

# THE MACHINE LEARNING LIBRARY

---

**Mathematical derivation of 20 algorithms and  
their implementation with only Python**

---



*Imanol Navarro Martínez*

## **Abstract**

This book aims to show the progress of a graduate with a Master's degree in theoretical physics learning in depth about the broad field of Machine Learning. The book will have the following structure: a first chapter in which I will introduce several concepts of importance in this field that I will resort to throughout the book. Then, a few chapters in which I will work on twenty of the most relevant algorithms in Machine Learning (from linear regressions to complex neural networks). Each algorithm section will include a thorough mathematical development of the equations involved and my implementation in Python. Lastly, a final chapter in which I will talk about some of the most common metrics to evaluate the performance of a model. The goal of the book is to increase my knowledge and proficiency in machine learning while also serving as a bibliographic resource for anyone looking to learn more about the field or a particular method. This book is not meant to rival library implementations such as Scikitlearn, as my code is not optimized to run with such performance in terms of speed.

## **Acknowledgement**

I want to show all my gratitude to my love Iratxe for supporting me in this decision to carry out the project, encouraging me to pursue my goals, and always putting all her faith in me, giving me words of encouragement in those moments when I doubted if carrying out the project was a wise choice. Also, special thanks to my dear friend Jaime, who introduced me to the world of Machine Learning in one of our conversations when I was unsure about what path to take.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Training data and testing data . . . . .	7
1.2	Bias-variance tradeoff (overfitting and underfitting) . . . . .	8
1.3	Cross-validation . . . . .	11
1.4	Normalization of the data . . . . .	12
1.5	Supervised and Unsupervised learning . . . . .	13
<b>2</b>	<b>Regularized Linear Regression Algorithms</b>	<b>14</b>
2.1	Ordinary Least Squares [Supervised] . . . . .	17
2.1.1	Mathematical derivation . . . . .	17
2.1.2	Algorithm implementation . . . . .	19
2.2	Ridge regression [Supervised] . . . . .	23
2.2.1	Context . . . . .	23
2.2.2	Mathematical derivation . . . . .	26
2.2.3	Algorithm implementation . . . . .	28
2.3	Lasso regression [Supervised] . . . . .	34
2.3.1	Context . . . . .	34
2.3.2	Mathematical derivation . . . . .	36
2.3.3	Algorithm implementation . . . . .	41
2.4	Elastic-Net [Supervised] . . . . .	47
2.4.1	Mathematical derivation . . . . .	47
2.4.2	Algorithm implementation . . . . .	49
2.5	Results . . . . .	53
<b>3</b>	<b>Other Regression Algorithms</b>	<b>54</b>
3.1	Adaptive regression through hinges [Supervised] . . . . .	56
3.1.1	Mathematical derivation . . . . .	56

3.1.2	Algorithm implementation . . . . .	60
3.2	Locally weighted linear regression [Supervised] . . . . .	73
3.2.1	Mathematical derivation . . . . .	73
3.2.2	Algorithm implementation . . . . .	75
3.3	Bayesian linear regression [Supervised] . . . . .	81
3.3.1	Mathematical derivation . . . . .	81
3.3.2	Algorithm implementation . . . . .	89
<b>4</b>	<b>Association Rule Learning Algorithms</b>	<b>95</b>
4.1	Apriori Algorithm [Unsupervised] . . . . .	95
4.1.1	Mathematical derivation . . . . .	95
4.1.2	Algorithm implementation . . . . .	98
<b>5</b>	<b>Classification Algorithms</b>	<b>104</b>
5.1	Logistic regression [Supervised] . . . . .	106
5.1.1	Mathematical derivation . . . . .	106
5.1.2	Algorithm implementation . . . . .	112
5.2	K-nearest neighbors [Supervised] . . . . .	121
5.2.1	Mathematical derivation . . . . .	121
5.2.2	Algorithm implementation . . . . .	123
5.3	LVQ [Supervised] . . . . .	131
5.3.1	Mathematical derivation . . . . .	131
5.3.2	Algorithm implementation . . . . .	133
5.4	SVM [Supervised] . . . . .	145
5.4.1	Mathematical derivation . . . . .	145
5.4.2	Algorithm implementation . . . . .	157
5.5	K-means clustering [Unsupervised] . . . . .	179
5.5.1	Mathematical derivation . . . . .	179
5.5.2	Algorithm implementation . . . . .	180
<b>6</b>	<b>Dimensionality Reduction Algorithms</b>	<b>187</b>
6.1	Principal component analysis [Unsupervised] . . . . .	187
6.1.1	Mathematical derivation . . . . .	187
6.1.2	Algorithm implementation . . . . .	192
<b>7</b>	<b>Classification Trees</b>	<b>196</b>

7.1	CART (Classification and regression trees) [Supervised] . . . . .	197
7.1.1	Mathematical derivation . . . . .	197
7.1.2	Algorithm implementation . . . . .	201
7.2	Random forest [Supervised] . . . . .	214
7.2.1	Mathematical derivation . . . . .	214
7.2.2	Algorithm implementation . . . . .	215
7.3	Gradient boosted classification trees [Supervised] . . . . .	219
7.3.1	Mathematical derivation . . . . .	219
7.3.2	Algorithm implementation . . . . .	226
<b>8</b>	<b>Image Recognition Algorithms</b>	<b>235</b>
8.1	Eigenfaces [Supervised] . . . . .	240
8.1.1	Mathematical derivation . . . . .	240
8.1.2	Algorithm implementation . . . . .	241
8.2	Neural Networks [Supervised] . . . . .	246
8.2.1	Mathematical derivation . . . . .	246
8.2.2	Algorithm implementation . . . . .	250
8.3	CNN [Supervised] . . . . .	258
8.3.1	Mathematical derivation . . . . .	258
8.3.2	Algorithm implementation . . . . .	274
<b>9</b>	<b>Evaluation metrics</b>	<b>288</b>
9.1	Regression metrics . . . . .	288
9.1.1	Mean Squared Error (MSE) . . . . .	288
9.1.2	Root Mean Squared Error (RMSE) . . . . .	289
9.1.3	Mean Absolute Error (MAE) . . . . .	289
9.1.4	Root Mean Squared Logarithmic Error (RMSLE) . . . . .	290
9.1.5	R-Squared factor ( $R^2$ ) . . . . .	291
9.1.6	Adjusted R-Squared factor ( $\bar{R}^2$ ) . . . . .	291
9.2	Classification metrics . . . . .	292
9.2.1	Accuracy (ACC) . . . . .	292
9.2.2	Logarithmic Loss/ Cross-Entropy Loss (LL) . . . . .	292
9.2.3	ROC-AUC . . . . .	293
9.2.4	Classification report (CR) . . . . .	298

9.3 Multiclass classification . . . . .	299
9.3.1 One vs all . . . . .	299
9.3.2 One vs One . . . . .	300
<b>Bibliography</b>	<b>302</b>

# Chapter 1

## Introduction

Before working on the algorithms, it is essential to introduce certain concepts that are very recurrent in Machine Learning. We will divide this chapter into a few sections where we will talk about topics such as *bias-variance tradeoff*, *cross-validation* or the *normalization* of the data, among others; topics that we need to understand before learning about the different algorithms.

### 1.1 Training data and testing data

One of the things that we need to understand is how a particular data collection is split in order to evaluate the performance of a model.

- **Training data:** When we have a data collection, we will use a percentage of it to train our algorithm. This data will be used as input to adapt our algorithm's parameters so that the model's predictions on this training data and the actual outcomes converge to the desired extent. Here, we indicate that we want the outcome and predictions to converge to the desired extent because of a concept called *overfitting*, which will be explained in the following section.
- **Testing data:** The ultimate goal of a machine learning algorithm is to make predictions on unseen data, i.e., data on which the model has not been trained. Hence, another percentage of our data collection will be used to test our model, quantifying how much our prediction over this data deviates from the actual values.

## 1.2 Bias-variance tradeoff (overfitting and underfitting)

[38] Suppose we have a collection of training data composed of a variable  $x$  and outcome  $y$  associated with it, i.e., a collection  $\{x_i, y_i\}_{i=1}^N$ , which we will use to train the model. Now, we will assume that after training our model, we predict an outcome  $\hat{y}_i$  for each point. We can quantify how the model adapts to the training data with the following quantities:

- **Bias:** The bias of a data point is the difference between the average prediction of our model and the outcome of that data point. If we want to generalize it to our model, we will average the bias for every training point and get an expression for the total bias:

$$b = \sum_{i=1}^N \frac{\langle \hat{y} \rangle - y_i}{N} \quad (1.1)$$

If we were to make a regression such that our curve perfectly fits all the training points, we would get a bias equal to zero. This is because we would have  $\langle \hat{y} \rangle = (\hat{y}_1 + \hat{y}_2 + \dots + \hat{y}_N)/N = (y_1 + y_2 + \dots + y_N)/N$ .

- **Variance:** The variance tells us how spread the predictions of our model are. It quantifies how much the predictions for the data points deviate from the average prediction.

$$\sigma^2 = \sum_{i=1}^N \frac{(\hat{y}_i - \langle \hat{y} \rangle)^2}{N} \quad \text{where} \quad \langle \hat{y} \rangle = \sum_{i=1}^N \frac{\hat{y}_i}{N} \quad (1.2)$$

Now, we want to use the model to make predictions on unseen testing data, and for that, we want to estimate the magnitude of the error involved. Suppose we have another collection of testing data  $\{x_j, y_j\}_{j=1}^M$ .

- **Mean squared error:** This quantity tells us the average of the square of the deviation between the prediction of the model on unseen data and the actual outcomes:

$$MSE = \frac{\sum_{j=1}^M (\hat{y}_j - y_j)^2}{M} \quad (1.3)$$

A critical characteristic of the MSE is that we can write it in terms of the bias and variance of the model:

$$MSE = b^2 + \sigma^2 + \epsilon^2 \quad (1.4)$$

As we have mentioned before, the bias will be the error introduced because of making assumptions on our model, i.e., simplifying it so it does not precisely adjust to the training data. On the other hand, the variance is an error that will punish us whenever we make a model with a high dispersion or spreading on predictions since it will most likely lead to errors when predicting new data. Last,  $\epsilon^2$  is called the irreducible error; we cannot eliminate this term due to the noise in our data; it does not matter how good our model is.

- **Bias-variance tradeoff:** We must understand that bias and variance go hand in hand, contributing to the total error. If we make our model fit the training data too well and hence have a low bias, this will likely mean that the prediction will have a high dispersion, leading to a significant error when predicting unseen data, i.e., we will have an **overfit** model. On the other hand, if the model does not fit the training data too well, i.e., it is a way too simple model, it will have a low variance, but the bias will be high instead, leading to a high error too. This is called an **underfit** model. The idea is to find a well-balanced model with a low bias and variance that captures the structure of the data, leading to a low MSE when predicting unseen data.

To see how all these concepts work together, I created a dispersion of points around the  $\sin(x)$  function. After splitting the data into training (green points) and testing (blue points), I chose three simple models to predict the data: first, some splines connecting all the training points; second, the line  $y=1$ ; and last, the  $\sin(x)$  function. Computing these three models' bias, variance, and MSE involved some coding but not a machine learning algorithm.

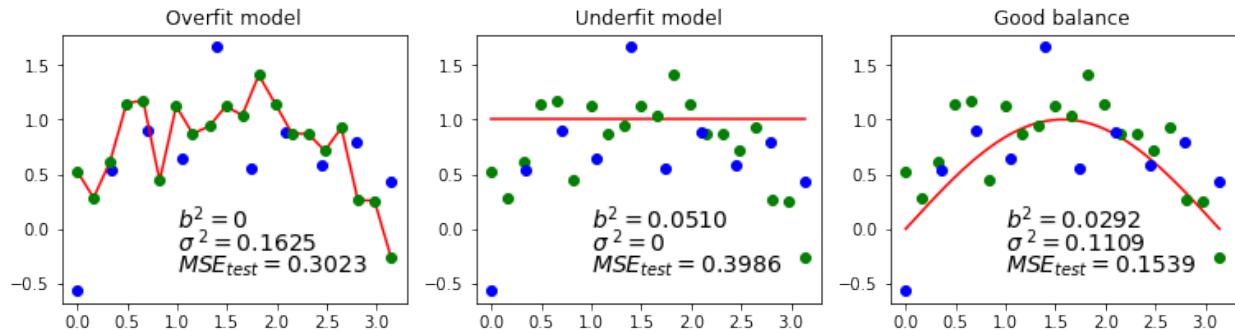


Figure 1.1: Example of an overfit, underfit, and well-balanced model of the  $\sin(x)$  function.

As shown in the graph above, the splines adjust perfectly to the training data, leading to a bias equal to zero and a high dispersion, ultimately leading to a high MSE. The horizontal line leads to a high bias and a variance equal to zero, resulting in a high MSE too. On the other hand, the  $\sin(x)$  function keeps a good balance between bias and variance, leading to an MSE twice smaller than the other two models. I also plotted the results of the bias, variance, and MSE for these three models and fitted some exponential functions on them to show the tendencies of the quantities, which I did after learning the OLS algorithm. The values of the x-axis quantifying the complexity of the model are merely for illustration purposes. I recommend returning to this and reading the content of the green box after reading the OLS section to understand how these exponential curves were fitted to the data. It is also necessary to understand how the gradient descent method works. It helps us to iteratively obtain the variable value for which a function is minimal. Gradient descent appears in other sections of the book, as I used it on other algorithms.

This involved writing the general expression for the exponential:  $e^{a+b x}$ , then writing the expression for the MSE ignoring the constant as  $MSE = \sum_i^N (e^{a+b x_i} - y_i)^2$ , computing the derivative of the MSE with respect to  $a$  and  $b$  and finally, writing an iteration for the parameters using gradient descent to find which value of these parameters led to the minimum of the MSE.

$$a^{(k+1)} = a^{(k)} - 2\sigma \sum_i^N (e^{a^{(k)}+b^{(k)} x_i} - y_i) e^{a^{(k)}+b^{(k)} x_i}$$

$$b^{(k+1)} = b^{(k)} - 2\sigma \sum_i^N (e^{a^{(k)}+b^{(k)} x_i} - y_i) e^{a^{(k)}+b^{(k)} x_i} x_i$$

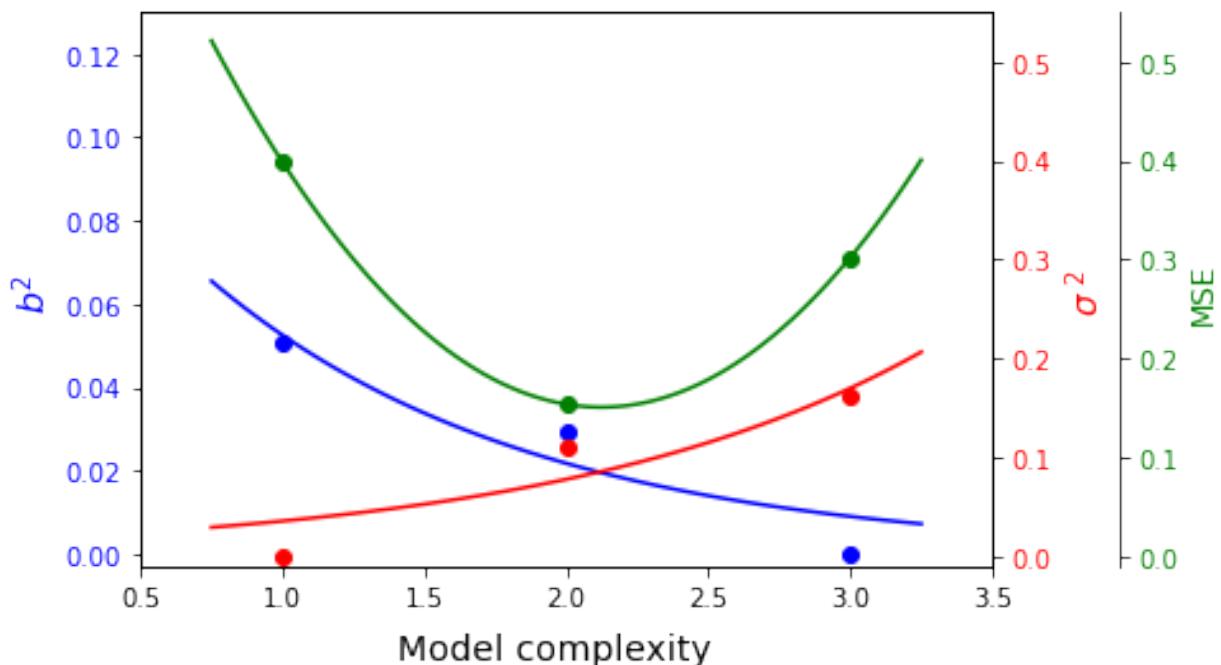


Figure 1.2: Tendency of the bias, variance, and MSE in terms of the model complexity.

## 1.3 Cross-validation

Whenever we have some data, we have to test different models to see which is better to make predictions, and for this, we have to find the best configuration for each specific model or algorithm we try. Among other things, whenever we split this data into training and testing sets to test a specific model, we have to make sure our results are independent of this choice because we do not want to get either lucky or unlucky with the choice and get biased results, we want to know the performance or the error of the model on average. For this, we introduce the concept of **cross-validation**. Cross-validation splits the data into training and testing data randomly multiple times. For each of these splits, we perform a prediction on the testing data after training the model on the training data and then average the results of all the different splits. This way, we can make the result independent of our choice of training data and get an idea of how well our algorithm performs. Cross-validation is the method I will employ throughout the book to get an estimate of the performance of my models.

There is a particular case of cross-validation, which is called **K-fold cross-validation**. Here, we divide the data set into K folds, performing different runs using each of the folds as the testing set and all the remaining folds as the training set. There is another thing worth mentioning: suppose we have a dataset with 100 data points that we want to split as 80% training data and 20% testing data. We will have  $100!/(80!20!)$  combinations which is of the order of  $10^{20}$ . An iteration number of  $10^5$  for any complex model will already be slow. Hence, if we want to compare two different models and we split the data randomly, the likelihood of the models having a common split will be low. To fix this issue and compare models fairly, we can use *random states*, which are random shuffles of the data labeled by an integer. The advantage of the *random states* is that we will get the same shuffle every time we set the same integer, allowing us to reproduce results or compare models over the same shuffles (we can reproduce the training/testing splits).

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Run 1	Testing	Training	Training	Training	Training
Run 2	Training	Testing	Training	Training	Training
Run 3	Training	Training	Testing	Training	Training
Run 4	Training	Training	Training	Testing	Training
Run 5	Training	Training	Training	Training	Testing

Table 1.1: 5-fold cross-validation illustration.

## 1.4 Normalization of the data

Another essential concept is that before working with data, it is a good practice to normalize it; this means centralizing it around the zeros of the axis and scaling down the feature intervals. The main reason for doing this is that we may have features or variables in our dataset with different scales, and we want all of them to have the same relevance. Normalizing the data sets a common scale for the different variables while keeping the essence of the data. In some cases where the algorithm uses the Euclidean distances on the variable space, this is a required step (the RBF kernel on the SVM algorithm, for example). It is essential to split the data into the training and testing datasets before normalizing, as we want to avoid any information leakage from the testing data into the training data. Normalizing also simplifies much of the computation since the new averages of the variables over the training data are zero. To normalize a variable  $X$ , we need to compute its average value and standard deviation over the training data and then update the variable for all the data with the equation:

$$X_i^{norm} = \frac{X_i - \langle X \rangle}{\sigma} \quad \text{where } \sigma = \sqrt{\frac{\sum_{i=1}^N (X_i - \langle X \rangle)^2}{N}} \quad (1.5)$$

A couple more things worth mentioning is that if we have two variables,  $X$  and  $Z$ , these need to be normalized separately. Also, to normalize the testing data, we need to use the mean and standard deviation of the variables over the training data so it does not scale differently. The function I will be using to normalize the data is the following:

```
[1]: def normalization(X_train,Y_train,X_test,Y_test):

    X_avg = np.mean(X_train, axis=0)
    X_std = np.std(X_train, axis=0)
    Y_avg = np.mean(Y_train, axis=0)
    Y_std = np.std(Y_train, axis=0)

    X_train_norm = (X_train - X_avg) / X_std
    Y_train_norm = (Y_train - Y_avg) / Y_std
    X_test_norm = (X_test - X_avg) / X_std
    Y_test_norm = (Y_test - Y_avg) / Y_std

    return(X_train_norm, Y_train_norm,
           X_test_norm, Y_test_norm)
```

## 1.5 Supervised and Unsupervised learning

Lastly, it is essential to introduce the concepts of **supervised** and **unsupervised** learning, which are the groups that most algorithms classify into.

- **Supervised** learning refers to those algorithms that work with labeled data collections, i.e., data with an outcome. These algorithms are based on the knowledge of the outcomes, minimizing the difference between the predictions and the real values.
- On the other hand, **unsupervised** learning refers to the algorithms that work with unlabeled data collections, i.e., data without an outcome. The only information they have is the variable values of each data point. An example of an unsupervised algorithm is the k-means clustering model, which identifies point clusters and associates them together without any information about their outcomes, just based on the features.

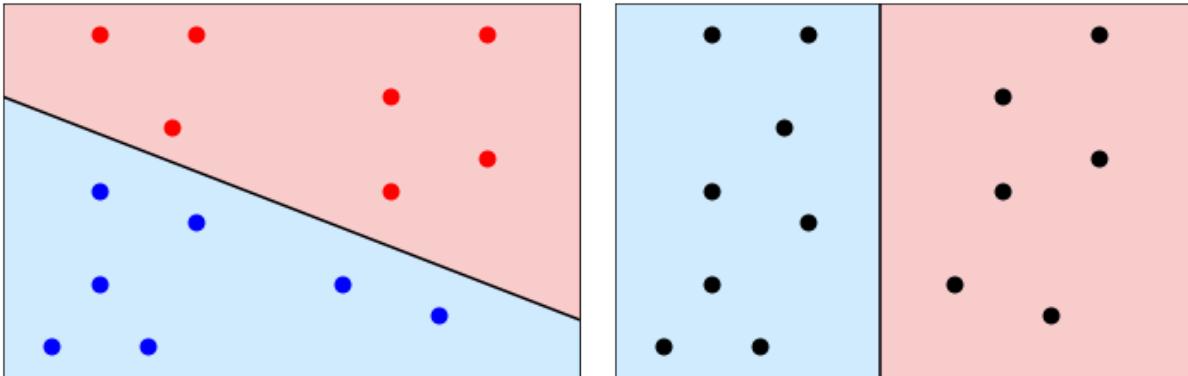


Figure 1.3: Visualization of a classification supervised model (left) and unsupervised model (right).

As shown in the previous graph, the difference between working with labeled and unlabeled data can be substantial. On the left plot, we have a supervised model that knows the label or outcome (red or blue) of the different data points and classifies the space based on that. On the right plot, however, since we do not have any information about the labels, dividing the data by clusters based on proximity would lead to a completely different classification of the space. In some cases, we will not have access to the labels of the data collection and will be forced to use unsupervised models. Another possibility is that an unsupervised model will classify the data wrong (as shown in the graph above). However, ignoring the labels and using an unsupervised algorithm will sometimes lead to better predictions. It all depends on the data collection we are working with.

# Chapter 2

## Regularized Linear Regression Algorithms

Next, I will introduce a self-made example (Table 2.1) that I will work with on the algorithms of this chapter: OLS, Ridge, Lasso, and Elastic-net. There is a reason behind all the variables in this dataset, which have been explicitly picked to see the differences between the first four algorithms. I will briefly mention the algorithms' strengths to understand the reasoning behind the variable choice.

- **Ridge** is useful when we have collinear or multicollinear variables; it excels in picking up these redundancies. This is why  $Total\ study\ time \simeq Study\ time\ at\ univ. + Study\ time\ at\ home$  and also  $Stress\ level \simeq 10 - 0.3 * Total\ study\ time$ . It is also worth mentioning that there is no perfect collinearity between those variables, and this is because the OLS model stops working when this happens (as we will see, this model involves computing the inverse of a matrix, which is not possible when there is collinearity, due to at least one column of the matrix being a linear combination of the others and hence the rank of the matrix being lower than its dimensions).
- **Lasso** regression is good at identifying those variables that do not impact the outcome and are irrelevant, so I included the *Age* and *Height* variables.
- The **Elastic Net** is a combination of Ridge and Lasso.

To illustrate the strengths I just mentioned, I will form four data collections out of the data set. The first one contains only the variable: *Total study time*, the second one contains the variables: *Total study time*, *Study time at univ.*, *Study time at home* and *Stress levels*. The third collection will be formed by: *Total study time*, *Age* and *Height* and the last one by all the variables. I will test OLS, Ridge, and Lasso on all four data collections and Elastic-net only on the fourth one.

We will use regular cross-validation to test the models, picking 80% of the data as training data and the remaining 20% as testing data. The models will be tested by averaging over the first 2,000 random states.

Student #	Total study time (hours)	Study time at university (hours)	Study time at home (hours)	Stress levels (0-10)	Age	Height (cm)	Score
1	0	0	0	10	19	165	3.3
2	1	1	0	9.7	20	161	2.3
3	1	0	1	9.6	20	187	3.3
4	2	1	1	9.4	22	162	5.0
5	3	2	1	9.1	21	170	5.1
6	4	2	2	8.8	21	181	2.7
7	4	1	3	8.7	19	180	5.0
8	5	4	1	8.5	18	185	4.3
9	6	3	3	8.2	18	177	4.8
10	6	4	2	8.1	21	165	5.8
11	7	4	3	7.9	20	171	5.9
12	8	8	0	7.6	20	182	7.7
13	9	3	6	7.4	22	166	7.4
14	9	4	5	7.2	21	170	7.2
15	10	7	3	7.0	22	173	8.0
16	10	5	5	7.1	19	162	9.6
17	11	5	6	6.7	18	164	10
18	12	2	10	6.4	20	176	9.1
19	13	5	8	6.1	20	182	9.6
20	14	7	7	5.8	19	191	8.5
21	15	4	10	5.5	19	162	6.8
22	16	8	8	5.2	20	154	10
23	18	9	8	4.6	21	165	10
24	19	9	9	4.3	22	178	8.6
25	20	10	10	4.0	20	190	10
26	21	15	6	3.7	19	157	7.9
27	22	16	6	3.3	18	160	9.4
28	23	12	10	3.0	19	178	9.1
29	24	12	12	2.8	21	168	10
30	25	17	6	2.5	21	165	10

Table 2.1: Self-made dataset describing the scores obtained in a test in terms of different variables.

I will also show a regular linear regression on the different variable and outcome pairs of the data collection to see the dependence of the outcomes on these variables, together with another example of overfitting.

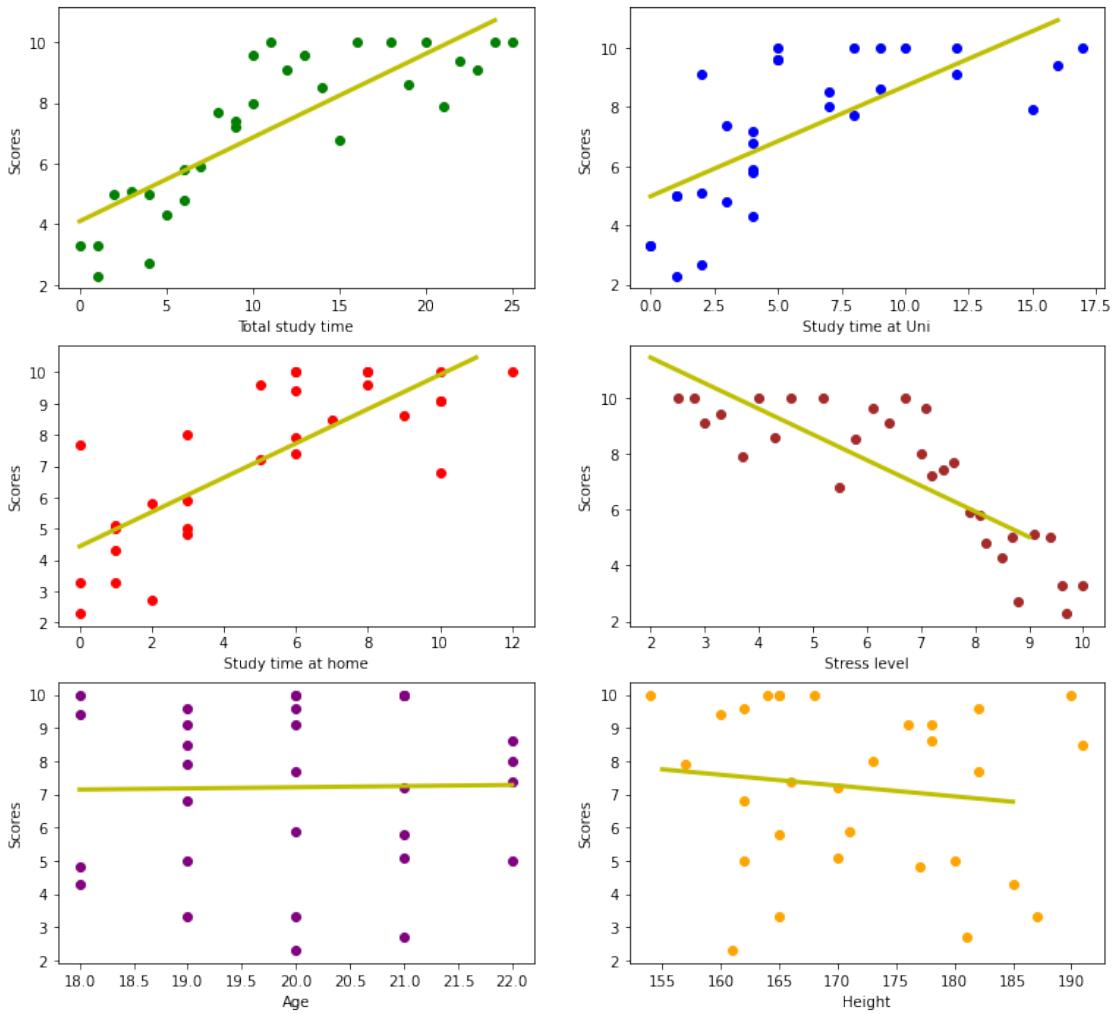


Figure 2.1: OLS regression on all the possible variable and outcome pairs.

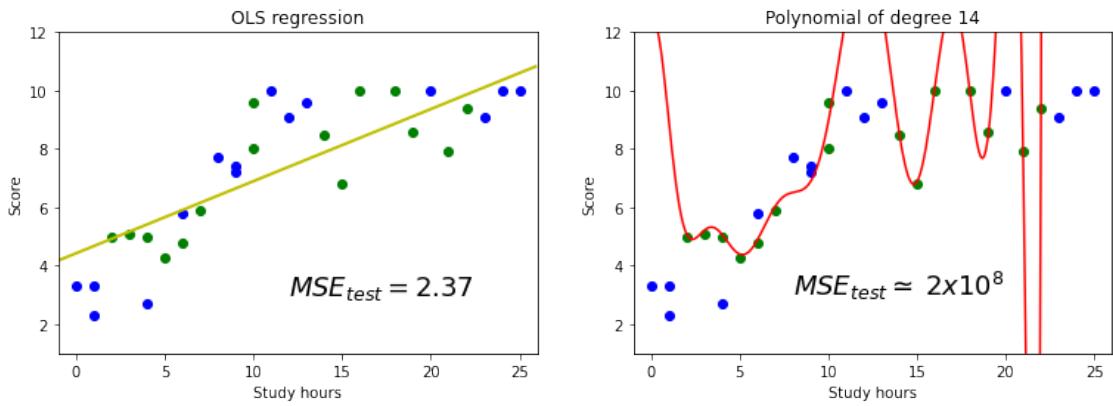


Figure 2.2: Example of overfitting: linear regression vs polynomial of degree 14.

## 2.1 Ordinary Least Squares [Supervised]

### 2.1.1 Mathematical derivation

[51] We start by recovering the mean squared error equation:

$$MSE = \frac{\sum_{i=1}^N (\hat{y}_i - y_i)^2}{N} \quad (2.1)$$

The OLS algorithm is a type of linear regression; this is one of the simplest we will work with. This model aims to find a line that fits the data and allows us to make predictions about different points of the real line for which we do not know the outcome. This method is extended to D dimensions, i.e., it will allow us to obtain a hyperplane of D-1 dimensions for a dataset with D-1 variables and an outcome. To get this hyperplane, we will require the MSE to be minimal.

Suppose we have a discrete dataset given by:  $\{\mathbf{X}_i, y_i\}_{i=1}^N$ . Following the example I have shown in the introduction of this chapter, the index  $i$  identifies the different students,  $y_i$  is the result scored by the  $i^{th}$  student on the test and  $\mathbf{X}_i = [X_{i,1}, X_{i,2}, \dots, X_{i,D-1}]$  is a vector that saves information about each of the students. We can think of  $X_{i,1}$  as the number of hours the  $i^{th}$  student studied before the test,  $X_{i,4}$  as the stress levels of the  $i^{th}$  student and so on.

Considering a two-dimensional linear regression of a variable and outcome pair, the problem is reduced to finding the parameters  $a$  and  $b$  of the line equation:  $y = ax + b$ . This way, we can predict the outcome of different points with the expression:  $\hat{y}_i = ax_i + b$ . But as I already mentioned, the idea is to generalize this algorithm to D dimensions, so we have to modify our two-dimensional linear equation to a D-dimensional linear equation:

$$\hat{y}_i = \beta_1 X_{i,1} + \beta_2 X_{i,2} + \dots + \beta_{D-1} X_{i,D-1} + C \quad (2.2)$$

Moreover, by introducing equation 2.2 in equation 2.1 and removing the constant N, the quantity that we want to minimize is:

$$S(\beta) = \sum_{i=1}^N (y_i - [\beta_1 X_{i,1} + \beta_2 X_{i,2} + \dots + \beta_{D-1} X_{i,D-1} + C])^2 \quad (2.3)$$

It is often helpful to use matrix and vector notation:

$$X = \begin{pmatrix} X_{1,1} & X_{1,2} & \dots & X_{1,D-1} \\ X_{2,1} & X_{2,2} & \dots & X_{2,D-1} \\ \vdots & \vdots & \ddots & \vdots \\ X_{N,1} & X_{N,2} & \dots & X_{N,D-1} \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{D-1} \end{pmatrix} \quad \hat{y} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{pmatrix} \quad (2.4)$$

$N$  is the number of data points and  $D - 1$  is the number of variables ( $D$  are the dimensions of the data space, i.e., variables and outcome combined). In general, the standard

procedure is to add a column of ones to the matrix  $X$ ; this way, we can introduce our constant  $C$  into the  $\beta$  vector as  $\beta_0$  so we can write equation 2.2 more compactly as  $\hat{y} = X\beta$ , where:

$$X = \begin{pmatrix} 1 & X_{1,1} & X_{1,2} & \cdots & X_{1,D-1} \\ 1 & X_{2,1} & X_{2,2} & \cdots & X_{2,D-1} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & X_{N,1} & X_{N,2} & \cdots & X_{N,D-1} \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{D-1} \end{pmatrix} \quad \hat{y} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{pmatrix} \quad (2.5)$$

Now, all that is left to do is to find the vector  $\beta$  that minimizes the quantity  $S(\beta)$  (minimizing  $S(\beta)$  is equivalent to minimizing the MSE since they are proportional). After finding the vector  $\beta$ , we have the equation for our hyperplane:  $\hat{y}_i = \beta_1 X_{i,1} + \beta_2 X_{i,2} + \cdots + \beta_{D-1} X_{i,D-1} + \beta_0$  which we can use to make predictions for any values. To obtain the value of  $\beta$  so that  $S(\beta)$  is minimal, we simply compute the partial derivative:

$$D_a = \frac{\partial S(\beta)}{\partial \beta_a} = -2 \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right) \quad (2.6)$$

We have an expression like this one for every possible value of the index  $a$ . To obtain the value of  $\beta$  for which  $S(\beta)$  is minimum, we have to make it equal to zero; we obtain:

$$\sum_{i=1}^N X_{i,a} y_i = \sum_{i=1}^N \sum_{j=0}^{D-1} X_{i,a} X_{i,j} \beta_j \quad (2.7)$$

If we write some of the terms:

$$\begin{aligned} X_{1,1} y_1 + X_{2,1} y_2 + \cdots + X_{N,1} y_N &= X_{1,1} (\beta_0 + X_{1,1} \beta_1 + X_{1,2} \beta_2 + \cdots + X_{1,D-1} \beta_{D-1}) \\ &\quad + X_{2,1} (\beta_0 + X_{2,1} \beta_1 + X_{2,2} \beta_2 + \cdots + X_{2,D-1} \beta_{D-1}) + \cdots + \\ &\quad X_{N,1} (\beta_0 + X_{N,1} \beta_1 + X_{N,2} \beta_2 + \cdots + X_{N,D-1} \beta_{D-1}) \end{aligned}$$

$$\begin{aligned} X_{1,2} y_1 + X_{2,2} y_2 + \cdots + X_{N,2} y_N &= X_{1,2} (\beta_0 + X_{1,1} \beta_1 + X_{1,2} \beta_2 + \cdots + X_{1,D-1} \beta_{D-1}) \\ &\quad + X_{2,2} (\beta_0 + X_{2,1} \beta_1 + X_{2,2} \beta_2 + \cdots + X_{2,D-1} \beta_{D-1}) + \cdots + \\ &\quad X_{N,2} (\beta_0 + X_{N,1} \beta_1 + X_{N,2} \beta_2 + \cdots + X_{N,D-1} \beta_{D-1}) \end{aligned}$$

This can be easily read and generalized as:

$$X^T y = (X^T X) \beta \quad (2.8)$$

With a matrix maneuver, we obtain that:

$$\beta = (X^T X)^{-1} X^T y \quad (2.9)$$

The element  $\beta_0$  (which is the constant of our hyperplane) is usually referred to as *intercept* while  $\beta_i$  are called coefficients. To summarize, we have just obtained the  $\beta$  parameters that define the hyperplane that fits our data collection so that the error (MSE) is minimal, allowing us to make predictions on the whole hyperspace.

## 2.1.2 Algorithm implementation

In this first part of the code, I built the Ordinary Least Squares class, which contains all the tools necessary to apply this model. I will use this class in the next steps of the subsection. Next, I will proceed to explain the elements involved.

```
[1]: import matplotlib.pyplot as plt
import numpy as np

class OLS():
    def __init__(self,X,Y):
        self.X=X
        self.Y=np.reshape(Y,(len(Y),1))
        self.beta = np.zeros((len(X[0])+1,1))
```

This first part of the code contains a function that inserts a column of ones into the variable storing the data. This is achieved by creating a matrix full of ones of dimensions  $(M, N+1)$  where  $(M, N)$  are the original dimensions of the data and then inserting the original matrix into the new one.

```
def extend(self,X):
    M = len(X)
    N = len(X[0])
    extended_X = np.ones((M,N+1))
    extended_X[:,1:] = X
    return(extended_X)
```

Next, we have the *train model* function, which performs all the necessary operations to calculate the beta coefficients.

```
def train_model(self,X,Y):
    # We add the column of 1s to X
    X = self.extend(self.X)
    # We compute the different products involved in Beta
    X_T = np.transpose(X)
    product_1=np.linalg.inv(np.dot(X_T,X))
    product_2=np.dot(X_T,self.Y)
    #We compute the Beta coefficients
    self.beta=np.dot(product_1,product_2)
    return(self.beta)
```

Finally, we have the *coefficients* function, which checks if the array storing the beta coefficients has been modified with respect to the initialization values. If it has, it simply returns the array, and if it has not been changed, it updates it by executing the *train model* function. On the other hand, we have a function that uses the value of the beta

coefficients to make predictions on new data and another one that calculates the value of the MSE of the predictions made.

```
def coefficients(self):
    if np.sum(self.beta) == 0:
        self.beta = self.train_model(self.X, self.Y)
    return(np.transpose(self.beta))

def prediction(self, X_test):
    if np.sum(self.beta) == 0:
        self.beta = self.train_model(self.X, self.Y)
    X_test = self.extend(X_test)
    prediction = np.dot(X_test, self.beta)
    return(prediction)

def MSE(self, X_test, Y_test):
    Y_test = np.reshape(Y_test, (len(Y_test), 1))
    error_array = self.prediction(X_test) - Y_test
    MSE = np.sum(error_array*error_array)/len(Y_test)
    return(MSE)
```

As mentioned in the introduction, I created a dataset storing the scores of a test in terms of other variables to test the strengths and weaknesses of the first four algorithms. Since Ridge and Lasso converge to the OLS algorithm when their respective hyperparameters are set to zero, we will see how the OLS performs in those chapters.

Nevertheless, we will see an example using the [20] Kaggle housing dataset. First, I will plot some of the variables to understand the dataset. It has an outcome tagged as *Price* and twelve variables, among which we can find *area*, *bedrooms*, *bathrooms*, etc. I will limit the dataset to the variables saving numerical values as this is not a classification model.

I plotted an OLS regression using each variable and outcome pair separately in the following graph. This way, we can see how the outcome changes as the variable values increase and get an initial idea of each feature's impact.

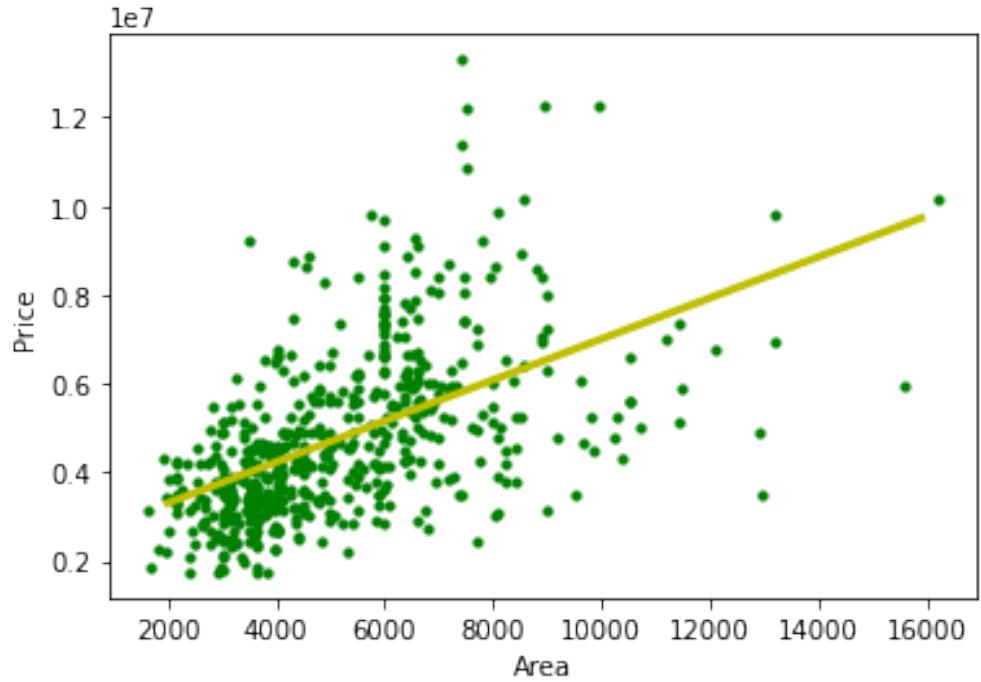


Figure 2.3: OLS regression on the Kaggle housing dataset: price vs. area.

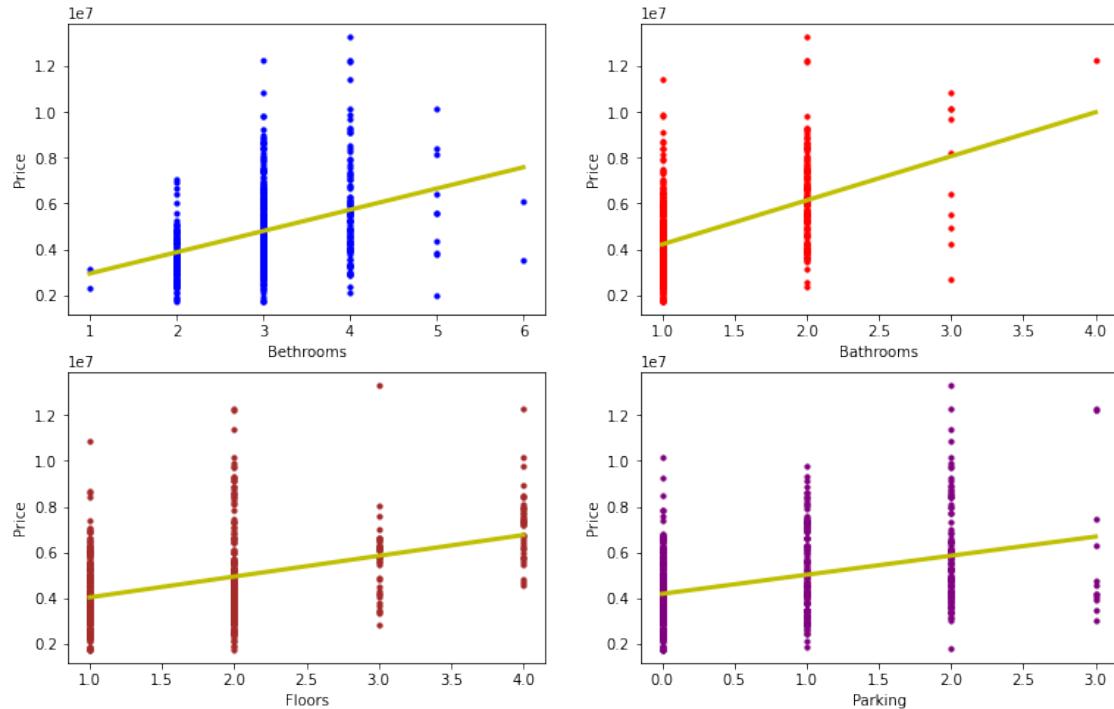


Figure 2.4: OLS regression on the Kaggle housing dataset: price vs. bedrooms, price vs. bathrooms, price vs. floors, and price vs. parking spots.

After normalizing our data, we will perform an OLS regression with the five variables combined. We will use cross-validation to have a reasonable estimate of how the model performs on this dataset. I will show how I proceed on the different steps.

```
[3]: import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv( "Housing.csv" )
Y = df.iloc[:,0].values
X = pd.DataFrame(np.c_[df['area'], df['bedrooms'],df['bathrooms'],
                      df['stories'],df['parking']],
                  columns = ['area', 'bedrooms', 'bathrooms',
                             'stories', 'parking']).values

def cross_validation(X,Y,iterations):

    MSE_values =[]
    for i in range(iterations):

        X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
                                                          test_size = 2/10, random_state = i)

        # NORMALIZATION
        X_train,Y_train,X_test,Y_test = normalization(X_train,
                                                       Y_train,X_test,Y_test)

        regression = OLS(X_train,Y_train)
        MSE = regression.MSE(X_test,Y_test)
        MSE_values.append(MSE)
    return(np.mean(MSE_values))

print("MSE: ",cross_validation(X,Y,2000))
```

```
MSE:  0.4531780959020959
```

We can see the results of the MSE for the testing dataset after averaging the results over the first 2000 random states, taking 80% of the points randomly as training data and the remaining 20% as testing data. The result is an MSE of 0.4532. We will test different models on this dataset and compare the results using the same number of iterations (2000) and the same size of the splits (80/20) to be consistent.

## 2.2 Ridge regression [Supervised]

### 2.2.1 Context

After working with the Ordinary Least Squares linear algorithm, the following question arises: Why do we need more linear algorithms apart from the OLS? As we have seen, the OLS focuses on finding the line that results in a lower MSE for the training data; this may result in overfitting depending on the dataset we are working with. [49][52] As I briefly mentioned in the introduction of the chapter, one of the things that the OLS struggles with is whenever we encounter collinear or multicollinear variables, that is, when there is a collinearity relation between k variables such that:

$$C_0 + C_1 X_{i,1} + C_2 X_{i,2} + \cdots + C_k X_{i,k} = 0 \quad (2.10)$$

If this is true for every data point  $\mathbf{x}_i = (X_{i,1}, X_{i,2}, \dots, X_{i,k})$ , then we have perfect collinearity. If the equation is almost true for all the data points, we will have imperfect collinearity. [10] To quantify the collinearity between any two variables, X and Y, we can introduce the equation of the correlation factor:

$$R(X, Y) = \frac{\sum_i^N X_i Y_i - N < X > < Y >}{N\sigma(X)\sigma(Y)} \quad (2.11)$$

Where  $< X >$  and  $< Y >$  are the respective averages and  $\sigma(X)$  and  $\sigma(Y)$  are the standard deviations. One could use the correlation factor to see how a variable correlates to the outcome, i.e., how much impact it has on the outcome. Alternatively, we can also check how much linear dependence there is between two *independent* variables, i.e., how much redundancy they contain. The correlation factor ranges from -1 to 1. If given a couple of variables X and Y, their correlation factor equals -1, this means that  $Y_i = C - X_i$ . [43] Another thing that is important to mention is that we can get an idea of the collinearity of a dataset by computing the correlation matrix and then computing the eigenvalues of the matrix; the closer they are to zero, the higher the multicollinearity. The correlation matrix is given by:

$$\mathbb{R} = \begin{pmatrix} R(X_1, X_1) & R(X_1, X_2) & \cdots & R(X_1, X_{D-1}) \\ R(X_2, X_1) & R(X_2, X_2) & \cdots & R(X_2, X_{D-1}) \\ \vdots & \vdots & \ddots & \vdots \\ R(X_{D-1}, X_1) & R(X_{D-1}, X_2) & \cdots & R(X_{D-1}, X_{D-1}) \end{pmatrix} \quad (2.12)$$

$X_1 \dots X_{D-1}$  are the variables containing information about the N data points. Given a matrix A, we can compute its eigenvalues by solving the following problem:

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M,1} & A_{M,2} & \cdots & A_{M,N} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix} = \lambda \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix} \quad (2.13)$$

Those  $\lambda$  parameters are called the eigenvalues, and  $v$  are the eigenvectors associated with them. Computing the eigenvalues boils down to solving the determinant of the following matrix and making it equal to zero:

$$\begin{vmatrix} A_{1,1} - \lambda & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} - \lambda & \cdots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M,1} & A_{M,2} & \cdots & A_{M,N} - \lambda \end{vmatrix} = 0 \quad (2.14)$$

Then, to obtain the eigenvectors, we would have to plug each eigenvalue into equation 2.13 and solve the system of N equations. [22] As checking if the eigenvalues are close to zero is a vague definition for multicollinearity, we introduce the idea of condition numbers:

$$N_k = \sqrt{\frac{\lambda_{\max}}{\lambda_k}} \quad (2.15)$$

Where  $\lambda_k$  are the eigenvalues of the matrix. As Jong Hae Kim's paper mentioned, a condition number between 10 and 30 indicates the presence of multicollinearity. When a value is greater than 30, the multicollinearity is considered substantial. We can apply all these concepts to the dataset introduced at the start of this chapter. This code computes the correlation matrix by computing the correlation between all the variable pairs.

```
[1]: import numpy as np
def correlation(X, Y):
    N = len(X)
    sum_XY = np.sum(X*Y)
    X_avg = np.mean(X, axis=0)
    X_std = np.std(X, axis=0)
    Y_avg = np.mean(Y, axis=0)
    Y_std = np.std(Y, axis=0)
    correlation = (sum_XY - N*X_avg*Y_avg)/(N*X_std*Y_std)
    return(correlation)

import pandas as pd
dataset = pd.read_csv( "Studyscores.csv" )
variables = ["Total study time", "Study time at university",
             "Study time at home", "Stress levels", "Age", "Height"]

corr_matrix = np.zeros((6,6))

for i in range(len(variables)):
    for j in range(len(variables)):
        corr_matrix[i][j] = correlation(dataset[variables[i]].values,
                                         dataset[variables[j]].values)
```

```
print(corr_matrix)
```

```
[[ 1.          0.91028891  0.83156107 -0.9997506  -0.02170073 -0.13038117]
 [ 0.91028891  1.          0.53187368 -0.91148952 -0.05765437 -0.17874128]
 [ 0.83156107  0.53187368  1.          -0.82939455  0.0076254  -0.02291255]
 [-0.9997506  -0.91148952 -0.82939455  1.          0.02530646  0.12646324]
 [-0.02170073 -0.05765437  0.0076254   0.02530646  1.          -0.06766678]
 [-0.13038117 -0.17874128 -0.02291255  0.12646324 -0.06766678  1.      ]
 [ 1.          ]]
```

As expected, the diagonal of the matrix is composed of ones since any variable is collinear with itself. Next, we compute the eigenvalues of the matrix and the condition numbers using the *NumPy* package.

```
[2]: from numpy import linalg as la
def eigenvalues(M):
    eigenvalues, eigenvectors = la.eig(M)
    eigenvalues = np.array([eigen.real for eigen in eigenvalues])
    count_number = max(eigenvalues)/eigenvalues
    return(count_number)
print(eigenvalues(corr_matrix))
```

```
[1.00000000e+00 3.33219512e+00 3.76746004e+00 7.95732141e+00
 1.66205243e+04 1.32194876e+03]
```

Since the multicollinearity is almost perfect among some variables, we can see that some condition numbers vastly exceed the boundary of 30 units.

## 2.2.2 Mathematical derivation

When we implemented the OLS, we wanted to minimize the quantity:

$$S(\beta) = \sum_{i=1}^N (y_i - [\beta_1 X_{i,1} + \beta_2 X_{i,2} + \cdots + \beta_{D-1} X_{i,D-1} + \beta_0])^2 \quad (2.16)$$

If we want to address the problem of overfitting, it would be more beneficial to have a model with smaller slopes in some cases. The idea behind the Ridge regression is to expand the OLS algorithm by adding an extra term to  $S(\beta)$ . This term is called the penalty term, which controls the size of the  $\beta$  coefficients.

$$S(\beta) = \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + \alpha \sum_{j=0}^{D-1} \beta_j^2 \quad (2.17)$$

As we can see now, if we attempt to minimize  $S(\beta)$ , we will have two terms contributing to it. The first term computes the MSE, and the second one contains the size of the  $\beta$  coefficients. Depending on the value of the parameter  $\alpha$ , we will set the importance of the second term, and the model will find a balance among the terms when choosing  $\beta_i$  to minimize  $S(\beta)$ .

We can now understand why Ridge performs well when multicollinearity is present. If we have some linearly dependent variables, the OLS algorithm will account for the same information many times, making it more likely to lead to overfitting. This effect will be lower on Ridge as the reduction of the beta coefficients will counter this overfitting. As we proceed before, we start by computing the partial derivative:

$$D_a = \frac{\partial S(\beta)}{\partial \beta_a} = -2 \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right) + 2\alpha \beta_a \quad (2.18)$$

Furthermore, by setting the derivative equal to zero, we obtain the equality:

$$\sum_{i=1}^N X_{i,a} y_i = \sum_{i=1}^N \sum_{j=0}^{D-1} X_{i,a} X_{i,j} \beta_j + \alpha \beta_a \quad (2.19)$$

This can be written in matrix notation in a more compact equation:

$$X^T y = (X^T X + \alpha \mathbb{I}) \beta \quad (2.20)$$

$\mathbb{I}$  is an identity matrix of size  $D \times D$ . By multiplying on both sides of the equation by  $(X^T X + \alpha \mathbb{I})^{-1}$ , we finally obtain:

$$\beta = (X^T X + \alpha \mathbb{I})^{-1} X^T y \quad (2.21)$$

Remember that our vector  $\beta$  has the dimensions of the number of variables in the data set plus one. This is because we added a constant term to  $\beta$  for the OLS regression, which

also applies to the Ridge model. As the Ridge model's motivation is to fix the issues presented when using the OLS model, such as overfitting due to multicollinearity, we only want the penalty term to affect the coefficients, not the intercept. To fix this issue and prevent the algorithm from minimizing the constant term, we put a 0 on the position (1, 1) of the identity matrix so that:

$$\beta = (X^T X + \alpha \mathbb{I}')^{-1} X^T y \quad (2.22)$$

Where now, instead of the regular identity matrix, we have:

$$\mathbb{I}' = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \quad (2.23)$$

This change will force the model to minimize the elements of  $\beta$  that contribute to the slopes of the regression but not the constant term. This does not mean that the constant term of the Ridge regression will be equal to the constant term of the OLS regression as  $\beta_0$  will also depend on the choice of  $\alpha$ .

### 2.2.3 Algorithm implementation

Now, we proceed with the implementation of the Ridge model:

```
[1]: import matplotlib.pyplot as plt
import numpy as np

class RIDGE():
    def __init__(self,X,Y,alpha):
        self.X=X
        self.Y=np.reshape(Y,(len(Y),1))
        self.beta = np.zeros((len(X[0])+1,1))
        self.alpha = alpha

    def extend(self,X):
        M = len(X)
        N = len(X[0])
        extended_X = np.ones((M,N+1))
        extended_X[:,1:] = X
        return(extended_X)
```

The code for the Ridge algorithm is very similar to the implementation of the OLS. The only changes are on the *train model* function, which includes a couple of extra lines where I build the identity matrix and a few changes on the computation of the beta coefficients.

```
def train_model(self,X,Y):
    # We add the column of 1s to X
    X = self.extend(self.X)
    # We build the identity matrix with a 0 on (0,0)
    Identity=np.eye(len(X[0]))
    Identity[0][0]=0
    # We compute the different products involved in Beta
    X_T = np.transpose(X)
    product_1=np.linalg.inv(np.dot(X_T,X) + self.alpha * Identity)
    product_2=np.dot(X_T,self.Y)
    #We compute the Beta coefficients
    self.beta=np.dot(product_1,product_2)
    return(self.beta)

def coefficients(self):
    if np.sum(self.beta) == 0:
        self.beta = self.train_model(self.X,self.Y)
    return(np.transpose(self.beta))

def prediction(self,X_test):
```

```

    if np.sum(self.beta) == 0:
        self.beta = self.train_model(self.X, self.Y)
    X_test = self.extend(X_test)
    prediction = np.dot(X_test, self.beta)
    return(prediction)

def MSE(self, X_test, Y_test):
    Y_test = np.reshape(Y_test, (len(Y_test), 1))
    error_array = self.prediction(X_test) - Y_test
    MSE = np.sum(error_array * error_array) / len(Y_test)
    return(MSE, self.beta)

```

The first step will be to work with the dataset I created, which can be found in table 2.1. The following code shows how to read data from a dataset saved on a text file. As we can see, we can load the different features by accessing their indexes or the feature names. I loaded the variables of the four different data collections we will test.

```

[2]: import pandas as pd
import numpy as np
dataset = pd.read_csv("Studyscores.csv")

Y = dataset.iloc[:, 6].values
Y = np.reshape(Y, (len(Y), 1))

#DATASET 1
X_1 = dataset.iloc[:, 0].values
X_1 = np.reshape(X_1, (len(X_1), 1))

#DATASET 2
X_2 = dataset.iloc[:, 0:4].values

#DATASET 3
X_3 = pd.DataFrame(np.c_[dataset['Total study time'],
dataset['Age'], dataset['Height']],
columns = ['Total study time', 'Age', 'Height']).values

#DATASET 4
X_4 = dataset.iloc[:, 0:6].values

```

I also built a small function that splits the data into 80% training data and 20% testing data and averages the results of the MSE for as many random states as we want for a given  $\alpha$  value. As we can see, the MSE was computed for the testing dataset. I also averaged the values of the  $\beta$  coefficients over the random states to plot their evolution in terms of  $\alpha$ .

```
[3]: from sklearn.model_selection import train_test_split

def cross_validation_RIDGE(X,Y,iterations,alpha):
    MSE_values = []
    beta_coef = []

    for i in range(iterations):

        X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
        test_size = 2/10,random_state=i)

        # NORMALIZATION
        X_train,Y_train,X_test,Y_test = normalization(X_train,
                                                       Y_train,X_test,Y_test)

        regression = RIDGE(X_train,Y_train,alpha)
        MSE,beta = regression.MSE(X_test,Y_test)
        MSE_values.append(MSE)
        beta_coef.append(beta)

    beta_coef = np.transpose(np.mean(np.array(beta_coef),axis=0))
    return(np.mean(MSE_values),beta_coef)
```

Lastly, I called the cross-validation function for different values of  $\alpha$  and stored the MSE and  $\beta$  coefficients. We can see an example of how the model is executed for the second data collection. I averaged the MSE and  $\beta$  coefficients across the first 2000 random states for each value of alpha on each of the data collections. The more random states we cover, the better accuracy we get. I set the number of iterations to 2000 because other models, such as the Elastic-net, were much slower.

```
[4]: #RIDGE ON DATA 2
alpha_2 = np.arange(0,30,0.3)
result_2 = np.array([cross_validation_RIDGE(X_2,Y,2000,
                                             alpha_i) for alpha_i in alpha_2],dtype=object)
# -----
B_2 = np.array([result_2[:,1][i][0] for i in range(len(alpha_2))])
MSE_2 = np.array([result_2[:,0][i] for i in range(len(alpha_2))])
print("Min value of Ridge: ",min(MSE_2))
print("OLS value: ",MSE_2[0])
```

```
Min value of Ridge:  0.34458144806123925
OLS value:  0.37555112867504026
```

Keep in mind that for the plots where the scale is logarithmic, I used a different distribution of values for alpha, covering all the decimal ranges.

The first data collection did not contain multicollinear or irrelevant variables; hence, we can expect the Ridge model not to improve the OLS results significantly. As we can see, the only coefficient  $\beta_1$  associated with the variable *total study time* is reduced, leading to a slight improvement on the MSE (0.02 % reduction on the MSE).

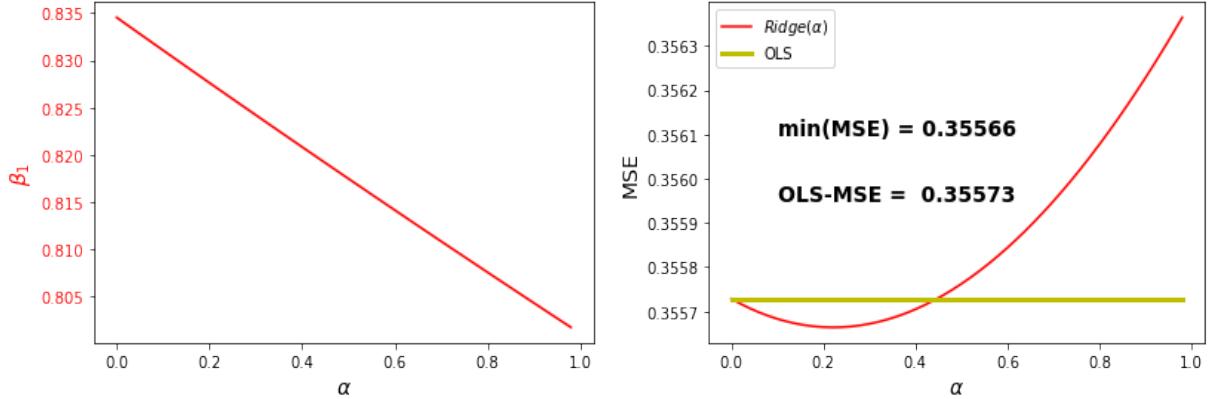


Figure 2.5: Accuracy results and the evolution of the  $\beta$  parameters of the Ridge model for the first data collection from table 2.1.

The second data collection contained some linearly dependent variables. We can see that the Ridge model lowers the MSE by 8.25% compared to the OLS result. We have four  $\beta$  coefficients  $\beta_1, \beta_2, \beta_3$  and  $\beta_4$  associated to the variables *total study time*, *study time at university*, *study time at home* and *stress levels*. As we can see,  $\beta_1$  and  $\beta_4$  are the largest  $\beta$  coefficients; hence, the model chooses to reduce them. On the first  $\alpha$  values, the Ridge algorithm increases  $\beta_2$  and  $\beta_3$  slightly, as the reduction of the first term of Equation 2.17 compensates for increasing the second. As  $\alpha$  increases and  $\beta_1$  and  $\beta_4$  are low enough, all the coefficients are reduced at the same time.

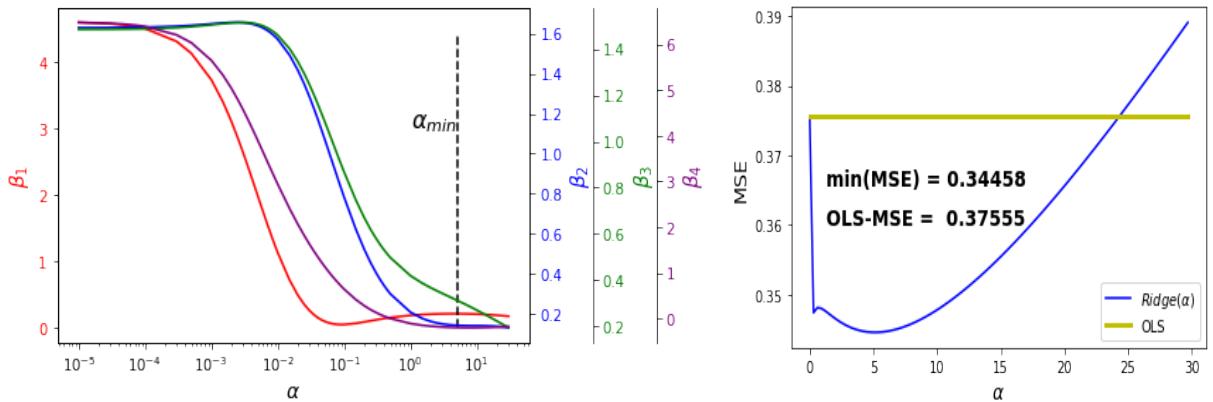


Figure 2.6: Accuracy results and the evolution of the  $\beta$  parameters of the Ridge model for the second data collection from table 2.1.

As we can see, the Ridge model slightly improves the results of the third dataset, reducing the MSE by 0.60%. We have three  $\beta$  coefficients  $\beta_1$ ,  $\beta_5$  and  $\beta_6$  associated to the variables *total study time*, *age* and *height*. The model reduces the  $\beta_1$  coefficient and adjusts  $\beta_5$  and  $\beta_6$  accordingly to minimize the first term of  $S(\beta)$ . The MSE is improved at first because reducing  $\beta_1$  removes some of the overfitting by the OLS model. As we increase  $\alpha$  further, the model leads to underfitting.

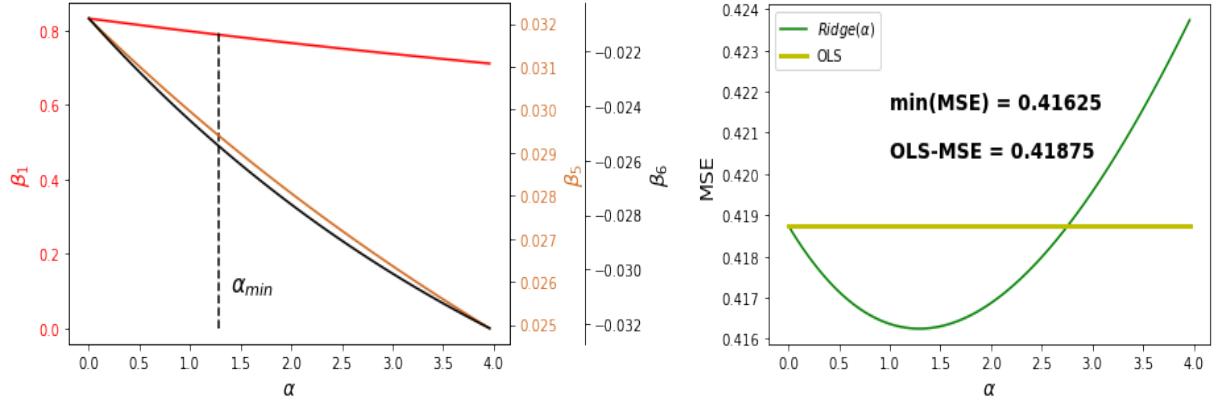


Figure 2.7: Accuracy results and the evolution of the  $\beta$  parameters of the Ridge model for the third data collection from table 2.1.

Last, the fourth data collection contains a combination of all the variables from the dataset, including linearly dependent variables. Hence, we find an improvement of the MSE of 15.01%. As we can see, the evolution of the  $\beta$  coefficients is similar to that from the second dataset. The linearly dependent variables with more significant  $\beta$  coefficients are reduced, and the other coefficients are adjusted accordingly to reduce the magnitude of Equation 2.17. As  $\alpha$  grows and the predominant  $\beta$  coefficients are smaller, the second term of the equation becomes dominant, and all the coefficients are reduced.

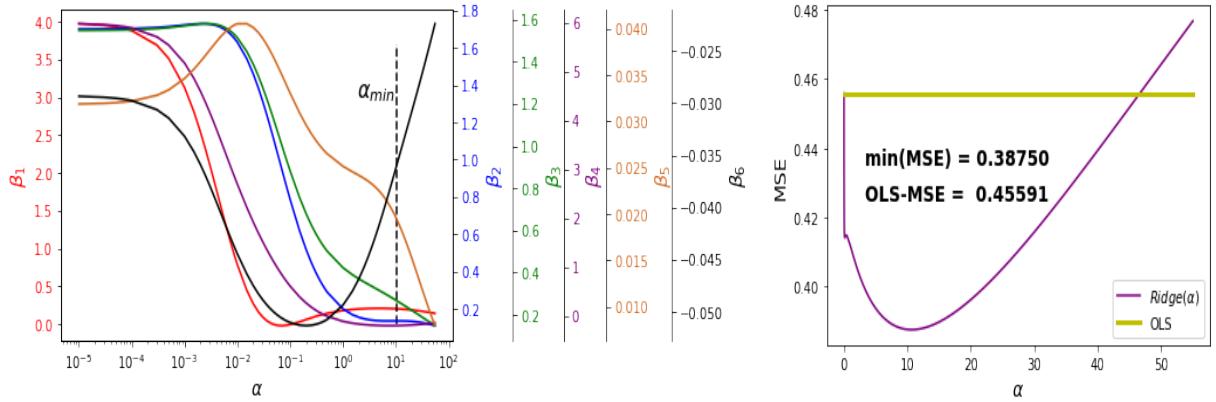


Figure 2.8: Accuracy results and the evolution of the  $\beta$  parameters of the Ridge model for the fourth data collection from table 2.1.

We will apply the model to the Kaggle housing dataset, including the following variables: *Area*, *Bedrooms*, *Bathrooms*, *Stories*, and *Parking* in that order. Before doing it, we will check the correlation matrix and the condition numbers associated with the eigenvalues. The correlation matrix of the dataset is the following:

$$\mathbb{R} = \begin{pmatrix} 1 & 0.15185849 & 0.19381953 & 0.08399605 & 0.35298048 \\ 0.15185849 & 1 & 0.37393024 & 0.40856424 & 0.1392699 \\ 0.19381953 & 0.37393024 & 1 & 0.32616471 & 0.17749582 \\ 0.08399605 & 0.40856424 & 0.32616471 & 1 & 0.04554709 \\ 0.35298048 & 0.1392699 & 0.17749582 & 0.04554709 & 1 \end{pmatrix} \quad (2.24)$$

And the condition numbers are :  $\{1.0, 1.61303591, 3.33738604, 2.98598942, 2.92665924\}$ . As we can see, none of the values are close to 10 to suggest the presence of any strong multicollinearity. Now, we proceed to apply the Ridge model to the dataset.

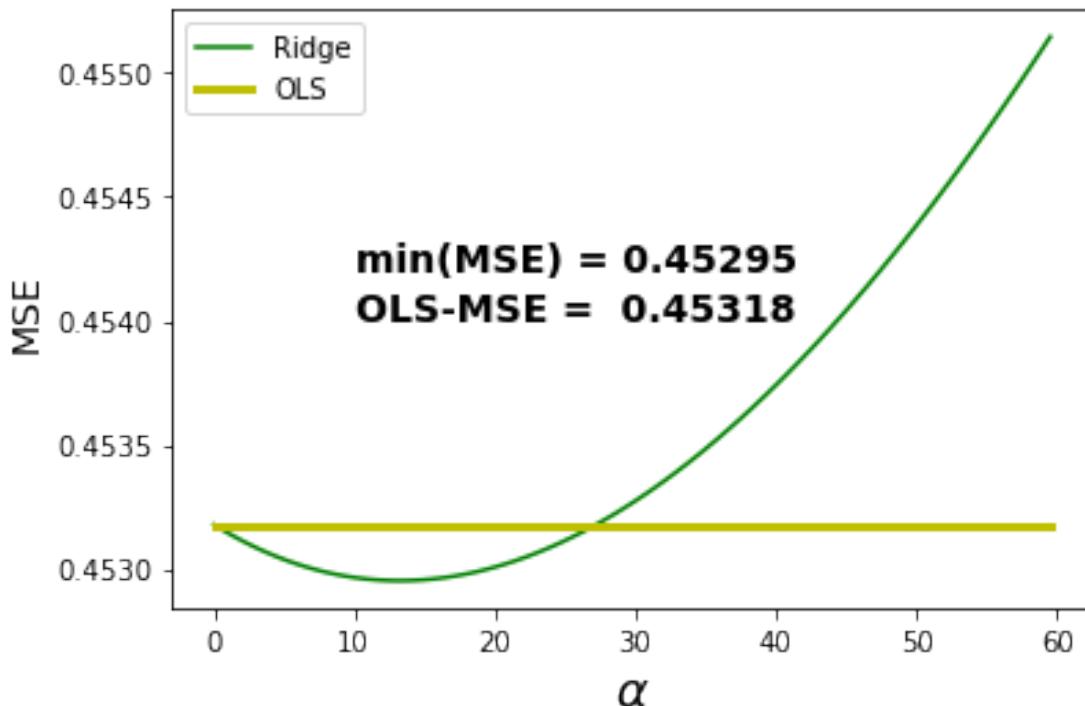


Figure 2.9: Ridge regression on the Kaggle housing dataset.

As we can see, the Ridge model provides an MSE improvement of 0.05% compared to the OLS result. This improvement is insignificant due to the dataset not having multicollinearity. Another thing worth mentioning is that we are only getting such smooth curves on these plots because we are taking random states. If we were to remove this parameter when splitting the data, the plot would be spiked due to having different shuffles for each  $\alpha$  value and, hence, a considerable deviation between the MSE values.

