots \\ f(x_{2,N_2-1}{}^T) \\ f(x_{2,N_2}{}^T) \\ \vdots \\ \vdots \\ f(x_{T,1}{}^T) \\ f(x_{T,2}{}^T) \\ \vdots \\ f(x_{T,N_T-1}{}^T) \\ f(x_{T,N_T}{}^T) \end{pmatrix} + \begin{pmatrix} \boldsymbol{\varepsilon_2} \\ \boldsymbol{\varepsilon_3} \\ \vdots \\ \boldsymbol{\varepsilon_{t-1}} \\ \boldsymbol{\varepsilon_t} \\ \boldsymbol{\varepsilon_{t+1}} \\ \vdots \\ \boldsymbol{\varepsilon_{T-1}} \\ \boldsymbol{\varepsilon_T} \end{pmatrix} \tag{1.8}$$

where

$$R^E = \begin{pmatrix} \boldsymbol{r}_2{}^{\boldsymbol{E}} \\ \boldsymbol{r}_3{}^{\boldsymbol{E}} \\ \vdots \\ \boldsymbol{r}_{\boldsymbol{t-1}}{}^{\boldsymbol{E}} \\ \boldsymbol{r}_{\boldsymbol{t}}{}^{\boldsymbol{E}} \\ \boldsymbol{r}_{\boldsymbol{t+1}}{}^{\boldsymbol{E}} \\ \vdots \\ \boldsymbol{r}_{\boldsymbol{T-1}}{}^{\boldsymbol{E}} \\ \boldsymbol{r}_{\boldsymbol{T}}{}^{\boldsymbol{E}} \end{pmatrix}, \quad \boldsymbol{r}_{\boldsymbol{t}}{}^{\boldsymbol{E}} = \begin{pmatrix} r_{t,1}{}^E \\ r_{t,2}{}^E \\ \vdots \\ r_{t,i-1}{}^E \\ r_{t,i}{}^E \\ r_{t,i+1}{}^E \\ \vdots \\ r_{t,N_t-1}{}^E \\ r_{t,N_t}{}^E \end{pmatrix}, \quad \boldsymbol{\varepsilon_t} = \begin{pmatrix} \varepsilon_{t,1} \\ \varepsilon_{t,2} \\ \vdots \\ \varepsilon_{t,i-1} \\ \varepsilon_{t,i} \\ \varepsilon_{t,i+1} \\ \vdots \\ \varepsilon_{t,N_t-1} \\ \varepsilon_{t,N_t} \end{pmatrix} \qquad ,$$

$$X = \begin{pmatrix} x_{1,1}{}^T \\ x_{1,2}{}^T \\ \vdots \\ x_{1,N_1-1}{}^T \\ x_{1,N_1}{}^T \\ x_{2,1}{}^T \\ x_{2,2}{}^T \\ \vdots \\ x_{2,N_2-1}{}^T \\ x_{2,N_2}{}^T \\ \vdots \\ \vdots \\ x_{T,1}{}^T \\ x_{T,2}{}^T \\ \vdots \\ x_{T,N_T-1}{}^T \\ x_{T,N_T}{}^T \end{pmatrix} = \begin{pmatrix} (c_{1,1}{}^T, z_{1,1}{}^T, s_{1,1}{}^T) \\ (c_{1,2}{}^T, z_{1,2}{}^T, s_{1,2}{}^T) \\ \vdots \\ (c_{1,N_1-1}{}^T, z_{1,N_1-1}{}^T, s_{1,N_1-1}{}^T) \\ (c_{1,N_1}{}^T, z_{1,N_1}{}^T, s_{1,N_1}{}^T) \\ (c_{2,1}{}^T, z_{2,1}{}^T, s_{2,1}{}^T) \\ (c_{2,2}{}^T, z_{2,2}{}^T, s_{2,2}{}^T) \\ \vdots \\ (c_{2,N_2-1}{}^T, z_{2,N_2-1}{}^T, s_{2,N_2-1}{}^T) \\ (c_{2,N_2}{}^T, z_{2,N_2}{}^T, s_{2,N_2}{}^T) \\ \vdots \\ \vdots \\ (c_{T,1}{}^T, z_{T,1}{}^T, s_{T,1}{}^T) \\ (c_{T,2}{}^T, z_{T,2}{}^T, s_{T,2}{}^T) \\ \vdots \\ (c_{T,N_T-1}{}^T, z_{T,N_T-1}{}^T, s_{T,N_T-1}{}^T) \\ (c_{T,N_T}{}^T, z_{T,N_T}{}^T, s_{T,N_T}{}^T) \end{pmatrix}$$

Note that as $\boldsymbol{r}_{\boldsymbol{t}}{}^{\boldsymbol{E}}$ is an $(N_t, 1)$ dimensional vector where $t = 2, \dots, T$, which stakes the returns for all stocks available at that time. Therefore, we denote R^E as a $(\sum_{t=2}^{T} N_t, 1)$ vector or simply a $(NT, 1)$ vector[3]. Moreover, we denote X the feature matrix, which has $(n_c + n_z + n_s) = n$ columns with also NT observations. Consequently, the feature matrix forms a $(NT,\ n)$ matrix. In addition, the vector of unpredictable components that we denote E forms a $(NT,\ 1)$ vector. Lastly, the functional form in $f(.)$ remains unchanged as it is neither depend on i nor t, but is applied element-wised to all rows.

[3] We use this notation abuse because it is more convenient to argue that we have NT observations rather than $\sum_{t=2}^{T} N_t$ observations.

2. Data pre-processing

Data-preprocessing is a very important step in machine learning since it conditions all the results we will obtain. Indeed, this concept is often described by the motto “garbage-in garbage-out”, which states if we train our model with low-quality data, we will obtain low quality results accordingly. For that purpose, several data-preprocessing steps such as: handling missing values, encoding categorical variables, standardizing numerical data, must be performed appropriately before fitting any model. During this chapter, we present all data-preprocessing steps involved in our paper along with simplified examples.

2.1. Encoding categorical variables

Encoding refers to replacing nominal categorical variables by a number that can be interpreted by the computer. Categorical variables differ from numerical variables in the sense that they can take only a finite number of values, whereas numerical variables can take an infinite number of values.

The first type of categorical variable is called ordinal and usually does not require any encoding. Indeed, for ordinal numerical variables, each number is correctly interpreted by the algorithm from the lowest to the largest. However, for ordinal variables composed of character strings, we must perform encoding to convert the character strings into numbers. For instance, for character strings it can be the tee-shirt size from S to XL which will be converted into integers from 1 to 4, whereas for ordinal variables it can be the age in years of an individual from 0 to 100.

The second type is called nominal and requires to be encoded accordingly. They can be represented either as character strings or as integers that do not have any logical order. For example, a nominal categorical variable can be all the possible animal names which are represented as character strings, or it can be the SIC code which appears this time as integers.

For that purpose, two encoding methods are generally considered with one which is specific to non-numerical nominal variables like the animals we have just seen. In this case, the encoding method is called nominal encoding and consists in attributing a number from 1 to the number of classes based on the alphabetical order. For instance, if we wanted to encode three possible animals like cat, dog, and bird, here is the obtained column we would obtain:

Table 2.1 : Nominal encoding

Animal	Animal Code
Cat	2
Dog	3
Bird	1
Dog	3
Cat	2
Bird	1
Cat	2
Dog	3

However, this encoding method may sometimes lead to bad results since there is no reason why Bird should be interpreted as lower than a Cat, or a Cat less than a Dog. Therefore, the other encoding method is widely used and is known as One-Hot encoding. This method consists in creating one column for each class and assigning a value of 1 if the value for one particular row belongs to this class. This is the method we use in our paper for the SIC code variable, but to avoid perfect multicollinearity we choose to remove the first class, namely the first SIC code. Here is an example of One-Hot encoding with and without removing the first class for the SIC code:

Table 2.2 : **One-Hot encoding without multicollinearity**

Stock	*SIC*	*SIC 45*	*SIC 5*	*SIC 9*
A	5	0	1	0
B	13	0	0	0
C	45	1	0	0
D	9	0	0	1
E	45	1	0	0

Table 2.3: **One-Hot encoding with multicollinearity**

Stock	*SIC*	*SIC 13*	*SIC 45*	*SIC 5*	*SIC 9*
A	5	0	0	1	0
B	13	1	0	0	0
C	45	0	1	0	0
D	9	0	0	0	1
E	45	0	1	0	0

Here, we note that the 13th SIC code column is not created in table (2.2) as it forms the reference class.

At first glance, we could think that this method is not memory-efficient since for each class we create a column, thus we can end up with a large number of columns. Indeed, in our application the SIC code can take 74 distinct values, thus adding 74 additional columns to the 920 individual characteristics we already have. Nevertheless, as the new matrix formed is a sparse matrix with a lot of zero values, the data is stored internally using the Compressed Sparse Row (CSR) format. The CSR format stores elements of a matrix into three vectors, one which stores the non-zero values, the other for the index of the row where each non-zero value is located, and another vector for the column index. Here is an illustrative example of an original matrix along with its CSR representation:

Table 2.4: One-hot encoded matrix

Row/Col	**0**	**1**	**2**	**3**	**4**
0	0	0	28	0	0
1	13	0	0	0	0
2	0	0	0	9	0
3	0	0	0	0	23
4	5	0	0	0	0

Table 2.5 : Data vector with CSR format

13	5	28	9	23

Table 2.6 :Row index vector with CSR format

1	4	0	2	3

Table 2.7: **Column index vector with CSR format**

0	0	2	3	4

2.2. Handling missing values

Dealing with missing values consists in imputing or deleting data which is not available in our dataset. These missing values can appear in multiple columns for one particular row, or even across all rows for one particular column. The method for dealing with missing values must be chosen carefully and to accomplish two main methods are available.

First of all, we can simply delete all missing values, namely dropping all rows in the dataset where there are missing values in one or multiples columns. To follow the same methodology than Gu and al. (2020), we delete all rows where the excess return columns has missing values. In fact, this is common practice when imputing the dependent variable, especially in predicting equity excess returns, since imputing returns is a very difficult task.

Indeed, this would imply “predicting” the return value for that missing value, which may lead to biases if the return is excessively large. However, one drawback of deleting missing values is that it can sometimes significantly reduce the size of our dataset, even though in our application it is not the case.

For that, a second method called data imputation is sometimes more appropriate to handle missingness in our dataset. This method consists in replacing missing values with one particular value computed based on existing ones. For numerical values, namely all the regressors except the SIC regressor, this value can be a constant value, the mean, the median, or other statistical values.

In our application, by following the methodology of Gu and al. (2020), we perform cross-sectional median imputation for each date across all stocks. The median imputation possesses several advantages for financial data compared to the standard mean imputation. Indeed, as the distribution of financial data is often highly skewed and heavy-tailed, the median is a more accurate statistical measure of the centrality of our data. Specifically, this method can be illustrated with the following illustrative example:

Table 2.8 : Original synthetic dataset before imputation

Date	Stock	X
2023-01-31	A	Na
	B	0
	C	10
2023-02-28	A	Na
	B	2
	C	8
2023-03-31	A	4
	B	Na
	C	Na

Table 2.9: New synthetic dataset after imputation

Date	Stock	X
2023-01-31	A	5
	B	0
	C	10
2023-02-28	A	5
	B	2
	C	8
2023-03-31	A	4
	B	4
	C	4

For the first two dates, we can notice that the missing values in the column X for the stock A is replaced by the median in the column X for the stocks B and C. For the last date, as there is only one non-missing value for the stock A, the missing values for the stocks B and C are simply replaced by the median of four which is nothing less than four.

However, after the cross-sectional median imputation, missing values might still exist. Indeed, if there are missing values across all stocks for one particular date the imputation is no longer possible. To that end, the remaining missing values are replaced by the constant value of zero, which might not be problematic in this case. In fact, a lot of missing values are observed for accounting data which are updated at the minimum quarterly. Therefore, dropping all the observations where missing values are observed in one particular column would lead to a significant reduction of our dataset's size. Moreover, those accounting variables appear to have the least predictive power, so for all models especially non-parametric models like RF, this should not be problematic as they would not be retained by the model.

On the other hand, for categorical values namely the SIC code, we decide to apply imputation techniques too, even though it was not mentioned in the paper of Gu and al. (2020). For that purpose, we replace missing values with the most frequent value across all dates for each stock, which is the most appropriate technique in this context. Indeed, as the SIC code can change over time for one particular company, replacing missing values with the most frequent one seems to be the best option. For instance, this change of SIC code over time can be due to mergers and acquisitions that can change the overall sector in which the companies operate. Moreover, if a missing value is observed across all dates for one particular stock, we choose to create another column called *"Others"* as it could be a hidden information in our dataset that we might not know. Indeed, stocks that do not have any SIC code since their creation might display similar behaviors. For example, if a company operates in a non-traditional industry or an emerging industry that is not adequately covered by existing SIC codes, it might not have an SIC code. This data-preprocessing step described above is often referred to feature extraction which consists in extracting hidden features in our dataset.

2.3. Ranking and standardizing numerical values

Once missing values have been treated correctly, we can cross-sectionally rank all stock characteristics for each time period and map these ranks into the [-1,1] interval. The cross-sectional operation is performed the same way we did in the cross-sectional median imputation step, namely we group all stocks for each date and apply the appropriate transformation.

The ranking is performed by assigning a unique value from one to the number of stocks for each date based on the value of one particular variable in the ascending order. This method is particularly relevant when dealing with financial data which have highly skewed and heavy-tailed distributions. In fact, ranking can help reducing the impact of outliers by scaling each regressor uniformly. Here is a synthetic dataset along with its ranked column for the size variable:

Table 2.10 : Synthetic dataset with rank transformation

Date	Stock	Size	Size Ranked
2023-01-01	A	100	2
	B	200	3
	C	50	1
2023-02-01	A	120	2
	B	220	3
	C	60	1

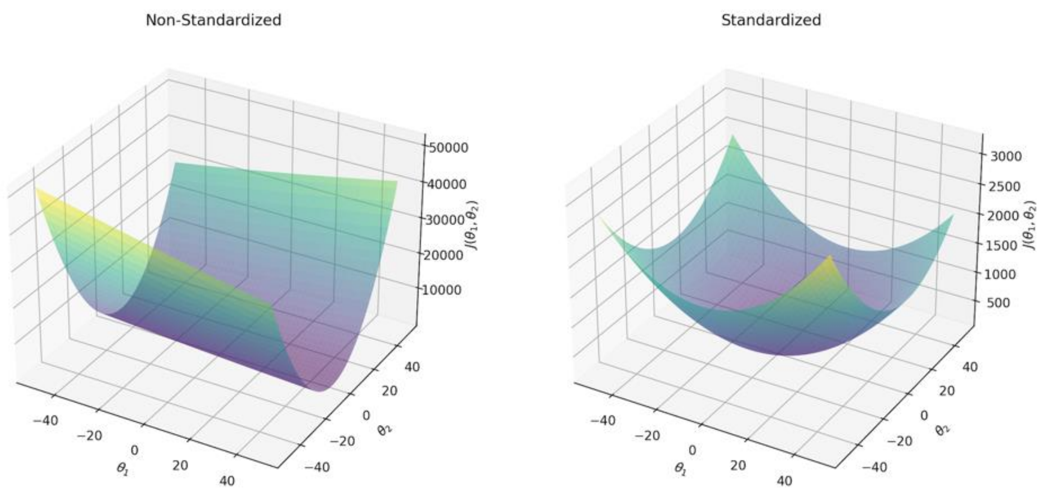
We can note that across each date, the rank for the size column is the same with a value of one for the lowest and a value of three for the highest as there are only three stocks. Once the data has been ranked, we can map the ranks into the [-1; 1] interval.

The normalization is performed so as to ensure that all the regressors have the same scale, where the lowest and highest values are -1 and 1 respectively. The general formula to map from the original variable range to the [a;b] interval can be can be obtained as follows:

$$x_{Norm} = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)} \tag{2.1}$$

Consequently, normalization enables to boost the gradient descent algorithm because it makes learning faster. As a matter of fact, each step descent is sensitive to the parameter value associated with each the regressor. To illustrate this point, let us consider the case where we have two regressors, namely the market capitalization and the beta where their ranges are [1;1000] and [-1;1] respectively. On the one hand, since the market cap takes very large values relative to the beta, the parameter value associated with it will take smaller values so it will have a low range. On the other hand, the parameter associated with the regressor beta will take larger values since the range of the beta variable is smaller than the one of the market-cap. Therefore, the MSE cost function which is function of the parameters will not have its standard bowl shape. We can illustrate this fact with two graphs, where the first one is the non-standardized cost function and the other is the standardized one, which both have a global minimum at (0;0).

Figure 2.1: Cost function with and without normalization



We can notice that in the non-standardized cost function, the cost function is almost perfectly flat in the θ_1 dimension, therefore taking very small steps in the descent. In fact, if we initiate the parameter θ_1 at 40, it will take a lot of steps before reaching the minimum at 0. In contrast, the standardized cost function has a perfect bowl shape which ensures that no matter at which starting point we begin, each step will be significative.[4]

Next, we can extend the table (2.10) by adding the ranked and normalized column as follows:

[4] Even though the closest to the global minimum we initiate our parameters, the fastest the converge will be.

Table 2.11 : Synthetic dataset with rank transformation and normalization

Date	Stock	Size	Size Ranked	Size Ranked Normalized
2023-01-01	A	100	2	0
	B	200	3	1
	C	50	1	-1
2023-02-01	A	120	2	0
	B	220	3	1
	C	60	1	-1

Hence, we can notice that for the last column the lowest, highest, and middle values are -1, 1, and 0 respectively.

2.4. Lagging the regressors

One important feature to consider when dealing with time series data is the temporal order in which the data appear. Indeed, we can only use data which are available at the time the forecast is made, otherwise it would create a look-ahead bias. To avoid potential pitfalls leading to too optimistic results, we generally consider all data from the previous period to predict the dependent variable at the next period, which gives the general formulation as follows:

$$y_t = L(x_t) + \epsilon_t \tag{2.2}$$

where

$$L(x_{t-p}) = x_{t-p},$$

y_t is the dependent variable at time t, x_t is the regressor observed at time t, $L(.)$ is the lag operator where $L(x_t) = x_{t-1}$, and ϵ_t is a noise with mean $\mu = 0$ and standard deviation $\sigma_\epsilon = 1$

In our paper, we not only consider monthly lags, but also quarterly as well as semi-annual lags. The quarterly and annual lags are used because the Securities and Exchange Commission (SEC) in the United-States requires that companies report their financial statements at those particular frequencies. Nevertheless, accounting data that come from financial statements such as: balance sheets, income statements, and cash flow statements, are not instantly available. Indeed, it takes time for companies to close their books at the end of a quarter or a year, reconcile all information, have it audited, and then report it. For instance, a company's Q1 financials may not be publicly available until partway through Q2. Therefore, models that pretend this information is instantaneously available at the close of Q1 would not be realistic. If the data is collected monthly, after adding the quarterly along with the semi-annual lags, the model formula in (2.2) becomes:

$$y_t = L(x_t) + L^4(x_t) + L^6(x_t) + \epsilon_t$$

However, since we have a panel data where each date is replicated by the number of stocks where a return is observed for that period, we must ensure that we apply the lag transformation properly. To that end, we group by all stocks and apply the appropriate lag transformation following the methodology of Gu and al. (2020) on all regressors, except the SIC score which does not require any lagging.

3. Models' specification

Machine learning models can appear into two basic forms, each one having its advantages and drawbacks.

On the one hand, parametric models consider only a fixed number of parameters to estimate the relationship between the dependent variable and the regressors. Nevertheless, parametric models make strong assumptions about the relationship between the dependent variable and the regressors and are not flexible enough to accommodate complex relationships. In fact, this is a major issue in modelling stock returns for parametric models.

On the other hand, non-parametric models do not make any strong if any assumptions about the data and are generally data driven. This type of model enables to model very sophisticated relationships and highly very flexible. Indeed, non-parametric models can adapt their complexity depending on the dataset.

In this paper, we distinguish between parametric and non-parametric models based only on their flexibility and complexity. As a matter of fact, feed forward Neural Networks are generally considered as parametric models since the number of parameters to be estimated is fixed. Though, more advanced architectures like Convolutional Neural Networks and Recurrent Neural Networks can adapt their number of parameters according to the observations and are considered non-parametric in the broad sense. Nonetheless, because of their complexity and high flexibility, we decide to classify feed forward Neural Networks as non-parametric, as we do not want simpler models like Ordinary Least Square (OLS) to be in the same bucket.