## 2.3 Lasso regression [Supervised]

### 2.3.1 Context

Let us recover the quantity we wanted to minimize on the Ridge regression:

$$S(\beta) = \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + \alpha \sum_{j=0}^{D-1} \beta_j^2 \quad (2.25)$$

The Lasso regression will have a similar expression, but instead of a quadratic term on  $\beta$ , it will have a term proportional to the  $l_1$  norm of beta, given by  $|\beta|_{l_1} = |\beta_0| + |\beta_1| + \dots + |\beta_{D-1}|$ , i.e., the whole expression reads:

$$S(\beta) = \frac{1}{2} \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + \lambda \sum_{j=0}^{D-1} |\beta_j| \quad (2.26)$$

[19] Following the explanation by Boris Giba showing the difference between Ridge and Lasso, if we check both expressions, we can realize that in both cases, we are introducing a term proportional to a power of  $\beta$  to control their size. The bigger the coefficient multiplying the powers of  $\beta$ , the more the algorithm will focus on reducing the magnitude of  $\beta$  rather than the MSE. However, there is a subtle difference between the Ridge and Lasso algorithms. Imagine we have some beta coefficients given by  $\beta = (\beta_1, \beta_2, \beta_3)$ . Now we subtract a unit from the different coefficients leading to three different configurations:  $\beta^{(1)} = (\beta_1 - 1, \beta_2, \beta_3)$ ,  $\beta^{(2)} = (\beta_1, \beta_2 - 1, \beta_3)$  and  $\beta^{(3)} = (\beta_1, \beta_2, \beta_3 - 1)$  and hence  $\alpha \sum_i \beta_i^2$  will be given by the following expressions respectively:

$$\begin{aligned} \alpha \sum_i [\beta_i^{(1)}]^2 &= \alpha [\beta_1^2 + \beta_2^2 + \beta_3^2 - 2\beta_1 + 1] \\ \alpha \sum_i [\beta_i^{(2)}]^2 &= \alpha [\beta_1^2 + \beta_2^2 + \beta_3^2 - 2\beta_2 + 1] \\ \alpha \sum_i [\beta_i^{(3)}]^2 &= \alpha [\beta_1^2 + \beta_2^2 + \beta_3^2 - 2\beta_3 + 1] \end{aligned}$$

As we can see for the Ridge algorithm, subtracting a unit from one of the beta elements results in a difference on the quadratic term that directly depends on this beta element, i.e., the more significant that beta element is, the bigger it will be the difference when subtracting a unit from it. Since the algorithm wants to reduce the quantity  $S(\beta)$ , it will prioritize reducing those beta parameters that are large since reducing significant beta parameters causes a more drastic reduction to the quadratic term of  $S(\beta)$ . Another thing that is important to point out is that  $\beta_i^2 < \beta_i$  when  $0 < \beta_i < 1$  and hence the Ridge penalty (the quadratic term) will get weaker as the beta coefficients get close to zero, i.e., with the Ridge regression the parameters will get small but never zero as when they approach small values the quadratic term will lose relevance.

On the other hand, the Lasso regression works differently. If we apply the same subtraction to the beta coefficients, we find the following:

$$\begin{aligned} |\beta^{(1)}|_{l_1} &= |\beta_1 - 1| + |\beta_2| + |\beta_3| \\ |\beta^{(2)}|_{l_1} &= |\beta_1| + |\beta_2 - 1| + |\beta_3| \\ |\beta^{(3)}|_{l_1} &= |\beta_1| + |\beta_2| + |\beta_3 - 1| \end{aligned}$$

As we can see, the Lasso regression does not care about the size of the coefficients. This is because the penalty is a linear term; hence, the reduction of a large or small coefficient impacts the penalty term equally. The Lasso regression will start reducing the  $\beta$  coefficients to reduce the magnitude of  $S(\beta)$  until a balance is reached (the balance is reached when reducing a coefficient further is no longer optimal since it results in increasing the size of the MSE factor more than the penalty reduction and hence leading to a bigger  $S(\beta)$ ). The best part about the Lasso regression is that if we have a variable that the outcome does not depend on, reducing its beta coefficient will lead to the MSE factor not being altered much. The algorithm will set this beta coefficient to low or even zero. The Lasso regression will remove variables or coefficients irrelevant to the data set, which cannot happen on the Ridge implementation since the coefficients will be low but never zero.

### 2.3.2 Mathematical derivation

For the Lasso model, we will choose to minimize a variation of the function we had for the Ridge algorithm. Instead of having a quadratic term on the  $\beta$  coefficients, we have the  $l_1$  norm of  $\beta$ .

$$S(\beta) = \frac{1}{2} \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + \lambda \sum_{j=0}^{D-1} |\beta_j| \quad (2.27)$$

I have introduced a constant term  $1/2$ , which is entirely optional, to remove a factor of 2 that appears on that first term when we compute the derivative. [37] In this section, I will follow the derivation by Xavier Bourret Sicotte. We will divide our function  $S(\beta)$  into the contributions of the OLS and the Lasso term:

$$S(\beta) = S^{OLS}(\beta) + S^{LASSO}(\beta) \quad (2.28)$$

The derivative of the first term was computed in the first section:

$$D_a^{OLS} = \frac{\partial S^{OLS}(\beta)}{\partial \beta_a} = - \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right) \quad (2.29)$$

Now, we are going to split the result into different contributions, one containing the term proportional to  $\beta_a$  and the other one containing the remaining terms:

$$D_a^{OLS} = - \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j \neq a} \beta_j X_{i,j} \right) + \sum_{i=1}^N X_{i,a} X_{i,a} \beta_a \quad (2.30)$$

If we name the quantities on the previous equation the following way:

$$P_a = \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j \neq a} \beta_j X_{i,j} \right) = \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right) + Q_a \beta_a \quad (2.31)$$

$$Q_a = \sum_{i=1}^N (X_{i,a})^2 \quad (2.32)$$

We have that:

$$D_a^{OLS} = -P_a + Q_a \beta_a \quad (2.33)$$

The term concerning the Lasso regression is more problematic. To see why we consider the absolute value function  $|x|$ , the derivative of this function is given by  $-1$  for  $x < 0$  and by  $1$  for  $x > 0$ , and hence the derivative is not well defined at the point  $x = 0$ . To fix this issue it is essential to introduce the concept of sub-differentials.

---

## SUB-DIFFERENTIALS

We know that for every differentiable function, we can always approximate the value of this function at any point by a Taylor series:

$$f(x + \Delta x) \approx f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 + \dots \quad (2.34)$$

We will keep only the first two terms of the expansion. For a convex function, the first order of the Taylor series is a global underestimation of the function, meaning that:

$$f(x + \Delta x) \geq f(x) + f'(x)\Delta x \quad (2.35)$$

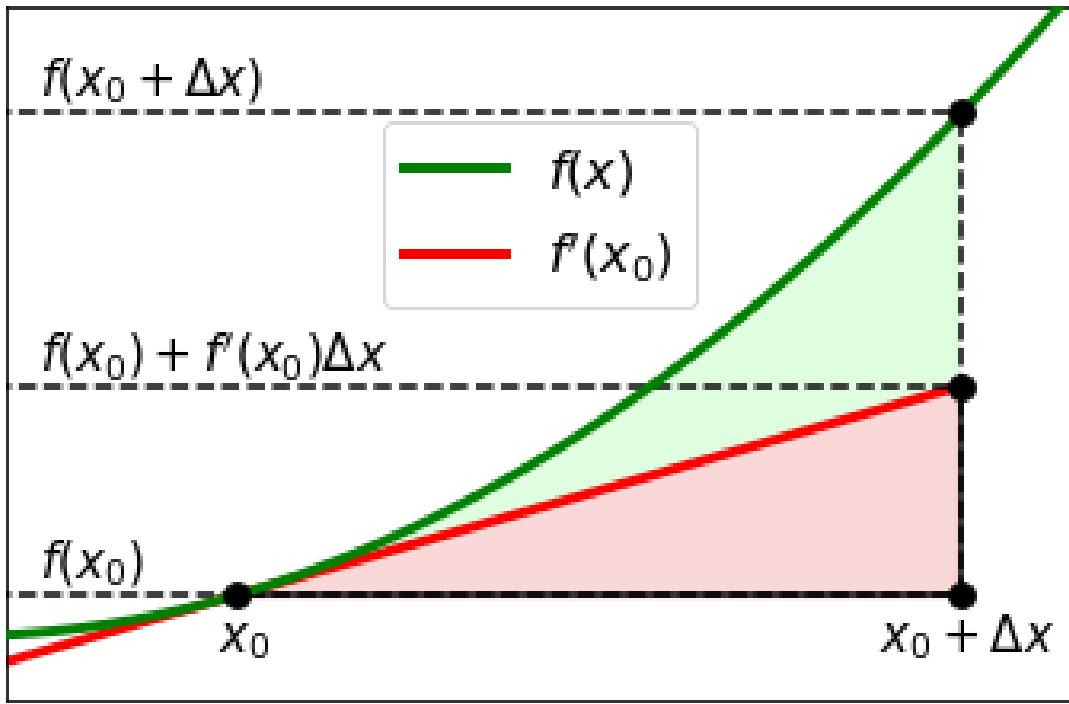


Figure 2.10: Graphical illustration of equation 2.35 for a convex function.

In the previous graph, we can see that  $f(x_0 + \Delta x) \geq f(x_0) + f'(x_0)\Delta x$  and that if  $\Delta x$  is small, it is a good approximation for  $f(x_0 + \Delta x)$ . Now we introduce the concept of a sub-differential for a convex function  $f(x)$ :

$$\partial_x^{sub} f(x_0) = \{k : f(x_0 + \Delta x) > f(x_0) + k \Delta x \quad \forall \Delta x \in \mathbb{R}\} \quad (2.36)$$

The sub-differential of a function  $f(x)$  in a point  $x_0$  is the collection of all the slopes  $k$  such that  $f(x_0 + \Delta x) > f(x_0) + k \Delta x$  for any possible value of  $\Delta x$ . Each value labeled with a  $k$  is called a sub-gradient or sub-derivative of  $f(x)$  in  $x_0$ .

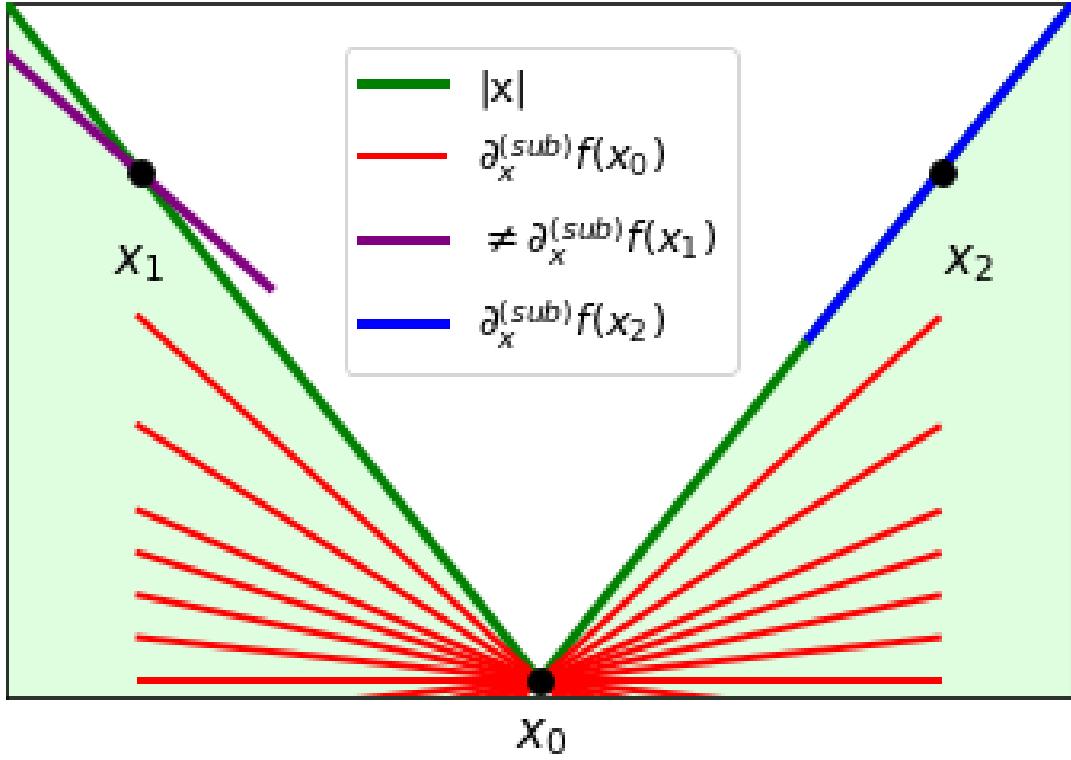


Figure 2.11: Sub-differentials of the  $|x|$  function on the regions  $x < 0$ ,  $x > 0$  and  $x = 0$ .

So, as we just mentioned, the sub-differential of a function  $f(x)$  on a point  $x_0$  is the set of all those slopes that form lines that contain the point  $(x_0, f(x_0))$  but that remain under or barely touching the  $f(x)$  function. We can write the sub-differential of a function  $f(x)$  on a point  $x_0$  as a closed interval  $[a, b]$  where:

$$a = \lim_{x \rightarrow x_0^-} \frac{f(x) - f(x_0)}{x - x_0} \quad (2.37)$$

$$b = \lim_{x \rightarrow x_0^+} \frac{f(x) - f(x_0)}{x - x_0} \quad (2.38)$$

The following properties are fulfilled:

- A function is differentiable on a point  $x_0$  if and only if the sub-differential is composed of a single value: the function's derivative on  $x_0$ .
- A point  $x_0$  is a global minimum of a convex function  $f(x)$  if and only if the zero is contained in its sub-differential (i.e., we can draw a horizontal line of slope zero on  $x_0$  which remains under the graph).
- If  $f(x)$  and  $g(x)$  are convex functions with respective sub-differentials:  $\partial_x^{sub} f(x)$  and  $\partial_x^{sub} g(x)$ , then the following is fulfilled:  $\partial_x^{sub} [f(x) + g(x)] = \partial_x^{sub} f(x) + \partial_x^{sub} g(x)$ .

If we recover our function  $f(x) = |x|$ , we can draw the following conclusions:

- $f(x) = |x|$  is convex but not differentiable on  $x = 0$  since the sub-differential is not composed of a single value.
  - On  $x = 0$ , the sub-differential of this function is the interval  $[-1, 1]$  since 1 and -1 are the respective values of the derivatives on  $0^+$  and  $0^-$ .
  - Since the zero is contained on the sub-differential of  $x = 0$ ,  $x = 0$  is a global minimum.
  - For  $x < 0$ , the function only has one sub-differential value, which is  $-1$ , and for  $x > 0$ , only the sub-differential value of  $1$ ; hence the function is differentiable on every point except  $x = 0$ . If we look at figure 2.11, we can see why any other slope apart from the sub-differentials on  $x < 0$  or  $x > 0$  intercepts the function.
- 

Once we have finished introducing the concept of sub-differentials, we can start developing the **coordinate descent** method. Coordinate descent obtains the values of the coefficients for which  $S(\beta)$  is minimum by computing the derivatives and isolating the coefficients. Since the derivative of the function is not well defined for  $\beta_i = 0$ , the algorithm obtains under which condition  $\beta_i = 0$  leads to a minimum of  $S(\beta)$  by ensuring the zero is contained in the sub-differential of the problematic point.

$$D_a^{LASSO} = \frac{\partial S^{LASSO}(\beta)}{\partial \beta_a} = \lambda \partial_{\beta_a} |\beta_a| \quad (2.39)$$

So if we combine both terms, we have the total sub-differential of our Lasso model (remember that the sub-differential of a differentiable function is simply the derivative, so the sub-differential of the OLS term is already computed):

$$D_a^{total} = -P_a + Q_a \beta_a + \lambda \partial_{\beta_a} |\beta_a| \quad (2.40)$$

Making our expression equal to zero, we find the following relations:

$$0 = \begin{cases} -P_a + Q_a \beta_a - \lambda & (\beta_a < 0) \\ [-P_a - \lambda, -P_a + \lambda] & (\beta_a = 0) \\ -P_a + Q_a \beta_a + \lambda & (\beta_a > 0) \end{cases} \quad (2.41)$$

Computing the value of  $\beta_a$  for  $\beta_a < 0$  and  $\beta_a > 0$  is straightforward; we have to isolate the variable on equation 2.41 and we are guaranteed that those values of  $\beta_a$  lead to the global minimum of  $S(\beta)$ . For  $\beta_a = 0$ , it is more tricky. Since we want to find the configuration for which a value of  $\beta_a = 0$  leads to a global minimum, we need the zero to be contained on the interval  $[-P_a - \lambda, -P_a + \lambda]$ , i.e., we want:

$$0 \in [-P_a - \lambda, -P_a + \lambda] \quad (2.42)$$

From this expression, we can find the following conditions:

$$-P_a - \lambda \leq 0 \quad -P_a + \lambda \geq 0 \quad (2.43)$$

These two conditions can be written as  $-\lambda \leq P_a \leq \lambda$ , so we know that  $\beta_a = 0$  will be a global minimum as long as  $-\lambda \leq P_a \leq \lambda$ . The other two cases can be solved quickly, and one can check that they correspond to the cases where  $P_a \leq -\lambda$  and  $P_a \geq \lambda$ . Taking all this into account, one can write the iteration form as:

$$\begin{cases} \beta_a = \frac{P_a + \lambda}{Q_a} & \text{if } P_a \leq -\lambda \\ \beta_a = 0 & \text{if } -\lambda \leq P_a \leq \lambda \\ \beta_a = \frac{P_a - \lambda}{Q_a} & \text{if } P_a \geq \lambda \end{cases} \quad (2.44)$$

All we have to do is to iterate with this equation until we reach the desired convergence, and we will find the  $\beta$  coefficients for which  $S(\beta)$  is minimum. Remember that the intercept will not have a penalty term on the Lasso model, similar to the Ridge model, and hence:

$$D_0^{total} = -P_0 + Q_0\beta_0 \quad (2.45)$$

Which leads to:

$$\beta_0 = \frac{P_0}{Q_0} \quad (2.46)$$

[39] One last thing that must be mentioned is that libraries such as Scikit-learn define the minimization problem around the function:

$$S(\beta) = \frac{1}{2N} \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + \lambda \sum_{j=0}^{D-1} |\beta_j| \quad (2.47)$$

This only changes the definition of our  $P_a$  and  $Q_a$  factors, dividing them by N. If we change this small detail, the algorithm will be more sensitive to  $\alpha$  as the MSE term will be smaller. So, a value of  $\alpha$  with the N factor will have more impact than the same  $\alpha$  value without the N factor.

It is also worth mentioning that we must update the  $\beta$  coefficients one by one with equation 2.44. If we upgrade all the values simultaneously, we can encounter divergencies on the  $\beta$  coefficients.

### 2.3.3 Algorithm implementation

Now, we start implementing the Lasso model with the coordinate descent method.

```
[1]: import numpy as np
import copy as copy

class Lasso():

    def __init__(self,Lambda,D):

        self.beta = np.zeros((D+1,1))
        self.Lambda = Lambda
        self.epsilon = 0.0000001

    def extend(self,X):

        M = len(X)
        N = len(X[0])
        extended_X = np.ones((M,N+1))
        extended_X[:,1:] = X
        return(extended_X)
```

The *train model* function of the Lasso model will first compute the  $Q_a$  factors. It iterates and updates the  $P_a$  elements individually and then updates the beta factors depending on the relative values between  $P_a$  and  $\lambda$ . I also divided all the elements involved by  $N$ , which is the dataset length, to synchronize the results with those of Scikit-learn. Here, I decided to iterate until the sum of the absolute value of the elements of the vector resulting from the subtraction of the  $\beta$  coefficients of two consecutive iterations was smaller than an  $\epsilon$  parameter, i.e., whenever  $\sum_j |\beta_j^{(k+1)} - \beta_j^{(k)}| < \epsilon$ .

```
def train_model(self,X,Y):
    Y = np.reshape(Y,(len(Y),1))
    N = len(Y)

    X = self.extend(X)
    X_T = np.transpose(X)

    Q = np.sum(X*X,axis=0)
    Q = np.reshape(Q,(len(Q),1))/N

    P = np.zeros((len(self.beta),1))
    beta = np.zeros(np.shape(self.beta))
```

```

for i in range(100000):

    for a in range(len(self.beta)):

        P[a] = np.dot(X_T[a],
                      (Y-self.pred(X))/N + Q[a]*self.beta[a])

        if (a==0):

            self.beta[a]=P[a]/Q[a]

        elif (P[a]<=-self.Lambda):
            self.beta[a]=(P[a]+self.Lambda)/Q[a]

        elif ((-self.Lambda<=P[a]) and (P[a]<=self.Lambda)):
            self.beta[a]=0

        elif (P[a]>=self.Lambda):
            self.beta[a]=(P[a]-self.Lambda)/Q[a]

        if (np.sum(abs(self.beta-beta)) < self.epsilon):

            break

        beta = copy.deepcopy(self.beta)

    return(self.beta)

def prediction(self,X_test):

    X_test = self.extend(X_test)
    prediction = np.dot(X_test,self.beta)
    return(prediction)

```

Another critical difference between the Lasso model and others is that I introduced another function that computes a prediction without extending the data (without adding the column of ones) so I could call it in the *train model* function when computing  $P_a$  in order to avoid adding more columns to the training data. The rest of the functions of the model remain unchanged.

```

def pred(self,X):

    prediction = np.dot(X,self.beta)
    return(prediction)

```

```

def MSE(self,X_test,Y_test):
    Y_test = np.reshape(Y_test,(len(Y_test),1))
    error_array = self.prediction(X_test) - Y_test
    MSE = np.sum(error_array*error_array)/len(Y_test)
    return(MSE)

```

First, we need to introduce the cross-validation function for the Lasso model. After realizing that the computation times were taking too long, I decided to initialize the  $\beta$  coefficients equal to the outcome of the previous random state to perform faster computations.

```

[2]: from sklearn.model_selection import train_test_split
def cross_validation_LASSO(X,Y,iterations,Lambda,D):
    MSE_values = []
    beta_coef = []

    regression = Lasso(Lambda,D)

    for i in range(iterations):

        X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
        test_size = 2/10, random_state = i)

        # NORMALIZATION
        X_train,Y_train,X_test,Y_test = normalization(X_train,
                                                       Y_train,X_test,Y_test)

        regression.train_model(X_train,Y_train)
        MSE,beta = regression.MSE(X_test,Y_test)
        MSE_values.append(MSE)
        beta_coef.append(beta)

    beta_coef = np.transpose(np.mean(np.array(beta_coef),axis=0))
    print(beta_coef)
    return(np.mean(MSE_values),beta_coef)

```

As we can see, I only called the Lasso model once for each lambda value. Then, using the *train model* function, we can update the  $\beta$  coefficients of the model with the new training data without having to restart the coefficients, and ensuring the convergence of the coefficients requires fewer iterations. Now, we will look at the results of the Lasso model for the example contained in table 2.1. As with the Ridge model, we will iterate over the first 2000 random states, splitting the data into 80% training data and 20% testing data.

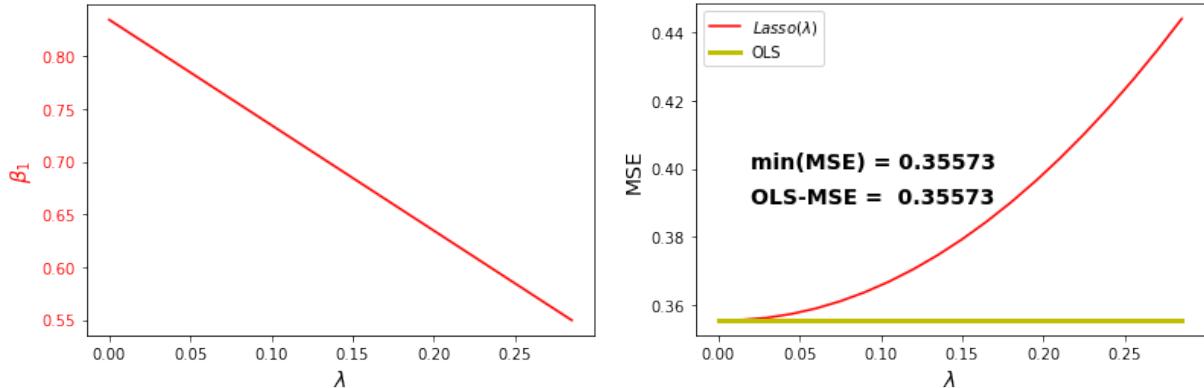


Figure 2.12: Accuracy results and the evolution of the  $\lambda$  parameters of the Lasso model for the first data collection from table 2.1.

We can see that the MSE of the first data collection does not improve at all. The second data collection contains linearly dependent variables. As we can see, as we increase the  $\lambda$  parameter, similarly to the Ridge model,  $\beta_1$  and  $\beta_4$ , which are the largest coefficients, are reduced. As  $\lambda$  increases further, the other two coefficients decrease too. The minimum value of the MSE is obtained when  $\beta_1$  and  $\beta_4$  are equal to zero, which provides an improvement of 7.82 % in comparison to the OLS. This is because the penalization term on the coefficients reduces some overfitting caused by the collinearity of the variables.

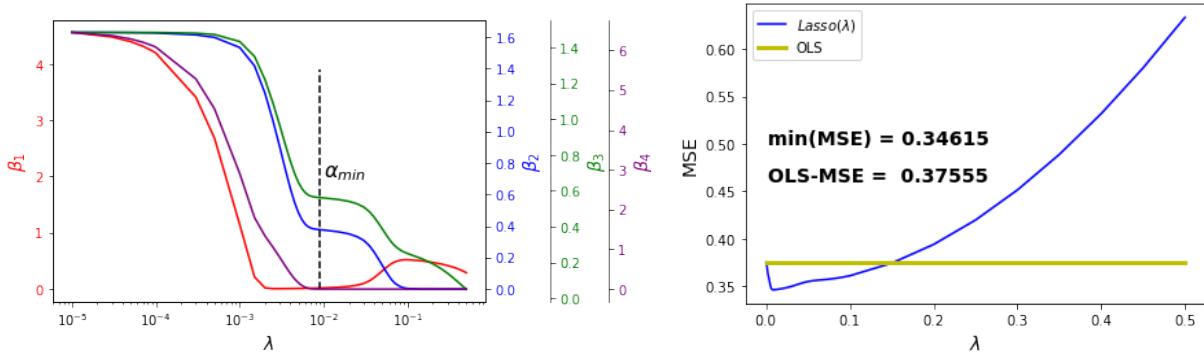


Figure 2.13: Accuracy results and the evolution of the  $\lambda$  parameters of the Lasso model for the second data collection from table 2.1.

The third data collection contained a couple of irrelevant variables (i.e., low correlation with the outcome), where the Lasso model excels. As we can see, as we increase the  $\lambda$  parameter, both  $\beta_5$  and  $\beta_6$  have their absolute values reduced. This leads to a new minimum for the MSE, which is improved by 10.70% in comparison to the OLS.

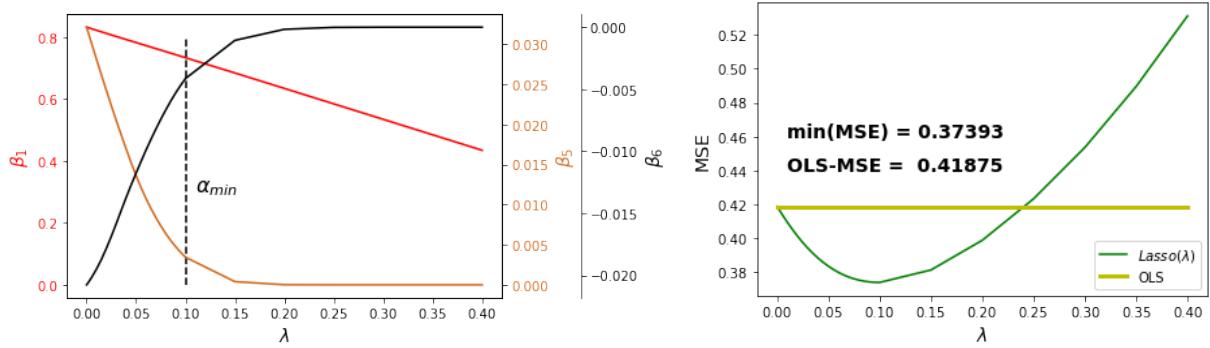


Figure 2.14: Accuracy results and the evolution of the  $\lambda$  parameters of the Lasso model for the third data collection from table 2.1.

Last, we have the fourth dataset; as one can see,  $\beta_1, \beta_2, \beta_3$ , and  $\beta_4$  have a similar evolution than in the second data collection, i.e., those coefficients that are larger are prioritized. The  $\beta_5$  and  $\beta_6$  coefficients start to decrease as  $\lambda$  grows larger to the point where they are negligible when the minimum of the MSE is reached. Overall, this configuration leads to a new minimum, which reduces the MSE by 18.27% compared to the OLS; this is because the Lasso model removes the irrelevant variables and counters some of the overfitting produced by the linearly dependent variables.

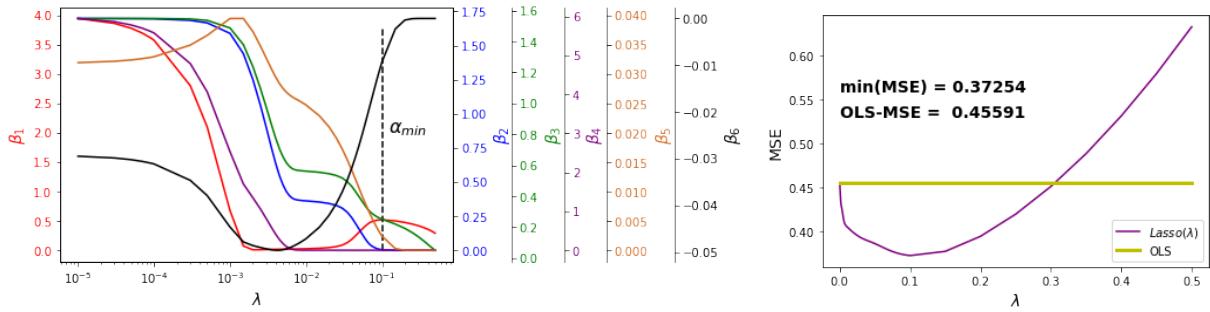


Figure 2.15: Accuracy results and the evolution of the  $\lambda$  parameters of the Lasso model for the fourth data collection from table 2.1.

Next, we are going to work with the Kaggle housing dataset. Similarly to what we did on the Ridge model, we will first predict the model's performance. Instead of computing the correlation between the variables, we are interested in the correlation between the different variables and the outcome. We expect the model to perform well if variables correlate poorly with the outcome.

The correlation factor between the feature area and the price is:

0.5359973457780801

The correlation factor between the feature bedrooms and the price is:

0.3664940257738684

The correlation factor between the feature bathrooms and the price is:  
0.5175453394550114

The correlation factor between the feature stories and the price is:  
0.4207123661886167

The correlation factor between the feature parking and the price is:  
0.38439364863572634

As we can observe, there is no indication that any of the variables are irrelevant to the prediction, as the smallest value on the correlation factor is 0.37. As shown in the graph and in accordance with these results, the Lasso model barely improves the results obtained by the OLS model. The improvement is of the order of  $10^{-5}$ .

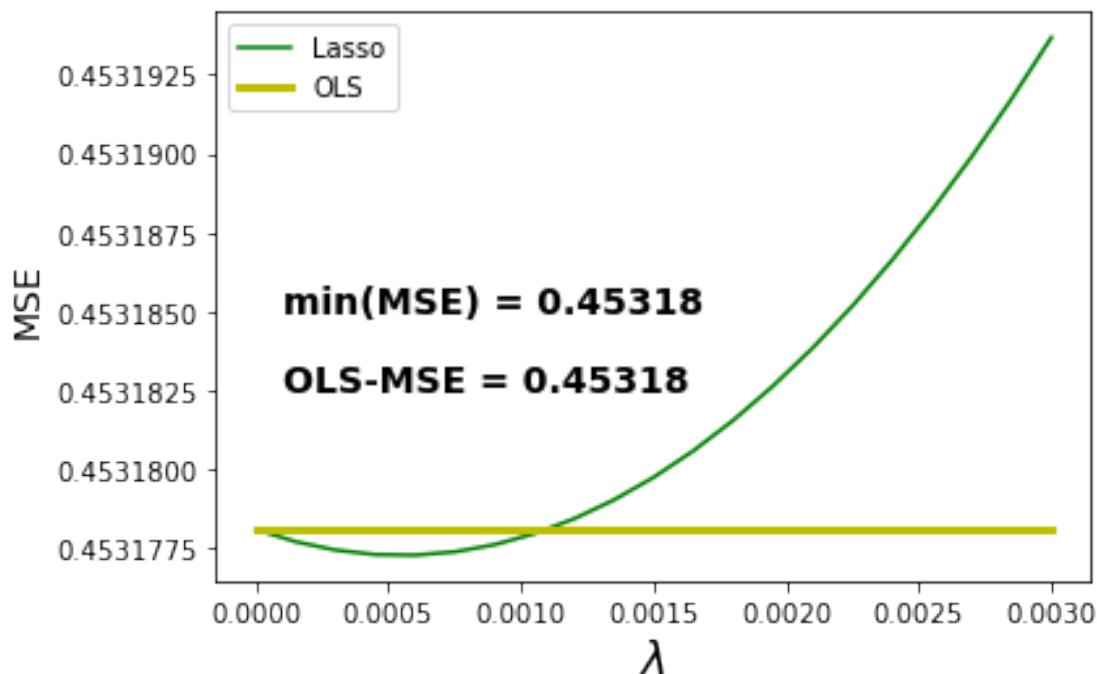


Figure 2.16: Lasso regression on the Kaggle housing dataset.

## 2.4 Elastic-Net [Supervised]

This method is a combination of the Ridge and Lasso models. It is helpful to reduce the impact of those variables that are multicollinear, and it also penalizes variables that have a low correlation with the outcome, allowing us to eliminate irrelevant features. This model will have both the l-1 norm and l-2 norm penalty terms.

### 2.4.1 Mathematical derivation

It follows that the quantity to minimize now is the following:

$$S(\beta) = \frac{1}{2N} \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + \frac{\alpha(1-\lambda)}{2} \sum_{j=0}^{D-1} \beta_j^2 + \alpha \lambda \sum_{j=0}^{D-1} |\beta_j| \quad (2.48)$$

The arrangement of the hyperparameters is different;  $\alpha$  can take any real value, and  $\lambda$  falls on the interval  $[0,1]$ , allowing us to enable the l-1 norm term, the l-2 norm term, or a combination of both. Similarly to the Lasso regression, we will implement the coordinate descent method. We start by writing the derivative for the OLS and Ridge as follows:

$$D_a^{OLS+RIDGE} = \frac{-1}{N} \sum_{i=1}^N X_{i,a} (y_i - \sum_{j \neq a} \beta_j X_{i,j}) + \frac{1}{N} \sum_{i=1}^N (X_{i,a})^2 \beta_a + \alpha(1-\lambda)\beta_a \quad (2.49)$$

Now, naming the quantities as we did in the previous section:

$$P_a = \frac{1}{N} \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j \neq a} \beta_j X_{i,j} \right) = \frac{1}{N} \sum_{i=1}^N X_{i,a} \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right) + Q_a \beta_a \quad (2.50)$$

$$Q_a = \frac{1}{N} \sum_{i=1}^N (X_{i,a})^2 \quad (2.51)$$

We can write the OLS and Ridge contributions as:

$$D_a^{OLS+RIDGE} = -P_a + Q_a \beta_a + \alpha(1-\lambda)\beta_a \quad (2.52)$$

Now, to compute the derivative of  $S(\beta)$  we need to add the contribution of the l-1 norm term:

$$D_a^{E-NET} = -P_a + Q_a \beta_a + \alpha(1-\lambda)\beta_a + \alpha \lambda \partial_{\beta_a} |\beta_a| \quad (2.53)$$

Since we are computing the sub-differential, we need to specify an interval for the singular problem at  $\beta_a = 0$  and then make all the expressions equal to zero:

$$0 = \begin{cases} -P_a + Q_a \beta_a + \alpha(1-\lambda)\beta_a - \alpha \lambda & (\beta_a < 0) \\ [-P_a - \alpha \lambda, -P_a + \alpha \lambda] & (\beta_a = 0) \\ -P_a + Q_a \beta_a + \alpha(1-\lambda)\beta_a + \alpha \lambda & (\beta_a > 0) \end{cases}$$

Similarly to what we did on the Lasso model, we want to impose that we have a global minimum of  $S(\beta)$  whenever we set  $\beta_a = 0$ , and for that, we need the zero to be contained in the interval, i.e., we need:

$$0 \in [-P_a - \alpha \lambda, -P_a + \alpha \lambda] \quad (2.54)$$

From this expression, we can find the following conditions:

$$-P_a - \alpha \lambda \leq 0 \quad -P_a + \alpha \lambda \geq 0 \quad (2.55)$$

These two conditions can be written as  $-\alpha \lambda \leq P_a \leq \alpha \lambda$ . Moreover, we know that  $\beta_a = 0$  will be a global minimum as long as the previous condition is fulfilled. The other two cases can be solved easily, and one can check that they correspond to  $P_a \leq -\alpha \lambda$  and  $P_a \geq \alpha \lambda$ . Taking all this into account, one can write the iteration form as:

$$\begin{cases} \beta_a = \frac{P_a + \alpha \lambda}{Q_a + \alpha(1-\lambda)} & \text{if } P_a \leq -\alpha \lambda \\ \beta_a = 0 & \text{if } -\alpha \lambda \leq P_a \leq \alpha \lambda \\ \beta_a = \frac{P_a - \alpha \lambda}{Q_a + \alpha(1-\lambda)} & \text{if } P_a \geq \alpha \lambda \end{cases} \quad (2.56)$$

For  $\beta_0$ , we set both  $\alpha$  and  $\lambda$  to zero since we only want the penalty terms affecting the slopes of the regression. Keep in mind that if we wanted to keep track of the different penalization terms separately, i.e., with an  $S(\beta)$  given by:

$$S(\beta) = \frac{1}{2N} \sum_{i=1}^N \left( y_i - \sum_{j=0}^{D-1} \beta_j X_{i,j} \right)^2 + a \sum_{j=0}^{D-1} \beta_j^2 + b \sum_{j=0}^{D-1} |\beta_j| \quad (2.57)$$

We obtain this relation among the coefficients:

$$\lambda = \frac{b}{2a + b} \quad \alpha = 2a + b \quad (2.58)$$

To work in terms of  $a$  and  $b$ , we must pick the desired values for these parameters, convert them to  $\alpha$  and  $\lambda$ , and then train the model with Equation 2.56.

## 2.4.2 Algorithm implementation

The implementation of this algorithm is straightforward; it is similar to the Lasso model. The only difference is in the hyperparameters we define and in the update of the beta coefficients, which are given by different equations.

```
[1]: import numpy as np
import copy as copy

class Enet():

    def __init__(self, alpha, Lambda, D):
        self.beta = np.zeros((D+1, 1))
        self.alpha = alpha
        self.Lambda = Lambda
        self.epsilon = 0.0000001

    def extend(self, X):
        M = len(X)
        N = len(X[0])
        extended_X = np.ones((M, N+1))
        extended_X[:, 1:] = X
        return(extended_X)

    def train_model(self, X, Y):
        Y = np.reshape(Y, (len(Y), 1))
        N = len(Y)

        X = self.extend(X)
        X_T = np.transpose(X)

        Q = np.sum(X*X, axis=0)
        Q = np.reshape(Q, (len(Q), 1))/N

        P = np.zeros((len(self.beta), 1))
        beta = np.zeros(np.shape(self.beta))

        constant_1 = self.alpha*(1-self.Lambda)/2
        constant_2 = self.alpha*self.Lambda

        for i in range(100000):
```

```

    for a in range(len(self.beta)):

        P[a] = np.dot(X_T[a],(Y-self.pred(X))/N
        P[a] = P[a] + Q[a]*self.beta[a]

        if (a==0):
            self.beta[a]=P[a]/Q[a]

        elif (P[a]<=constant_2):
            self.beta[a]=(P[a]+constant_2)/(Q[a]+2*constant_1)

        elif ((-constant_2<=P[a]) and (P[a]<=constant_2)):
            self.beta[a]=0

        elif (P[a]>=constant_2):
            self.beta[a]=(P[a]-constant_2)/(Q[a]+2*constant_1)

        if (np.sum(abs(self.beta-beta)) < self.epsilon):

            break

    beta = copy.deepcopy(self.beta)

    return(self.beta)

def prediction(self,X_test):

    X_test = self.extend(X_test)
    prediction = np.dot(X_test,self.beta)
    return(prediction)

def pred(self,X):

    prediction = np.dot(X,self.beta)
    return(prediction)

def MSE(self,X_test,Y_test):

    Y_test = np.reshape(Y_test,(len(Y_test),1))
    error_array = self.prediction(X_test) - Y_test
    MSE = np.sum(error_array*error_array)/len(Y_test)
    return(MSE)

```

Once we have the code, the next step is to load the data, which I will not show since I already put that code in previous sections. The elastic-net model will only be tested on the fourth dataset, including a combination of linearly dependent variables and other variables with negligible correlation with the output. I also built a cross-validation function for the Elastic-net, similar to the Lasso model's cross-validation.

```
[2]: from sklearn.model_selection import train_test_split

def cross_validation_ENET(X,Y,iterations,alpha,Lambda,D):

    MSE_values = []

    regression = Enet(alpha,Lambda,D)

    for i in range(iterations):
        X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
        test_size = 2/10, random_state = i)

        # NORMALIZATION
        X_train,Y_train,X_test,Y_test = normalization(X_train,
                                                       Y_train,X_test,Y_test)

        regression.train_model(X_train,Y_train)
        MSE = regression.MSE(X,Y)
        MSE_values.append(MSE)

    return(np.mean(MSE_values))
```

Next, I created a grid of  $\alpha$  and  $\lambda$  values on which we will test our model.  $\alpha$  ranges on the values [0, 0.6] and  $\lambda$  falls on the interval [0, 1]. Then, using the cross-validation function for the first 2000 random states, we obtain the MSE for the mesh values of the parameters. The minimum value and the parameters associated with it are the following.

```
[3]: print("Minimum value of the MSE: ",np.min(MSE))
a,b = np.where(MSE == np.min(MSE))
print("Alpha value for which the MSE is minimum: ",alpha_4[a[0]][b[0]])
print("Lambda value for which the MSE is minimum: ",Lambda_4[a[0]][b[0]])
```

```
Minimum value of the MSE:  0.3717527959183807
Alpha value for which the MSE is minimum:  0.2
Lambda value for which the MSE is minimum:  0.44
```

As we can see, the minimum value of the MSE after using the Elastic-net model is smaller than that of the Ridge and Lasso models. It leads to an MSE improvement of 18.47 %. If we take the values of  $\alpha = 0.2$  and  $\lambda = 0.44$ , we obtain  $a = 0.056$  and  $b = 0.088$ , from which we conclude that the l-1 norm hyperparameter has more weight than the l-2 norm

one, although both contributions are of the same order.

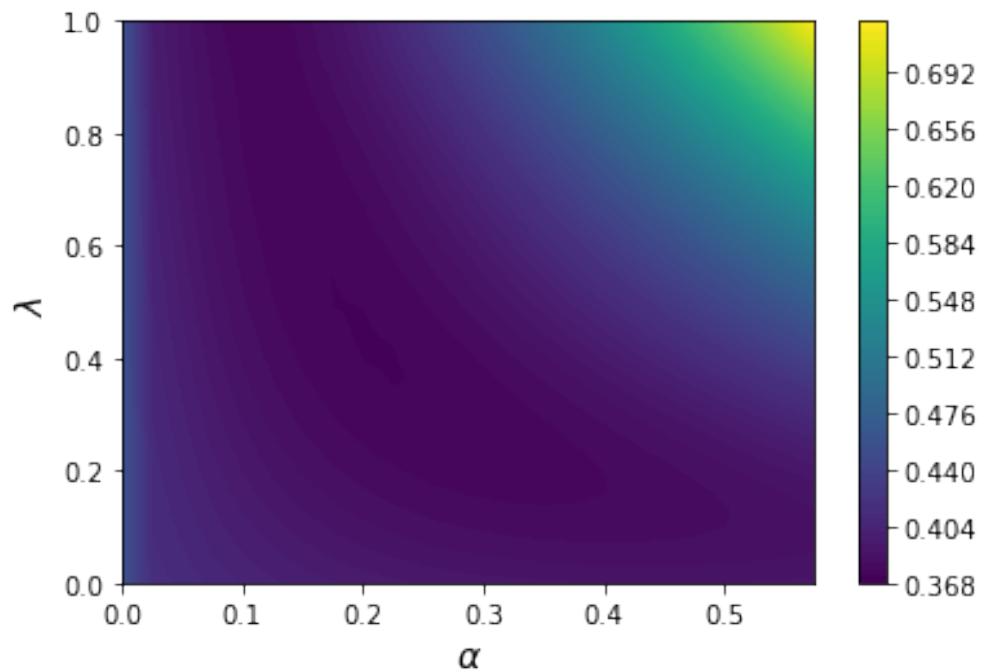


Figure 2.17: Results of the Elastic-net model for the dataset from Table 2.1.

## 2.5 Results

Data collection (and its properties)	OLS	RIDGE	LASSO	E-NET
<b>1st data collection</b> • Variable correlated to the outcome.	0.35573	-0.02 %	-0.00%	
<b>2nd data collection</b> • Variable correlated to the outcome. • Multicollinearity.	0.37555	-8.25%	-7.82%	
<b>3rd data collection</b> • Variable correlated to the outcome. • Irrelevant variables.	0.41875	-0.60%	-10.70%	
<b>4th data collection</b> • Variable correlated to the outcome. • Multicollinearity. • Irrelevant variables.	0.45591	-15.01%	-18.27%	-18.47%

Table 2.2: MSE errors of the OLS, Ridge, Lasso, and Elastic-net algorithms on the data from Table 2.1. If an algorithm excels compared to the baseline (OLS), it will be colored dark green; if it performs better, it will be light green; and if the improvement is moderate, it will be yellow.

I will summarize the results obtained after testing the OLS, Ridge, Lasso, and Elastic-net algorithms on the dataset from Table 2.1. Remember that I formed four data collections from this dataset. The first data collection contained a variable highly correlated to the outcome. The second data collection comprised this correlated variable and another three that were linearly dependent. The third data collection contained the correlated variable and another two irrelevant to the outcome. Moreover, the fourth data collection contained all the variables.

Concerning the first data collection, all the algorithms perform similarly; there is no considerable improvement by any of the models. For the second data collection, the Ridge model excels due to the multicollinearity of the variables. The lasso model also performs decently. For the third data collection, the Lasso algorithm performs way better than the Ridge due to irrelevant variables being a part of the dataset. Moreover, the Elastic-net algorithm comes ahead when we consider all the variables, although the other models perform decently.

# Chapter 3

## Other Regression Algorithms

In this chapter, we will work with three different regression algorithms. The regularized linear models we have worked with in the previous chapter adapt well to data with linear tendencies. Sometimes, we may encounter some data collection with high variance, and the linear models may struggle to adapt well. When this happens, we can consider different algorithms, such as polynomial regressions or other models, as we will see in this chapter. These three algorithms are the following:

- First, we have **ARTH**, which stands for Adaptive Regression through Hinges. It is an algorithm that will adjust to our data on segments. It will form multiple linear regressions between knots or points, allowing extra adaptability to datasets that do not contain linear tendencies.
- Next, we have the **LOESS** algorithm, also known as locally weighted linear regression. This model considers only the local data neighboring the testing data to make predictions. This allows the model to adapt to curved datasets well.
- Lastly, we have the **Bayesian linear regression**, which uses Gaussian distributions to estimate each point's prediction and the associated variance.

We will work with the [2] Ames housing dataset to illustrate the performance of these algorithms as this dataset has a high variance, and linear regression may not be the best choice when working with it. The Ames housing dataset has the prices of different houses as an outcome. Among the different features it stores, we will work with the year the houses were built. Although regular linear regression does not perform horribly, we will see that ARTH and LOESS perform better than the OLS by a considerable margin of improvement. We are not that interested in the performance of the Bayesian linear regression as we will mainly focus on the mathematical richness behind it.

We will use regular cross-validation to test the models, picking 80% of the data as training data and the remaining 20% as testing data. The models will be tested by

averaging over the first 50 random states as this is an extensive database and, hence, less sensitive to the training/testing split choice.

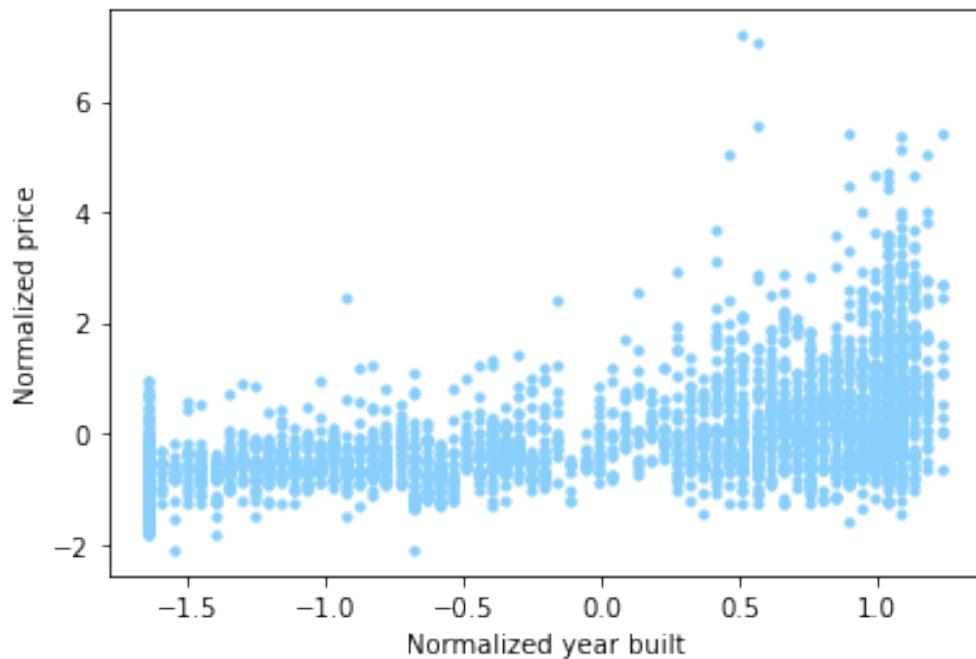


Figure 3.1: Ames housing dataset: price vs. year built.

## 3.1 Adaptive regression through hinges [Supervised]

The main idea behind this algorithm is to split the dataset into sections with boundaries determined by specific data points (knots) and then make separate linear regressions to minimize the MSE. Although implementing this model in D dimensions is straightforward, I will limit the implementation to 2D, so it is more instructive.

### 3.1.1 Mathematical derivation

Before explaining this model, it is necessary to introduce the concept of a fixed OLS regression. Let us suppose we have a point  $(x_0, y_0)$ , and we want our regression to contain this point; we can always write the regression like this:

$$\hat{y}_i = y_0 + \beta_1(x_i - x_0) \quad (3.1)$$

As we can see, the intercept  $\beta_0$  has a fixed value, ensuring that for  $x_0$ , we always get  $y_0$  as the outcome. Now, to find the value of  $\beta_1$ , all we have to do is to minimize the following quantity similarly to how we did on the OLS algorithm:

$$S(\beta) = \sum_{i=1}^N \left( y_i - [y_0 + \beta_1(x_i - x_0)] \right)^2 = \sum_{i=1}^N \left( (y_i - y_0) - \beta_1(x_i - x_0) \right)^2 \quad (3.2)$$

So now, computing the derivative:

$$\frac{\partial S(\beta)}{\partial \beta_1} = -2 \sum_{i=1}^N \left[ (y_i - y_0) - \beta_1(x_i - x_0) \right] (x_i - x_0) \quad (3.3)$$

The next step is to make the previous equation equal to zero in order to find the minimum:

$$\sum_{i=1}^N (y_i - y_0)(x_i - x_0) = \beta_1 \sum_{i=1}^N (x_i - x_0)(x_i - x_0) \quad (3.4)$$

Lastly, we isolate the  $\beta_1$  parameter:

$$\beta_1 = \frac{\sum_{i=1}^N (y_i - y_0)(x_i - x_0)}{\sum_{i=1}^N (x_i - x_0)(x_i - x_0)} \quad (3.5)$$

Once we know how to obtain an OLS regression containing a point, we can explain the different steps involved in this model [27][50].

#### Forward pass: first knot.

The first step of the algorithm is to consider every data point as a potential knot and compute a fixed OLS regression on both sides of the different potential knots. One of these knots will lead to a lower value of the MSE (Mean squared error), and we will keep

this point as our first knot. The reasoning behind the name of the algorithm is that we can always write the equation of the regression using hinge functions given by:

$$h(x - a) = \max(0, x - a) = \begin{cases} 0 & \text{for } x < a \\ x - a & \text{for } x > a \end{cases} \quad (3.6)$$

For example, let us consider the following continuous equation:

$$f(x) = \begin{cases} x + 0.16 & \text{for } x \leq 2 \\ -x + 4.16 & \text{for } 2 < x < 3.2 \\ 1.2x - 2.88 & \text{for } x \geq 3.2 \end{cases} \quad (3.7)$$

Its hinge form will be given by the following expression:

$$f(x) = 2.16 - h(2 - x) - h(x - 2) + 2.2 h(x - 3.2) \quad (3.8)$$

The hinge expression for each of the intervals reads:

$$f(x) = 2.16 - (2 - x) - 0 + 0 = 0.16 + x \quad \text{for } x < 2$$

$$f(x) = 2.16 - 0 - (x - 2) + 0 = 4.16 - x \quad \text{for } 2 < x < 3.2$$

$$f(x) = 2.16 - 0 - (x - 2) + 2.2(x - 3.2) = 1.2x - 2.88 \quad \text{for } x > 3.2$$

Despite having introduced the hinge functions, it is not necessary to use them when implementing this algorithm; one can stick to the notation and structure of Equation 3.7, as we will do.

### **Forward pass: remaining knots.**

Once we have a first knot, we have to find the following knots to improve our algorithm's accuracy further. Again, we will consider each data point as a new knot and compute the MSE of the configuration. One must remember that as we include knots, two will be on the edges (left and right). To predict the left side of the left knot, we will implement a fixed OLS, and the same goes for the right side of the right knot. For all the points between two given knots, the line that connects the knots will give the regression. The idea is simple: We consider each data point as a potential knot, then we compute the MSE of the configuration that emerges from the addition of that potential knot, and last, we keep the knot that leads to a lower error.

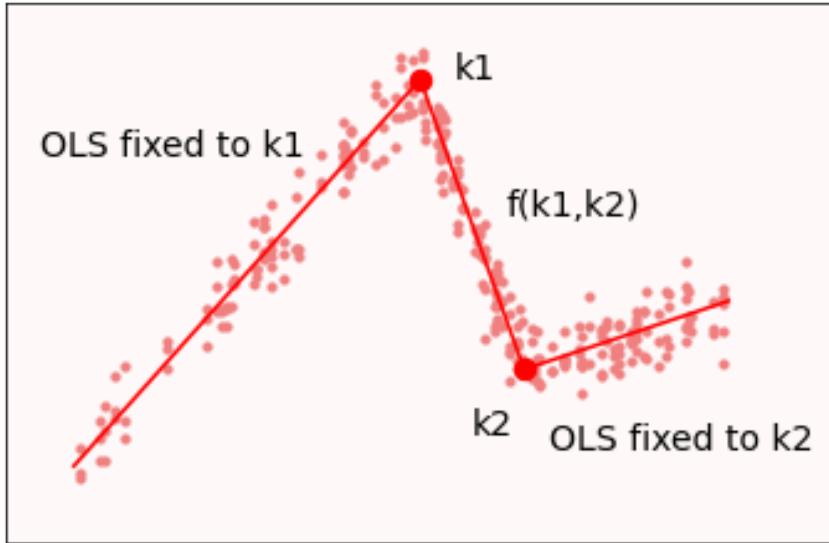


Figure 3.2: Visualization of how the *adaptive regression through hinges* algorithm works.

### Backward pass: pruning.

The next step is pruning. Remember that we do not want a massive number of knots because this will lead to an over-fit model, which may not predict new data correctly; hence, we want to remove some of these knots. One may wonder what is the logic in adding more knots to remove them later. This is because the first knots found in the previous process might not be the most relevant when more knots are added, as we will see now.

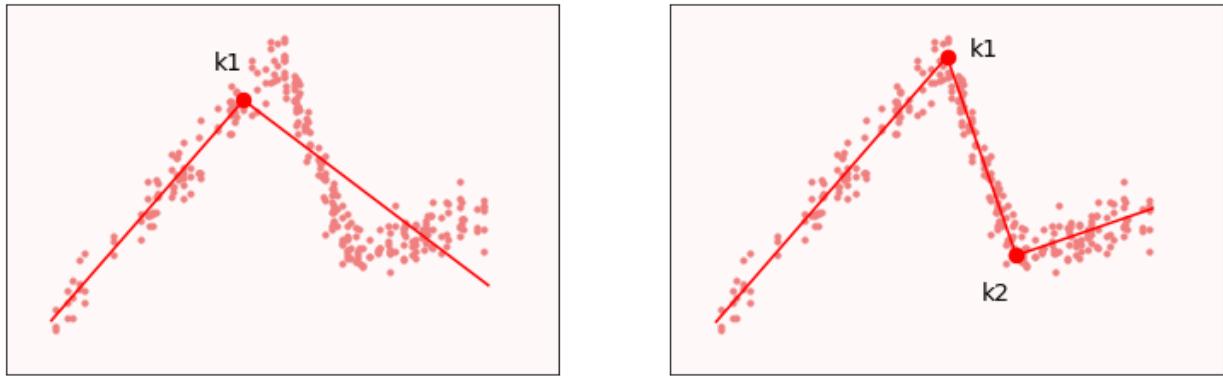


Figure 3.3: a) Single knot that leads to the lowest MSE. b) Pair of knots that lead to the lowest MSE.

On the previous graph, the left plot shows which knot leads to the lowest MSE. On the right, we can see the pair of optimal knots in reducing the MSE, which coincides with what common sense dictates. As we can see, the first point added by the algorithm is

not in the plot on the right. That is, if we train the algorithm to find the first ten knots and remove the eight less relevant ones afterward, the first knot added is not among the survivors.

So, as we just mentioned, we will add knots one by one and then remove some of them. To choose which knot to remove, we remove those knots that increase the MSE the least when removed. In order to know when to stop removing knots, we need something to base our choice on. For this, we introduce the concept of the GCV (Generalized cross-validation), which for this algorithm we can write as:

$$GCV = \frac{MSE}{\left[1 - \frac{(2+p)knots-1}{N}\right]^2} \quad (3.9)$$

The factor  $p$  is called the penalization term and is usually chosen to be 2 or 3. As we can see, the GCV penalizes the addition of knots; the more knots, the closer the denominator gets to zero, and the larger the GCV grows. This concept helps us find a good balance between adding too many knots and not enough.

What we want to do is to remove the knots one by one until the GCV is not further reduced (at first, removing knots will not impact the MSE, but the GCV will get lower by the fact that we are removing knots, but at some point, the MSE will increase considerably when we remove knots and this will lead to the GCV not decreasing further, that is when we stop removing knots).

### 3.1.2 Algorithm implementation

As we can see, this algorithm was more complex to implement than the previous ones; it required many functions to work together.

```
[1]: import numpy as np
import copy as copy

class ARTH():

    def __init__(self,X,Y,knot_number,print_result,prune = "no"):

        X = np.reshape(X,(len(X),))
        Y = np.reshape(Y,(len(Y),))
        Y = [x for _,x in sorted(zip(X,Y))]
        self.Y = np.reshape(Y,(len(Y),1))
        self.X = np.reshape(np.sort(X),(len(X),1))

        self.knot_number = knot_number
        self.print_result = print_result
        self.prune=prune
        self.penalty = 2.0

        self.knot_index = np.array([], dtype=np.int64)
        self.knot_x = np.array([])
        self.knot_y = np.array([])
        self.beta = np.array([])
```

One thing that made coding this algorithm easier was to have the data ordered from lower to higher on the x-axis; for this reason, on the *init* function of the algorithm, I sorted X from the lowest value to the highest and Y accordingly. The sorting of the variables only worked when these were one-dimensional arrays, so I had to reshape them before sorting them and then again after sorting them. There is also a variable that controls the knot number of the model, a penalty term for the GCV, and four different variables that store the knots' data values, the knots' indexes, and the beta coefficients of the regression. There is also an input variable called *print-result*, which allows us to decide whether the result is printed.

```
def fixed_OLS(self,X,Y,fixed_index):

    X_T = np.transpose(X)

    fixed_X = self.X[fixed_index][0]

    if (np.std(X) < 10**-10) or fixed_index == 0 or
```

```

    fixed_index == len(self.X)-1:

beta = (10.0**60.0)*np.ones((2,1))

else:

B_0 = self.Y[fixed_index][0]
Quantity_1=np.linalg.inv(np.dot(X_T-fixed_X,X-fixed_X))
Quantity_2=np.dot(X_T - fixed_X, Y- B_0)
B_1= np.dot(Quantity_1,Quantity_2)[0][0]

beta = np.reshape([B_0,B_1],(2,1))

Y_pred = (X - fixed_X)*beta[1] + beta[0]
error = Y - Y_pred
MSE = np.sum(error*error)

return(MSE,beta)

```

This function computes a fixed OLS regression; it has the X and Y variable portions to be used on the regression and the index of the point the regression is fixed to as input. At the start, there is an *if* statement that sets an absurd value for the coefficients in case the standard deviation of X is close to zero. This was to avoid a conflict when computing the inverse of the matrix in *Quantity*<sub>1</sub> in case all the values on X were equal (imagine we take the third data point as a knot and the first three values share the exact value of X, for example). I also avoided computing the OLS regression on the first and last points since a linear regression on a single point does not make sense.

```

def knot_lines(self,a,b):

if (self.X[a]==self.X[b]):

beta = (10.0**60.0)*np.ones((2,1))

else:

delta_X = self.X[b][0]-self.X[a][0]
delta_Y = self.Y[b][0]-self.Y[a][0]

B_0 = self.Y[a][0] - (delta_Y* self.X[a][0] /delta_X)
B_1 = delta_Y / delta_X
beta = np.reshape([B_0,B_1],(2,1))

```

```

X_cut = self.X[a:b,:]
Y_cut = self.Y[a:b,:]

Y_pred = (X_cut)*beta[1] + beta[0]
error = Y_cut - Y_pred
MSE = np.sum(error*error)

return(MSE,beta)

```

The *knot-lines* function computes the line that connects two different knots given by the indexes  $a$  and  $b$ . There is also an *if* statement on the function, which avoids knots with the same value of  $X$ , as this would lead to an infinite slope. This part of the code is straightforward: We compute the intercept and the slope ( $\beta_1$ ) and then make predictions on the data points between the knots to estimate the MSE.

```

def regression(self,knot_index):

    beta_list = np.array([])
    knot_index = np.sort(knot_index)
    a = knot_index[0]
    z = knot_index[-1]

    MSE = 0

    MSE_a,beta_a = self.fixed_OLS(self.X[:a],self.Y[:a],a)
    MSE = MSE + MSE_a
    beta_list = np.append(beta_list,beta_a)

    if (a!=z):

        for j in range(len(knot_index)-1):

            MSE_j,beta_j = self.knot_lines(knot_index[j],
                                            knot_index[j+1])
            MSE = MSE + MSE_j
            beta_list = np.append(beta_list,beta_j)

        MSE_z,beta_z = self.fixed_OLS(self.X[z:],self.Y[z:],z)
        MSE = MSE + MSE_z
        beta_list = np.append(beta_list,beta_z)

    return(MSE,beta_list)

```

The *regression* function takes a list of knots and computes the MSE and the beta coefficients using the training dataset. First, it sorts the knots from left to right (smallest index to largest), then it labels the left knot as  $a$  and the right knot as  $z$ . It computes a fixed OLS regression with  $a$  as the fixing point. Whenever  $a$  is not equal to  $z$ , i.e., there is more than one knot, it calls for the *knot-lines* function to make the regression on the section between the knots. Finally, it computes another fixed OLS regression with the right knot as the fixing point.

```
def forward(self):

    X = self.X
    Y = self.Y

    for k in range(self.knot_number):

        MSE_list = np.array([])
        MSE_list = np.append(MSE_list, 10.0**60.0)

        for i in range(len(X)-2):
            i = i+1
            if (k==0):

                knot_index = np.array([i])

            else:

                knot_index = copy.deepcopy(self.knot_index)
                knot_index = np.append(knot_index,i)

        MSE = self.regression(knot_index)[0]

        MSE_list = np.append(MSE_list,MSE)

        min_i = np.where(MSE_list == min(MSE_list))[0][0]

        self.knot_index = np.append(self.knot_index,min_i)

    return(self.knot_index)
```

The *forward* function is one of the main parts of the code; as the name suggests, it contains the forward pass of the algorithm. It starts an iteration with a length equal to the requested number of knots. For each of these iterations, there is another iteration

through all the data points (except for the first and last). On the first iteration, as the variable *knot-index* is not defined yet, the function saves the different data indexes one by one on the array. The MSE is computed using the *regression* function, and the index leading to the lowest MSE is appended to the *self.knot-index* variable. The remaining iterations are similar, but the potential knots are appended to a copy of the *self.knot-index* variable, which contains the knots of previous iterations. I appended a value of  $10^{60}$  to the list that saves the MSE values at the start of the function to compensate for not including the first data point on the iteration (so I did not have to shift the index).

```
def pruning(self,knots):

    base_MSE = self.regression(knots)[0]
    base_GCV = self.GCV(base_MSE,len(knots))

    MSE_list = np.array([])

    for i in range(len(knots)):

        knots_copy = copy.deepcopy(knots)
        knots_copy = np.delete(knots_copy,i)

        MSE = self.regression(knots_copy)[0]
        MSE_list = np.append(MSE_list,MSE)

    min_i = np.where(MSE_list == min(MSE_list))[0][0]
    min_MSE = MSE_list[min_i]
    min_GCV = self.GCV(min_MSE,len(knots)-1)

    if (min_GCV < base_GCV):
        self.knot_index = np.delete(knots,min_i)

    return(self.knot_index)
```

The *pruning* function will start computing the baseline MSE and GCV (the values before the pruning of a knot). It iterates through all different knots and removes them one by one from a copy of the array containing them; then it computes the MSE and GCV that result from removing the knots and appends them to a list. Lastly, if the GCV associated with the minimum of the MSE is lower than the baseline GCV, the knot that leads to that configuration is deleted. If that is not the case, the whole knot list is returned.

```
def GCV(self,MSE,knot_number):

    efective_parameters = (2.0+self.penalty)*knot_number-1
```

```

GCV_value = MSE/((1-(effective_parameters/len(self.X)))*2.0)

return(GCV_value)

def training(self):

    knots = self.forward()

    for iterations in range(len(knots)-1):
        if (self.prune == "yes"):
            self.knot_index = self.pruning(self.knot_index)

    self.knot_index = np.sort(self.knot_index)
    self.knot_x,self.knot_y,self.beta = self.results(self.knot_index)

    return(self.knot_x,self.knot_y,
           np.reshape(self.beta,(len(self.knot_x)+1,2)))

```

The *training* function is the core of the algorithm. It starts by computing the knots by calling the *forward* function. Then, it calls for the *pruning* function (if specified by the input variable *prune*). As we can see, the number of iterations equals the number of knots minus one, so we will end up with at least one knot after the pruning pass. Last, it calls for the *results* function, which prints the results of the model if the user specifies it.

```

def predict(self,x_test):

    x_test = np.reshape(np.sort(x_test),(len(x_test),1))

    y_pred = np.array([])
    B = np.reshape(self.beta,(len(self.knot_index)+1,2))

    for i in range(len(x_test)):

        if x_test[i] < self.knot_x[0]:
            ypred = B[0][0] + B[0][1]*(x_test[i]-self.knot_x[0])
            y_pred = np.append(y_pred,ypred)

        elif x_test[i] >= self.knot_x[-1]:
            ypred = B[-1][0] + B[-1][1]*(x_test[i]-self.knot_x[-1])
            y_pred = np.append(y_pred,ypred)

        else:

```

```

    for k in range(len(self.knot_index)-1):

        if (self.knot_x[k]<=x_test[i] ) and \
        (self.knot_x[k+1]>x_test[i]):

            ypred = B[k+1][0] + B[k+1][1]*x_test[i]
            y_pred = np.append(y_pred,ypred)

    return(y_pred)

```

The *predict* function is quite simple; it runs through the values of the  $X_{test}$  variable and checks the relative position to the knots, then makes a prediction using the beta coefficients associated with the interval containing the test points.

```

def MSE(self,x_test,y_test):

    y_test = [x for _,x in sorted(zip(x_test,y_test))]
    x_test = np.reshape(np.sort(x_test),(len(x_test),1))

    y_pred = self.predict(x_test)

    MSE = np.sum((y_pred - y_test)*(y_pred - y_test))/len(y_pred)

    return(MSE)

def results(knots):

    MSE,beta = self.regression(knots)
    MSE = MSE/len(self.Y)
    GCV = self.GCV(MSE,len(knots))
    beta = np.reshape(beta,(len(knots)+1,2))
    knots_x = np.array([self.X[i] for i in knots])
    knots_y = np.array([self.Y[i] for i in knots])

    if (self.print_result=="no"):

        return(np.transpose(knots_x)[0],
               np.transpose(knots_y)[0],
               np.reshape(beta,(len(beta)*2,)))

    print("-----RESULTS-----")
    print("    MSE = ",np.around(MSE,4),
          " and    GCV = ",np.around(GCV,4))
    print("-----")

```

```

print("      x < ",np.around(knots_x[0][0],3)," | ",
      np.around(beta[0][0],3)," + ",
      np.around(beta[0][1],3)," (x-x0)")

for j in range(len(knots)-1):

    print(np.around(knots_x[j][0],3),"< x < ",
          np.around(knots_x[j+1][0],3)," | ",
          np.around(beta[j+1][0],3)," + ",
          np.around(beta[j+1][1],3)," x")

print("      x > ",np.around(knots_x[-1][0],3)," | ",
      np.around(beta[-1][0],3)," + ",
      np.around(beta[-1][1],3)," (x-x0)")

return(np.transpose(knots_x)[0],
       np.transpose(knots_y)[0],
       np.reshape(beta,(len(beta)*2,)))

```

Once the code is ready, we can start testing it on a dataset; first, we will start by testing it on a simple dataset, which is a dispersion around three lines that form a continuous equation.

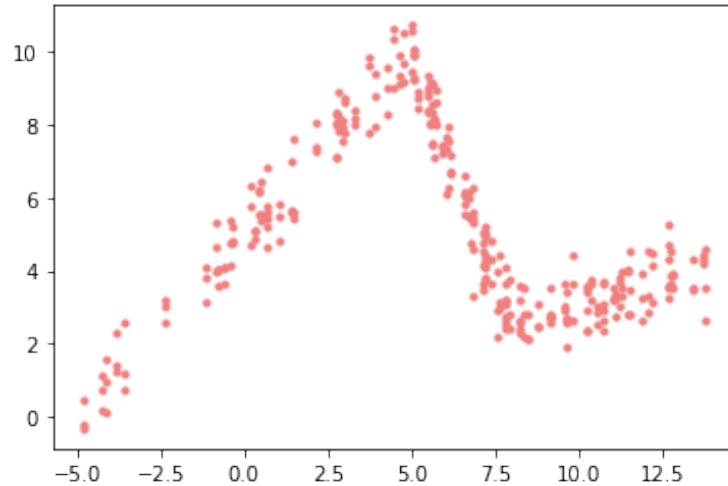


Figure 3.4: Plotting of the dataset created out of the dispersion of three continuous lines.  $y = x + 5$  for  $x \in [-5, 5]$ ,  $y = 22.5 - 2.5x$  for  $x \in [5, 8]$  and  $y = 0.1 + 0.3x$  for  $x \in [8, 14]$ .

If we execute the following piece of code, requesting six different knots to be obtained and both enabling and not enabling the pruning pass:

```
[2]: import matplotlib.pyplot as plt

x_test = np.arange(-5,14,0.1)

reg = ARTH(X,Y,6,print_result = "yes", prune = "no")
knot_x, knot_y, beta = reg.training()
ypred = reg.predict(x_test)

reg_2 = ARTH(X,Y,6,print_result = "yes", prune = "yes")
knot_x_2, knot_y_2, beta_2 = reg_2.training()
ypred_2 = reg_2.predict(x_test)

fig, ax = plt.subplots(1, 2, figsize=(12.8,4.0))

ax[0].scatter(X,Y,color="lightcoral",s=10)
ax[0].scatter(knot_x,knot_y,color="red",s=50,zorder=10)
ax[0].plot(x_test,ypred,"red")

ax[1].scatter(X,Y,color="lightcoral",s=10)
ax[1].scatter(knot_x_2,knot_y_2,color="red",s=50,zorder=10)
ax[1].plot(x_test,ypred_2,"red")
```

```
-----RESULTS-----
MSE = 0.3746 and GCV = 0.4393
-----
      x < 3.326    | 8.36 + 0.998 (x-x0)
3.326 < x < 4.447 | 6.465 + 0.57 x
        4.447 < x < 4.786 | 0.091 + 2.003 x
        4.786 < x < 7.784 | 20.045 + -2.166 x
        7.784 < x < 8.341 | 8.919 + -0.737 x
        8.341 < x < 9.557 | 2.011 + 0.092 x
          x > 9.557    | 2.885 + 0.294 (x-x0)
-----RESULTS-----
MSE = 0.3723 and GCV = 0.4011
-----
      x < 4.786    | 9.678 + 0.974 (x-x0)
4.786 < x < 7.784 | 20.045 + -2.166 x
        7.784 < x < 8.341 | 8.919 + -0.737 x
          x > 8.341    | 2.774 + 0.226 (x-x0)
```

As we can see, pruning decreases the value of the GCV due to decreasing the number of knots. As mentioned, general cross-validation (pruning phase) helps us avoid making an overfit model and properly tuning our parameters. It would be the equivalent of testing the different  $\alpha$  parameters on the Ridge model. Of course, we will also apply the usual cross-validation techniques by testing the model on different dataset splits. We can also

see that in this case, the pruning reduces the value of the MSE on the training data, which might not always be the case as the pruning is meant to sacrifice bias for variance. Below, we can see the plot of these results.

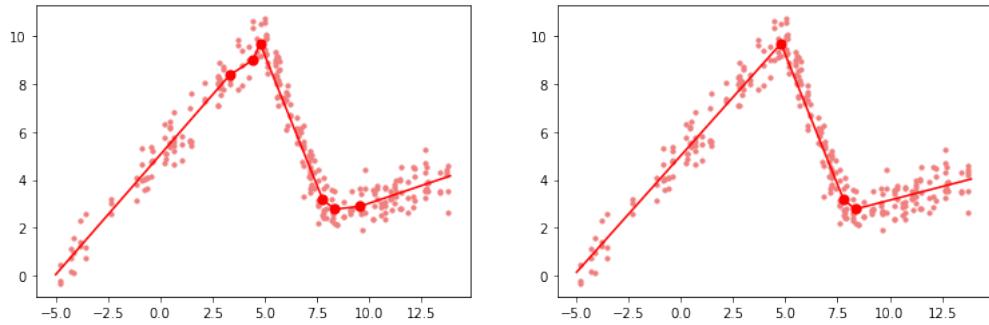


Figure 3.5: ARTH regression on the data from Figure 3.4, requesting six knots with and without pruning.

We can also see how the model starts building the different knots on the plot below; as we can see, the regressive line adapts to the nature of the scatter data as we increase the knots.

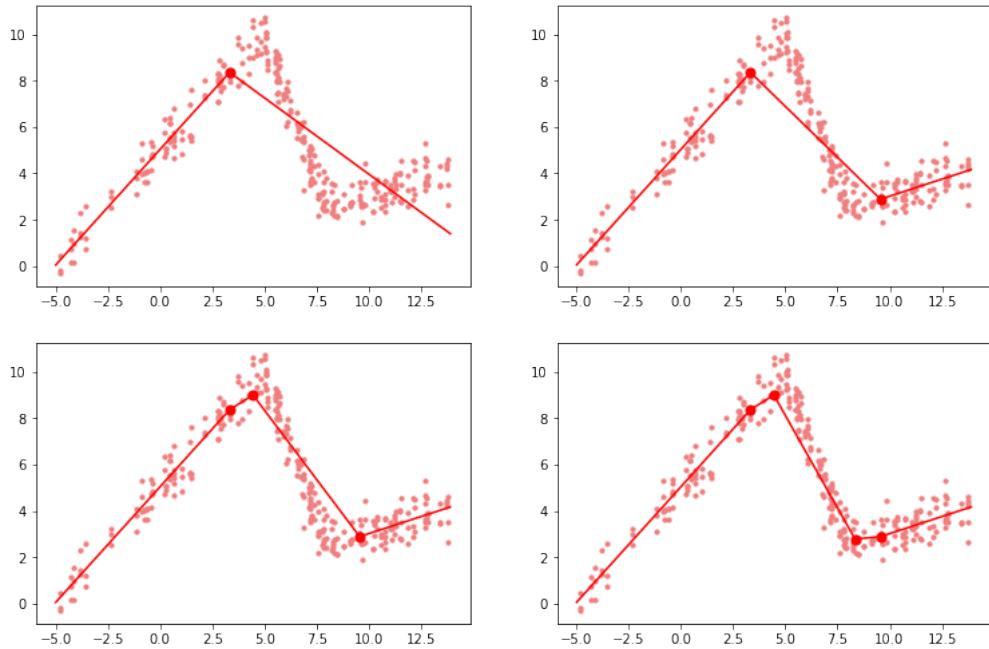


Figure 3.6: ARTH regression on the data from Figure 3.4, evolution from one knot to four knots.

The next step is to work with the Ames housing dataset. Below, we can see the results and the plot of applying the model with pruning for 10 and 100 knots. Keep in mind that here, I normalized the data as a whole since we just want to see how the model works,

and normalizing makes the scale easier to interpret. This will not be the case whenever we test the model's accuracy, as we do not want information leakage from the testing data to the training data.

RESULTS	
MSE = 0.691	and GCV = 0.6982
<hr/>	
$x < 0.371$	$  0.115 + 0.466 (x-x_0)$
$0.371 < x < 0.419$	$  -2.112 + 6.006 x$
$0.419 < x < 0.754$	$  0.684 + -0.671 x$
$0.754 < x < 0.946$	$  -0.413 + 0.783 x$
$x > 0.946$	$  0.328 + 3.573 (x-x_0)$
RESULTS	
MSE = 0.6842	and GCV = 0.7008
<hr/>	
$x < -0.732$	$  -0.361 + 0.477 (x-x_0)$
$-0.732 < x < -0.684$	$  -4.661 + -5.875 x$
$-0.684 < x < -0.157$	$  0.032 + 0.985 x$
$-0.157 < x < -0.109$	$  -2.024 + -12.142 x$
$-0.109 < x < 0.083$	$  -0.169 + 4.929 x$
$0.083 < x < 0.371$	$  0.277 + -0.435 x$
$0.371 < x < 0.419$	$  -2.112 + 6.006 x$
$0.419 < x < 0.802$	$  0.71 + -0.733 x$
$0.802 < x < 0.994$	$  -1.211 + 1.661 x$
$x > 0.994$	$  0.441 + 4.102 (x-x_0)$

Increasing the number of knots led to a final configuration with more knots after pruning, which increased the GCV but lowered the MSE on the training data. This happens because when we request a low number of knots, we may never add relevant knots that capture the essence of the data. Ideally, we want to set a large number of knots and then let the pruning phase remove the overfitting of our model.

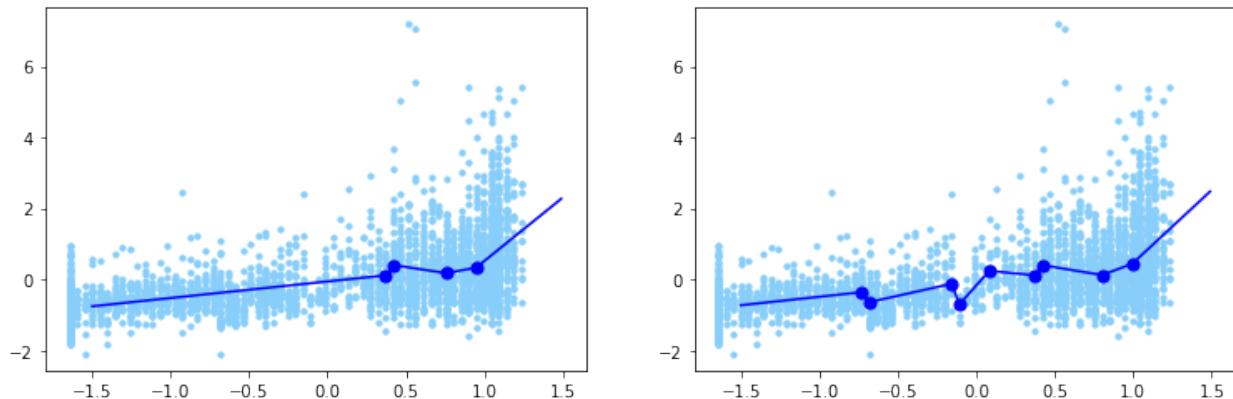


Figure 3.7: ARTH regression on the Ames Housing dataset for 10 knots and 100 knots with pruning.

The last step is to implement a cross-validation function, which splits the dataset into training (80%) and testing data (20%) for a desired number of iterations. It averages the MSE of the testing data across a number of random states.

```
[3]: from sklearn.model_selection import train_test_split

def cross_validation_ARTH(X,Y,iterations,knots):
    MSE_values = []
    for i in range(iterations):

        X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
        test_size = 2/10, random_state = i)

        # NORMALIZATION
        X_train,Y_train,X_test,Y_test = normalization(X_train,
                                                       Y_train,X_test,Y_test)

        regression = ARTH(X_train,Y_train,knots,"no","yes")
        regression.training()
        MSE = regression.MSE(X_test,Y_test)
        MSE_values.append(MSE)

    return(np.mean(MSE_values))
```

As mentioned, our knot selection has already been done by implementing the pruning. There is no need to test the hyper-parameter of the model (knot number); the pruning phase will do this task. We need to ensure a high initial number of knots. We will execute the cross-validation for a range of knot values to see how the MSE converges to a value when the knots are large enough. This will be done for the first 50 random states as the dataset has a large number of data points and, hence, is stable. It will be good enough to understand the model's performance.

```
[4]: knots = np.arange(10,90,10)
MSE = np.array([cross_validation_ARTH(X,Y,iterations=50,
                                      knots=knots_i) for knots_i in knots])
print(np.min(MSE))
```

```
0.6806292633024489
```

We find an MSE of 0.6806. The next step is to compare this result with the result provided by the OLS to see if there is any improvement.

```
[5]: print(cross_validation_OLS(X,Y,50))
```

```
0.7012784487188155
```

As we can see, the ARTH model reduces the MSE by 2.94% compared to the OLS.

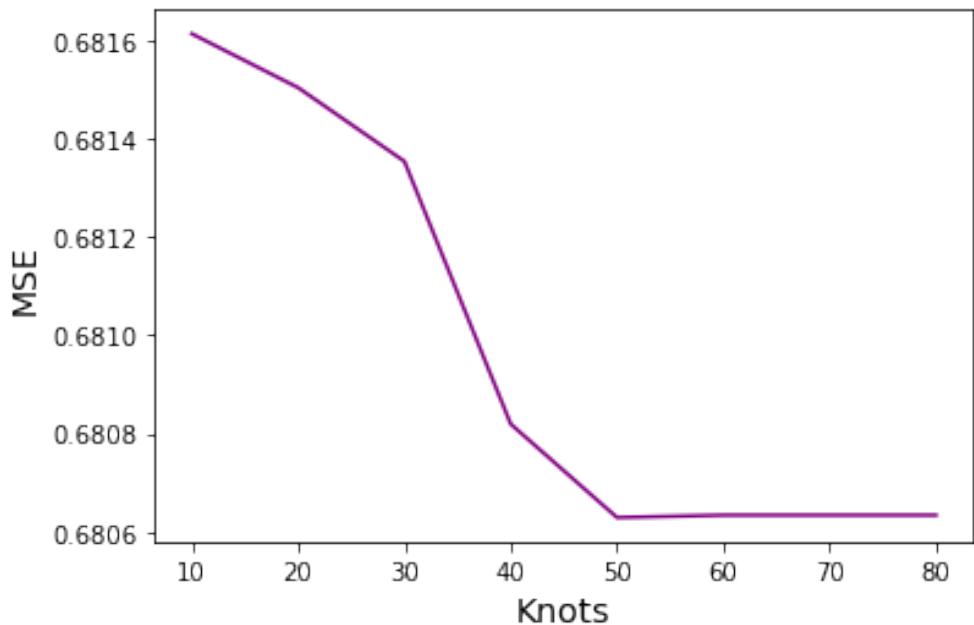


Figure 3.8: MSE on the Ames Housing dataset vs the initial knot number with pruning enabled.

## 3.2 Locally weighted linear regression [Supervised]

### 3.2.1 Mathematical derivation

[36] The general idea of this model is to perform a localized regression for each point we want to predict, where only the neighboring training data will influence this regression. For each point we want to predict, we will make a polynomial regression where the influence of the surrounding points will decrease as the distance from the point increases. To control the influence of the neighboring points, we will use a weight function that will decrease as we move away from the point.

Let us translate all this into an equation. Suppose there is a point  $X_0$  that we want to predict, and we have a set of training data  $\{X_i\}_{i=1}^N$ . We can always write the prediction of this value in terms of the coefficients of the polynomial regression of order  $k$  as  $Y_0 = \beta_0^{(0)} + \beta_1^{(0)}X_0 + \beta_2^{(0)}X_0^2 + \dots + \beta_k^{(0)}X_0^k$ . Now, in order to find the coefficients of the regression, we will minimize the following quantity:

$$MSE^{(0)} = \frac{1}{N} \sum_{i=1}^N w(X_i - X_0) [Y_i - \beta_0^{(0)} - \beta_1^{(0)}X_i - \beta_2^{(0)}X_i^2 - \dots - \beta_k^{(0)}X_i^k]^2 \quad (3.10)$$

Where this weight function  $w(X_i - X_0)$  must be a function that decreases as the distance between the points increases. As we can see, the quantity we want to minimize is the MSE, but only considering the contribution of those training points closer to the point we want to predict. So, these coefficients we will obtain will be such that the MSE of the points adjacent to  $X_0$  will be small, hence a good prediction for  $X_0$ . The weight function that I will be using is a Gaussian of the form:

$$w(X_i - X_j) = e^{-\frac{(X_i - X_j)^2}{2\sigma^2}} \quad (3.11)$$

I will call the parameter  $\sigma$  instead of  $\sigma^2$  for simplicity when implementing the algorithm. Now, the next step is to minimize the MSE associated to the point  $X_0$ , i.e., the MSE built out of the contribution of its surrounding training data.

$$\frac{MSE^{(0)}}{\beta_m} = \frac{-2}{N} \sum_{i=1}^N w(X_i - X_0) [Y_i - \beta_0^{(0)} - \beta_1^{(0)}X_i - \beta_2^{(0)}X_i^2 - \dots - \beta_k^{(0)}X_i^k] X_i^m = 0 \quad (3.12)$$

We have an equation for each of the values  $m = 0, 1, 2 \dots k$ , which can also be written as:

$$\begin{aligned} & \left[ \sum_{i=1}^N w(X_i - X_0) X_i^m \right] \beta_0^{(0)} + \left[ \sum_{i=1}^N w(X_i - X_0) X_i^{m+1} \right] \beta_1^{(0)} + \dots + \\ & \left[ \sum_{i=1}^N w(X_i - X_0) X_i^{m+k} \right] \beta_k^{(0)} = \sum_{i=1}^N w(X_i - X_0) Y_i X_i^m \end{aligned} \quad (3.13)$$

These results can be summarized using matrix notation by  $\mathbb{M}\beta^{(0)} = \mathbb{V}$  where:

$$\mathbb{M} = \begin{pmatrix} \sum_{i=1}^n w(X_i - X_0) & \sum_{i=1}^n w(X_i - X_0)X_i & \cdots & w(X_i - X_0)X_i^k \\ \sum_{i=1}^n w(X_i - X_0)X_i & \sum_{i=1}^n w(X_i - X_0)X_i^2 & \cdots & w(X_i - X_0)X_i^{k+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^n w(X_i - X_0)X_i^k & \sum_{i=1}^n w(X_i - X_0)X_i^{k+1} & \cdots & w(X_i - X_0)X_i^{2k} \end{pmatrix} \quad (3.14)$$

$$\beta^{(0)} = \begin{pmatrix} \beta_0^{(0)} \\ \beta_1^{(0)} \\ \beta_2^{(0)} \\ \vdots \\ \beta_k^{(0)} \end{pmatrix} \quad \mathbb{V} = \begin{pmatrix} \sum_{i=1}^n w(X_i - X_0)Y_i \\ \sum_{i=1}^n w(X_i - X_0)Y_iX_i \\ \sum_{i=1}^n w(X_i - X_0)Y_iX_i^2 \\ \vdots \\ \sum_{i=1}^n w(X_i - X_0)Y_iX_i^k \end{pmatrix} \quad (3.15)$$

All that is left to do is to solve  $\beta^{(0)} = \mathbb{M}^{-1}\mathbb{V}$ . So for every  $X_0$  that we want to predict, we need to build the matrices  $\mathbb{M}$  and  $\mathbb{V}$  taking the values of all the training data into consideration, then find the coefficients and predict  $Y_0 = \beta_0^{(0)} + \beta_1^{(0)}X_0 + \beta_2^{(0)}X_0^2 + \cdots + \beta_k^{(0)}X_0^k$ .

### 3.2.2 Algorithm implementation

Next, we will see the implementation of the LOESS model, which will include polynomial regressions of any order.

```
[1]: from numpy.linalg import inv
import numpy as np

class LOESS():

    def __init__(self,X,Y,sigma,order):

        self.X = np.array(X)
        self.Y = np.array(Y)
        self.sigma = sigma
        self.order = order

    def gaussian(self,X,X0):

        return (np.exp(-(X-X0)**2/(2*self.sigma)))
```

The *build-matrices* function builds the  $\mathbb{M}$  and  $\mathbb{V}$  matrices for each test point. It also returns an array with the powers of the test point that will be needed later to predict the outcome of this point. This has been relatively easy to implement because NumPy allows us to calculate matrix powers (the powers of the different matrix elements).

```
def build_matrices(self,X_test):

    M = np.zeros((self.order+1,self.order+1))
    V = np.zeros((self.order+1,1))
    X_test_powers = np.zeros((1,self.order+1))

    for i in range(len(M)):

        V[i] = np.sum(self.gaussian(self.X,X_test)*self.Y*(self.X**i))
        X_test_powers[0][i] = X_test**i

        for j in range(len(M[i])):

            M[i][j] = np.sum(self.gaussian(self.X,
                                            X_test)*(self.X**((i+j))))


    return(M,V,X_test_powers)
```

The `predict` function calls the `build_matrices` function for each of the elements in the collection of test points. It then calculates the beta coefficients and computes a prediction for the test points given by a scalar product between the powers of these test points and the regression coefficients.

```

def predict(self,X_test):

    y_pred = np.zeros((len(X_test),1))

    for i in range(len(X_test)):

        M, V , X_test_powers = self.build_matrices(X_test[i])
        beta = inv(M).dot(V)

        y_pred[i][0] = X_test_powers.dot(beta)

    return(y_pred)

def MSE(self,X_test,Y_test):

    Y_test = np.reshape(Y_test,(len(Y_test),1))
    y_pred = self.predict(X_test)

    MSE = np.sum((Y_test-y_pred)**2.0)/len(Y_test)

    return(MSE)

```

We will start testing the model on a scattering of points around the sine function. One of the things we have yet to mention is the influence of the sigma parameter of the Gaussian in charge of assigning the weights of the neighboring points. As mentioned above, the weights are given by:

$$w(X_i - X_j) = e^{\frac{-(X_i - X_j)^2}{2\sigma^2}} \quad (3.16)$$

As we can see, if we increase the size of the  $\sigma$  parameter, the negative exponentials become smaller, increasing the weights and, therefore, the relevance range of the neighboring points. The opposite happens when we decrease the value of  $\sigma$ . We will see this in the sine function example.

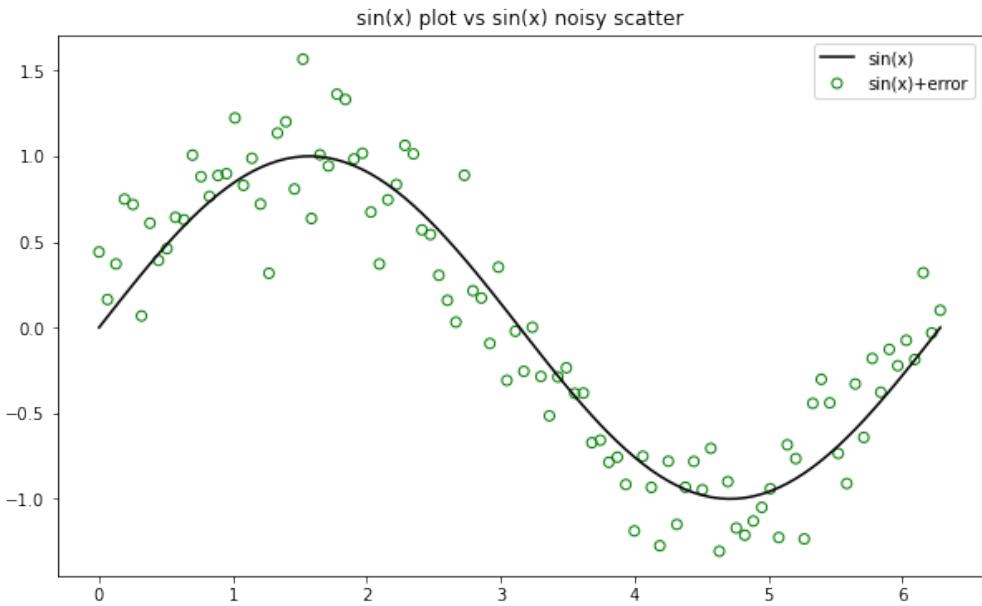


Figure 3.9: Scattering of points around the sine function.

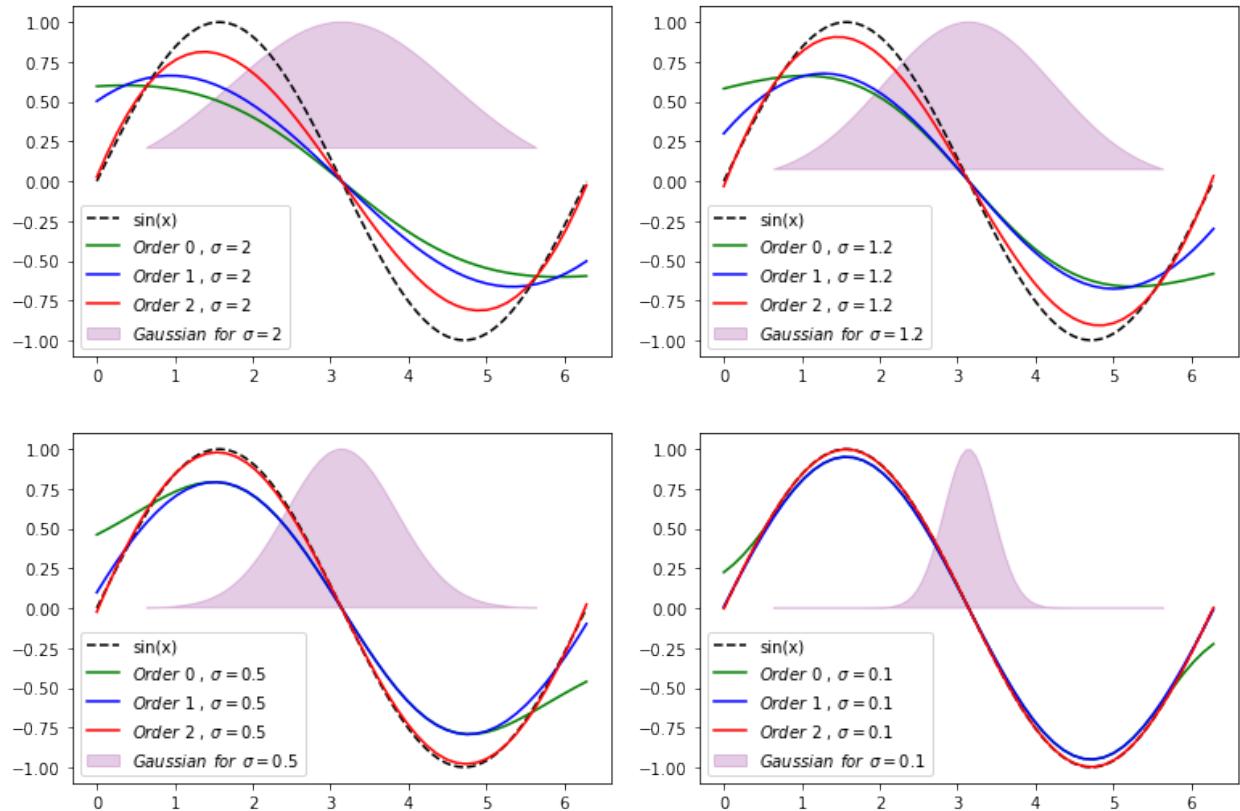


Figure 3.10: LOESS regression of order zero, one, and two for different values of the  $\sigma$  parameter.

As we can see, as we decrease the value of sigma, the range of the Gaussian weight function also decreases, which can be seen in the Gaussians in Figure 3.10. As the Gaussians are narrower, the regressions fit the sine function better. Next, we will see in another example that using a very large and a very small  $\sigma$  is counterproductive for our model. It is necessary to find a balance. To see this, we recover the Ames housing dataset once again.

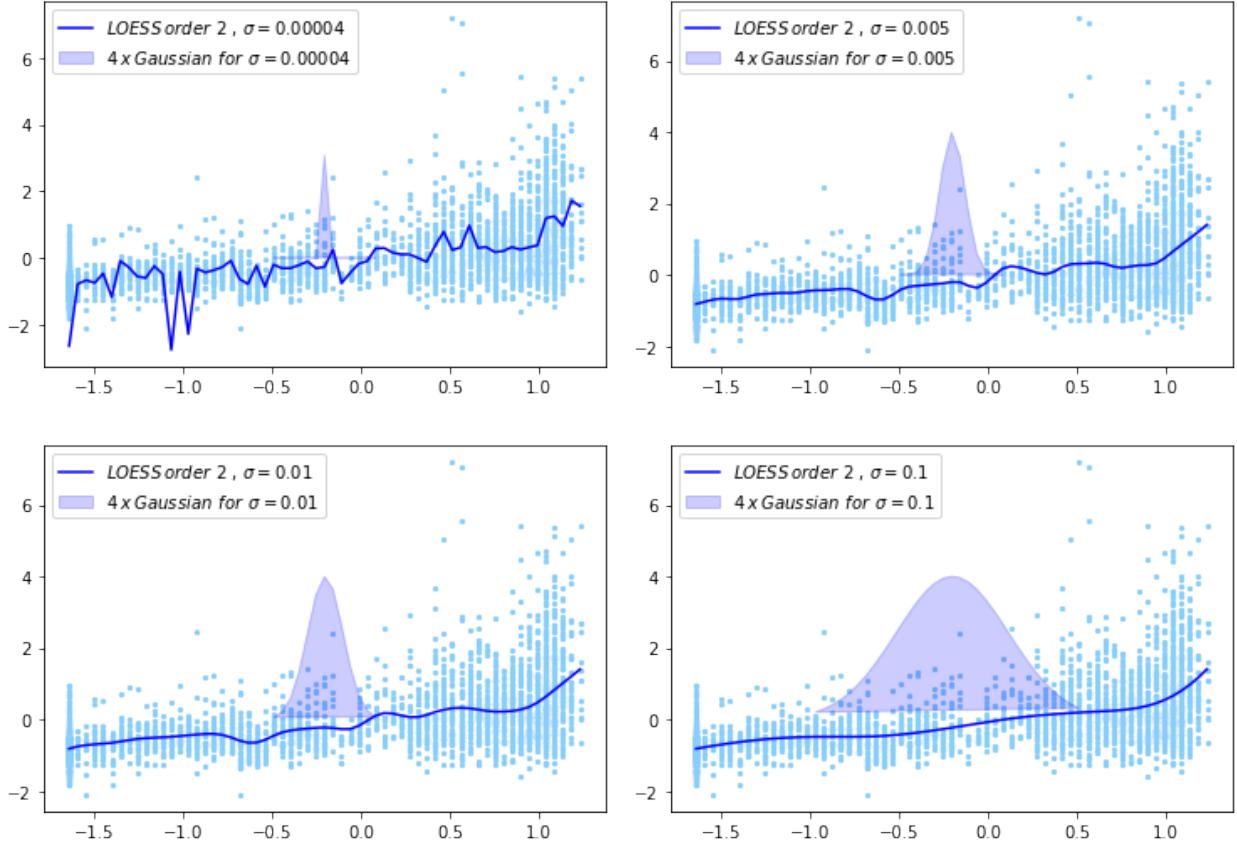


Figure 3.11: LOESS regression of order two for different values of the  $\sigma$  parameter.

As shown, spikes are generated in the plot when the sigma parameter is too small because most of the contribution to the prediction of a test point is given by a few training points. As we increase the  $\sigma$  value, the curve becomes smoother. If  $\sigma$  is too large, the model will slowly approach the OLS model, as all training points will contribute to the regression similarly.

Once we have an intuition of how the model evolves in terms of  $\sigma$ , we can test the model on this dataset by building a cross-validation function as we have done with previous models:

```
[2]: from sklearn.model_selection import train_test_split

def cross_validation_LOESS(X,Y,sigma,order,iterations):
    MSE_values = []
    for i in range(iterations):

        X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
        test_size = 2/10, random_state = i)

        # NORMALIZATION
        X_train,Y_train,X_test,Y_test = normalization(X_train,
                                                       Y_train,X_test,Y_test)

        regression = LOESS(X_train,Y_train,sigma,order)
        MSE = regression.MSE(X_test,Y_test)
        MSE_values.append(MSE)

    return(np.mean(MSE_values))
```

The next step is to generate a list of  $\sigma$  values to test the model. I chose to generate a distribution of points using an exponential function to cover values on all the decimal ranges.

```
[3]: sigma = np.array([0.1*np.exp(i*0.5)*0.000001 \
                     for i in range(31)])
sigma_2 = sigma[10:]

MSE_0 = np.zeros((len(sigma)))
MSE_1 = np.zeros((len(sigma_2)))
MSE_2 = np.zeros((len(sigma_2)))

for i in range(len(sigma)):
    MSE_0[i] = cross_validation_LOESS(X,Y,sigma[i],0,50)
for i in range(len(sigma_2)):
    MSE_1[i] = cross_validation_LOESS(X,Y,sigma_2[i],1,50)
    MSE_2[i] = cross_validation_LOESS(X,Y,sigma_2[i],2,50)
```

I used the cross-validation function, averaging the results over the first 50 random states on the zeroth, first and second order of the weighted regression. After the results were gathered, I printed the minimum value of the MSE and the associated  $\sigma$  parameter. I also plotted all the results to see how the MSE evolves in terms of  $\sigma$  for these different orders.

```
[4]: print("The minimum value of the MSE for the zeroth order is: ",
      min(MSE_0)," which is given for sigma = ",
      sigma[np.where(MSE_0 == min(MSE_0))[0]])
```

```

print("The minimum value of the MSE for the first order is: ",
      min(MSE_1)," which is given for sigma = ",
      sigma_2[np.where(MSE_1 == min(MSE_1))][0])
print("The minimum value of the MSE for the second order is: ",
      min(MSE_2)," which is given for sigma = ",
      sigma_2[np.where(MSE_2 == min(MSE_2))][0])

```

```

The minimum value of the MSE for the zeroth order is:  0.6775792906911764
which is given for sigma =  0.0022026465794806717
The minimum value of the MSE for the first order is:  0.6778432950619355
which is given for sigma =  0.005987414171519782
The minimum value of the MSE for the second order is:  0.6778501776639635
which is given for sigma =  0.016275479141900393

```

As we can see, all the orders of the weighted regression provide a lower MSE than the OLS and the Adaptive regression under the proper configuration. To recap the results, we found that the OLS led to an MSE of 0.7013 and the Adaptive regression to an MSE of 0.6806. When using the locally weighted regression, it goes down to 0.6776.

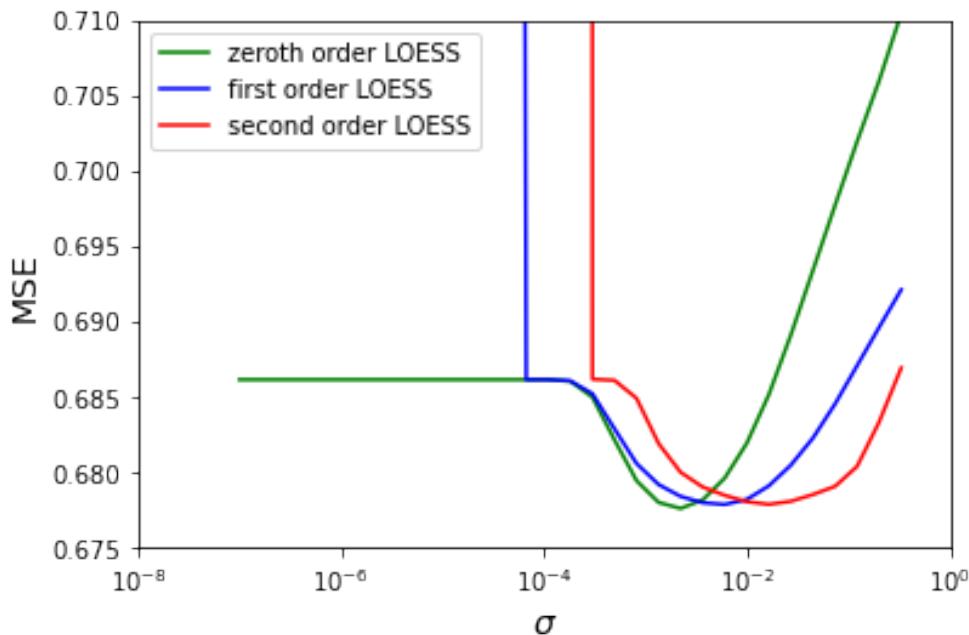


Figure 3.12: LOESS regression of order zero, one and two in terms of the  $\sigma$  parameter for the Ames housing dataset.

A conclusion that can be drawn from these results is that all the regression orders provide a similar performance under the proper configuration of  $\sigma$ . The higher the order, the more unstable it gets when  $\sigma$  is small. Also, as the  $\sigma$  parameter increases, the model unstabilizes more the lower the order of the polynomial regression is.

### 3.3 Bayesian linear regression [Supervised]

#### 3.3.1 Mathematical derivation

[16] [30] To start this chapter, we have to begin introducing Baye's theorem, that is, the probability of an event A happening conditional on an event B is given by:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \& B)}{\mathbb{P}(B)} \quad (3.17)$$

Where  $\mathbb{P}(A \& B)$  is the joint probability of A and B happening and  $\mathbb{P}(B)$  is the probability of the event B. As an example for this theorem, I will use the [45] employment status of the civilian population by sex and age provided by the US Bureau of Labor Statistics (where civilian population refers to non-institutional civilian population).

Age	Civilian Male	Civilian Fem.	Employed Male	Employed Fem.
$\geq 16$ years	129,050	135,795	87,864	77,102
$\geq 20$ years	120,360	127,345	84,694	73,930
16 to 19 years		17,139		6,343

Table 3.1: The U.S. Employment Status-December 2022.

Now, if we compute the probability of working conditional on being a female, i.e., the fraction of females that work:

$$\begin{aligned} \mathbb{P}(\text{Working}|\text{Female}) &= \frac{\mathbb{P}(\text{Working} \& \text{Female})}{\mathbb{P}(\text{Female})} = \\ &= \frac{77102/(129050 + 135795)}{135795/(129050 + 135795)} = \frac{77102}{135795} = 0.57 \end{aligned} \quad (3.18)$$

As we expected, it is equal to the number of working females divided by the number of total females in the pool. Although this example might seem obvious and trivial, this theorem provides a mechanism to compute probabilities that would be hard to obtain at first glance. One can write Baye's theorem in an alternative way by using that  $\mathbb{P}(A \& B) = \mathbb{P}(A)\mathbb{P}(B|A)$ . This is, of course, because the probability of two events happening is given by the product of the respective probabilities of both events. Hence, the probability of A and B happening is the product between the probability of A and the probability of B conditional on event A happening. Using this relation, we can write Baye's theorem like:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A)\mathbb{P}(B|A)}{\mathbb{P}(B)} \quad (3.19)$$

We will focus on the likelihood of the different  $\beta$  coefficients (i.e., our regression coefficients) for the Bayesian regression. Contrary to previous methods, where we found the coefficients that lead to the smallest MSE, here we will assign a likelihood to each possible

value of the coefficient, which will depend on the data provided to the algorithm. We will write the theorem the following way:

$$p(\beta|\mathbf{x}, y) = \frac{p(y|\mathbf{x}, \beta)p(\beta)}{p(\mathbf{x}, y)} \quad (3.20)$$

Let us understand this equation: the probability of  $\beta$  subject to a variable and outcome pair  $(\mathbf{x}, y)$  is given by the probability of having  $\beta$  multiplied by the probability of the  $(\beta, \mathbf{x})$  configuration leading to the outcome  $y$ . The equation is also divided by a term that tells us the probability of a certain  $\mathbf{x}$  leading to the outcome  $y$ . This normalization term helps us ensure the probabilities add up to one. All these quantities have their respective proper names.. .

- $p(\beta)$  is called the **prior**; it describes the initial guess and knowledge of the parameters, which values are likely and which are not.
- $p(\mathbf{x}, y)$  is the **evidence**, it tells us the probability of the variables leading to the different outcomes.
- $p(y|\mathbf{x}, \beta)$  is the **likelihood**, it gives a probability for the outcomes given the coefficients and variables.
- $p(\beta|\mathbf{x}, y)$  is the **posterior**, it describes the probability of the parameters given the data and the outcomes.

This is useful because we can write the probability of the different possible values for the coefficients in terms of the quantities prior, likelihood, and evidence, which can be computed. We will start with an initial guess of the probabilistic distribution of  $\beta$  and then update it with iterations based on Baye's theorem. Nevertheless, another detail needs to be mentioned before jumping into the matter.

Similar to before,  $y_i = \sum_j X_{i,j}\beta_j + \epsilon_i$  where  $\epsilon$  is an error term called noise that compensates for the differences between our predictions and the actual outcome values. For simplicity, I will assume that  $\epsilon$  has a mean of zero and a variance  $\sigma^2$ . There are different ways to estimate this variance, but we will choose the MSE obtained using the OLS regression. The reasoning behind the relation between the variance and MSE will be explained in the *algorithm implementation* section.

Lastly, we need to define the elements involved in Baye's theorem; since they are probabilistic distributions, we will use Gaussian (also known as normal) distributions. A Gaussian distribution is given by:

$$G(X_0, X_1, \dots, X_{D-1} | \bar{\mathbf{x}}, \mathbf{C}) = \frac{\exp[-\frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})\mathbf{C}^{-1}(\mathbf{x} - \bar{\mathbf{x}})^T]}{\sqrt{(2\pi)^D |\mathbf{C}|}} \quad (3.21)$$

Where  $\mathbf{x}$  is the row vector containing the variable values of a data point and  $\bar{\mathbf{x}}$  is the expected or average value of these variables. It must be mention that if  $\mathbf{x}$  was a column

vector the Gaussian would have the factor  $(\mathbf{x} - \bar{\mathbf{x}})^T \mathbb{C}^{-1} (\mathbf{x} - \bar{\mathbf{x}})$  instead.

$$X = \begin{pmatrix} 1 & X_{1,1} & X_{1,2} & \cdots & X_{1,D-1} \\ 1 & X_{2,1} & X_{2,2} & \cdots & X_{2,D-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & X_{N,1} & X_{N,2} & \cdots & X_{N,D-1} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{pmatrix} = (\mathbf{X}_0, \mathbf{X}_1 \cdots \mathbf{X}_{D-1}) \quad (3.22)$$

Lastly,  $\mathbb{C}$  is the covariance matrix which is given by:

$$\mathbb{C} = \begin{pmatrix} \sigma_{\mathbf{X}_0, \mathbf{X}_0}^2 & \sigma_{\mathbf{X}_0, \mathbf{X}_1}^2 & \cdots & \sigma_{\mathbf{X}_0, \mathbf{X}_{D-1}}^2 \\ \sigma_{\mathbf{X}_1, \mathbf{X}_0}^2 & \sigma_{\mathbf{X}_1, \mathbf{X}_1}^2 & \cdots & \sigma_{\mathbf{X}_1, \mathbf{X}_{D-1}}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{\mathbf{X}_{D-1}, \mathbf{X}_0}^2 & \sigma_{\mathbf{X}_{D-1}, \mathbf{X}_1}^2 & \cdots & \sigma_{\mathbf{X}_{D-1}, \mathbf{X}_{D-1}}^2 \end{pmatrix} \quad (3.23)$$

The covariance of two variables,  $\mathbf{X}_j$  and  $\mathbf{X}_k$ , tells us if there is a linear correlation between them.

$$\sigma_{\mathbf{X}_j, \mathbf{X}_k}^2 = \frac{\sum_i^N (X_{i,j} - \bar{X}_j)(X_{i,k} - \bar{X}_k)}{N} \quad (3.24)$$

$\bar{X}_j$  and  $\bar{X}_k$  are the respective average or expected values of the variables. If we make this computation with a sample and not the whole population, the constant  $N$  must be replaced with  $N - 1$ , resulting in a bigger variance to account for the error introduced by sampling the data. Considering that all the data points have the form  $\mathbf{x} = (X_0, X_1 \cdots X_{D-1})$ , the covariance matrix can be written as:

$$\begin{aligned} \mathbb{C} &= E \left[ [\mathbf{x} - E(\mathbf{x})]^T [\mathbf{x} - E(\mathbf{x})] \right] = \\ &E \left[ \begin{pmatrix} X_0 - \bar{X}_0 \\ X_1 - \bar{X}_1 \\ \vdots \\ X_{D-1} - \bar{X}_{D-1} \end{pmatrix} (X_0 - \bar{X}_0, X_1 - \bar{X}_1, \dots, X_{D-1} - \bar{X}_{D-1}) \right] = \\ &E \left[ \begin{pmatrix} (X_0 - \bar{X}_0)(X_0 - \bar{X}_0) & \cdots & (X_0 - \bar{X}_0)(X_{D-1} - \bar{X}_{D-1}) \\ (X_1 - \bar{X}_1)(X_0 - \bar{X}_0) & \cdots & (X_1 - \bar{X}_1)(X_{D-1} - \bar{X}_{D-1}) \\ \vdots & \ddots & \vdots \\ (X_{D-1} - \bar{X}_{D-1})(X_0 - \bar{X}_0) & \cdots & (X_{D-1} - \bar{X}_{D-1})(X_{D-1} - \bar{X}_{D-1}) \end{pmatrix} \right] = \\ &\begin{pmatrix} \sum_i (X_{i,0} - \bar{X}_0)(X_{i,0} - \bar{X}_0)/N & \cdots & \sum_i (X_{i,0} - \bar{X}_0)(X_{i,D-1} - \bar{X}_{D-1})/N \\ \sum_i (X_{i,1} - \bar{X}_1)(X_{i,0} - \bar{X}_0)/N & \cdots & \sum_i (X_{i,1} - \bar{X}_1)(X_{i,D-1} - \bar{X}_{D-1})/N \\ \vdots & \ddots & \vdots \\ \sum_i (X_{i,D-1} - \bar{X}_{D-1})(X_{i,0} - \bar{X}_0)/N & \cdots & \sum_i (X_{i,D-1} - \bar{X}_{D-1})(X_{i,D-1} - \bar{X}_{D-1})/N \end{pmatrix} \quad (3.25) \end{aligned}$$

The operator  $E$  represents the expected value. This leads to the same result as Equation 3.23. All we need to know for the Gaussian distribution of a set of variables is the covariance matrix  $\mathbb{C}$  and a D-dimensional vector  $\bar{\mathbf{x}}$ , which contains the expected values of the involved variables. The notation I used for the Gaussian is the following:  $\mathbb{G}(X_0, X_1, \dots, X_{D-1} | \bar{\mathbf{x}}, \mathbb{C})$ , i.e., the probability of a data point  $\{X_0, X_1 \dots X_{D-1}\}$  conditional on an expected value and covariance matrix pair:  $\{\bar{\mathbf{x}}, \mathbb{C}\}$ .

As an example, the one-dimensional Gaussian, where  $\mathbb{C} = \sigma^2$  is given by the following equation:

$$\mathbb{G}(X | \bar{X}, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[\frac{-(X - \bar{X})^2}{2\sigma^2}\right] \quad (3.26)$$

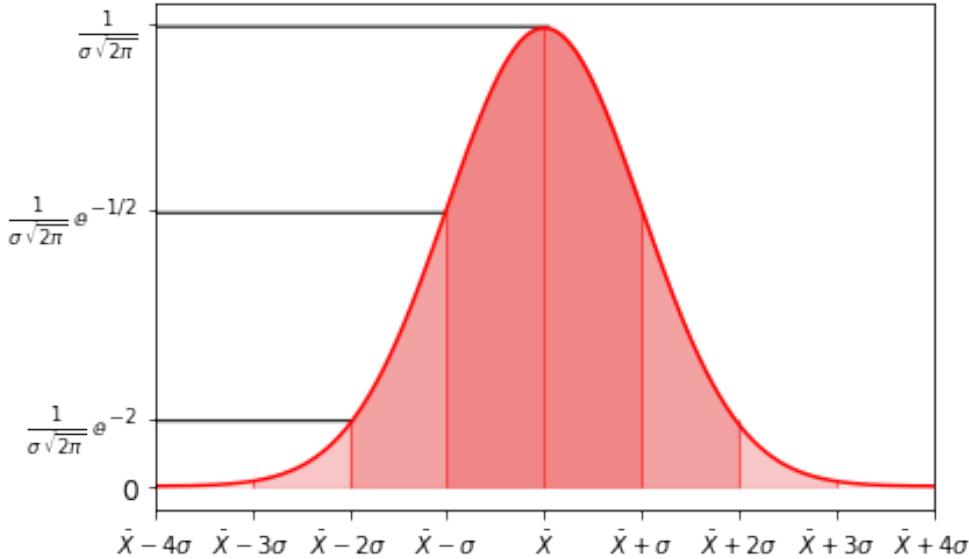


Figure 3.13: Probability density of a one-dimensional Gaussian distribution.

Having introduced all the elements, let us make sense of all this by bringing back Baye's theorem:

$$p(\beta | \mathbf{x}, y) = \frac{p(y | \mathbf{x}, \beta)p(\beta)}{p(\mathbf{x}, y)} \quad (3.27)$$

Moreover, let us remember what our data looks like:

$$X = \begin{pmatrix} 1 & X_{1,1} & X_{1,2} & \cdots & X_{1,D-1} \\ 1 & X_{2,1} & X_{2,2} & \cdots & X_{2,D-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & X_{N,1} & X_{N,2} & \cdots & X_{N,D-1} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{pmatrix} \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{D-1} \end{pmatrix} \quad (3.28)$$

When considering all the data, Baye's model reads:

$$p(\beta | X, Y) = \frac{p(Y | X, \beta)p(\beta)}{p(Y | X)} \quad (3.29)$$

We can write in the denominator  $p(Y|X)$ , which is the probability of the different  $y_i$  conditional to our input variables. Each of the terms involved in Equation 3.29 will be given by a Gaussian; we start with the **prior**:

$$p(\beta) = \mathbb{G}(\beta | \bar{\beta}^{init}, \mathbb{C}_\beta^{init}) \quad (3.30)$$

As we can see, the prior takes a simple form; we have a Gaussian on the  $\beta$  coefficients, which is conditional on an initial guess of the mean or expected value of  $\beta$  and a covariance matrix relating the different terms of  $\beta$ .

$$\mathbb{C}_\beta^{init} = \begin{pmatrix} \sigma_{\beta_0, \beta_0}^2 & \sigma_{\beta_0, \beta_1}^2 & \cdots & \sigma_{\beta_0, \beta_{D-1}}^2 \\ \sigma_{\beta_1, \beta_0}^2 & \sigma_{\beta_1, \beta_1}^2 & \cdots & \sigma_{\beta_1, \beta_{D-1}}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{\beta_{D-1}, \beta_0}^2 & \sigma_{\beta_{D-1}, \beta_1}^2 & \cdots & \sigma_{\beta_{D-1}, \beta_{D-1}}^2 \end{pmatrix} \quad (3.31)$$

Since this will be an iterative model, we can initialize both  $(\bar{\beta}^{init}, \mathbb{C}_\beta^{init})$  to random values of the proper dimensions. Next, we continue with the **likelihood**, which we simply write as:

$$p(Y|X, \beta) = \mathbb{G}(Y|\bar{Y} = X\beta, \mathbb{C}_Y = \sigma^2 \mathbb{I}_{N \times N}) \quad (3.32)$$

As we can see, the probabilistic distribution of the outcomes is also a Gaussian, conditional on both  $X$  and  $\beta$ . Hence, the mean or expected value is given by the prediction of the outcomes, and the covariance matrix of the outcomes is approximated to a diagonal matrix where the elements are given by the variance of the noise. Last, we have the **evidence**, which is the joint probabilities of  $X$  and  $Y$ :

$$p(Y|X) = \int p(Y|X, \beta)p(\beta)d\beta \quad (3.33)$$

Since this last quantity has to encapsulate all the information about the joint probabilities of the data  $X$  and the outcomes  $Y$ , it computes the likelihood (probability of each  $Y$  conditional to  $X$  and beta) integrated over all the possible values of beta multiplied by their prior (or initial estimated probabilistic distribution). As we can see, this term ensures that all the probabilities of the posterior add up to 1; it is a normalization term. As we will see in the next steps, writing the evidence in its Gaussian form will not be necessary.

Once we have all three different terms, we can write the **posterior**, which we need in order to estimate our beta coefficients. We must keep in mind that both  $\beta$  and  $Y$  are column vectors.

$$\begin{aligned} p(\beta|X, Y) &= \frac{p(Y|X, \beta)p(\beta)}{p(Y|X)} = \frac{\mathbb{G}(Y|X\beta, \sigma^2 \mathbb{I}) \mathbb{G}(\beta | \bar{\beta}^{init}, \mathbb{C}_\beta^{init})}{p(Y|X)} = \\ &\frac{1}{p(Y|X)} \frac{\exp[-\frac{1}{2}\sigma^{-2}(Y - X\beta)^T(Y - X\beta)]}{\sqrt{(2\pi)^N \sigma^{2N}}} \frac{\exp[-\frac{1}{2}(\beta - \bar{\beta}^{init})^T(\mathbb{C}_\beta^{init})^{-1}(\beta - \bar{\beta}^{init})]}{\sqrt{(2\pi)^D |\mathbb{C}_\beta^{init}|}} \end{aligned} \quad (3.34)$$

Now, since the **evidence** does not depend on the  $\beta$  coefficients we will take the log of the **posterior** and work with the **prior** and **likelihood** log forms:

$$\begin{aligned}
\log p(\beta|X, Y) &= C_1 - \frac{1}{2}\sigma^{-2}(Y - X\beta)^T(Y - X\beta) - \frac{1}{2}(\beta - \bar{\beta}^{init})^T(\mathbb{C}_\beta^{init})^{-1}(\beta - \bar{\beta}^{init}) = \\
C_1 - \frac{1}{2} &\left[ \sigma^{-2}Y^T Y - 2\sigma^{-2}Y^T X\beta + \sigma^{-2}\beta^T X^2\beta + \beta^T(\mathbb{C}_\beta^{init})^{-1}\beta - 2(\bar{\beta}^{init})^T(\mathbb{C}_\beta^{init})^{-1}\beta + \right. \\
&\left. (\bar{\beta}^{init})^T(\mathbb{C}_\beta^{init})^{-1}\bar{\beta}^{init} \right] = C_2 - \frac{1}{2}\left[ \beta^T \left( \sigma^{-2}X^2 + (\mathbb{C}_\beta^{init})^{-1} \right) \beta - \right. \\
2 &\left( \sigma^{-2}Y^T X + (\bar{\beta}^{init})^T(\mathbb{C}_\beta^{init})^{-1} \right) \beta \Big] = C_2 - \frac{1}{2}\left[ \beta^T \left( \sigma^{-2}X^2 + (\mathbb{C}_\beta^{init})^{-1} \right) \beta - \right. \\
&\left. - 2 \left( \sigma^{-2}X^T Y + (\mathbb{C}_\beta^{init})^{-1} \bar{\beta}^{init} \right)^T \beta \right] \tag{3.35}
\end{aligned}$$

Having arrived at this result, we can draw some conclusions; first of all, the log of the **posterior** has a constant term, a linear term on the  $\beta$  coefficients, and a quadratic term, the fact that the quadratic term is negative hints us that this has a Gaussian form. We want to write this term with the following form:

$$\begin{aligned}
p(\beta|X, Y) &= \mathbb{G}(\beta \mid \bar{\beta}^{final}, \mathbb{C}_\beta^{final}) \propto \exp[-\frac{1}{2}(\beta - \bar{\beta}^{final})^T(\mathbb{C}_\beta^{final})^{-1}(\beta - \bar{\beta}^{final})] = \\
&= \exp\left(-\frac{1}{2}\left[\beta^T(\mathbb{C}_\beta^{final})^{-1}\beta - 2(\bar{\beta}^{final})^T(\mathbb{C}_\beta^{final})^{-1}\beta + C_3\right]\right) \\
&= \exp\left(-\frac{1}{2}\left[\beta^T(\mathbb{C}_\beta^{final})^{-1}\beta - 2\left((\mathbb{C}_\beta^{final})^{-1}\bar{\beta}^{final}\right)^T\beta + C_3\right]\right) \tag{3.36}
\end{aligned}$$

By comparison of both expressions, we can obtain the mean and covariance matrix:

$$\mathbb{C}_\beta^{final} = \left( (\mathbb{C}_\beta^{init})^{-1} + \sigma^{-2}X^T X \right)^{-1} \tag{3.37}$$

$$\bar{\beta}^{final} = \mathbb{C}_\beta^{final} \left( (\mathbb{C}_\beta^{init})^{-1}\bar{\beta}^{init} + \sigma^{-2}X^T Y \right) \tag{3.38}$$

Remember that in this last step and previous steps, I have used the property of symmetry of the covariance matrix (the covariance matrix is equal to its transpose). All those

constants I wrote and also the **evidence**  $p(Y|X)$  contribute to the normalization of the Gaussian, but we avoided the effort of computing them just by computing the linear and quadratic terms and by abusing the normalized nature of the Gaussian. With this, we have all the information we need; we know which is the mean value of the beta coefficients  $\bar{\beta}^{final}$  and we also know the covariance matrix  $\mathbb{C}_\beta^{final}$  which tells us about the dispersion of this result.

Now, all that is left to do is to see how to predict values using the results we have found. Let us suppose we have new input that we want to predict given by:

$$\mathbf{x}^* = (1 \ x_1^* \ x_2^* \ \cdots \ x_{D-1}^*) \quad (3.39)$$

One can define the **posterior predictive** distribution as:

$$p(y^* | X, Y, \mathbf{x}^*) = \int p(y^* | \mathbf{x}^*, \beta) p(\beta | X, Y) d\beta \quad (3.40)$$

This, of course, is the probability of an outcome  $y^*$  conditional on the values of the training data:  $X, Y$  and the test data:  $\mathbf{x}^*$ , which is equal to an integral over the coefficient space involving a product of the posterior distribution and the likelihood distribution for the testing data, i.e., the probability of  $y^*$  being the outcome of  $\mathbf{x}^*$  is given by the probabilities of the  $\beta$  coefficients multiplied by the probabilities of these coefficients leading to the outcome integrated over the coefficient space. There are different ways of computing this integral, although we will take a different approach, which is more straightforward. We start by writing the linear regression equation for our testing data:

$$y^* = \mathbf{x}^* \beta + \epsilon \quad (3.41)$$

As we know, three of these quantities have a Gaussian form given by:

$$\begin{aligned} &\mathbb{G}(y^* | \bar{y}^*, \mathbb{C}_{y^*}) \\ &\mathbb{G}(\beta | \bar{\beta}^f, \mathbb{C}_\beta^f) \\ &\mathbb{G}(\epsilon | 0, \sigma^2) \end{aligned} \quad (3.42)$$

Now, by using the properties of the Gaussian functions under a linear transformation and the linearity properties of the expected value operator, we can write  $\{\bar{y}^*, \mathbb{C}_{y^*}\}$  in terms of the other quantities. I will use  $\bar{X} \equiv E(X)$  for a variable  $X$ . The expected value is straightforward:

$$\bar{y}^* = E(y^*) = \mathbf{x}^* E(\beta) + E(\epsilon) = \mathbf{x}^* \bar{\beta}^f \quad (3.43)$$

The covariance matrix can be easily computed following Equation 3.25. Since  $y^*$  is a one-dimensional vector of unit length, we can write:

$$\begin{aligned}
\mathbb{C}_{y^*} &= E \left[ [\mathbf{x}^* \beta + \epsilon - E(\mathbf{x}^* \beta + \epsilon)] [\mathbf{x}^* \beta + \epsilon - E(\mathbf{x}^* \beta + \epsilon)]^T \right] = \\
&E \left[ [\mathbf{x}^* \beta + \epsilon - \mathbf{x}^* E(\beta)] [\mathbf{x}^* \beta + \epsilon - \mathbf{x}^* E(\beta)]^T \right] = \\
&E \left[ [\mathbf{x}^* \beta - \mathbf{x}^* E(\beta)] [\mathbf{x}^* \beta - \mathbf{x}^* E(\beta)]^T \right] + 2E \left[ [\mathbf{x}^* \beta - \mathbf{x}^* E(\beta)] \epsilon^T \right] + E \left[ \epsilon \epsilon^T \right] = \\
&E \left[ [\mathbf{x}^* [\beta - E(\beta)] [\beta - E(\beta)]^T (\mathbf{x}^*)^T \right] + 2E \left[ \mathbf{x}^* [\beta - E(\beta)] \epsilon^T \right] \quad (3.44) \\
&+ E \left[ [\epsilon - E(\epsilon)] [\epsilon - E(\epsilon)]^T \right] = \mathbf{x}^* E \left[ [\beta - E(\beta)] [\beta - E(\beta)]^T \right] (\mathbf{x}^*)^T + \\
&2\mathbf{x}^* E \left[ [\beta - E(\beta)] \epsilon^T \right] + E \left[ [\epsilon - E(\epsilon)] [\epsilon - E(\epsilon)]^T \right] = \\
&\mathbf{x}^* E \left[ [\beta - E(\beta)] [\beta - E(\beta)]^T \right] (\mathbf{x}^*)^T + E \left[ [\epsilon - E(\epsilon)] [\epsilon - E(\epsilon)]^T \right]
\end{aligned}$$

The second term is set to zero since the following is true when  $X$  and  $Y$  are independent variables:  $E(XY) = E(X)E(Y)$ . And since  $\beta$  and  $\epsilon$  are independent and  $E(\epsilon^T) = 0$  the term vanishes. So the final result reads:

$$\mathbb{C}_{y^*} = \mathbf{x}^* \mathbb{C}_\beta^{final} (\mathbf{x}^*)^T + \mathbb{C}_\epsilon = \mathbf{x}^* \mathbb{C}_\beta^{final} (\mathbf{x}^*)^T + \sigma^2 \quad (3.45)$$

Hence, gathering the results of Equations 3.43 and 3.45, the posterior predictive reads:

$$p(y^* | X, Y, \mathbf{x}^*) = \mathbb{G}(y^* | \mathbf{x}^* \bar{\beta}^{final}, \mathbf{x}^* \mathbb{C}_\beta^{final} (\mathbf{x}^*)^T + \sigma^2) \quad (3.46)$$

So, as one could imagine, the expected value for the outcome of the testing data is given by:  $\mathbf{x}^* \bar{\beta}^{final}$ . The Gaussian has two terms contributing to the covariance. One contribution comes from the noise variance, and the other one from the uncertainty of the beta coefficients.

### 3.3.2 Algorithm implementation

As mentioned in the previous section, the noise variance will be estimated to be the MSE of the OLS regression. The link between both quantities is the following:

$$C_\epsilon = \sigma^2 = E\left[\left[\epsilon - E(\epsilon)\right]^2\right] = E\left[\epsilon^2\right] = E\left[\left(y - \hat{y}\right)^2\right] = \sum_{i=1}^N \frac{(y_i - \hat{y}_i)^2}{N} = MSE \quad (3.47)$$

We will again work with the [2] Ames housing dataset. The accuracy results of this model will not be computed as the motivation behind this section is to use this dataset to show the intricacies of the Bayesian regression. We start by applying the OLS model to the data collection.

```
[1]: reg = OLS(X_ames, Y_ames)
print("[B0,B1] = ", np.round(reg.coef[0], 5))
pred = reg.predict(X_ames)
print("MSE = ", np.round(reg.mse_resid, 5))
print("E(error) = ", np.mean(Y_ames - pred))
```

```
[B0,B1] = [0.          0.53297]
MSE =  0.71594
E(error) =  0.0
```

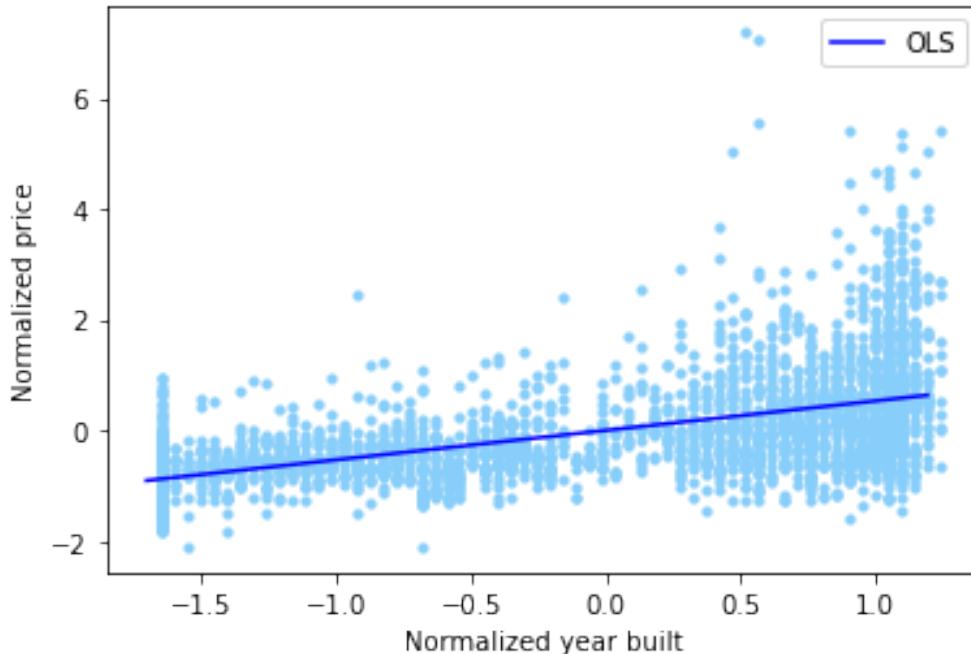


Figure 3.14: Ames housing dataset: price vs. year built.

As we can see, as stated in the mathematical derivation, the error has a mean of zero and hence, a variance that can be estimated to be the MSE. Picking the OLS regression

to compute the MSE is a reasonable estimation as the Bayesian model is also a linear regression.

The next step is to write the Gaussian distribution of the beta coefficients (prior); this function has four input variables. First, a variable pair  $\{\beta_0, \beta_1\}$ , which is a mesh grid of intercept and slope values on a specific interval. And also the mean value and covariance associated with the  $\beta$  coefficients. This function computes the probability density of the beta coefficients, which is given by:

$$p(\beta) = \frac{\exp\left(-\frac{1}{2}[\beta - \bar{\beta}]^T C_{\beta}^{-1} [\beta - \bar{\beta}]\right)}{\sqrt{(2\pi)^D |C_{\beta}|}} \quad (3.48)$$

```
[2]: def B_gaussian(B0,B1,mean,C):
    inv_C = np.linalg.inv(C)
    p = np.zeros(np.shape(B0))

    for i in range(len(p)):
        for j in range(len(p[0])):
            beta = np.array([B0[i][j],B1[i][j]])
            beta = np.reshape(beta,(len(beta),1))

            p[i][j] = np.exp(-np.dot(np.transpose(beta-mean),
                                      np.dot(inv_C,beta-mean))/2.0)
            p[i][j] = p[i][j] / (2*np.pi*np.sqrt(np.linalg.det(C)))

    return(p)

B0 = np.arange(-2.5,4.5,0.01)
B1 = np.arange(-4,1.5,0.01)
B0,B1 = np.meshgrid(B0,B1)

#Beta mean and covariance initialization
B_i = np.array([[1.0],[-1.3]])
C_B_i = np.array([[2.0,0.0],[0.0,1.0]])
p_B = B_gaussian(B0,B1,B_i,C_B_i)
#PLOT
plt.contourf(B0, B1, p_B, levels=15)
plt.scatter(0,0.53297,color="red",label="OLS")
plt.legend()
plt.xlabel("Intercept",fontsize=12)
plt.ylabel("Slope",fontsize=12)
```

At the second half of the code, we generate a mesh grid of  $\beta_0$  and  $\beta_1$ , guess an initial estimate for the mean and covariance of the beta coefficients, and then compute the probability density of  $\beta$ .

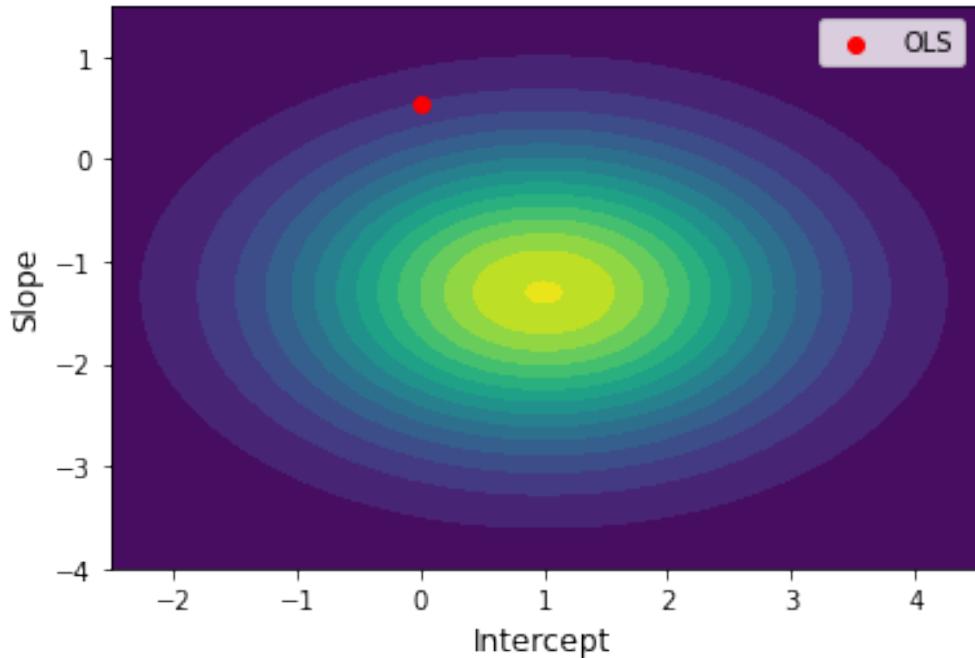


Figure 3.15: Gaussian prior distribution of the beta coefficients with an initial guess of the expected value and covariance matrix.

As shown in the figure, our initial guess of the distribution is quite distant from the coefficients obtained by the OLS model. In the next step, we will update the parameters of the probabilistic distribution using the update rule of Equation 3.37. and Equation 3.38. For this, I built the following function, which picks a percentage of the data and uses it for a given number of iterations to update the mean and covariance of the beta coefficients.

```
[3]: def Bayesian_reg(X,Y,B_i,C_i,var,data_percentage,iterations):

    #Adding a column of ones to the data
    extended_X = np.ones((len(X),2))
    extended_X[:,1:] = X
    #Splitting the data
    X, X_test, Y, Y_test = train_test_split(extended_X, Y,
                                             test_size = 1.0-(data_percentage/100.0), random_state = 0)

    X_T = np.transpose(X)

    #Updating the mean and covariance
```

```

for i in range(iterations):
    inv_C_i = np.linalg.inv(C_i)
    C_i = np.linalg.inv(inv_C_i + np.dot(X_T,X)/var)
    B_i = np.dot(C_i , (np.dot(inv_C_i,B_i) + np.dot(X_T,Y)/var))
return(B_i,C_i)

```

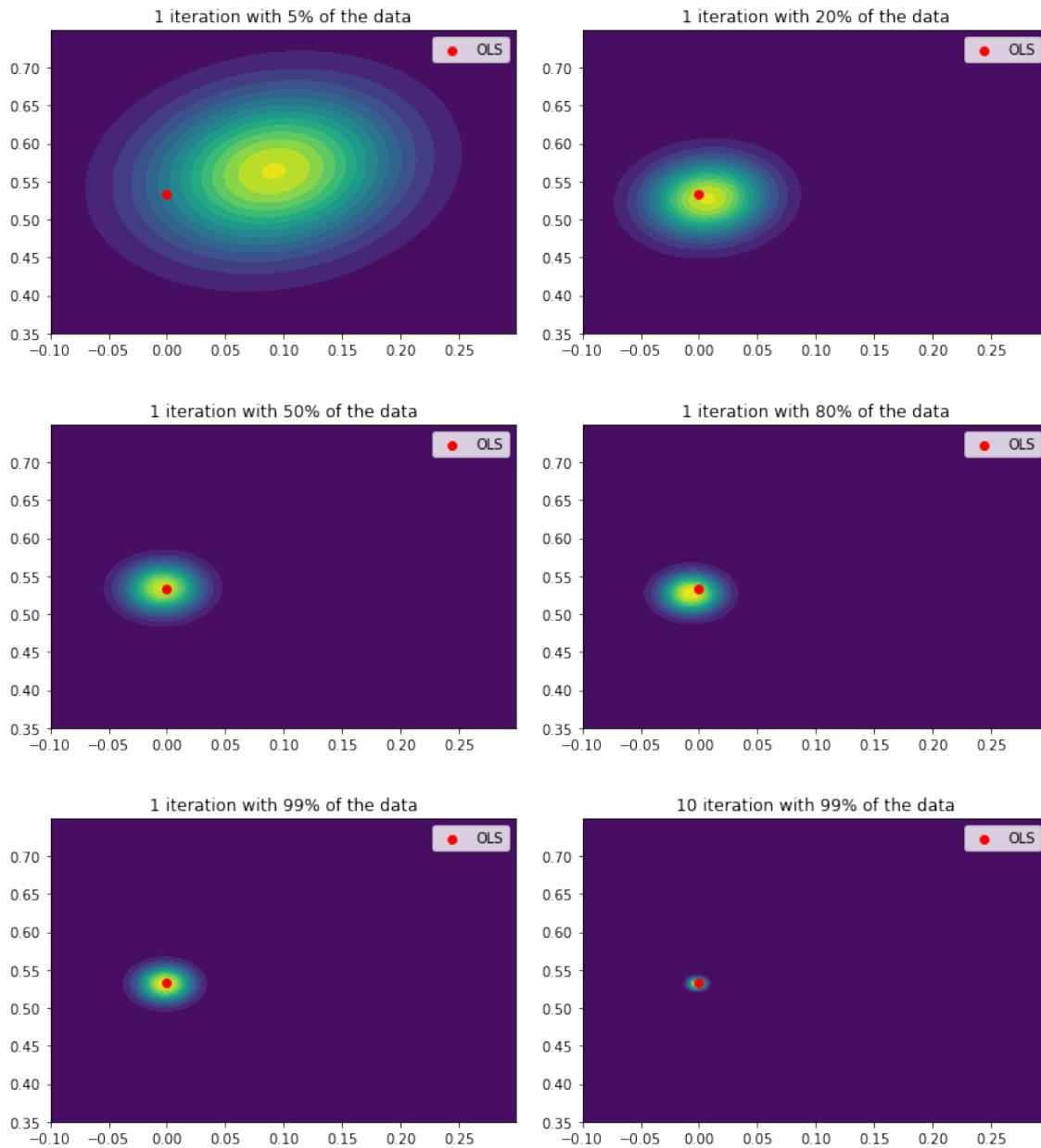


Figure 3.16: Gaussian distribution of the beta coefficients after updating the expected value and covariance matrix with different percentages of the data.

We can see the Gaussian posterior distribution of the  $\beta$  coefficients that results from updating  $\bar{\beta}^f$  and  $\mathbb{C}_\beta^f$  using a percentage of the data for a given number of iterations. As shown, the distribution converges to the value predicted by the OLS as we increase the percentage of the training data or the number of iterations. As we can see, the Bayesian regression is just an alternative way of approaching linear regression. The difference with regular linear regression is that the beta coefficients and the predicted values for the testing data will be distributions instead of numbers so that we can set confidence bounds on our predictions. Another key difference is that we can use our prior knowledge of the coefficients for this method and see how it evolves with the data. The next step is to visualize the posterior predictive, for which we will use the following code.

```
[4]: x_test = np.arange(-1.7,1.4,0.1)
y_test = np.arange(-3,6,0.1)
x_test,y_test = np.meshgrid(x_test,y_test)

B_i = np.array([[1.0],[-1.3]])
C_B_i = np.array([[2.0,0.0],[0.0,1.0]])
B_f,C_B_f = Bayesian_reg(X_ames,Y_ames,B_i,C_B_i,0.71594,99,1)

pred_y = B_f[0] + B_f[1]*x_test
C_y = C_B_f[0][0] + 2*x_test*C_B_f[1][0] + (x_test**2.0)*C_B_f[1][1]
C_y = C_y + 0.71594

def Y_gaussian(y_test,pred_y,C_y):
    p = np.zeros(np.shape(pred_y))

    for i in range(len(p)):
        for j in range(len(p[0])):
            Delta_y = y_test[i][j]-pred_y[i][j]
            p[i][j] = np.exp(-((Delta_y)**2.0)/(2*C_y[i][j]))
            p[i][j] = p[i][j] / np.sqrt(2*np.pi * C_y[i][j])

    return(p)
```

First, we generate a mesh grid of the variable and outcome pair  $\{X, Y\}$ . Next, I initialized the parameters of the prior and updated them to build the Gaussian posterior distribution of the beta coefficients using 99% of the data once. With the following equation, we can obtain the mean and covariance of the predictive posterior.

$$p(y^* | X, Y, \mathbf{x}^*) = \mathbb{G}(y^* | \mathbf{x}^* \bar{\beta}^{final}, \mathbf{x}^* \mathbb{C}_\beta^{final} (\mathbf{x}^*)^T + \sigma^2) \quad (3.49)$$

Since this is a linear regression involving a single variable, we can write  $\bar{y}^* = \bar{\beta}_0^f + x^* \bar{\beta}_1^f$ . On the other hand, the covariance matrix reads:

$$\mathbb{C}_{y^*} = (1 - x^*) \begin{pmatrix} \mathbb{C}_{\beta_0, \beta_0}^f & \mathbb{C}_{\beta_0, \beta_1}^f \\ \mathbb{C}_{\beta_1, \beta_0}^f & \mathbb{C}_{\beta_1, \beta_1}^f \end{pmatrix} \begin{pmatrix} 1 \\ x^* \end{pmatrix} + \sigma^2 = \mathbb{C}_{\beta_0, \beta_0}^f + 2x^* \mathbb{C}_{\beta_1, \beta_0}^f + (x^*)^2 \mathbb{C}_{\beta_1, \beta_1}^f + \sigma^2 \quad (3.50)$$

Lastly, the posterior predictive distribution was plotted, and we can see the probabilistic distribution on the y-axis together with the scatter plot of the data. Keep in mind that Equation 3.49 gives a one-dimensional Gaussian for each value of  $x^*$  (which we will write as  $x^*$  since it is a one-dimensional variable). What we can see on the plot below is exactly that: an overlap of one-dimensional Gaussians on the y-axis for each value of the x-axis.

```
[5]: p = Y_gaussian(y_test,pred_y,C_y)

#PLOT
plt.contourf(x_test, y_test, p, levels=20)
plt.scatter(X_ames,Y_ames,color="lightcoral",s=1)
plt.axis([-1.7,1.3,-3,5.9])
```

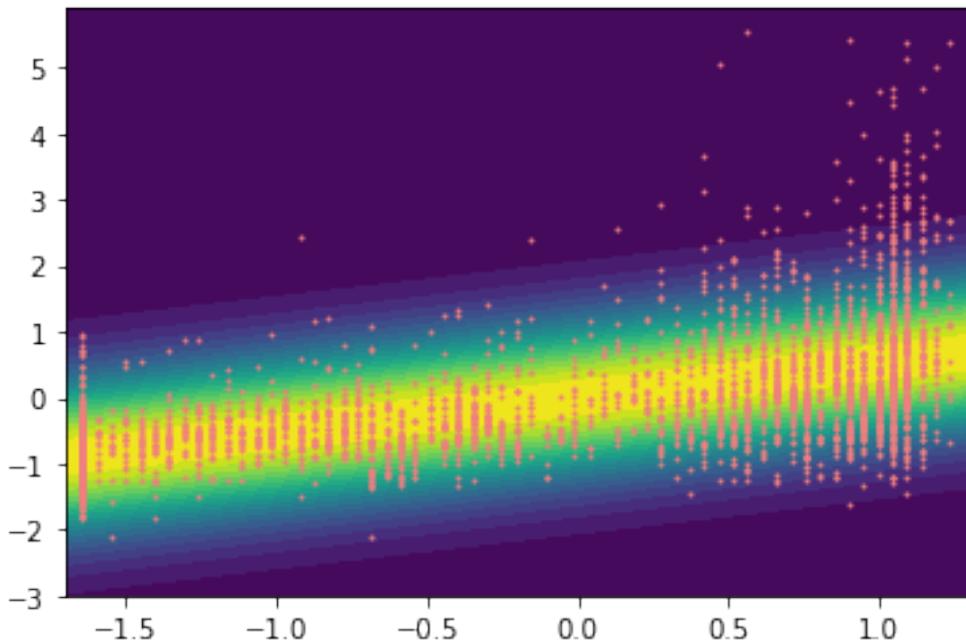


Figure 3.17: Posterior predictive distribution representation.

# Chapter 4

## Association Rule Learning Algorithms

These algorithms find relations between the variables in a data collection. Usually, this type of algorithm is unsupervised. They will be fed a large amount of unlabeled data, and based on a few parameters chosen by us and some logical statistic rules deployed in the model, they will obtain relations between the different items or variables of the data collection. This will make more sense when reading the contents of the **Apriori** algorithm, which is an excellent example.

### 4.1 Apriori Algorithm [Unsupervised]

#### 4.1.1 Mathematical derivation

[25] Let us see the following database containing a list of bookstore transactions, as shown in the table.