In the following section, we will focus on the following parametric models and explain their pros and cons. Specifically, we will consider: Ordinary Least Squares (OLS), Partial Least Squares (PLS) and Principal Component Regression (PCR), Elastic Net (ENet) models, and Generalized Linear Models with Group Lasso (GLM).

Afterwards, we will achieve the same steps than for parametric model but this time for non-parametric models. Specifically, we will study: Random Forest (RF), Gradient Boosting Regression Tress (GBRT), and Ensemble Neural Networks with 1 up to 5 hidden layers that we denote (NN1, NN2, NN3, NN4, and NN5) respectively.

3.1. Parametric models

3.1.1. Ordinary least squares model

The first parametric model is the standard OLS model which is the simplest and the most interpretable one. Following the methodology of Gu and al. (2020), we consider an OLS model with all the regressors that we denote OLS, and another one with the three most important predictors chosen by Lasso regression, namely size, book-to-market, and momentum that we denote OLS-3. As a matter of fact, as most predictors are noisy and therefore unnecessary, the OLS model is sometimes unfeasible. Indeed, when the number of predictors equals or exceeds the number of observations, the OLS model tends to overfit the data, namely it will not be able to generalize well on new data. In addition, we denote the OLS-3's matrix of predictors X^{OLS-3} which is a $(NT, 3)$ matrix. However, we use the standard matrix of predictors X for the OLS model with all the predictors.

Carhart (1997) proposed an extension of the 3-factor model (Fama and French (1994)) by adding a momentum factor. This is similar to the OLS-3 model which is specified in Gu and al. (2020) in the substance but not the form. In fact, they both capture the momentum effect, which it is used to capture the tendency of stocks with high past returns to continue outperforming stocks with low past returns. Nevertheless, the methodology is different since the OLS-3 model chooses its predictors according to Lasso regression, while Carhart uses a fixed set of four predictors. Furthermore, as predictors' importance changes over time due to market cycles, the OLS-3 model can adapt to new data and can possibly capture additional regressors that the Carhart model might not account for. For the OLS model with all the regressors, we can re-write the general matrix formulation in equation (1.7) as follows:

$$R^E = f_{OLS}(X\,;\theta) + \mathrm{E}\,, \tag{3.1}$$

where

$$\begin{aligned} f_{OLS} :\ \mathbb{R}^{NT\times n} &\longmapsto \mathbb{R} \\ X &\longmapsto f_{OLS}(X;\theta) \end{aligned}$$

Next, can also re-write the equation (1.7) for the OLS-3 model using the reduced matrix of predictors X^{OLS-3}:

$$R^E = f_{OLS-3}(X_{OLS-3}\,;\theta) + \mathrm{E}\,, \tag{3.2}$$

where

$$f_{OLS-3} : \mathbb{R}^{NT\times 3} \longmapsto \mathbb{R}$$
$$X_{OLS-3} \longmapsto f_{OLS-3}(X_{OLS-3};\theta)$$

For the OLS-3 model, there exists closed-form solutions for the optimal parameter $\hat{\theta}$, which can be obtained using the normal equations:

$$\hat{\theta} = \left(X^{OLS-3\prime}X^{OLS-3}\right)^{-1}X^{OLS-3\prime}R^E\,, \tag{3.3}$$

where $rank(X^{OLS-3}) = 3$

However, since the full matrix of predictors X is not fully column ranked as there are too many very correlated features, namely when $rank(X) < n$, it is not possible to find an analytical solution for the full OLS model. As a matter of fact, such a matrix is singular and cannot be inverted analytically. Therefore, we must rely on iterative computation techniques such as L-BFGS to find the optimal model's parameters. Nevertheless, in practice $X'X$ can be inverted numerically using pseudo inverse but it is usually better to solve it using an iterative search which is optimized in high-dimensional problem.[5]

In addition, the Huber cost function which is an extension of the MSE to deal with heavy-tailed distributions, does not have any analytical solution.[6] Therefore, we must also find the optimal parameters numerically using an iterative procedure like gradient descent or L-BFGS. The reason is that the Huber cost function is not a quadratic function and does not have a global minimum that can be easily found using linear algebra techniques like in the standard OLS model. Furthermore, as the Huber cost function has three different cases depending on the sign and magnitude of the residuals, the gradient computation requires a conditional evaluation for each residual. This can make the computation of the gradient more time-consuming compared to the standard MSE cost function, where the gradient computation is more straightforward. In facts, for the OLS along with the OLS-3 model, we use the L-BFGS algorithm[7] to find the minima of the Huber cost function following the methodology of Gu and al. (2020) in our replication results.

[5] Indeed, the inversion of $X'X$ which is a $(920\,,920)$ could take a lot of time to be solved.
[6] We will go in depth about this cost function in the methodology chapter.
[7] In fact, L-BFGS is the low memory implementation of the standard BFGS algorithm that bypass the computation of the inversed Hessian matrix-gradient product, which is optimized for large-scale applications.

3.1.2. Elastic Net regression

Even though the OLS model is the most interpretable and the easiest one to fit to the data, it suffers from many caveats. Indeed, the OLS model with all regressors is not satisfactory since it is not full column ranked, which leads to inaccurate estimates of the optimal parameters $\hat{\theta}$. To remedy this issue, we have specified a reduced representation of the feature matrix X that only considers three regressors. Nevertheless, the OLS-3 model does not leverage the huge number of potentially predictive regressors that can provide better results.

To this end, an alternative solution is to let the model discard noisy predictors by reducing the magnitude or assigning a coefficient of zero to these ones. This type of regression is called "penalized" since we penalize the initial cost function with a term which is function of the vector of parameters θ. The purpose behind this penalization term is to encourage sparser representations of the initial set of predictors so as to enable the model to generalize better on new data, namely reducing overfitting.

On the one hand, Lasso regression is often used as a variable selection model, either as a data-preprocessing step and/or a predictive model. Hence, Lasso regression is the traditional OLS model, namely it uses all the predictors, but modifies the cost function to be minimized as follows:

$$\mathcal{J}_{Lasso}(\theta) = \mathcal{J}(\theta) + \lambda \sum_{j=1}^{n} |\theta_j| \,, \tag{3.4}$$

where $\mathcal{J}(.)$ is the baseline cost function (i.e the MSE or the Huber), and $\lambda \in \mathbb{R}^+$is a regularization parameter with higher values for sparser representations and vice versa [8]

On the other hand, Ridge regression is commonly used as a parameter shrinkage method, namely it penalizes large coefficients by reducing their magnitude with an l-2 penalization term as follows:

$$\mathcal{J}_{Ridge}(\theta;\lambda) = \mathcal{J}(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2 \,, \tag{3.5}$$

where $\lambda \in \mathbb{R}^+$ is a regularization parameter with higher values for lower coefficients and vice versa

[8] It includes OLS regression as a particular case when $\lambda = 0$.

Lastly, Elastic Net regression is a hybrid method that combines the properties of both Lasso and Ridge regression. It employs a penalty term which is a linear combination of the l-1 penalty from Lasso regression and the l-2 penalty from Ridge regression. The modified cost function for Elastic Net regression is given as follows:

$$\mathcal{J}_{ENet}(\theta;\lambda,\rho) = \mathcal{J}(\theta) + \lambda\left((1-\rho)\sum_{j=1}^{n}\left|\theta_j\right| + \rho\sum_{j=1}^{n}\theta_j^2\right), \tag{3.6}$$

where $\lambda \in \mathbb{R}^+$ is the regularization parameter that controls the overall sparsity of the model, and $\rho \in [0,1]$ is the mixing parameter that determines the balance between the Lasso and Ridge penalties[9]

Elastic Net is particularly useful when dealing with correlated features as well as noisy features in the dataset. By combining the sparsity-inducing property of the Lasso penalty with the shrinkage property of the Ridge penalty, Elastic Net can effectively balance feature selection and coefficient regularization, leading to better generalization and more accurate predictions on new data. The primary difference between Lasso and Ridge regression lies in the properties of the penalty term and the effect it has on the model's coefficients. Indeed, the properties of the l-1 and l-2 penalization terms lead to different geometric representations of the optimal coefficients.

On the one hand, the l-1 penalty restrains the initial unconstrained space, namely $\Theta_j \in]-\infty;\ +\infty[$ for $j = 1, \dots, n$, into a new constrained space Θ^{Lasso} such as:

$$\exists\, c \in \mathbb{R}^+ \ /\ \Theta^{constr} = \{\Theta :\ \sum_{j=1}^{n}\left|\Theta_j\right| \leq c\}, \tag{3.7}$$

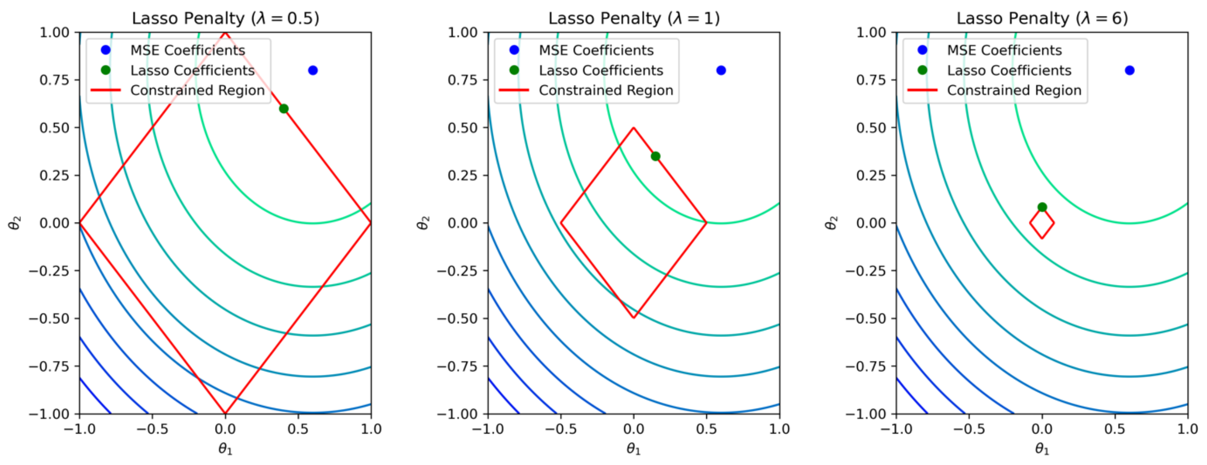
where $|\ .\ |$ is the absolute value function

The value of c is unknown, but we know that it is a positive real number that majors the initial unconstrained space Θ by some value.[10] However, we know that c is controlled by the regularization parameter λ with higher values of lambda means lower values of c and vice-versa. We can plot in the two-dimensional case this constrained space on top of the MSE's contour plot with different regularization strengths to highlight our thoughts:

[9] It includes Lasso regression as a particular case when $\rho = 0$ and Ridge regression when $\rho = 1$.

[10] Indeed, since $|x| \geq 0\ \forall x \in \mathbb{R}$, then if $c \geq |x| \geq 0$ so $c \geq 0$.

Figure 3.1: MSE's contour plot with the constrained region for Lasso regression



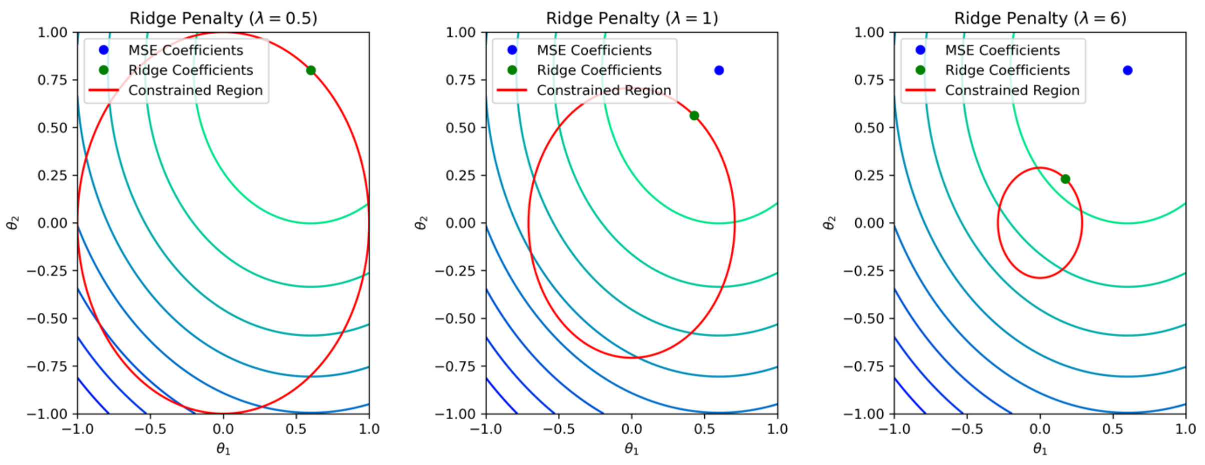
We can visualize a diamond shape in the space of restrained coefficients which is a direct consequence of the absolute value in the l-1 penalty term. The vertices of this diamond lie on the axes, which represent a value of zero for one of the two optimal coefficients. When the Lasso optimization process is solved, the optimal coefficients are found at the point where the contours of the MSE term intersect the L1-norm penalty region. Therefore, when there is enough regularization, there is a high probability that optimal parameter values on the edges that are the closest to the global minimum of the MSE, are located at the vertices of the diamond-shaped constrained region. Nevertheless, it is exactly at those corners that one of the two coefficients is equal to 0, which is why Lasso regression tends to apply a 0 coefficient to noisy regressors. We can observe this phenomenon on the plot when the regularization is high, namely when $\lambda = 6$.

On the other hand, the l-2 norm penalty in Ridge regression also constrains the coefficients, but it does so in a different way compared to Lasso. The l-2 norm penalty term restrains the initial unconstrained space for the optimal coefficients $\Theta_j \in]-\infty; +\infty[$ for $j = 1, \ldots, n$, into a new constrained space Θ^{Ridge}, such that:

$$\exists\, c \in \mathbb{R}^+ \;/\; \Theta^{Ridge} = \{\Theta : \sum_{j=1}^{n} \Theta_j^2 \leq c^2\}, \qquad (3.8)$$

The value of c plays a similar role to the one for Lasso, but here it is represented as the radius of the circle formed by $\sum_{j=1}^{n} \Theta_j^2$. We can also plot the figure (3.1) but this time for ridge regression:

Figure 3.2: MSE's contour plot with the constrained region for Ridge regression



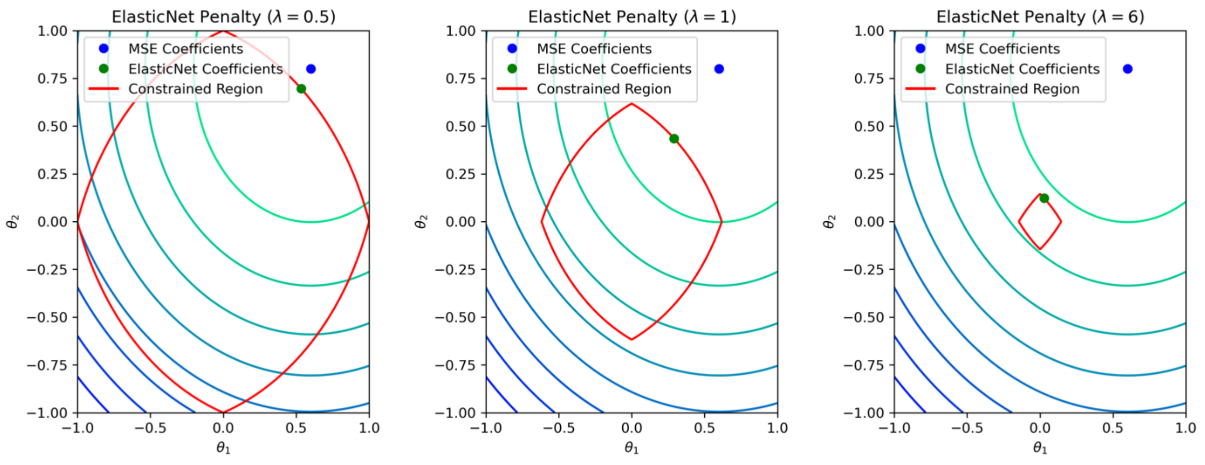
This time, when we visualize the constrained space for the Ridge penalty, we can see that it takes the shape of a circle (or an n-dimensional sphere in the general case) centered at the origin (0; 0). This is a direct consequence of the square in the l-2 norm penalty term. The optimal coefficients are found at the point where the contours of the MSE term intersect the l-2 norm penalty region. Since the circle has no corners or sharp edges, the probability of obtaining sparse coefficients (i.e., coefficients equal to zero) is much lower compared to Lasso. Instead, Ridge regression tends to shrink the coefficients towards zero, while allowing for non-zero values.

Finally, the Elastic Net penalty term creates this new constrained space Θ^{ENet}, such that:

$$\exists\, c \in \mathbb{R} \;/\; \Theta^{ENet} = \{\Theta :\; (1-\rho)\sum_{j=1}^{n}|\Theta_j| + \rho\sum_{j=1}^{n}\Theta_j^2 \leq c\}\,, \tag{3.9}$$

We can also plot the figures (3.1) and (3.2) but this time for the Elastic Net's constrained space:

Figure 3.3: MSE's contour plot with the constrained region for ElasticNet regression



The constrained space for the Elastic Net penalty takes the shape of an ellipse (or an n-dimensional ellipsoid in the general case) also centered at the origin (0; 0). The shape of the constrained space comes from the combination of the l-1 and l-2 penalty terms, which allows both sparsity (like Lasso) and a smooth continuous shrinkage of coefficients (like Ridge).

2.1.3. Principal component regression

Similar to ElasticNet (ENet), principal component regression (PCR) uses a reduced space of the original feature predictors with the same purposes than ENet. However, unlike ENet, PCR performs a dimensionality reduction of the original space of predictors before fitting the model's coefficients. To accomplish this, the PCR algorithm uses a two-step procedure with the first one being the dimensionality reduction technique known as Principal Component Analysis (PCA), and the last one being the traditional linear regression. The purpose behind PCA is to transform the initial feature matrix X into a set of uncorrelated linear combinations of each feature called principal components. These principal components capture most of the variance in the original data, effectively reducing noise and multicollinearity among regressors.