Transaction number	Item 1	Item 2	Item 3	Item 4
1	Notebook	pencil	eraser	
2	Notebook	pencil	pen	ruler
3	Notebook	pencil	eraser	
4	pencil	eraser	pen	
5	Notebook	folder	pencil	eraser
6	pencil	eraser	folder	pen
7	pencil	eraser	ruler	
8	Notebook	eraser	pencil	
9	Notebook	eraser	folder	ruler
10	Notebook	pencil	eraser	pen
11	Notebook	pencil	eraser	
12	Notebook	pen		

Table 4.1: A list of transactions from a bookstore.

One needs to define two quantities to apply the apriori algorithm: the support level and the confidence level. For this example, we will set the support level at 33 % and the confidence level at 70 %. I will explain the second one later on; for now, I will explain the support level, which is needed in the first part of the explanation. Given an itemset  $\{I\}$ , the support level of that set will be given by:

$$S(\{I\}) = f(\{I\})/N \quad (4.1)$$

Where  $f(\{I\})$  denotes the frequency of that itemset (the number of times the itemset appears), and  $N$  is the total number of transactions. This means that the support level gives us the fraction of appearance of the itemset on the transactions. The first step will be to compute the frequency of the itemsets and filter them based on the required support level.

We start with those itemsets composed of one item. Since we set the support level at 33 % and we have a total of 12 transactions, we need the frequency to be over or equal to 4.

Itemsets	Frequency
pencil	10
eraser	10
Notebook	9
pen	5
folder	3
ruler	3

Table 4.2: The frequency of itemsets composed of one item.

Now we do the same for the itemsets composed of two items, of course only those itemsets that do not contain any of the items that did not qualify on the previous step, i.e., we need to combine { Pencil, eraser, notebook, pen } on pairs:

Itemsets	Frequency
{ Pencil, eraser }	9
{ Pencil, notebook }	7
{ Pencil, pen }	4
{ eraser, notebook }	7
{ eraser, pen }	3
{ notebook, pen }	3

Table 4.3: The frequency of itemsets composed of two items.

Now, for the itemsets that are composed of three items, taking into account the previously qualified sets:

Itemsets	Frequency
{ Pencil, eraser , notebook }	6
{ Pencil, eraser , pen }	3
{ Pencil, notebook, pen }	2
{ eraser, notebook , pen }	1

Table 4.4: The frequency of itemsets composed of three items.

This first step is over since we cannot form any sets of four items that qualify for the support level requisites. Now is the time to introduce the definition of the confidence level. Suppose we have an itemset  $\{I_1\}$  and another itemset  $\{I_2\}$ , given a rule  $\{I_1\} \rightarrow \{I_2\}$ , the confidence level of this rule will be given by:

$$C(\{I_1\} \rightarrow \{I_2\}) = \frac{f(\{I_1, I_2\})}{f(\{I_1\})} \quad (4.2)$$

That is, the probability of the set  $\{I_2\}$  being acquired if the set  $\{I_1\}$  is acquired will be equal to the number of times the combination of the itemsets is acquired divided by the number of times  $\{I_1\}$  is acquired.

Now, if we call our set  $I = \{Pencil, eraser, notebook\}$  we have to check the confidence level of all the rules  $\{S\} \rightarrow \{I - S\}$  where  $S$  is a non-empty subset of  $I$ .

$$\begin{aligned} S_1 &= \{Pencil\}, & I - S_1 &= \{Eraser, notebook\} & C(S_1 \rightarrow I - S_1) &= 6/10 \\ S_2 &= \{Eraser\}, & I - S_2 &= \{Pencil, notebook\} & C(S_2 \rightarrow I - S_2) &= 6/10 \\ S_3 &= \{Notebook\}, & I - S_3 &= \{Pencil, Eraser\} & C(S_3 \rightarrow I - S_3) &= 6/9 \\ S_4 &= \{Pencil, Eraser\}, & I - S_4 &= \{Notebook\} & C(S_4 \rightarrow I - S_4) &= 6/9 \\ S_5 &= \{Pencil, Notebook\}, & I - S_5 &= \{Eraser\} & C(S_5 \rightarrow I - S_5) &= 6/7 \\ S_6 &= \{Eraser, Notebook\}, & I - S_6 &= \{Pencil\} & C(S_6 \rightarrow I - S_6) &= 6/7 \end{aligned}$$

There are only two rules that fulfill the confidence requisites:

$$\text{Rule 1 : } \{Pencil, Notebook\} \rightarrow \{Eraser\} \text{ with a confidence of 86\%} \quad (4.3)$$

$$\text{Rule 2 : } \{Eraser, Notebook\} \rightarrow \{Pencil\} \text{ with a confidence of 86\%} \quad (4.4)$$

As I mentioned before, this parameter gives the likelihood of the second itemset being purchased if the first itemset has been purchased. Businesses often use this algorithm to incentivize customers to buy more by setting association rules between the items and applying discounts smartly. After this small exercise, we know, for example, that the itemset  $\{Pencil, eraser, notebook\}$  is the most purchased one. We also know that whenever a customer picks either  $\{Pencil, Notebook\}$  or  $\{Eraser, Notebook\}$ , they are more likely to pick the remaining item of the set. If they pick  $\{Pencil, eraser\}$ , they are less likely to pick the Notebook. We can use this information to apply discounts and place the different items smartly or to see how these decisions affect purchases in the future.

## 4.1.2 Algorithm implementation

Now is the time to implement this algorithm. First, we will start by loading the data set we worked with in the previous section.

```
[1]: import pandas as pd  
import numpy as np  
  
data = pd.read_csv( "transactions.csv" )  
print(data)
```

	item1	item2	item3	item4
0	Notebook	pencil	eraser	NaN
1	Notebook	pencil	pen	ruler
2	Notebook	pencil	eraser	NaN
3	pencil	eraser	pen	NaN
4	Notebook	folder	pencil	eraser
5	pencil	eraser	folder	pen
6	pencil	eraser	ruler	NaN
7	Notebook	eraser	pencil	NaN
8	Notebook	eraser	folder	ruler
9	Notebook	pencil	eraser	pen
10	Notebook	pencil	eraser	NaN
11	Notebook	pen	NaN	NaN

It is vital to remove the NaN values so that we do not encounter any errors. For that, I will replace them with empty string values.

```
[2]: data = data.fillna("")  
values = data.values  
print(values)
```

[[['Notebook' 'pencil' 'eraser' '']]
[['Notebook' 'pencil' 'pen' 'ruler']]
[['Notebook' 'pencil' 'eraser' '']]
[['pencil' 'eraser' 'pen' '']]
[['Notebook' 'folder' 'pencil' 'eraser']]
[['pencil' 'eraser' 'folder' 'pen']]
[['pencil' 'eraser' 'ruler' '']]
[['Notebook' 'eraser' 'pencil' '']]
[['Notebook' 'eraser' 'folder' 'ruler']]
[['Notebook' 'pencil' 'eraser' 'pen']]
[['Notebook' 'pencil' 'eraser' '']]
[['Notebook' 'pen' ' ' '']]

```
[3]: import itertools
import copy as copy

class apriori():

    def __init__(self, data, support, confidence):

        self.data = data
        self.support = support
        self.confidence = confidence
```

The *element-list* function uses the *set* function to pick all the unique elements from the data. It must be turned into a list if we want it to be an array.

```
def element_list(self,data):

    data = self.data
    elements = list(set(element for row in data\
                         for element in row if element != ""))
    return(elements)
```

Next, we have the *frequency* function. Given that we have our data storing the transactions on the form  $data = [t_1, t_2 \dots t_N]$  and an itemset given by  $I = [i_1, i_2 \dots i_M]$ , this function first builds the following matrix:

$$Count = \begin{pmatrix} Count(i_1, t_1) & Count(i_2, t_1) & \cdots & Count(i_M, t_1) \\ Count(i_1, t_2) & Count(i_2, t_2) & \cdots & Count(i_M, t_2) \\ \vdots & \vdots & \ddots & \vdots \\ Count(i_1, t_N) & Count(i_2, t_N) & \cdots & Count(i_M, t_N) \end{pmatrix} \quad (4.5)$$

Where  $Count(i_M, t_N)$  gives us the times the item  $i_M$  appears in the transaction  $t_N$ . Then the following line counts the number of times there is a zero on each of the rows of the matrix, i.e., we obtain a 1D array that saves the value zero in the  $j^{th}$  position if all the items of the itemset appear on the  $j^{th}$  transaction and a number different than zero if one or more items are missing. Then, if we count the number of zeros on the 1D array, we obtain the number of transactions where the whole set appears. This process is done for all the itemsets fed to the *frequency* function as input.

```

def frequency(self, item_sets):

    data = self.data
    frequencies = np.zeros((len(item_sets)))

    for N in range(len(item_sets)):

        itemset = item_sets[N]

        count = np.array([[np.count_nonzero(data[i] == itemset[j]) \
                          for j in range(len(itemset))] for i in range(len(data))])
        count = np.array([np.count_nonzero(count[i,:] == 0) \
                         for i in range(len(count))])

        frequencies[N] = np.count_nonzero(count == 0)

    return(frequencies)

```

This following function is straightforward; it saves the sets that fulfill the support requirement and their frequency values on two arrays.

```

def set_filter(self, item_sets):

    frequencies = self.frequency(item_sets)

    qualified_sets = []
    qualified_frec = []

    for i in range(len(item_sets)):

        if (frequencies[i]/len(self.data) > (self.support/100)):

            qualified_sets.append(item_sets[i])
            qualified_frec.append(frequencies[i])

    return(np.array(qualified_sets),np.array(qualified_frec))

```

The *initial phase* function executes all the steps needed on the model. It starts iterating on the itemset size  $N$  ranging from one to the size of the *elements* array. For each iteration, it uses the *itertools.combinations(array, N)* function, which stores all the itemsets of size  $N$  obtainable from the array. Then, we save all the sets that qualify in terms of the support level. If the number of the qualified sets is zero, then we return the sets of the previous

iteration, i.e., the sets that are one unit smaller. If not, we save the qualified sets of the current iteration on the variable *previous qualified sets* for the next iteration. Then, we print the results by calling the *printing function*. We also update the *elements* array by picking them from the size N itemsets. This is done in order to remove those items discarded in previous iterations.

```
def initial_phase(self):

    elements = self.element_list(self.data)

    for N in range(1,len(elements)+1):

        N_item_sets = list(itertools.combinations(elements, N))

        qualified_sets, qualified_frec = self.set_filter(N_item_sets)

        if (len(qualified_sets)==0):

            return(prev_qualified_sets,prev_qualified_frec)

        prev_qualified_sets = copy.deepcopy(qualified_sets)
        prev_qualified_frec = copy.deepcopy(qualified_frec)

        self.printing(N,qualified_sets,qualified_frec)
        elements = self.element_list(N_item_sets)

    return(qualified_sets,qualified_frec)
```

Last, we have the *rules* function. It picks the different sets stored in the *qualified sets* array. Then, for each qualified set, it generates all the possible subsets of different sizes. For each subset, we compute the frequency and then the confidence level. We also compute the adjunct subset, which is composed of all those elements that are not part of the subset but appear on the set. The results are printed, presenting the rules that surpass the confidence level.

```
def rules(self,qualified_sets,qualified_frec):

    for I_set in qualified_sets:

        set_frec = self.frequency([I_set])[0]

        for i in range(len(I_set)-1):
```

```

        subsets=list(itertools.combinations(I_set, i+1))

        for subset in subsets:

            subset_frec = self.frequency([subset])[0]

            confidence_lvl = set_frec/subset_frec

            if (confidence_lvl > self.confidence/100):

                adjunct_subset = [a for a in I_set \
                                  if a not in subset]

                print("Rule: ",subset,"--->",adjunct_subset,
                      "    conf = ",np.round(confidence_lvl*100,2),"%")

```

Among the remaining functions, we have the *printing* function, which prints the support levels obtained for the different itemset sizes and the rules, and the *main\_part* function, which calls the main functions of the code.

```

def printing(self,N,qualified_sets,qualified_frec):
    print("-----")
    print(" |           Itemsets of size ",N," | ")
    print("-----")
    for i in range(len(qualified_sets)):
        print(" | ",qualified_sets[i]," | ",qualified_frec[i],
              " | supp = ",np.round(qualified_frec[i]*100/len(data),1)," %")
    print("-----")

def main_part(self):

    qualified_sets,qualified_frec = self.initial_phase()
    self.rules(qualified_sets,qualified_frec)

```

I executed the code requesting a support level of 33% and a confidence level of 70%. The results are precisely those we obtained in the previous section.

```
[4]: reg = apriori(values,33,70)
result = reg.main_part()
```

```
|-----|  
|           Itemsets of size  1 |  
|-----|  
| ['Notebook'] |  9.0  | supp =  75.0  %  
|-----|  
| ['eraser']   | 10.0  | supp =  83.3  %  
|-----|  
| ['pencil']   | 10.0  | supp =  83.3  %  
|-----|  
| ['pen']      |  5.0  | supp =  41.7  %  
|-----|  
|-----|  
|           Itemsets of size  2 |  
|-----|  
| ['Notebook' 'eraser'] |  7.0  | supp =  58.3  %  
|-----|  
| ['Notebook' 'pencil']  |  7.0  | supp =  58.3  %  
|-----|  
| ['eraser' 'pencil']   |  9.0  | supp =  75.0  %  
|-----|  
| ['pencil' 'pen']     |  4.0  | supp =  33.3  %  
|-----|  
|-----|  
|           Itemsets of size  3 |  
|-----|  
| ['Notebook' 'eraser' 'pencil'] |  6.0  | supp =  50.0  %  
|-----|  
Rule:  ('Notebook', 'eraser') ----> ['pencil']    conf =  85.71 %  
Rule:  ('Notebook', 'pencil')  ----> ['eraser']    conf =  85.71 %
```

# Chapter 5

## Classification Algorithms

This next chapter will contain classification algorithms, i.e., algorithms that work with data where the possible outcomes or labels are given by a finite collection of integer numbers, each representing an idea or a group. For example, we could have a data collection about tumors with a binary outcome (0 or 1) associated with the concepts *benign* or *malignant*. In particular, we will work with a ternary database with the outcomes  $\{0, 1, 2\}$ . We are going to test five different algorithms here:

- The **Logistic regression** is a regression algorithm, as it adjusts a curve to the data. However, it is used on data with a binary outcome  $\{0, 1\}$ . The idea is to find the Sigmoid curve that adjusts to our data best, as this function is contained on the interval  $[0, 1]$ . Hence, we must work with the outcomes on pairs and shift labels.
- The **KNN** model makes predictions based on the proximity of the K nearest neighbors. We consider a testing point and assign it the most common outcome among the K nearest neighbors. KNN can be used for databases with more than two outcomes, in this case,  $\{0, 1, 2\}$ .
- Next, we have the **LVQ** model, which generates random neurons, each representing an outcome (i.e., labeled neurons). Then, it updates the neuron's position by interacting with the training data. Lastly, the testing points are predicted based on the proximity to the neurons. LVQ can be used for databases with more than two outcomes, in this case,  $\{0, 1, 2\}$ .
- The **Support Vector Machines** is the most complex algorithm out of all these classification models. SVM aims to find a hyperplane to divide the data classes as accurately as possible. Support vector machines are limited to databases with two outcomes with the labels -1 and 1. Hence, we will have to work with label pairs and shift them from  $\{0, 1, 2\}$  to -1 and 1.
- Last, we have the **K-means Clustering** algorithm, which is the only unsupervised model of the lot. The idea is to generate random neurons similar to the LVQ model. Nevertheless, instead of using the knowledge of the training data's outcome, we will assign the training data an outcome based on the proximity to the neurons and

then update the neurons to the centroids of the clusters that share an outcome with them. The number of clusters and, hence, the number of classes in which we divide the data will be chosen by us.

In this chapter, we will work with the [44] Iris dataset, which stores data about three different Iris flowers: *Iris Setosa*, *Iris Versicolor*, and *Iris Virginica*. It stores variables such as sepal length, sepal width, petal length, and petal width. We will limit it to the first two variables since two of the classes are overlapped, and it is an excellent database to see how the algorithms work. To load the dataset, we use the following piece of code.

```
[1]: import pandas as pd
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data[:, 0:2]
X = (X - np.mean(X, axis=0))/np.std(X, axis=0)
Y = iris.target
```

As we can see, against what I indicated in the first chapter, I will normalize the data before splitting it into the training and testing sets. The reason for this is that we just want to see that the algorithms work correctly; we are not trying to assess the exact performance of the models. Keep in mind that by doing so, we are leaking some information from the testing data into the training data, which means that the performance of our models will be slightly better than it would otherwise (for the KNN algorithm, for example, the accuracy drops from 79.12 % to 79.10 %). We will use 5-fold cross-validation, averaged over the first 200 random states, to test these classification models.

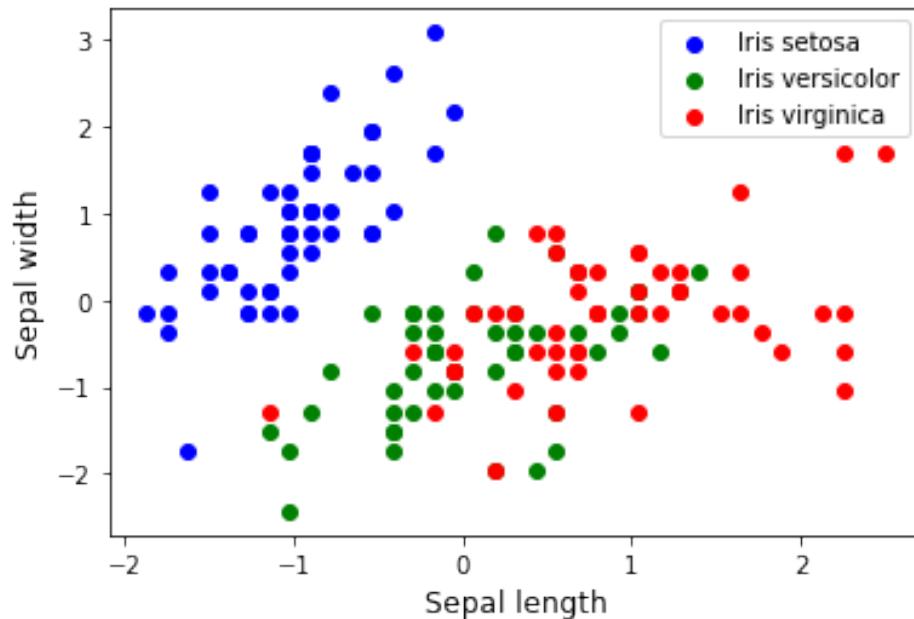


Figure 5.1: The plot of the first two variables of the Iris dataset.

## 5.1 Logistic regression [Supervised]

### 5.1.1 Mathematical derivation

[1] As mentioned before, logistic regression is a classification algorithm in the sense that instead of predicting an outcome that belongs to the real numbers, this model will give us the probability of a point being associated with each of the elements of the label pair {0,1}.

Let's consider the following example where we have a classification on whether someone uses public transport to get to work or walks to work based on the minutes it would take to get there by walking. We assign an outcome of one to those who take public transport and zero to those who walk to work; the idea behind this is to make a regression, and depending on whether the probability is above or below the bisecting point 0.5, we classify the point on one group or the other.

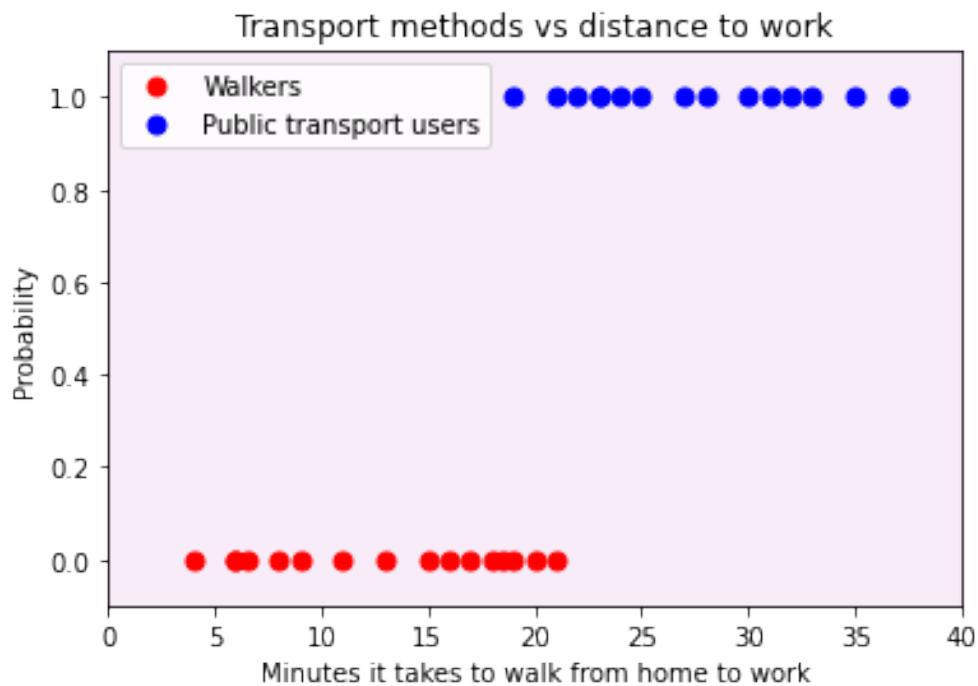


Figure 5.2: Example of a data collection with a binary outcome.

As we would expect, whenever the time it takes to go to work increases, people start using public transport, and when the time is low, they consider walking to work. One could think that a linear regression is good enough to approach this case. Let us see what happens when we fit a linear regression to this dataset.

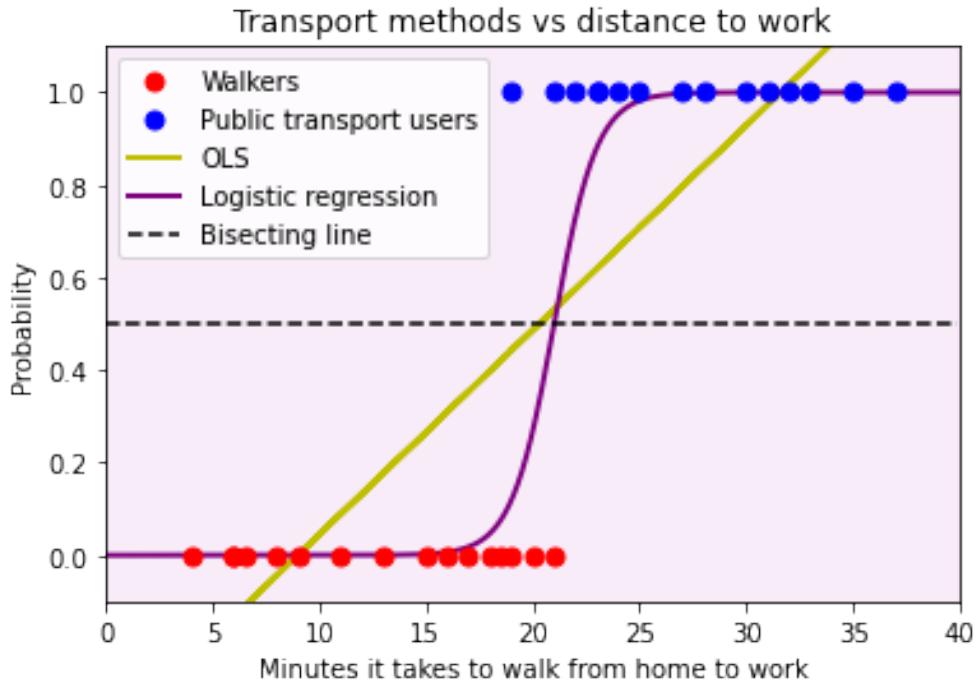


Figure 5.3: OLS and Logistic regression on the example of Figure 5.2.

As shown in Figure 5.3, the problem with this approach is that the values of the OLS line go from negative infinite to infinite and are not inside the interval  $[0,1]$ . To fix this, we will use a different function to make a regression out of our data, plotted as a purple curve in the previous graph.

First, we start by introducing the concept of odds. Let's imagine we have the probability of an event occurring denoted by the variable  $\mathbb{P}$ . The odds of this event are given by the quotient between the probability of the event happening and the event not happening, i.e.:

$$odds = \frac{\mathbb{P}}{1 - \mathbb{P}} \quad (5.1)$$

If we introduce the Logit function, we can define:

$$L = \ln\left(\frac{\mathbb{P}}{1 - \mathbb{P}}\right) \quad (5.2)$$

Now, by taking the exponential of both sides:

$$e^L = \left(\frac{\mathbb{P}}{1 - \mathbb{P}}\right) \quad (5.3)$$

Hence, we obtain that the probability is given by:

$$\mathbb{P} = \frac{1}{1 + e^{-L}} \quad (5.4)$$

This function is called the Sigmoid curve, and its main advantage over the OLS is that it is contained in the interval  $[0, 1]$ . We can see an example of a Sigmoid in Figure 5.3. All we need to do now is to determine how to compute  $L$  to make a regression on our data points. For this, we will write the probabilities as the following:

$$\mathbb{P}(x_i) = \frac{1}{1 + e^{-\sum_j \beta_j X_{i,j}}} \quad (5.5)$$

As we have seen in previous algorithms, the quantities involved in Equation 5.5 are given by:

$$X_{i,j} = \begin{pmatrix} 1 & X_{1,1} & X_{1,2} & \cdots & X_{1,D-1} \\ 1 & X_{2,1} & X_{2,2} & \cdots & X_{2,D-1} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & X_{N,1} & X_{N,2} & \cdots & X_{N,D-1} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{D-1} \end{pmatrix} \quad (5.6)$$

The next step is to find these beta coefficients to obtain a Sigmoid curve that adjusts well to our training data points. The approach is similar to that of the OLS model. We will minimize the difference between the predictions of the regression and the actual values. This is equivalent to maximizing the following quantity: the **likelihood function**.

$$S(\beta) = \prod_{i=1}^N S_i(\beta) = \prod_{i=1}^N \mathbb{P}(x_i)^{y_i} [1 - \mathbb{P}(x_i)]^{1-y_i} \quad (5.7)$$

As we can see, each  $S_i(\beta)$  is greater than zero and smaller than one, since  $\mathbb{P}(x_i) \in [0, 1]$ . Its maximum value will be achieved whenever either  $y_i = \mathbb{P}(x_i) = 1$  or  $y_i = \mathbb{P}(x_i) = 0$ , which is what we wanted. Since  $S(\beta) \in [0, 1]$  maximizing this function is equal to maximizing its logarithm, which is called **log-likelihood** and is contained on the interval  $[-\infty, 0]$ .

$$\ln S(\beta) = \sum_{i=1}^N \left[ y_i \ln \mathbb{P}(x_i) + (1 - y_i) \ln [1 - \mathbb{P}(x_i)] \right] \quad (5.8)$$

The next step is to work with the logarithms to simplify the expression further:

$$\begin{aligned} \ln S(\beta) &= \sum_{i=1}^N \left[ y_i \ln \left( \frac{1}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) + (1 - y_i) \ln \left( 1 - \left( \frac{1}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) \right) \right] = \\ &= \sum_{i=1}^N \left[ y_i \ln \left( \frac{1}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) + (1 - y_i) \ln \left( \frac{e^{-\sum_j \beta_j X_{i,j}}}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) \right] = \\ &= \sum_{i=1}^N \left[ y_i \ln \left( e^{\sum_j \beta_j X_{i,j}} \right) + \ln \left( \frac{e^{-\sum_j \beta_j X_{i,j}}}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) \right] = \\ &= \sum_{i=1}^N \left[ y_i \ln \left( e^{\sum_j \beta_j X_{i,j}} \right) + \ln \left( \frac{1}{1 + e^{\sum_j \beta_j X_{i,j}}} \right) \right] = \sum_{i=1}^N \left[ y_i \sum_j \beta_j X_{i,j} + \ln \left( \frac{1}{1 + e^{\sum_j \beta_j X_{i,j}}} \right) \right] \end{aligned} \quad (5.9)$$

From which we obtain the final expression for the **log-likelihood**:

$$S^*(\beta) = \ln S(\beta) = \sum_{i=1}^N \left[ y_i \sum_j \beta_j X_{i,j} - \ln \left( 1 + e^{\sum_j \beta_j X_{i,j}} \right) \right] \quad (5.10)$$


---

## NEWTON'S METHOD

The next step is to find the  $\beta$  coefficients that maximize the log-likelihood  $S^*(\beta)$ . Although there are different approaches to this, I will introduce Newton's method, which consists of an iterative method to obtain the coefficients.

Suppose we have a function  $f(x)$ , we can expand this function in a Taylor series, which reads as the following:

$$\begin{aligned} f_T(x) &= f_T(x_0 + \Delta x) = f(x_0) + \frac{\partial f(x)}{\partial x} \Big|_{x=x_0} \Delta x + \frac{1}{2} \frac{\partial^2 f(x)}{\partial x^2} \Big|_{x=x_0} \Delta x^2 + \mathcal{O}(\Delta x^3) = \\ &= f(x_0) + \frac{\partial f(x)}{\partial x} \Big|_{x=x_0} (x - x_0) + \frac{1}{2} \frac{\partial^2 f(x)}{\partial x^2} \Big|_{x=x_0} (x - x_0)^2 + \mathcal{O}(x^3) \end{aligned} \quad (5.11)$$

To obtain for which value of the variable the function is maximum or minimum, we proceed to compute the derivative of the Taylor series and make it equal to zero:

$$\frac{\partial}{\partial x} f_T(x) = \frac{\partial f(x)}{\partial x} \Big|_{x=x_0} + \frac{\partial^2 f(x)}{\partial x^2} \Big|_{x=x_0} (x - x_0) + \mathcal{O}(x^2) = 0 \quad (5.12)$$

From this, we find the following result:

$$x_{min/max} = x_0 - \frac{\frac{\partial f(x)}{\partial x} \Big|_{x=x_0}}{\frac{\partial^2 f(x)}{\partial x^2} \Big|_{x=x_0}} \quad (5.13)$$

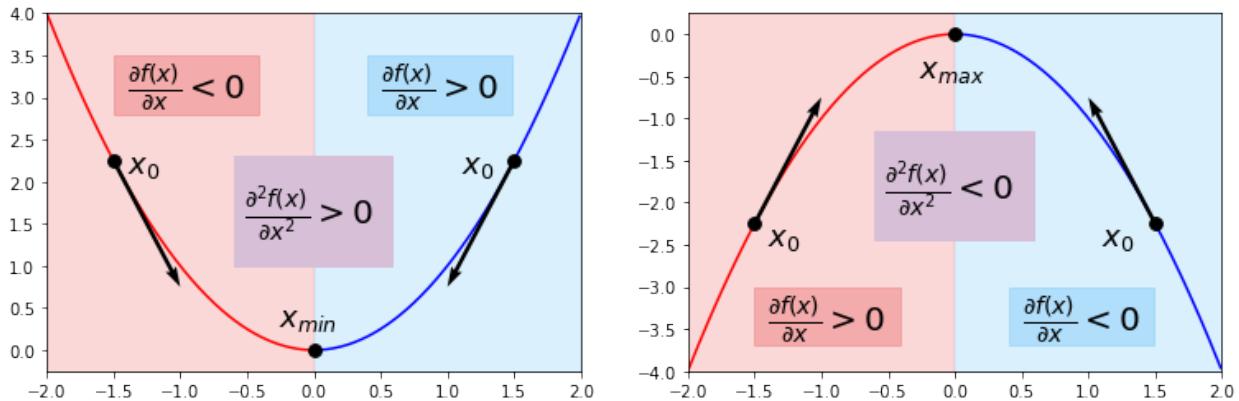


Figure 5.4: Visualization of Newton's iterative method.

As shown in Figure 5.4, this expression works for minimum and maximum points; it always converges to the extreme points of the function. Of course, it is trivial to generalize this to a function that depends on more variables. Let's suppose we have a function of  $\mathbf{x} = (x_1, x_2, \dots, x_m)$ , where  $\mathbf{x}$  is given by a column vector. We can write the Taylor expansion like this:

$$f_T(\mathbf{x}) = f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^T \nabla f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0} + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbb{H}_f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0} (\mathbf{x} - \mathbf{x}_0) + \mathcal{O}(\mathbf{x}^3) \quad (5.14)$$

Where these quantities:  $\nabla f(\mathbf{x})$  and  $\mathbb{H}_f(\mathbf{x})$  are called the gradient and Hessian matrix, respectively, and they have the following form:

$$\mathbb{H}_f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_m} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_m} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_m \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_m \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_m \partial x_3} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_m \partial x_m} \end{pmatrix} \quad \nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{x_1} \\ \frac{\partial f(\mathbf{x})}{x_2} \\ \frac{\partial f(\mathbf{x})}{x_3} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{x_m} \end{pmatrix} \quad (5.15)$$

By doing the same process as before (computing the gradient of the Taylor expansion and making it equal to zero), we obtain the following:

$$\nabla f_T(\mathbf{x}) = \nabla f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0} + \mathbb{H}_f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0} (\mathbf{x} - \mathbf{x}_0) + \mathcal{O}(\mathbf{x}^2) = 0 \quad (5.16)$$

From this, we find that:

$$\mathbf{x} = \mathbf{x}_0 - (\mathbb{H}_f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0})^{-1} \nabla f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0} \quad (5.17)$$


---

Now, we have an expression that we can use to iterate and find the  $\beta$  coefficients that lead to the maximum value of  $S^*(\beta)$ . All we need to do is to compute the first and second partial derivatives of the function  $S^*(\beta)$  so we can construct our gradient and Hessian matrix:

$$\begin{aligned} \frac{\partial S^*(\beta)}{\partial \beta_a} &= \frac{\partial}{\partial \beta_a} \sum_{i=1}^N \left[ y_i \sum_j \beta_j X_{i,j} - \ln \left( 1 + e^{\sum_j \beta_j X_{i,j}} \right) \right] = \sum_{i=1}^N \left[ y_i X_{i,a} - \frac{X_{i,a} e^{\sum_j \beta_j X_{i,j}}}{1 + e^{\sum_j \beta_j X_{i,j}}} \right] = \\ &= \sum_{i=1}^N \left[ y_i X_{i,a} - \frac{X_{i,a}}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right] = \sum_{i=1}^N \left[ \left( y_i - \mathbb{P}(x_i) \right) X_{i,a} \right] \quad (5.18) \end{aligned}$$

And now the second derivative:

$$\begin{aligned} \frac{\partial^2 S^*(\beta)}{\partial \beta_a \partial \beta_b} &= \sum_{i=1}^N \left[ \frac{\partial}{\partial \beta_b} \left( \frac{-X_{i,a}}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) \right] = \sum_{i=1}^N \left[ -e^{-\sum_j \beta_j X_{i,j}} X_{i,a} X_{i,b} \left( \frac{1}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right)^2 \right] = \\ &= \sum_{i=1}^N \left[ -X_{i,a} X_{i,b} \left( \frac{1}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) \left( \frac{e^{-\sum_j \beta_j X_{i,j}}}{1 + e^{-\sum_j \beta_j X_{i,j}}} \right) \right] = \sum_{i=1}^N \left[ -X_{i,a} X_{i,b} \mathbb{P}(x_i) (1 - \mathbb{P}(x_i)) \right] \quad (5.19) \end{aligned}$$

With this, we have all the necessary tools to write the algorithm and implement the logistic regression model. We can write the iterations as the following:

$$\boldsymbol{\beta} = \boldsymbol{\beta}_0 - \left( \mathbb{H}_{S^*}(\boldsymbol{\beta}_0) \right)^{-1} \nabla S^*(\boldsymbol{\beta}_0) \quad (5.20)$$

These quantities can be written in matrix form as:

$$\nabla S^*(\boldsymbol{\beta}) = X^T(y - P) \quad (5.21)$$

$$\mathbb{H}_{S^*}(\boldsymbol{\beta}) = -X^T Q X \quad (5.22)$$

The terms involved in equations 5.21 and 5.22 are defined as the following:

$$X = \begin{pmatrix} 1 & X_{1,1} & X_{1,2} & \cdots & X_{1,D-1} \\ 1 & X_{2,1} & X_{2,2} & \cdots & X_{2,D-1} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & X_{N,1} & X_{N,2} & \cdots & X_{N,D-1} \end{pmatrix}; \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{D-1} \end{pmatrix}; \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}; \quad P = \begin{pmatrix} \mathbb{P}(x_1) \\ \mathbb{P}(x_2) \\ \vdots \\ \mathbb{P}(x_N) \end{pmatrix} \quad (5.23)$$

$$Q = \begin{pmatrix} \mathbb{P}(x_1)[1 - \mathbb{P}(x_1)] & 0 & 0 & \cdots & 0 \\ 0 & \mathbb{P}(x_2)[1 - \mathbb{P}(x_2)] & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \mathbb{P}(x_N)[1 - \mathbb{P}(x_N)] \end{pmatrix} \quad (5.24)$$

## 5.1.2 Algorithm implementation

We proceed with the implementation of the logistic regression. Since this is a classification algorithm, we will work with accuracy instead of MSE, storing how many of the labels are correctly predicted:

```
[1]: from numpy.linalg import inv
import numpy as np

class logistic():

    def __init__(self,X,Y,iterations):

        self.X = X
        self.Y = np.reshape(Y,(len(Y),1))
        self.iterations = iterations
        self.beta = np.zeros((len(X[0])+1,1))
        self.lr = 0.1
```

It is essential to mention here that the code takes a matrix of two dimensions as input for  $X$ ; otherwise, it will raise an error when defining  $self.\beta$ , so it is vital to reshape it before executing the code if necessary.

```
def extend(self,X):

    M = len(X)
    N = len(X[0])
    extended_X = np.ones((M,N+1))
    extended_X[:,1:] = X
    return(extended_X)
```

As we will see later, sometimes we may encounter data with more than two labels, i.e., more than two possible outcomes. Although we will only work with them on pairs, we may need to change the outcome values to either zero or one to fit a logistic regression.

```
def adjust_y(self):

    zero_count = np.count_nonzero(self.Y==0)
    one_count = np.count_nonzero(self.Y==1)

    for i in range(len(self.Y)):

        if (zero_count == 0 and self.Y[i]!=1):
            self.Y[i] = 0
```

```

    elif (one_count == 0 and self.Y[i]!=0):
        self.Y[i] = 1

    return(self.Y)

```

This following function is straightforward; it computes the probability of a data point  $x_i$ , yielding a number between zero and one.

```

def prob(self,x_i,beta):

    x_i = np.reshape(x_i,(1,len(x_i)))
    P_i = (1.0/(1.0 + np.exp(-np.dot(x_i,beta))))

    return(P_i)

```

The *training* function is the core of the algorithm. It starts by adding a column of ones to the matrix containing the variables. Then, it starts to iterate and update the gradient, Hessian matrix and the beta coefficients on each iteration.

```

def training(self):

    X = self.extend(self.X)
    X_T = np.transpose(X)
    self.Y = self.adjust_y()

    for it in range(self.iterations):

        P = [self.prob(X[i,:],self.beta) for i in range(len(X))]
        P = np.reshape(P,(len(P),1))
        Q = P*np.eye(len(P))*(1-P)

        grad = np.dot(X_T,(self.Y - P))
        H = -np.dot(X_T,np.dot(Q,X))

        self.beta = self.beta - self.lr*np.dot(inv(H),grad)

    return(self.beta)

```

The prediction function extends the testing data and then computes the probabilities of the testing data by calling the *prob* function.

```

def prediction(self,X_test):

    if np.sum(self.beta) == 0:
        self.beta = self.training()

    X_test = self.extend(X_test)
    P = [self.prob(X_test[i,:],self.beta) for i in range(len(X_test))]
    P = np.reshape(P,(len(P),1))

    return(P)

```

To end the code, we have the *accuracy* function, which computes the difference between the predictions and the actual outcomes of the testing data and stores them in a variable called *error-array*. Finally, it counts how many values of this array have an absolute value bigger or equal to 0.5, i.e., how many terms have been wrongly predicted, and it returns the model's accuracy.

```

def accuracy(self,X_test,Y_test):

    Y_test = np.reshape(Y_test,(len(Y_test),1))
    test_values = len(Y_test)

    error_array = self.prediction(X_test) - Y_test
    error_count = np.count_nonzero(abs(error_array)>=0.5)

    return(100*(test_values - error_count)/test_values)

```

To test the model, we will start working with the [4] breast cancer dataset, which stores if a tumor is benign or malignant in terms of several variables such as *mean radius*, *mean texture*, *mean perimeter*, *mean area*, etc. Next, we can see how to load the dataset from the *sklearn* data collection:

```
[2]: from sklearn.datasets import load_breast_cancer
import pandas as pd

cancer = load_breast_cancer()
data_pd = pd.DataFrame(cancer.data)
outcome_pd = pd.DataFrame(cancer.target)

X = data_pd.iloc[:,0].values
X = (X - np.mean(X, axis=0))/np.std(X, axis=0)
X = np.reshape(X,(len(X),1))
Y = outcome_pd.values
```

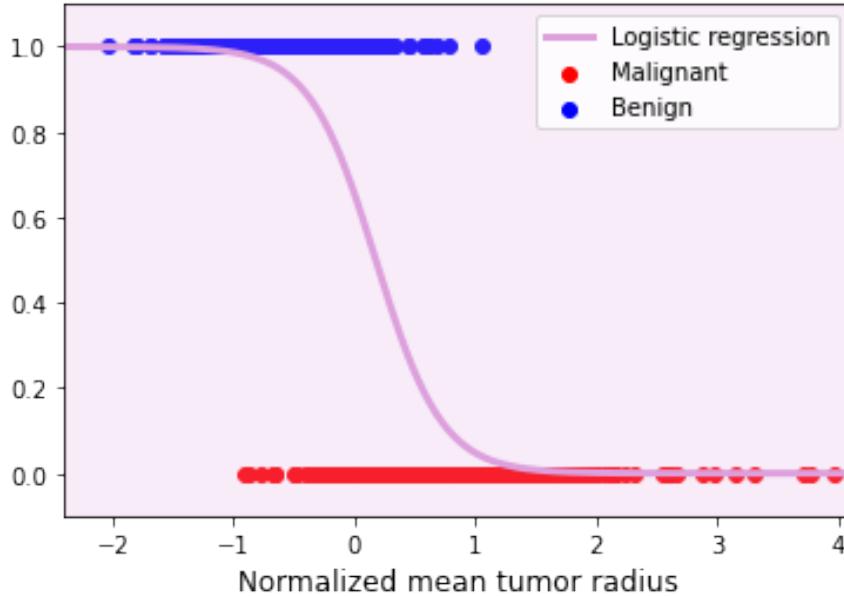


Figure 5.5: Scatter plot of the classification of a tumor in terms of the normalized mean radius of the tumor and a logistic regression on the data.

Now, we will use 5-fold cross-validation; the *Kfold* module from *Sklearn* splits the data automatically. The data will be split into five folds. Each will be set as testing data, while the remaining four folds will be training data. Hence, we will have five times the number of iterations specified on the cross-validation function.

```
[3]: from sklearn.model_selection import KFold

def K_fold_cross_validation_LOGI(X,Y,iterations):

    precision_values = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])
            X_test = np.array([X[i] for i in test_index])
            Y_test = np.array([Y[i] for i in test_index])

            reg = logistic(X_train,Y_train,200)
```

```

precision = reg.accuracy(X_test,Y_test)
precision_values.append(precision)

return(np.mean(precision_values))

```

As we can see in the results below, we obtain an accuracy of 87.79 % on the testing data.

```
[4]: print("The avg total accuracy is: ",
      K_fold_cross_validation_LOGI(X,Y,200), " %")
```

The avg total accuracy is: 87.78952802359883 %

Next, we will work with the Iris dataset. As I hinted before, whenever we encounter a dataset with more than two outcomes, we will make a regression for every possible pair of outcomes and then average the results to make predictions on the whole data.

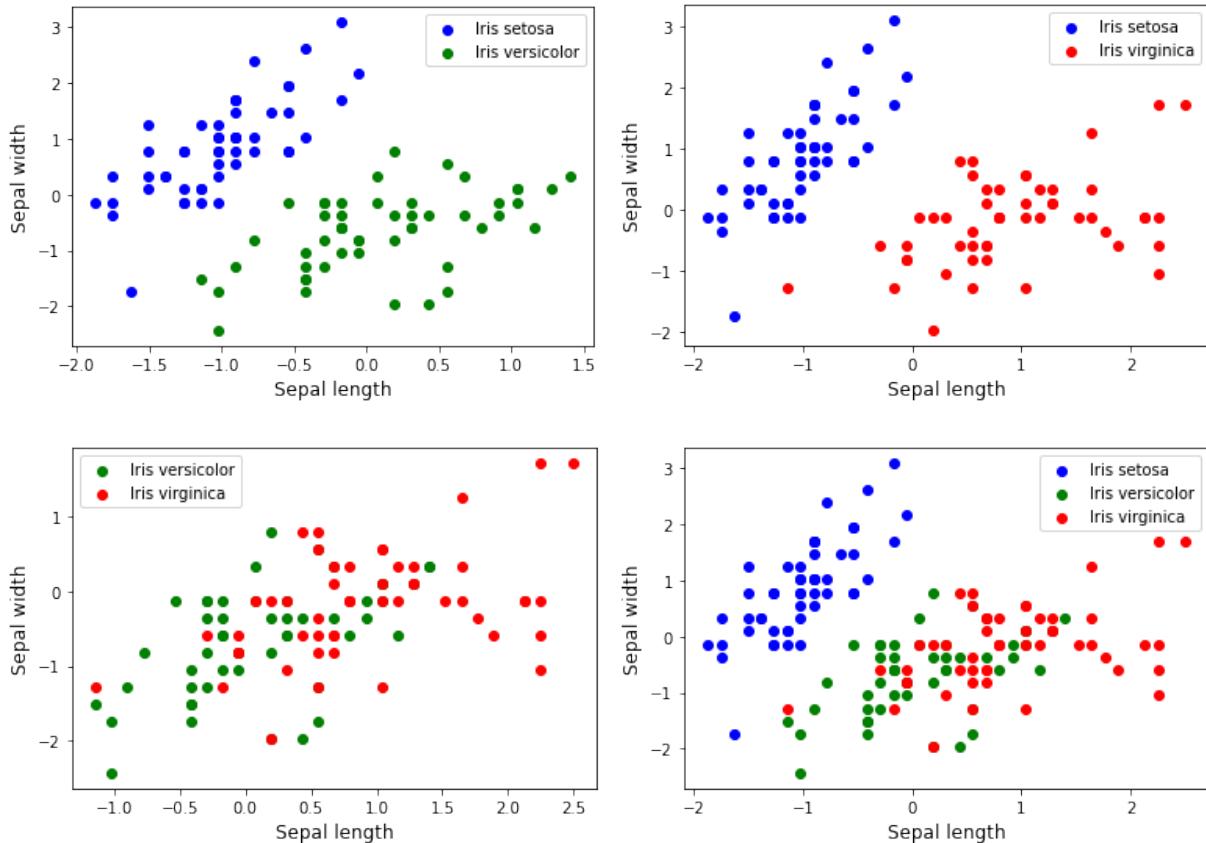


Figure 5.6: The plot of the sepal length and width variables of the Iris data: *Iris Setosa* vs. *Iris versicolor*, *Iris Setosa* vs. *Iris virginica*, *Iris versicolor* vs. *Iris virginica*, and the three outcomes combined.

If we apply the model to the different pairs of flowers separately and then make predic-

tions on the 2D space, we can find the results in Figure 5.7. To combine the three results into one plot, I did the following process (*one vs one* method). First we need to mention that the outcomes for *Iris Setosa*, *Iris Versicolor* and *Iris Virginica* are labeled by **0**, **1** and **2** respectively. I will refer to these three types of flowers as **A**, **B**, and **C**, respectively. On the first pairing, we will have **A** and **B**, and the code will keep the labels 0 and 1 for these two classes. When we predict an outcome for a particular point, this value will be  $P_1 \in [0, 1]$  and hence the probability of it being **A** will be  $(1 - P_1)$  and the probability of belonging to **B** will be  $P_1$ . On the second pairing, we have **A** with a label of 0 and **C**, for which the code will shift the label from 2 to 1. And hence the probability of a point being **A** will be  $(1 - P_2)$  and **C** will be  $P_2$ . Finally, the last pairing will have **B** with the label 1, and **C** will have the label shifted from 2 to 0. Hence, the probability of **B** will be  $P_3$  and the probability of **C** is given by  $1 - P_3$ . Now, to predict the outcome of this point, we build the combined probability of each class by averaging the results of the different configurations and then assign the point the outcome of the class with the highest probability.

$$\begin{aligned} P_{Setosa} &= (1 - P_1)/3 + (1 - P_2)/3 \\ P_{Versicolor} &= (P_1 + P_3)/3 \\ P_{Virginica} &= P_2/3 + (1 - P_3)/3 \end{aligned} \quad (5.25)$$

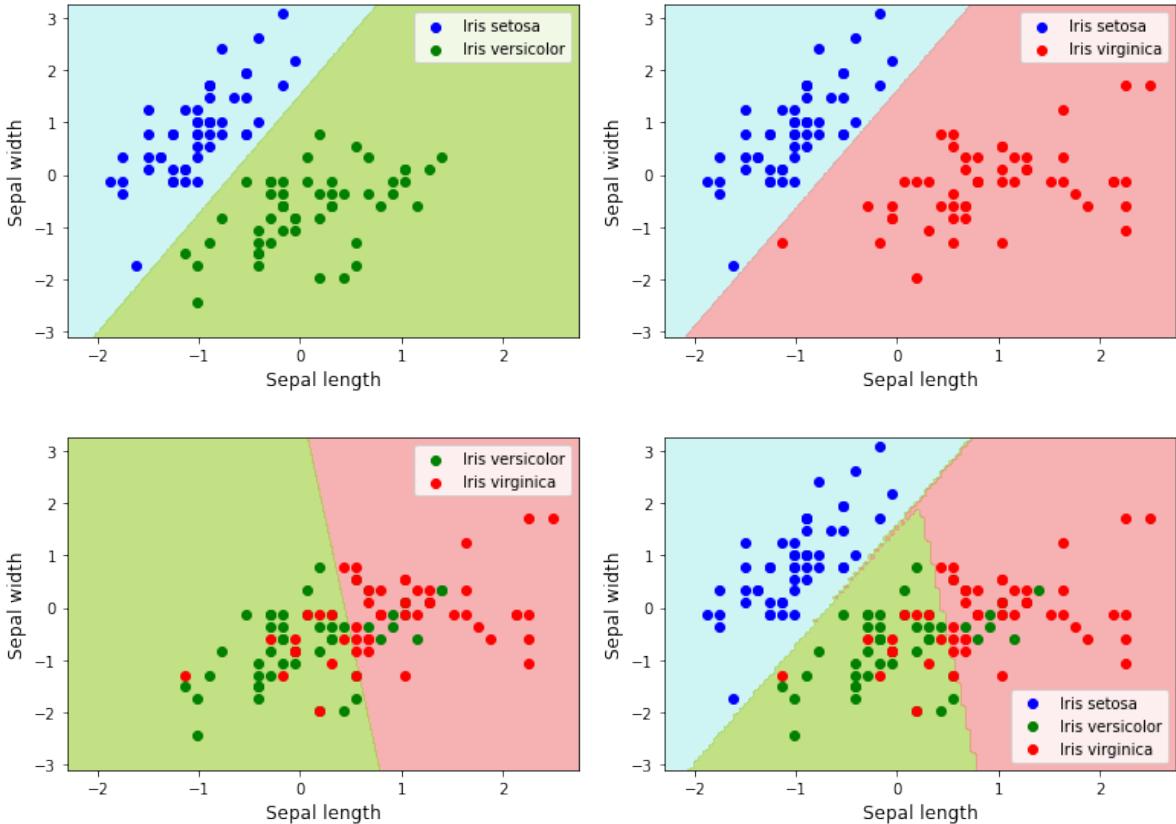


Figure 5.7: Logistic regression on the Iris dataset by taking Iris class pairs and plot of the combination of these results.

Last, we have the following code, which merges cross-validation and the concept I just explained of merging the results of the label pairs.

```
[5]: from sklearn.model_selection import KFold

def multi_K_fold_cross_validation_LOGI(X,Y,iterations):

    train_acc = []
    test_acc = []
    total_acc = []
```

This first piece of code splits the data into five folds for each iteration, setting each of these folds as the testing data. Then, it generates the training data, gathering the labels on pairs: zero and one, zero and two, and lastly, one and two.

```
for randomstates in range(iterations):

    KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
    KF.get_n_splits(X)

    for i, (train_index, test_index) in enumerate(KF.split(X)):

        X_train_01 = np.array([X[i] for i in train_index if Y[i]!=2])
        Y_train_01 = np.array([Y[i] for i in train_index if Y[i]!=2])
        X_train_02 = np.array([X[i] for i in train_index if Y[i]!=1])
        Y_train_02 = np.array([Y[i] for i in train_index if Y[i]!=1])
        X_train_12 = np.array([X[i] for i in train_index if Y[i]==0])
        Y_train_12 = np.array([Y[i] for i in train_index if Y[i]==0])
```

Now, we train the model with the different training data formed by the label pairs, and we predict the outcomes for the whole data, which will be stored in three different arrays. Then we build the probabilities of the classes **A**, **B** and **C** for each point by adding the appropriate contributions from the three regressions.

```
reg_01 = logistic(X_train_01,Y_train_01,200)
pred_01 = reg_01.predict(X)

reg_02 = logistic(X_train_02,Y_train_02,200)
pred_02 = reg_02.predict(X)

reg_12 = logistic(X_train_12,Y_train_12,200)
pred_12 = reg_12.predict(X)

blue_prob = ((1-pred_01) + (1-pred_02))/3
```

```

green_prob = (pred_01 + pred_12) /3
red_prob = (pred_02 + (1-pred_12))/3

```

The next step is to iterate over all the data points and generate an array labeled as  $probs_i$ , where we will store the probability of each label being the outcome of the  $i^{th}$  data point. Then, we save as the prediction for that point the outcome of the class with the highest probability, i.e., the position on which the maximum value is saved.

```

predictions = np.zeros(np.shape(blue_prob))

for i in range(len(blue_prob)):

    probs_i=np.array([blue_prob[i][0],green_prob[i][0],
                      red_prob[i][0]])
    predictions[i][0] = np.where(probs_i == \
                                np.max(probs_i))[0][0]

```

Once we have a prediction for all the points, we compute the difference between the predictions and the actual outcomes and count how many of these values equal zero to compute the accuracy. We will also do this after filtering the data points that have been employed to train the model and those that have not (testing points). This will be done to compute the accuracy of the training data, the testing data, and the global accuracy.

```

# COMPUTING THE ERRORS
error = predictions - np.reshape(Y,(len(Y),1))
test_error = np.array([error[i] for i \
                      in range(len(error)) if i in test_index])
train_error = np.array([error[i] for i \
                      in range(len(error)) if i in train_index])

# COUNTING THE CORRECT PREDICTIONS
correct_train_pred = np.count_nonzero(train_error==0)
correct_test_pred = np.count_nonzero(test_error==0)
correct_total_pred = np.count_nonzero(error==0)

# COMPUTING THE ACCURACY
train_acc.append(100*(correct_train_pred/len(train_error)))
test_acc.append(100*(correct_test_pred/len(test_error)))
total_acc.append(100*(correct_total_pred/len(error)))

return(np.mean(total_acc),np.mean(train_acc),np.mean(test_acc))

```

If we now execute the code for the first 200 random states (a total of 1000 iterations if we take into account that we are using 5-fold cross-validation), we obtain the following results:

```
[6]: accuracy,train_acc,test_acc=multi_K_fold_cross_validation_LOGI(X,Y,200)
      print("The avg total accuracy is: ",accuracy,"%")
      print("The avg accuracy on training data is: ",train_acc,"%")
      print("The avg accuracy on testing data is: ",test_acc,"%")
```

```
The avg total accuracy is: 81.78133333333334 %
The avg accuracy on training data is: 82.27083333333334 %
The avg accuracy on testing data is: 79.82333333333334 %
```

## 5.2 K-nearest neighbors [Supervised]

### 5.2.1 Mathematical derivation

Although implementing it might be challenging, KNN is quite simple in concept and we can use it for classification or regression purposes. The idea behind this algorithm is to consider the k nearest neighbors of a point we want to predict and use their outcome to make a prediction. If we set k to be a low integer, we will have an overfit model since only the nearest neighbors will have impact, and if k is too large, we will have an underfit model where the data's reach will be large.

#### CLASSIFICATION

When using the k-nearest neighbors algorithm for classification, we will consider a point from the data space and find the k-nearest neighbors; then, we will assign to that point the most common label among the k-nearest neighbors. So if, for example, we consider k=4 to classify a point on a binary data collection, and half the neighbors correspond to one class and the other half to the second class, that point will be unclassified, i.e., part of the boundary that divides the classes.

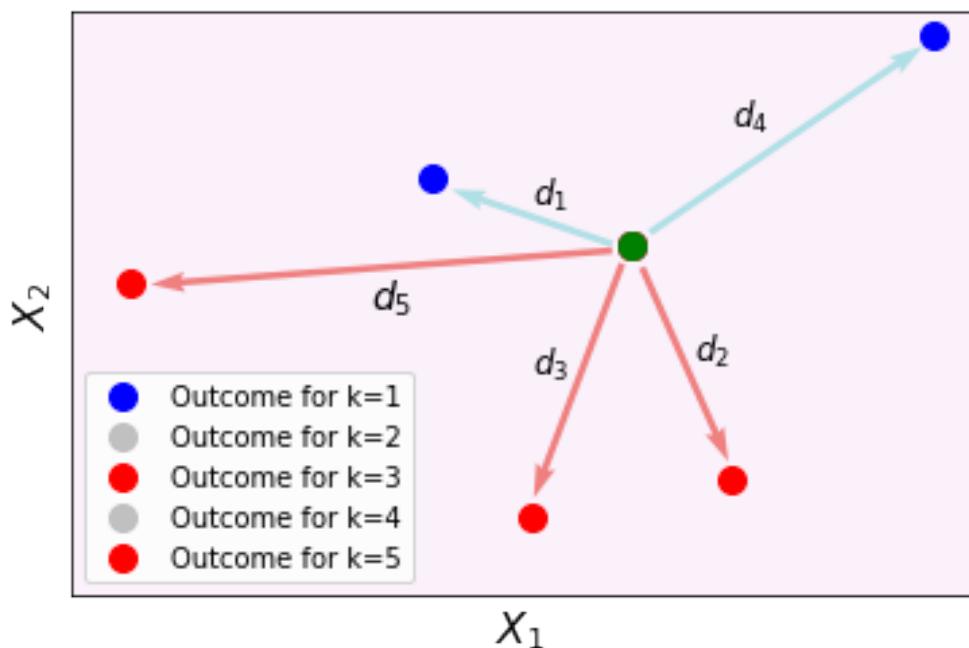


Figure 5.8: K-NN classification in terms of a 2D variable.

The outcome of the green point is shown in the plot legend for a different number of neighbors. It must be mentioned that  $d_1 < d_2 < d_3 < d_4 < d_5$ , so for instance, if we only consider the closest neighbor, i.e., k=1, the outcome is blue.

## REGRESSION

Whenever we use KNN for regression purposes, the process is similar. However, instead of classifying the point based on which label is more common among the nearest neighbors, the output will be the average of the outputs of the  $k$  nearest neighbors.

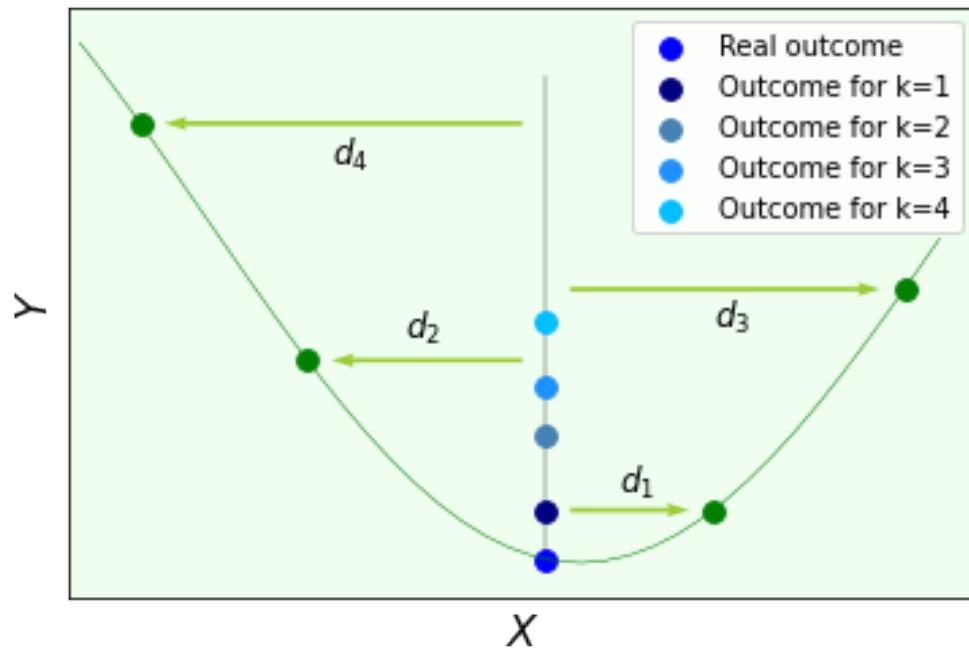


Figure 5.9: K-NN regression in terms of a 1D variable.

To illustrate the KNN regression, I only considered one variable  $x$ , and an outcome  $y$ . We have that  $d_1 < d_2 < d_3 < d_4$ . In the plot legend, we can see the outcomes when considering a different number of neighbors, given that these outcomes are an average of the  $y$ -values of the neighbors.

KNN is often used for classification purposes more than regression; modifying the algorithm for regression is trivial, so I will only implement it for the first case. It is also worth mentioning that normalizing the data is mandatory for this model since the idea of computing a distance breaks if the magnitude of one of the variables is larger than that of the other variables.

## 5.2.2 Algorithm implementation

Next, we have the implementation of the KNN model for two variables  $X$  and  $Y$  and the outcome  $Z$ .

```
[1]: import numpy as np
import math as math
class KNN_classification():

    def __init__(self,X,Y,Z,k,untie):

        self.X=X
        self.Y=Y
        self.Z=Z
        self.k=k
        self.untie=untie

    def distance(self,X_1,Y_1,X_2,Y_2):

        d=math.sqrt(((X_1-X_2)**2.0) + ((Y_1-Y_2)**2.0))

        return(d)
```

This function computes the distance from all the data points to a certain point  $(X_0, Y_0)$ . Then, we order the indexes based on these distances (from the index associated to the smallest distance to the largest). Only the first  $k$  indexes, i.e., those associated with the  $k$  smallest distances, are kept as neighbors.

```
def neighbor_list(self,X0,Y0,k):

    indexes = np.arange(0,len(self.X),1)
    distances = np.array([self.distance(self.X[i],
                                         self.Y[i],X0,Y0) for i in indexes])

    indexes = [x for _,x in sorted(zip(distances,indexes))]
    neighbors = indexes[0:k]

    return(neighbors)
```

The following function computes the prediction on new data; it obtains the  $k$  nearest neighbors by calling the previous function and counts the different labels among the neighbors for each testing data point. Then, depending on which of these values is

higher, it assigns the proper outcome. If there is a tie between the counts, the model runs into two options. The first is to set a value of three to those points where we encounter a tie, i.e., unclassified points. Also, there is a parameter labeled as *untie* on the code, which, if enabled, forces the code to predict an outcome by calling the function *tiebreak*.

```
def predict(self,X_test,Y_test):

    Z_predict = np.zeros(np.shape(X_test))

    for i in range(len(X_test)):

        neighbors = self.neighbor_list(X_test[i],Y_test[i],self.k)
        neighbor_outcomes = np.array([self.Z[i] for i in neighbors])

        zero_count = np.count_nonzero(neighbor_outcomes == 0)
        one_count = np.count_nonzero(neighbor_outcomes == 1)
        two_count = np.count_nonzero(neighbor_outcomes == 2)

        if (zero_count > one_count and zero_count > two_count):

            Z_predict[i] = 0

        elif (one_count > zero_count and one_count > two_count):

            Z_predict[i] = 1

        elif (two_count > zero_count and two_count > one_count):

            Z_predict[i] = 2

        else:

            if (self.untie == "no"):

                Z_predict[i] = 3

            else:

                Z_predict[i] = self.tiebreak(X_test[i],Y_test[i],
                                             neighbors,neighbor_outcomes)

    return(Z_predict)
```

The *tiebreak* function helps us label those points where we have a tie between two or more outcomes. We obtain the nearest neighbor among the tied neighbors and assign its label to the point. The function first counts the number of zeros, ones, and twos among the outcomes of the neighbors and stores them on an array. Then, it starts iterating on the list of neighbors, which are ordered based on the distance from lowest to largest; if the outcome of one of the neighbors has a count equal to the maximum value on the count array, then the outcome of the neighbor is returned by the function.

```
def tiebreak(self,x_test,y_test,neighbors,neighbor_outcomes):

    zero_count = np.count_nonzero(neighbor_outcomes == 0)
    one_count = np.count_nonzero(neighbor_outcomes == 1)
    two_count = np.count_nonzero(neighbor_outcomes == 2)

    count = np.array([zero_count,one_count,two_count])

    for i in neighbors:

        outcome = self.Z[i]

        if count[outcome] == np.max(count):
            break

    return(outcome)

def accuracy(self,X_test,Y_test,Z_test):

    error_array = self.predict(X_test,Y_test) - Z_test
    right_predictions = np.count_nonzero(error_array == 0)

    return(100*right_predictions/len(Z_test))
```

Next, this code allows us to make fast plots based on the KNN model. It first creates a data grid (which is adapted for the iris dataset on which we will test the model). Then, it calls for the KNN model, and last, it makes a contour plot of the results.

```
[2]: def PLOT(X,Y,Z,k,untie):

    # CREATING TESTING GRID
    x_grid, y_grid = np.meshgrid(np.arange(-2.3,2.8,0.05),
                                 np.arange(-3.1,3.3,0.05))
    a,b = len(x_grid),len(x_grid[0])
    x_flat, y_flat = np.reshape(x_grid,(a*b,)), np.reshape(y_grid,(a*b,))

    # USING THE K-NN MODEL
```

```

reg = KNN_classification(X,Y,Z,k,untie)
z_flat = reg.predict(x_flat, y_flat)
z_grid = np.reshape(z_flat,(a,b))

# PLOT
color=["blue","green","red"]

plt.title("K = "+str(k)+", tiebreak = "+str(untie))
for i in range(len(X)):

    plt.scatter(X[i],Y[i],color=color[Z[i]],s=6,zorder=2)

if untie == "yes":

    plt.contourf(x_grid,y_grid,z_grid,2,colors=[ "paleturquoise",
                                                 "yellowgreen", "lightcoral"],alpha=0.6)

else:

    plt.contourf(x_grid,y_grid,z_grid,3,colors=[ "paleturquoise",
                                                 "yellowgreen", "lightcoral", "grey"],alpha=0.6)

```

We can observe the result of the algorithm on the Iris data set for  $K=5$ , both with the *untie* parameter on and off. As we see on the first graph, the model uses the  $K=5$  nearest neighbors to assign an outcome to every point of the 2D space. Some points are tied among different classes and hence are plotted with grey. When we turn the *untie* parameter on, the model is forced to pick the nearest neighbor among the tied labels.

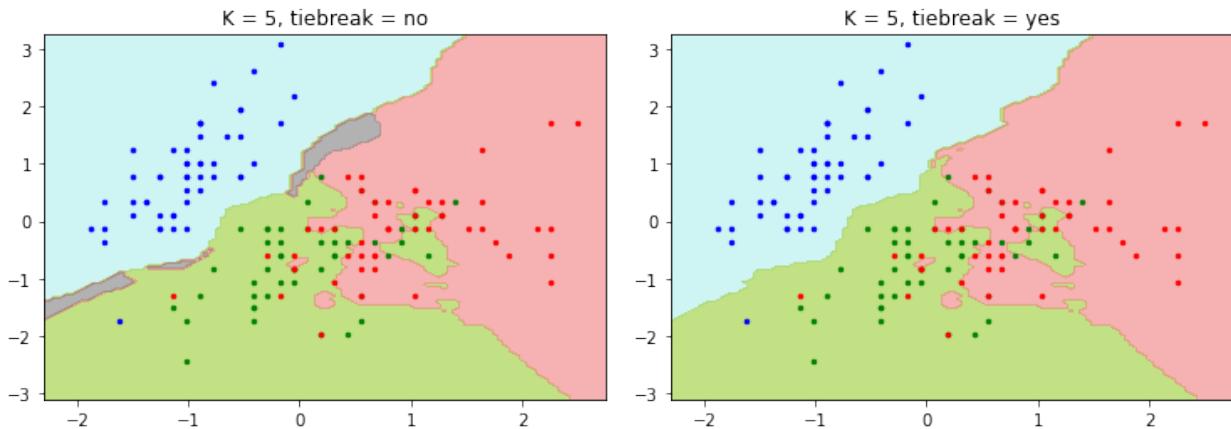


Figure 5.10: KNN model for  $K=5$  with tiebreak and without tiebreak between classes on the Iris dataset.

I also tested the model for different values of  $k$  to see how it evolves. As we can see, when  $k$  is low, we have an overfit model with a lot of isles, where delimiting the areas belonging to each class is harder. As we increase the value of  $k$ , the model becomes more simple, and the areas of the different classes are easily delimited. The other case that will not predict data well is when  $k$  grows too large, leading to an underfit model.

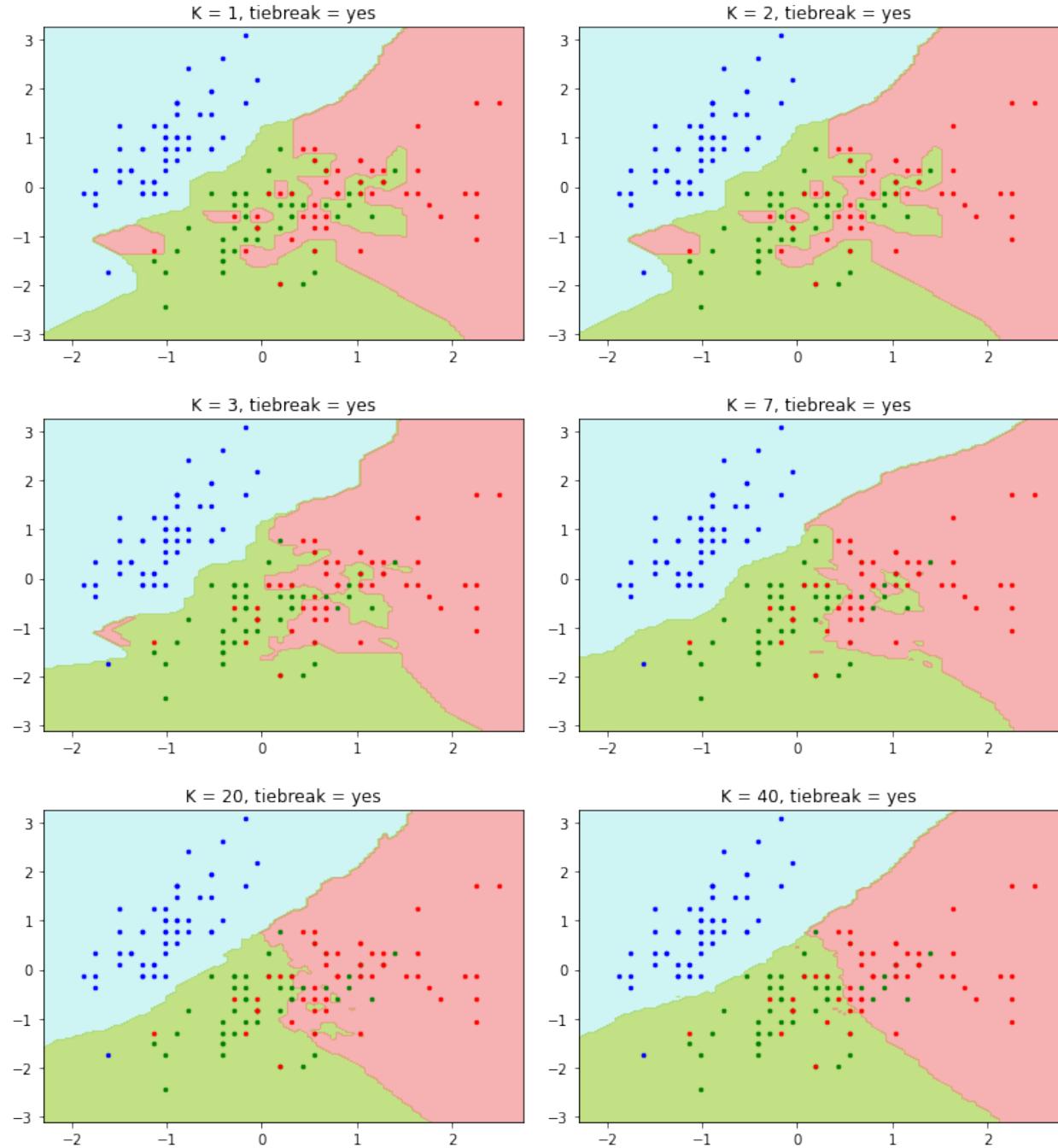


Figure 5.11: KNN model for different values of  $K$  on the Iris dataset.

The last step is to use cross-validation to test our model on the Iris dataset, similar to what we did with the previous algorithm.

```
[3]: from sklearn.model_selection import KFold

def K_fold_cross_validation_KNN(X,Y,k,iterations):
    accuracy = []
    test_accuracy = []
    train_accuracy = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])

            reg = KNN_classification(X_train[:,0],X_train[:,1],
                                      Y_train,k,untie = "yes")

            # PREDICTIONS
            prediction = reg.predict(X[:,0],X[:,1])

            # ERRORS
            error = prediction - Y
            test_error = np.array([error[i] for i in \
                                  range(len(error)) if i in test_index])
            train_error = np.array([error[i] for i in \
                                  range(len(error)) if i in train_index])

            # CORRECT PREDICTIONS
            correct_pred = np.count_nonzero(error == 0)
            correct_test_pred = np.count_nonzero(test_error == 0)
            correct_train_pred = np.count_nonzero(train_error == 0)

            accuracy.append(100*correct_pred/len(Y))
            test_accuracy.append(100*correct_test_pred/len(test_index))
            train_accuracy.append(100*correct_train_pred/len(train_index))

    return(np.mean(accuracy),np.mean(test_accuracy),
           np.mean(train_accuracy))
```

First, we start by executing our 5-fold cross-validation function for the first 200 random states for the neighbor numbers ranging from 1 to 50. We then plot the accuracy results over the training data, the testing data, and a combination of both.

```
[4]: k = np.arange(1,51,1)
results = np.array([K_fold_cross_validation_KNN(X,Y,
                                                k_i,200) for k_i in k])

accuracy = results[:,0]
test_accuracy = results[:,1]
train_accuracy = results[:,2]
```

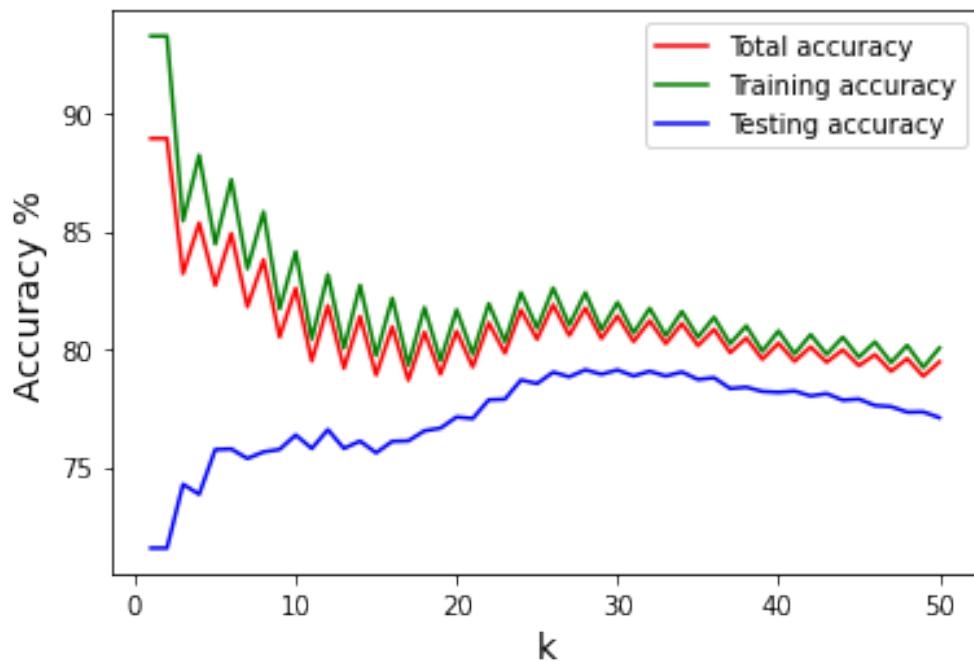


Figure 5.12: Accuracy of the KNN model on the Iris data collection.