To that end, the PCA algorithm transforms our (NT, n) feature matrix X into a (NT, k) reduced matrix $\mathrm{X}^{\mathrm{reduced}}$ where $k \leq n$. This new matrix $\mathrm{X}^{\mathrm{reduced}}$ is obtained by multiplying the original (NT, n) feature matrix X with the first k columns of the matrix U, which is obtained via Singular Value Decomposition (SVD) of the covariance matrix of X. The covariance matrix which we denote as Σ, is applied to the scaled version of the feature matrix X, that we denote $\mathrm{X}^{\mathrm{scaled}}$. This is because the PCA algorithm is a distance-based algorithm which is sensitive to the scale of our data. Formally, the covariance matrix Σ is expressed as follows:

$$\Sigma = \mathrm{Cov}\left(\mathrm{X}^{\mathrm{scaled}}\right) = \mathbb{E}\ (\ (\mathrm{X}^{\mathrm{scaled}} - \mathbb{E}\left(\mathrm{X}^{\mathrm{scaled}}\right))\ ((\mathrm{X}^{\mathrm{scaled}} - \mathbb{E}\left(\mathrm{X}^{\mathrm{scaled}}\right)'))\ , \quad (3.10)$$

where $\Sigma \in \mathbb{R}^{\mathrm{n}\times\mathrm{n}}$ is a symmetric and positive semi-definite matrix

From this covariance matrix, we can compute its eigenvectors that are stacked in the columns of the matrix U, along with its diagonal matrix of eigenvalues S which are both (n, n) squared matrices. The matrix of eigenvectors, U, contains each principal component in its columns which are all orthogonal to each other. This matrix U informs us about the direction in which there is the most variation in the data. In contrast, the matrix of

eigenvalues, S, is a diagonal matrix where the eigenvalue located in the i-th row and i-th column, is associated with the eigenvector in the i-th column of the matrix U. The eigenvalue is a scale factor that provides the amount of variance explained by each component. Thus, the first principal component has the largest eigenvalue, the second has the second largest one and so on. These matrices can be obtained using SVD or eigenvalue decomposition of Σ.
Formally, we write:

$$\mathrm{X}^{\mathrm{red}} = XU_k \tag{3.11}$$

where

$$U = \begin{pmatrix} u_1^{[1]} & u_1^{[2]} & \dots & u_1^{[k-1]} & u_1^{[k]} & \dots & u_1^{[n-1]} & u_1^{[n]} \\ u_2^{[1]} & u_2^{[2]} & \dots & u_2^{[k-1]} & u_2^{[k]} & \dots & u_2^{[n-1]} & u_2^{[n]} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \dots & \vdots & \vdots \\ u_i^{[1]} & u_i^{[2]} & \dots & u_i^{[k-1]} & u_i^{[k]} & \dots & u_i^{[n-1]} & u_i^{[n]} \\ \vdots & \vdots & \dots & \vdots & \vdots & \ddots & \vdots & \vdots \\ u_{n-1}^{[1]} & u_{n-1}^{[2]} & \dots & u_{n-1}^{[k-1]} & u_{n-1}^{[k]} & \dots & u_{n-1}^{[n-1]} & u_{n-1}^{[n]} \\ u_n^{[1]} & u_n^{[2]} & \dots & u_n^{[k-1]} & u_n^{[k]} & \dots & u_n^{[n-1]} & u_n^{[n]} \end{pmatrix}$$

$U_k \in \mathbb{R}^{n\times k}$ is k-th first columns of the matrix U, and $\mathrm{X}^{\mathrm{red}} \in \mathbb{R}^{NT\times k}$ is the reduced feature matrix

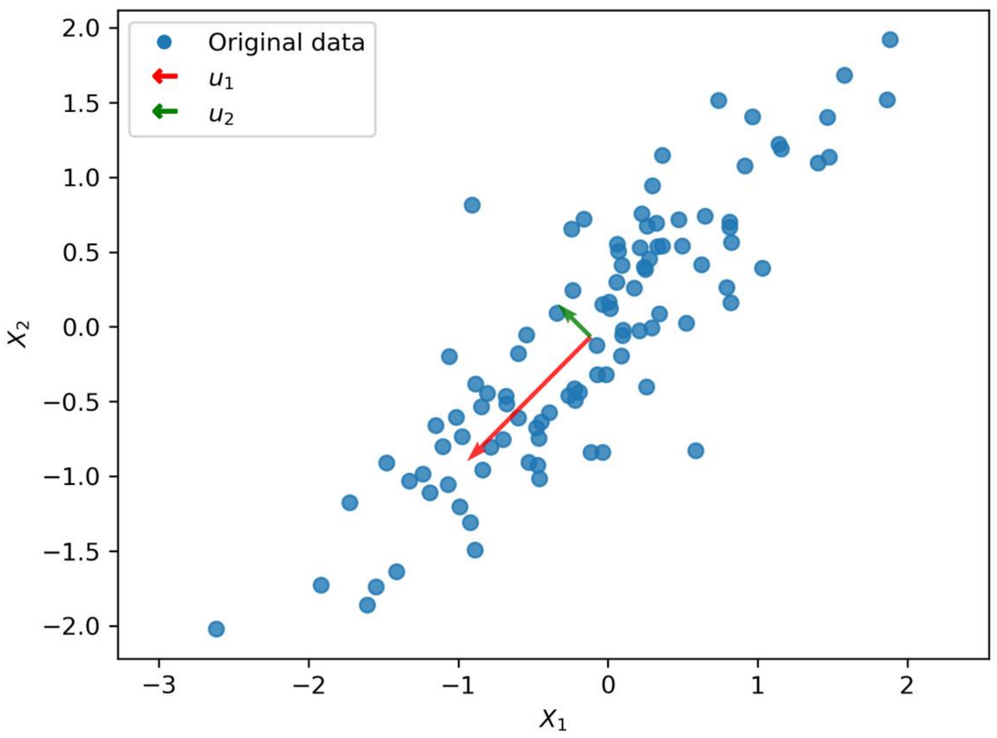
The optimization problem performed by PCA is to construct these principal components by maximizing their total variance. Formally, we write:

$$U = \arg\max_{U} Tr(\mathrm{X}^{\mathrm{red}\prime}\mathrm{X}^{\mathrm{red}}) \tag{3.12}$$

where $\mathrm{X}^{\mathrm{red}\prime}\mathrm{X}^{\mathrm{red}} \in \mathbb{R}^{k\times k}$ is the total variance of the projected data, and $Tr(\,.\,)$ is the trace operator

We can provide a graphical representation using a synthetic dataset so as to highlight the concepts of eigenvectors and eigenvalues:

Figure 3.4 : Principal components superimposed on the original data



In this plot, the original data points are shown as blue dots, and the two principal components, $u_1$and u_2, are represented by red and green arrows respectively.

On the one hand, the red arrow represents the first principal component, u_1, which is the direction in the feature space that captures the most variance in the data. In this case, we can observe that the red arrow aligns with the primary direction in which the data points are spread out. The length of the arrow is proportional to the amount of variance explained by this component, so a longer arrow indicates more variance.

On the other hand, the green arrow represents the second principal component, u_2, which is the direction orthogonal to the first principal component u_1 that captures the maximum remaining variance in the data. The green arrow is shorter than the red arrow, indicating that the variance explained by this component is less than the one of the first component.

The fact that the first principal component u_1 has a larger norm than the second u_2, suggests that the first principal component explains more of the variance in the data. This is a common result when using PCA, as the algorithm seeks to find the directions that successively maximize the explained variance. In other words, the first principal component will always explain the most variance, followed by the second component, and so on.

From the reduced representation of the feature matrix we can simply construct the PCR algorithm as follows:

$$f^{PCR}(X\,;\theta_k) = (XU_k)\theta_k\,, \tag{3.13}$$

where $(XU_k) \in \mathbb{R}^{NT\times k}$, and $\theta_k \in \mathbb{R}^k$

Note that now the number of parameters to be estimated is lower, as now we have k instead of n parameters where in some application k can be very low. Actually, the optimal value of k can be chosen so as to retain at least 95% or 99% of the original variance in our feature matrix X. This can be performed by taking the sum of the first k eigenvalues of the matrix S, and dividing it by the trace of the matrix S (i.e. the sum of all the eigenvalues), so that this ratio exceeds either 95% or 99%. Nevertheless, in our application we choose it from the hyperparameter tuning process.

The model in (3.13) can be fitted using traditional regression techniques, but it will be less costly to train and less prone to overfitting than traditional regression models. However, the major issue with this model is that now the original set of predictors is no more interpretable, which is a major issue from a regulatory point of view in financial institutions. Indeed, we can no longer compute the marginal effects of each regressor to the individual excess return, which may lead to poor risk management practices.

2.1.4. Partial Least Squares regression

Partial Least Squares (PLS) regression is an alternative to PCR and ENet for dealing with multicollinearity and noisy regressor variables. In this section, we aim to provide an intuition about how PLS works without going too much into the mathematical details. Indeed, it would take several pages of pure mathematical proofs which is not the purpose of this paper. Actually, there are several variations of the PLS algorithm such as: PLS1 for univariate regression, PLS2 when there are multiple dependent variables, or more advanced like kernel PLS which is more suited for non-linear regression problems. In this section, we focus on the PLS1 variation which is the simplest version. Additionally, we explain the similarities and differences between PCR and PLS.

Similar to PCR, PLS is a two-step procedure. Indeed, it performs a dimensionality reduction step that creates a reduced set of regressors, which are called the latent variables

T[11], followed by the linear regression one. However, unlike PCR, the dimensionality reduction step is performed by incorporating the dependent variable, namely the individual excess return ${r_{t+1,i}}^E$ at time $t+1$ into the optimization process. As a matter of fact, the primary goal of PLS is to find a linear relationship between the predictor variables and the dependent variable, by creating a reduced set of latent variables that maximizes the covariance between the regressors and the dependent variable. In addition, in contrast to PCR, PLS can handle situations where the number of predictor variables is greater than the number of observations, which is not possible with traditional linear regression and PCR. In the following section, we denote m as the number of obserbations and n as the number of regressors which is consistent with the machine learning convention. The main steps of PLS can be described as follows.

The weights initialization is the first step and is performed only for the first component $T_1 \in \mathbb{R}^m$. This step involves regressing each individual regressor $X_j \in \mathbb{R}^{m\times 1}$ for $j = 1, \dots, n$ on the dependent variable, namely the vector of excess returns, $R^E \in \mathbb{R}^{m\times 1}$, so as to extract each partial coefficients that we denote ϕ_j for $j = 1, \dots, n$. If the feature matrix X is mean-centered and scaled before performing all these steps, the coefficients vector , $\phi \in \mathbb{R}^n$, can be constructed simply as follows:

$$\phi = X' R^E, \tag{3.14}$$

When we said the coefficients vector ϕ is obtained by regressing each regressor on the dependent variables, this is because the partial coefficient ϕ_j obtained from OLS is computed as $\phi_j = \frac{Cov(X_j, R^E)}{Var(X_j)}$. Therefore, and if $X_j$is mean-centered and scaled, namely $\mathbb{E}(X_j) = 0$ and $\mathbb{V}(X_j) = 1$, then we have $Cov(X_j, R^E) = \frac{1}{m-1} X_j' R^E$. However, since $\mathbb{V}(X_j) = 1$, we have $\phi_j = \frac{1}{m-1} X_j' R^E$. Thus, to simplify[12] if we omit this scale factor $\frac{1}{m-1}$, we just write $\phi_j = X_j' R^E \; for \; j = 1, \dots, n.$

The next step is to project the feature matrix X onto the weights vector ϕ to form the first scores T for the first component, which is a linear combination of the predictor variables weighted by their respective coefficients ϕ_j. This step is similar to the first step in PCA,

[11] In the literature they are also known as the scores.

[12] This simplification is possible because the PLS algorithm normalizes the weight vector ϕ during each iteration, so the scaling factor becomes irrelevant.

namely finding the first eigenvector in which there is the maximum variation in the data. In contrast, PCR aims to find the direction in the data that maximizes the covariance between the regressors and the dependent variable. Thus, when the values in the scores vector T are high, it indicates a stronger relationship between the predictors and the dependent variable. The magnitude of the scores can be thought of as a measure of the strength of the relationship. However, it is important to remember that the goal of PLS is not just to maximize these scores, but also to find a balance between explaining the variation in the predictor variables and the dependent variable, in order to build a more accurate and stable prediction model. Formally, this step can be written as follows:

$$T = X\phi \tag{3.15}$$

$$U = R^E c \tag{3.16}$$

where $T \in \mathbb{R}^m$ and $U \in \mathbb{R}^m$ are the scores associated with the regressors and dependent variable respectively, and $c \in \mathbb{R}$ is a scalar.

The purpose of introducing the scalar weight c is to ensure that the dependent variable and the scores are on the same scale during the PLS optimization process. In fact, when the algorithm iterates and updates the weights vector, it tries to maximize the covariance between X and R^E. Consequently, introducing a scalar weight for R^E ensures that the scale of R^Edoes not dominate the computation. In practice the value of c is often set to one, especially when R^E is already mean-centered and scaled. However, introducing this scalar weight provides the flexibility to account for cases where the dependent variable R^E might have a different scale than the predictor variables X.

The next step is to readjust the initial weights ϕ and c by creating standardized and scaled coefficients. Indeed, these previous weights, especially ϕ, gives use the “gross” relationship between X and R^E since the coefficients are obtained using approximations. Even though those initial coefficients are obtained fitting one OLS model for each regressor, they do not extract all the covariance across all regressors since each model was fitted using only one regressor. As a matter of fact, PLS aims to find a combination of these regressors that explains the maximum covariance between the predictor variables and the dependent variable, taking into account the relationships between all the predictor variables simultaneously. For that purpose, we update the new weights, namely ϕ_{new} and c_{new} as follows:

$$\phi_{new} = X'T/\,U'U\,, \tag{3.16}$$

$$c_{new} = R^{E\prime}T/\,T'T\,, \tag{3.17}$$

where $\phi_{new} \in \mathbb{R}^n$ is the new updated weights, $c_{new} \in \mathbb{R}$ is the new scalar weight, $U'U \in \mathbb{R}$ and $T'T \in \mathbb{R}$ are the total variations in U and T respectively

Normalizing the updated weights vector is the next step into the optimization procedure. The purpose of this normalization step is to guarantee that each element in ϕ_{new}, namely $\phi_{new,j}\ for\ j = 1, \ldots, n$, as a unit length while maintaining its direction. By normalizing the weights vector, we ensure that it captures the direction of the relationship between R^E and X without being influenced by the magnitude of the coefficients. This makes the algorithm more stable and less sensitive to changes in the scale of the input variables. Additionally, normalizing the weights vector allows for an easier comparison between the PLS components and a better interpretability of the results. Formally, this step is performed as follows:

$$\phi_{norm} = \frac{\phi_{new}}{||\phi_{new}||}\,, \tag{3.18}$$

where $\phi_{norm} \in \mathbb{R}^n$, and $||\phi_{new}|| \in \mathbb{R}$ is the Euclidean norm of the vector ϕ_{new}

Once the normalized weights have been computed, the deflation step is performed. This step aims to remove the influence of the current PLS component from both the predictor variables and the dependent variable, so that the next PLS component can be extracted. By deflating the data, we ensure that subsequent components capture sources of variation in the data that were not captured by the previous components. Similar to PCA, this step ensures that each PLS component is orthogonalized to each other (i.e. that they are not correlated). Formally, we write:

$$X_{new} = X - t\phi'_{norm}\,, \tag{3.19}$$

$$Y_{new} = Y - tc_{new}\,, \tag{3.20}$$

Finally, steps $(3.14) - (3.20)$ are performed $k - 1$ times where k is chosen ex-ante[13], or until a specific criterion is met.

[13] In most cases just a few components are enough, but usually $k \leq rank(X)$.

2.1.5. Generalized Linear Model with Group Lasso

So far, all the models we have considered are based on the assumption that the regressors and the dependent variable are related in a linear way. In the case of empirical asset pricing, this assumption might be correct for simple cases. Nevertheless, when the relationship becomes more difficult to model, linear models may lead to approximation errors, namely the specified functional form is too far from the true unknown functional form.

For that purpose, we introduce the Generalized Linear Model (GLM), which constitutes a family of models that extend the linear regression to cases where the response variable has a non-normal distribution. However, in the paper of Gu and al. (2020), they may use the term GLM in the broader sense to refer to a more flexible model than the standard linear model. In fact, in the strict sense GLM introduces a link function of the dependent variable, which is not used in the original paper of Gu and al. (2020). Indeed, the model we introduce is just a standard linear model that transforms the initial predictors set with a power spline series of order two.

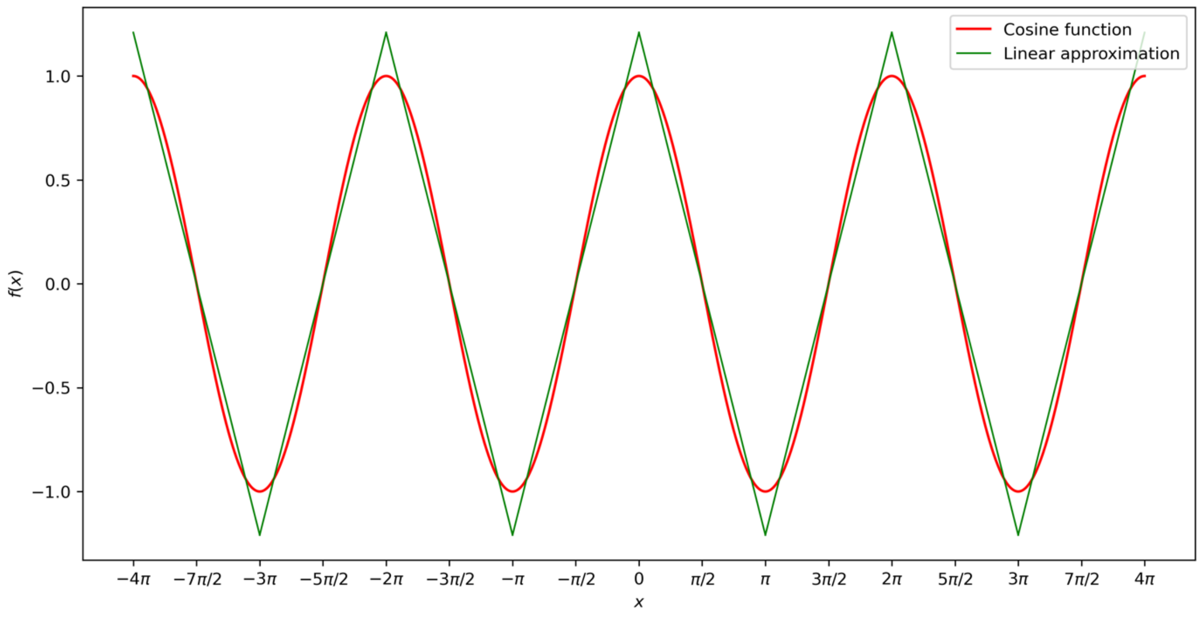
On the other hand, the Group Lasso is an extension of the Lasso method, which is used in the same way than the ENet model we have seen previously. Group Lasso enforces group-wise sparsity, meaning that it selects entire groups of variables rather than individual variables. This is particularly useful when dealing with high-dimensional datasets, where the variables can be naturally grouped or when there is a known structure among them. In the context of our paper, we consider Group Lasso because we use a spline series of order two from our original predictors set X.

To motive the use of GLM, let us try an analogy with Taylor's theorem. Suppose we want to compute the cosine function, but we do not know how to do it. Nevertheless, we know its Taylor's expansion which is expressed as follows:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + \frac{(-1)^n x^{2n}}{2n!} + o(x^{2n+1}), \quad \forall\, x \in \mathbb{R}$$

For that, we could try to compute this infinite linear combination of polynomials $\forall\, x$. However, if we focus on restricted supports of x, for instance if each interval is π-periodic, we can approximate this very complex function using a simple first-order polynomial. Graphically, this can be represented below:

Figure 3.5 : Cosine function superimposed with a first order polynomial approximation on each π interval



This linear approximation provides a reasonable approximation of this complex function, which is by construction an infinite polynomial. This example highlights the motivation behind the use of spline functions that involves in fitting an M-order polynomial on each interval.

In the rest of this section, we assume that we have only one observation to not overload the notation, but we can easily extrapolate to all the observations. Indeed, as we are using a Pool OLS, we consider the same functional form for all stocks at all dates. Therefore, we denote x_j as the value of the regressor j for a particular stock at a particular date, and x a vector of n regressor values.

In spline functions, the intervals are delimited by specific points known as "knots", which are computed by discretizing the range of each regressor into bins of equal length. By using spline functions, we can capture the underlying structure of complex functions with a simpler piecewise polynomial representation. The spline functions used in the original paper of Gu and al. (2020), and therefore the ones that we use take the following form:

$$s(x) = \sum_{j=1}^{n} \left(\sum_{m=1}^{K+2} \theta_{j,m} b_m(x_j) \right) \tag{3.21}$$

where

$$b_m(x_j): \mathbb{R} \longmapsto \mathbb{R}$$

b_m is a basis function for $j = 1, \dots, n$, and $\theta_{j,m} \in \mathbb{R}$ is the coefficient associated with the j-th regressor and the m-th transformation.

Therefore, there are a total of $K + 2$ degrees of freedom, that is the basis functions, that we can adjust according to the data to balance between overfitting and underfitting. The parameter K corresponds to the number of knots that we choose to discretize the range of values of the regressors X. High values of K means higher flexibility but that can come at the cost of overfitting the data. However, in our paper the predictor variables are normalized between -1 and 1, so only a few knots are sufficient.[14]

One important property of spline functions is that when the polynomial degree and the number of knots are fixed, the set of all possible spline functions forms a "vector" space. Here, we use the word "vector" because vector spaces are usually referred as K-vector spaces where K denotes the underlying set of numbers used to represent each individual elements in the vector[15]. For that purpose, as the set formed by all possible spline functions is a vector space, it possesses interesting properties.

Firstly, the closeness to addition and scalar multiplication enables to create any spline function as a linear combination of other spline functions. This is illustrated in equation (3.21) where each spline function b_m is combined linearly with its coefficients θ_m to form a spline function s(x). This is because the basis functions are also spline functions themselves since they constitute a basis of the vector space created from all spline functions.

Secondly, as this is a vector space we can find many equivalent bases of this space, in this case basis functions, to represent any spline function. This means that we can use different basis functions depending on the specific problem at hand, with some of them being better than others in some applications. In our paper we use the following basis functions which are expressed as follows:

$$b_1(x_j) = 1 \, , \tag{3.22}$$

$$b_2(x_j) = x_j \, , \tag{3.23}$$

$$b_{k+2}(x_j) = \left(x_j - c_{j,k}\right)^2 for \; k = 1, \dots, K \, , \tag{3.24}$$

where $c_{j,k}$ $for \; k = 1, \dots, K$ is the knot k associated with the predictor j

[14] In fact, we only use three knots in our application.
[15] For instance, we use say that E is a K-vector space where K can be $\mathbb{N}, \; \mathbb{Z}, \; \mathbb{Q}, \; \mathbb{R}, \; or \; \mathbb{C}$.

Nevertheless, such spline transformations rapidly expand the initial predictors set of length n, which may lead to overfitting. Indeed, for each predictor j we create $K+1$ additional terms[16] (the constant term and the K terms with the knots in (3.22) and (3.24)), which leads to a new matrix of regressors X_{spline} which has now $n(K+2)$ columns. This is why Group Lasso is used to select either all the $K+2$ terms associated with each predictor variable or none of them. Therefore, the modified cost functions follow the same structure than the one for ENet, namely penalizing the standard MSE or Huber cost function with the additional penalty term as follows:

$$\mathcal{J}_{GroupLasso}(\theta; \lambda, K) = \mathcal{J}(\theta) + \lambda \sum_{j=1}^{n} \left(\sum_{m=1}^{K+2} \theta_{j,m}^2 \right)^{1/2}, \tag{3.25}$$

$$\mathcal{J}^H{}_{GroupLasso}(\theta; \lambda, K) = \mathcal{J}^H(\theta) + \lambda \sum_{j=1}^{n} \left(\sum_{m=1}^{K+2} \theta_{j,m}^2 \right)^{1/2}, \tag{3.26}$$

3.2. Non-parametric models

3.2.1. Decision tree models

Thus far, we have focused on parametric models that consider a fixed number of parameters that are specified ex-ante by the econometrician. Nevertheless, by using such models we are making strong assumptions about the functional formal in which each predictor is related to the dependent variable. Indeed, for linear models we assumed that the relationship between the regressors and the dependent variable can be represented as a linear combination of the predictors. To that end, we used a more flexible approach by considering GLM with Group Lasso in which the interpretability is still conserved. However, with GLM we made the assumption that the true relationship can be modeled using spline series of order two. Furthermore, GLM does not consider any variable interactions between our predictors, and if we were attempted to include every one of them, we would end up with thousands of variables. Thus, GLM would definitely overfit the data and would be too computationally expensive to train.

[16] For the purpose of illustration, we include a constant term for each predictor, but in practice we would add only one constant term since it would be replicated n times.

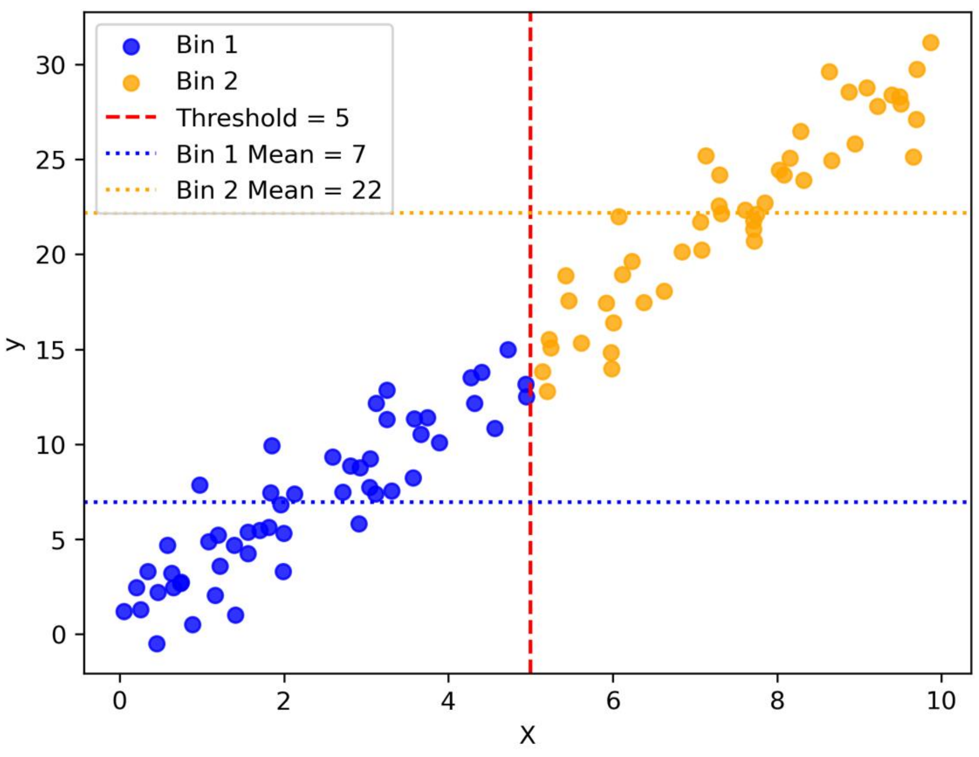
This is why we consider decision tree models, which are non-parametric models that adapt their functional form to the data. By that, we mean that it is the model itself that chooses which predictor is the most important to consider so as to predict the dependent variable. Therefore, it can reduce the computational time and the complexity of the model. Decision tree models operate by partitioning the data into homogenous buckets called "nodes", in which all the observations have similar characteristics (in terms of regressors values).

Let us motivate the use of decision tree models by making an analogy with the famous Fama-French model. This model uses the market capitalization as a regressor to distinguish between small-caps and large-caps. Indeed, the FF-model aims to exploit the small-firm effect, which is a stock anomaly that goes against the Efficient Market Hypothesis (EMH). The small-firm effect states that small-caps tend to have higher abnormal returns than large-caps. Hence, we can use the market capitalization to predict the return for a particular stock based on its size. To that end, the FF-model uses a regressor called Small Minus Big (SMB), which is the difference in excess return between a portfolio composed of stocks with low market-caps and a portfolio composed of stocks with high-market caps. This factor is constructed to reflect the risk associated with size, as smaller companies are often riskier investments compared to larger ones. The same reasoning can be made with the High Minus Low (HML) factor, which is the difference between high and low book-to-market values. Here is the FF 3-factor model:

$$\mathbb{E}(R_i) = R_f + \beta_{1,i}\big(\mathbb{E}(R_m) - R_f\big) + \beta_{2,i}\, SMB + \beta_{3,i} HML\,, \tag{3.27}$$

However, the main difference between the FF-model model and a decision tree model is that a decision tree model is more conservative, in the sense that it chooses a particular market cap value to create two buckets. This threshold value helps to determine in which bucket each stock belongs to. Afterwards, each stock's market cap value is assigned to its corresponding bucket, and the predicted return value for that stock is the average return value for all existing stocks in that bucket. Graphically, we can illustrate this point using a synthetic dataset as follows:

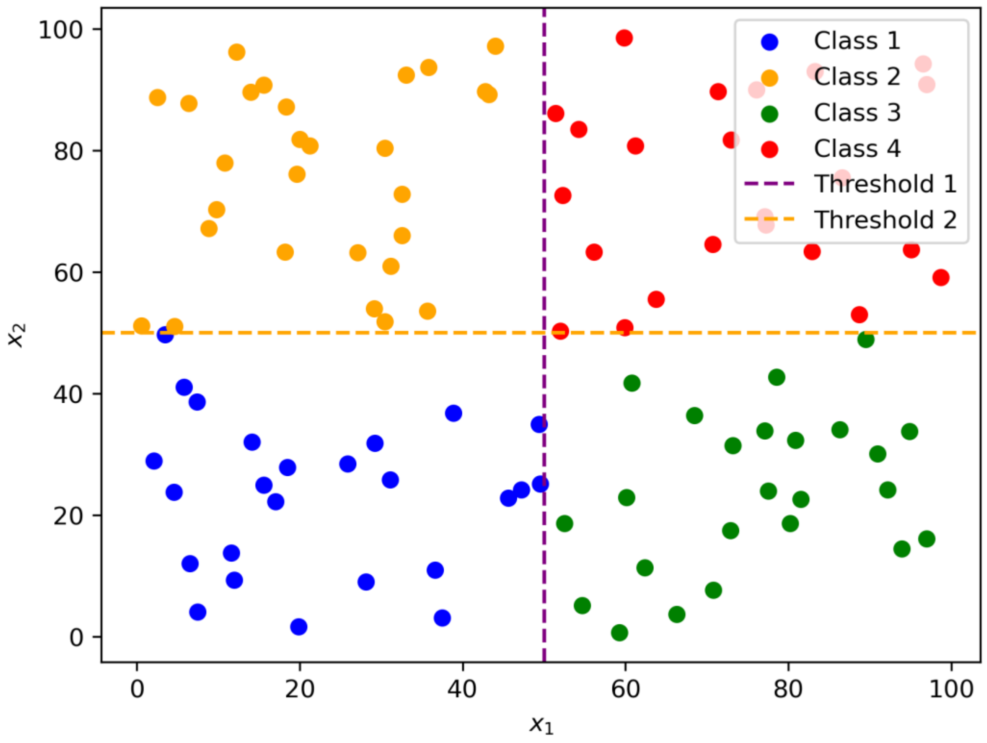
Figure 3.6: **Data partitioning for a regression tree model with 1 regressor**



We can notice that all the observations that have a value less than 5 for the regressor x the predicted value for y is 7, whereas for a regressor value larger than 5 the predicted value is 22.

Nevertheless, the decision tree model that we have just seen only considers one split for one particular regressor. Therefore, it might not be powerful enough to capture more complex relationships. To this end, by repeating the previous step multiple times for each bucket, we can thus model more complex relationships. Indeed, maybe discriminating between small-caps and large-caps might not be enough, so we could let the model create a bucket for middle-caps or even one for very large-caps. In addition, for each partition of data we could even consider other regressors to create the splits like the momentum regressor in the Carhart model. For instance, this is the partitions we obtain when we consider two regressors for a 4-class classification problem:

Figure 3.7: Data partitioning for a classification tree model with 2 regressors



In this 4-class classification problem, we can note that each partition of data is perfectly homogeneous and that both x1 and x2 are considered for partitioning the data into each group.

Now that we have enough intuition about how a tree works, we can define what a tree actually is. A tree is a sequence of if else statements that tries to best discriminate between groups of similar observations. One particular if-else statement is called a node in which one regressor along with a threshold value is chosen by the algorithm so as to create the most homogenous groups as possible. The first node is called the root of the tree, which specifies where the tree begins to grow. Afterwards, the child nodes who inherit from the parent node themselves partition the data into sub-groups. To accomplish this, the algorithm chooses among all possible regressors along with threshold value for that regressor, the ones that minimize the cost function at each node i. The cost function is similar to the previous ones we have seen, but they adapt to the specificity of decision tree models. Formally, the cost function associated with the i-th node, for the feature f and a threshold value of t is expressed as follows:

$$\mathcal{J}_i(f_j, t_j) = \frac{N_{left}}{N_i}\mathcal{J}_{left}(f_j, t_j) + \frac{N_{right}}{N_i}\mathcal{J}_{right}(f_j, t_j)\,, \tag{3.28}$$

where N_i is the total number of observations at node i before the splitting, N_{left} and N_{right} are the number of observations on the left and right child nodes respectively, and $\mathcal{J}_{left}(.)$ and $\mathcal{J}_{right}(.)$ are the cost functions associated with the left and right child nodes respectively

We can note that in the context of decision tree models, the cost function is no longer a function of a set of parameters θ. In contrast, it depends on the chosen feature (or regressor) f_j and a specific threshold value t_j. Therefore, the optimization problem can be expressed as follows:

$$\widehat{f_j}, \widehat{t_j} = \arg\min_{\substack{\widehat{f_j} \in F \\ \widehat{t_j} \in T_j}} \mathcal{J}_i(f_j, t_j) \; for \; i = 1, \dots, n_{nodes} \; , \tag{3.29}$$

where $F = \{X_1, X_2, \dots, X_n\}$ is the set of all n features, $T_j \in \sup(X_j)$ belongs to the support of the feature X_j , and n_{nodes} is the total number of non-leaf nodes (i.e, nodes that can be split further) in the tree

The total number of splits in a tree is called the maximum depth, which determines the complexity of the tree. A higher max depth can lead to more complex trees, but can rapidly lead to overfitting. Indeed, trees can grow infinitely so as to predict each observation in the training set, namely creating a group for each observation. Such trees are called unpruned trees, and this is why in the literature it is said that trees are very sensitive to overfitting. In fact, the default setting in machine learning libraries sets the *max_depth* argument to *None,* meaning that all possible splits will be considered. For that reason, the max depth argument must be tuned appropriately to balance between overfitting and underfitting. This can be achieved after the fitting process, in this case we call it post-pruning, or before and in this case, we call it pre-pruning. Once a tree has finished growing, namely creating new splits, it makes predictions in each group as the average value of all observations (the returns in our case) seen in the training set. Such particular nodes are called the "leaves" of the tree, and the final prediction can be expressed as follows:

$$f_{Tree}(X; \Theta) = \sum_{i=1}^{N_{leaves}} w_i \mathbb{1}(X \in Leaf_i) \tag{3.30}$$

where N_{leaves} is the total number of terminal nodes, w_i the predicted value (e.g., mean value) for the samples in the leaf node i, Θ represents the regressors along with their threshold values chosen for each split, $\mathbb{1}(\,.\,)$ is the indicator function that gives 1 if the predictor values belong in the i-th leave and 0 else, and $Leaf_i$ is a set of predictors values associated with the i-th node

Nevertheless, by tuning the trees too much, it might not be sufficient to capture very complex relationships, this is why ensemble models are used. Ensemble method is a machine learning technique that involves combining the predictions of models to create more powerful predictions. It operates either by majority vote for classification problems, or by averaging the predictions for regression problems. This method is based on the philosophical principle known as the Wisdom of the Crowd which was introduced in Plato's book "*The Republic*". This principle states that as long as a crowd is sufficiently large, diverse and experienced, they are generally righter than a single individual.

3.2.1.1. Random Forest

One popular ensemble method is known as bagging which stands for bootstrap aggregation. This ensemble method consists in fitting "strong learners" (i.e. complex models[17]) in parallel on a bootstrap sample of the original dataset. Thereafter, bagging aggregates the predictions of each model in order to reduce overfitting.

Random Forest is a popular bagging method, which in addition to the bootstrap procedure in bagging, adds another layer of noise in each model by sub-setting a random sample of features for each split[18]. Indeed, when there are only a few relevant features, which is the case in stock returns prediction, each decision tree tends to choose the same features in each split, leading to similar predictions. Therefore, by choosing only a random set of features for each split, we ensure that the diversity principle is satisfied. Furthermore, by choosing a large number of trees[19] we also ensure that the number principle is satisfied. Consequently, the prediction for the Random Forest model can be expressed as follows:

$$f_{RandomForest}(X;\Theta) = \frac{\sum_{i=1}^{N_{trees}} f_{tree,i}(X;\Theta)}{N_{trees}} \tag{3.31}$$

[17] In the context of ensemble based-tree models, this refers for instance to decision trees having a high maximum depth (typically higher than 2).

[18] This is an hyperparameter that can be tuned from the dev set accordingly.

[19] In most machine learning packages, the default value for the number of trees is 100, but in our paper, we consider a fixed number of 300.

where N_{trees} is the total number of trees in the forest, and $f_{tree,i}(.)$ is the prediction of the i-th tree

We can note that the prediction for Random Forest is simply the average predictions between all decision tree models.

3.2.1.2. Gradient Boosting Decision Trees

Another popular ensemble method is known as boosting, in which a set of "weak learners"[20] is combined sequentially where each model tries to correct the "mistakes" of the previous one. Gradient boosting is a popular boosting technique that performs very well when the base learner is a decision tree model. Indeed, as the main issue with decision tree models is overfitting, gradient boosting provides many hyperparameters that can help reducing overfitting without leading to underfitting, namely creating too simple models. Gradient Boosting Decision Trees (GBDT) are very popular in the machine learning community which has come up with optimized libraries like *XGBoost*, *LightGBM*, and *CatBoost*. In fact, these libraries are designed to handle large and complex datasets, and they possess parallel computing along with cluster computing techniques that can speed up significantly the training process.

The term "gradient" in gradient boosting arises from the fact that the algorithm uses gradient descent to minimize the cost function. In each step, it tries to move in the direction of the steepest descent in the cost function's space to find the best new model to add to the ensemble. Formally, the functional form associated with the gradient boosting algorithm can be formulated as:

$$f_{GradientBoosting}(X;\Theta) = \sum_{i=1}^{N_{trees}} \alpha_i f_{tree,i}(X;\Theta) \tag{3.32}$$

where α_i is the contribution of i-th tree to the final prediction

[20] In this case, a tree where the maximum depth can be either 1 or 2.

2.2.2. Neural Networks

Neural networks are state-of-the art non-parametric models that have re-emerged in recent years. Indeed, thanks to Big data, increases in computational powers such as: Central Processing Units (CPUs) and Graphical Processing Units (GPUs), along with innovations brought by the academic research, neural networks appear to be the best machine learning model. This type of model if often described as mimicking the human brain, even though this is not true as the human brain is much more complex. However, this analogy can still be made to emphasize some of its principles. In fact, the neural network takes information from the previous layer of neurons, in analogy to a human neuron that takes information from dendrites, performs some computation with it, and transmit this information to other neurons via synaptic connections. This type of model is nowadays ubiquitous in the machine learning literature and has led to a new area of machine learning called deep learning. These models appear to be the best kind of predictive model available, but only when the amount of data is very large (typically millions of observations), which explain their re-emergence in recent years.

However, these models suffer from low interpretability and for that reason are often qualified as “black boxes”. Indeed, we can no longer use the sign of the coefficients or their intensity like we can do in OLS, to understand how the predictor variables are associated with the dependent variable. This is a big issue in financial risk management, where we want to assess how a small change of one variable can affect our pay-off. Moreover, from a regulation standpoint, the ACPR in 2020 published a discussion paper entitled “Governance of Artificial intelligence in finance” that raises the alarm of the possible deviations of “black box” models. Nevertheless, at the same time we want predictive models, thus a trade-off appears between interpretability and predictability.