As shown in the plot, when  $k$  is small, the accuracy of the training data rises, and the accuracy of the testing data drops due to having an overfit model. Since we want to use our model to predict new data, we want to keep the parameter that achieves a better performance on the testing data.

```
[5]: max_testing = np.where(test_accuracy == \
                           np.max(test_accuracy))[0][0]
print("The maximum of the testing data accuracy is for k =", 
      k[max_testing])
print("Accuracy for k =", k[max_testing])
print("-----")
print("Training data: ", train_accuracy[max_testing], "%")
```

```
print("Testing data: ",test_accuracy[max_testing],"%")
print("All data: ",accuracy[max_testing], "%")
```

The maximum of the testing data accuracy is for k = 28  
Accuracy for k = 28

-----  
Training data: 82.3975 %  
Testing data: 79.12333333333335 %  
All data: 81.74266666666665 %

As we can see, the most optimal results are found for k = 28, achieving an accuracy similar to the Logistic regression.

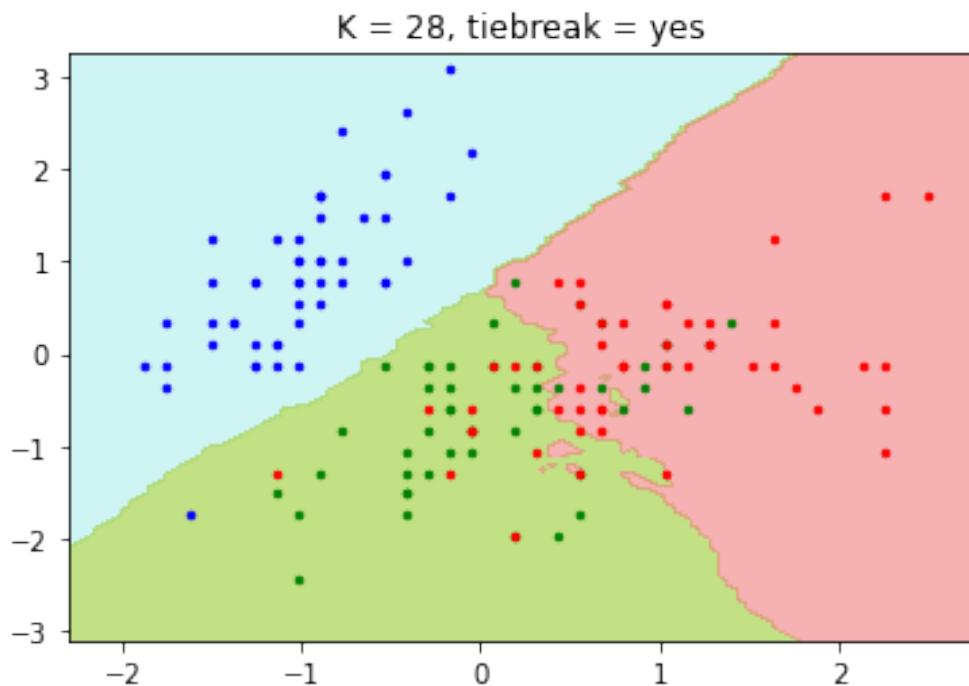


Figure 5.13: KNN model for  $k = 28$  on the Iris dataset.

## 5.3 LVQ [Supervised]

### 5.3.1 Mathematical derivation

[18] This algorithm is quite simple in concept, and hence, it will also be explained in a few steps:

- First, we introduce a set of neurons (points). These neurons will be the points that will predict our data based on proximity. We can choose how many neurons to introduce; if, for instance, we have three different possible outcomes, the simplest case will be to introduce three neurons, one of each data class. I chose to draw the neurons from the training set and then exclude these training points from the set.
- These neurons will have a weight  $\mathbf{W}_m$  assigned; this weight will be the coordinates of the neuron in the variable space. If we choose a training data point as a neuron, the weight will be the coordinates of the data point.

$$\mathbf{W}_m = (X_{m,1}, X_{m,2} \dots X_{m,D-1}) \quad (5.26)$$

- We will feed the algorithm training data once our neurons have an initial weight (position). We will start iterating through the different training points. The algorithm will choose the neuron closest to each data point  $\mathbf{X}_i$ , and depending on whether the outcome of the neuron and the training data point match or not, it will define a new weight for the neuron given by:

$$\mathbf{W}_m^* = \mathbf{W}_m \pm k(\mathbf{X}_i - \mathbf{W}_m) \quad (5.27)$$

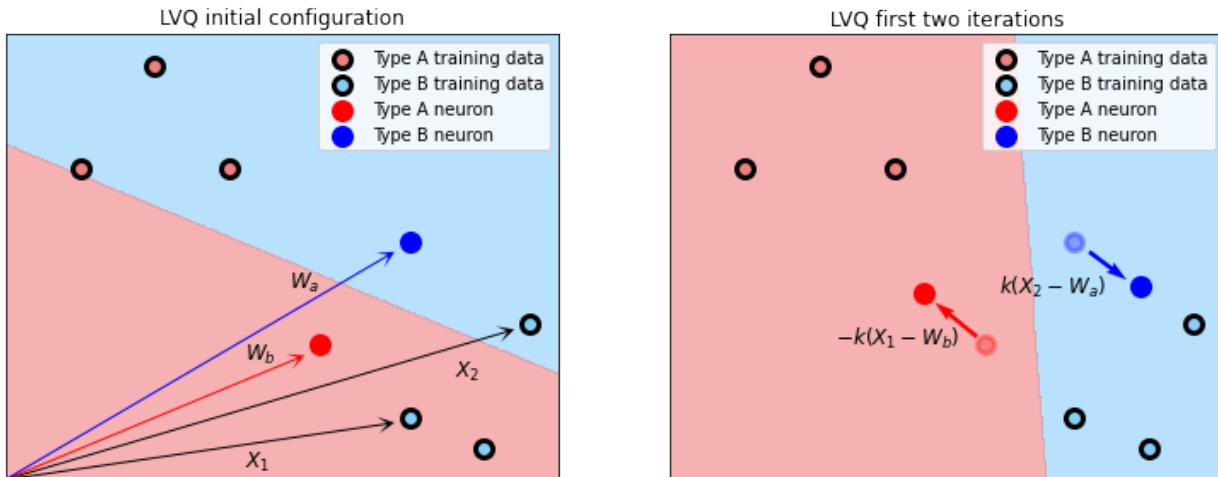


Figure 5.14: Visualization of the weight update of the neurons on the LVQ model.

As we can see in the plot I created, whenever the outcome of the neuron and the data point do not match, the neuron is displaced away from the data point. Moreover, whenever they match, it is moved towards the data point.

This process is done for every single data point, and for the number of epochs determined as an input, the system will slowly run into a stationary state.

- [32] Once we have iterated over all the epochs and we have a final configuration of neurons, we can implement a function that removes inactive or irrelevant neurons. We can run the algorithm for another epoch and remove those neurons with low interactions with the training data (this also removes those neurons that have been pushed away from the data cluster). We can also check if there are neurons whose interactions with training points of their same label are low, i.e., misplaced neurons, and replace them for neurons of the class they interact with the most.
- Lastly, we will make predictions on the data based on the proximity to the neurons, using the Euclidean distance to measure it.

### 5.3.2 Algorithm implementation

We proceed with the implementation of the LVQ model. Despite the logic behind the algorithm being simple, the implementation was challenging.

```
[1]: import numpy as np
import math as math
import random as random
import copy as copy

class LVQ_classification():
```

The LVQ model takes the following variables as input. First, we have the training data given by a pair of variables  $\{X, Y\}$  and an outcome  $Z$ . Then, three integers indicate the number of neurons of the model for each label, and last, the learning rate, the number of epochs (i.e., iterations), and a parameter labeled as *fixing*. If the input of this last parameter is *yes*, we will prune some of the neurons and change the outcome of others based on their encounter with the training data.

```
def __init__(self,X,Y,Z,A_neurons,B_neurons,C_neurons,
            lr,epochs,fixing):

    self.X = X
    self.Y = Y
    self.Z = Z

    self.A_neurons = A_neurons
    self.B_neurons = B_neurons
    self.C_neurons = C_neurons
    self.neurons = A_neurons + B_neurons + C_neurons

    self.lr = lr
    self.epochs = epochs
    self.fixing = fixing

    self.min_activity=len(self.X)/(2*self.neurons)

def distance(self,X_1,Y_1,X_2,Y_2):
    D = math.sqrt(((X_1-X_2)**2.0) + ((Y_1-Y_2)**2.0))

    return(D)
```

The following function generates the initial configuration of the neurons. As I briefly introduced in the *mathematical derivation* section, the neurons are taken from the training data. First, we create a random integer array of length three times the length of the training data, which contains integers ranging from zero to the size of the training data minus one. Then, we divide all these indexes into three different arrays depending on their outcome, and we keep only the first A, B, and C values respectively, where A, B, and C represent the neuron number for each label. Last, we build the weight functions for the neurons and update the values of the training data, excluding those values associated with the indexes saved as neurons.

```

def neuron_generator(self):

    r_index = np.array([random.randint(0, len(self.X)-1) \
                       for i in range(3*len(self.X))])

    A_neurons = np.array([r_index[i] for i in range(len(r_index)) \
                          if self.Z[r_index[i]]==0])[0:self.A_neurons]

    B_neurons = np.array([r_index[i] for i in range(len(r_index)) \
                          if self.Z[r_index[i]]==1])[0:self.B_neurons]

    C_neurons = np.array([r_index[i] for i in range(len(r_index)) \
                          if self.Z[r_index[i]]==2])[0:self.C_neurons]

    neurons = np.concatenate((A_neurons, B_neurons, C_neurons))

    W_X = np.array([self.X[i] for i in neurons])
    W_Y = np.array([self.Y[i] for i in neurons])
    W_Z = np.array([self.Z[i] for i in neurons])

    self.X = np.array([self.X[i] for i in range(len(self.X)) \
                      if i not in neurons])
    self.Y = np.array([self.Y[i] for i in range(len(self.Y)) \
                      if i not in neurons])
    self.Z = np.array([self.Z[i] for i in range(len(self.Z)) \
                      if i not in neurons])

    return(W_X,W_Y,W_Z)

```

This function is the algorithm's core; it starts by initializing the neurons' weights. It also has three variables that will be used to save the results of the weights for every iteration. Then, we start to iterate for the number of epochs indicated by the input variable. We compute the distance from each data point to the different neurons for each iteration. After this, we update the weights of the neuron closest to the data point (the index of

the neuron closest to the data point is labeled as *min D* on the code). I also stored the neurons' number of encounters and wrong encounters in two variables labeled *enc* and *w-enc*. A wrong encounter is defined as an encounter of a neuron with a data point with a different outcome. The last step of this function is to call for the *pruning* and *neuron update* functions if indicated by the input variable *fixing*.

```

def training(self):

    W_X_history = []
    W_Y_history = []
    W_Z_history = []

    W_X,W_Y,W_Z = self.neuron_generator()

    W_X_history.append(copy.deepcopy(W_X))
    W_Y_history.append(copy.deepcopy(W_Y))
    W_Z_history.append(copy.deepcopy(W_Z))

    for it in range(self.epochs):

        enc = np.zeros((len(W_X),3))
        w_enc = np.zeros((len(W_X),3))

        for i in range(len(self.X)):

            D = np.array([self.distance(self.X[i],self.Y[i],
                                         W_X[m],W_Y[m]) for m in range(len(W_X))])

            min_D = np.where(D == np.min(D))[0][0]

            if (W_Z[min_D]==self.Z[i]):

                W_X[min_D]=W_X[min_D] + self.lr*(self.X[i]-W_X[min_D])
                W_Y[min_D]=W_Y[min_D] + self.lr*(self.Y[i]-W_Y[min_D])
                enc[min_D][self.Z[i]] = enc[min_D][self.Z[i]] +1

            else:

                W_X[min_D]=W_X[min_D] - self.lr*(self.X[i]-W_X[min_D])
                W_Y[min_D]=W_Y[min_D] - self.lr*(self.Y[i]-W_Y[min_D])
                enc[min_D][self.Z[i]] = enc[min_D][self.Z[i]] +1
                w_enc[min_D][self.Z[i]] = w_enc[min_D][self.Z[i]] +1

        W_X_history.append(copy.deepcopy(W_X))
    
```

```

W_Y_history.append(copy.deepcopy(W_Y))
W_Z_history.append(copy.deepcopy(W_Z))

if self.fixing == "yes":

    W_X, W_Y, W_Z, enc, w_enc = self.prune(W_X,W_Y,W_Z,enc,w_enc)
    W_X, W_Y, W_Z = self.update_neuron(W_X,W_Y,W_Z,enc,w_enc)

    W_X_history.append(copy.deepcopy(W_X))
    W_Y_history.append(copy.deepcopy(W_Y))
    W_Z_history.append(copy.deepcopy(W_Z))

return(W_X_history, W_Y_history, W_Z_history)

```

The pruning phase of the code iterates, setting the number of iterations equal to the number of neurons of the model as it is a safe number to make sure all the inactive neurons are removed. On each iteration, each neuron's total number of interactions is stored in an array, and then the index of the neuron with the lowest interactions is found. If this neuron's number of interactions is lower than an amount chosen by us, the weights of this neuron are removed. Then, the encounter arrays are updated to be analyzed on the next iteration to remove more neurons. In the first version of the code I wrote, I was simultaneously removing all the neurons with low interactions, which was a wrong approach. Let us imagine a configuration with a cluster of neurons of the same class close to each other. Each of these neurons may have a low encounter number due to the high density of neurons in that area. That is why removing them one by one is better, making the rest of the neurons increase their interactions as we remove others.

```

def prune(self,W_X,W_Y,W_Z,enc,w_enc):

    Z = self.Z

    for it in range(self.neurons):

        neuron_activity = np.array([np.sum(enc[i,:]) \
                                    for i in range(len(W_X))])
        inactive_N = np.where (neuron_activity == \
                              np.min(neuron_activity))[0][0]

        if neuron_activity[inactive_N] < self.min_activity:

            W_X = np.array([W_X[i] for i in range(len(W_X)) \
                           if i != inactive_N])
            W_Y = np.array([W_Y[i] for i in range(len(W_Y)) \
                           if i != inactive_N])

```

```

W_Z = np.array([W_Z[i] for i in range(len(W_Z)) \
                if i != inactive_N])

enc = np.zeros((len(W_X),3))
w_enc = np.zeros((len(W_X),3))

for i in range(len(self.X)):

    D = np.array([self.distance(self.X[i],self.Y[i],
                                W_X[m],W_Y[m]) for m in range(len(W_X))])

    min_D = np.where(D == np.min(D))[0][0]

    if (W_Z[min_D]==Z[i]):

        enc[min_D][Z[i]] = enc[min_D][Z[i]] +1

    else:

        enc[min_D][Z[i]] = enc[min_D][Z[i]] +1
        w_enc[min_D][Z[i]] = w_enc[min_D][Z[i]] +1

    else:

        break

return(W_X,W_Y,W_Z,enc,w_enc)

```

This following function is simple; it updates the neuron's outcome if its encounters with data points of the same class are lower than a third of the total interactions. It updates the label to that of the data class encountered the most by the neuron.

```

def update_neuron(self,W_X,W_Y,W_Z,enc,w_enc):

    for j in range(len(W_X)):

        if (np.sum(w_enc[j,:])/np.sum(enc[j,:]) +10**-10)>(2.0/3.0):

            W_Z[j]=np.where(w_enc[j,:]==np.amax(w_enc[j,:]))[0][0]

        else:
            continue

    return(W_X,W_Y,W_Z)

```

The `predict` function makes predictions on the testing data. First, it loads the values of the weights, then computes the distance from each testing point to the different neurons and assigns these testing points the outcome of the closest neuron. Then, we have another function that computes the model's accuracy.

```

def predict(self,X_test,Y_test):

    W_X_history, W_Y_history, W_Z_history = self.training()
    W_X = W_X_history[-1]
    W_Y = W_Y_history[-1]
    W_Z = W_Z_history[-1]

    pred = np.zeros((len(X_test)))

    for i in range(len(X_test)):

        D = np.array([self.distance(X_test[i],Y_test[i],
                                    W_X[m],W_Y[m]) for m in range(len(W_X))])
        min_D = np.where(D == np.min(D))[0][0]

        pred[i] = W_Z[min_D]

    return(pred,W_X_history,W_Y_history,W_Z_history)

def accuracy(self,X_test,Y_test,Z_test):

    error_array = self.predict(X_test,Y_test)[0] - Z_test
    right_predictions = np.count_nonzero(error_array == 0)

    return(100*right_predictions/len(Z_test))

```

Next, we have a piece of code to plot the model's results. It is composed of a double graph; on the left plot, we have the evolution of the neurons (a cross for the initial position and a point for the final configuration) and a contour plot of the prediction in the 2D space. And on the right plot, the same contour plot and the scatter of the training data.

```
[2]: def PLOT(X,Y,Z,A,B,C):

    # CREATING TESTING GRID
    x_grid, y_grid = np.meshgrid(np.arange(-2.3,2.8,0.05),
                                 np.arange(-3.1,3.3,0.05))
    a,b = len(x_grid),len(x_grid[0])
    x_flat, y_flat = np.reshape(x_grid,(a*b,)), np.reshape(y_grid,(a*b,))

    # USING THE K-NN MODEL
    reg = LVQ_classification(X,Y,Z,A,B,C,0.01,200,"yes")
    z_flat,W_X,W_Y,W_Z = reg.predict(x_flat, y_flat)
    z_grid = np.reshape(z_flat,(a,b))

    # PLOT
    fig, ax = plt.subplots(1, 2, figsize=(12.8,4))
    color=["blue","green","red"]

    ax[0].set_title("A,B,C type neuron numbers: " +
                    "+str(A)+" , "+str(B)+" , "+str(C))
    ax[1].set_title("Training data scattered over contour plot")

    #SCATTER OF DATA
    for i in range(len(X)):

        ax[1].scatter(X[i],Y[i],color=color[Z[i]],s=20,zorder=7)

    #INITIAL CONFIGURATION OF THE NEURONS
    for j in range(len(W_X[0])):

        ax[0].scatter(W_X[0][j],W_Y[0][j],
                      color=color[W_Z[0][j]],s=30,zorder=4,marker="x")

    #FINAL CONFIGURATION OF THE NEURONS
    for j in range(len(W_X[-1])):

        ax[0].scatter(W_X[-1][j],W_Y[-1][j],
                      color=color[W_Z[-1][j]],s=30,zorder=4)

    #TRAJECTORY OF THE NEURONS
    W_X_t = np.array(W_X[0:-2])
    W_Y_t = np.array(W_Y[0:-2])
    W_Z_t = np.array(W_Z[0:-2])

    for j in range(len(W_X_t[0])):
```

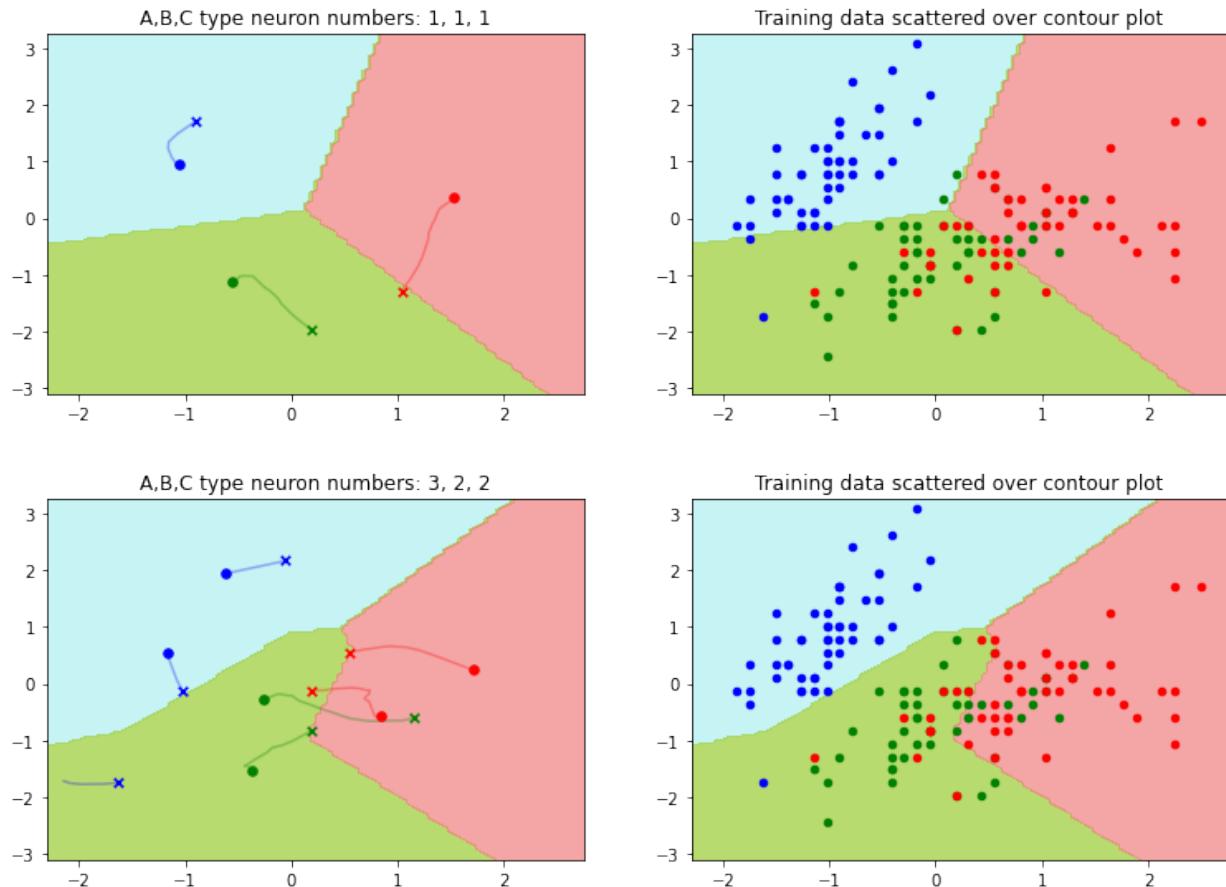
```

ax[0].plot(W_X_t[:,j],W_Y_t[:,j],
            color=color[W_Z_t[0,j]],alpha=0.3)

#PREDICTION ON THE 2D SPACE
ax[0].contourf(x_grid,y_grid,z_grid,2,colors=["paleturquoise",
                                              "yellowgreen","lightcoral"],alpha=0.7)
ax[1].contourf(x_grid,y_grid,z_grid,2,colors=["paleturquoise",
                                              "yellowgreen","lightcoral"],alpha=0.7)

```

We once again work with the Iris dataset. We can see some of the plots below for different choices of neuron numbers. On the graphs, we can see how the neurons evolve: their trajectory, and their final position. We can also see how the trajectory of some of the neurons ends up in a position where there are no final points; these correspond to neurons that have been pruned due to low interactions with the data. On the last graph, we can also see an example of a red neuron that changes to a blue neuron due to the *neuron update* function (this is due to this neuron having almost no interactions with its own class).



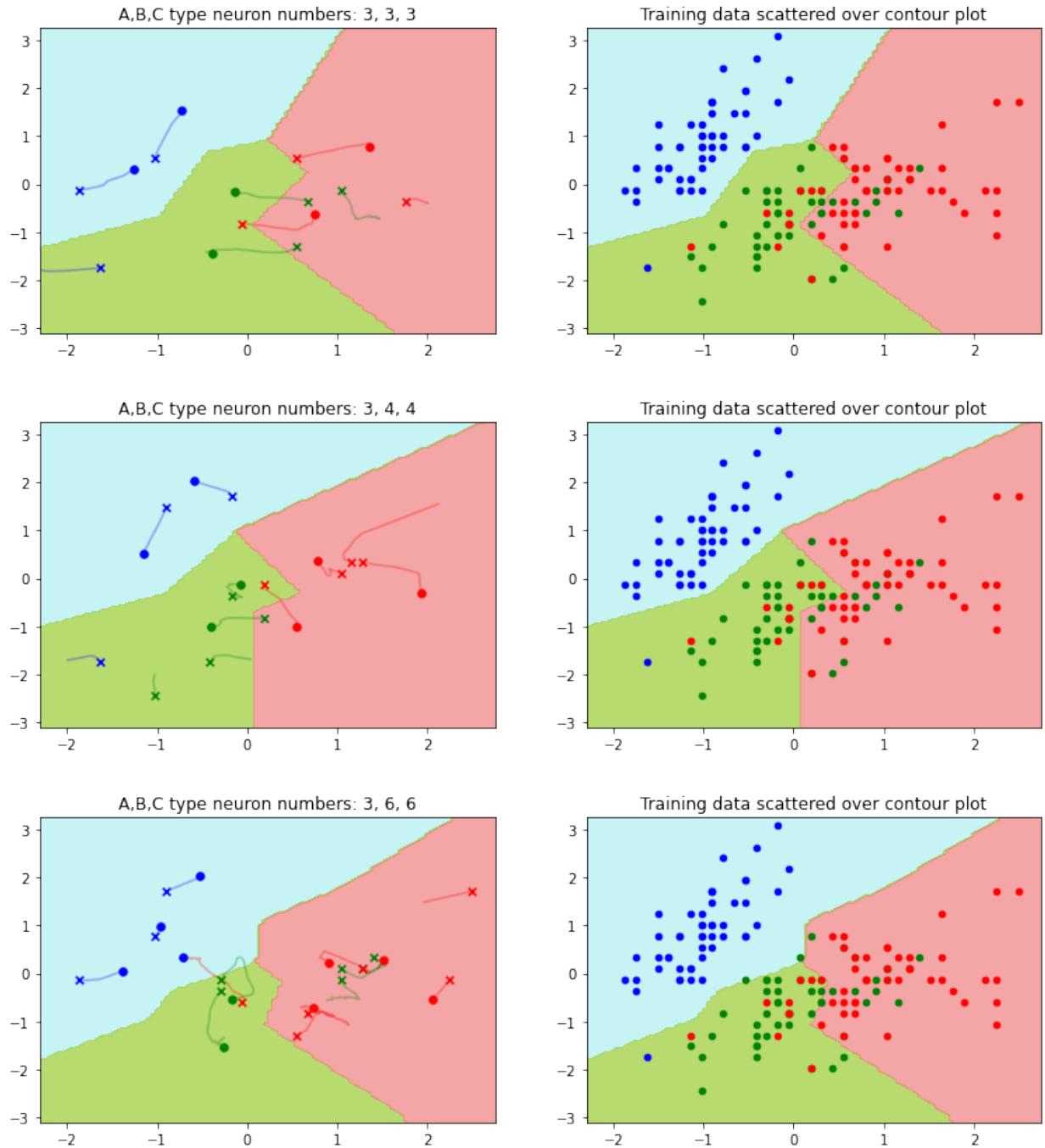


Figure 5.15: LVQ model showing the neuron evolution and posterior prediction on the 2D variable space for different numbers of neurons.

Next, we introduce a cross-validation function, which we will use to test the model's accuracy on the Iris dataset. This function is similar to those of previous models.

```
[3]: from sklearn.model_selection import KFold

def k_fold_cross_validation_LVQ(X,Y,A,B,C,iterations,fix):

    accuracy_values = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])
            X_test = np.array([X[i] for i in test_index])
            Y_test = np.array([Y[i] for i in test_index])

            reg = LVQ_classification(X_train[:,0],X_train[:,1],
                                      Y_train,A,B,C,0.01,200,fix)
            precision = reg.accuracy(X_test[:,0],X_test[:,1],Y_test)

            accuracy_values.append(precision)

    return(np.mean(accuracy_values))
```

We will test the cross-validation function on the Iris data, enabling and not enabling the *fixing* parameter. As in previous models, we will have 5-fold cross-validation and iterate over the first 200 random states. This computation took 20 hours to complete.

```
[4]: B = np.arange(1,9,1)
C = np.arange(1,9,1)
accuracy_fixing = np.zeros((len(B),len(C)))
accuracy_no_fixing = np.zeros((len(B),len(C)))

for i in range(len(B)):
    for j in range(len(C)):

        accuracy_fixing[i][j] = k_fold_cross_validation_LVQ(X,Y,
                                              3,B[i],C[j],200,"yes")
        accuracy_no_fixing[i][j] = k_fold_cross_validation_LVQ(X,Y,
                                              3,B[i],C[j],200,"no")

print("The max value of the accuracy fixing the neurons is: ",
      np.max(accuracy_fixing))
```

```
print("The max value of the accuracy without fixing the neurons is: ",  
      np.max(accuracy_no_fixing))
```

```
The max value of the accuracy fixing the neurons is: 77.47  
The max value of the accuracy without fixing the neurons is:  
77.46333333333332
```

As we can see, the maximum value of the accuracy when we activate the pruning of the neurons is slightly higher than when we do not. Although it may seem that this function did not achieve much improvement, we will see that it does indeed impact the result. We start by showing the contour plot of both cases, displaying the model's accuracy in terms of the neuron numbers (B and C).

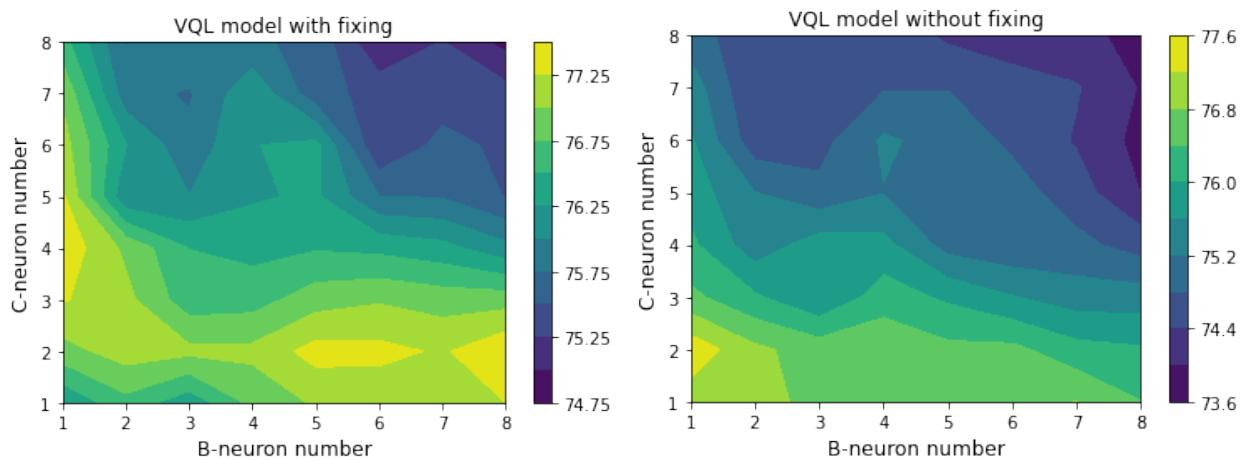


Figure 5.16: Accuracy numbers of the LVQ model with and without fixing the neurons in terms of the B and C type neurons.

To see the impact of the *prune* and *neuron update* functions, we will plot the difference between the accuracy of both configurations. The blue sections correspond to positive values of the difference. This case corresponds to an accuracy improvement when enabling the *fixing* parameter. Red sections correspond to negative values, i.e., neuron numbers for which the accuracy lowers when enabling the *fixing* parameter. We use the following code to make a contour plot (color mesh) of the difference.

```
[5]: from matplotlib.colors import LinearSegmentedColormap  
from matplotlib import colors  
fig,ax = plt.subplots()  
Blue_red_cmap = LinearSegmentedColormap.from_list('br',  
                                                 ["lightcoral", "w", "lightskyblue"], N=256)  
normalize = colors.TwoSlopeNorm(vcenter=0)  
contourplot=plt.pcolormesh(B, C, accuracy_fixing-accuracy_no_fixing,
```

```

cmap=Blue_red_cmap, shading="auto", norm=normalize)
cbar = fig.colorbar(contourplot)
plt.xlabel("B-neuron number", fontsize=12)
plt.ylabel("C-neuron number", fontsize=12)

```

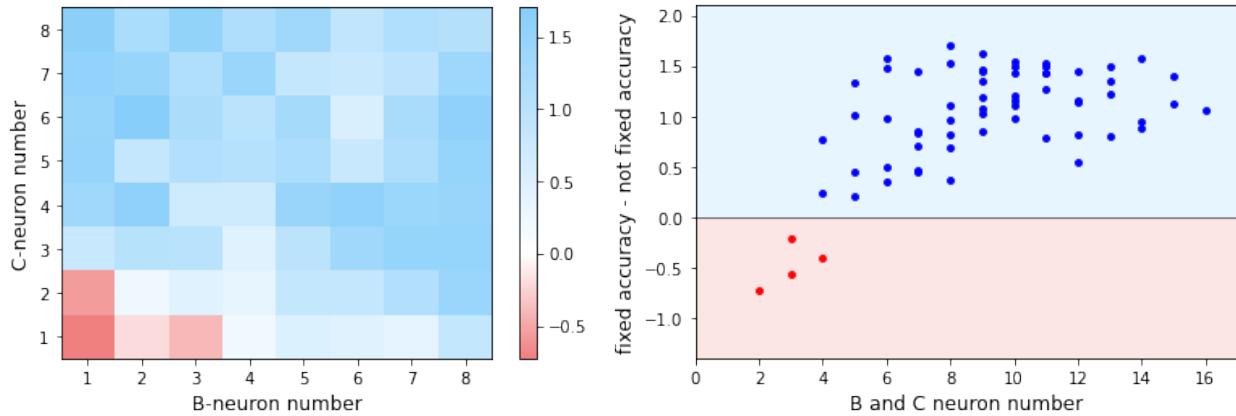
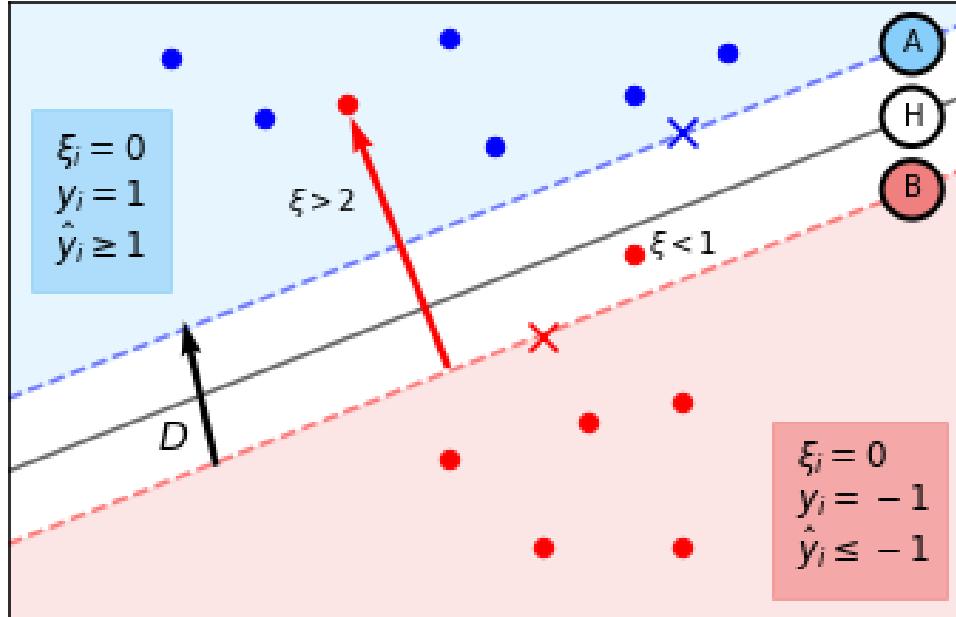


Figure 5.17: Contour plot of the difference between the accuracy results with and without fixing in terms of the B and C type neurons. Plotting of the difference between the accuracy results with and without fixing in terms of the sum of the B and C-type neurons.

As we can see, there are a few cases where the accuracy does not improve when fixing the neurons. As shown in the plot, this happens for lower numbers of neurons. As the number of neurons increases, the improvement in the accuracy is more significant. This increasing effect is stabilized as the neurons grow too large in numbers.

## 5.4 SVM [Supervised]

### 5.4.1 Mathematical derivation



[34][53] The support vector machine algorithm is a classification algorithm that classifies the data into two classes, A (blue) and B (red). We assign the outcome values of 1 and -1 to each class, respectively. I will denote with  $y_i$  the actual outcomes and with  $\hat{y}_i$  the predicted values.

The objective of the algorithm is to find a line of the following form  $w_1X_1 + w_2X_2 + w_0 = 0$  which I labeled on the plot as **H**, as we can observe this is just another way to write the typical equation for a line given by  $y = ax + b$  where now  $y = X_2$ ,  $a = -w_1/w_2$  and  $b = -w_0/w_2$ .

Apart from this line, we have another two lines (one must keep in mind that these lines will be hyperplanes for more dimensions, but I will limit it to two dimensions for simplicity) given by the labels A and B on the plot, the first one (**A**) corresponds to  $w_1X_1 + w_2X_2 + w_0 = 1$  and the second one (**B**) to  $w_1X_1 + w_2X_2 + w_0 = -1$ . These lines serve the purpose of classifying the space into the A and B classes.

The idea is that the points above the blue dotted line are represented by  $w_1X_1 + w_2X_2 + w_0 \geq 1$  and the points below the red dotted line by  $w_1X_1 + w_2X_2 + w_0 \leq -1$ , so classifying the points is trivial. Suppose a point given by the coordinates  $(X_1^*, X_2^*)$ , we predict the outcome of the point with the equation  $\hat{y}^* = w_1X_1^* + w_2X_2^* + w_0$ . If this value is smaller than minus one, we assign it the outcome  $\hat{y}^* = -1$ , and if it is greater

than one, we assign it the outcome  $\hat{y}^* = 1$ . The point is unclassified or unbound if it lies between the red and blue lines, called the margin.

The points closer to the hyperplanes A and B, represented by a cross on my plot, are called support vector points. Many lines can divide the space between the two classes; the intuition we follow to pick the  $\mathbf{w}$  parameters that define the three lines is to choose them so that the space between A and B is maximal, and hence the boundary between the classes is wider.

Given a set of parallel lines  $ax + by + c_1 = 0$  and  $ax + by + c_2 = 0$ , the distance between them is given by:

$$D = \frac{|c_2 - c_1|}{\sqrt{a^2 + b^2}} \quad (5.28)$$

From this, we find that the distance between A and B is:

$$D = \frac{2}{\sqrt{w_1^2 + w_2^2}} = \frac{2}{\sqrt{\mathbf{w}^2}} \quad (5.29)$$

As we can see, the distance increases when the size of  $\mathbf{w}$  decreases; suppose that all the points can be divided by a line (which is not the case in the example I set). The exercise would be straightforward; we would only have to minimize the following quantity:

$$\mathbb{L}_H = \frac{1}{2} \mathbf{w}^2 \quad (5.30)$$

With a constraint for all the elements given by the inequality:

$$y_i \hat{y}_i = y_i (\mathbf{w} \mathbf{X}_i + w_0) \geq 1 \quad (5.31)$$

This constraint implies that all the points are correctly classified, i.e., either  $\hat{y}_i \geq 1$  and  $y_i = 1$  or  $\hat{y}_i \leq -1$  and  $y_i = -1$ . This case is known as the **hard margin SVM**, which will be very unlikely applicable because, most of the time, the classes are not separable by a line.

Whenever this is not the case, we consider the **soft margin SVM**, which adds a new set of parameters  $\xi_i$  representing each point's errors. The problem is similar to the previous one; we now have to minimize the following quantity:

$$\mathbb{L}_S = \frac{1}{2} \mathbf{w}^2 + C \sum_i^N \xi_i \quad (5.32)$$

The parameter C helps us control the importance of the errors  $\xi_i$ . If we set C to be minor, we allow the  $\xi_i$  to be nonzero, making the algorithm prioritize minimizing the size of  $\mathbf{w}$  and hence setting the A and B hyperplanes far apart. This will mean some points being wrongly classified. On the other hand, if we set C to be significant, the algorithm will

prioritize reducing the size of the errors  $\xi_i$ , and hence, A and B will be close since this minimizes the errors. Setting C high results in overfitting. The constraint now is:

$$y_i \hat{y}_i = y_i (\mathbf{w} \mathbf{X}_i + w_0) \geq 1 - \xi_i \quad (5.33)$$

We want both to maximize the distance between the hyperplanes A and B and minimize the errors on the classifications; all this is subject to the constraint on the points, which translates to these points being incorrectly classified only by an amount up to their respective errors.

The approach to solve this is using Lagrangian multipliers, given a function  $f(x)$  subject to a constraint  $g(x) \geq 0$ , the minimal of the function  $f(x)$  subject to the constraint is obtained by minimizing the Lagrangian:  $\mathbb{L} = f(x) - \lambda g(x)$ . This is because in order for the boundary  $g(x)=0$  to be optimal, we require both gradients to point in the same direction, i.e.,  $\nabla f(x) = \lambda \nabla g(x)$ . If we wanted to maximize the Lagrangian, we would require the gradients to point in opposite directions, and hence, we would have to maximize  $\mathbb{L} = f(x) + \lambda g(x)$  instead. This Lagrangian will now be a function of  $x$  and  $\lambda$ , so we have to compute the partial derivatives with respect to both and set them equal to zero. This turns our problem into the following, where the p stands for *primal*:

$$\begin{aligned} \mathbb{L}_p &= \frac{1}{2} \mathbf{w}^2 + C \sum_i^N \xi_i - \sum_i^N \alpha_i [y_i (\mathbf{w} \mathbf{X}_i + w_0) - 1 + \xi_i] = \\ &= \frac{1}{2} \mathbf{w}^2 - \sum_i^N \alpha_i y_i \mathbf{w} \mathbf{X}_i - \sum_i^N \alpha_i y_i w_0 + \sum_i^N \alpha_i + \sum_i^N (C - \alpha_i) \xi_i \end{aligned} \quad (5.34)$$

[6] One can minimize this primal Lagrangian with respect to  $\mathbf{w}$ ,  $w_0$  and  $\alpha_i$  or consider the dual problem. For the dual of the Lagrangian to be plausible, we need the Lagrange multipliers  $\alpha_i$  to be positive. To write the dual of the Lagrangian, we need to obtain the primal variables  $\mathbf{w}$  and  $w_0$  that minimize the primal Lagrangian and then substitute the values on the expression. This will result in a Lagrangian only dependent on  $\alpha_i$  with a series of constraints deduced from the minimization over  $\mathbf{w}$  and  $w_0$ . One must remember that the primal Lagrangian will be an upper bound of the dual Lagrangian, so minimizing the primal Lagrangian is equivalent to maximizing the dual. So, the last step will be to maximize the dual with respect to  $\alpha_i$ .

$$\frac{\partial \mathbb{L}_p}{\partial \mathbf{w}} = \mathbf{w} - \sum_i^N \alpha_i y_i \mathbf{X}_i = 0 \quad (5.35)$$

$$\frac{\partial \mathbb{L}_p}{\partial w_0} = \sum_i^N \alpha_i y_i = 0 \quad (5.36)$$

Now we have to substitute these results on  $\mathbb{L}_p$  to obtain the dual form of the Lagrangian:

$$\begin{aligned}\mathbb{L}_d &= \frac{1}{2} \left( \sum_i^N \alpha_i y_i \mathbf{X}_i \right) \left( \sum_j^N \alpha_j y_j \mathbf{X}_j \right) - \sum_i^N \alpha_i y_i \left( \sum_j^N \alpha_j y_j \mathbf{X}_j \right) \mathbf{X}_i + \sum_i^N \alpha_i + \sum_i^N (C - \alpha_i) \xi_i = \\ &= -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K_{i,j} + \sum_i^N \alpha_i + \sum_i^N (C - \alpha_i) \xi_i\end{aligned}\tag{5.37}$$

Here, I wrote  $K_{i,j} = \mathbf{X}_i \cdot \mathbf{X}_j$ ; the reason for this is that later on, I will introduce different kernels (types of scalar products) and I want the results to be generalized. This concept will make more sense later on. If we pay close attention to the last term of the dual Lagrangian, we can conclude that it is a constraint. It has the appropriate positive sign, and hence, we can take  $C - \alpha_i \geq 0$ . The Lagrangian takes its final form:

$$\mathbb{L}_d = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K_{i,j} + \sum_i^N \alpha_i\tag{5.38}$$

Now, we have two constraints, one coming from the minimization of the primal Lagrangian with respect to  $w_0$  and the other one from the structure of the dual Lagrangian.

$$\sum_i^N \alpha_i y_i = 0 \quad 0 \leq \alpha_i \leq C\tag{5.39}$$

[31] Following the great insight on inner products by R. Berwick, let us pay attention to a couple of things. The first term of the dual Lagrangian has a sum of elements that go as  $-\alpha_i \alpha_j y_i y_j K_{i,j}$ , if the angle between two different vectors  $\mathbf{X}_i, \mathbf{X}_j$  is less than  $90^\circ$ , this term will be greater than zero whenever  $y_i y_j = -1$  and smaller than zero whenever  $y_i y_j = 1$ , this means that the algorithm will prioritize products among points with opposite classes. As we can see on the plot, if we have, for example, a point of class B among the points of class A (i.e. a misplaced point) such as  $\mathbf{X}_1$ , this point will have  $\alpha_i = C$ , this is because this element will have a similar position vector to points of the opposite class such as  $\mathbf{X}_2$  and  $\mathbf{X}_3$  and hence its terms will contribute significantly to increasing the dual Lagrangian. Support vectors such as  $\mathbf{X}_4$  will also have high  $\alpha$  parameters, although not as high as the misplaced points. This is because the support vectors are the closest to the data points of the opposite class. Lastly, the rest of the points will most likely have  $\alpha$  values close to zero unless there is a misplaced point of the opposite class close to them. This is because, generally, these points will mainly contribute negatively to the Lagrangian due to having points of the same class close to them.

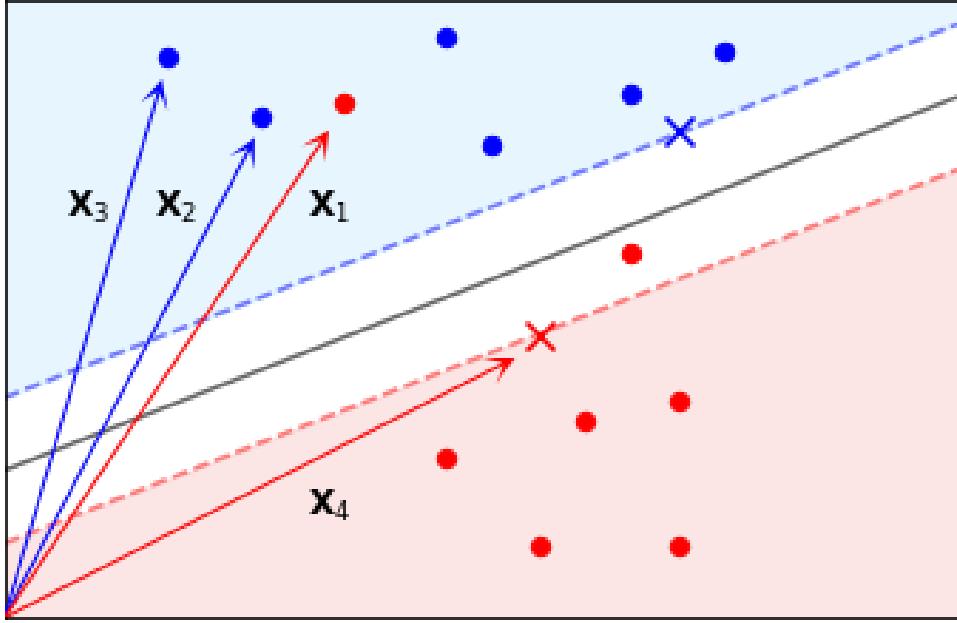


Figure 5.18: Illustration of the inner product relevance among data elements.

All these results can be classified as the **KKT conditions**, which are given by the following:

$$\alpha_i = 0 \quad y_i \hat{y}_i \geq 1 \quad \text{correctly classified} \quad (5.40)$$

$$0 \leq \alpha_i \leq C \quad y_i \hat{y}_i = 1 \quad \text{support vectors} \quad (5.41)$$

$$\alpha_i = C \quad y_i \hat{y}_i \leq 1 \quad \text{incorrectly classified} \quad (5.42)$$

With the expression for the dual Lagrangian, the constraints, and the KKT conditions, we can proceed to solve the maximization problem. We will approach this with the Sequential Minimal Optimization algorithm (SMO).

## SMO

[12][40][46][54] The idea behind this algorithm is to initialize  $w_0 = 0$  and  $\alpha_i = 0$  and update the  $\alpha_i$  on pairs (remember that using Equation 5.35, we can obtain  $\mathbf{w}$  in terms of the  $\alpha_i$  parameters), for this we consider a pair  $(\alpha_i, \alpha_j)$ . The first thing to take into account is that due to the constraint  $\sum_i \alpha_i y_i = 0$ , if we were to change the values of the  $\alpha$  pair, we require that:  $\Delta \alpha_i y_i = -\Delta \alpha_j y_j$  and since  $y_i$  takes either the value of 1 or -1 we can write:  $\Delta \alpha_i = -y_i y_j \Delta \alpha_j$  which simply takes the form:

$$\alpha_i^o + s \alpha_j^o = \alpha_i^f + s \alpha_j^f = \lambda \quad \text{with} \quad s = y_i y_j \quad \text{and} \quad \Delta \alpha_i = \alpha_i^f - \alpha_i^o \quad (5.43)$$

Just for clarification purposes I will use the following notation to refer to the status of the variables pre-update:  $w_0^o, \mathbf{w}^o, \alpha_i^o, \alpha_j^o, \hat{y}_i^o, \hat{y}_j^o$  and the following variables will be the variables post-update  $w_0^f, \mathbf{w}^f, \alpha_i^f, \alpha_j^f, \hat{y}_i^f, \hat{y}_j^f$ . Now we work on the Lagrangian a bit:

$$\begin{aligned} \mathbb{L}_d = & -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i^f \alpha_j^f y_i y_j K_{i,j} + \sum_i^N \alpha_i^f = \alpha_i^f + \alpha_j^f - \frac{1}{2} \alpha_i^f \alpha_i^f y_i^2 K_{i,i} - \frac{1}{2} \alpha_j^f \alpha_j^f y_j^2 K_{j,j} \\ & - \alpha_i^f \alpha_j^f y_i y_j K_{i,j} - \alpha_i^f y_i v_i - \alpha_j^f y_j v_j + O(\alpha_{k \neq i,j}^f) = (\lambda - s \alpha_j^f) + \alpha_j^f \\ & - \frac{1}{2} (\lambda - s \alpha_j^f)^2 K_{i,i} - \frac{1}{2} (\alpha_j^f)^2 K_{j,j} - s \alpha_j^f (\lambda - s \alpha_j^f) K_{i,j} - (\lambda - s \alpha_j^f) y_i v_i - \alpha_j^f y_j v_j \quad (5.44) \end{aligned}$$

With the following definitions:

$$v_m = \sum_{k \neq i,j}^N \alpha_k^f y_k K_{k,m} \quad (5.45)$$

Doing some work on this quantity knowing that  $\alpha_k^o = \alpha_k^f$  for  $k \neq i, j$  since only the  $i^{th}$  and  $j^{th}$  terms will be updated on this step, using Equation 5.35 and  $\hat{y} = \mathbf{w} \mathbf{X} + w_0$ :

$$\begin{aligned} v_m = & \sum_{k \neq i,j}^N \alpha_k^o y_k K_{k,m} = (\mathbf{w}^o - \alpha_i^o y_i \mathbf{X}_i - \alpha_j^o y_j \mathbf{X}_j) \mathbf{X}_m = \\ & \mathbf{w}^o \mathbf{X}_m - \alpha_i^o y_i K_{i,m} - \alpha_j^o y_j K_{j,m} = \hat{y}_m^o - w_0^o - \alpha_i^o y_i K_{i,m} - \alpha_j^o y_j K_{j,m} \end{aligned} \quad (5.46)$$

Once we have these results, we can compute the derivative of the dual Lagrangian respect to  $\alpha_j^f$  to obtain which value maximizes it so we can update it to that value.

$$\frac{\partial \mathbb{L}_d}{\partial \alpha_j^f} = -s + 1 + s(\lambda - s \alpha_j^f) K_{i,i} - \alpha_j^f K_{j,j} - s \lambda K_{i,j} + 2 \alpha_j^f K_{i,j} + s y_i v_i - y_j v_j \quad (5.47)$$

Here, I used the fact that  $s^2 = 1$  and in the next step, I also use that  $s y_i = y_i y_i y_j = y_j$  and hence:

$$\frac{\partial \mathbb{L}_d}{\partial \alpha_j^f} = -\alpha_j^f (K_{i,i} + K_{j,j} - 2K_{i,j}) + s(K_{i,i} - K_{i,j}) \lambda + y_j(v_i - v_j) + 1 - s \quad (5.48)$$

Before making this partial derivative equal to zero, we have to ensure that the result that we will find will be a maximum value and not a minimum value. We compute the second derivative, looking for a value smaller than zero.

$$\frac{\partial^2 \mathbb{L}_d}{\partial \alpha_j^f} = -(K_{i,i} + K_{j,j} - 2K_{i,j}) \leq 0 \quad \text{for all } i, j \quad (5.49)$$

Now, making the derivative equal to zero, we find:

$$\begin{aligned} \alpha_j^f (K_{i,i} + K_{j,j} - 2K_{i,j}) &= s(K_{i,i} - K_{i,j}) \lambda + y_j(v_i - v_j) + 1 - s = s(K_{i,i} - K_{i,j})(\alpha_i^o + s \alpha_j^o) \\ &+ y_j(\hat{y}_i^o - w_0^o - \alpha_i^o y_i K_{i,i} - \alpha_j^o y_j K_{j,i} - \hat{y}_j^o + w_0^o + \alpha_i^o y_i K_{i,j} + \alpha_j^o y_j K_{j,j}) + y_j y_j - y_j y_i \\ &= y_j(\hat{y}_i^o - \hat{y}_j^o - y_i + y_j) + \alpha_j^o (K_{i,i} + K_{j,j} - 2K_{i,j}) \end{aligned} \quad (5.50)$$

Defining  $E_k = \hat{y}_k^o - y_k$  and  $\eta_{i,j} = K_{i,i} + K_{j,j} - 2K_{i,j}$  the result we get is:

$$\alpha_j^f = \alpha_j^o + \frac{y_j(E_i - E_j)}{\eta_{i,j}} \quad (5.51)$$

Now that we have how  $\alpha_j$  has to be updated to find the maximum Lagrangian, the update on  $\alpha_i$  is trivial since we know that it has to fulfill  $\alpha_i^f = \lambda - s\alpha_j^f$ . We must keep in mind that after applying these changes, we maximized the dual Lagrangian and ensured we did not break the constraint:  $\sum_i \alpha_i y_i = 0$ . However, we did not impose the second constraint:  $0 \leq \alpha_i \leq C$  and we will make sure of that now by clipping the value of  $\alpha_j$  to an interval.

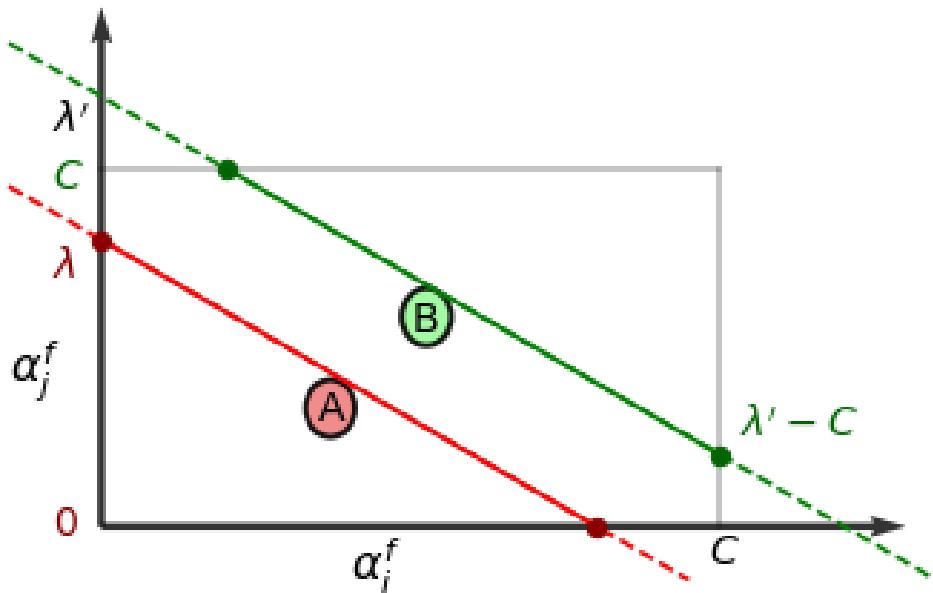


Figure 5.19: Clipping of  $\alpha_i$  and  $\alpha_j$  for  $y_i y_j = 1$  to ensure the constraint  $0 \leq \alpha_i \leq C$ .

In this first figure we consider the first case, where the chosen pair are of the same class, and hence  $\alpha_i^o + \alpha_j^o = \alpha_i^f + \alpha_j^f = \lambda$ . I plotted both possible cases, the first one labeled as A is for  $\lambda < C$  and the second case labeled as B for  $\lambda > C$ . The green and red lines correspond to the equation  $\alpha_i^f + \alpha_j^f = \lambda$  for each case.

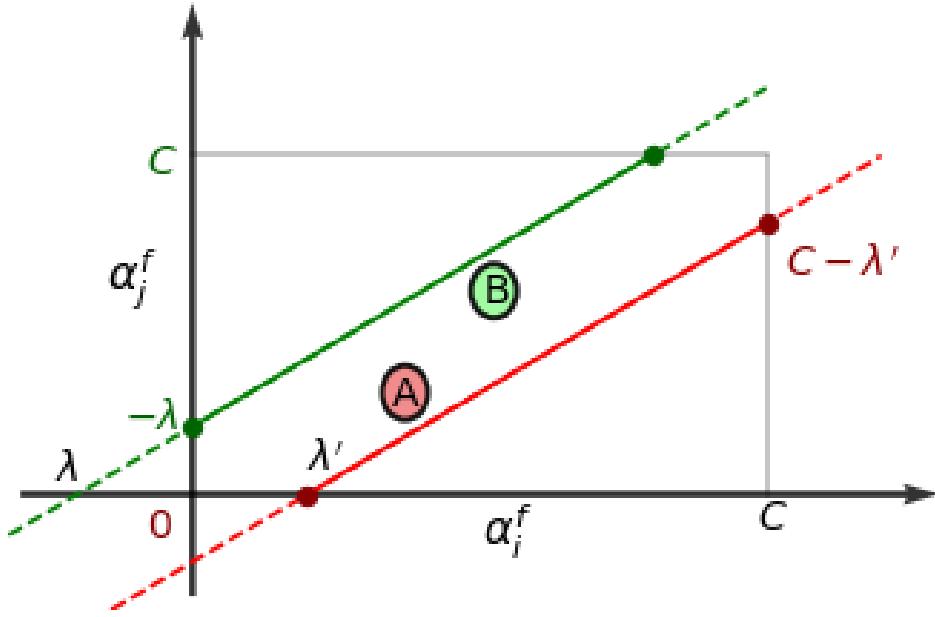


Figure 5.20: Clipping of  $\alpha_i$  and  $\alpha_j$  for  $y_i y_j = -1$  to ensure the constraint  $0 \leq \alpha_i \leq C$ .

The second figure considers the second case, where the chosen pair are of opposite classes. Hence,  $\alpha_i^o - \alpha_j^o = \alpha_i^f - \alpha_j^f = \lambda$ . I plotted once again both possible cases, the first one labeled as A is for  $\lambda > 0$  and the second one labeled as B for  $\lambda < 0$ . The green and red lines correspond to the equation  $\alpha_i^f - \alpha_j^f = \lambda$  for each case.

Once the allowed intervals are established, I will write the clipped version of  $\alpha_j^f$ , which I will denote by  $\alpha_j^c$ .

$$\alpha_j^c = \begin{cases} H & \text{if } \alpha_j^f \geq H \\ \alpha_j^f & \text{if } L < \alpha_j^f < H \\ L & \text{if } \alpha_j^f \leq L \end{cases} \quad (5.52)$$

H and L denote the high and low limits for  $\alpha_j^f$ , which we can deduce from the plot by looking at the minimum and maximum values of the green and red lines on the y-axis. Keep in mind that the dashed portion of the lines breaks the constraint  $0 \leq \alpha_i \leq C$ .

$$L = 0, \quad H = \lambda \quad \text{for } y_i y_j = 1, \quad \lambda < C \quad (5.53)$$

$$L = \lambda - C, \quad H = C \quad \text{for } y_i y_j = 1, \quad \lambda > C \quad (5.54)$$

$$L = -\lambda, \quad H = C \quad \text{for } y_i y_j = -1, \quad \lambda < 0 \quad (5.55)$$

$$L = 0, \quad H = C - \lambda \quad \text{for } y_i y_j = -1, \quad \lambda > 0 \quad (5.56)$$

The last thing there is left to do is to update  $w_0$ , which is given by  $w_0^f = (w_0^{(i)} + w_0^{(j)})/2$  with:

$$w_0^{(i)} = w_0^o - E_i - y_i(\alpha_i^c - \alpha_i^o)K_{i,i} - y_j(\alpha_j^c - \alpha_j^o)K_{i,j} \quad (5.57)$$

$$w_0^{(j)} = w_0^o - E_j - y_i(\alpha_i^c - \alpha_i^o)K_{i,j} - y_j(\alpha_j^c - \alpha_j^o)K_{j,j} \quad (5.58)$$

Where  $w_0^{(i)}$  is chosen so for the input  $\mathbf{X}_i$ , the output is  $y_i$ . The same logic is applied to obtain  $w_0^{(j)}$ . Then,  $w_0^f$  is chosen as the average of these two values. We will check this for one of the values:

$$\begin{aligned} \hat{y}_i^f &= \mathbf{w}^f \mathbf{X}_i + w_0^{(i)} = \alpha_i^c y_i K_{i,i} + \alpha_j^c y_j K_{i,j} + \sum_{k \neq i,j} \alpha_k^o y_k K_{i,k} + w_0^{(i)} = \\ &\alpha_i^c y_i K_{i,i} + \alpha_j^c y_j K_{i,j} + \sum_{k \neq i,j} \alpha_k^o y_k K_{i,k} + w_0^o + y_i - \hat{y}_i^o - y_i(\alpha_i^c - \alpha_i^o)K_{i,i} - y_j(\alpha_j^c - \alpha_j^o)K_{i,j} = \\ &y_i + (\sum_k \alpha_k^o y_k K_{i,k} + w_0^o) - \hat{y}_i^o = y_i + (\mathbf{w}^o \mathbf{X}_i + w_0^o) - \hat{y}_i^o = y_i \end{aligned} \quad (5.59)$$

Now, all that is left to do is to write the steps of the algorithm based on all the results we found:

## STEPS

- 1) Initialize  $w_0$  and  $\alpha_i$  to zero.
- 2) Do the following process until the KKT conditions are fulfilled to within a small value epsilon:
  - 2.a) We pick an  $\alpha_j$  that violates the KKT conditions (prioritizing unbound points).
  - 2.b) Then we pick an  $\alpha_i$  following some heuristics or priorities; if the change on the alphas is not sufficient, we pick a different point. The heuristics are:
    - We find the unbound point that maximizes  $|E_i - E_j|$ .
    - We iterate through the unbound points randomly.
    - We iterate through all the points randomly
  - 2.c) We update  $\alpha_i$  and  $\alpha_j$  on the same step.
  - 2.d) We compute the constant term  $w_0$ .

## KERNEL TYPES

[3][13][41] Let's imagine a data collection given by two variables,  $X_1$  and  $X_2$ . Suppose this dataset is not linearly separable; hence, the linear SVM we introduced cannot split the dataset. We can always extend the dimensions of the problem by introducing new variables, i.e., take our data to a higher dimensional space. We will see the example of taking the dimensions of our problem from 2D to 3D.

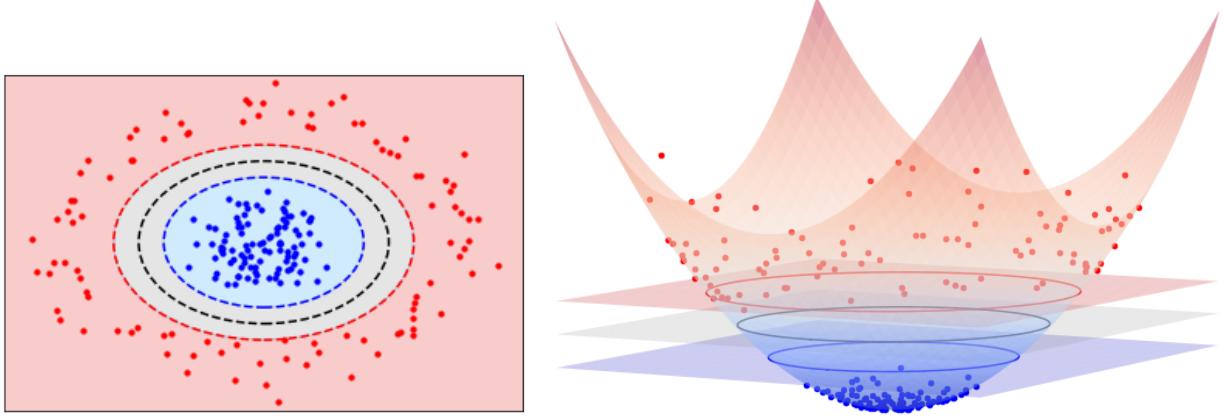


Figure 5.21: SVM model on a non-linearly separable 2D model extended to 3D.

As shown in the figure, the data is not linearly separable. We can introduce a new variable given by  $R = X_1^2 + X_2^2$  and end up with three variables  $(X_1, X_2, R)$ . As we can see, the data is now linearly separable by the equivalent of a line in 3D, a hyperplane plotted with the color grey. Then, we have another two hyperplanes, colored red and blue, which delimit the zones of the classes. Remember that now that we introduced a third dimension,  $w$  will also have a length of three. So the steps we have to follow are the following:

- We extend the dataset to higher dimensions.
- We use the SVM model to find the  $w$  parameters that define the three hyperplanes. This will require us to define the inner product  $K_{i,j}$  in this higher dimensional space.
- Once we have  $w$  we make predictions on the data, for the 3D case we just explored we would have  $\hat{y} = w_1 X_1 + w_2 X_2 + w_R R + w_0$ .
- Then we can plot the results in terms of the original variable pair  $(X_1, X_2)$  to obtain the plot on the left.

The only thing we have to do now is to define the inner products for these higher dimensional spaces. Remember that the only reason we computed the dual Lagrangian and generalized the inner product as  $K_{i,j}$  was to use the results for these different higher dimensional kernels. Let us see, as an example, the inner product of this 3D basis we built.

Consider that we have two different points given by:

$$\mathbf{X} = (X_1, X_2, \sqrt{X_1^2 + X_2^2}) \quad (5.60)$$

$$\mathbf{X}^* = (X_1^*, X_2^*, \sqrt{X_1^{*2} + X_2^{*2}}) \quad (5.61)$$

The inner product between  $\mathbf{X}$  and  $\mathbf{X}^*$  will be given by the following:

$$K = X_1 X_1^* + X_2 X_2^* + \sqrt{X_1^2 + X_2^2} \sqrt{X_1^{*2} + X_2^{*2}} \quad (5.62)$$

The reason why I chose this basis was because it led to an excellent plot to understand the reasoning behind the kernels. Now we will consider the following 3D basis instead:

$$\mathbf{X} = (X_1^2, X_2^2, \sqrt{2} X_1 X_2) \quad (5.63)$$

Now, let's see the inner product between  $\mathbf{X}$  and  $\mathbf{X}^*$ :

$$K = X_1^2 X_1^{*2} + X_2^2 X_2^{*2} + 2 X_1 X_2 X_1^* X_2^* = (X_1 X_1^* + X_2 X_2^*)^2 = (\mathbf{X} \cdot \mathbf{X}^*)^2 \quad (5.64)$$

As we can see for this particular choice of basis, the inner product is the square of the regular scalar product. Let's see as an example now the inner product of a 4D basis given by  $\mathbf{X} = (X_1^3, X_2^3, \sqrt{3} X_1^2 X_2, \sqrt{3} X_1 X_2^2)$ .

$$K = X_1^3 X_1^{*3} + X_2^3 X_2^{*3} + 3 X_1^2 X_2 X_1^* X_2^2 + 3 X_1 X_2^2 X_1^* X_2^{*2} = \\ (X_1 X_1^* + X_2 X_2^*)^3 = (\mathbf{X} \cdot \mathbf{X}^*)^3 \quad (5.65)$$

Now we can start to see the advantages of these particular basis choices. In general, for an inner product given by:

$$K = (\mathbf{X} \cdot \mathbf{X}^*)^D \quad (5.66)$$

We will have a basis given by the following elements:

$$X_{basis} = \left\{ X_1^i X_2^{D-i} \sqrt{\frac{D!}{i! (D-i)!}} \right\}_{i=0,1,\dots,D} \quad (5.67)$$

We need to introduce one more type of kernel, the RBF kernel, which stands for Radial Basis Function. We will start by defining this kernel's scalar product and then work out the basis. This kernel is based on the Gaussian function, and hence, its scalar product will be an exponential of the form:

$$K = e^{-\gamma(\mathbf{X}-\mathbf{X}^*)^2} \quad (5.68)$$

Here, we introduce a new parameter  $\gamma$ , which will add flexibility to the reach of the data points. Now, we have to find a basis for the inner product to be written in terms of the regular scalar product. To do this, we will start by working out the term a bit:

$$K = e^{-\gamma(\mathbf{X}-\mathbf{X}^*)^2} = e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2} - 2\mathbf{X}\mathbf{X}^*)} = e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2})} e^{2\gamma \mathbf{X}\mathbf{X}^*} \quad (5.69)$$

The next step is doing a Taylor expansion on the exponential:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^N}{N!} \quad (5.70)$$

Expanding the exponential of Equation 5.69, we can keep on working on the terms:

$$\begin{aligned} K &= e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2})} \left[ 1 + 2\gamma \mathbf{X} \mathbf{X}^* + \frac{(2\gamma \mathbf{X} \mathbf{X}^*)^2}{2!} + \dots + \frac{(2\gamma \mathbf{X} \mathbf{X}^*)^N}{N!} \right] = \\ &e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2})} \sum_{D=0}^N \frac{(2\gamma \mathbf{X} \cdot \mathbf{X}^*)^D}{D!} = e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2})} \sum_{D=0}^N \frac{(\sqrt{2\gamma} \mathbf{X} \cdot \sqrt{2\gamma} \mathbf{X}^*)^D}{D!} \end{aligned} \quad (5.71)$$

We know how to write the power of that inner product since we know that it is equal to the regular scalar product of an extended basis, so we can write it using the basis of Equation 5.67.

$$\begin{aligned} K &= e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2})} \sum_{D=0}^N \sum_{i=0}^D \frac{1}{D!} \left[ (\sqrt{2\gamma} X_1)^i (\sqrt{2\gamma} X_2)^{D-i} \sqrt{\frac{D!}{i! (D-i)!}} \right] \\ &\quad \left[ (\sqrt{2\gamma} X_1^*)^i (\sqrt{2\gamma} X_2^*)^{D-i} \sqrt{\frac{D!}{i! (D-i)!}} \right] = \\ &e^{-\gamma(\mathbf{X}^2 + \mathbf{X}^{*2})} \sum_{D=0}^N \sum_{i=0}^D (\sqrt{2\gamma} X_1)^i (\sqrt{2\gamma} X_2)^{D-i} (\sqrt{2\gamma} X_1^*)^i (\sqrt{2\gamma} X_2^*)^{D-i} \frac{1!}{i! (D-i)!} \end{aligned} \quad (5.72)$$

Moreover, now it is trivial to write our basis like:

$$X_{basis} = \left\{ e^{-\gamma \mathbf{X}^2} (\sqrt{2\gamma} X_1)^i (\sqrt{2\gamma} X_2)^{D-i} \sqrt{\frac{1}{i! (D-i)!}} \right\}_{\substack{i=0,1..D \\ D=0,1,...N}} \quad (5.73)$$

The basis for the RBF kernel is an infinite basis composed of different powers of  $X_1$  and  $X_2$ . We can construct the basis using Equation 5.73 with an integer  $N$  large enough so that the Taylor expansion is a good approximation. Then, the scalar product of Equation 5.68 will be equal to the regular scalar product of this new *infinite* extended basis.

The  $\gamma$  parameter influences the reach of the different data points. As we increase this value, the reach of the data points will get lower, i.e., the data points will only care about the points close to them, leading to an overfit model.

## 5.4.2 Algorithm implementation

We implement the SVM model with the Radial Basis function as the kernel.

```
[1]: import random
import numpy as np
import math as math
import copy as copy
import warnings

class svm():
```

The SVM model takes the following variables as input. First, we have the training data given by a pair of variables  $\{X, Y\}$  and an outcome  $Z$ . Since we will apply this model to datasets containing more than two classes and whose outcomes are likely not to be minus one or one, the *index-change one* and *index-change two* variables are given by one-dimensional arrays of two elements which indicate the initial and final value of the class A outcome and the class B outcome. It filters two class types and changes their outcomes accordingly. Then, we have the parameters  $C$  and  $\gamma$ , explained in the previous section. Last, I defined the number of iterations and the  $\epsilon$  parameter, allowing a slight deviation on the  $\alpha$  parameters. I will also allow a tolerance error of 0.001 on the KKT conditions.

```
def __init__(self,X,Y,Z,index_change_1,
            index_change_2,C,gamma = 1.0):

    self.C = C
    self.gamma = gamma
    self.index_change_1 = index_change_1
    self.index_change_2 = index_change_2

    self.X,self.Y,self.Z = self.adjust_data(X,Y,Z)

    self.iterations = 20000
    self.epsilon = C/1000.0
    self.tol = 0.001

    self.alpha = np.zeros((len(self.X),1))
    self.w_0 = 0
    self.basis = self.basis_expand(self.X,self.Y)
    self.K = np.array([[self.product(self.X,self.Y,i,j) \
                      for i in range(len(self.X))]\ \
                      for j in range(len(self.X))])
```

This function takes as input the variables  $X$ ,  $Y$ , and  $Z$ . It keeps only the data points whose outcomes are equal to those indicated by the variables *index-change one* and *index-change two* and then changes the outcomes to either minus one or one.

```
def adjust_data(self,X,Y,Z):

    X = np.array([X[i] for i in range(len(X)) if \
                  (Z[i] == self.index_change_1[0] or \
                   Z[i] == self.index_change_2[0]))]
    Y = np.array([Y[i] for i in range(len(Y)) if \
                  (Z[i] == self.index_change_1[0] or \
                   Z[i] == self.index_change_2[0]))]
    Z = np.array([Z[i] for i in range(len(Z)) if \
                  (Z[i] == self.index_change_1[0] or \
                   Z[i] == self.index_change_2[0]))]

    for i in range(len(Z)):
        if (Z[i] == self.index_change_1[0]):
            Z[i] = self.index_change_1[1]
        elif (Z[i] == self.index_change_2[0]):
            Z[i] = self.index_change_2[1]

    return(np.array(X),np.array(Y),np.array(Z))
```

The *product* function of the algorithm computes the scalar product between two points given by the exponential of Equation 5.68.

```
def product(self,X,Y,i,j):

    R = (X[i]-X[j])**2.0 + (Y[i]-Y[j])**2.0
    product = math.exp(-self.gamma*R)

    return(product)
```

The following function expands the data to a higher dimensional basis, following Equation 5.73. We append each iteration a row containing a basis element for all data to the empty array *basis*. Although theoretically, the basis is meant to be infinite, I took the number of elements of the expansion equal to 170 to avoid divergency problems with

the factorials.

```
def basis_expand(self, X, Y):  
  
    basis = []  
  
    X = X*math.sqrt(2*self.gamma)  
    Y = Y*math.sqrt(2*self.gamma)  
  
    for D in range(170):  
  
        for i in range(D+1):  
  
            exponential = np.exp(-(X**2.0 + Y**2.0)/2)  
            powers = (X**i)*(Y**(D-i))  
            factorial = math.sqrt(math.factorial(i)*\  
                                  math.factorial(D-i))  
  
            basis_element = exponential*powers/factorial  
            basis.append(basis_element)  
    basis = np.array(basis)  
  
    return(basis)
```

The *KKT-check* function starts initializing the variable *KKT-value*, which stores the product of the prediction and the outcome, i.e.,  $\hat{y}_i y_i$  for all the training data. Then, it creates a random shuffle of all the training indexes and starts iterating over the shuffled values. It checks for the three possibilities of the KKT conditions. As soon as it finds one data point that does not fulfill the KKT conditions, it saves this index on the variable *j-value* and also returns the outcome *no* on a variable called *KKT-valid*. If all the data points fulfill the KKT conditions, the function returns *KKT-valid = yes*.

```
def KKT_check(self, w):  
  
    KKT_value = (np.dot(np.transpose(self.basis), w) + self.w_0)*self.Z  
    index = np.arange(0, len(self.alpha), 1)  
    random.shuffle(index)  
    j_value = 0  
  
    for i in index:  
  
        if (self.alpha[i] < self.epsilon):
```

```

KKT_value[i] = 1 - KKT_value[i]

if (KKT_value[i] > 0):

    j_value = i

    return("no",j_value)

elif (0 < self.alpha[i] and self.alpha[i] < self.C):

    KKT_value[i] = abs(1 - KKT_value[i]) - self.tol

    if (KKT_value[i] > 0):

        j_value = i

        return("no",j_value)

    elif (abs(self.C-self.alpha[i]) < self.epsilon):

        KKT_value[i] = KKT_value[i] - 1

        if (KKT_value[i] > 0):

            j_value = i

            return("no",j_value)

    return("yes",j_value)

```

This is the algorithm's core; first, it starts running the KKT- condition check. If the function returns a *yes*, the loop is terminated. Once the  $j$ -value has been picked, the code will find the unbound indexes, i.e., those for which  $0 \leq \alpha \leq C$ . The next step is to find the index of the second alpha parameter  $\alpha_i$ . To do this, we will use three heuristics. Each heuristic will be run if the previous one does not find the  $i$  index. If none of the heuristics is successful, we will jump to the next iteration without changing the parameters. After finding a  $(i, j)$  index pair, it updates the  $\alpha$  parameters and then computes the value of  $w_0$ .

```

def SMO(self):

    warnings.filterwarnings('ignore')

    for iteration in range(self.iterations):

        #Computing w

        w = np.dot(self.basis, self.alpha*self.Z)

        #KKT check + j_value choice

        KKT_valid, j_value = self.KKT_check(w)[:]

        if KKT_valid == "yes":
            break

        unbound_index = np.array([i for i in range(len(self.alpha))\
                                if (self.alpha[i] > 0 and self.alpha[i] < self.C)])

        # FIRST HEURISTIC FOR i_value
        i_value, validity = self.first_heuristic(w,
                                                unbound_index,j_value)

        # SECOND HEURISTIC FOR i_value
        if (validity != "true"):

            i_value, validity = self.second_heuristic(w,
                                                unbound_index,j_value)

        # THIRD HEURISTIC FOR i_value
        if (validity != "true"):

            i_value, validity = self.third_heuristic(w,j_value)

        if (validity != "true"):

            continue

        #alpha update
        Ei_Ej,Delta_alpha_j,lambda_ = self.Ei_Ej_difference(w,
                                                j_value,i_value)
        Delta_alpha_j, lambda_ = self.clip(Ei_Ej,

```

```

        Delta_alpha_j, lamda, j_value, i_value)

alpha_j_pre = self.alpha[j_value]
alpha_i_pre = self.alpha[i_value]
self.alpha[j_value] = self.alpha[j_value] + Delta_alpha_j
self.alpha[i_value] = lamda - self.Z[j_value] \
    *self.Z[i_value]*self.alpha[j_value]

#W_0 computation
self.w_0 = self.w0_computation(w , alpha_j_pre ,
                                alpha_i_pre ,j_value ,i_value )
#last update of w
w = np.dot(self.basis, self.alpha*self.Z)

return(self.w_0, w, self.alpha)

```

The first heuristic will find the unbound index  $i$  that maximizes the quantity  $|E_i - E_j|$  and check if, after clipping the values, there is a change on the  $\alpha$  parameters. If this is not the case, we run the second heuristic.

```

def first_heuristic(self,w,unbound_index,j_value):

    if (len(unbound_index)==0):

        return(0,"false")

    #COMPUTE MAX VALUE OF  $E_i - E_j$ 
    Ei_Ej_values = np.array([self.Ei_Ej_difference(w,
                                                    j_value,i_value)[0] for i_value in unbound_index])
    Ei_Ej_values = np.nan_to_num(Ei_Ej_values)

    Ei_Ej_max = np.max(np.abs(Ei_Ej_values))
    loc = np.where(np.abs(Ei_Ej_values) == Ei_Ej_max)[0][0]
    i_value = unbound_index[loc]

    # CHECK IF ALPHA CHANGES AFTER CLIPPING
    Ei_Ej,Delta_alpha_j, lamda = self.Ei_Ej_difference(w,
                                                       j_value,i_value)
    Delta_alpha_j, lamda = self.clip(Ei_Ej,
                                    Delta_alpha_j, lamda,j_value,i_value)

    if ( abs(Delta_alpha_j) > 0 ):

        validity = "true"

```

```

    else:

        validity = "false"

    return(i_value,validity)

```

For the second heuristic, we will loop through all the unbound indexes randomly and check if any leads to a nonzero alpha change.

```

def second_heuristic(self,w,unbound_index,j_value):

    i_value = 0
    validity = "false"

    if len(unbound_index) > 0 :

        random.shuffle(unbound_index)

        for i_value in unbound_index:

            Ei_Ej,Delta_alpha_j,lambda = self.Ei_Ej_difference(w,
                                                               j_value,i_value)
            Delta_alpha_j, lambda = self.clip(Ei_Ej,
                                              Delta_alpha_j,lambda,j_value,i_value)

            if ( abs(Delta_alpha_j) > 0 ):

                validity = "true"
                return(i_value,validity)

    return(i_value,validity)

```

Last, we will loop through all the indexes randomly and check if any lead to a nonzero alpha change.

```

def third_heuristic(self,w,j_value):

    i_value = 0
    validity = "false"

    index = np.arange(0,len(self.alpha))
    random.shuffle(index)

```

```

for i_value in index:

    Ei_Ej,Delta_alpha_j,lambda = self.Ei_Ej_difference(w,
                                                       j_value,i_value)
    Delta_alpha_j, lambda = self.clip(Ei_Ej,
                                       Delta_alpha_j,lambda,j_value,i_value)

    if ( abs(Delta_alpha_j) > 0 ):

        validity = "true"
        return(i_value,validity)

return(i_value,validity)

```

This function is straightforward; it computes the difference  $E_i - E_j$  and  $\Delta\alpha_j$ .

```

def Ei_Ej_difference(self,w,j,i):

    alpha = self.alpha
    w_0 = self.w_0
    basis = self.basis
    K = self.K

    alpha_i_pre = alpha[i]
    alpha_j_pre = alpha[j]

    eta_ij = K[i][i] + K[j][j] - 2*K[i][j]
    Ei_Ej = self.Z[j]-self.Z[i] + basis[:,i].dot(w)-basis[:,j].dot(w)

    lambda = alpha_i_pre + self.Z[j]*self.Z[i]*alpha_j_pre
    Delta_alpha_j = self.Z[j]*Ei_Ej /eta_ij

    return(Ei_Ej,Delta_alpha_j,lambda)

```

The clip function ensures that after changing the alpha parameters, none lay outside the range  $0 \leq \alpha \leq C$ . For this, we use the bounds of equations 5.52-5.56.

```

def clip(self,Ei_Ej,Delta_alpha_j,lambda,j,i):

    L = 0
    H = 0

    alpha = self.alpha
    w_0 = self.w_0

```

```

basis = self.basis
K = self.K

alpha_j_pre = alpha[j]

alpha_j = alpha_j_pre + Delta_alpha_j

# UPPER AND LOWER BOUNDS L,H

if (self.Z[j]*self.Z[i] == 1 and lamda <= self.C):

    L = 0
    H = lamda

elif (self.Z[j]*self.Z[i] == 1 and lamda >= self.C):

    L = lamda-self.C
    H = self.C

elif (self.Z[j]*self.Z[i] == -1 and lamda <= 0.0):

    L = -lamda
    H = self.C

elif (self.Z[j]*self.Z[i] == -1 and lamda >= 0.0):

    L = 0
    H = self.C -lamda

# ALPHA CLIPPING

if alpha_j >= H:

    alpha_j = H

elif (L < alpha_j and alpha_j < H):

    alpha_j = alpha_j

elif alpha_j <= L:

    alpha_j = L

return(alpha_j-alpha_j_pre, lamda)

```

Last, we have the function that computes  $w_0$  by averaging the results of Equations 5.57 and 5.58, and then the *predict* and *accuracy* functions, which are similar to those of other classification algorithms with the exception that we need to extend the testing data to a higher dimensional space.

```

def w0_computation(self,w,alpha_j_pre,alpha_i_pre,j,i):

    alpha = self.alpha
    w_0 = self.w_0
    basis = self.basis
    K = self.K

    alpha_j=alpha[j]
    alpha_i=alpha[i]

    w_0_i=w_0 -(basis[:,i].dot(w)+w_0-self.Z[i])\
    -self.Z[i]*(alpha_i-alpha_i_pre)*K[i,i]\ 
    -self.Z[j]*(alpha_j-alpha_j_pre)*K[i,j]

    w_0_j=w_0 -(basis[:,j].dot(w)+w_0-self.Z[j])\
    -self.Z[i]*(alpha_i-alpha_i_pre)*K[i,j]\ 
    -self.Z[j]*(alpha_j-alpha_j_pre)*K[j,j]

    w_0=(w_0_i + w_0_j)/2

    return(w_0)

def predict(self,X_test,Y_test):

    w_0, w, alpha = self.SMO()
    test_basis = self.basis_expand(X_test,Y_test)
    Z_pred = np.dot(np.transpose(test_basis),w)+w_0

    return(Z_pred)

def accuracy(self,X_test,Y_test,Z_test):

    X_test,Y_test,Z_test = self.adjust_data(X_test,Y_test,Z_test)
    Z_pred = self.predict(X_test,Y_test)
    right_predictions = np.count_nonzero(Z_test*Z_pred >= 0)

    return(100*right_predictions/len(Z_test))

```

Next, we will test the model on the Iris data set; first, I built a function that plotted all the results for the RBF kernel for a given value of the  $C$  and  $\gamma$  parameters. The graph is composed of four different plots, the first three containing the results of all possible class combinations (*setosa-versicolor*, *setosa-virginica* and *versicolor-virginica*) since as we know the SVM model works for outcome pairs. Furthermore, a fourth plot that combines all the results in a manner that will be explained next.

```
[2]: import matplotlib.colors as pltcolor
from matplotlib.colors import LinearSegmentedColormap
import matplotlib.pyplot as plt

def PLOT(X,Y,C,gamma):

    # CREATING TESTING GRID
    x_grid, y_grid = np.meshgrid(np.arange(-2.3,2.8,0.05),
                                np.arange(-3.1,3.3,0.05))
    a,b = len(x_grid),len(x_grid[0])
    x_flat, y_flat = np.reshape(x_grid,(a*b,)), np.reshape(y_grid,(a*b,))

    # PLOT PARAMETERS
    fig, ax = plt.subplots(2, 2, figsize=(12.8,8))
    normalize = pltcolor.TwoSlopeNorm(vcenter=0)
    fig.suptitle("RBF kernel for the parameter values C = " + str(C) +
                 + r'$ \backslash :\backslash : $ and $\backslash :\backslash :\backslash :\backslash :\gamma = $' + str(gamma),
                 x=0.5, y=0.94, fontsize=16)
    color = ["blue", "green", "red"]
```

This first part takes the data with outcomes equal to zero and one (classes A and B) and makes a SVM classification. The outcomes equal to zero are replaced by minus one, and the outcomes equal to one are maintained. Then, the results are plotted with a color map with a range of colors between blue and green.

```

# COMPUTING SVM OF CLASS A VS B (index: 0-->-1 , 1 -->1)

reg_01 = svm(X[:,0],X[:,1],Y,[0,-1],[1,1],C,gamma)
pred_01 = reg_01.predict(x_flat, y_flat)
z_grid_01 = np.reshape(pred_01,(a,b))

# PLOT OF THE RESULTS OF CLASS A VS B
colormap = LinearSegmentedColormap.from_list('map',
                                              ["lightskyblue", "w", "palegreen"], N=256)
ax[0][0].pcolormesh(x_grid,y_grid,z_grid_01,
                     cmap=colormap, shading="auto", norm=normalize)

```

```

for i in range(len(Y)):      #SCATTER
    if (Y[i] == 0 or Y[i] == 1):
        ax[0][0].scatter(X[i,0],X[i,1],color=color[int(Y[i])],
                           s=6,zorder=2)

```

This second part takes the data with outcomes equal to zero and two (classes A and C) and makes a SVM classification. The outcomes equal to zero are replaced by minus one, and the outcomes equal to two are replaced by one. Then, the results are plotted with a color map with a range of colors between blue and red.

```

# COMPUTING SVM OF CLASS A VS C (index: 0-->-1 , 2 -->1)

reg_02 = svm(X[:,0],X[:,1],Y,[0,-1],[2,1],C,gamma)
pred_02 = reg_02.predict(x_flat, y_flat)
z_grid_02 = np.reshape(pred_02,(a,b))

# PLOT OF THE RESULTS OF CLASS A VS C
colormap = LinearSegmentedColormap.from_list('map',
                                              ["lightskyblue", "w", "lightcoral"], N=256)
ax[0][1].pcolormesh(x_grid,y_grid,z_grid_02,
                     cmap=colormap, shading="auto", norm=normalize)

for i in range(len(Y)):      #SCATTER
    if (Y[i] == 0 or Y[i] == 2):
        ax[0][1].scatter(X[i,0],X[i,1],color=color[int(Y[i])],
                           s=6,zorder=2)

```

This third part takes the data with outcomes equal to one and two (classes B and C) and makes a SVM classification. The outcomes equal to one are replaced by minus one, and the outcomes equal to two are replaced by one. Then, the results are plotted with a color map with a range of colors between green and red.

```

# COMPUTING SVM OF CLASS B VS C (index: 1-->-1 , 2 -->1)

reg_12 = svm(X[:,0],X[:,1],Y,[1,-1],[2,1],C,gamma)
pred_12 = reg_12.predict(x_flat, y_flat)
z_grid_12 = np.reshape(pred_12,(a,b))

# PLOT OF THE RESULTS OF CLASS B VS C
colormap = LinearSegmentedColormap.from_list('map',
                                              ["palegreen", "w", "lightcoral"], N=256)
ax[1][0].pcolormesh(x_grid,y_grid,z_grid_12,
                     cmap=colormap, shading="auto", norm=normalize)

```

```

for i in range(len(Y)):      #SCATTER
    if (Y[i] == 1 or Y[i] == 2):
        ax[1][0].scatter(X[i,0],X[i,1],color=color[int(Y[i])],
                           s=6,zorder=2)

```

The last part of the code takes the different three arrays containing the predictions and uses the *sign* function to convert the values lower than zero into minus one and the values higher than zero to one. Then, we store the total probability each point has to be labeled with 0, 1 or 2 into three arrays. The elements on the arrays *blue-probability*, *green-probability*, and *red-probability* will have the following possible values  $\{-2, 0, 2\}$ . Let us see this with an example. Suppose the first element of the arrays *blue-probability*, *green-probability* and *red-probability* correspond to the information of a certain point  $x_0, y_0$  and that the real outcome of the point is 0. Imagine that on the first regression (A vs B), we predict an outcome of 0, on the second regression (A vs C) we predict an outcome of 0 and on the last regression (B vs C) an outcome of 2. The first element of the *blue-probability* array will be equal to 2, the first element of the *green-probability* array will be equal to -2, and for the *red-probability* we will have a value of 0. Then, the next part of the code will take the maximum value among the first elements of these three arrays and predict the outcome associated with the color, in this case 0. Lastly, the results are plotted (*one vs one* method).

```

#COMBINING THE RESULTS
pred_01 = np.array([np.sign(pred_01[i]) for i in range(len(pred_01))])
pred_02 = np.array([np.sign(pred_02[i]) for i in range(len(pred_02))])
pred_12 = np.array([np.sign(pred_12[i]) for i in range(len(pred_12))])

blue_prob = -pred_01 - pred_02
green_prob = pred_01 - pred_12
red_prob = pred_02 + pred_12

predictions = np.zeros(np.shape(blue_prob))

for i in range(len(blue_prob)):
    probs_i=np.array([blue_prob[i][0],
                      green_prob[i][0],red_prob[i][0]])
    predictions[i][0] = np.where(probs_i == np.max(probs_i))[0][0]

predictions = np.reshape(predictions,(a,b))

#PLOT OF THE COMBINED RESULTS
ax[1][1].contourf(x_grid,y_grid,predictions,2, alpha = 0.6,
                   colors=["lightskyblue","palegreen","lightcoral"])

```

```

for i in range(len(X)):
    ax[1][1].scatter(X[i][0], X[i][1], color=color[Y[i][0]], s=6, zorder=2)

```

Next, we have the cross-validation function, a combination of the plot function and the k-fold cross-validation function of other classification models. We will test the accuracy of the training data and also of the testing data.

```

[3]: from sklearn.model_selection import KFold

def multi_K_fold_cross_validation_SVM(X,Y,C,gamma,iterations):

    train_acc = []
    test_acc = []
    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train_01 = np.array([X[i] for i in train_index if Y[i]!=2])
            Y_train_01 = np.array([Y[i] for i in train_index if Y[i]!=2])
            X_train_02 = np.array([X[i] for i in train_index if Y[i]!=1])
            Y_train_02 = np.array([Y[i] for i in train_index if Y[i]!=1])
            X_train_12 = np.array([X[i] for i in train_index if Y[i]==0])
            Y_train_12 = np.array([Y[i] for i in train_index if Y[i]==0])

            #SVM OF CLASS A VS B (index: 0-->-1 , 1 -->1)
            reg_01 = svm(X_train_01[:,0],X_train_01[:,1],
                          Y_train_01,[0,-1],[1,1],C,gamma)
            pred_01 = reg_01.predict(X[:,0],X[:,1])
            pred_01 = np.array([np.sign(pred_01[i]) \
                               for i in range(len(pred_01))])

            #SVM OF CLASS A VS C (index: 0-->-1 , 2 -->1)
            reg_02 = svm(X_train_02[:,0],X_train_02[:,1],
                          Y_train_02,[0,-1],[2,1],C,gamma)
            pred_02 = reg_02.predict(X[:,0],X[:,1])
            pred_02 = np.array([np.sign(pred_02[i]) \
                               for i in range(len(pred_02))])

            #SVM OF CLASS B VS C (index: 1-->-1 , 2 -->1)
            reg_12 = svm(X_train_12[:,0],X_train_12[:,1],

```

```

Y_train_12,[1,-1],[2,1],C,gamma)
pred_12 = reg_12.predict(X[:,0],X[:,1])
pred_12 = np.array([np.sign(pred_12[i]) \
                    for i in range(len(pred_12))])

#COMBINING RESULTS
blue_prob = -pred_01 -pred_02
green_prob = pred_01 - pred_12
red_prob = pred_02 + pred_12

# PREDICTIONS
predictions = np.zeros(np.shape(blue_prob))

for i in range(len(blue_prob)):

    probs_i=np.array([blue_prob[i][0],green_prob[i][0],
                      red_prob[i][0]])
    predictions[i][0] = np.where(probs_i == \
                                 np.max(probs_i))[0][0]

# COMPUTING THE ERRORS
error = predictions - np.reshape(Y,(len(Y),1))
test_error = np.array([error[i] for i \
                      in range(len(error)) if i in test_index])
train_error = np.array([error[i] for i \
                      in range(len(error)) if i in train_index])

# COUNTING THE CORRECT PREDICTIONS
correct_train_pred = np.count_nonzero(train_error==0)
correct_test_pred = np.count_nonzero(test_error==0)

# COMPUTING THE ACCURACY
train_acc.append(100*(correct_train_pred/len(train_error)))
test_acc.append(100*(correct_test_pred/len(test_error)))

return(np.mean(train_acc),np.mean(test_acc))

```

Once we have the plot and cross-validation functions, we can use them on the Iris data collection. As before, we will use 5-fold cross-validation over the first 200 random states.

```
[4]: train_accuracy,test_accuracy = multi_K_fold_cross_validation_SVM(X,
                                         Y,C = 10,gamma = 1, iterations=200)
print("Accuracy for training data:",train_accuracy,"%")
print("Accuracy for testing data:",test_accuracy,"%")
```

Accuracy for training data: 83.23916666666668 %
Accuracy for testing data: 76.43666666666665 %

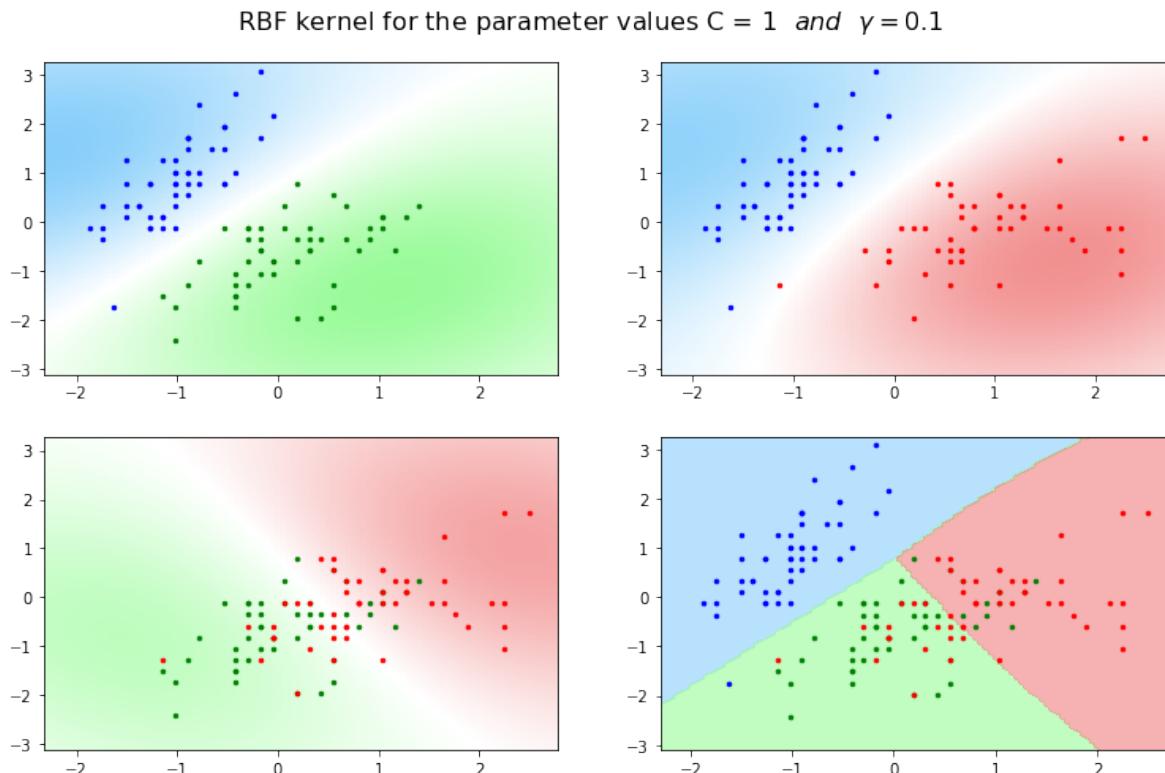
I used the functions on a mesh of  $C$  and  $\gamma$  values. The obtained results can be seen in the following tables and graphs. As we can see, as we increase the value of the  $C$  and  $\gamma$  parameters, the error over the training data is minimized. The effect on the testing data is the opposite; as we increase the value of the parameters, we overfit the model more, reducing the accuracy over the testing data.

	$\gamma = 0.1$	$\gamma = 1$	$\gamma = 10$
$C = 1$	79.62%	81.56%	87.05%
$C = 10$	81.11%	83.23%	92.20%
$C = 100$	82.05%	85.33%	93.88%

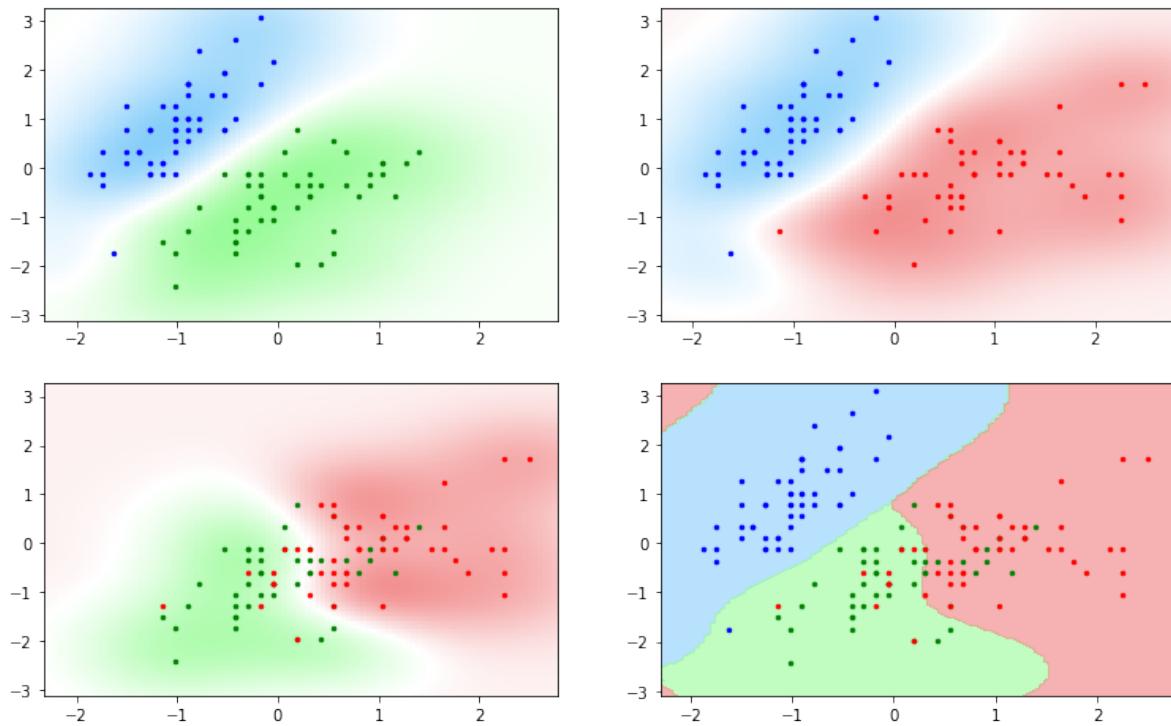
Table 5.1: Accuracy of the model over the training data using cross-validation on the first 200 random states, in terms of the  $C$  and  $\gamma$  parameters.

	$\gamma = 0.1$	$\gamma = 1$	$\gamma = 10$
$C = 1$	77.98%	77.56%	71.55%
$C = 10$	77.33%	76.44%	67.74%
$C = 100$	78.02%	73.62%	67.0%

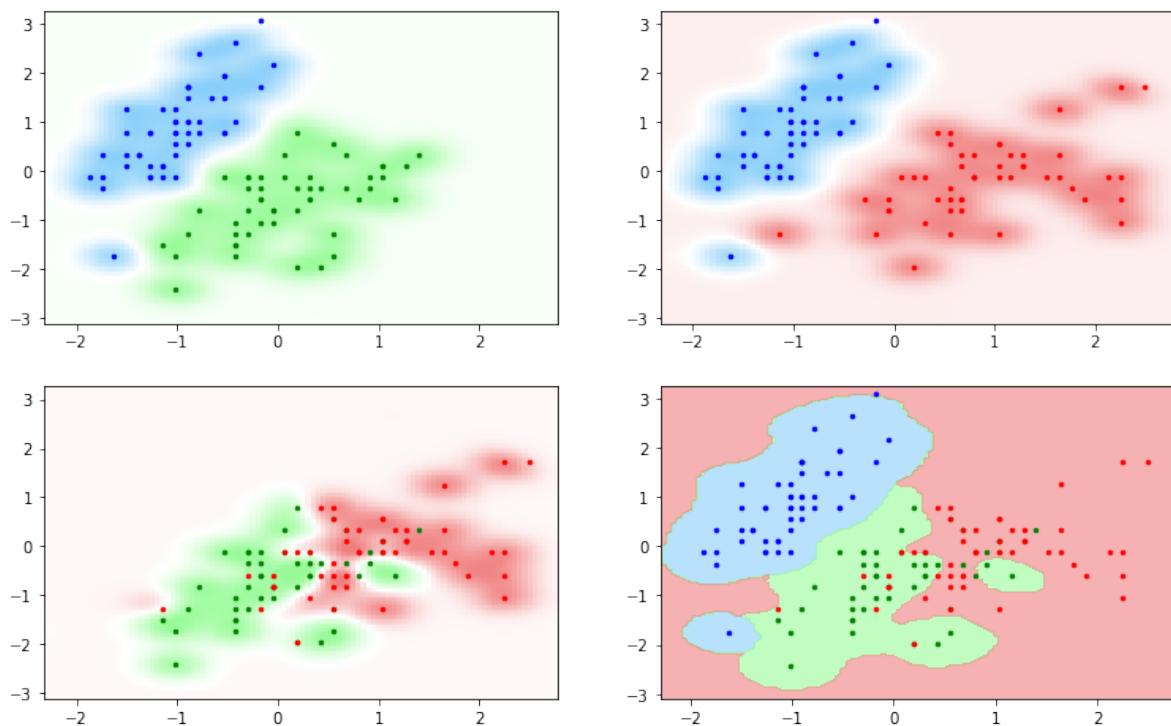
Table 5.2: Accuracy of the model over the testing data using cross-validation on the first 200 random states, in terms of the  $C$  and  $\gamma$  parameters.



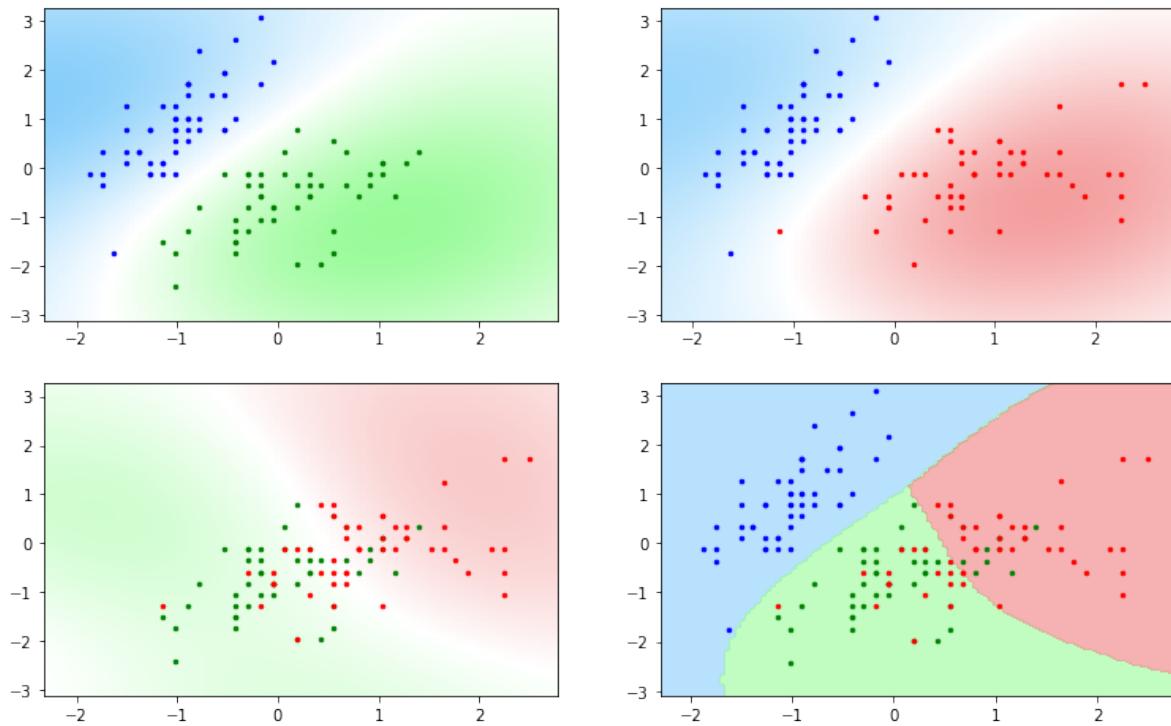
RBF kernel for the parameter values  $C = 1$  and  $\gamma = 1$



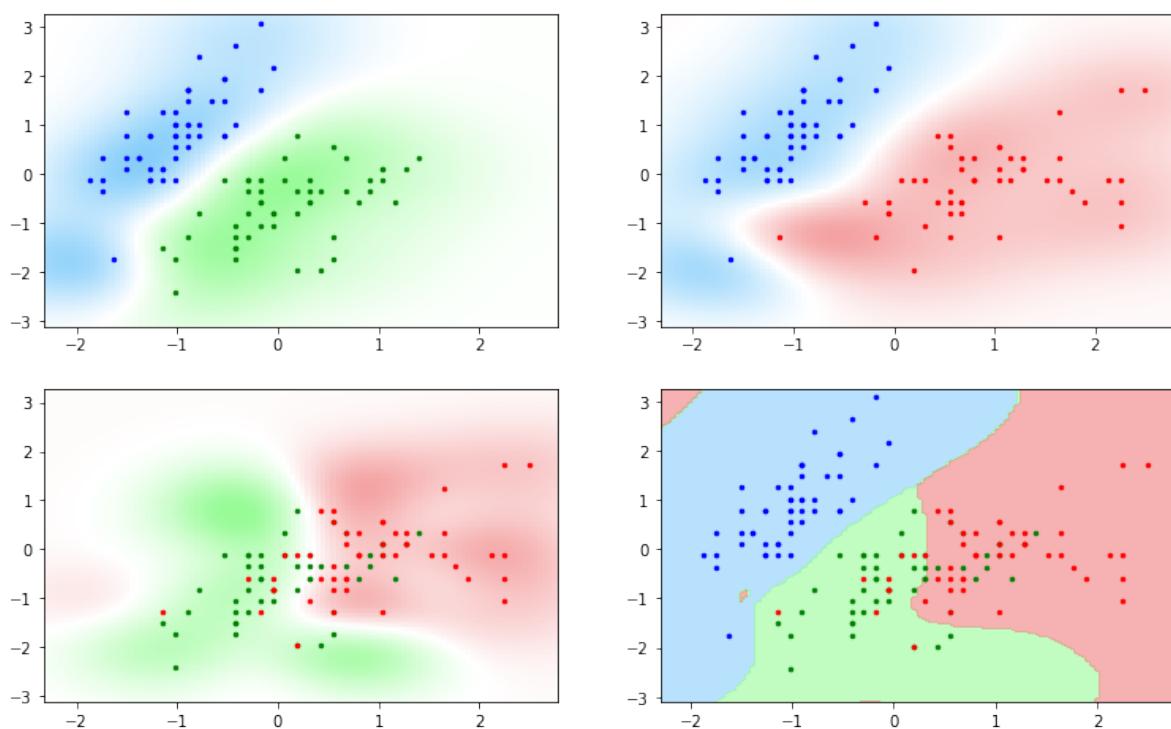
RBF kernel for the parameter values  $C = 1$  and  $\gamma = 10$



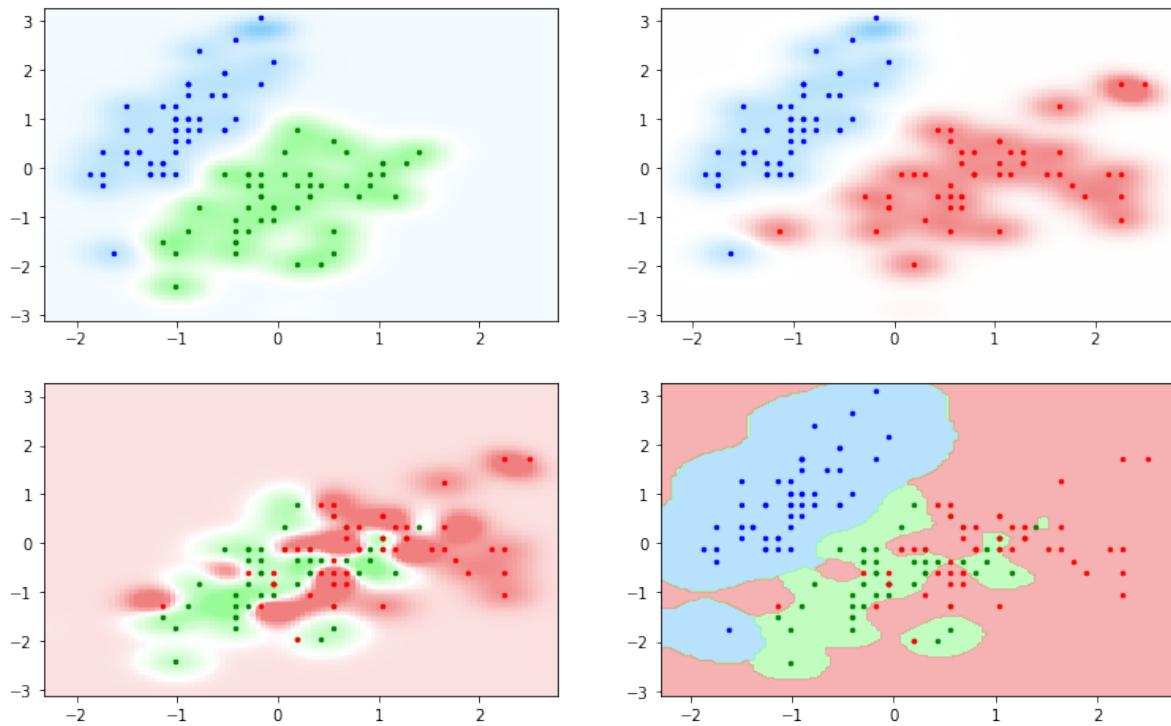
RBF kernel for the parameter values  $C = 10$  and  $\gamma = 0.1$



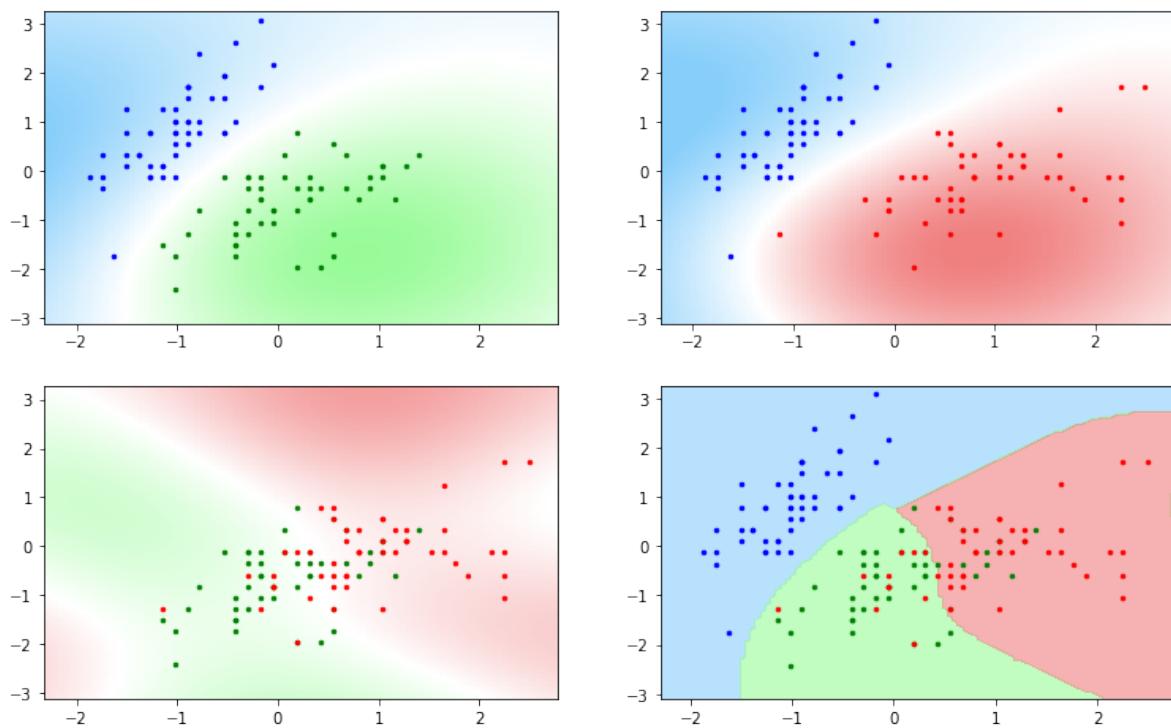
RBF kernel for the parameter values  $C = 10$  and  $\gamma = 1$



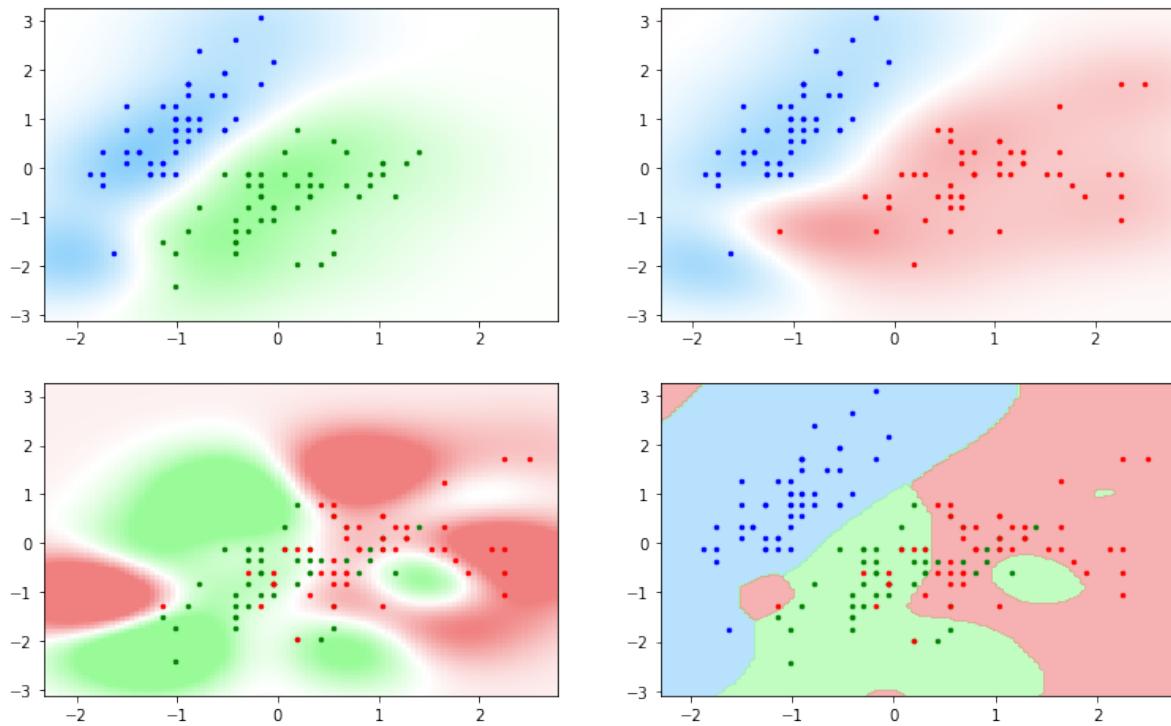
RBF kernel for the parameter values  $C = 10$  and  $\gamma = 10$



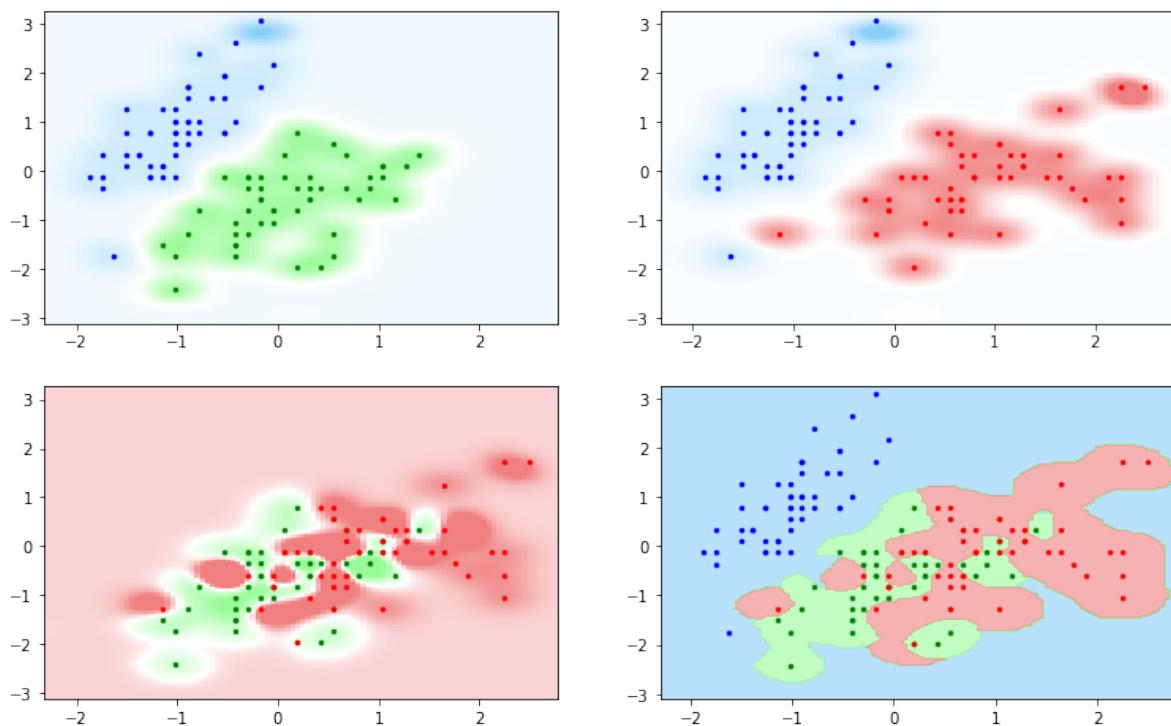
RBF kernel for the parameter values  $C = 100$  and  $\gamma = 0.1$



RBF kernel for the parameter values  $C = 100$  and  $\gamma = 1$



RBF kernel for the parameter values  $C = 100$  and  $\gamma = 10$



As mentioned in this chapter, increasing the  $C$  parameter makes the algorithm prioritize reducing the errors in the training data. We can see on the plots that when we increase this parameter, the model classifies more training points correctly. The  $\gamma$  parameter influences the size of the scalar product between the different points. When the reach is lower (higher  $\gamma$ ), distant points will have no interaction, and hence, the predictions will be more localized, i.e., we will have more isles in the distribution of the space.

Since I wanted to explore higher values of  $C$ , but the model took more time to converge as we increased  $C$ , and for low  $C$  values was already computationally expensive, I decided to use the implementation of the *Sklearn* for a few more parameter values.

	$\gamma = 0.1$	$\gamma = 1$	$\gamma = 10$
$C = 1$	78.0% (+0.02%)	77.57% (+0.01%)	71.61% (+0.06%)
$C = 10$	77.33% (+0%)	76.45% (+0.01%)	68.38% (+0.64%)
$C = 100$	78.01% (-0.01%)	73.62% (+0%)	68.3% (+1.30%)

Table 5.3: Accuracy of *Sklearn.svm* on the testing data using cross-validation on the first 200 random states and the deviations inside the parenthesis in comparison to my model.

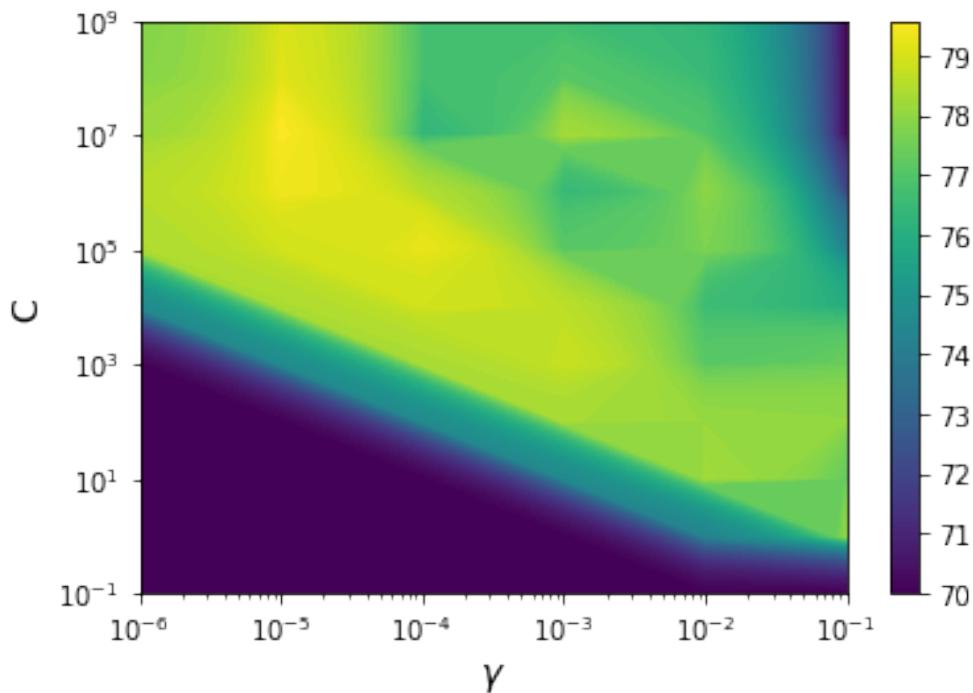


Figure 5.22: Accuracy of *Sklearn.svm* on the testing data using cross-validation on the first 200 random states.

As shown in the table, my model predicted the accuracy similarly to the *Scikitlearn* implementation. The difference increases as  $\gamma$  grows larger. This error may be due to the expansion of Equation 5.71 needing more terms as  $\gamma$  grows larger. Remember that adding more terms to the basis does not impact the computing time much, as it will only be used to update  $w$ . I did not increase the terms because the factorial of those numbers were so large that Python could not store them. Although I am sure there is a workaround to avoid this issue, we do not mind that accuracy difference as we only wanted to test the model.

The maximum value of the accuracy using the *Scikitlearn* implementation was found to be 79.55% when testing the model on the parameter ranges  $C \in [10^{-1}, 10^9]$  and  $\gamma \in [10^{-6}, 10^{-1}]$ .

## 5.5 K-means clustering [Unsupervised]

### 5.5.1 Mathematical derivation

This algorithm is straightforward in concept; given a data set of unlabeled data (i.e., we do not know the target or outcome of the data), we can use the K-means algorithm as a clustering algorithm, i.e., to divide the data set in clusters based on the proximity of the points. The steps followed are the following:

- 1) We initialize the algorithm by choosing  $k$  random data points, where  $k$  is the desired number of clusters. I will call these points cluster nodes.
- 2) We repeat the following steps for a given number of iterations:
  - We generate the data clusters by assigning each data point to the closest cluster node.
  - We update the position of the cluster nodes by substituting them for the centroids of the clusters (the centroid of a cluster will be the mean position of the points assigned to it).
  - We check if the displacement of the cluster nodes is smaller than an  $\epsilon$  parameter, and we terminate the iterations if this is the case.

## 5.5.2 Algorithm implementation

First, we start with the implementation of the K-means algorithm.

```
[1]: import random
import numpy as np
import math as math
import copy as copy

class kmeans():

    def __init__(self,X,Y,k,eps):

        self.X = X
        self.Y = Y
        self.k = k
        self.eps = eps

        self.k_x, self.k_y = self.generate_clusters()
```

This first function generates k random cluster nodes by picking k indexes randomly from the data points. The first cluster node will have the integer zero as the outcome, the second node will have the integer one as the outcome, and so on.

```
def generate_clusters(self):

    k_x = []
    k_y = []

    for i in range(self.k):

        k_value = random.randrange(len(self.X))
        k_x.append(self.X[k_value])
        k_y.append(self.Y[k_value])

    return(np.array(k_x),np.array(k_y))
```

The next function computes the distance from all the data points to the different cluster nodes. Then, we assign to each point the outcome of the closest cluster node.

```
def distances(self,X,Y,k_x,k_y):

    distances = np.zeros((len(X),len(k_x)))
    classification = np.zeros((len(X)))
```

```

for i in range(len(X)):
    for j in range(len(k_x)):

        distances[i][j] = math.sqrt((X[i]-k_x[j])**2.0 \
            + (Y[i]-k_y[j])**2.0)

for i in range(len(X)):

    cluster = np.where(distances[i] == np.min(distances[i]))
    classification[i] = cluster[0][0]

return(classification)

```

The *centroid* function computes the average position of the different clusters. It does this by computing the average position of the points belonging to the clusters.

```

def centroid(self,classification):

    k_x = []
    k_y = []

    for i in range(self.k):
        k_mean_x = 0
        k_mean_y = 0
        counter = 0

        for j in range(len(self.X)):

            if classification[j] == i:
                k_mean_x = k_mean_x + self.X[j]
                k_mean_y = k_mean_y + self.Y[j]
                counter= counter +1
            else:
                continue

        if counter>0:

            k_x.append(k_mean_x / counter)
            k_y.append(k_mean_y / counter)
        else:
            k_x.append(0.0)
            k_y.append(0.0)

    k_x = np.array(k_x)

```

```

k_y = np.array(k_y)

return(k_x,k_y)

```

This function is the central part of the code. We will iterate until the difference between the position of the centroids on consecutive iterations is smaller than an  $\epsilon$  parameter. For each of these iterations, we will save the position of the centroids on the variables  $kx\_prev$  and  $ky\_prev$  to be used on the next iteration. Then, we will classify the points (assign an outcome) based on proximity to the cluster nodes. Furthermore, update the cluster node positions with the *centroid* function.

```

def main_part(self):

    k_x_prev , k_y_prev = np.zeros(len(self.k_x)), \
                          np.zeros(len(self.k_y))

    while (sum(abs(k_x_prev - self.k_x)) > self.eps and \
           sum(abs(k_y_prev - self.k_y)) > self.eps):

        k_x_prev,k_y_prev = copy.deepcopy(self.k_x), \
                            copy.deepcopy(self.k_y)
        classification = self.distances(self.X,self.Y,
                                         k_x_prev,k_y_prev)
        self.k_x,self.k_y = self.centroid(classification)

    return(self.k_x,self.k_y)

def predict(self,X_test,Y_test):

    prediction = self.distances(X_test,Y_test,self.k_x,self.k_y)

    return(prediction,self.k_x,self.k_y)

```

Then, with the following piece of code, we can plot the final configuration of the clusters.

```

[2]: def PLOT(X,Y,k,eps):

    # CREATING TESTING GRID
    x_grid, y_grid = np.meshgrid(np.arange(-2.3,2.8,0.05),
                                 np.arange(-3.1,3.3,0.05))
    a,b = len(x_grid),len(x_grid[0])
    x_flat, y_flat = np.reshape(x_grid,(a*b,)), np.reshape(y_grid,(a*b,))

    # USING THE K-MEANS MODEL

```

```

reg = kmeans(X,Y,k,eps)
reg.main_part()

z_flat,k_x,k_y = reg.predict(x_flat, y_flat)
data_pred = reg.predict(X,Y)[0]
z_grid = np.reshape(z_flat,(a,b))

# PLOT
color=["blue","green","red"]

#SCATTER OF DATA
for i in range(len(X)):

    plt.scatter(X[i],Y[i],color=color[int(data_pred[i])],s=6,zorder=7)

#SCATTER OF CLUSTER NODES
for j in range(len(k_x)):

    plt.scatter(k_x[j],k_y[j],
                color=color[j],s=60,zorder=4,marker="x")

#PREDICTION ON THE 2D SPACE
plt.contourf(x_grid,y_grid,z_grid,2,colors=["paleturquoise",
                                              "yellowgreen","lightcoral"],alpha=0.7)

```

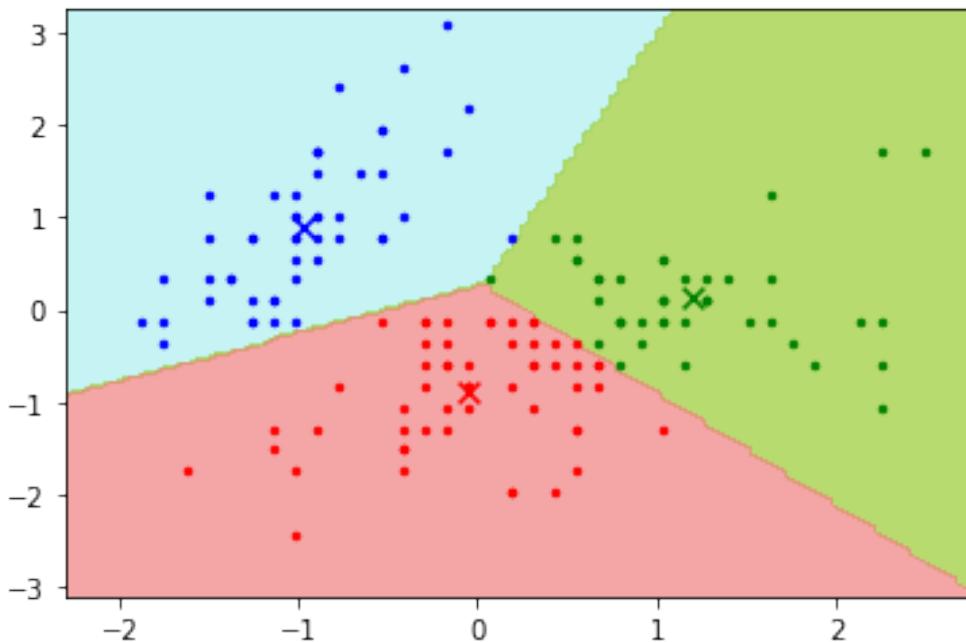


Figure 5.23: Visualization of the K-means model on the Iris data collection.

Last, we have the function that computes the accuracy. Keep in mind that when we trained this model, we had no information about the outcomes of the data. Now, we will use that information to see how accurately the model divided the data. We will have three arrays composed of three elements; the array named *cluster zero*, for example, will store the results of those data points predicted to have outcome zero. It will count how many of those points predicted to have an outcome of zero had an actual outcome of zero, one, or two. We do not care that the predicted and actual outcomes match, as the model had no information about the outcomes, and the cluster nodes were picked randomly. What we care about is what percentage of the points that shared the same outcome were labeled under the same cluster; this is why we will pick the largest number of the array *cluster zero*, the largest number of the array *cluster one* and the largest number of the array *cluster two* divided by the total number of points as the accuracy. We will not split the data collection into training and testing sets; we will just see what percentage of the points are clustered properly.

```
[3]: def K_means_accuracy(X,Y,k,eps,iterations,printing):

    accuracy = []

    for i in range(iterations):

        cluster_0 = [0,0,0]
        cluster_1 = [0,0,0]
        cluster_2 = [0,0,0]

        reg = kmeans(X,Y,k,eps)
        reg.main_part()
        results = reg.predict(X,Y)[0]

        for i in range(len(X)):

            if results[i]==0 :
                cluster_0[Z[i]] = cluster_0[Z[i]] + 1

            elif results[i]==1 :
                cluster_1[Z[i]] = cluster_1[Z[i]] + 1

            elif results[i]==2 :
                cluster_2[Z[i]] = cluster_2[Z[i]] + 1

        right_predictions = max(cluster_0) + max(cluster_1) \
                           + max(cluster_2)

        if (printing == "yes"):
```

```

print("outcome = 0 | outcome = 1 | outcome = 2")
print("cluster 1 ",cluster_0)
print("cluster 2 ",cluster_1)
print("cluster 3 ",cluster_2)
print("Accuracy : "+str(100*right_predictions/len(X))+ "%")

accuracy.append(100*right_predictions/len(X))

return(np.mean(accuracy))

```

We will execute the code for one iteration to see the results. As we can see, the model predicts 51 points to have an outcome of zero, and 34 of those points have an outcome equal to one. Then, it predicts 50 points to have an outcome of one, and 34 of those points have a real outcome of two. Furthermore, we have 49 points with an outcome of 0, which were predicted to have an outcome of 2. All the points grouped by this last cluster shared the same outcome.

```
[4]: print(K_means_accuracy(X,Y,k=3,eps=0.000001 ,
                           iterations=1 ,printing="yes"))
```

```

outcome = 0 | outcome = 1 | outcome = 2
cluster 1  [1, 34, 16]
cluster 2  [0, 16, 34]
cluster 3  [49, 0, 0]
Accuracy : 78.0%

```

If we now execute the code for different values of the error  $\epsilon$  and average it over 100,000 iterations, we can see the influence of the error on the predictions.

```
[5]: epsilon = np.array([10**(-i) for i in range(1,14)])
accuracy = np.array([K_means_accuracy(X,Y,k=3,eps=i ,
                                       iterations=100000 ,printing="no") for i in epsilon])
```

As shown in the graph below, as soon as we get under a threshold on  $\epsilon$ , the accuracy does not further improve. The accuracy we obtain with this model is 76.83%, which is a good result considering this is an unsupervised model (unlabeled training data).

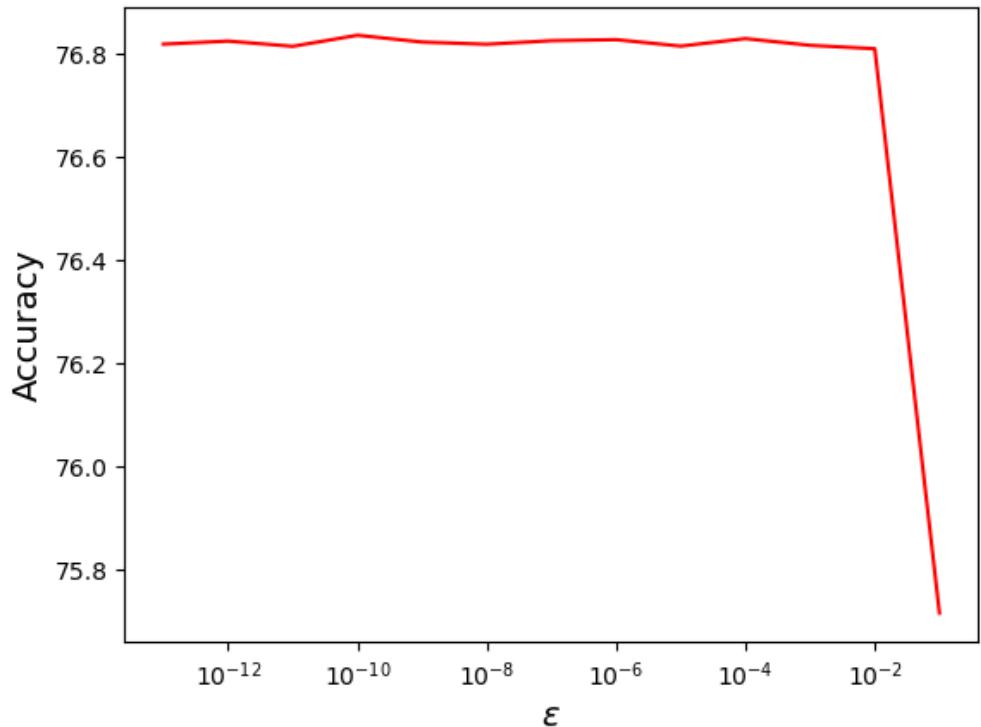


Figure 5.24: Accuracy of the K-means algorithm on the Iris dataset.

# Chapter 6

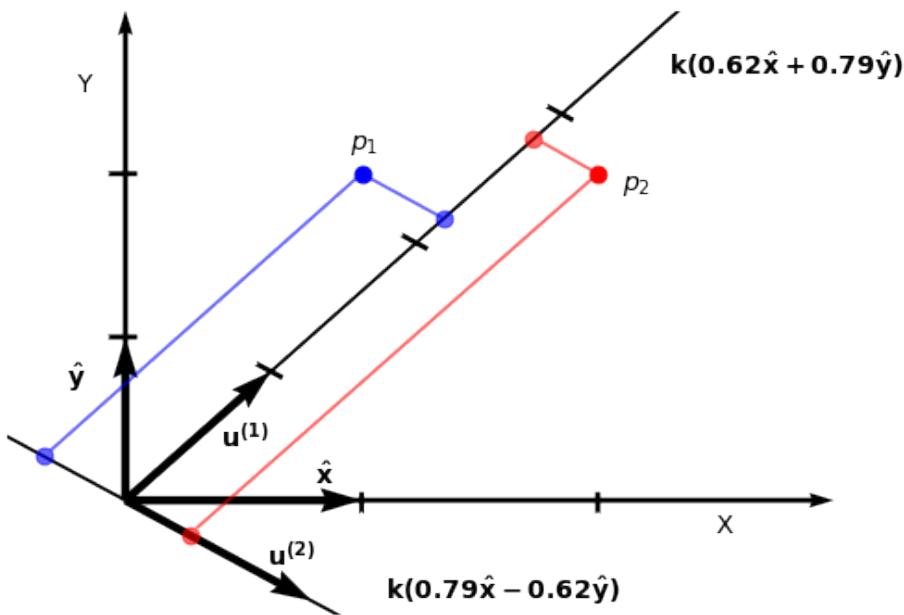
## Dimensionality Reduction Algorithms

### 6.1 Principal component analysis [Unsupervised]

There will be times when we have data that depends on many variables, and visualizing this data and working with it can be challenging. This is where dimensionality reduction comes in handy. The idea behind this is that we can define new variables, which are a combination of the initial variables, to reduce the dimensions of the problem. We will choose those variables that allow us to keep as much information as possible about the data.

#### 6.1.1 Mathematical derivation

[48] Principal Component Analysis (PCA) is an example of a dimensionality reduction algorithm. To see this in more depth, we need to introduce the concept of a projection.



Suppose we have two variables X and Y, and we have a couple of data points  $\mathbf{p}_1 = (x_1, y_1)$  and  $\mathbf{p}_2 = (x_2, y_2)$ . The pair  $(x_i, y_i)$  are the coordinates of the points on the basis  $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ . Now, let's consider the projection of the point  $\mathbf{p}_1$  on the elements of a new basis of unit norm  $(\mathbf{u}^{(1)}, \mathbf{u}^{(2)})$ .

$$\begin{aligned}\mathbb{P}_{\mathbf{p}_1|\mathbf{u}^{(1)}} &= (\mathbf{p}_1 \cdot \mathbf{u}^{(1)}) \mathbf{u}^{(1)} = (1, 2) \begin{pmatrix} 0.62 \\ 0.79 \end{pmatrix} (0.62 \hat{\mathbf{x}} + 0.79 \hat{\mathbf{y}}) = \\ &= 2.2 (0.62 \hat{\mathbf{x}} + 0.79 \hat{\mathbf{y}}) = 2.2 \mathbf{u}^{(1)}\end{aligned}\quad (6.1)$$

$$\begin{aligned}\mathbb{P}_{\mathbf{p}_1|\mathbf{u}^{(2)}} &= (\mathbf{p}_1 \cdot \mathbf{u}^{(2)}) \mathbf{u}^{(2)} = (1, 2) \begin{pmatrix} 0.79 \\ -0.62 \end{pmatrix} (0.79 \hat{\mathbf{x}} - 0.62 \hat{\mathbf{y}}) = \\ &= -0.45 (0.79 \hat{\mathbf{x}} - 0.62 \hat{\mathbf{y}}) = -0.45 \mathbf{u}^{(2)}\end{aligned}\quad (6.2)$$

As shown, the coordinates of a point  $\mathbf{p}_1$  on a basis  $(\mathbf{u}^{(1)}, \mathbf{u}^{(2)})$  is given by the scalar product of the point and each of the basis elements.

Once we have introduced this concept, we can start working on the algorithm. Suppose we have a data collection given by the matrix  $X$ , and an axis  $\mathbf{u}$  given by its coordinates on the  $(\hat{\mathbf{x}}^{(1)}, \hat{\mathbf{x}}^{(2)}, \dots, \hat{\mathbf{x}}^{(D)})$  basis.

$$X = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D} \end{pmatrix} \quad \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_D \end{pmatrix} \quad (6.3)$$

The idea is simple: we want to find an axis  $\mathbf{u}$  so that the projection of the points on this axis is close to the original points, i.e., we want the following quantity to be small:

$$\mathbb{L} = \frac{1}{N} \sum_{i=1}^N [\mathbf{x}_i - (\mathbf{x}_i \cdot \mathbf{u}) \mathbf{u}]^2 = \frac{1}{N} \sum_{i=1}^N [\mathbf{x}_i^2 - (\mathbf{x}_i \cdot \mathbf{u})^2] \quad (6.4)$$

Minimizing that quantity is equivalent to making the second term maximal since the first term does not depend on the choice of the axis. We can write this in matrix notation:

$$\mathbb{L}^* = \frac{1}{N} \sum_{i=1}^N [\mathbf{x}_i \cdot \mathbf{u}]^2 = \frac{1}{N} [X \cdot \mathbf{u}]^T [X \cdot \mathbf{u}] = \frac{1}{N} \mathbf{u}^T X^T X \mathbf{u} = \mathbf{u}^T M \mathbf{u} \quad (6.5)$$

We have the matrix  $M = \frac{1}{N} X^T X$ ; since we want to find the maximum of Equation 6.5, we will work with Lagrange multipliers. As we have seen in other algorithms given the Lagrangian  $\mathbb{L}^*$  subject to the constraint  $g(x) \geq 0$ , if we want to maximize the Lagrangian we have to consider  $\mathbb{L} = \mathbb{L}^* + \lambda g(x)$ . The constraint we consider is  $1 - \mathbf{u} \cdot \mathbf{u} \geq 0$ , which translates to the axis being normalized. Hence, we can write:

$$\mathbb{L} = \mathbf{u}^T M \mathbf{u} + \lambda (1 - \mathbf{u} \cdot \mathbf{u}) \quad (6.6)$$

Computing the derivatives with respect to the parameter  $\lambda$  and the axis, we find:

$$\frac{\partial \mathbb{L}}{\partial \lambda} = 1 - \mathbf{u} \cdot \mathbf{u} = 0 \quad (6.7)$$

$$\frac{\partial \mathbb{L}}{\partial \mathbf{u}} = 2M\mathbf{u} - 2\lambda\mathbf{u} = 0 \quad (6.8)$$

As we can see, we have obtained an eigenvalue problem given by:

$$M\mathbf{u} = \lambda\mathbf{u} \quad (6.9)$$

This result is significant because if we pay close attention, the matrix  $M$  is precisely the covariance matrix whenever the data is normalized, i.e., whenever the mean of each variable among the data is zero. The interpretation of this is clear: when solving the eigenvalue problem, the eigenvectors, which we will call Principal Components (PCs), will be such that the maximum amount of data variation is preserved. The greater the eigenvalue, the more variation or information of the data is preserved. If we rewrite this, the eigenvalues are found easily:

$$M\mathbf{u} = \lambda\mathbb{I}\mathbf{u} \rightarrow (M - \lambda\mathbb{I})\mathbf{u} = 0 \quad (6.10)$$

This is achieved when the determinant of  $M - \lambda\mathbb{I}$  equals zero. Then to obtain the eigenvectors is straightforward by introducing the eigenvalues on the equation:  $M\mathbf{u} = \lambda\mathbf{u}$ . Since the matrix  $M$  will have  $D \times D$  dimensions, we can find a maximum of  $D$  axes if all the variables are linearly independent. To determine how much information or variance of the data is preserved by a given PC labeled by  $\mathbf{u}^{(k)}$  we can compute.

$$\mathbb{I}_k = \frac{\lambda_k}{\sum_{i=1}^D \lambda_i} \quad (6.11)$$

Now that the algorithm's objective is clear, we can reduce the data dimensions from  $D$  to a lower number by keeping only those axes of the new basis that preserve the most information or variance about the data. One last thing that is important to mention is that whenever the number of data points  $N$  is lower than the dimensions  $D$ , only  $N$  out of the  $D$  eigenvalues will be relevant, which comes from the algebraic properties of the eigenvalue problem.