A neural network consists of layers of neurons, where each neuron known as hidden unit, computes a linear function of the inputs from the previous layer, and then apply a non-linear function to this linear transformation. The non-linear activation function is necessary since otherwise the output layer would consists of a linear combination of the original data, which would give the same results than a simple linear model. The choice of the activation function sometimes depends on the specificity of the problem. For instance, the Long Short-Term Memory (LSTM) model which possesses a specific neural network architecture known as Recurrent Neural Network (RNN), requires the sigmoid as well as the tangent hyperbolic (tanh) activation functions. In contrast, for feed forward Neural Networks the Rectified Linear unit (ReLu) activation function is always preferred over the two previous ones. As a matter of fact, one major improvement in fitting neural networks was switching from the sigmoid activation function to the ReLu activation. This is because the issue with the sigmoid is that the gradients become very low when the antecedents of the sigmoid function are very large (in absolute value term). Consequently, as the gradient descent algorithm aim to minimize the cost function, taking larger steps in the descent enables the algorithm to converge faster. In contrast, the ReLu activation function has a slope of one when the antecedent is positive and zero otherwise. In addition, the ReLu activation function encourages sparsity among neurons since it “deletes” non-predicative neurons, namely it assigns a value of zero to the negative linear parts of the neuron since they have non-predictive capacity. On the other hand, the tanh activation function possesses the advantage to be centered at zero, has larger gradients than the sigmoid activation function but still suffers from the same problem. Here are the formulas of those three activation functions along with a plot for each of them:

$$g(z)_{Sigmoid} = \frac{1}{1 + e^{-z}} \tag{3.33}$$

$$g(z)_{ReLu} = \max(0, z) \tag{3.34}$$

$$g(z)_{Tanh} = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{3.35}$$

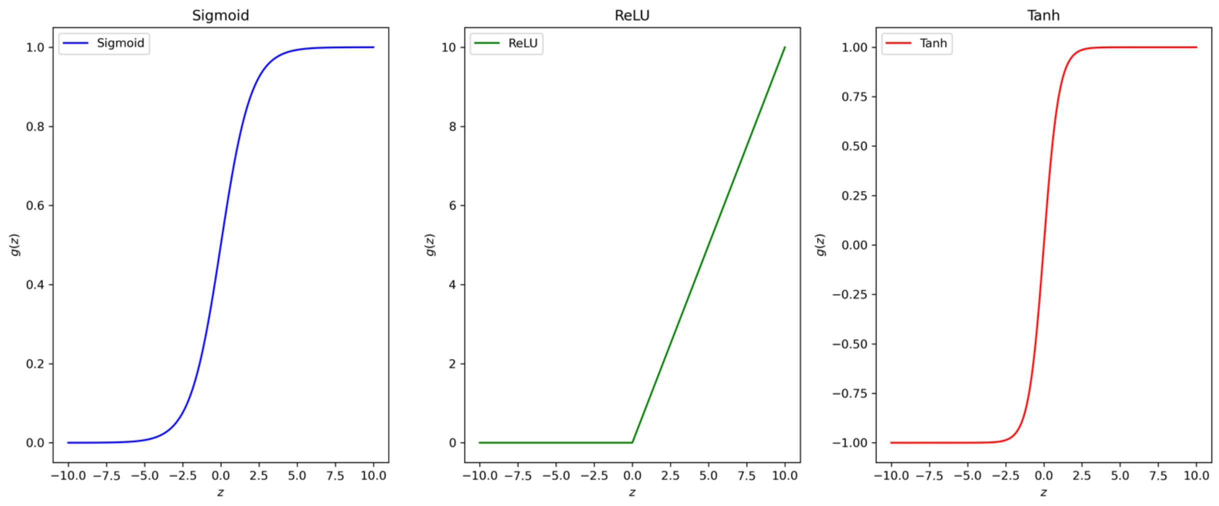
where

$$g(z)_{Sigmoid} : \mathbb{R} \longmapsto [0, 1]$$

$$g(z)_{ReLu} : \mathbb{R} \longmapsto \mathbb{R}^{+}$$

$$g(z)_{Tanh} : \mathbb{R} \longmapsto [-1, 1]$$

Figure 3.8 : Sigmoid, ReLu, and Tanh activation functions for neural networks



The first layer of a neural network is called the input layer and corresponds to the original data. The layers in the middle are called hidden layers[21] and the last layer is called the output layer, which is nothing less than the prediction. For the rest of this section, we assume that $x \in \mathbb{R}^n$ is a n-dimensional vector of input features for one training example. In fact, in the deep learning convention the full design matrix $X \in \mathbb{R}^{n \times m}$ is a matrix where the observations are stacked in columns instead of rows. Furthermore, we denote $w \in \mathbb{R}^n$ the weights associated with the input features and $b \in \mathbb{R}$ is the bias term. This deep learning convention is different from the traditional machine learning convention, where it chooses to represent the weights and the bias separately. In the simplest case, we can represent the shallowest neural network which consists of only one layer, namely the output layer. For that, the prediction is simply a linear combination of the input predictors plus a bias term:

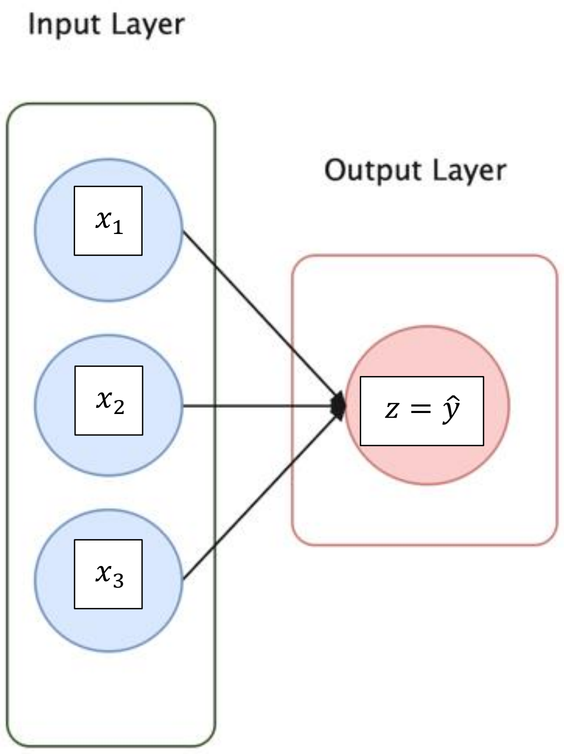
$$z = w'x + b\,, \tag{3.36}$$

$$\hat{y} = g(z) = z \tag{3.37}$$

As in our problem the dependent variable, namely the excess return, can take any real values, we apply the identity activation function $g(z) = z$, namely we do not transform the linear prediction part in (3.36). Then, we can provide a graphical representation of such a model, which is a neural network with an input layer composed of three regressors and an output layer which is a single real number:

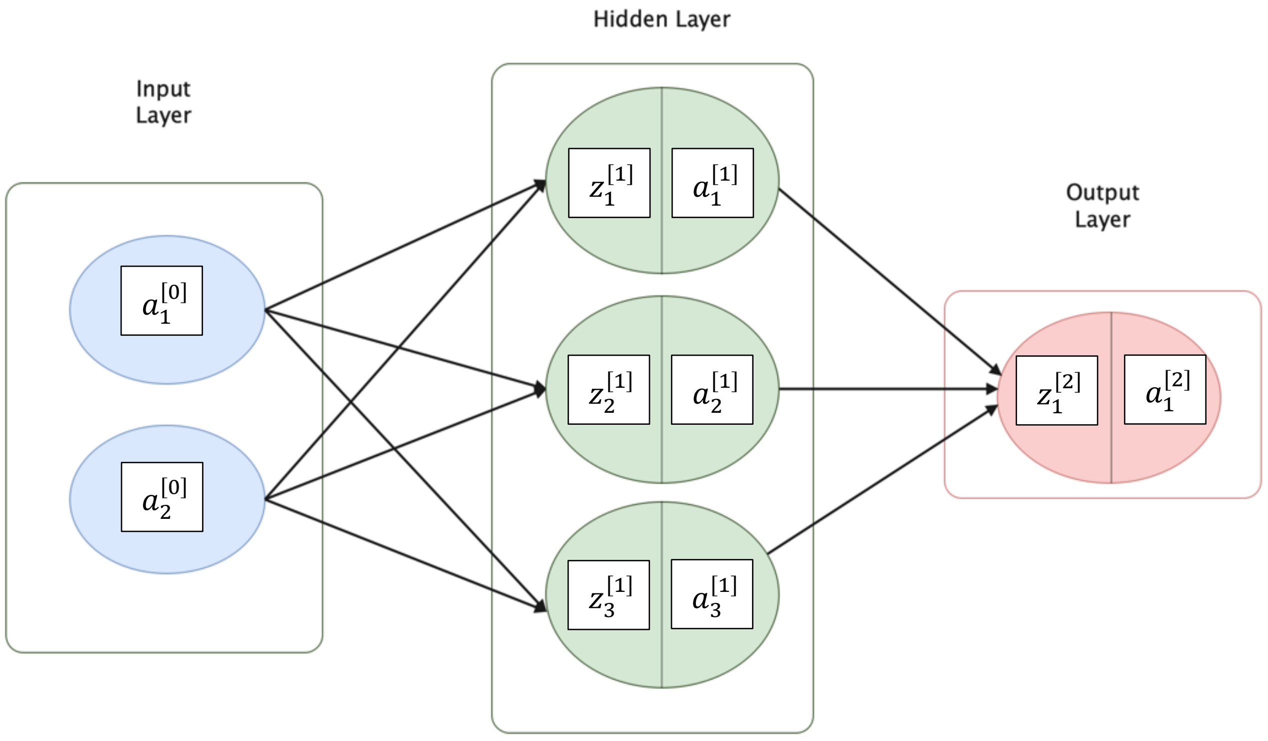
[21] Since the user of the neural network only sees the input and output layers.

Figure 3.9: **Shallow neural network with three regressors in the input layer and one output layer**



However, this simple neural network is exactly the same than the traditional linear model, as it does not perform any non-linear transformation of the input variables. For that reason, we can add one hidden layer between the input and output layers and on which we apply the ReLu activation function. To illustrate how it differs from the previous neural network, let us visualize the representation of this kind of neural network with three nodes in the hidden layer but this time with two regressors:

Figure 3.10 : Neural network with 2 layers



In the deep learning convention, the number of layers in a neural network is the number of its the hidden layers plus the output layer, as the input layer is not considered as a layer. Thus, the input layer is often referred as the layer 0, and the input vector is often denoted as $a^{[0]}$.

To go from the input to the output layer which computes the final prediction, the neural network model performs what is called forward-propagation. The forward-propagation for the neural network model illustrated in the figure (3.10) is performed as follows:

$$z_1^{[1]} = w'^{[1]}_1 a^{[0]} + b_1^{[1]}$$
$$a_1^{[1]} = g^{[1]}\left(z_1^{[1]}\right)$$

$$z_2^{[1]} = w'^{[1]}_2 a^{[0]} + b_2^{[1]}$$
$$a_2^{[1]} = g^{[1]}(z_2^{[1]})$$

$$z_3^{[1]} = w'^{[1]}_3 a^{[0]} + b_3^{[1]}$$
$$a_3^{[1]} = g^{[1]}\left(z_3^{[1]}\right)$$

$$z_1^{[2]} = w'^{[2]}_1 a^{[1]} + b_1^{[2]}$$
$$a_1^{[2]} = g^{[2]}\left(z_1^{[2]}\right) = \hat{y}$$

where $a_i^{[l]}$ is the i-th hidden unit in layer l, $b_i^{[l]}$ is the bias associated with the i-th node in layer l, $z_i^{[l]}$ is the linear part of the i-th node in layer l, and $w_j^{[l]}$ is the vector of weights in the l-th layer associated with the nodes in layer $(l-1)$, and $g^{[l]}$ is the activation function in layer l

Using matrix notation, we can re-write the forward propagation as follows:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$
$$a^{[1]} = g\left(z^{[1]}\right)$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g\left(z^{[2]}\right) = \hat{y}$$

where $a^{[l]}$ is the vector of hidden units in layer l, $W^{[l]}$ is the vector of weights transposed and stacked in rows, $b^{[l]}$ is the vector of biases in layer l, and $g^{[l]}$ is the activation function in layer l that performs element wise operation on each element of the vector $z^{[l]}$

The vector and matrix representation of each element involved in the computations in the first layer can be presented as follows:

$$a^{[0]} = \begin{pmatrix} a_1^{[0]} \\ a_2^{[0]} \end{pmatrix},\ b^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{pmatrix},\ W^{[1]}\begin{pmatrix} w'^{[1]}_1 \\ w'^{[1]}_2 \\ w'^{[1]}_3 \end{pmatrix} = \begin{pmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} \\ w_{3,1}^{[1]} & w_{3,2}^{[1]} \end{pmatrix},$$

$$z^{[1]} = \begin{pmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{pmatrix},\ a^{[1]} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{pmatrix} = \begin{pmatrix} g^{[1]}(z_1^{[1]}) \\ g^{[1]}(z_2^{[1]}) \\ g^{[1]}(z_3^{[1]}) \end{pmatrix}$$

Next, we can generalize to L layers with $n^{[l]}$ hidden units in layer l, which gives the general formulation as follows:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g\left(z^{[2]}\right) = \hat{y}$$

$$\vdots$$
$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g(z^{[l]})$$
$$\vdots$$
$$z^{[L-1]} = W^{[L-1]}a^{[L-2]} + b^{[L-1]}$$
$$a^{[L-1]} = g\left(z^{[L-1]}\right)$$

$$z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$$
$$a^{[L]} = g\left(z^{[L]}\right) = \hat{y}$$

where $z^{[l]} \in \mathbb{R}^{n^{[l]}}$ is a vector of the linear parts of hidden units in layer l, $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ is the matrix of weights in layer l associated with the activations in the previous layer $a^{[l-1]}$, $b^{[l]} \in \mathbb{R}^{n^{[l]}}$ is the vector of biases in layer l, and $a^{[l]} \in \mathbb{R}^{n^{[l]}}$ is the output produced in layer l [22]

In contrast to forward-propagation, the backward-propagation starts from the predictions $a^{[L]}$, and goes backward in the network so as to compute the derivatives of the cost function with respect to the model parameters, namely $W^{[l]}$and $b^{[l]}$ for $l = 1, \dots, L$. It is called backward-propagation, in the sense that we use the previous information to compute the next one, but this time starting from the end. These derivatives can be computed using the derivative chain rule, and once computed, we can find the optimal parameters using the gradient descent algorithm. Since neural networks are every expensive to train, different methods have been developed to reduce the training time and improve the score on the test set.

First of all, in addition to normalizing the input layer we also normalize the hidden layers before applying the activation functions. Normalizing the raw regressors is a standard approach for many machine learning algorithms, which enables to speed the convergence of the gradient descent algorithm. However, for neural networks, the linear components of each neuron, z, can also be normalized so that it has a mean of zero and a variance of one across all sets. There is still a debate in the deep learning literature about what we should normalize: z or a, but in practice normalizing z is used much more often. Batch Normalization is used to remedy the issue of the distribution's change of all hidden units values which is called

[22] Note that in this case $g\left(z^{[l]}\right)$ performs an element-wise operation on each element of the vector $z^{[l]}$.

"covariate shift". This change of distribution comes from the fact that on the training, dev, and test set the values of the regressors can be different, which may lead to a poor performance on the test set. Therefore, batch normalization enables the input values to become more stable, as no matter how much z can change, at least it has the same mean and variance. However, the downside of this approach is that we must optimize two additional parameters, namely γ and β that control the mean and variance respectively.

Secondly, neural networks have a specific regularization method called dropout regularization, which consist in eliminating some neurons by specifying a probability for which each neuron of each layer to disappear. The intuition behind dropout is that we cannot rely on only one feature because it can disappear during dropout, so we must spread out the weights across all features. The idea is that at each iteration in the gradient descent, we train a different model that uses only a set of neurons. Thus, neurons become less sensitive to the activation of one specific neuron as the distribution of data may be different on the train, dev and test sets. We can apply a probability p differently on each layer with bigger and lower values for massive and small layers respectively. Technically, we could also apply dropout to the input layer, but in practice it is not used that often. Nevertheless, the issue is that we have more hyperparameters to tune and the cost function is no longer well-defined (i.e. hard to calculate).

Lastly, ensemble methods can also be applied to neural networks so as to obtain more robust previsions in the test set. Nevertheless, such a method is very computationally expensive and would be impossible in deep learning applications where training just one neural network can take several weeks or even months. However, in the context of replicating the results of Gu and al. (2020), we use this method for each of our five neural networks. Each model's architecture varies from shallow to deep neural networks, but the number of units in each layer is obtained using the pyramidal rule: 32, 16, 8, 4, 2.
Specifically, the first model's architecture has only 2 layers with 32 units in its hidden layer, the second has 32 units in its first hidden layer and 16 in its second and so on.

4. Methodology

During this chapter, we will present the methodology used to fit our models to the data so as to obtain the highest predictive performance. The metric used to assess the predictive performance of all our models is the Pooled R-Squared, which provides the percentage of the dependent variable (i.e the excess return) explained by all regressors. However, according to Gu and al. (2020), in the denominator it is better to replace the Total Sum of Squares (TTS) by the sum of squared excess returns, because the historical expected return is too noisy so applying a naive forecast of zero gives better results. The formula for computing the Pooled R-Squared is given as follows:

$$R^2_{Pool} = 1 - \frac{\sum_{i\,=\,1}^{N} \sum_{t\,=\,1}^{T} \left(r_{t+1,i}{}^{E} - \widehat{r_{t+1,i}{}^{E}} \right)^2}{\sum_{i\,=\,1}^{N} \sum_{t\,=\,1}^{T} \left(r_{t+1,i}{}^{E} \right)^2}$$

Therefore, the purpose of this paper, is to identify which model gives the highest Pooled R-Squared. Nevertheless, using all observations for evaluating our models gives biased results, since the models would have already seen this data thus leading to unrealistic results. Indeed, the purpose of the fitting process is to minimize the cost function, which is approximately equivalent to maximizing the R-Squared[23].

Thus, in the next sections, we will explain in detail the cost functions used as well as the optimization algorithms to minimize it. Afterwards, we will present the train, dev, and test est sets, which are used to train our model (minimizing the cost function), searching for the best model complexity, and testing the model using new data. Finally, we will present the three hyperparameter tuning methods used in this paper, namely Grid search, Random search, and Bayesian search so as to find the model that performs the best on the test set.

4.1. Optimization framework

4.1.1. Cost function

Before introducing the cost function, let us distinguish between the loss function and cost function. On the one hand, the loss function is the prediction's error between the true excess return of stock i at time $t + 1$, $r_{t+1,i}{}^{E}$, and its forecasted value $\widehat{r_{t+1,i}{}^{E}}$. In contrast to the cost function which is applied to all the observations, the loss function is applied for only one

[23] For instance, the R-2 can be seen as a modified MSE.

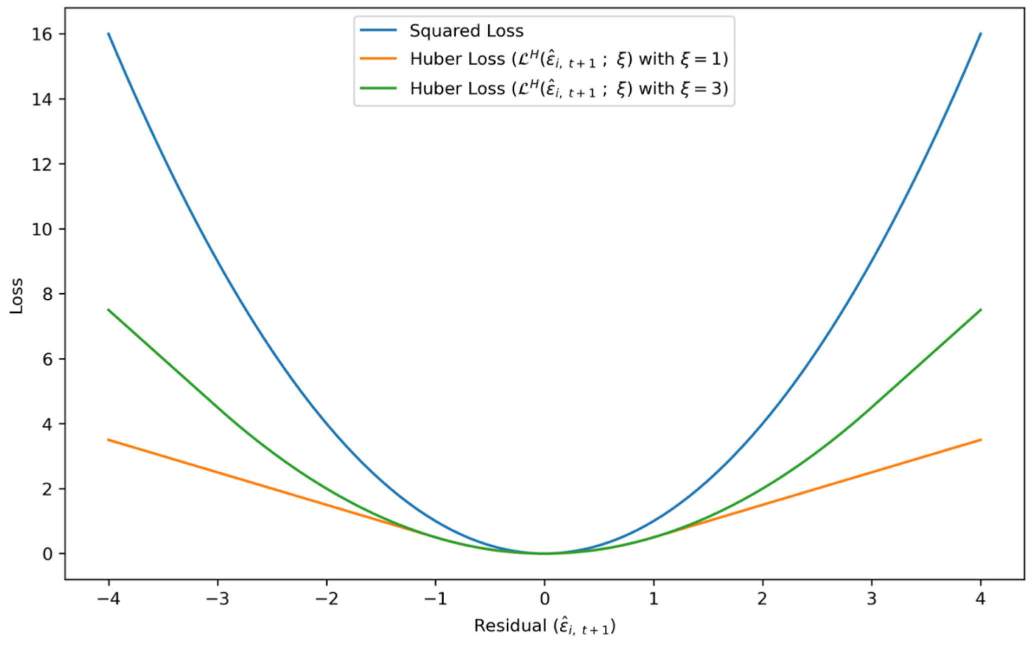
observation. In most of our models, the loss function is simply the squared deviation from the true value, which we simply denote as $\mathcal{L}\,(\,.\,)$. However, as financial data is very often skewed and heavy-tailed, it can be more appropriate to use the Huber loss function that we denote as $\mathcal{L}^H(\,.\,)$. The Huber loss function is defined with a parameter ξ, that controls the balance between an absolute deviation and a square deviation from the true value. To this end, for residuals in absolute value that do not exceed ξ, the Huber loss function penalizes the model quadratically, whereas for residuals larger than this threshold, the model panelizes errors linearly. In addition, we define $\hat{\varepsilon}_{i,t+1} = {r_{t+1,i}}^E - \widehat{r_{t+1,\iota}}^E$ as the prediction error (i.e. the residual) for stock i at time $t+1$. Formally, the squared loss function, $\mathcal{L}\,(\,.\,)$, and the Huber loss, $\mathcal{L}^H(\,.\,)$, are defined as follows:

$$\mathcal{L}\left(\hat{\varepsilon}_{t+1,i}\right) = {\hat{\varepsilon}_{t+1,i}}^2 \tag{4.1}$$

$$\mathcal{L}^H\left(\hat{\varepsilon}_{t+1,i}\,;\,\xi\right) = \begin{cases} {\hat{\varepsilon}_{t+1,i}}^2, & if\ \left|\hat{\varepsilon}_{t+1,i}\right| \leq \xi \\ 2\xi\left|\hat{\varepsilon}_{t+1,i}\right| - \xi^2, & else \end{cases} \tag{4.2}$$

In addition, we can provide a plot with the squared loss[24] as well as the Huber loss with $\xi = 1$ and $\xi = 3$ respectively:

Figure 4.1 : Squared loss and Huber loss with $\xi = 1$ and $\xi = 3$



[24] The words error, loss, and residual are used interchangeably.