Let us put this to the test with the example of the graph. We can see we have two data points  $\mathbf{x}_1 = (1, 2)$  and  $\mathbf{x}_2 = (2, 2)$ . To find the basis in which most information is preserved, we have to solve the eigenvalue problem of the matrix :

$$M = \frac{1}{2} \begin{pmatrix} X_{1,1} & X_{2,1} \\ X_{1,2} & X_{2,2} \end{pmatrix} \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix} \quad (6.12)$$

Now, we have to find the lambda values and vector coordinates  $u_1$  and  $u_2$  so that:

$$M \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \lambda \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \quad (6.13)$$

For our example, M is given by:

$$M = \begin{pmatrix} 5/2 & 3 \\ 3 & 4 \end{pmatrix} \quad (6.14)$$

To solve the eigenvalue problem, we have to solve the determinant of the following matrix and make it equal to zero:

$$\begin{vmatrix} 5/2 - \lambda & 3 \\ 3 & 4 - \lambda \end{vmatrix} = (5/2 - \lambda)(4 - \lambda) - 9 = \lambda^2 - \frac{13}{2}\lambda + 1 = 0 \quad (6.15)$$

This equation has the solutions  $\lambda_1 = 6.34233$  and  $\lambda_2 = 0.15767$ . As mentioned, the eigenvalues are a measurement of how much information is preserved by keeping the different axes; we can measure this by computing the proportion of each eigenvalue with respect to the sum of the eigenvalues:

$$i_1 = \frac{6.34233}{6.34233 + 0.15767} = 0.976 \quad i_2 = \frac{0.15767}{6.34233 + 0.15767} = 0.024 \quad (6.16)$$

By keeping the first axis, we preserve 98 % of the information, and adding the second axis only results in an information gain of 2 %. We could reduce the 2D problem to a 1D problem for this given data. Now, we must find the eigenvectors (axis) associated with these eigenvalues. We start with the first one:

$$\begin{pmatrix} 5/2 & 3 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} u_1^{(1)} \\ u_2^{(1)} \end{pmatrix} = 6.34233 \begin{pmatrix} u_1^{(1)} \\ u_2^{(1)} \end{pmatrix} \quad (6.17)$$

Both equations obtained from this are equivalent; simply taking the first one:  $2.5u_1^{(1)} + 3u_2^{(1)} = 6.34233u_1^{(1)}$  we find:  $u_2^{(1)} = 1.2808u_1^{(1)}$ . From this we can take the vector  $\mathbf{u}^{(1)} = (1, 1.2808)$  and after normalizing it:  $\hat{\mathbf{u}}^{(1)} = (0.6154, 0.7882)$ . Now for the least relevant eigenvalue:

$$\begin{pmatrix} 5/2 & 3 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} u_1^{(2)} \\ u_2^{(2)} \end{pmatrix} = 0.15767 \begin{pmatrix} u_1^{(2)} \\ u_2^{(2)} \end{pmatrix} \quad (6.18)$$

Both equations obtained from this are equivalent; simply taking the first one:  $2.5u_1^{(2)} + 3u_2^{(2)} = 0.15767u_1^{(2)}$  we find:  $u_2^{(2)} = -0.7808u_1^{(2)}$ . From this we can take the vector  $\mathbf{u}^{(2)} = (1, -0.7808)$  and after normalizing it:  $\hat{\mathbf{u}}^{(2)} = (0.7882, -0.6154)$ . So, to summarize the results, we have found the eigenvalues and eigenvectors:

$$\lambda^{(1)} = 6.34233 \quad \hat{\mathbf{u}}^{(1)} = (0.6154, 0.7882) \quad (6.19)$$

$$\lambda^{(2)} = 0.15767 \quad \hat{\mathbf{u}}^{(2)} = (0.7882, -0.6154) \quad (6.20)$$

Now, all there is left to do is to compute the coordinates of the data points on the new basis:

$$\mathbf{x}_1 = (1, 2) \rightarrow \mathbf{x}_1^* = (\hat{\mathbf{u}}^{(1)} \cdot \mathbf{x}_1, \hat{\mathbf{u}}^{(2)} \cdot \mathbf{x}_1) = (2.19, -0.44) \quad (6.21)$$

$$\mathbf{x}_2 = (2, 2) \rightarrow \mathbf{x}_2^* = (\hat{\mathbf{u}}^{(1)} \cdot \mathbf{x}_2, \hat{\mathbf{u}}^{(2)} \cdot \mathbf{x}_2) = (2.81, 0.35) \quad (6.22)$$

So if, for example, we wanted to reduce the dimensions of the problem from 2D to 1D, we could keep only the axis  $\mathbf{u}^{(1)}$  retaining 98% of the information. To compute the value of this new variable for our data points, we would have the equation  $X_{i,*} = 0.62X_{i,1} + 0.79X_{i,2}$ .

## 6.1.2 Algorithm implementation

We proceed with the implementation of the PCA algorithm, which is straightforward.

```
[1]: import numpy as np
import copy as copy
from numpy import linalg as la

class PCA():

    def __init__(self,X,dimensions,labels):

        self.X = X
        self.dimensions = dimensions
        self.labels = np.array(labels)
```

This function is the central part of the code; it computes the matrix  $M$ , then obtains the eigenvalues and eigenvectors of the matrix, computes the information gain of each axis, and then calls for the  $D$  reduction function to obtain the data on the new basis. It also limits the eigenvalues to  $N$  whenever we have fewer data points  $N$  than dimensions  $D$ .

```
def pca_main(self):

    D = self.dimensions
    X = np.array(self.X)
    X = X - np.mean(X, axis=0) #mean centering
    M = X.transpose().dot(X)/len(X)

    eigenvalues, eigenvectors = self.eigenvalues(M)

    eigenvalues = np.real(eigenvalues)
    eigenvectors = np.real(eigenvectors)

    if (len(X) < len(X[0])):

        eigenvalues = copy.deepcopy(eigenvalues[0:len(X)])
        info = [100*eigenvalues[i]/sum(eigenvalues) \
                for i in range(len(eigenvalues))]

        new_data = self.D_reduction(eigenvectors[0:D],info)

    return(new_data)
```

The `eigenvalues` function computes the eigenvalues and eigenvectors and then sorts them in descending order.

```
def eigenvalues(self,M):  
  
    eigenvalues, eigenvectors = la.eig(M)  
  
    # I put the - sign so it is sorted in descending order  
    sorting = np.argsort(-eigenvalues, axis=-1)  
  
    eigenvalues = eigenvalues[sorting]  
    eigenvectors = eigenvectors[:,sorting]  
  
    return(eigenvalues, eigenvectors.transpose())
```

The last function prints the results of the new basis and then computes the coordinates of the data on this new basis.

```
def D_reduction(self,eigenvectors,info):  
  
    X = np.array(self.X)  
    eigenvectors = np.array(eigenvectors)  
  
    print("-----RESULTS-----")  
    print(self.labels)  
    print("-----")  
  
    for i in range(len(eigenvectors)):  
  
        print("Axis n° ",i+1,", information gain of ",  
              np.around(info[i],2), "%")  
        print("-----")  
        print("----- Coefficients -----")  
        print(eigenvectors[i])  
        print("-----")  
  
    D = self.dimensions  
  
    New_X = np.array([[X[i].dot(eigenvectors[j]) \br/>                      for j in range(D)] for i in range(len(X))])  
  
    return(New_X)
```

If we load all the variables of the Iris dataset (keep in mind that we do not normalize the data as we are already mean-centering it inside the class, and we do not want to divide it by the standard deviation since we want the eigenvalues of the Covariance matrix) and then use the PCA implementation requesting for the two dimensions with the highest information gain, we obtain the following results.

```
[2]: from sklearn import datasets
import matplotlib.pyplot as plt

iris = datasets.load_iris()
X = iris.data
outcome = iris.target

labels = ["sepal length" , "sepal width" ,
          "petal length" , "petal width"]

reg = PCA(X,2,labels)
result = reg.pca_main()

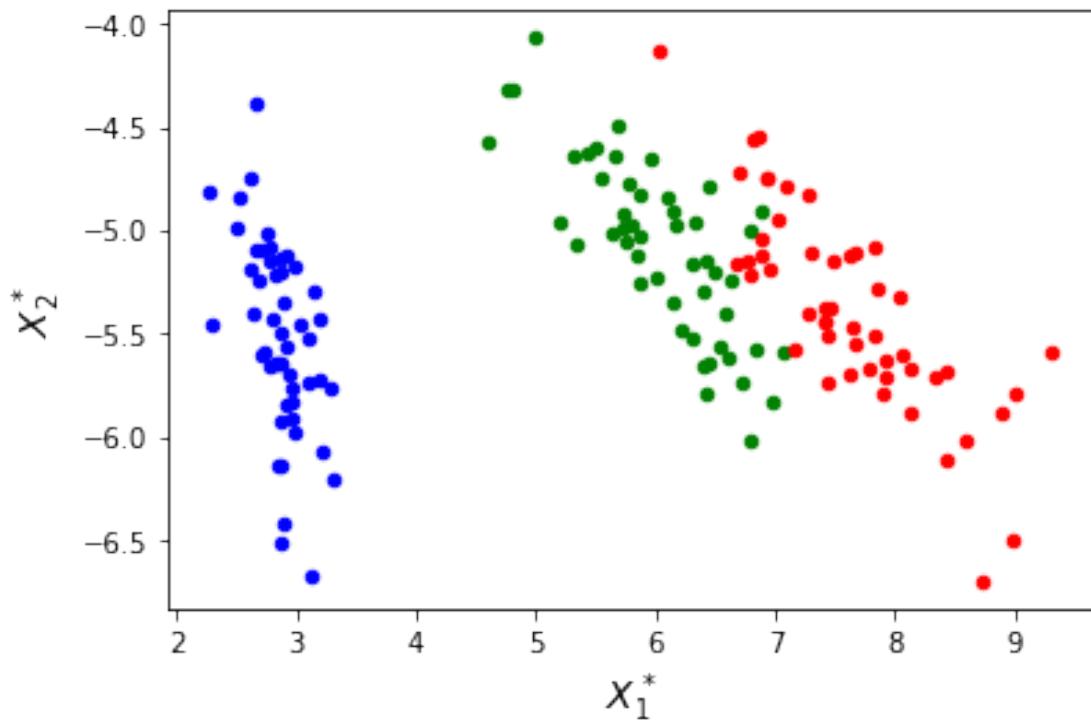
color = ["blue","green","red"]
plt.xlabel(r'$X_1$' , fontsize = 14)
plt.ylabel(r'$X_2$' , fontsize = 14)
for i in range(len(result)):

    plt.scatter(result[i][0],result[i][1],
                color=color[outcome[i]],s=20,zorder=10,alpha=1)
```

```
-----RESULTS-----
['sepal length' 'sepal width' 'petal length' 'petal width']
-----
Axis n° 1 , information gain of  92.46 %
-----
----- Coefficients-----
[ 0.36138659 -0.08452251  0.85667061  0.3582892 ]
-----
Axis n° 2 , information gain of  5.31 %
-----
----- Coefficients-----
[-0.65658877 -0.73016143  0.17337266  0.07548102]
-----
```

As we can see, the variable associated with the first PC, which is given by the equation  $X_1^* = 0.36 \text{ sepal length} - 0.085 \text{ sepal width} + 0.86 \text{ petal length} + 0.36 \text{ petal width}$ , retains 92.46% of the information. The variable associated with the second PC is given by  $X_2^* = -0.66$

*sepal length*  $-0.73$  *sepal width*  $+0.17$  *petal length*  $+0.075$  *petal width* and retains 5.31% of the variation. In total, these two variables retain 97.79% of the variation of the data, and we managed to reduce the dimensions of the database from 4D to 2D.



# Chapter 7

## Classification Trees

Classification trees divide data collections into a branched structure where each generated node corresponds to an interval of the variables from the database. The data points are classified on each of these nodes based on the value of their variables. We associate the outcome of the most predominant class to each node. There are different ways to decide how to branch the trees, which we will see in the following sections.

To test these models, we will work with the Iris dataset, both with the regular dataset and with the result of applying the PCA model to it.

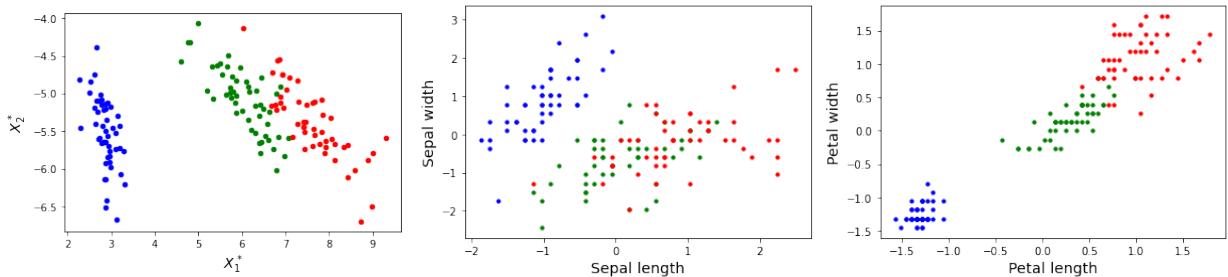


Figure 7.1: Iris data collection plotted on variable pairs and the database that results from the PCA algorithm.

I will label the different variables the following way:  $X_1 \equiv \text{Sepal length}$ ,  $X_2 \equiv \text{Sepal width}$ ,  $X_3 \equiv \text{Petal length}$  and  $X_4 \equiv \text{Petal width}$ . We also have  $X_1^* = 0.36 X_1 - 0.085 X_2 + 0.86 X_3 + 0.36 X_4$  and  $X_2^* = -0.66 X_1 - 0.73 X_2 + 0.17 X_3 + 0.075 X_4$ . Once again, I will normalize the data before splitting it into the training and testing sets. Since we are not trying to assess the exact performance of the models, we do not mind a slight information leakage from the testing data into the training data, as it will not have a big impact on the result, and we are more interested in the learning aspect of the implementation. We will use 5-fold cross-validation, averaged over the first ten random states, to test these classification models.

## 7.1 CART (Classification and regression trees) [Supervised]

### 7.1.1 Mathematical derivation

[11] I will mainly focus on classification trees since they are the most commonly used for this kind of algorithm. As the name suggests, this algorithm classifies the space into different sections based on the outcome (classes) of the data. Here, we have an example of how the algorithm divides the space when we feed it some ternary data.

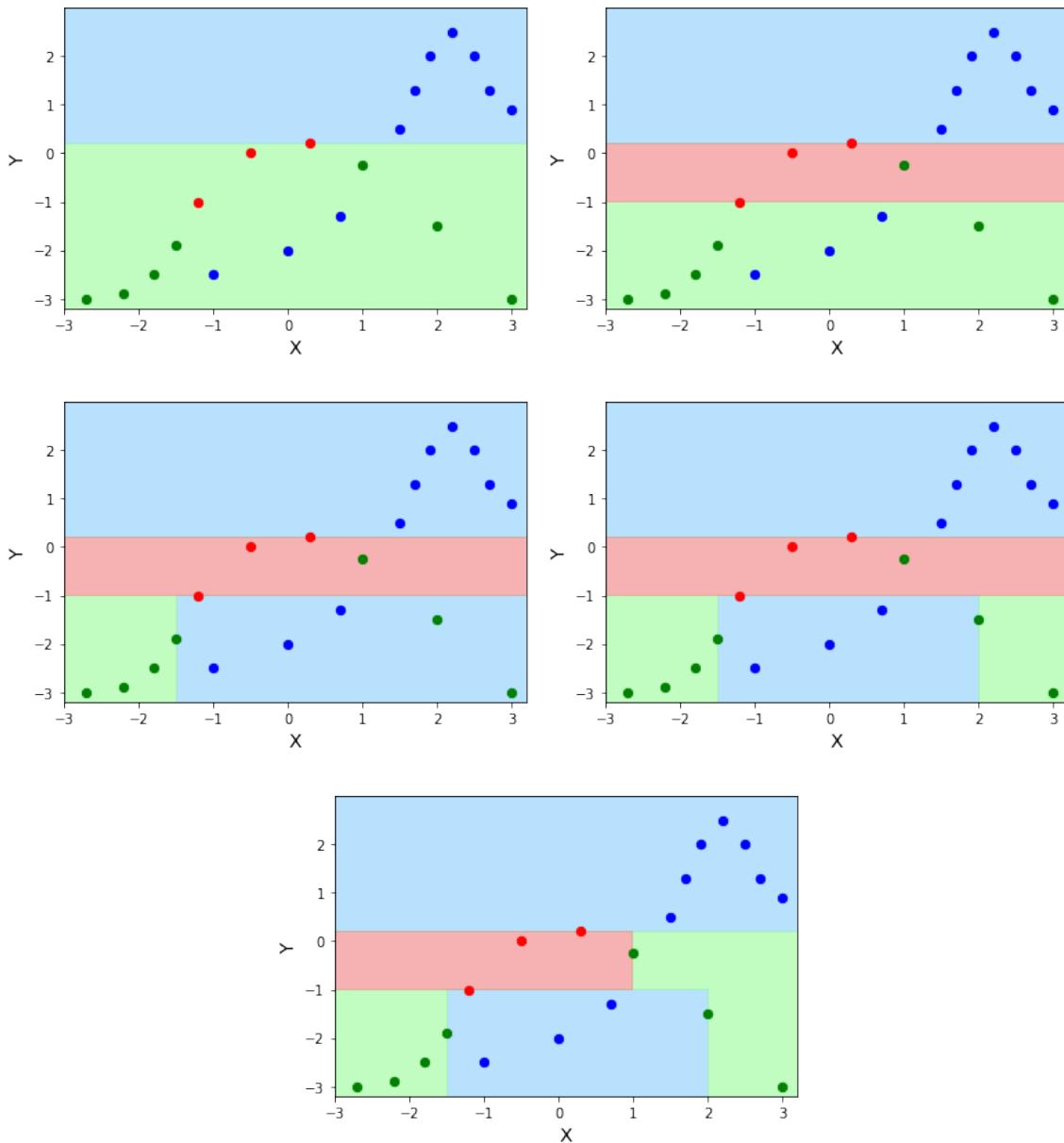


Figure 7.2: Visualization of the iterations of the CART algorithm on a ternary data set.

Next, we will show the classification tree, which has the following elements:

- The root node, which contains all the data.
- The internal nodes are those represented by the light brown color.
- The leaf nodes are those nodes at the bottom of the tree. Depending on the number of iterations, these will contain one class type or more.

A point is predicted to have the outcome of the predominant class of the leaf node in which it falls.

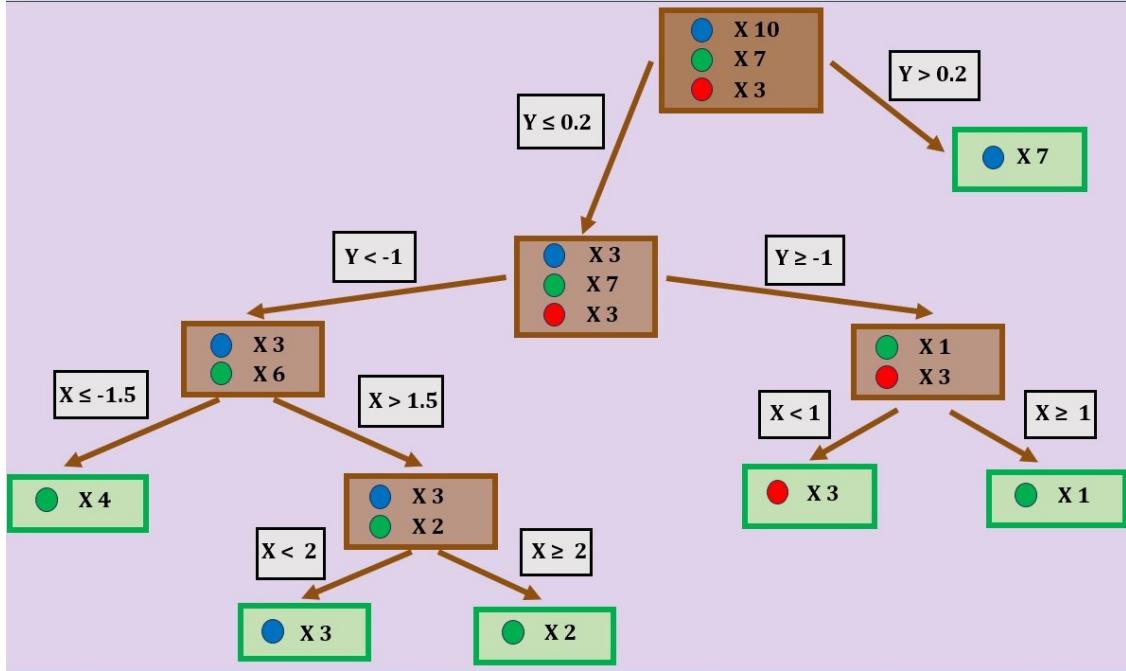


Figure 7.3: Example of the classification tree on the dataset from Figure 7.2.

One may realize that there are several ways to split the space, and this is where the algorithm comes into play. First, we must introduce two concepts: the Gini impurity and the information gain. The Gini impurity of a node is defined as:

$$G(\text{node}) = 1 - \sum_i p_i^2 \quad (7.1)$$

Where  $p_i$  is the fraction of the  $i^{th}$  class on the node. Notice how if a single class is present on a node, the Gini impurity will be equal to zero. Hence, the objective is to minimize the Gini impurity of the child nodes. The information gain reads:

$$IG = w_p G_p(\text{parent node}) - \sum_i w_i G_i(\text{child nodes}) \quad (7.2)$$

$w_i$  is a weight factor equal to the fraction of data points on the  $i^{th}$  node. If the parent node is not pure, i.e., with a nonzero Gini factor, and we split the node so that the child

nodes have a weighted average Gini factor smaller than the parent node, we will have a positive information gain, which is what we are looking for. The term information gain is often used for ID3 and C4.5 models to label a variable that quantifies the importance of the splits. Here, I chose to call this quantity information gain analogously to these other algorithms.

$w_p$  and  $w_i$  must be global weights, meaning that we take the fraction of points out of the whole data collection and not out of the parent node. We will see why this is important with an example.

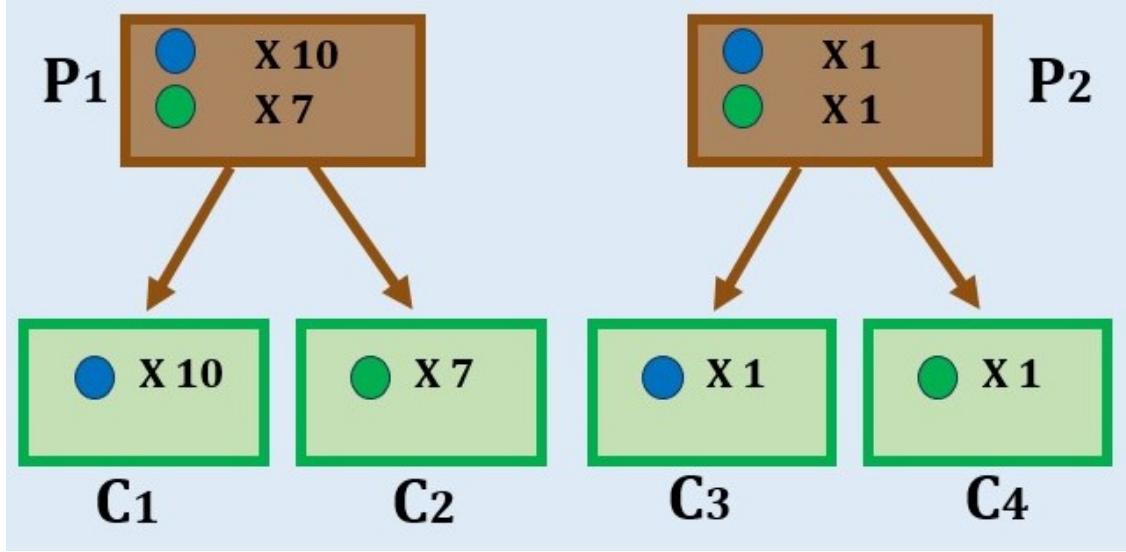


Figure 7.4: Example of a classification tree.

$$G(P_1) = 1 - \left(\frac{7}{17}\right)^2 - \left(\frac{10}{17}\right)^2 = 0.4844$$

$$G(P_2) = 1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2 = 0.5$$

$$G(C_1) = G(C_2) = G(C_3) = G(C_4) = 0$$

If we only take local weights, we have the following:

$$IG(P_1 \rightarrow C_1, C_2) = \frac{17}{17}G(P_1) - \frac{10}{17}G(C_1) - \frac{7}{17}G(C_2) = 0.4844$$

$$IG(P_2 \rightarrow C_3, C_4) = \frac{2}{2}G(P_2) - \frac{1}{2}G(C_3) - \frac{1}{2}G(C_4) = 0.5$$

As shown, the second quantity is greater than the first one, which does not make sense as splitting the first node classifies more points. This is why we need global weights, leading

to the information gain factors:

$$IG(P_1 \rightarrow C_1, C_2) = \frac{17}{19}G(P_1) - \frac{10}{19}G(C_1) - \frac{7}{19}G(C_2) = 0.4334$$

$$IG(P_2 \rightarrow C_3, C_4) = \frac{2}{19}G(P_2) - \frac{1}{19}G(C_3) - \frac{1}{19}G(C_4) = 0.053$$

The split on the left node is significantly prioritized over the second one. This hierarchy matters because we will split only one of the nodes on each iteration. On the first version of the code, I split all the nodes on every single iteration. Hence, comparing the information gains among the nodes was irrelevant. This previous model led to some data sections being overfitted, in contrast to others that were underfitted, making it impossible to make a balanced classification. This new implementation is computationally more expensive as it requires more iterations, but it leads to better results because it lets us control the model's evolution better. The steps of the CART classification algorithm are the following, which are repeated for a given number of iterations:

- Iterate over all the  $X_{i,j}$  values of the data collection, split the data at those points, and compute the information gain of the configuration.
- Keep the split resulting in a more significant information gain and update the intervals of the emerging nodes.

## 7.1.2 Algorithm implementation

We proceed with the implementation of the CART algorithm. Although the steps behind the algorithm seem easy to implement, it was challenging as there are so many index and coordinates to keep track of as the tree is branched.

```
[1]: import math as math
import copy as copy
import numpy as np

class CART():

    def __init__(self,X,Z,iterations,results):

        self.X = X
        self.Z = Z
        self.epsilon = 10**(-5)
        self.iterations = iterations
        self.results = results
```

The *Gini* function computes the Gini impurity of a node by determining the size of the classes among the node. If the total size of the data points is equal to zero (i.e., the input is an empty array of indexes), we will set the Gini index equal to its maximum value.

```
def Gini(self,index):

    Z = self.Z
    class_A_size = len([i for i in index if Z[i] == 0])
    class_B_size = len([i for i in index if Z[i] == 1])
    class_C_size = len([i for i in index if Z[i] == 2])

    total_size = class_A_size + class_B_size + class_C_size

    if (total_size != 0):

        Gini = 1.0 - (class_A_size/total_size)**2.0 \
            - (class_B_size/total_size)**2.0 \
            - (class_C_size/total_size)**2.0
    else:
        Gini = 1.0

    return(Gini)
```

Next, we have the *info gain* function, which computes the information gain of the transition from the parent node to the child nodes given by Equation 7.2.

```

def Info_gain(self, indexes, index_l, index_r):

    s_l = len(index_l)
    s_r = len(index_r)
    s_t = s_l + s_r

    IG = (s_t/len(self.X))*self.Gini(indexes) \
        - (s_l/len(self.X))*self.Gini(index_l) \
        - (s_r/len(self.X))*self.Gini(index_r)

    return(IG)

```

This following function takes a pair of indexes  $(i, j)$  as input; the first index corresponds to a data point, and the second corresponds to a variable of the data collection. It stores all the  $k$  indexes for which  $X_{k,j} < X_{i,j}$  on an array labeled as *left* and those for which  $X_{k,j} > X_{i,j}$  on an array labeled as *right*. It does this both, including the splitting point in the left array and in the right array. Then, it determines which configuration leads to a higher information gain and returns the *left* and *right* arrays storing the indexes. It also returns a one if the splitting point is stored on the left array and a two if it is stored on the right.

```

def index_split(self, index, i, j):

    index_l_1=[k for k in index if self.X[k][j]<=self.X[i][j]]
    index_r_1=[k for k in index if self.X[k][j]>self.X[i][j]]

    IG_1= self.Info_gain(index, index_l_1, index_r_1)

    index_l_2=[k for k in index if self.X[k][j]<self.X[i][j]]
    index_r_2=[k for k in index if self.X[k][j]>=self.X[i][j]]

    IG_2= self.Info_gain(index, index_l_2, index_r_2)

    if (IG_1 >= IG_2):

        return(index_l_1, index_r_1, IG_1, 1)
    else:

        return(index_l_2, index_r_2, IG_2, 2)

```

The *node choosing* function iterates over all the combinations of  $i$  and  $j$  indexes and computes the information gain resulting from splitting the indexes on  $X_{i,j}$ . Then, it obtains the  $(i, j)$  pair for which the information gain is maximum and returns the  $(i, j)$  pair and the arrays containing the split indexes.

```

def node_choosing(self, indexes):

    IG_values=np.zeros((len(indexes),len(self.X[0])))

    for i in range(len(indexes)):
        for j in range(len(self.X[0])):

            IG_values[i][j] = self.index_split(indexes,
                                                indexes[i],j)[2]

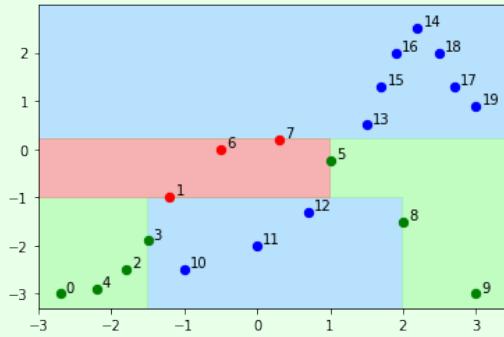
    max_i = np.where(IG_values == np.amax(IG_values))[0][0]
    max_j = np.where(IG_values == np.amax(IG_values))[1][0]

    index_l,index_r,IG,sign_type = self.index_split(indexes,
                                                    indexes[max_i],max_j)

    return(index_l,index_r,IG,sign_type,max_j,indexes[max_i])

```

The *main part* function starts by creating a matrix of size (K, N) where K is the number of iterations plus one and N is the number of data points. This matrix will store the information about the different splits. Let us take the example from before to show this.



$$IS = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 2 & 2 & 2 & 4 & 3 & 3 & 3 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 5 & 2 & 2 & 4 & 4 & 3 & 3 & 3 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The first row of the matrix is composed of zeros, as the index has yet to be split. Since this is the case, the function will take all the indexes in the

first iteration and call the *node-choosing* function. Then, the code replaces the zeros of the second row corresponding to the indexes on the right side of the split by ones. As we can see, these are the indexes of the points in the plot's upper region. On the next iteration, the iterator labeled as *it* on the code will have the value of one, and hence *n* will run through the values zero and one. That is, we will form two arrays, one with those indexes that correspond to a zero on the second row and another one with those that correspond to one, i.e., we will try to split the two nodes we obtained on the first iteration, then store the respective information gain factors and pick the configuration leading to a higher value. Lastly, the indexes of the right side of the new split are changed from either zero or one (depending on which of the nodes that emerged from the first iteration was split) to two. The same steps are repeated for the desired number of iterations.

```

def main_part(self):

    index_splits = np.zeros((self.iterations+1,
                            len(self.X)), dtype = int)

    intervals = [[[min(self.X[:,j])- 1.0,max(self.X[:,j]) + 1.0] \
                  for j in range(len(self.X[0]))]]

    for it in range(self.iterations):

        IG_list = np.zeros((it + 1))

        # TEST ALL THE NODES FOR POTENTIAL SPLITS
        for n in range(it + 1):

            index = np.array([i for i in range(len(self.X))]\ 
                            if index_splits[it,i]==n])

            index_l,index_r,IG,sign_type,max_j,max_i = \
                self.node_choosing(index)

            IG_list[n] = IG

        # SPLIT THE NODE THAT RESULTS ON THE LARGEST IG
        max_pos = np.where(IG_list == np.amax(IG_list))[0][0]

        index = np.array([i for i in range(len(self.X))]\ 
                        if index_splits[it,i]==max_pos])

        index_l,index_r,IG,sign_type,max_j,max_i = \
            self.node_choosing(index)

        index_splits[it+1] = copy.deepcopy(index_splits[it])

        # UPDATE THE RIGHT SIDE OF THE SPLIT INDEX WITH A NEW INT
        for i in index_r:

            index_splits[it+1][i] = it+1

        if (IG == 0):
            indexes = index_splits[it]
            break

```

The next part of the code appends the interval of the split node to the *intervals* array. Then, it updates the right interval of the left split and the left interval of the right split. If, for example, we are on the fifth iteration and split the node labeled by the integer 0, this part of the code will update the right interval of the 0<sup>th</sup> row and the left interval of the 5<sup>th</sup> row which will represent the new emergent node labeled by the integer 5. This  $\epsilon$  term is a trick I introduced, so I did not need to keep track of the four possibilities for inequalities  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ . We can include the splitting point on the left or right interval by adding or subtracting this term, which I made to be of the order of  $10^{-5}$ .

```
# ADD AND UPDATE THE INTERVALS FOR THE NEW NODE
intervals.append(copy.deepcopy(intervals[max_pos]))

intervals[max_pos][max_j][1] = self.X[max_i][max_j] \
    - self.epsilon*(-1)**sign_type

intervals[it+1][max_j][0] = self.X[max_i][max_j] \
    - self.epsilon*(-1)**sign_type

indexes = index_splits[it+1]

intervals = np.array(intervals)

# COMPUTE THE OUTCOMES OF EACH NODE
outcomes = self.outcome(indexes)

if self.results == "yes":

    self.print_results(intervals, outcomes)

return(intervals, outcomes)
```

This next function assigns an outcome to each node by counting the number of times the labels zero, one, and two appear on the node. It reads which node each index is classified into by accessing the last row of the *index splits* matrix.

```
def outcome(self, indexes, X_intervals, Y_intervals):

    outcomes = np.zeros((np.max(indexes)+1), dtype=int)

    for n in range(np.max(indexes)+1):

        outcome_n = np.array([self.Z[i] for i in \
            range(len(indexes)) if indexes[i]== n])
```

```

Z0_count = np.count_nonzero(outcome_n == 0)
Z1_count = np.count_nonzero(outcome_n == 1)
Z2_count = np.count_nonzero(outcome_n == 2)

if (Z0_count >= Z1_count and Z0_count >= Z2_count):
    outcomes[n] = 0

elif (Z1_count >= Z0_count and Z1_count >= Z2_count):
    outcomes[n] = 1

else:
    outcomes[n] = 2

return(outcomes)

def print_results(self,intervals, outcomes):

    intervals = np.round(intervals,6)

    print("          RESULTS")
    for i in range(len(outcomes)):
        print("-----")
        print("          Leaf n° ",i+1, "--> outcome = ",
              int(outcomes[i]))
        print("-----")
        for j in range(len(intervals[i])):
            print("          ",intervals[i][j][0],
                  "< X",j+1,"<",intervals[i][j][1])

```

The predict function takes some testing data as input given by the variable X. For each testing point, it runs through all the intervals stored in the *intervals* array and assigns to the point the outcome of the node associated to the interval where it falls.

```

def predict(self,X):

    intervals, outcomes = self.main_part()

    Z = np.zeros((len(X),1))

```

```

for i in range(len(Z)):

    for splits in range(len(intervals)):

        count = 0

        for var in range(len(intervals[0])):

            if (intervals[splits][var][0] < X[i][var] \
            and X[i][var] < intervals[splits][var][1]):

                count = count + 1

            if (count == len(intervals[0])):

                Z[i] = outcomes[splits]
                break

return(np.int_(Z))

```

Last, we have the accuracy function, which is no different from the functions of other algorithms. It subtracts the prediction and outcome arrays and counts the number of elements equal to zero on that array.

```

def accuracy(self,X,Z):

    Z = np.reshape(Z,(len(Z),1))

    test_values = len(Z)

    error_array = self.predict(X) - Z
    right_pred = np.count_nonzero(error_array == 0)

    return(100*right_pred/test_values)

```

Next, we have the plot function, which is similar to those of other classification algorithms. We will use this function to plot the results of applying the CART model to the 2D Iris data collection and to the Iris dataset modified by the PCA algorithm.

```
[2]: def PLOT_2D(X,Z,iterations):

    # CREATING TESTING GRID
    x_grid, y_grid = np.meshgrid(np.arange(np.min(X[:,0])-0.3,
                                             np.max(X[:,0])+0.3,0.05),

```

```

        np.arange(np.min(X[:,1])-0.3,
                   np.max(X[:,1])+0.3,0.05))

data = np.array([[x_grid[i][j],y_grid[i][j]] \
    for i in range(len(x_grid)) \
    for j in range(len(x_grid[0]))])

# USING THE K-NN MODEL
reg = CART(X,Z,iterations,"no")
z_grid = reg.predict(data)
z_grid= np.reshape(z_grid,(len(x_grid),len(x_grid[0])))

z_grid[-1][0], z_grid[-1][1],z_grid[-1][2] = 0,1,2

plt.title("CART for "+str(iterations)+" iterations")

# PLOT
color=["blue","green","red"]

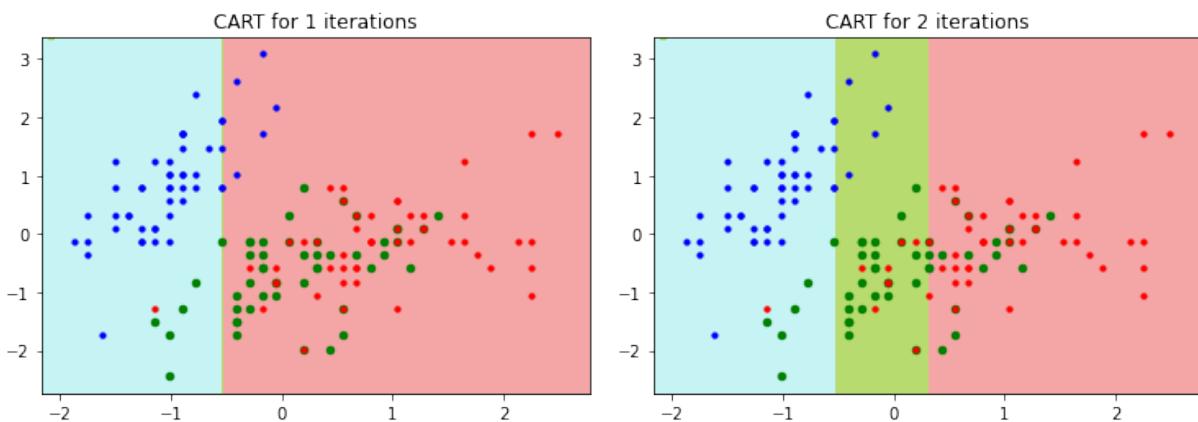
s = [10,20,10]
zorder = [2,2,3]

for i in range(len(X)):

    plt.scatter(X[i,0],X[i,1],color=color[Z[i]],
                s = s[Z[i]],zorder=zorder[Z[i]])

plt.contourf(x_grid,y_grid,z_grid,2,colors=["paleturquoise",
                                             "yellowgreen","lightcoral"],alpha=0.7)

```



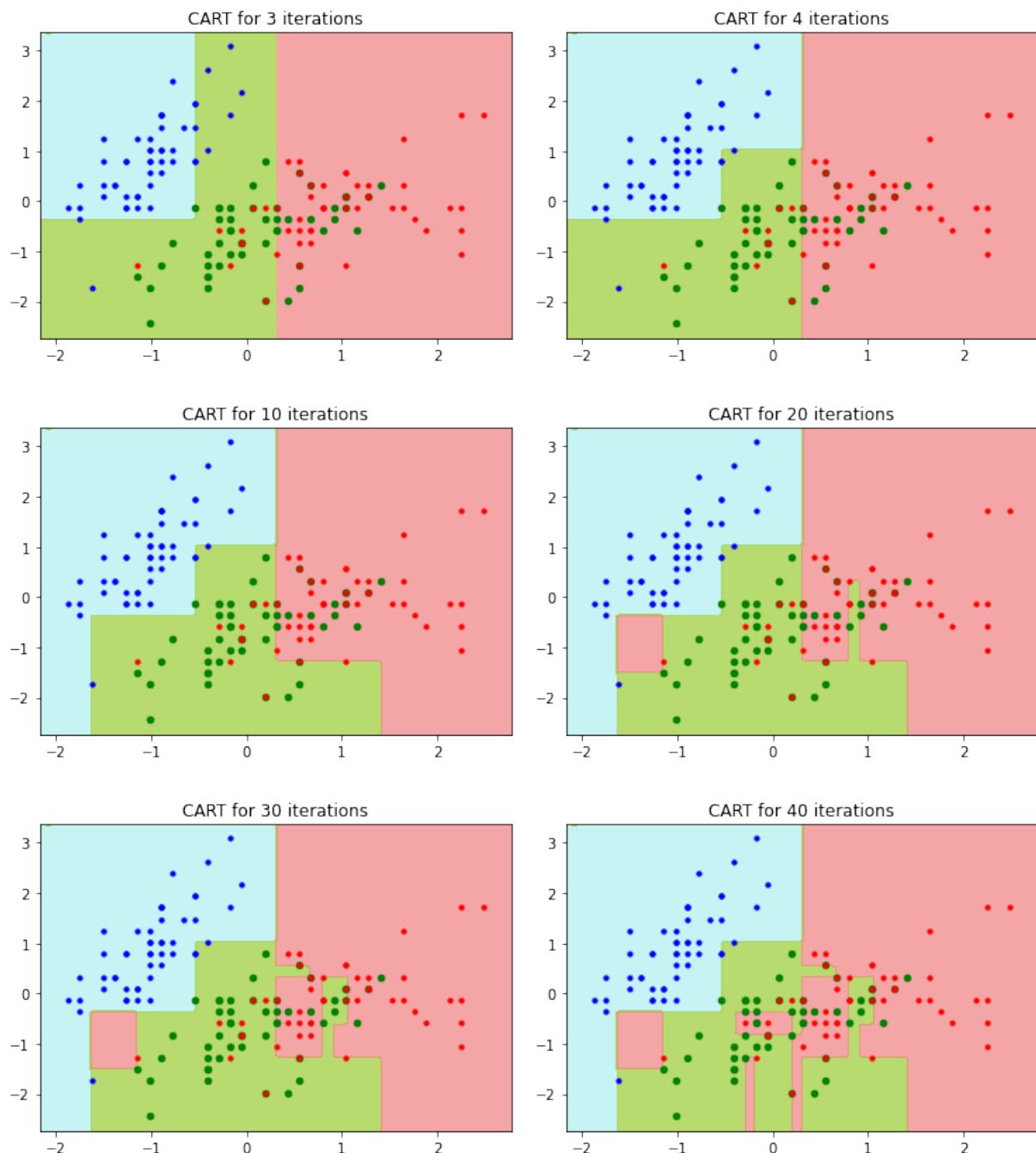


Figure 7.5: A plot of the CART results on the first two variables of the Iris dataset.

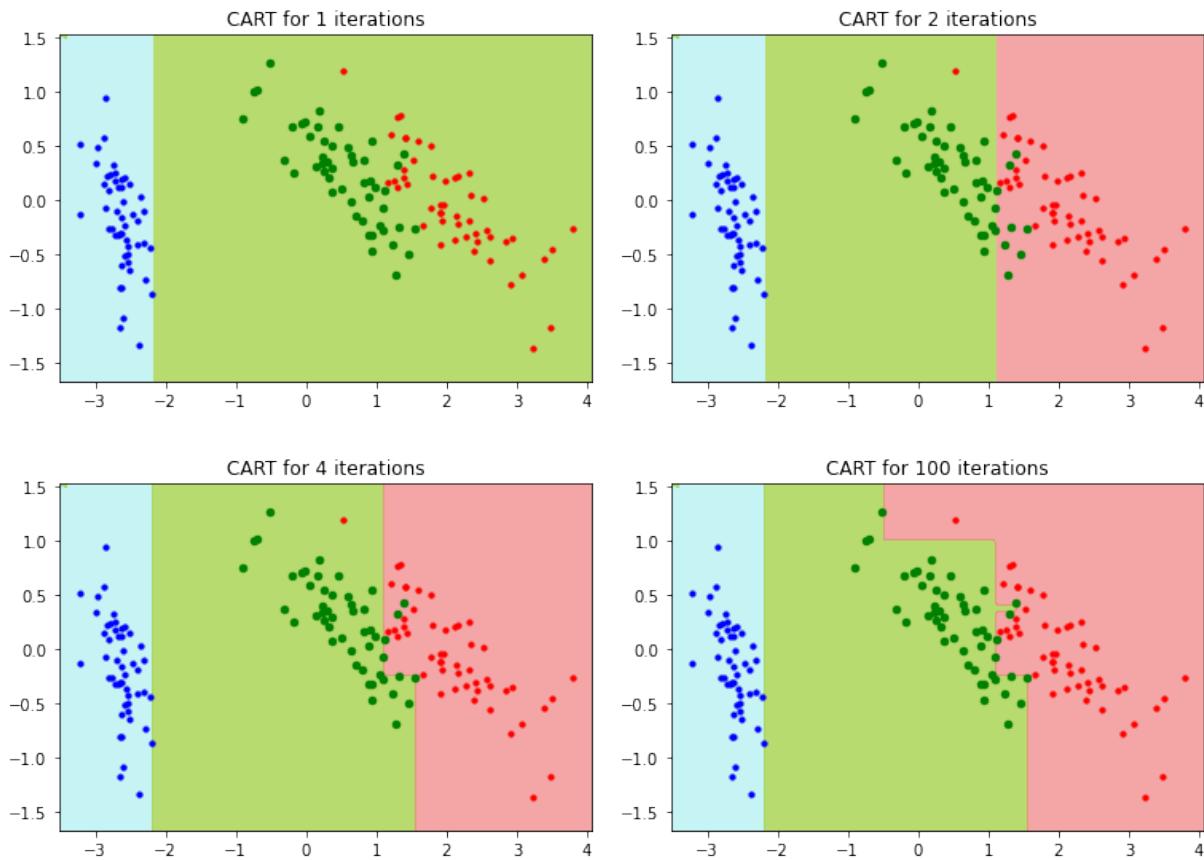


Figure 7.6: A plot of the CART results on the Iris dataset modified by the PCA algorithm.

To test that the algorithm works properly, we call the accuracy function on the training data for the 2D Iris data collection with a high number of iterations.

```
[3]: reg = CART(X,Z,105,"no")
print("Accuracy on training data: ",
      reg.accuracy(X[:,0], X[:,1],Z))
```

```
Accuracy on training data: 92.66666666666667
```

Since the data collection is composed of 150 data points, an accuracy of 92.66% means 11 points wrongly predicted. If we go to any plot in Figure 7.5, we can see ten overlaps between red and green data points, which explains ten of the wrong predictions as the model either predicts them as red or green. As I was missing one of the points, I built a small piece of code to count the overlaps and found out that there is a quadruple overlap at  $X = -0.053$  and  $Y = -0.82$ , which explains that missing point.

```
[4]: unique_values, count = np.unique(X, axis=0, return_counts=True)
unique_values = np.array([unique_values[i] \
                        for i in range(len(count)) if count[i] > 2])
```

```

count = np.array([count[i] \
                 for i in range(len(count)) if count[i] > 2])

for j in range(len(count)):
    outcome = []
    for i in range(len(X)):

        if (unique_values[j][0] == X[i][0] and \
            unique_values[j][1] == X[i][1]):

            outcome.append(Z[i])

    print(count[j]," points at [X,Y]=",unique_values[j],
          " with outcomes",outcome)

```

```

3 points at [X,Y]= [-0.90068117  1.70959465] with outcomes [0, 0, 0]
4 points at [X,Y]= [-0.05250608 -0.82256978] with outcomes [1, 1, 2, 2]
3 points at [X,Y]= [ 0.79566902 -0.13197948] with outcomes [2, 2, 2]
3 points at [X,Y]= [1.03800476  0.09821729] with outcomes [1, 1, 2]
3 points at [X,Y]= [1.2803405   0.09821729] with outcomes [1, 2, 2]

```

The last step is to build the cross-validation function for the CART algorithm and use it on the whole data collection.

```

[5]: from sklearn.model_selection import KFold

def K_fold_cross_validation_CART(X,Y,node_levels,iterations):

    test_accuracy = []
    train_accuracy = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])
            X_test = np.array([X[i] for i in test_index])
            Y_test = np.array([Y[i] for i in test_index])

```

```

reg = CART(X_train,Y_train,node_levels, "no")

train_acc = reg.accuracy(X_train,Y_train)
test_acc = reg.accuracy(X_test,Y_test)

test_accuracy.append(test_acc)
train_accuracy.append(train_acc)

return(np.mean(test_accuracy),np.mean(train_accuracy))

```

I called the cross-validation function for the first ten random states as the following algorithm was computationally expensive, and I wanted to compare the results. This was done for a different number of splits (iterations of the CART algorithm).

```

[6]: test_accuracy = np.zeros((6))
train_accuracy = np.zeros((6))

for i in range(2,8):
    test_accuracy[i-2],\
    train_accuracy[i-2] = K_fold_cross_validation_CART(X,Z,i,10)

test_accuracy_2 = np.zeros((10))
train_accuracy_2 = np.zeros((10))

for i in range(2,12):
    test_accuracy_2[i-2],\
    train_accuracy_2[i-2] = K_fold_cross_validation_CART(X_PCA,Z,i,10)

```

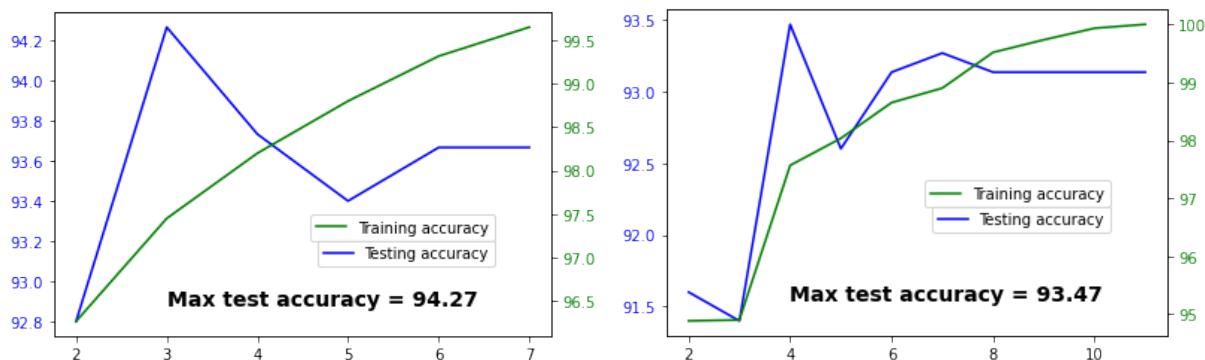


Figure 7.7: Accuracy results on the Iris dataset and the Iris dataset modified by the PCA algorithm.

As we can see, when we consider all the data collection variables, the accuracy improves drastically to 94.27%. Even when we take the vital information out of these variables and

reduce the dimensions of the problem to 2D with PCA, we reach an accuracy of 93.47%.

Since we will work only with two of the three classes or outcomes of the Iris dataset on the gradient-boosted trees, we will also compute the average accuracy for this configuration. The accuracy should be slightly reduced since we remove the least overlapped class (blue points).

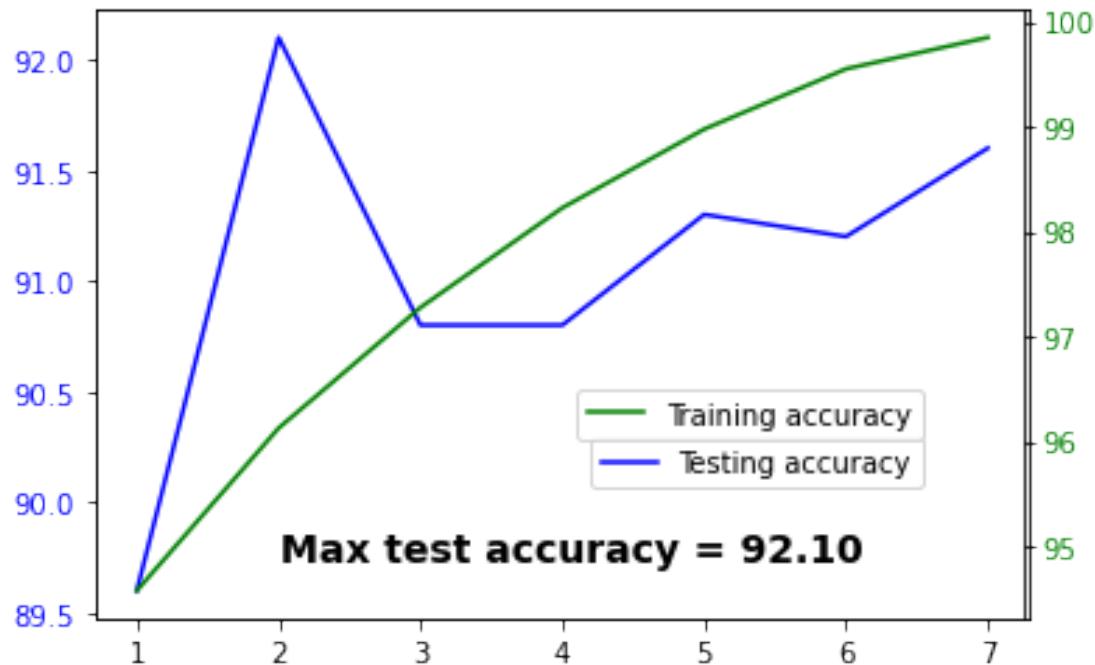


Figure 7.8: Accuracy results of the CART model on two of the classes of the Iris dataset (green and red points).

## 7.2 Random forest [Supervised]

### 7.2.1 Mathematical derivation

[29] Random forest is an algorithm that tries to remove some of the dependency that the CART model has on training data. To accomplish this, we will build different CART trees to make predictions on our data and average the results.

The process is the following: we will draw data randomly from our training data set; we do not care if we draw the same sample once or more times; this is called *bootstrapping*. The idea is to make the process as random as it can be. We will also cut down the information on these data points we draw and take only some of the variables, which will be decided randomly before the drawing. Once we have generated new random training data, we build the CART tree and then make predictions on the testing data. We will do the same process for each tree generated via bootstrapping and then assign the most predicted outcome among the trees to each testing point.

1	2	3	4	5
		$X_1$		
		$X_2$		
		$X_3$		
		$X_4$		

Table 7.1: An example of a data collection composed of five data points and four variables.

1	1	2	4	4
		$X_2$		
		$X_3$		

Table 7.2: Example of a random tree from the random forest.

2	2	3	4	5
		$X_3$		
		$X_4$		

Table 7.3: Another example of a random tree from the random forest.

## 7.2.2 Algorithm implementation

The implementation of the random forest algorithm is straightforward as it makes use of the CART implementation.

```
[1]: import itertools
import random

class random_forest():

    def __init__(self,X,Z,tree_number,splits,var):

        self.X = X
        self.Z = Z
        self.tree_number = tree_number
        self.splits = splits
        self.var = var
```

The first function of the code is the *accuracy* function. First, it will create an array containing the index of the different variables; in the case of the Iris data collection, this array will store the values 0,1,2 and 3. Next, it generates all the combinations of size *var* out of these variables. Usually, the variables involved in the random forest trees are picked randomly. However, since the number of trees I used was not too high because it was computationally expensive, I decided to use the different variable combinations evenly. The next step was to generate the new training data using the *generate-data* function. Then we call the CART algorithm, make predictions on the testing data (adapted only to those variables picked for the training data), and finally append the outcomes on a matrix labeled as *pred*. The last step picks the most frequent prediction on each column (which will be associated with each testing point) and assigns those outcomes to the points.

```
def accuracy(self,X_test,Z_test):

    pred = []
    outcome = np.zeros(len(X_test))

    elements = np.arange(0,len(self.X[0]))
    comb = list(itertools.combinations(elements,self.var))

    for i in range(self.tree_number):

        variables = comb[i%len(comb)]

        X_new, Z_new = self.generate_data(self.X,self.Z,variables)
```

```

X_test_new = np.array([[X_test[m][j] for j in variables]\n                      for m in range(len(X_test))])\n\nreg = CART(X_new,Z_new,self.splits,"no")\npred.append(reg.predict(X_test_new))\n\npred = np.array(pred)\n\nfor i in range(len(X_test)):\n\n    zero_count = np.count_nonzero(pred[:,i] == 0)\n    one_count = np.count_nonzero(pred[:,i] == 1)\n    two_count = np.count_nonzero(pred[:,i] == 2)\n\n    counts = np.array([zero_count, one_count, two_count])\n\n    outcome[i] = np.where(counts == np.max(counts))[0][0]\n\nerror = Z_test-outcome\nacc = 100*np.count_nonzero(error == 0)/len(X_test)\nreturn(acc)

```

The *generate-data* function picks N random integers on the range  $(0, N - 1)$  where N is the number of training points. Then, it generates the training data for the variables declared as the function's input.

```

def generate_data(self,X,Z,variables):\n\n    generated_X = []\n    generated_Z = []\n\n    #PICK N RANDOM POINTS (CAN BE REPEATED)\n    for i in range(len(X)):\n\n        c = random.randrange(len(X))\n\n        generated_X.append([X[c][j] for j in variables])\n        generated_Z.append(Z[c])\n\n    generated_X = np.array(generated_X)\n    generated_Z = np.array(generated_Z)\n\n    return(generated_X,generated_Z)

```

The next step was to use the cross-validation function for the random forest.

```
[2]: from sklearn.model_selection import KFold

def K_fold_cross_validation_FOREST(X,Y,splits,iterations,tree_number,var):

    test_accuracy = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])
            X_test = np.array([X[i] for i in test_index])
            Y_test = np.array([Y[i] for i in test_index])

            reg = random_forest(X_train,Y_train,tree_number,splits,var)

            acc = reg.accuracy(X_test,Y_test)

            test_accuracy.append(acc)

    return(np.mean(test_accuracy))
```

I called the cross-validation function for three different configurations. In the first one, the trees of the random forest are generated by picking variable pairs. The second one was composed of trees using variable triplets, and finally, the third configuration used all four variables of the Iris data collection.

```
[3]: test_acc_2 = np.zeros((7))
test_acc_3 = np.zeros((7))
test_acc_4 = np.zeros((7))

for i in range(2,9):

    test_acc_2[i-2] = K_fold_cross_validation_FOREST(X,Z,splits=i,
        iterations=10,tree_number=50,var=2)
    test_acc_3[i-2] = K_fold_cross_validation_FOREST(X,Z,splits=i,
        iterations=10,tree_number=50,var=3)
    test_acc_4[i-2] = K_fold_cross_validation_FOREST(X,Z,splits=i,
        iterations=10,tree_number=50,var=4)
```

As we can see, when using the random forest implementation for the first ten random states and using 50 different trees for each state, we find that the accuracy is improved by more than 1.5% with respect to the regular CART algorithm, reaching a maximum of 95.80% accuracy when using variable triplets. The results of the three different configurations are satisfactory as they improve the accuracy compared to the CART model.

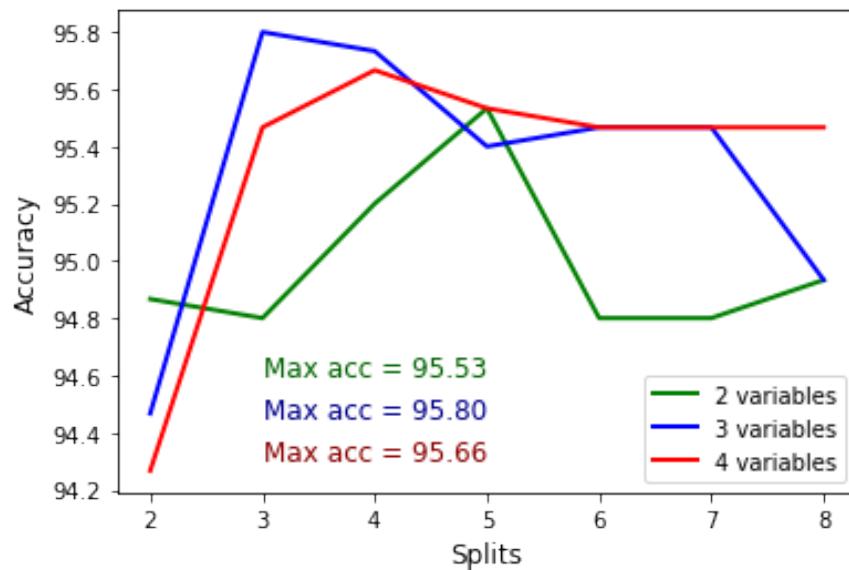


Figure 7.9: Accuracy results of the random forest implementation.

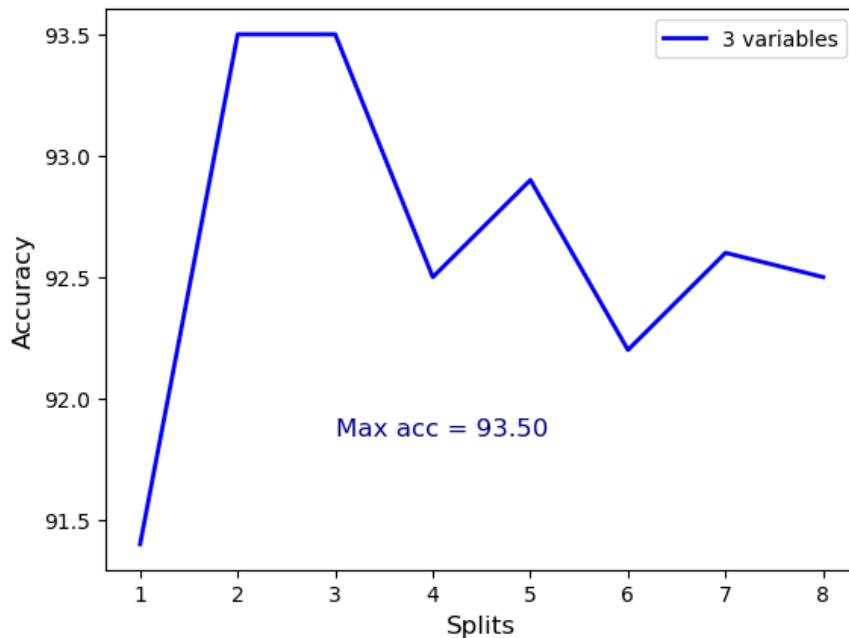


Figure 7.10: Accuracy results of the random forest implementation on two of the class types of the Iris dataset (green and red points).

## 7.3 Gradient boosted classification trees

### [Supervised]

#### 7.3.1 Mathematical derivation

[23][26][42] Before jumping into the equations behind this algorithm, let us introduce a simple binary data collection to walk through the different steps more clearly.

Data point	Age	Daily hours at home	Plays videogames
1	17	17	1
2	26	12	0
3	33	20	1
4	49	15	0
5	64	19	0

Table 7.4: An example of a data collection composed of five data points, two variables, and a binary outcome (where an outcome of one means that the person plays videogames, and an outcome of zero means the opposite).

As in other algorithms, we need a function to quantify the difference between the model's predictions for the training data and the outcomes. In order to do this, we introduce the log-likelihood function, which is given by:

$$\mathbb{L} = - \sum_i [y_i \ln(P_i) + (1 - y_i) \ln(1 - P_i)] \quad (7.3)$$

Where  $y_i$  are the different outcomes of the training data and  $P_i$  are the predictive probabilities, which gives the probability of a point having an outcome of one or zero (depending on whether these values are closer to zero or one). As we can see, for a given  $y_k = 0$ , the first term will cancel, and the second term will be equal to  $\ln(1 - P_k)$ , which will be the largest for  $P_k = 0$  and hence contributes to minimizing the log-likelihood function due to the minus sign. In the same way, this function is minimized for  $y_k = 1$  when  $P_k$  is closer to one. Since this will be an iterative model, on our first iteration, we will set  $P_i = P^{(0)}$ , where  $P^{(0)}$  will be a global probability that tells us the probability of having an outcome of one or zero based on the outcome density of the data collection. Hence, we will start writing the log-likelihood for the first iteration as:

$$\mathbb{L}^{(0)} = - \sum_i [y_i \ln(P^{(0)}) + (1 - y_i) \ln(1 - P^{(0)})] \quad (7.4)$$

Apart from the predictive probability, we will also introduce the concept of odds.

$$odds = \frac{P}{1 - P} \quad (7.5)$$

Using the Sigmoid function to write the predictive probability in terms of the odds will also be helpful.

$$P = \frac{odds}{1 + odds} = \frac{1}{1 + odds^{-1}} = \frac{1}{1 + e^{-\ln(odds)}} = \frac{1}{1 + e^{-\gamma}} \quad (7.6)$$

Where  $\gamma = \ln(odds)$ . Now, let's write the log-likelihood as a function of the odds, as it will be easier to work with this quantity than with predictive probabilities.

$$\begin{aligned} \mathbb{L}^{(0)} &= - \sum_i [y_i \ln(P^{(0)}) + (1 - y_i) \ln(1 - P^{(0)})] = - \sum_i \left[ y_i \ln\left(\frac{P^{(0)}}{1 - P^{(0)}}\right) + \ln(1 - P^{(0)}) \right] \\ &= - \sum_i \left[ y_i \ln(odds^{(0)}) + \ln\left(1 - \frac{1}{1 + e^{-\gamma^{(0)}}}\right) \right] = - \sum_i \left[ y_i \gamma^{(0)} + \ln\left(\frac{e^{-\gamma^{(0)}}}{1 + e^{-\gamma^{(0)}}}\right) \right] \\ &= - \sum_i \left[ y_i \gamma^{(0)} + \ln\left(\frac{1}{1 + e^{\gamma^{(0)}}}\right) \right] = - \sum_i \left[ y_i \gamma^{(0)} - \ln(1 + e^{\gamma^{(0)}}) \right] \end{aligned} \quad (7.7)$$

Now, if we compute the derivative with respect to  $\gamma^{(0)}$ :

$$\frac{\partial \mathbb{L}^{(0)}}{\partial \gamma^{(0)}} = - \sum_i \left[ y_i - \frac{e^{\gamma^{(0)}}}{1 + e^{\gamma^{(0)}}} \right] = - \sum_i \left[ y_i - \frac{1}{1 + e^{-\gamma^{(0)}}} \right] = \sum_i (P^{(0)} - y_i) \quad (7.8)$$

Making the partial derivative equal to zero, we find that:

$$P^{(0)} = \frac{\sum_i y_i}{N} \quad (7.9)$$

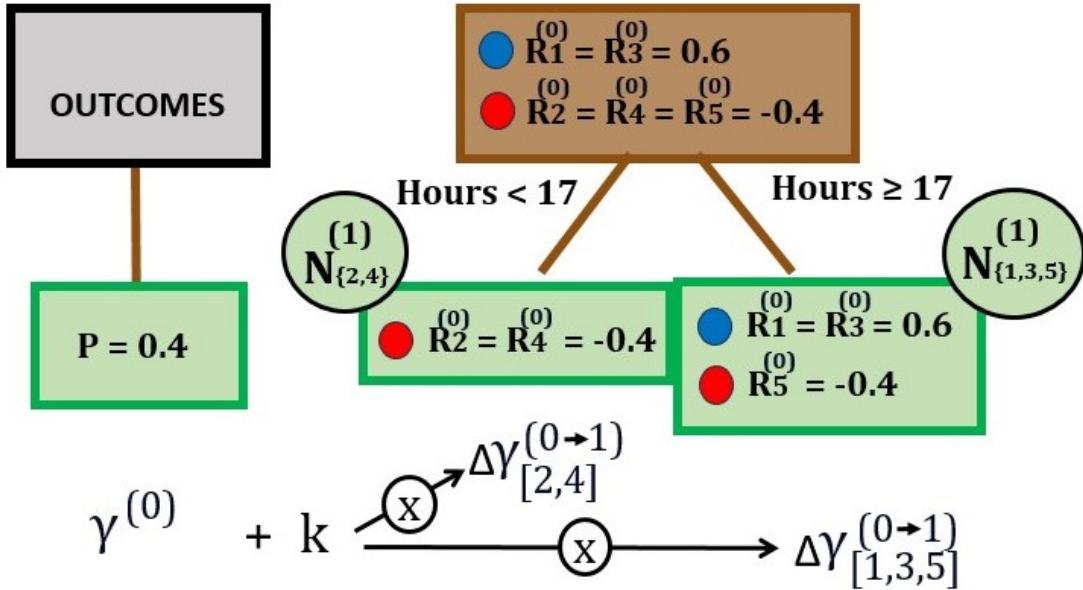
As expected, the predictive probability will be an average of the outcomes, so for example, for our data collection, we find that  $P^{(0)} = 2/5 = 0.4$ . If we plug this value into Equation 7.5, we find that  $odds^{(0)} = 0.66$ , meaning that we have three negative outcomes for every two positive outcomes. We also find  $\gamma^{(0)} = -0.405$ . This will be our initial estimation for the  $\gamma$  parameter, but as the name of the algorithm suggests, we will combine gradient descent and the CART algorithm to update this value.

We introduce the concept of *pseudo-residuals*, which is the difference between the outcomes and the predictive probabilities of the different data points. As one can imagine, having arrived at an initial global probability  $P^{(0)} = 0.4$ , some data points will have higher residuals than others. This means that the data points will need different corrections to their  $\gamma$  parameters and hence to their predictive probabilities.

This hints towards the next step of this algorithm. We will split the data points with a tree where we will input the pseudo-residuals as the outcomes of the points and then update  $\gamma^{(0)}$  by a different amount for each leaf node. First, we start by computing the pseudo-residuals of the base iteration labeled as  $R_i^{(0)}$ .

Data point	Age	Hours	Videogames	$R_i^{(0)}$
1	17	17	1	0.6
2	26	12	0	-0.4
3	33	20	1	0.6
4	49	15	0	-0.4
5	64	19	0	-0.4

Table 7.5: Update of the contents of Table 7.4 with the pseudo-residuals.



For simplicity, we will choose a CART tree composed only of a couple of leaf nodes. Usually, this algorithm's optimal number of leaves ranges from 2 to 8. We have an initial estimation of  $\gamma$  given by  $\gamma^{(0)}$ , which is computed from the predictive probability of the data collection. Then we can update this parameter by creating a CART tree with the pseudo-residuals and generate the updates for the  $\gamma$  parameter for each leaf node given by  $\Delta\gamma_{L_1}^{(0 \rightarrow 1)}$  and  $\Delta\gamma_{L_2}^{(0 \rightarrow 1)}$  where the upper index refers to the transition between iterations. The lower index refers to a list  $L_k$  storing the indexes of each leaf labeled by  $k$ . The only thing we have to do now is to obtain the equation of the  $\gamma$  updates in terms of the pseudo-residuals of the leaf nodes. To do this, we will follow the same process as before; we will find the updates of  $\gamma$  to minimize the log-likelihood function. Now we will have two contributions to this function for each of the leaf nodes labeled as  $N_{S_1}^{(1)}$  and  $N_{S_2}^{(1)}$  on the plot; similarly to the gamma values, the lower index refers to a set  $S_k$  composed of the indexes of the leaf node  $k$ .

$$\mathbb{L}_1^{(1)} = - \sum_{i \in S_1} \left[ y_i (\gamma^{(0)} + k \Delta\gamma_i^{(0 \rightarrow 1)}) - \ln \left( 1 + e^{(\gamma^{(0)} + k \Delta\gamma_i^{(0 \rightarrow 1)})} \right) \right] \quad (7.10)$$

$$\mathbb{L}_2^{(1)} = - \sum_{i \in S_2} \left[ y_i (\gamma^{(0)} + k\Delta\gamma_i^{(0 \rightarrow 1)}) - \ln \left( 1 + e^{(\gamma^{(0)} + k\Delta\gamma_i^{(0 \rightarrow 1)})} \right) \right] \quad (7.11)$$

Finding these quantities to minimize the log-likelihood can be troublesome; hence, we will approximate it with a Taylor expansion. We will use a generic expression for the log-likelihood function to remove some of the index and make it more readable.

$$\mathbb{L} = \mathbb{L} |_{\gamma=\gamma^{(0)}} + \frac{\partial \mathbb{L}}{\partial \gamma} |_{\gamma=\gamma^{(0)}} \Delta\gamma + \frac{1}{2} \frac{\partial^2 \mathbb{L}}{\partial \gamma^2} |_{\gamma=\gamma^{(0)}} \Delta\gamma^2 \quad (7.12)$$

Now, if we compute the derivative with respect to  $\gamma$  (keeping in mind that  $\Delta\gamma = \gamma - \gamma^{(0)}$ ) and make it equal to zero, we find:

$$\frac{\partial \mathbb{L}}{\partial \gamma} = \frac{\partial \mathbb{L}}{\partial \gamma} |_{\gamma=\gamma^{(0)}} + \frac{\partial^2 \mathbb{L}}{\partial \gamma^2} |_{\gamma=\gamma^{(0)}} \Delta\gamma = 0 \quad (7.13)$$

From which it is straightforward to find that:

$$\Delta\gamma = - \frac{\frac{\partial \mathbb{L}}{\partial \gamma} |_{\gamma=\gamma^{(0)}}}{\frac{\partial^2 \mathbb{L}}{\partial \gamma^2} |_{\gamma=\gamma^{(0)}}} \quad (7.14)$$

After taking a look at Equation 7.8. we realize that the numerator is nothing but the sum of the pseudo-residuals. Let us compute the second derivative to see what it looks like.

$$\begin{aligned} \frac{\partial^2 \mathbb{L}}{\partial \gamma^2} |_{\gamma=\gamma^{(0)}} &= - \frac{\partial^2}{\partial \gamma^2} \sum_i \left[ y_i \gamma - \ln(1 + e^\gamma) \right] |_{\gamma=\gamma^{(0)}} = \sum_i \frac{\partial}{\partial \gamma} \left[ -y_i + \frac{e^\gamma}{1 + e^\gamma} \right] |_{\gamma=\gamma^{(0)}} = \\ &\sum_i \frac{e^{\gamma^{(0)}}}{(1 + e^{\gamma^{(0)}})^2} = \sum_i \frac{e^{\gamma^{(0)}}}{(1 + e^{\gamma^{(0)}})} \frac{1}{(1 + e^{\gamma^{(0)}})} = \sum_i P^{(0)} (1 - P^{(0)}) \end{aligned} \quad (7.15)$$

Moreover, we can finally write:

$$\Delta\gamma = \frac{\sum_i R_i^{(0)}}{\sum_i P^{(0)} (1 - P^{(0)})} \quad (7.16)$$

In general, for any given iteration, we can write the update of the  $\gamma$  parameter of each data point by the following equation:

$$\gamma_i^{(n+1)} = \gamma_i^{(n)} + k\Delta\gamma_i^{(n \rightarrow n+1)} = \gamma_i^{(n)} + k \frac{\sum_{j \in S_m} R_j^{(n)}}{\sum_{j \in S_m} P_j^{(n)} (1 - P_j^{(n)})} \quad (7.17)$$

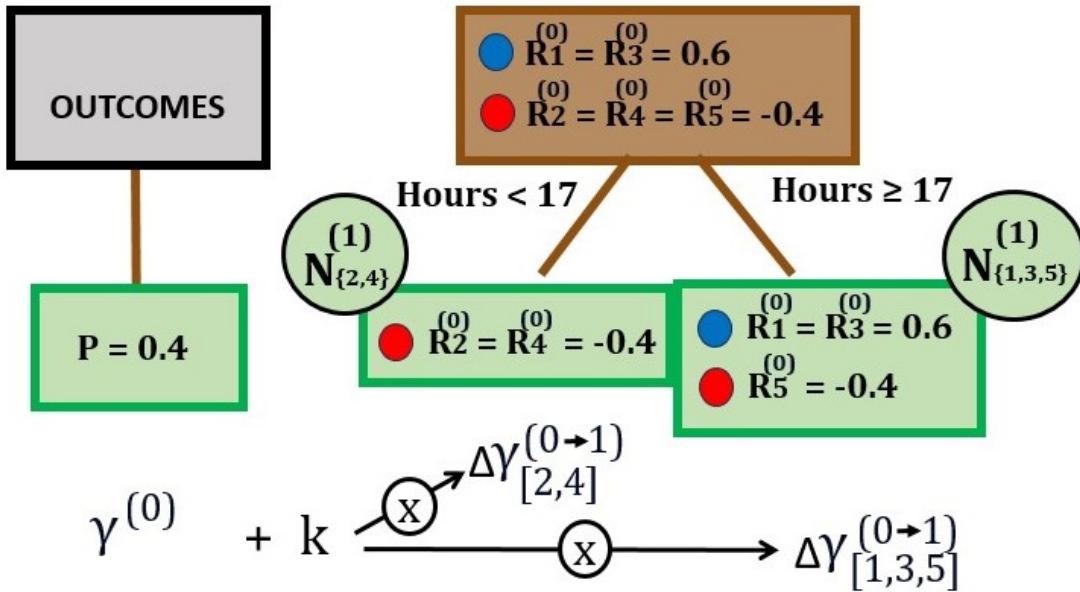
Where  $S_m$  is the set of indexes on the leaf node where the index  $i$  falls. The idea is that after updating the  $\gamma$  coefficients for each data point, we can compute their new predictive probabilities with Equation 7.6. This process will be repeated for the desired number of trees. Now, we will write the steps of the algorithm and then follow them for our example data collection.