On the other hand, the cost function that we define as $\mathcal{J}(\,.\,)$, is applied to all the observations in contrast to the loss function which is applied to only one single example. The size of the training set, namely the data that is used to fit the model is denoted as m_{train}, whereas the sizes for the validation and test sets are denoted as m_{val} and m_{test} respectively. Unlike the loss function $\mathcal{L}\,(\,.\,)$ which is function of the discrepancy between the predicted and actual value, the cost function is function of the model parameters that we denote θ which is a $(n, 1)$ vector. Even though non-parametric models such as trees and ensembles based-tree models do not have any parameters, they still rely on a cost function as we have seen earlier. Therefore, the cost function associated with the squared loss is called the Mean Squared Error (MSE) that we denote as $\mathcal{J}(\theta)$, and the one with the Huber loss is simply called the Huber cost function that we denote as $\mathcal{J}^H(\theta)$, and can be expressed as follows:

$$\mathcal{J}(\theta) = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}\mathcal{L}\left(\hat{\varepsilon}_{t+1,i}\right) \tag{4.3}$$

$$\mathcal{J}^H(\theta) = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}\mathcal{L}^H\left(\hat{\varepsilon}_{t+1,i}\,;\xi\right) \tag{4.4}$$

The optimal parameters $\hat{\theta}$ can be found by minimizing one of those cost functions, which are function of those parameters. Formally, we write the optimization problem for the MSE as well as for the Huber cost function as follows:

$$\hat{\theta} = \arg\min_{\hat{\theta}\,\in\Theta} \mathcal{J}(\theta) \tag{4.5}$$

$$\widehat{\theta^H} = \arg\min_{\hat{\theta}\,\in\Theta} \mathcal{J}^H(\theta) \tag{4.6}$$

where Θ is the set of all feasible parameter values

This type of optimization problem can be solved in two ways. At first, we can try to find a closed-form solution that expresses the optimal parameter values $\hat{\theta}$ using simple mathematical operations. Nevertheless, with complex optimization problems such as (4.6), an analytic solution does not exist, which forces us to rely on numerical solutions using efficient algorithms. We propose to explain the most important one, namely gradient descent algorithm.

4.1.2. Gradient descent algorithm

The gradient descent is a first order linear search that consists in finding the global minima of a cost function[25] by computing its derivatives with respect to the model parameters. The optimal parameter values are obtained where the derivative is equal to zero. Indeed, a global minimum is attained when the derivative at this point is equal to zero since there is no infinitesimal improvement of the function at this point. For regression problems, which is our case, the MSE is a convex function so there are no local minima. However, for classification problems the MSE is a non-convex function so other cost functions such as the log loss are used. For a minimization problem, the gradient descent starts from an initial starting point and improves it by moving into the opposite direction of the gradient. The size of the improvement relies on two components, namely the gradient computed on the previous iteration and an hyperparameter called the learning rate α. The gradient is not in our control, but the learning rate can be chosen so as to improve the convergence. The learning rate is the most important hyperparameter and its typical ranges is [0; 1]. Nonetheless, the learning rate needs to be tunned based on techniques that we will see in the next sections. The algorithm used in the gradient descent algorithm for one parameter θ can be presented as follows:

$$Initialize\ \theta_0 = 0$$
$$For\ i = 1, \dots, n:$$
$$\theta_{i+1} = \theta_i - \alpha \frac{d\, \mathcal{J}(\theta_i)}{d\, \theta_i}$$

In higher dimensional problems, namely when θ becomes a vector of parameters, the updating rule remains the same. Nevertheless, now the cost function becomes a multivariate function, so instead of using the differential term d we use the partial's derivative sign ∂. Moreover, we must update each parameter at the same time and not using optimal ones to fill in the cost function with so as to optimize other parameters. For simple models, the initial parameter value of θ_0 can be initialized at zero. However, for neural network models we must randomly initialize the weights W close to zero and not exactly to zero, so as to avoid vanishing and exploding gradient. Thus, if there are p parameters, the algorithm now becomes:

[25] It can also be used to find the maxima of any function.

$$Initialize\ \theta_0 = 0$$
$$For\ i = 1, \dots, n:$$

$$\theta_{i+1}^1 = \theta_i^1 - \alpha \frac{\partial\, \mathcal{J}(\theta_i^1, \theta_2^1, \dots, \theta_p^1)}{\partial \theta_i^1}$$

$$\theta_{i+1}^2 = \theta_i^2 - \alpha \frac{\partial\, \mathcal{J}(\theta_i^1, \theta_2^1, \dots, \theta_p^1)}{\partial \theta_i^2}$$

$$\vdots$$

$$\theta_{i+1}^j = \theta_i^j - \alpha \frac{\partial\, \mathcal{J}(\theta_i^1, \theta_2^1, \dots, \theta_p^1)}{\partial \theta_i^j}$$

$$\vdots$$

$$\theta_{i+1}^{p-1} = \theta_i^{p-1} - \alpha \frac{\partial\, \mathcal{J}(\theta_i^1, \theta_2^1, \dots, \theta_p^1)}{\partial \theta_i^{p-1}}$$

$$\theta_{i+1}^p = \theta_i^p - \alpha \frac{\partial\, \mathcal{J}(\theta_i^1, \theta_2^1, \dots, \theta_p^1)}{\partial \theta_i^p}$$

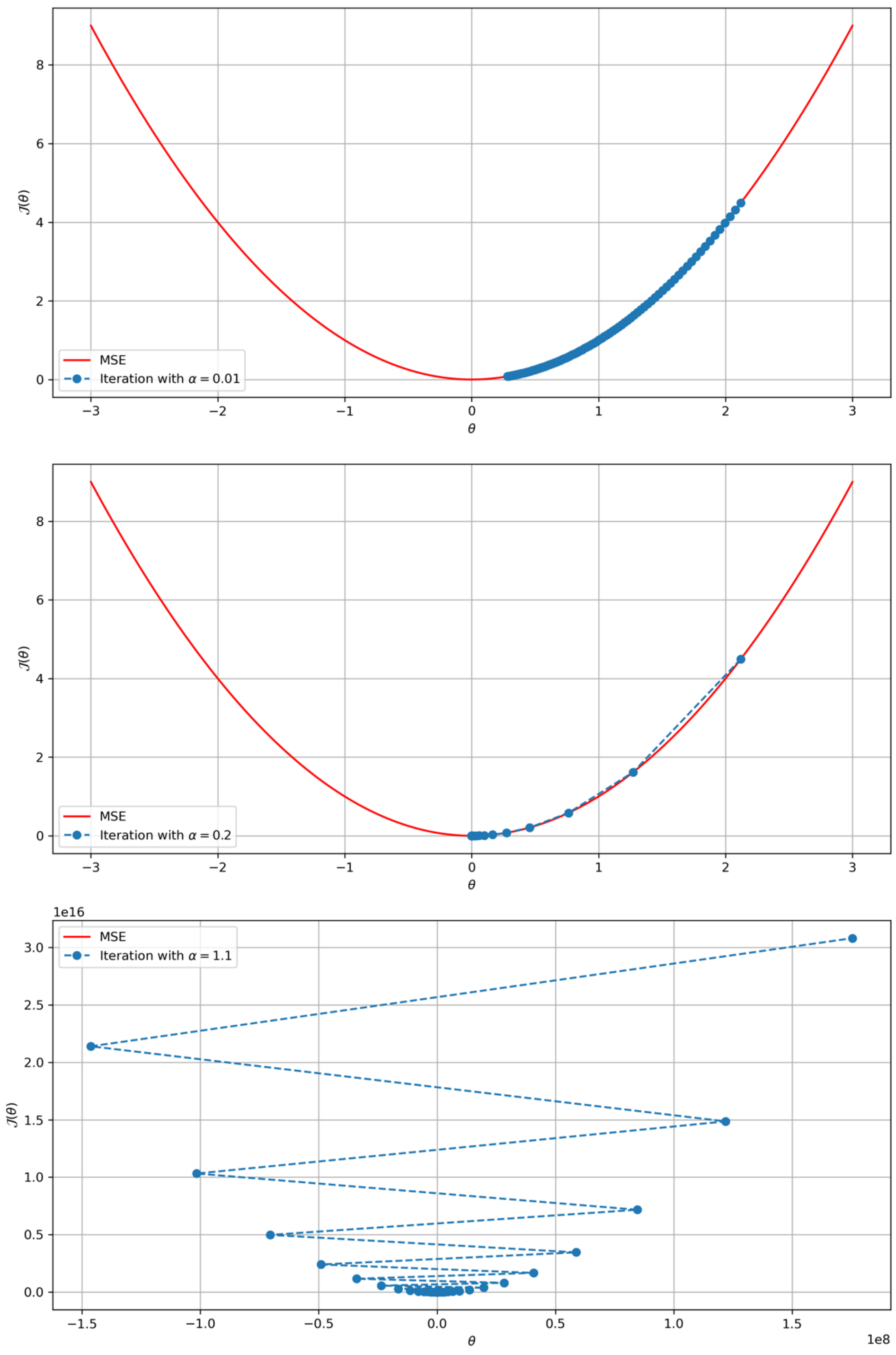
where θ_i^j is the parameter value j at iteration i

Therefore, when the gradient equals to 0, then $\theta_{i+1} = \theta_i$ and the parameters are no longer updated so the algorithm stops. However, in practice we may never achieve the global minima at the exact decimal point, and actually this is not what we want. Indeed, in numerical optimization we are looking for an accurate estimate of the optimal values, which can be achieved at a low computational cost. Moreover, the descent may take too much time to converge, especially in deep learning applications. For that purpose, several methods have been developed.

One useful method to reduce the number of iterations is called early stopping. This method consists in stopping the algorithm when for a specified number of iterations either the parameters of the cost function or the cost function itself has stopped improving by some tolerance parameter ϵ. In our application, this method is particularly useful for training neural networks and performing Bayesian hyperparameter tuning. Indeed, in average it divides by three the total number of iterations specified initially, which helped us saving dozens of hours of training.

The learning rate α controls the step size at each iteration and must be chosen carefully. In fact, a too high value may lead to overshooting the optimal regions, while a too small value may lead to unsignificant improvements between each iteration. This is why in deep learning applications there is a method called learning rate decay, which involves taking larger steps at the beginning of the descent, and either as time passes or as the number of iterations increases, the learning rate decreases accordingly. To highlight our thought, we can plot the gradient descent algorithm for one parameter θ for with three different learning rate values:

Figure 4.2: Gradient descent algorithm with low, medium, and high learning rates



A powerful extension of the gradient descent which is popular to train neural network models is the Adam solver. This optimizer is very efficient to train neural network models as it is a combination of two efficient algorithms, namely Momentum and RMSProp. The Adam solver uses exponential weighted moving averages to store the gradients of the closest iterations so as to adapt the step of the descent for each parameter.

4.2. Data splitting and cross-validation

The data splitting method consists in splitting our original data into three sets, namely the training, development, and test set so as to obtain the best results on new data.

First of all, the training set is composed of observations which are used to feed the model so as to minimize the cost function. The more data we got, the more accurate our model will be at capturing the actual relationship between the dependent variable and the regressors. This is why in the pre-deep learning area when neural networks were not as popular as today, it was said “It is not who has the best algorithm that wins. It is who has the most data”. As a matter of fact, even though in the post-deep learning where we mostly focus on specific neural network architectures or regularization techniques to increase the out-of-sample performance, increasing the training set size is still a good option to obtain better results.

Secondly, the development set, sometimes juts called the dev set is used to choose the complexity of the model that performs the best using data that the model has not been trained on. To that end, we specify ex-ante the model’s complexity and evaluate the performance on this dev set, so that the best model is the one that performs the best on the dev set. Indeed, one major issue in machine learning is that sometimes machine learning models are too flexible, and they end up fitting noise in the data. This issue is commonly referred to “overfitting” and is ubiquitous in stock returns prediction since the signal-to-noise ratio is very low. The complexity of the model is controlled by what are called hyperparameters and will be discussed in depth in the next section.

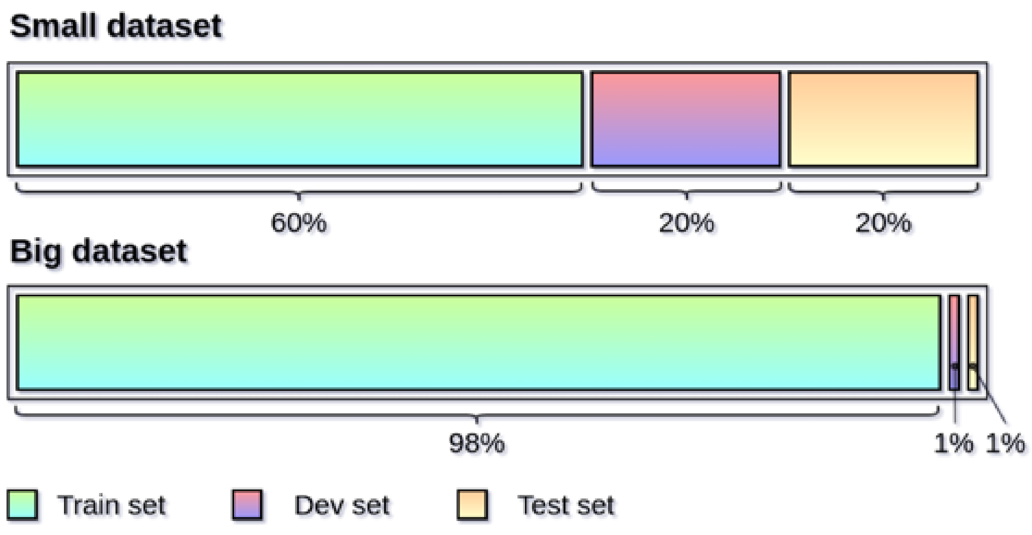
For instance, if we take the ENet model it has an hyperparameter called λ, which controls the strength of the regularization with a higher value means a less complex model and vice versa. Therefore, the optimal hyperparameter value for lambda is the one that maximizes the Pooled R-Squared evaluated on the dev set as follows:

$$\lambda^* = \arg\max_{\lambda \in \Theta} R^2_{Dev}$$

where R^2_{Dev} is the Pooled R-Squared evaluated on the dev set, and Θ is the space for the hyperparameter λ i.e $\mathbb{R}^+$

Finally, the test set is used to ensure that the performance on the dev set is unbiased. Indeed, assessing the out-of-sample performance on the dev set may lead to unrealistic results. This is because even though the model has not seen this data, we have still chosen our model's complexity as the one that achieved the highest out-of-sample performance.
In most machine learning applications excepting time series applications, the train-dev-test percentage split is usually chosen as 60/20/20 so that 60%, 20%, and 20% of the whole dataset is allocated to the training, dev, and test sets respectively. In contrast, in deep learning applications when we are dealing with millions of observations, the percentage split is typically 99/1/1, or even 99.5/0.5/0.5. Here is an example of the two most popular sample splitting:

Figure 4.3: Train-dev-test split with a small and big dataset

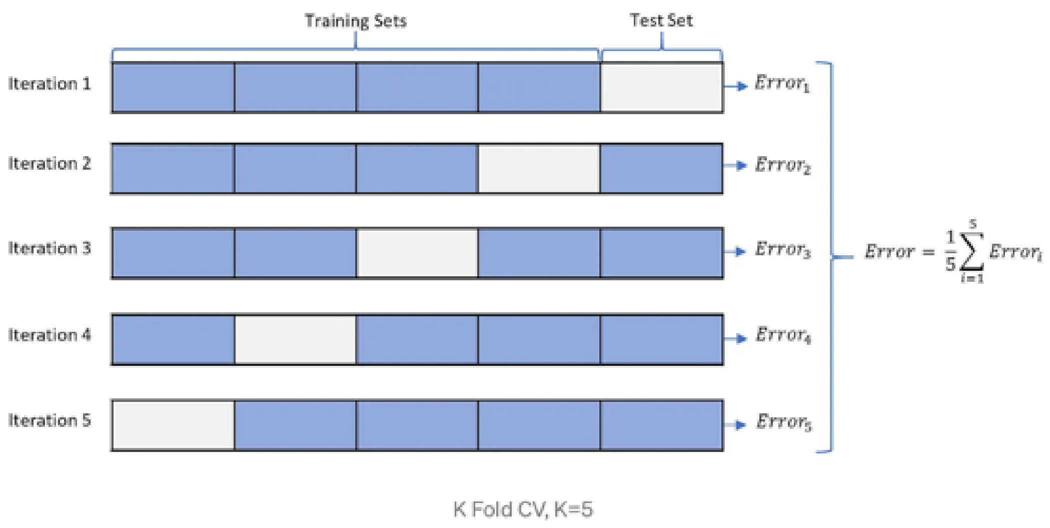


Source: meeyland.com

In these standard machine learning applications, before splitting the data, the dataset is randomly shuffled to ensure unbiased results. However, randomly shuffling the data might still not be enough as each set might not be perfectly homogenous. For instance, there might be too much large-caps in the training set and too many small-caps in the test set.

For that purpose, one popular machine learning technique is called cross-validation and involved creating multiple splits and evaluating the model on each of them. Afterwards, it retains the final model’s performance as the mean of the scores across each split. There are many cross-validation methods available, with the standard one being K-fold that can be illustrated as follows:

Figure 4.4: K-fold cross validation with 5 splits

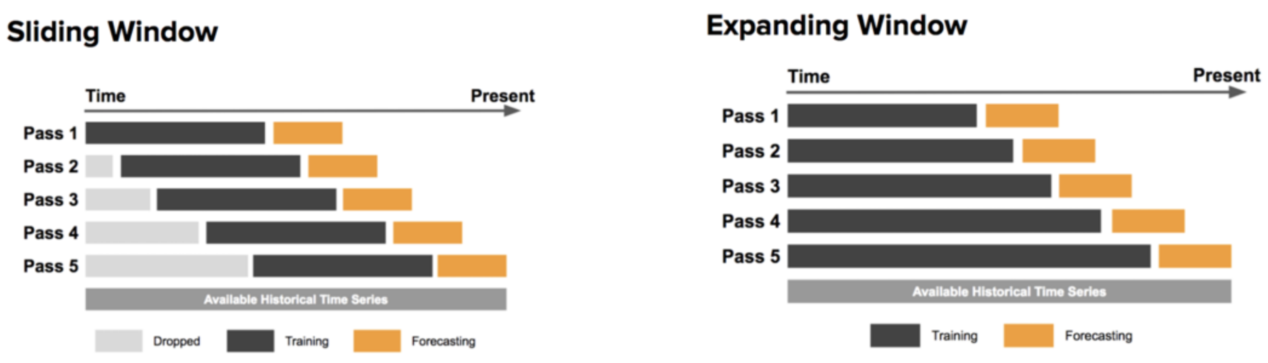


Source: towardsdatascience.com

Nevertheless, the previous steps described above, namely randomly shuffling the data before splitting and using standard cross-validation techniques such as k-folds are not suited when dealing with time-series data. Indeed, with time-series data we must preserve the temporal order in order to avoid the look-ahead bias. For instance, if we considered k-fold cross validation as illustrated in figure (4.4) and if each set preserved the temporal order, the four last iterations would use data from the future to predict data in the past, thus creating a bias. Therefore, for time-series data there exists two major cross-validation schemes that account for the specificity of time-series data.

On the one hand, the first is a fixed sliding window that moves one observation ahead at each cross-validation and discards the first observation. This method is relevant for most applications as in time-series models, past data have less impact than more recent ones. On the other hand, the other method is an expanding window, which unlike the fixed sliding window does not drop the first observations at each iteration in the cross-validation. Here is a plot displaying each method:

Figure 4.5: Sliding and expanding windows with 5 splits



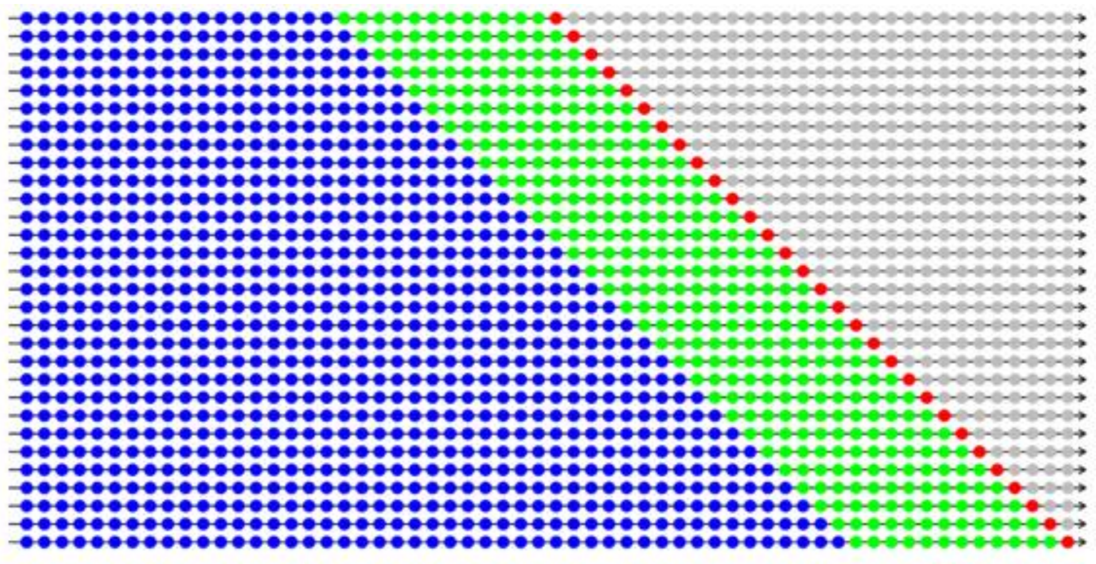
Source: uber.com

The benefit of using an expanding window is that the number of observations increases at each iteration, but the first observations might not be relevant anymore as market conditions change very fast. However, as we are following the methodology of Gu and al. (2020), we consider the expanding window as our cross-validation method. Moreover, one particular benefit of using cross-validation for predicting equity excess returns, is that as market conditions can change very fast, the complexity changes accordingly so the hyperparameter values can change significantly across time. The validation scheme used in the paper of Gu and al. (2020) involves thirty iterations where the period is one year. At each iteration, the training set expands by one year, whereas the validation and test sets use a fixed period of twelve years and one year respectively.

In the first iteration, the training period goes from 1957 to 1976, the validation period from 1977 to 1986, and the testing period is the year 1987. Once the data has been split, we train our model on the training set and find the best hyperparameter value on the dev set. However, we choose to re-estimate our model on the training and dev set using the best hyperparameter values found on the dev set before evaluating our model on the test set. Indeed, as machine learning models require a lot of data to perform well, considering twelve years of data that appear just before the testing year might yield better results rather than discarding them. Next, we compute the prediction errors on the test set.

In the second iteration, the training period now goes from 1957 to 1977, the dev period from 1978 to 1987, and the testing data is the year 1988. This process is repeated until the 30th iteration and the R-Squared is computed based on the prediction errors at each iteration. The scheme can be illustrated as follows:

Figure 4.6: Cross-validation scheme in Gu and al. (2020)



Source: shihaogu.com

Each blue dot corresponds to one year of training data, the green one to one year of dev data and the red represents one year of test data.

4.3. Hyperparameter tuning

In the previous section, we have talked about the importance of using a dev set in order to choose the complexity of the model which is the most suited to our data. There are two major approaches for hyperparameter tuning, each one having its pros and cons.

The first approach is the most popular and is named the caviar approach[26] and consist in fitting each model in parallel. The second approach is called the panda approach in analogy to pandas having a few babies and taking care of them. This approach consists in fitting a few models and as time passes, the hyperparameter values are changed manually by the practitioner. For instance, we could manually adjust the learning rate for the gradient descent which controls the step size of the descent, by taking a higher value at the beginning and progressively reducing it when approaching the minimum of the cost function.

Choosing between the two approaches depends on the computational resources available (i.e CPU and GPU memory) along with the size of the dataset. The caviar approach is more suited when computational resources are not limited, which is mostly the case in machine learning applications. In contrast, the panda approach is most suited for deep

[26] In analogy to how fish pound eggs and not taking care of them

learning applications. Therefore, in our application which is closer to a machine learning application[27], we use the caviar approach, namely we train all our models in parallel without worrying about manually changing the hyperparameter values.

In recent years, hyperparameter tuning[28] has drawn the attention of the academic research that has come up with heuristic methods such as: Bayesian optimization or genetic algorithms, that can provide optimization results in a reasonable time. Indeed, the high computational time needed to fit machine learning models comes from the tuning process, which consists in fitting many times the same model, but using different values for each combination of hyperparameters. As a matter of fact, standard approaches like Grid search or Random search suffer from many caveats when the space of hyperparameters becomes excessively complex. For that reason, formal searches like Bayesian searches are popular nowadays as they take information from the previous iterations to choose the next most promising hyperparameter value to test.

Formally, hyperparameter tuning consists in finding the set of hyperparameter values such that with those values the model achieves the highest performance on the dev set. Hence, the problem formulation for all hyperparameter tuning methods can be expressed as follows:

$$x^* = \arg\max_{x \in X} R^2_{Dev} \tag{4.7}$$

where x is a set of hyperparameter values, x^* is the optimal ones, X is the set of all possible hyperparameter values, and R^2_{Dev} is the coefficient of determination computed on the dev set

4.3.1. Grid search

The first and most popular hyperparameter search that we use is called Grid search. This method comes from the Brute force algorithm in computer science that tries all possible combinations. For instance, to decipher a password if this one only contains a few lowercase letters, we can try all possible combinations in a reasonable time. Nevertheless, when the password is large and composed of special characters and numbers, the number of combinations expands exponentially as it could take thousands of years to test all possible

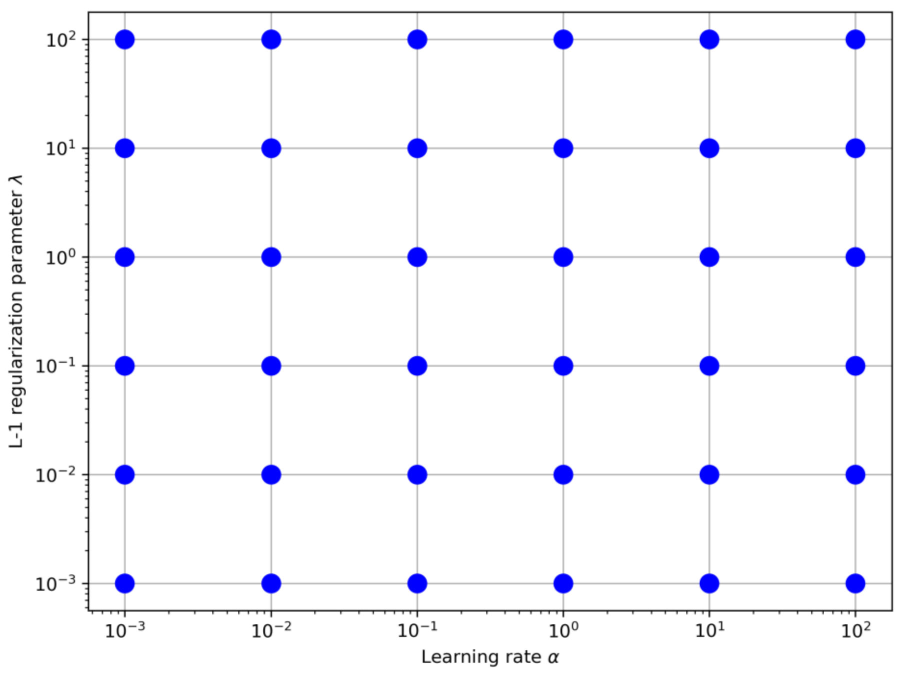
[27] Even though the full dataset has a few million observations, it is more appropriate to characterize the dataset size as the number of observations in the time-series dimensions only. In fact, because of the cross-sectional dependency of stocks, the incremental gained obtained by adding more stocks is too limited to be considered. Hence, we rather say that we have T monthly observations where $T = 12 \times 104 = 1248$, as we 12 monthly observations form 1957 to 2021.

[28] Now we refer to hyperparameter tuning as the caviar approach which is the standard one.

combinations. This phenomenon is often referred to the curse of dimensionality and is the major drawback of using such a method.

The Grid search for hyperparameter tuning operates the same way, but now each character in the password corresponds to one hyperparameter, and all possible values for that character correspond to all possible values for each hyperparameter. For example, if we have a hyperparameters to test and that for each hyperparameter we test b values, we will perform a^b combinations. Therefore, to test 100 values for 5 hyperparameters, we have $100^5 = 1^{10}$ models to train, which can be highly computationally expensive. However, one property of Grid search over other methods, is that as it tries all possible combinations it ensures that among all possible combinations the best model is retained. Here is an example of a Grid search with 36 possible combinations for the hyperparameters λ and α respectively:

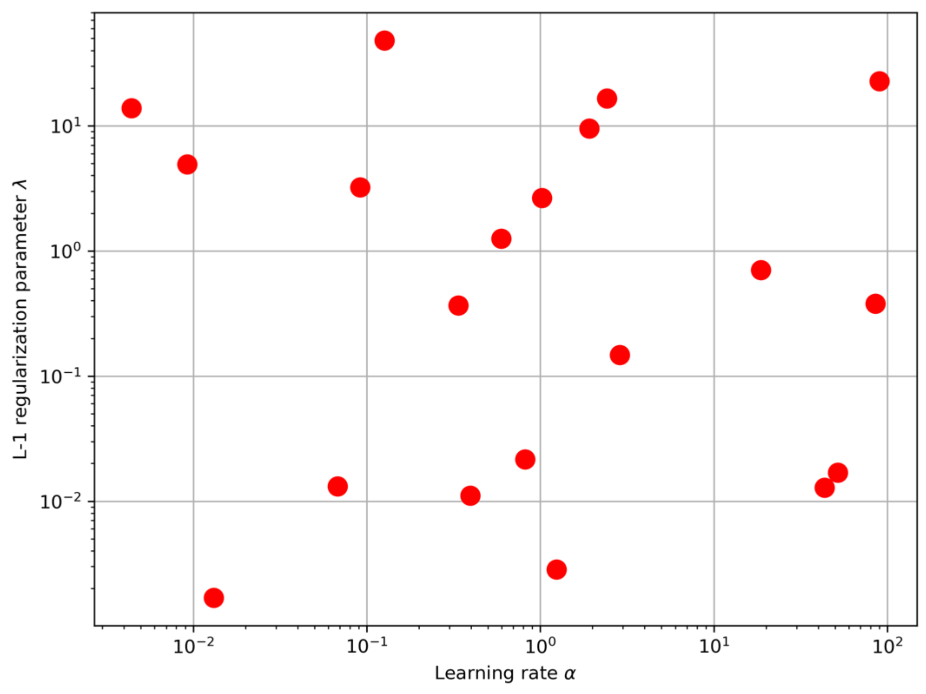
Figure 4.7: Grid search with hyperparameters alpha and lambda



4.3.2. Random search

One major issue with Grid search is that it can be computationally expensive when there are a lot of combinations to test. Unlike Grid search, Random search does not test all possible combinations. In fact, Random search only randomly samples a few of them where the number of samples drawn can be specified to adjust the computational time. Thus, in higher research spaces, typically in deep learning applications when there are many hyperparameters, Random search is usually preferred over Grid search. However, one issue with this approach is that we may not end up with the hyperparameter values that maximize the performance out-of-sample. Nonetheless, in practice we have limited computational resources, thus Random search is sometimes the best option. Here is the figure (4.7) but now with Random search:

Figure 4.8: Random search with hyperparameters alpha and lambda



4.3.3. Bayesian search

The two previous hyperparameters searches, namely Grid and Random searches are no longer optimal when the number of hyperparameters to test becomes large. Indeed, for Neural Networks and Gradient Boosting Decision Trees, we can end up tuning a dozen of hyperparameters. Thus, Random search will also suffer from the curse of dimensionality. In fact, the main issue with the two previous searches is that they are considered as "uninformed". This is because all the iterations are performed in parallel where each one as no impact on the following.

To counteract this issue, a new type of hyperparameter tuning method called "informed" search allows the results from the previous iterations to have an impact on the next ones. Specifically, informed searches enable to narrow down the range of promising hyperparameter values which have the highest predictive performance on the dev set. As a matter of fact, the issue with Grid and Random searches is that they may end up wasting too much time exploring hyperparameter regions that have no predictive performance on the dev set.

Bayesian search is a popular informed search which taps into the principles of Bayesian's inference, which is itself based on Bayes' theorem. This theorem states that based on some initial guess about the unknown probability distribution function (pdf), we can update this pdf based on the results we obtain from the experiment. In the context of hyperparameter tuning, the probability distribution of each hyperparameters from which we draw randomly just like Random search is modified at each iteration based on the score we obtain on the dev set.

Specifically, the purpose of Bayesian optimization in hyperparameter tuning is to minimize an unknown objective function, which takes as input the hyperparameter values and gives the objective function value as output. For that, Bayesian optimization constructs a probabilistic model of this unknown objective function which is updated after each iteration. Indeed, to obtain the full unknown objective function, we would need to evaluate the objective function at all possible hyperparameter values, which is computationally impossible. The probabilistic model is often called the surrogate model and the most common choice for it being a Gaussian process which can be expressed as follows:

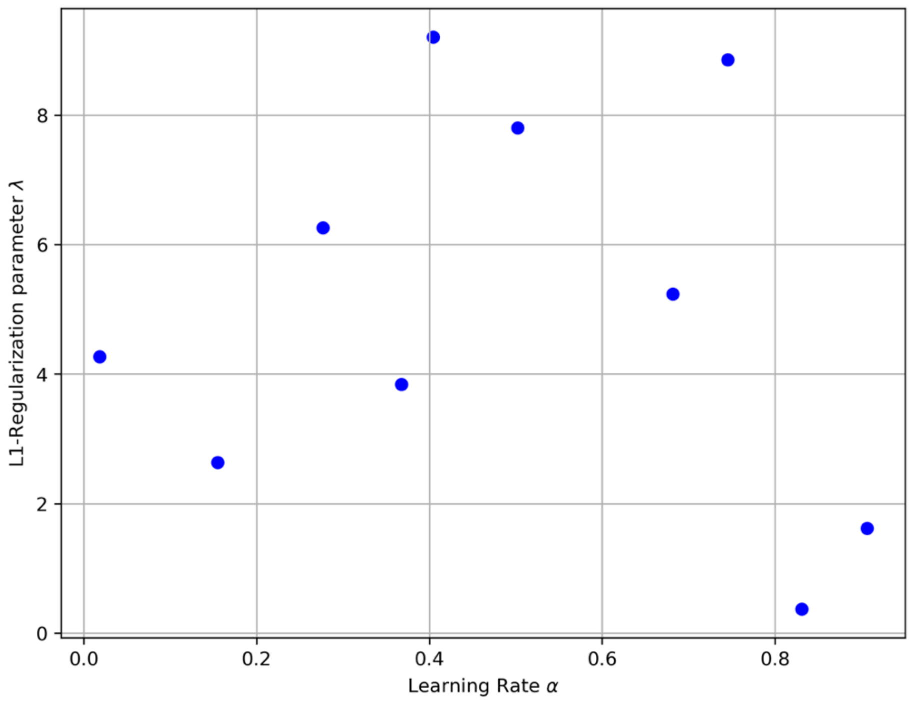
$$\mathbb{P}(y|x) \tag{4.8}$$

where y is the objective function value, and x is the hyperparameter values

This formula involves under the hood two important components of Bayesian's inference, namely the prior and posterior distributions.

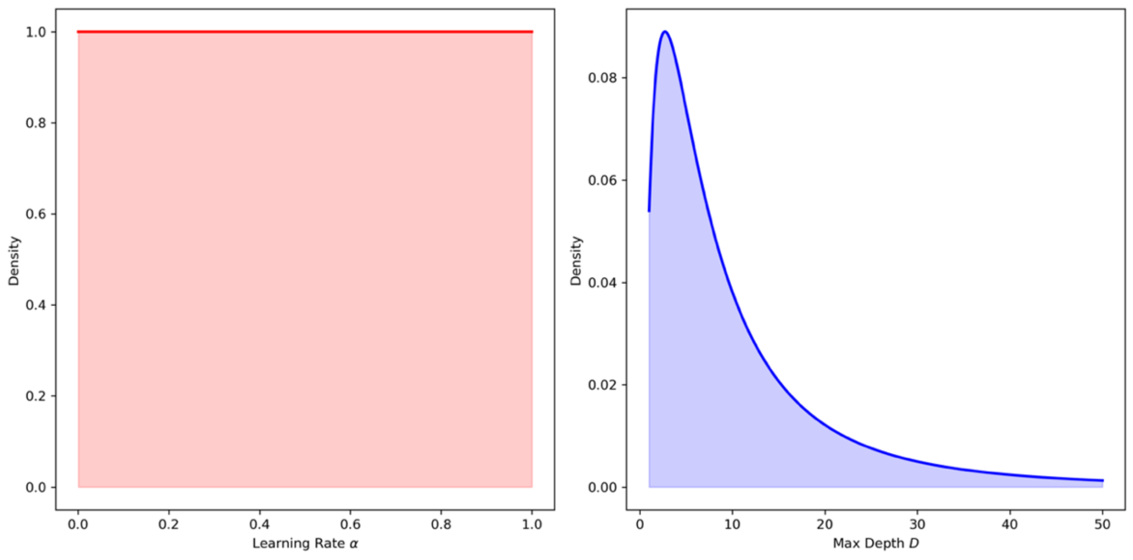
One the one hand, the prior distribution, which is sometimes abbreviated as the prior, intervenes in the first iteration of the Bayesian optimization process. Specifically, the prior corresponds to the initial distribution from which we draw randomly a sample of hyperparameter values and compute the objective function values based on those values. This first sampling is usually performed using Latin Hypercube Sampling (LHS), which is a statistical sampling method used to ensure that the sample covers the range of the prior distribution. Here is an example of LHS for a sample of ten combinations of hyperparameters for the learning rate α and the l-1 regularization hyperparameters λ, where their ranges are [0;1] and [0;10] respectively:

Figure 4.9: Latin Hypercube Sampling for the learning rate and l-1 regularization hyperparameters



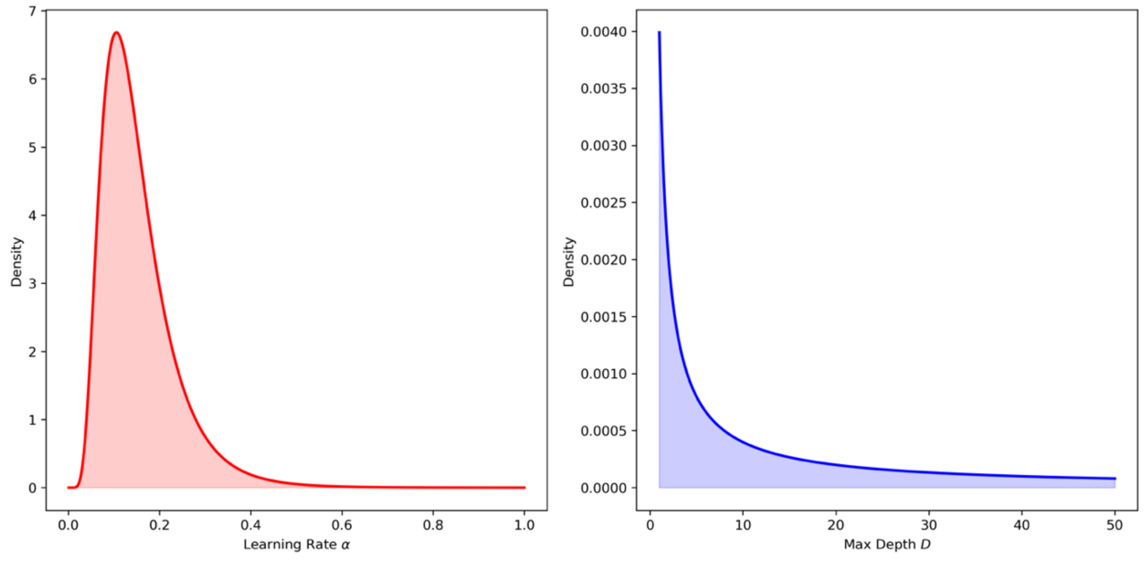
Furthermore, the prior distribution can be specified based on some prior knowledge, which in that case can speed up the optimization process. The standard prior distributions are the log normal, uniform, and beta distributions. Here is an example of a prior uniform distribution for the learning rate α, and a log-normal distribution for the maximum depth D hyperparameter for Gradient Boosting:

Figure 4.10: Prior uniform and log-normal distributions for the learning rate and the maximum depth hyperparameters



On the other hand, the posterior distribution is the updated version of the prior distribution based on the result of the experiment, namely the dev scores obtained from our sample of hyperparameter values. The posterior plays a key role in determining the next hyperparameter values to draw, as it assigns a higher probability to hyperparameter values giving a higher performance on the dev set and vice versa. For instance, in deep learning applications the optimal learning rates are usually in the interval [0.00; 0.1]. Thus, the uniform prior distribution in figure (4.10), will be me modified accordingly so that the highest density is located in this range. In contrast, for Gradient Boosting, the maximum depth hyperparameter is usually less than three in most applications. Therefore, the prior distribution in figure (4.10) closely resembles the posterior and will require less iterations than for the learning rate. This is an example of those posterior distributions at the end of the optimization process:

Figure 4.11: Posterior uniform and log-normal distributions for the learning rate and the maximum depth hyperparameters



For all iterations from the second one to the last one, a new hyperparameter value is chosen that we denote as x_{new}, so as to maximize an acquisition function (also called the selection function). There are several acquisition functions, each one balancing differently between exploration (exploring new hyperparameter regions) and exploitation (exploiting promising areas). Afterwards, the objective function score that we denote as y_{new} is computed, and the probabilistic model is updated based on the results. The most popular type of acquisition function is called the Expected Improvement (EI) and is expressed as follows:

$$x_{new} = \arg\max_{x \in \Theta} EI(x) \tag{4.9}$$

where

$$EI(x) = \int_{-\infty}^{+\infty} (y - y^*, 0)^+ \mathbb{P}(y|x)\, dy$$

x_{new} is the new set of hyperparameter values, y^* is the lowest R-Squared obtained so far, and y is the new R-Squared obtained based on the specified value of x_{new}, and $(.\ ,\ .)^+ = \max(.\ ,\ .)$

Basically, this formula states that the hyperparameter values for the next iteration are obtained such that the EI is maximized. Therefore, to maximize EI, the first part of the integral, $(y - y^*, 0)^+$must be high, namely the new score computed from x_{new}, namely y must be significantly larger than all the previous R-Squared obtained so far. However, the second part of the integral represents the uncertainty in those results. Most optimization packages perform minimization so in order to maximize the R-squared we must minimize ($-$ R-Squared) which is equivalent.

5. Results

The full dataset used in the original paper of Gu and al. (2020) encompasses 30000 stocks from 1957 to 2016. Nevertheless, in our application due to limited computational resources we use at the maximum 1000 of them. This is because we test for a lot of hyperparameter combinations, between 10 and 30, so it might be significantly larger than the number hyperparameter combinations tested in Gu and al. (2020). Indeed, in their paper there is no indication about how many hyperparameter searches are performed, which is an important parameter to consider as it can increase significantly computational time.

The hyperparameters tuning method used for each parametric and non-parametric model along with the number of iterations performed in our application can be presented as follows:

Table 5.1 : Hyperparameter tuning method and number of iterations performed for parametric models

Model / Method	OLS +H	OLS-3 +H	PCR	PLS	ENet +H	GLM
Grid			10	6		16
Random					30	
Bayesian						

Table 5.2: Hyperparameter tuning method and number of iterations performed for non-parametric models

Model / Method	RF	GBRT +H	NN1-NN5
Grid			
Random			
Bayesian	30	30	15

where +H denotes to the Huber loss

The two OLS models do not require any tuning. In fact, the only hyperparameter that could have been tuned is the ξ parameter for the Huber loss that controls the penalization of the residuals. However, like Gu and al. (2020), we do not tune it and we consider a fixed value which can be computed as the 99.9% percentile of the residuals. Therefore, to compute ξ we must first fit an OLS model in order to compute the residuals, but this is not computationally expensive as OLS models are easy to train.

Grid Search and Random Search are largely sufficient for parametric models as the hyperparameter space is not too complex. Nonetheless, for non-parametric models we prefer performing Bayesian Search as it yields better results. In addition to Bayesian Search, we use Early stopping with a tolerance of 0.01% and a patience of 5, which reduces significantly computational time. This means that if the best R-Squared computed on the dev set at a given iteration has not been improved by more than 0.01% 5 times since it has been reached, then the optimization process stops. For instance, with RF and GBRT the optimization process often stopped before the 30-th, so it can save up a lot of time.

Next, the hyperparameters along with their respective ranges of values can be presented below:

Table 5.3 : Hyperparameter ranges for parametric models

Model/Hyperparameter	**PCR**	**PLS**	**ENet**	**GLM**
n_components	[1, 2, 3, 4, 5, 8, 10, 12, 15, 20]	[1;6]		
l1_reg			(1e-4; 1e-1)	[1e-5, 1e-4, 1e-3, 1e-2]
group_reg				[1e-5, 1e-4, 1e-3, 1e-2]

Table 5.4: Hyperparameter ranges for non-parametric models

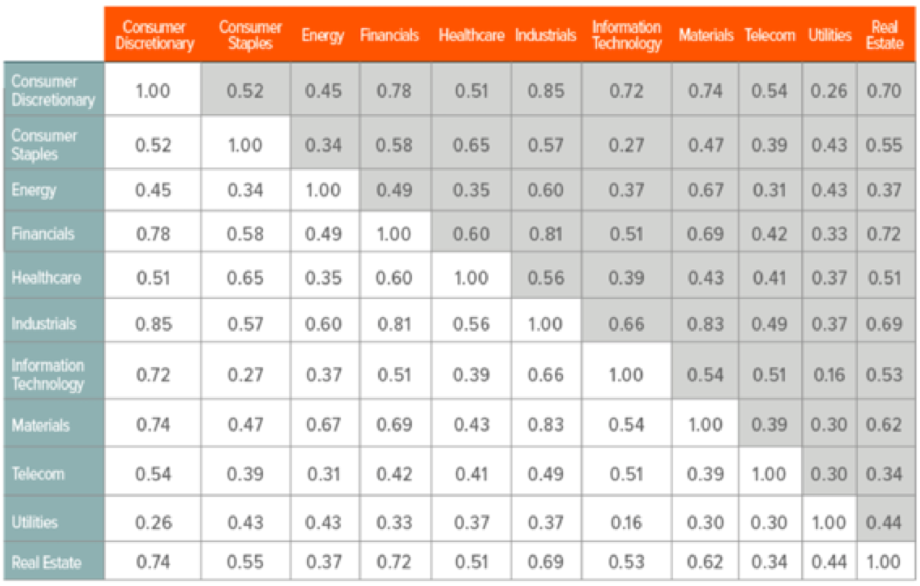
Model/Hyperparameter	**RF**	**GBRT**	**NN1-NN5**
n_estimators		[20; 1000]	
max_depth	[1; 6]	[1, 2]	
max_features	[3, 5, 8, 10, 12, 15, 20, 30, 40, 50, 80]		
learning_rate		(1e-2; 1e-1)	(1e-5; 1e-3)
l1_reg			(1e-5; 1e-3)
dropout_rate			[None, 0.05, 0.1, 0.15, 0.2]

where [ ] and ( ) denote to integer and floating hyperparameters, and a; b denotes to the range from a to b whereas [a, b, c] simply denotes to successive values to be tested

Gu and al. (2020) highlight in their paper that shallow learning outperforms deep learning which is a bit counterintuitive. In fact, for 1000 stocks the dataset accounts for about

130000 observations, thus for the full dataset it might be in millions. This is the usual size we have in deep learning applications where deep neural networks outperform shallow ones. However, this result obtained by Gu and al. (2020), might be due to the cross-sectional dependence of stocks, which may limit the incremental gain obtained by adding more stocks on the total number of observations. Cross-sectional dependence implies that the returns of different stocks are not independent of each other. They can be influenced by common factors such as economic conditions, industry trends, and so forth. Consequently, adding more cross-sectional observations (e.g., data on more stocks) may not provide as much new information as adding more time-series observations (e.g., data on the same stocks over more time periods). Here is a correlation matrix of returns between economic sectors in the US:

Figure 5.1: Correlation matrix of returns between economic sectors in the US



	Consumer Discretionary	Consumer Staples	Energy	Financials	Healthcare	Industrials	Information Technology	Materials	Telecom	Utilities	Real Estate
Consumer Discretionary	1.00	0.52	0.45	0.78	0.51	0.85	0.72	0.74	0.54	0.26	0.70
Consumer Staples	0.52	1.00	0.34	0.58	0.65	0.57	0.27	0.47	0.39	0.43	0.55
Energy	0.45	0.34	1.00	0.49	0.35	0.60	0.37	0.67	0.31	0.43	0.37
Financials	0.78	0.58	0.49	1.00	0.60	0.81	0.51	0.69	0.42	0.33	0.72
Healthcare	0.51	0.65	0.35	0.60	1.00	0.56	0.39	0.43	0.41	0.37	0.51
Industrials	0.85	0.57	0.60	0.81	0.56	1.00	0.66	0.83	0.49	0.37	0.69
Information Technology	0.72	0.27	0.37	0.51	0.39	0.66	1.00	0.54	0.51	0.16	0.53
Materials	0.74	0.47	0.67	0.69	0.43	0.83	0.54	1.00	0.39	0.30	0.62
Telecom	0.54	0.39	0.31	0.42	0.41	0.49	0.51	0.39	1.00	0.30	0.34
Utilities	0.26	0.43	0.43	0.33	0.37	0.37	0.16	0.30	0.30	1.00	0.44
Real Estate	0.74	0.55	0.37	0.72	0.51	0.69	0.53	0.62	0.34	0.44	1.00

Source: Bloomberg

Therefore, we prefer using the full time period rather than adding more stocks and less observations in the time-series dimension. Additionally, we dispose of the time-series observations from 2016 to 2021, which provides five more years of observations compared to the original paper. However, adding those extra observations are computationally expensive as we are using an expanding window in our cross-validation procedure that considers all the observations from 1957 for the training phase. This is especially true for the five ensemble neural networks models that took several days to be trained.

For 100 stocks, most of the models can be trained on one 4-core CPU computer in just a few hours. Moreover, the simplest models from OLS to Random Forest are trained using the *scikit-learn* machine learning library in Python, which is sufficient for low and middle scale machine learning applications. However, for the Gradient Boosting model, we prefer using the *CatBoost* library, which is very efficient and easy to implement as it shares the same Application Programming Interface (API) than *scikit-learn*. In fact, we prefer *CatBoost* over *XGBoost* as this last is more prone to overfitting when there are a only few observations. Indeed, the dataset's size for 100 stocks is only 13724 observations, which is a typical low-scale machine learning application. Nevertheless, we do not consider *scikit-learn* for Gradient Boosting, as it does not support the Huber loss unlike *Catboost*. Moreover, for our ensemble of 10 neural networks we opt for the *TensorFlow* library, especially the *Keras* package which provides a consistent API to construct and train neural network models in a highly efficient way. This is the results we obtain using the Pooled R-Squared evaluation metric for the parametric as well as the non-parametric models:

Table 5.5: Out-of-sample performance with 100 stocks for parametric models using the $\mathbf{R^2_{test}}$ evaluation metric

Model	OLS	OLS-3	PCR	PLS	ENet	GLM
$\boldsymbol{R^2_{test}}$	-0.44%	0.02%	0.13%	0.26%	0.15%	0.11%

Table 5.6: Out-of-sample performance with 1000 stocks for non-parametric models using the $\boldsymbol{R^2_{test}}$ evaluation metric

Model	RF	GB	NN1	NN2	NN3	NN4	NN5
$\boldsymbol{R^2_{test}}$	-0.06%	0.36%	0.14%	0.12%	-0.06 %	0.04%	0.11%

where R^2_{test} is the Pooled R-Squared computed on the test set

One thing to notice about these results is the very low R-Squared values, which are very close to zero but this is the typical results we obtain for predicting monthly excess returns. Nonetheless, the results obtained seem too optimistic compared to the original paper of Gu and al. (2020), as we assumed initially that the raw data incorporated the lagging of the regressors. As a result, we assumed that if the raw data did not contain any other data-preprocessing step, then there is a high probability that the lagging would have not been added too.

Nevertheless, for 1000 stocks it is no more possible to train all our models in a single machine, this is why we parallelize the training process into multiple machines. Each machine handles a few models and each one has an ensemble neural network as they are the most expensive to train. For that, we dispose of three virtual machines and two physical ones, with the virtual ones being less powerful than the physical ones. In fact, this is why it took many days to train the ensemble neural networks on the virtual machines. This is the set-up used to train our models along with the training time for each machine:

Table 5.7: Computational set-up with 1000 stocks

Machine	**Code**	**Command Line Option**	**Time**	**Status**
SL_1 (MV1)	memoire.py	run_ols, run_enet, run_glm, run_nn2	20 days	Unfinished for nn2
SL_2 (MV2)	memoire.py	run_ols3, run_rf, run_nn3	20 days	Finished
SL_3 (mac)	memoire.py	run_pls, run_gb, run_nn4	30 hours	Finished
SH_1	memoire.py	run_pcr, run_nn1, run_nn5	7 days	Unfinished for nn5

On each machine, each model was trained sequentially from the simplest to the most complex ones. Except ensemble neural network models and Generalized Linear model (GLM) with Group Lasso, all the other models were trained in just a few hours. The high computational cost for ensemble neural networks comes from the fact that the ensemble is composed of ten neural networks, which themselves are very computationally expensive to train. On the other hand, for GLM it comes from the spline series transformations that creates $(940 \times 3 + 1)$ extra regressors in addition to the already 940 existing ones. A more optimized implementation could have been achieved using the *gam* package in R, but we were running out of time. Specifically, here is the time it took to train all the models, except the NN2 and NN5 models which could have not been trained in a reasonable computational time:

Table 5.8: Computational time for each model with 1000 stocks

Model	OLS	OLS-3	PCR	PLS	ENet	GLM	RF	GB	NN1	NN3	NN4
Time	1h20	1h45	45min	40min	4h	18 days	1h15	3h	10 days	19 days	1 day

Finally, here is the out-of-sample performance for the parametric as well as the non-parametric models with the sample of 1000 stocks:

Table 5.9: Out-of-sample performance with 1000 stocks for parametric models using the $\mathbf{R^2_{test}}$ evaluation metric

Model	OLS	OLS-3	PCR	PLS	ENet	GLM
$\boldsymbol{R^2_{test}}$	-0.11%	-0.10%	0.09%	-0.25%	0.02%	0.06%

Table 5.10: Out-of-sample performance with 1000 stocks for non-parametric models using the R^2_{test} evaluation metric

Model	RF	GB	NN1	NN3	NN4
R^2_{test}	-4.22%	-1.19%	-0.21%	0.06%	0.08%

Most of the models have a negative R-Squared value, namely the residuals' variance is higher than the total variance in the excess returns which is clearly not satisfactory. Nevertheless, if we could have used the whole datasets, which is approximately thirty times larger than the one with 1000 stocks, even though there is the cross-sectional dependency, we might have obtained better results. In fact, the very negative results we obtain for RF and GB may come from the use of the *XGBoost* library, which is more prone to overfitting when there are not too many observations. Indeed, when computing the R-Squared on the training set after each split in the cross-validation, we noticed a very high value. The difference between the in-sample and out-of-sample performance is called the variance. Thus, if this difference is high, then this is characterized as a variance problem also known as overfitting. Therefore, increasing the training set size is the best way to reduce this variance problem without compromising the in-sample performance. In fact, another method for addressing overfitting is to specify a less complex model, namely a low number of trees and/or a low maximum depth. Nevertheless, this could lead to a too simple model, which will not be able to capture any complex patterns in the data. Consequently, this would lead to a low performance on the training set also known as underfitting. Otherwise, the other models achieve a similar result with the best one being the PCR model, which performs only less than three times worse than in the results of Gu and al. (2020).

Overall, we think that the performance is not so bad, as at least they are in the same range of values than the ones obtained by Gu and al. (2020). As a matter of fact, we do not have extreme negative values for all the other models except for RF and GB.

Conclusion

Predicting monthly equity excess returns is a complex task due to the inherently low signal-to-noise ratio that characterize equity returns, as well as the vast number of variables that influence stock returns. Machine learning models can be helpful in most applications as they provide efficient algorithms to process and learn complex relationships in the data. In fact, except Random Forest and Gradient Boosting which were prone to overfitting because of the small dataset, other machine learning models outperformed the standard OLS model. Our research delved into both parametric and non-parametric models, each with their own advantages and limitations.

Parametric models, including Principal Component Regression (PCR), Partial Least Squares (PLS), Elastic Net (ENet), and Generalized Linear Model (GLM), were found to be beneficial due to their inherent interpretability and simplicity. They allow a clear understanding of how different factors influence equity returns. On the other hand, non-parametric models, such as Random Forest (RF), Gradient Boosted Regression Trees (GBRT), and Ensemble Neural Networks (NN1-NN5), did not show high predictive performance, but as we argued they clearly overfitted the training data as the dataset was small. In larger datasets, they show an impressive ability to capture complex and non-linear relationships within the data despite the high computational cost associated with it. Therefore, they offer the advantage of modelling complex phenomena without the necessity to specify a functional form for the regressors. In terms of performance, we noted that the selection of appropriate hyperparameters, especially the Bayesian hyperparameter search played a crucial role and enabled to save a lot of time.

In the context of predicting excess equity returns, machine learning methods can provide useful tools to operate in the Big data area in which we are nowadays. However, in this paper we focused on the monthly period which led to fewer observations. In fact, machine learning models perform well in big datasets, which may explain the low performance we obtained. In fact, for higher frequency data machine learning models may have yield better results than the ones we obtained. This is why in recent years, an area of machine learning called deep learning has come up with neural networks architectures and methods to handle large datasets efficiently.

Future research in this field could focus on incorporating unstructured data, such as textual data or satellite images. In fact, one promising area in deep learning called Natural Language Processing (NLP) tries to decrypt the human language, which can be used for instance to analyze the market sentiment on social media. Furthermore, to remedy the curse of dimensionality, quantum algorithms have drawn the attention of the academic research in recent years, leading to an area of machine learning called quantum machine learning. Nevertheless, diving into deeper models leads to a lack of interpretability, which is a major issue and has drawn the attention of the regulators. For that, efforts to make these models more interpretable would be beneficial not only for the regulators but for every stakeholder. In fact, it can help the investor to gain deeper insights into the underlying factors influencing equity returns to manage risks properly, and even improving the predictive performance.

In conclusion, while predicting monthly equity excess returns remains a daunting task, our research reaffirms that machine learning, when applied thoughtfully and cautiously, can be a valuable tool in this context.

List of tables

List of figures

References

Scientific papers:

Carhart. (1997). On Mutual funds performance. *The journal of finance.*

Flachaire, E., Hacheme, G., Hué, S., & Laurent, S. (2021). GAM(L)A: An econometric models for interpretable Machine Learning.

French, F. a. (1993). Common factors in the returns on stocks and bonds. *Journal of financial economics* .

Gu, .a. (2020). Empirical asset pricing via machine learning. *The review of financial studies*.

Welch, Goyal. (2008). A Comprehensive Look at The Empirical Performance of Equity Premium Prediction. *The Review of Financial Studies*.

Books:

Hilpisch, Y. (2018). *Python for Finance: Mastering Data-Driven Finance.* O'Reilly Media.

Jansen, S. (2020). *Machine learning for algorithmic trading.* Packt.

Online courses:

Ng, A. (2013). *Machine learning specialization.* Coursera.

Ng, A. (2016.). *Deep learning specialization*. Coursera.

Webistes:

https://towardsdatascience.com/bayesian-optimization-concept-explained-in-layman-terms-1d2bcdeaf12f

https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502 https://towardsdatascience.com/xgboost-extreme-gradient-boosting-how-to-improve-on-regular-gradient-boosting-5c6acf66c70a

https://towardsdatascience.com/a-practical-introduction-to-grid-search-random-search-and-bayes-search-d5580b1d941d

https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4

https://medium.com/almabetter/catboost-the-fastest-algorithm-c21d44f8b990

Package documentations:

https://keras.io

https://catboost.ai

https://pandas.pydata.org/pandas-docs/stable/index.html

https://scikit-learn.org/stable/

https://xgboost.readthedocs.io/en/stable/

https://www.tensorflow.org/?hl=fr

https://keras.io