## STEPS

- Initialize the model with  $\gamma^{(0)}$ , obtained computing first the predictive probability with Equation 7.9.
- For the desired number of iterations:
  - Apply the CART model on the residuals
  - Compute the correction of the  $\gamma$  parameter for each data point with the equation.

$$\gamma_i^{(n+1)} = \gamma_i^{(n)} + k \frac{\sum_{j \in S_m} R_j^{(n)}}{\sum_{j \in S_m} P_j^{(n)} (1 - P_j^{(n)})} \quad (7.18)$$

- Compute the new predictive probabilities  $P_i^{(n+1)}$ .
- Compute the new pseudo-residuals  $R_i^{(n+1)}$  to be used on the next iteration.



Let us follow these steps for our example; on the first iteration, we have a tree with two leaf nodes, and the updates on the  $\gamma$  parameters read:

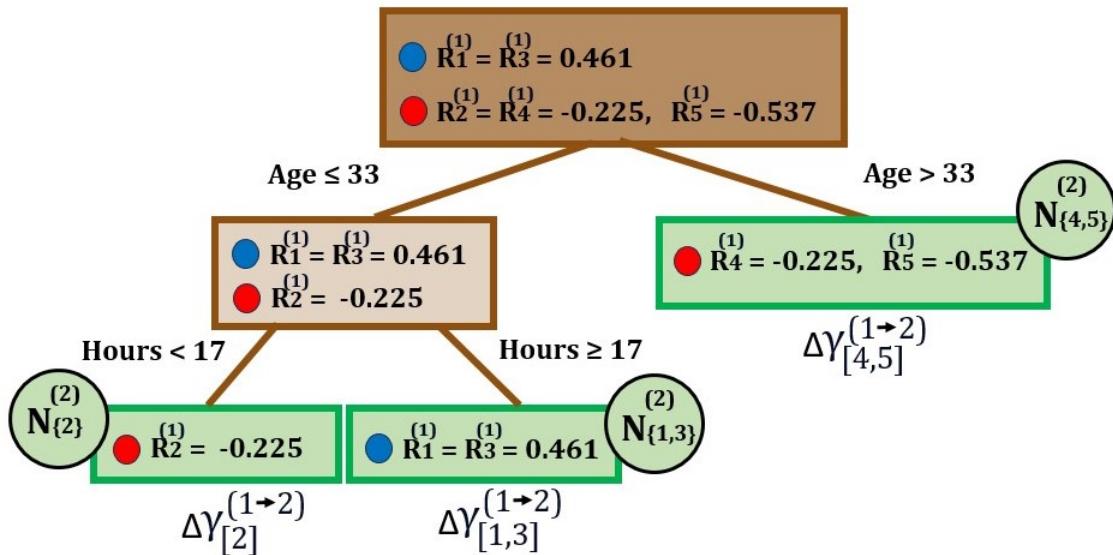
$$\Delta\gamma_{[2,4]}^{(0 \rightarrow 1)} = \frac{-0.4 - 0.4}{2 \times 0.4 \times (1 - 0.4)} = -1.66 \quad (7.19)$$

$$\Delta\gamma_{[1,3,5]}^{(0 \rightarrow 1)} = \frac{0.6 + 0.6 - 0.4}{3 \times 0.4 \times (1 - 0.4)} = 1.11 \quad (7.20)$$

Now, taking the learning rate  $k = 0.5$ , we can update all the necessary parameters, which can be found in the following table:

Data point	$y_i$	$\gamma_i^{(1)}$	$P_i^{(1)}$ (Eq. 7.6)	$R_i^{(1)} = y_i - P_i^{(1)}$
1	1	$-0.405 + 0.5*1.11 = 0.15$	0.537	0.461
2	0	$-0.405 - 0.5*1.66 = -1.235$	0.225	-0.225
3	1	$-0.405 + 0.5*1.11 = 0.15$	0.537	0.461
4	0	$-0.405 - 0.5*1.66 = -1.235$	0.225	-0.225
5	0	$-0.405 + 0.5*1.11 = 0.15$	0.537	-0.537

As we can see, four out of the five points are classified correctly with a single iteration. We will build a tree with one more leaf in the second iteration.



We want to split the negative and positive values of the residuals on different leaves. We need at least two splits within the trees to split them completely. There is no single value on the *Age* or *Hours* variables on which we can completely split the positive and negative values. Next, we compute the update of the  $\gamma$  parameters for the second iteration.

$$\Delta\gamma_{[2]}^{(1 \rightarrow 2)} = \frac{-0.225}{0.225 * (1 - 0.225)} = -1.290 \quad (7.21)$$

$$\Delta\gamma_{[1,3]}^{(1 \rightarrow 2)} = \frac{0.461 + 0.461}{2 * 0.537 * (1 - 0.537)} = 1.854 \quad (7.22)$$

$$\Delta\gamma_{[4,5]}^{(1 \rightarrow 2)} = \frac{-0.225 - 0.537}{0.537 * (1 - 0.537) + 0.225 * (1 - 0.225)} = -1.801 \quad (7.23)$$

We get the following results by updating the gamma and predictive probability values of each data point.

Data point	$y_i$	$\gamma_i^{(2)}$	$P_i^{(2)}$ (Eq. 7.6)	$R_i^{(2)} = y_i - P_i^{(2)}$
1	1	$0.15 + 0.5*1.854 = 1.08$	0.746	0.254
2	0	$-1.235 - 0.5*1.290 = -1.88$	0.132	-0.132
3	1	$0.15 + 0.5*1.854 = 1.08$	0.746	0.254
4	0	$-1.235 - 0.5*1.801 = -2.135$	0.106	-0.106
5	0	$0.15 - 0.5*1.801 = -0.75$	0.321	-0.321

Now, all the points are correctly classified; those where the probabilities are smaller than 0.5 are classified as zero, and those with probabilities greater than 0.5 are classified as one. To make predictions on new data, we initialize the  $\gamma$  parameter with the value  $\gamma^{(0)}$ . Then, we check in which leaf node the test point falls for each tree and update the  $\gamma$  value accordingly. Once this is done, we compute the predictive probability and decide the outcome of the test point based on this result.

### 7.3.2 Algorithm implementation

Now, we proceed with the implementation of the gradient-boosted classification trees. First, I will introduce a variant of my CART implementation for classification adapted to continuous outcomes, which also computes each leaf node's  $\gamma$  coefficients. The core of the gradient-boosted classification trees will be contained in another class.

```
[1]: import math as math
import copy as copy
import numpy as np

class Regression_tree():
```

We will have the  $Z$  input variable storing the outcomes of the data (the pseudo-residuals) and another variable labeled as  $out$  which will contain the real labels of the data collection (zeros and ones).

```
def __init__(self,X,Z,out,iterations):

    self.X = X
    self.Z = Z
    self.out = out
    self.epsilon = 10**(-5)
    self.iterations = iterations
```

Since the outcomes of the tree will be continuous (real numbers), we cannot use the Gini factors. Instead, we will use the mean squared error. Since the prediction associated to a leaf node for regression trees is given by the mean of the outcomes among the leaf, we will compute the MSE of the left and right splits, using these mean values as the predictions. This function also computes the MSE of the complete set of indexes and then returns the difference between the MSE after and before the split. This value should be smaller than zero, as the MSE should be reduced when we generate new leaves.

```
def MSE(self,indexes,index_l,index_r):

    avrg_l = self.outcome_avrg(index_l)
    avrg_r = self.outcome_avrg(index_r)
    avrg_total = self.outcome_avrg(indexes)

    MSE_l_list = [(self.Z[i]-avrg_l)**2.0 for i in index_l]
    MSE_r_list = [(self.Z[i]-avrg_r)**2.0 for i in index_r]
    MSE_total_list = [(self.Z[i]-avrg_total)**2.0 for i in indexes]

    MSE_l = sum(MSE_l_list)
    MSE_r = sum(MSE_r_list)
```

```

MSE_total = sum(MSE_total_list)

MSE_change = MSE_l + MSE_r - MSE_total

return(MSE_change)

```

The *outcome\_avrg* function computes the average outcome of a specific list of indexes.

```

def outcome_avrg(self, indexes):

    outcome = [self.Z[i] for i in indexes]
    avrg_outcome = sum(outcome)/len(outcome)
    return(avrg_outcome)

```

This function is similar to its equivalent in the CART section. Instead of computing the information gain of a split on a particular  $(i, j)$  index pair, we compute the MSE difference between the configuration after splitting and before the splitting by calling the *MSE* function.

```

def index_split(self, indexes, i, j):

    index_l_1=[k for k in indexes if self.X[k][j] <= self.X[i][j]]
    index_r_1=[k for k in indexes if self.X[k][j] > self.X[i][j]]

    if (len(index_l_1)==0) or (len(index_r_1)==0):
        MSE_1 = 0.0
    else:
        MSE_1= self.MSE(indexes, index_l_1, index_r_1)

    index_l_2=[k for k in indexes if self.X[k][j] < self.X[i][j]]
    index_r_2=[k for k in indexes if self.X[k][j] >= self.X[i][j]]

    if (len(index_l_2)==0) or (len(index_r_2)==0):
        MSE_2 = 0.0
    else:
        MSE_2= self.MSE(indexes, index_l_2, index_r_2)

    if (MSE_1 <= MSE_2):

        return(index_l_1, index_r_1, MSE_1, 1)
    else:

        return(index_l_2, index_r_2, MSE_2, 2)

```

The *node\_choosing* function is also similar to its equivalent on the CART implementation. However, instead of looking for the maximum value of the information gain, we look for the minimum value of the MSE change, as we want the MSE of the split configuration to be the lowest.

```
def node_choosing(self, indexes):

    MSE_values = np.zeros((len(indexes), len(self.X[0])))

    for i in range(len(indexes)):
        for j in range(len(self.X[0])):

            MSE_values[i][j] = self.index_split(indexes,
                                                indexes[i], j)[2]

    min_i = np.where(MSE_values == np.min(MSE_values))[0][0]
    min_j = np.where(MSE_values == np.min(MSE_values))[1][0]
    index_l, index_r, MSE, sign_type = self.index_split(indexes,
                                                        indexes[min_i], min_j)

    return(index_l, index_r, MSE, sign_type, min_j, indexes[min_i])
```

The *main\_part* function has a couple of minor changes in the code; the variables storing the indexes associated with maximum values are changed since now we are looking for the minimum value of the MSE change. Moreover, the variables storing the information gain now store the mean squared error. Also, at the end of the function, we call for the *gamma-values* function, which computes each leaf node's  $\gamma$  coefficients.

```
def main_part(self):

    index_splits = np.zeros((self.iterations+1,
                            len(self.X)), dtype = int)

    intervals = [[[min(self.X[:, j])- 1.0, max(self.X[:, j]) + 1.0]\n
                  for j in range(len(self.X[0]))]]

    for it in range(self.iterations):

        MSE_list = np.zeros((it + 1))

        # TEST ALL THE NODES FOR POTENTIAL SPLITS
        for n in range(it + 1):

            index = np.array([i for i in range(len(self.X))]\
```

```

        if index_splits[it,i]==n])

    index_l,index_r,MSE,sign_type,min_j,min_i = \
        self.node_choosing(index)

    MSE_list[n] = MSE

    # SPLIT THE NODE THAT RESULTS ON THE LARGEST IG
    min_pos = np.where(MSE_list == np.min(MSE_list))[0][0]

    index = np.array([i for i in range(len(self.X))\
        if index_splits[it,i]==min_pos])

    index_l,index_r,MSE,sign_type,min_j,min_i = \
        self.node_choosing(index)

    index_splits[it+1] = copy.deepcopy(index_splits[it])

    # UPDATE THE RIGHT SIDE OF THE SPLIT INDEX WITH A NEW INT
    for i in index_r:

        index_splits[it+1][i] = it+1

        if (MSE == 0):
            indexes = index_splits[it]
            break

    # ADD AND UPDATE THE INTERVALS FOR THE NEW NODE
    intervals.append(copy.deepcopy(intervals[min_pos]))

    intervals[min_pos][min_j][1] = self.X[min_i][min_j] \
        - self.epsilon*(-1)**sign_type

    intervals[it+1][min_j][0] = self.X[min_i][min_j] \
        - self.epsilon*(-1)**sign_type

    indexes = index_splits[it+1]

    intervals = np.array(intervals)

    # COMPUTE THE GAMMA VALUES OF EACH NODE
    gamma = self.gamma_values(indexes)

    return(intervals, gamma, indexes)

```

Lastly, we have the *gamma-values* function, which is exclusive to this algorithm. It generates a list containing the indexes of each leaf node; keep in mind that the *indexes* variable stores which leaf node each point is associated with (it contains integers ranging from 0 to k where k is the number of leaf nodes minus one). Then, we compute the residuals of each leaf node and the predictive probabilities. Last, we compute the  $\gamma$  values of each node.

```
def gamma_values(self, indexes):

    gamma = np.zeros((np.max(indexes)+1))

    for n in range(np.max(indexes)+1):

        index = np.array([i for i in range(len(indexes))\
                          if indexes[i] == n])

        Pseudo_res = np.array([self.Z[i] for i in index])
        outcomes = np.array([self.out[i] for i in index])
        Prob = outcomes - Pseudo_res
        one_minus_prob = np.ones((len(Prob))) - Prob

        gamma[n] = np.sum(Pseudo_res)/np.sum(Prob*one_minus_prob)

    return(gamma)
```

Next, we have the class for the gradient-boosted regression trees. It will have the number of trees, the number of splits on each tree, and the data collection stored on the variables X and Z as input.

```
[2]: class boosted_trees():

    def __init__(self,X,Z,tree_splits,tree_number):

        self.X = X
        self.Z = Z
        self.tree_splits = tree_splits
        self.tree_number = tree_number
```

This first function takes the gamma values of the leaf nodes of a tree, the intervals of each leaf node, and the variable values of a testing point as the input. Then, it outputs a gamma value for this testing point.

```
def tree_prediction(self,intervals,gamma,X_test):

    gamma_test = np.zeros(len(X_test))
```

```

for i in range(len(gamma_test)):

    for splits in range(len(intervals)):

        count = 0

        for var in range(len(intervals[0])):

            if (intervals[splits][var][0] < X_test[i][var] \
            and X_test[i][var] < intervals[splits][var][1]):

                count = count + 1

        if (count == len(intervals[0])):

            gamma_test[i] = gamma[splits]
            break

return(gamma_test)

```

The second function is the central part of the class. First, it initializes the values of the predictive probabilities  $P^{(0)}$  and gamma values  $\gamma^{(0)}$  for both the training and testing points. Then it starts iterating and doing the following steps for each iteration: first, it computes the residuals of the training data, then calls for the *Regression-tree* class with the residuals as the outcomes, which returns the gamma values of each leaf node. Since the *indexes* variable stores which leaf each training index belongs to, computing the  $\gamma$  changes for the training data is straightforward. For the testing points, we call for the *tree-prediction* function. Last, we compute the new  $\gamma$  factors and then update the predictive probabilities. After the last iteration is completed, we return the rounded values of the predictive probabilities for the testing data points.

```

def predict(self,X_test):

    P_0 = np.sum(self.Z)/len(self.Z)
    gamma_train = np.log(P_0/(1-P_0))*np.ones((len(self.X)))
    gamma_test = np.log(P_0/(1-P_0))*np.ones(len(X_test))

    P_train, P_test = P_0, P_0

    for i in range(self.tree_number):

        residuals = self.Z - P_train

```

```

reg = Regression_tree(self.X,residuals,self.Z,
                      self.tree_splits)
intervals,gamma,indexes = reg.main_part()

Delta_gamma_train = 0.1 * np.array([gamma[i] \
                                    for i in indexes])
Delta_gamma_test = 0.1* self.tree_prediction(intervals,
                                              gamma,X_test)

gamma_train = gamma_train + Delta_gamma_train
gamma_test = gamma_test + Delta_gamma_test

P_train = 1/(1+np.exp(-gamma_train))
P_test = 1/(1+np.exp(-gamma_test))

P_test = np.array([round(P_test[i]) \
                  for i in range(len(P_test))])
return(P_test)

```

Last, we have the accuracy function, which is no different from other algorithms.

```

def accuracy(self,X_test,Z_test):

    test_values = len(Z_test)

    error_array = self.predict(X_test) - Z_test
    right_pred = np.count_nonzero(error_array == 0)

    return(100*right_pred/test_values)

```

The next step is to build the cross-validation function for the gradient-boosted classification trees, which is identical to the function of other models. I will include it here in case someone wants to reproduce the results that will be shown next.

```

[3]: from sklearn.model_selection import KFold

def K_fold_cross_validation_boostedtree(X,Y,splits,
                                         iterations,tree_number):

    test_accuracy = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

```

```

for i, (train_index, test_index) in enumerate(KF.split(X)):

    X_train = np.array([X[i] for i in train_index])
    Y_train = np.array([Y[i] for i in train_index])
    X_test = np.array([X[i] for i in test_index])
    Y_test = np.array([Y[i] for i in test_index])

    reg = boosted_trees(X_train,Y_train,tree_splits = splits,
                         tree_number = tree_number)

    acc = reg.accuracy(X_test,Y_test)

    test_accuracy.append(acc)

return(np.mean(test_accuracy))

```

I decided to execute the cross-validation function for two of the classes of the Iris data collection (*Iris Versicolor* and *Iris Virginica*) as in previous CART models to compare the results (keep in mind that the labels of these classes need to be shifted to zero and one respectively). I executed the function for a different combination of splits and tree numbers. The maximum accuracy was 93.4%, close to the 93.5% provided by the Random Forest algorithm, improving the regular CART algorithm. A learning rate of 0.1 was used to obtain these results. It would be an excellent exercise to test different sizes of the learning rate and expand the number of splits and trees. However, since the objective of this book is not to find the most optimal configuration of the hyperparameters, we are satisfied to see that it performs better than the CART algorithm.

```

[4]: acc = np.zeros((9,18))

for splits in range(1,10):
    for trees in range(1,19):

        acc[splits-1][trees-1]= K_fold_cross_validation_boostedtree(X,Z,
                                                               splits ,iterations = 10, tree_number = trees)

max_position = [np.where(acc == np.max(acc))[0][0]+1,
                 np.where(acc == np.max(acc))[1][0]+1]
print("Max accuracy is ",np.max(acc),"for",
      max_position[0],"splits and ",max_position[1], "trees .")

```

Max accuracy is 93.4 for 2 splits and 13 trees.

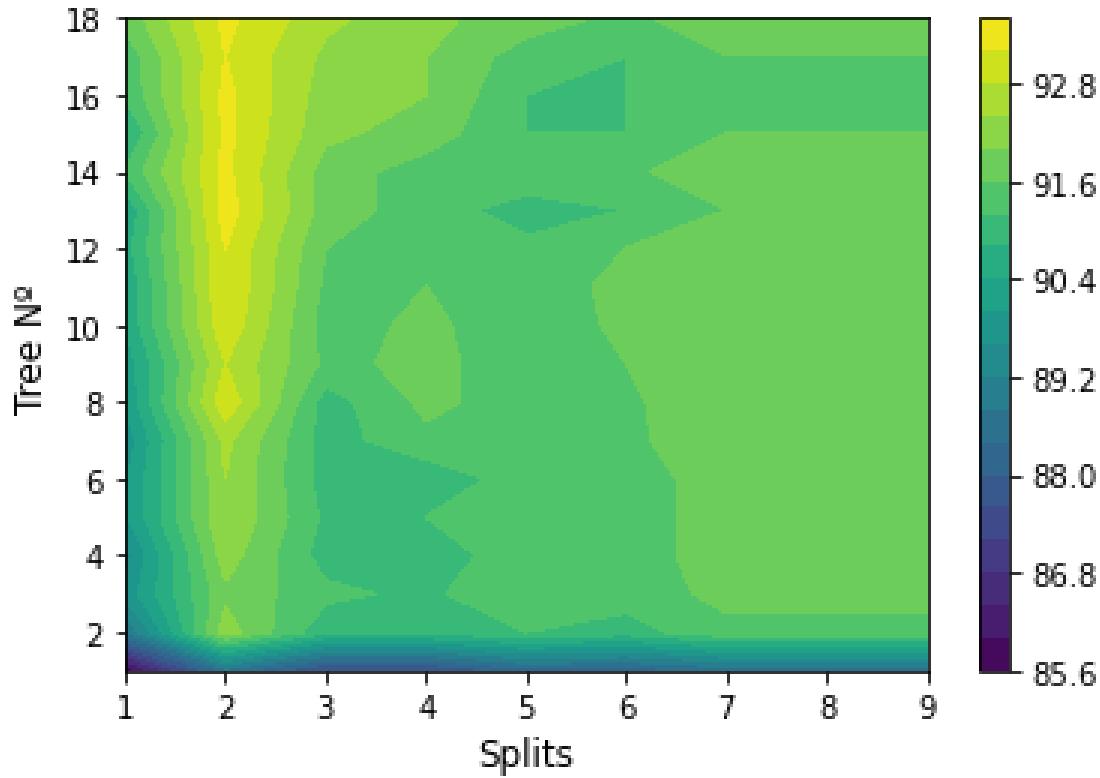
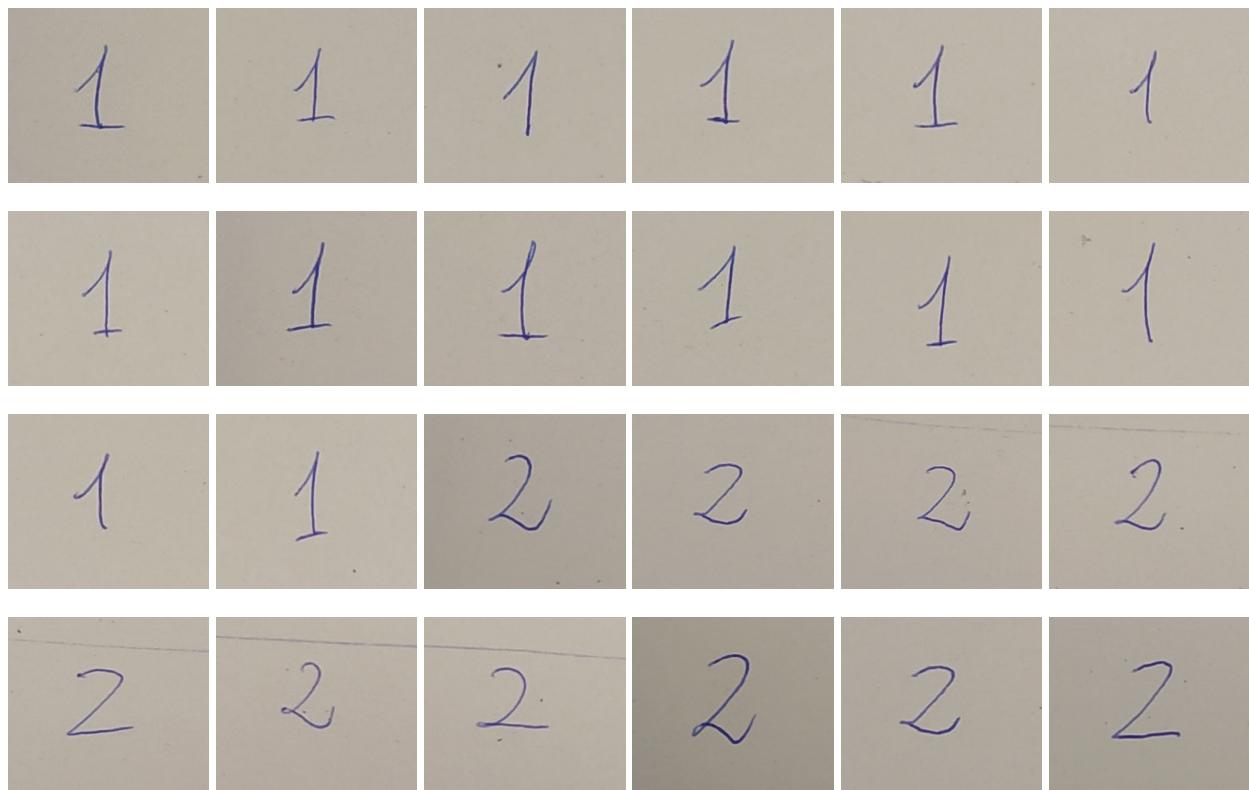


Figure 7.11: Accuracy distribution of the gradient boosted classification tree algorithm for two of the classes of the Iris data collection.

# Chapter 8

## Image Recognition Algorithms

In this chapter, we will work with three algorithms that will be used for image recognition; among them, we will have the *Eigenfaces* algorithm, built over the *Principal Component analysis* algorithm and two neural networks: a regular *Neural Network* and a *Convolutional Neural Network*. In order to test these algorithms, I created a dataset composed of the drawings of the numbers one, two, and three.



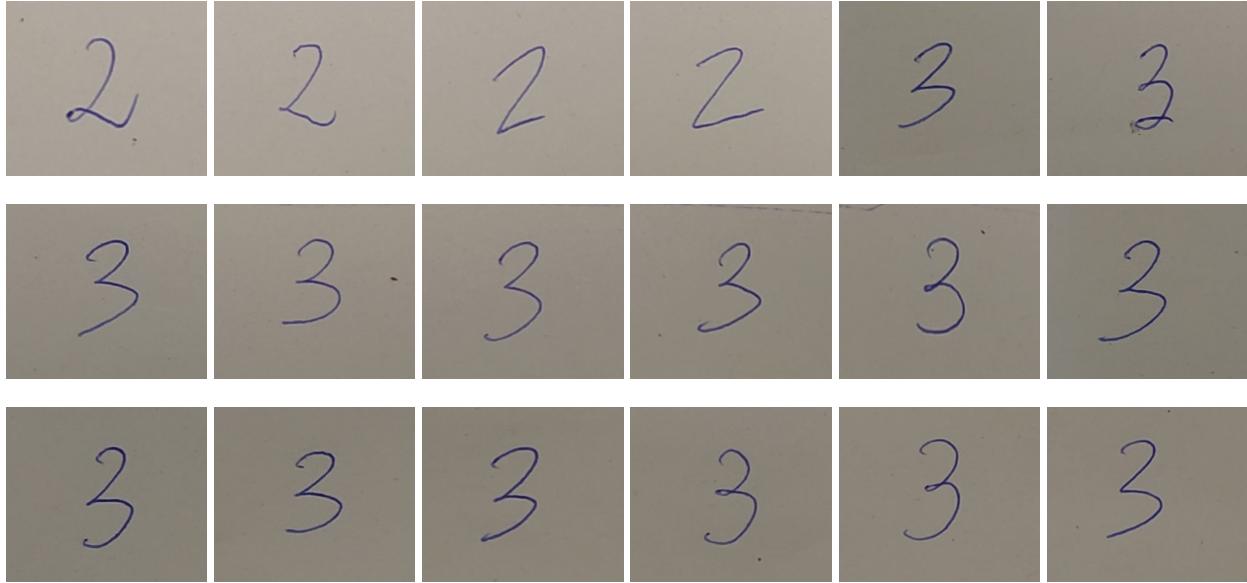


Figure 8.1: A data collection, including the drawing of a few numbers.

Some images containing the number two have some defects (horizontal lines), which will not matter as I will use a small piece of code to adapt the photos before being used by the algorithms.

```
[1]: import cv2
import numpy as np

class image_preparation():

    def __init__(self,photos):

        self.photos = photos
```

This first function reads the content of the input variable *photos*, which contains the file-names of the data collection, stored as *.jpg* files. Then, it calls for the different functions of the class, which will prepare the pictures for the algorithms, and then writes the results on some *.png* files.

```
def main_part(self):

    for photo in self.photos:
        image = cv2.imread(photo, cv2.IMREAD_GRAYSCALE)
        image = self.photo_cleaning(image)
        i_CM, j_CM, image = self.pixel_CoM(image)
        image = self.cutting_photo(image, i_CM, j_CM)
        image = self.lower_pixels(image)
```

```
cv2.imwrite(str(photo[0:len(photo)-4]+".png"),image)
```

The *photo-cleaning* function is meant to do a couple of tasks. First, let us remember that when loading the contents of a photo on a grayscale, we will have pixel values ranging from 0 to 255, where 0 corresponds to a black color and 255 to a white color, as it measures the white light intensity. The function's first task is to turn the photo background white and remove impurities. The function does this by setting the pixel intensity to 255 for those pixels with intensity higher than the mean intensity minus thirty and also for the top pixels where the horizontal lines are located. The second task is to revert the intensity of the photos, as I preferred the numbers to be over a dark background to facilitate the visualization.

```
def photo_cleaning(self,photo):  
  
    mean = np.mean(photo)  
  
    for i in range(len(photo)):  
        for j in range(len(photo[0])):  
  
            if (photo[i][j]>(mean-30) or i<40):  
                photo[i][j]= 255  
  
            photo[i][j] = abs(photo[i][j]-255)  
  
    return(photo)
```

Next, we have a function that computes the analogous of the center of mass for a photo; for example, one could call this *center of pixel intensity*. It computes the center of intensity of the pixels. The reason behind this is that we will cut some of the pixels later, and we want the numbers to be as centered as possible. Also, I scaled up all the pixels of the photo with an intensity higher than zero by an amount equal to 255 minus the maximum pixel intensity on the photo. So, let's say that the maximum value was 235; we would scale all the non-black pixels by 20.

```
def pixel_CoM(self,photo):  
  
    i_CM=0  
    j_CM=0  
    mean = np.mean(photo)  
    max_value = np.max(photo)  
  
    for i in range(len(photo)):  
        for j in range(len(photo[0])):
```

```

    i_CM = i_CM + i* photo[i][j]
    j_CM = j_CM + j* photo[i][j]

    if (photo[i][j]>0):

        photo[i][j] = photo[i][j] + (255-max_value)

    i_CM = i_CM / (len(photo)*len(photo[0])*mean)
    j_CM = j_CM / (len(photo)*len(photo[0])*mean)

    return(int(i_CM),int(j_CM),photo)

```

This following function takes each photo and takes 90 pixels on the left and right of the horizontal center of intensity and 90 pixels on top and bottom of the vertical center of pixel intensity, i.e., it returns the photos of the size 180 x 180 pixels.

```

def cutting_photo(self,photo,i_CM,j_CM):
    photo = photo[i_CM - 90 : i_CM + 90, j_CM - 90 : j_CM + 90]
    return(photo)

```

Lastly, we have the *lower-pixels* function, which averages 9 x 9 pixel blocks, resulting in some photos of size 20 x 20 pixels.

```

def lower_pixels(self,photo):
    a = 9
    b = 9
    photo_f = np.zeros((20,20))

    for i in range(20):
        for j in range(20):
            photo_f[i][j] = np.mean(photo[int(a*i):int(a*(i+1)),
                                         int(b*j):int(b*(j+1))])
    return(photo_f)

```

The next piece of code stores the photo filenames on an array and calls for the *image-preparation* function. This will generate some .png files; this new images can be seen in the following graph.

```
[2]: photos = ["1_1.jpg","1_2.jpg","1_3.jpg","1_4.jpg","1_5.jpg",
             "1_6.jpg","1_7.jpg","1_8.jpg","1_9.jpg","1_10.jpg",
             "1_11.jpg","1_12.jpg","1_13.jpg","1_14.jpg",
             "2_1.jpg","2_2.jpg","2_3.jpg","2_4.jpg","2_5.jpg",
             "2_6.jpg","2_7.jpg","2_8.jpg","2_9.jpg","2_10.jpg",
             "2_11.jpg","2_12.jpg","2_13.jpg","2_14.jpg",
```

```
"3_1.jpg", "3_2.jpg", "3_3.jpg", "3_4.jpg", "3_5.jpg",
"3_6.jpg", "3_7.jpg", "3_8.jpg", "3_9.jpg", "3_10.jpg",
"3_11.jpg", "3_12.jpg", "3_13.jpg", "3_14.jpg"]
image_prep = image_preparation(photos)
image_prep.main_part()
```

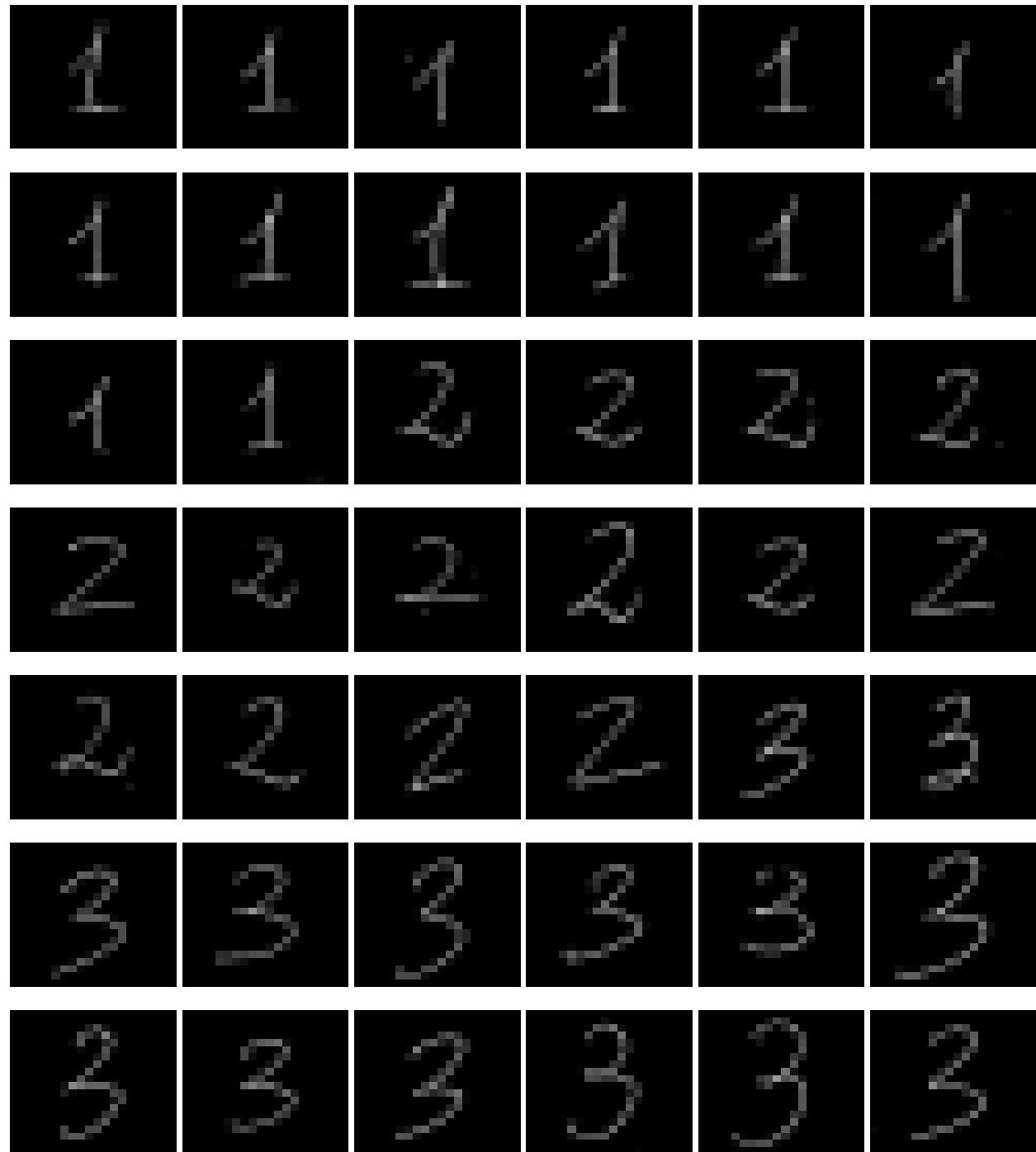


Figure 8.2: A data collection adapted for the image recognition algorithms.

## 8.1 Eigenfaces [Supervised]

### 8.1.1 Mathematical derivation

[33] The *Eigenfaces* algorithm is the first image recognition algorithm we will work with. The idea behind this algorithm is straightforward. Suppose we have a collection of  $N$  images of size  $P \times P$  pixels. We can flatten the matrices storing the intensity of the pixels and generate a matrix of size  $N \times P^2$ , where each row refers to a photo, and each of the  $P^2$  dimensions refers to a pixel. For the data collection introduced in this chapter, we will have 400 variables. The approach will be to use the Principal component analysis to obtain the PCs and reduce the dimensions of the problem from 400 to a lower number, keeping the essential information about the variation of the data. This will turn our problem from a 400-variable problem to a  $D$ -variable classification problem, which we can solve with any classification algorithm. In this case, I will use KNN to classify and make data predictions after reducing the problem's dimensions.

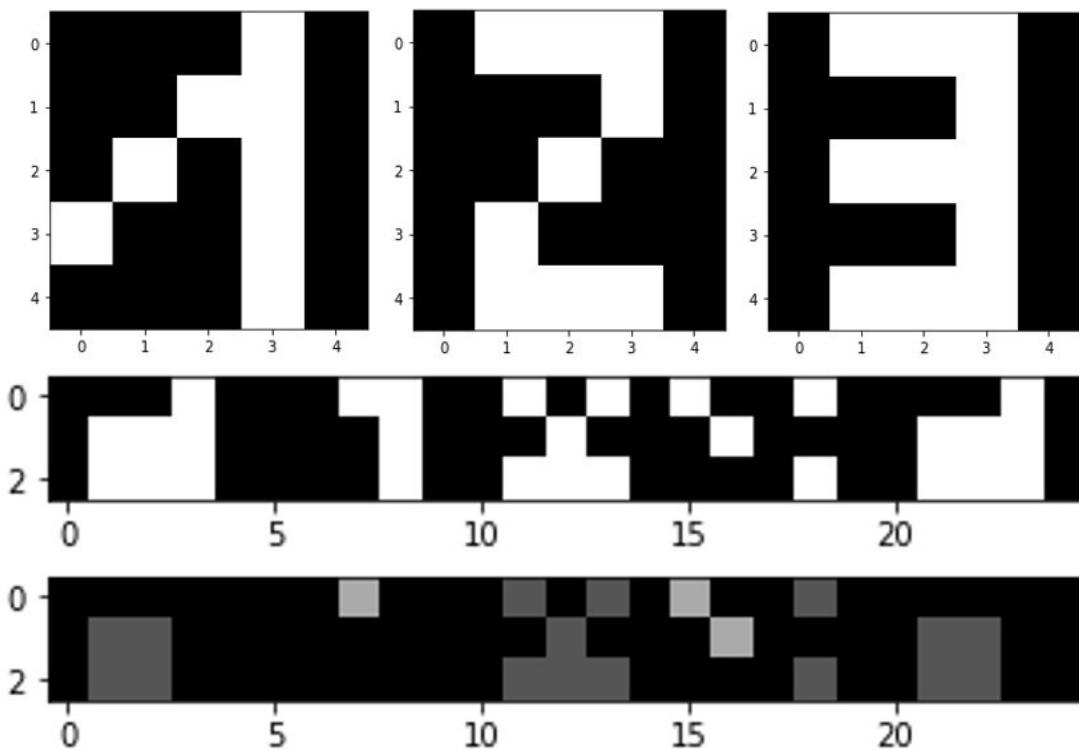


Figure 8.3: Example of the adaptation of a data collection for the *Eigenfaces* algorithm. We can see 5x5 pixel matrices for the numbers one, two, and three, the flattening of these matrices into a matrix of size  $3 \times 25$  pixels, and the posterior mean centering.

### 8.1.2 Algorithm implementation

To adapt the PCA algorithm to the *Eigenfaces* model, I had to change the printing portion of the *D-reduction* function since we will work with 400 variables, and printing the equations of the new variables was pointless. The rest of the functions remain the same.

```
[1]: import numpy as np
import copy as copy
from numpy import linalg as la

class PCA():

    :

def D_reduction(self,eigenvectors,info):

    D = self.dimensions
    X = np.array(self.X)
    eigenvectors = np.array(eigenvectors)

    print("          RESULTS (",self.dimensions,
          "axis )           ")
    print("-----")

    for i in range(len(eigenvectors)):

        print("Axis n° ",i+1," , information gain of ",
              np.around(info[i],2),"%")
        print("-----")

    print("          Total information gain:")
    print("          ",np.around(sum(info[0:D]),2))

    New_X = np.array([[X[i].dot(eigenvectors[j]) \
                      for j in range(D)] for i in range(len(X))])

    return(New_X)
```

Next, we have the implementation of the *Eigenfaces* algorithm.

```
[2]: import cv2
class eigenfaces():

    def __init__(self,basis,dim):
```

```

    self.basis=basis
    self.dim = dim

```

The *construct-basis* function starts by reading the information of the different photos, which is given to the class as input. It flattens the pixel matrices of the different photos and appends them to an empty array labeled as *new-basis*.

```

def construct_basis(self):

    new_basis = []
    for i in range(len(self.basis)):

        photo = cv2.imread(self.basis[i],
                           cv2.IMREAD_GRAYSCALE)
        photo = np.reshape(photo,(len(photo)*len(photo[0]),))
        new_basis.append(photo)

    new_basis = np.array(new_basis)

    return(new_basis)

```

Then, we have the primary function of the code. It starts calling the *construct-basis* function, then calls for the *PCA* class, which transforms the data into the variables associated with the requested Principal Components.

```

def main(self):

    new_basis = self.construct_basis()
    pca = PCA(new_basis,self.dim,[])
    new_data = pca.pca_main()
    return(new_data)

```

Next, if we load the photos into the input variable and then execute the *Eigenfaces* class, requesting the first two Principal Components, we obtain the following results.

```
[3]: photos = ["1_1.png","1_2.png","1_3.png","1_4.png","1_5.png",
             "1_6.png","1_7.png","1_8.png","1_9.png","1_10.png",
             "1_11.png","1_12.png","1_13.png","1_14.png",
             "2_1.png","2_2.png","2_3.png","2_4.png","2_5.png",
             "2_6.png","2_7.png","2_8.png","2_9.png","2_10.png",
             "2_11.png","2_12.png","2_13.png","2_14.png",
             "3_1.png","3_2.png","3_3.png","3_4.png","3_5.png",
             "3_6.png","3_7.png","3_8.png","3_9.png","3_10.png",
             "3_11.png","3_12.png","3_13.png","3_14.png"]
```

```
reg = eigenfaces(photos,2)
PCA_X = reg.main()
```

```
RESULTS ( 2 axis )
-----
Axis n° 1 , information gain of 20.65 %
-----
Axis n° 2 , information gain of 14.53 %
-----
Total information gain:
35.17
```

As we can see, by combining the two PCs, we preserve an information or data variation of 35.17%. As we will see, this is enough to cluster the images associated with each number. If we plot the results, we obtain the following graph.

```
[4]: import matplotlib.pyplot as plt

Z = [0,0,0,0,0,0,0,0,0,0,0,0,
      1,1,1,1,1,1,1,1,1,1,1,1,1,
      2,2,2,2,2,2,2,2,2,2,2,2,2]

color=["blue","green","red"]

for i in range(len(basis)):
    plt.scatter(PCA_X[i][0],PCA_X[i][1],color=color[Z[i]])
```

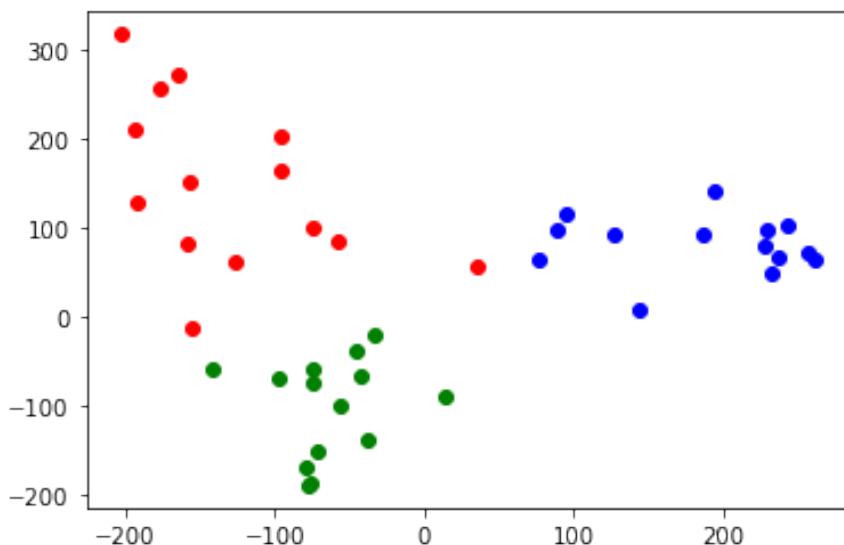


Figure 8.4: Variables associated to the first two PCs of the photo data collection.

Once our data is reduced from 400 dimensions to 2, we use the plot function of the KNN model to visualize the clusters. I did this for a few values of the nearest neighbors, which can be seen below.

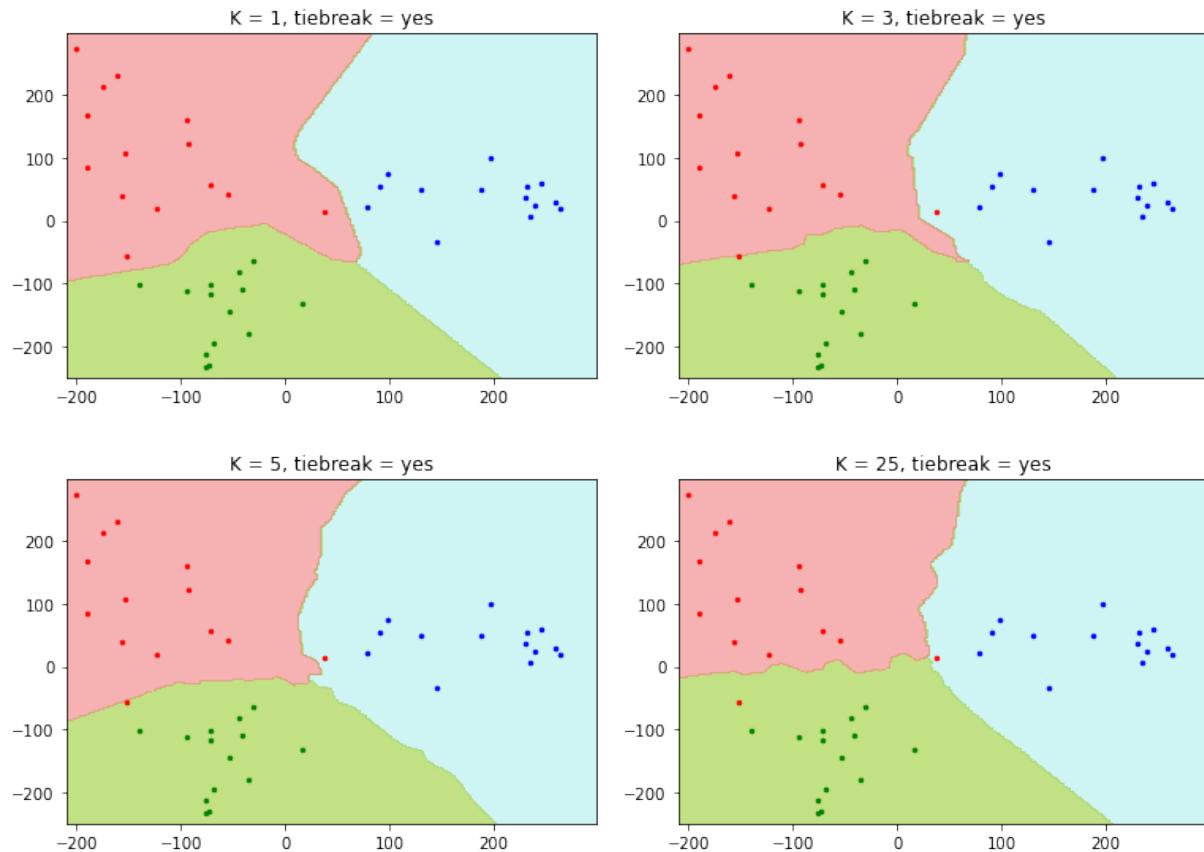


Figure 8.5: KNN on the variables associated to the first two principal components of the photo collection.

Using the KNN cross-validation function on the first 20 random states, we obtain an accuracy of 95.44 %. This means that the algorithm will predict roughly 19 out of 20 photos, which is impressive given that our data collection only comprised 42 photos.

```
[5]: k = np.arange(1,20,1)
results = np.array([K_fold_cross_validation_KNN(PCA_X,Z,
                                                k_i,20) for k_i in k])

test_accuracy = results[:,1]
train_accuracy = results[:,2]
print("Maximum value of the accuracy is:",np.max(test_accuracy))
```

Maximum value of the accuracy is: 95.44444444444446

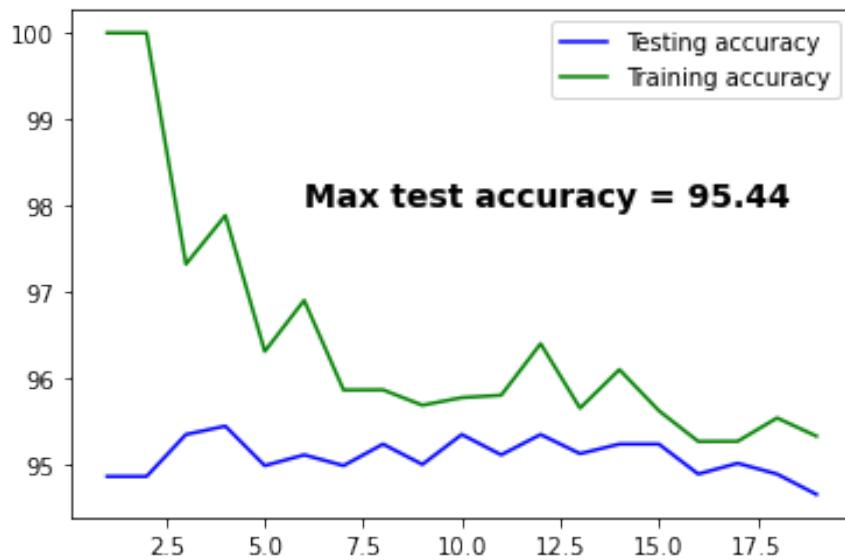
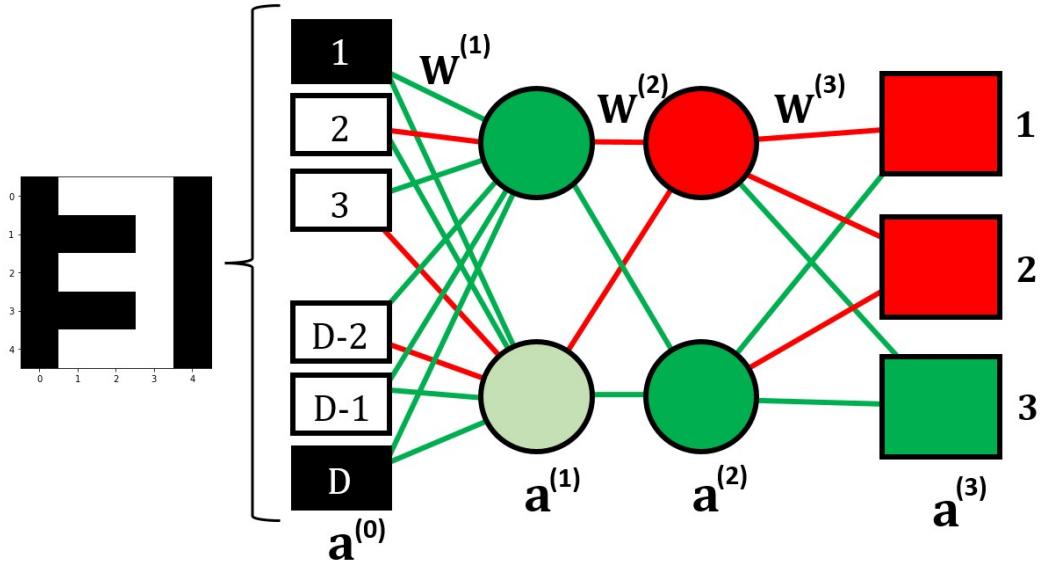


Figure 8.6: Accuracy of the KNN model on the variables associated to the first two PCs of the photo data collection.

## 8.2 Neural Networks [Supervised]

### 8.2.1 Mathematical derivation



[5][55][56][57][58] A *Neural Network* is composed of layers of interconnected neurons. A neuron is a number, an output of the previous layer. The neurons of different layers are connected by some weights that link them together and establish connections between them. First, we have an input layer, which, for image recognition, will be composed of  $D$  input neurons storing the pixel intensity of a photo. Then, we have the hidden layers, which are just a mechanism to add more variables and degrees of freedom to the neural network. The last layer is the output layer, which will contain the probability of the different outcomes. On the graph, we can see three neurons on the outcome layer, giving the probabilities of the prediction being one, two, or three.

The idea is the following: We plug the intensity of the pixels of a photo into the Neural Network; this activates the neurons of the following layers based on the connections between the neurons, leading to an activation of the neurons from the outcome layer, which will give a prediction on the photo.

Now, let's translate all this into the equations behind the model. We start with the input of the neural network, i.e., the values of the pixels  $a^{(0)}$ . We can compute the activation (values) of the next layer with the following equation:

$$a^{(1)} = \text{ReLU}(W^{(1)}a^{(0)} + b^{(1)}) \quad (8.1)$$

Where  $a^{(0)}$  is a column vector of size  $D$ ,  $a^{(1)}$  is a column vector of size  $M$ , and hence  $W^{(1)}$  is a matrix of size  $M \times D$ .  $b^{(1)}$  is also a column vector of size  $M$ . The weights  $W^{(1)}$  establish

a connection between the neurons of the two layers.  $a^{(0)}$  are the input neurons of the first layer, and  $a^{(1)}$  the output neurons.  $b^{(1)}$  can be understood as a barrier the neurons need to surpass to activate. Lastly, we have the leaky ReLU (rectified linear unit) function, which is the following:

$$\text{ReLU}(x) = \begin{cases} x & \text{for } x > 0 \\ ax & \text{for } x < 0 \end{cases}$$

Where  $a$  is a parameter smaller than one. This function penalizes the influence of negative neurons, making their values smaller. Before the leaky ReLU function, people used to implement the regular ReLU function, which sets the values of the neurons to zero when they are smaller than zero (red neurons). This resembles how a human brain works, where neurons might be active or inactive depending on the interaction with surrounding neurons.

One thing that needs to be mentioned is that the weights are initialized randomly and then adjusted by training the neural network. One problem that would arise with the regular ReLU function is that some neurons would be set inactive from the start and hence remain completely inactive; this update to the ReLU function allows the neurons to activate again if they are relevant. The activation of the following layers is given by:

$$a^{(2)} = \text{ReLU}(W^{(2)}a^{(1)} + b^{(2)}) \quad (8.2)$$

$$a^{(3)} = \text{ReLU}(W^{(3)}a^{(2)} + b^{(3)}) \quad (8.3)$$

Where  $a^{(3)}$  is the activation of the output layer; the last step is to introduce a way to measure the error in the prediction of a particular photo, which will be given by the Squared Error:

$$SE = (a^{(3)} - y)^2 = \sum_l (a_l^{(3)} - y_l)^2 \quad (8.4)$$

Keep in mind that this is the contribution of a single photo. For the Neural Networks, the outcomes and predictions will be given by arrays of length equal to the possible outcomes, in this case, three (the outcome of a photo containing the number three will be  $y_i = [0, 0, 1]^T$ ). We only need to write the contribution of a single photo, as the results will be averaged across the different photos.

The idea behind the neural networks is to initialize the weights randomly. When using the ReLU function, the most optimal choice for the weights is the He weights, given by  $W^{(m)} = G(0, \sqrt{2/\text{input layer size}})$ , i.e., a Gaussian distribution of mean zero and variance equal to  $\sqrt{2/\text{input layer size}}$ . Then, we take a photo, use the pixel values as input, and make a prediction. Then, we measure this prediction's error and update the weights using gradient descent so the photo is predicted correctly. This is done for all the training photos until the Neural Network predicts all of them correctly. This is why we introduce more degrees of freedom by introducing layers and neurons so that adjusting the weights is compatible with predicting all the training data. Using gradient descent to

correct the weights is known as *back-propagation*. The update rule of the weights in order to minimize the SE is given by:

$$W^{(m)} = W^{(m)} - \nabla_{W^{(m)}} SE \quad (8.5)$$

$$b^{(m)} = b^{(m)} - \nabla_{b^{(m)}} SE \quad (8.6)$$

Now, all that is left to do is to compute these quantities, and for that, we recover the expressions again:

$$\begin{aligned} a_j^{(1)} &= \text{ReLU}(Z_j^{(1)}) \quad \text{with} \quad Z_j^{(1)} = \sum_i W_{j,i}^{(1)} a_j^{(0)} + b_j^{(1)} \\ a_k^{(2)} &= \text{ReLU}(Z_k^{(2)}) \quad \text{with} \quad Z_k^{(2)} = \sum_j W_{k,j}^{(2)} a_j^{(1)} + b_k^{(2)} \\ a_l^{(3)} &= \text{ReLU}(Z_l^{(3)}) \quad \text{with} \quad Z_l^{(3)} = \sum_k W_{l,k}^{(3)} a_k^{(2)} + b_l^{(3)} \end{aligned}$$

We start computing the partial derivatives of the SE with respect to the different weights. First, we start with those of the last layer transition.

$$\frac{\partial SE}{\partial W_{l,k}^{(3)}} = \frac{\partial SE}{\partial a_l^{(3)}} \frac{\partial a_l^{(3)}}{\partial Z_l^{(3)}} \frac{\partial Z_l^{(3)}}{\partial W_{l,k}^{(3)}} = 2(a_l^{(3)} - y_l) \text{ReLU}'(Z_l^{(3)}) a_k^{(2)} \quad (8.7)$$

$$\frac{\partial SE}{\partial b_l^{(3)}} = \frac{\partial SE}{\partial a_l^{(3)}} \frac{\partial a_l^{(3)}}{\partial Z_l^{(3)}} \frac{\partial Z_l^{(3)}}{\partial b_l^{(3)}} = 2(a_l^{(3)} - y_l) \text{ReLU}'(Z_l^{(3)}) \quad (8.8)$$

Now, we proceed with the partial derivatives of the second layer:

$$\begin{aligned} \frac{\partial SE}{\partial W_{k,j}^{(2)}} &= \frac{\partial SE}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial Z_k^{(2)}} \frac{\partial Z_k^{(2)}}{\partial W_{k,j}^{(2)}} = a_j^{(1)} \text{ReLU}'(Z_k^{(2)}) \frac{\partial SE}{\partial a_k^{(2)}} = \\ &a_j^{(1)} \text{ReLU}'(Z_k^{(2)}) \frac{\partial \sum_l (a_l^{(3)} - y_l)^2}{\partial a_k^{(2)}} = \\ &\sum_l a_j^{(1)} \text{ReLU}'(Z_k^{(2)}) \frac{\partial (a_l^{(3)} - y_l)^2}{\partial a_l^{(3)}} \frac{\partial a_l^{(3)}}{\partial Z_l^{(3)}} \frac{\partial Z_l^{(3)}}{\partial a_k^{(2)}} = \\ &a_j^{(1)} \text{ReLU}'(Z_k^{(2)}) \sum_l 2(a_l^{(3)} - y_l) \text{ReLU}'(Z_l^{(3)}) W_{l,k}^{(3)} \end{aligned} \quad (8.9)$$

The remaining terms are straightforward using the previous result:

$$\frac{\partial SE}{\partial b_k^{(2)}} = \text{ReLU}'(Z_k^{(2)}) \sum_l 2(a_l^{(3)} - y_l) \text{ReLU}'(Z_l^{(3)}) W_{l,k}^{(3)} \quad (8.10)$$

In the same way, it is straightforward to find the partial derivatives of the first layer:

$$\frac{\partial SE}{\partial W_{j,i}^{(1)}} = a_i^{(0)} \text{ReLU}'(Z_j^{(1)}) \sum_l \sum_k 2(a_l^{(3)} - y_l) \text{ReLU}'(Z_l^{(3)}) W_{l,k}^{(3)} \text{ReLU}'(Z_k^{(2)}) W_{k,j}^{(2)} \quad (8.11)$$

$$\frac{\partial SE}{\partial b_j^{(1)}} = \text{ReLU}'(Z_j^{(1)}) \sum_l \sum_k 2(a_l^{(3)} - y_l) \text{ReLU}'(Z_l^{(3)}) W_{l,k}^{(3)} \text{ReLU}'(Z_k^{(2)}) W_{k,j}^{(2)} \quad (8.12)$$

Now that we have computed the derivatives, we will establish the steps that we will follow to build the *Neural Network* algorithm.

## STEPS

- Initialize all the weights  $W_{i,j}^{(m)}$  using He weights given by a Gaussian of zero mean and variance  $\sqrt{2/\text{input layer size}}$  and  $b_i^{(m)}$  which is initialized to zero.
- We iterate until convergence is reached and do the following steps:
  - Compute  $a_j^{(1)}, a_k^{(2)}, a_l^{(3)}$  for the different input photos on the same iteration.
  - Compute the partial derivatives  $\partial SE / \partial W_{i,j}^{(m)}$  and  $\partial SE / \partial b_i^{(m)}$  for all the photos, i.e., plugging the respective  $a_j^{(1)}, a_k^{(2)}, a_l^{(3)}$  of each photo.
  - We average the corrections of all the photos and update  $W_{i,j}^{(m)}$  and  $b_i^{(m)}$ .

## 8.2.2 Algorithm implementation

Now, we proceed with the implementation of the Neural Network, which we will test on the data collection I draw containing the numbers one, two, and three.

```
[1]: import numpy as np
import copy as copy

class neural_network():

    def __init__(self,X_train,Y_train,layer_size):

        self.X_train = X_train
        self.Y_train = Y_train
        self.iterations = 15000
        self.lr = 0.1
        self.a = len(X_train[0])
        self.b = layer_size
        self.c = layer_size
        self.d = len(Y_train[0])
```

The algorithm will take an array of filenames as input. This function will start reading the different files of the entry, then flatten them and append them to an empty array.

```
def construct_data(self,X):

    filenames = copy.deepcopy(X)
    X = []
    for pic in filenames:

        photo = cv2.imread(pic,cv2.IMREAD_GRAYSCALE)
        X.append(np.reshape(photo,(len(photo)*len(photo[0]),)))

    X = np.array(X)
    return(X)
```

The *main-part* function of the algorithm starts by adapting the training data and updating the size of the input layer. It also initializes the weights and the  $b_i^{(m)}$  factors.

```
def main_part(self):

    self.X_train = self.construct_data(self.X_train)
    self.a = len(self.X_train[0])

    #He weight initialization
```

```

Wji_1 = np.random.randn(self.b, self.a)*np.sqrt(2.0/self.a)
Wkj_2 = np.random.randn(self.c, self.b)*np.sqrt(2.0/self.b)
Wlk_3 = np.random.randn(self.d, self.c)*np.sqrt(2.0/self.c)

bj_1 = np.zeros((self.b,1))
bk_2 = np.zeros((self.c,1))
bl_3 = np.zeros((self.d,1))

```

Then, it starts iterating and, for each iteration, runs through all the photos of the training data, which will correspond to each of the rows of the variable  $X\text{-train}$ . For each of these photos, it loads the value of  $a_i^{(0)}$  and then calls for the *update* function, which will return the partial derivatives from Equation 8.7 to Equation 8.12. Then  $\Delta W_{i,j}^{(m)}$  and  $\Delta b_i^{(m)}$  obtained to correct the predictions of the different photos, divided by the length of the training data is added to some empty arrays, this is the equivalent of averaging the partial derivatives for all the training points. Lastly it updates  $W_{i,j}^{(m)}$  and  $b_i^{(m)}$ . There is also a conditional to break the loop if the difference between the predictions and the outcomes is smaller than 0.05.

```

for it in range(self.iterations):

    delta_Wji_1 = np.zeros((np.shape(Wji_1)))
    delta_Wkj_2 = np.zeros((np.shape(Wkj_2)))
    delta_Wlk_3 = np.zeros((np.shape(Wlk_3)))
    delta_bj_1 = np.zeros((np.shape(bj_1)))
    delta_bk_2 = np.zeros((np.shape(bk_2)))
    delta_bl_3 = np.zeros((np.shape(bl_3)))

    count = 0

    for pic in range(len(self.X_train)):

        ai_0 = np.array(self.X_train[pic])/255
        ai_0 = np.reshape(ai_0,(len(ai_0),1))
        y = self.Y_train[pic]

        derivatives = self.update(ai_0,y,Wji_1, Wkj_2, Wlk_3,
                                  bj_1 , bk_2 , bl_3)
        al_3 = derivatives[-1]

        if (abs(al_3[0]-y[0])<=0.05 and \
            abs(al_3[1]-y[1])<=0.05 and \
            abs(al_3[2]-y[2])<=0.05):

```

```

        count = count + 1

        delta_Wji_1 = delta_Wji_1 + \
                      np.array(derivatives[0])/len(self.X_train)
        delta_Wkj_2 = delta_Wkj_2 + \
                      np.array(derivatives[1])/len(self.X_train)
        delta_Wlk_3 = delta_Wlk_3 + \
                      np.array(derivatives[2])/len(self.X_train)
        delta_bj_1 = delta_bj_1 + \
                      np.array(derivatives[3])/len(self.X_train)
        delta_bk_2 = delta_bk_2 + \
                      np.array(derivatives[4])/len(self.X_train)
        delta_bl_3 = delta_bl_3 + \
                      np.array(derivatives[5])/len(self.X_train)

        Wji_1 = Wji_1 - self.lr * delta_Wji_1
        Wkj_2 = Wkj_2 - self.lr * delta_Wkj_2
        Wlk_3 = Wlk_3 - self.lr * delta_Wlk_3
        bj_1 = bj_1 - self.lr * delta_bj_1
        bk_2 = bk_2 - self.lr * delta_bk_2
        bl_3 = bl_3 - self.lr * delta_bl_3

    if (count == len(self.X_train)):

        break

    return(Wji_1,Wkj_2,Wlk_3,bj_1,bk_2,bl_3)

```

The *update* function takes  $W_{i,j}^{(m)}$ ,  $b_i^{(m)}$ ,  $a_i^{(0)}$  and the outcomes as input, then computes the activation of all the neurons on the different layers and lastly, the partial derivatives  $\Delta W_{i,j}^{(m)}$  and  $\Delta b_i^{(m)}$ .

```

def update(self,ai_0,y,Wji_1, Wkj_2, Wlk_3, bj_1 , bk_2 , bl_3):

    delta_Wji_1 = np.zeros((np.shape(Wji_1)))
    delta_Wkj_2 = np.zeros((np.shape(Wkj_2)))
    delta_Wlk_3 = np.zeros((np.shape(Wlk_3)))
    delta_bj_1 = np.zeros((np.shape(bj_1)))
    delta_bk_2 = np.zeros((np.shape(bk_2)))
    delta_bl_3 = np.zeros((np.shape(bl_3)))

    Zj_1 = Wji_1.dot(ai_0) + bj_1
    aj_1 = np.reshape(np.array([self.ReLU(x[0]) for x in Zj_1]),

```

```

np.shape(Zj_1))
Zk_2 = Wkj_2.dot(aj_1) + bk_2
ak_2 = np.reshape(np.array([self.ReLU(x[0]) for x in Zk_2]), np.shape(Zk_2))
Zl_3 = Wlk_3.dot(ak_2) + bl_3
al_3 = np.reshape(np.array([self.ReLU(x[0]) for x in Zl_3]), np.shape(Zl_3))

for k in range(self.c):
    for l in range(self.d):

        delta_b1_3[l][0] = 2*(al_3[l] - y[l])*\
                            self.ReLU_prime(Zl_3[l])
        delta_Wlk_3[l][k] = 2*(al_3[l] - y[l])*\
                            self.ReLU_prime(Zl_3[l])*ak_2[k]

        delta_bk_2[k][0] = delta_bk_2[k][0] + \
                            self.ReLU_prime(Zk_2[k])*2*(al_3[l] - y[l])*\
                            self.ReLU_prime(Zl_3[l])*Wlk_3[l][k]

    for j in range(self.b):

        delta_Wkj_2[k][j] = delta_Wkj_2[k][j] + \
                            aj_1[j]*self.ReLU_prime(Zk_2[k])* \
                            2*(al_3[l] - y[l])*self.ReLU_prime(Zl_3[l])\
                            *Wlk_3[l][k]

        delta_bj_1[j][0] = delta_bj_1[j][0] + \
                            self.ReLU_prime(Zj_1[j])*2*(al_3[l] - y[l])* \
                            self.ReLU_prime(Zl_3[l])*Wlk_3[l][k] \
                            *self.ReLU_prime(Zk_2[k])*Wkj_2[k][j]

    for i in range(self.a):
        for j in range(self.b):

            delta_Wji_1[j][i] = delta_bj_1[j][0] * ai_0[i]

return(delta_Wji_1, delta_Wkj_2 , delta_Wlk_3 ,
       delta_bj_1 , delta_bk_2 , delta_b1_3,al_3)

```

The *predict* function starts by adapting the testing data from filenames to a pixel intensity matrix (which will be scaled down to the pixel value interval [0,1], which was also

done for the training data). Then, it calls for the *main-part* function, which will train on the training data. Last, it computes  $a^{(3)}$  for the testing data and appends the rounded results.

```
def predict(self,X_test):

    X_test = self.construct_data(X_test)
    X_test = np.array(X_test)/255

    Wji_1,Wkj_2,Wlk_3,bj_1,bk_2,bl_3 = self.main_part()

    results = []

    for i in range(len(X_test)):

        ai_0 = np.reshape(X_test[i],(len(X_test[i]),1))
        Zj_1 = Wji_1.dot(ai_0) + bj_1
        aj_1 = np.reshape(np.array([self.ReLU(x[0]) \
                                    for x in Zj_1]),np.shape(Zj_1))
        Zk_2 = Wkj_2.dot(aj_1) + bk_2
        ak_2 = np.reshape(np.array([self.ReLU(x[0]) \
                                    for x in Zk_2]),np.shape(Zk_2))
        Zl_3 = Wlk_3.dot(ak_2) + bl_3
        al_3 = np.reshape(np.array([self.ReLU(x[0]) \
                                    for x in Zl_3]),np.shape(Zl_3))
        results.append(np.transpose(al_3)[0])
    results = np.round(np.array(results),2)
    return(results)
```

To end with the functions of the Neural Network class, we have the ReLU function and its derivative.

```
def ReLU(self,x):
    if (x<0):
        return(x/10)
    else:
        return(x)

def ReLU_prime(self,x):
    if (x<0):
        return(0.1)
    else:
        return(1.0)
```

Next, we load the data into the variables X and Y.

```
[2]: import cv2
import numpy as np

X = ["1_1.png", "1_2.png", "1_3.png", "1_4.png", "1_5.png",
     "1_6.png", "1_7.png", "1_8.png", "1_9.png", "1_10.png",
     "1_11.png", "1_12.png", "1_13.png", "1_14.png",
     "2_1.png", "2_2.png", "2_3.png", "2_4.png", "2_5.png",
     "2_6.png", "2_7.png", "2_8.png", "2_9.png", "2_10.png",
     "2_11.png", "2_12.png", "2_13.png", "2_14.png",
     "3_1.png", "3_2.png", "3_3.png", "3_4.png", "3_5.png",
     "3_6.png", "3_7.png", "3_8.png", "3_9.png", "3_10.png",
     "3_11.png", "3_12.png", "3_13.png", "3_14.png"]

Y = np.array([[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
              [1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
              [1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
              [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
              [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
              [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
              [0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],
              [0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],
              [0,0,1],[0,0,1],[0,0,1],[0,0,1]])
```

Now, we introduce the cross-validation function for the Neural Network, which is similar to the function of other algorithms. The only difference is that, in this case, the outcome of each data point (photo) will be given by an array. So, we will set the outcome equal to the position of the maximum value on the array. So an outcome  $y = [1, 0, 0]^T$  will be assigned a zero,  $y = [0, 1, 0]^T$  will be assigned a one, and so on. The same will be applied to the predictions which will be given in the same format. So if, for example, for a given testing point  $y = [1, 0, 0]^T$  and  $\hat{y} = [0.62, 0.2, 0.13]^T$ , the algorithm will take the position of the maximum value on the arrays, in this case, the zeroth position and hence determine that the testing point has been correctly predicted since it matches.

```
[3]: from sklearn.model_selection import KFold

def K_fold_cross_validation_NN(X,Y,layer_size,iterations):

    test_accuracy = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)
```

```

for i, (train_index, test_index) in enumerate(KF.split(X)):

    X_train = np.array([X[i] for i in train_index])
    Y_train = np.array([Y[i] for i in train_index])
    X_test = np.array([X[i] for i in test_index])
    Y_test = np.array([Y[i] for i in test_index])

    reg = neural_network(X_train,Y_train,layer_size)

    # PREDICTIONS
    Y_pred = reg.predict(X_test)

    Y_test = np.array([np.where(Y_test[i] == \
                                np.max(Y_test[i]))[0][0] \
                                for i in range(len(Y_test))])
    Y_pred = np.array([np.where(Y_pred[i] == \
                                np.max(Y_pred[i]))[0][0] \
                                for i in range(len(Y_pred))])

    # ERRORS
    error = Y_test - Y_pred

    # CORRECT PREDICTIONS
    correct_pred = np.count_nonzero(error == 0)

    test_accuracy.append(100*correct_pred/len(test_index))

return(np.mean(test_accuracy))

```

Next, we execute the cross-validation function for the first 20 random states and test different sizes of the inner layers.

```
[4]: accuracy = np.zeros((5))

for i in range(2,7):

    accuracy[i-2] = K_fold_cross_validation_NN(X,Y,i,20)
```

The accuracy tends to improve as we increase the size of the inner layers. It must be mentioned that I tested different executions of the cross-validation function for the first random state and for an inner layer size of two, and the accuracy had a large dispersion, ranging from 85% accuracy up to 97%. This is because the outcome depends on the initial choice of weights, and many configurations allow the classification of the training data correctly, leading to different predictions on the testing data depending on a random weight choice. This should improve when working with higher data collections. We

end up finding the maximum value of the accuracy equal to 96.51% for a layer size of 6. Although this accuracy may improve with larger layers, we are content with the results as this is just an exercise to test if the algorithm works correctly.

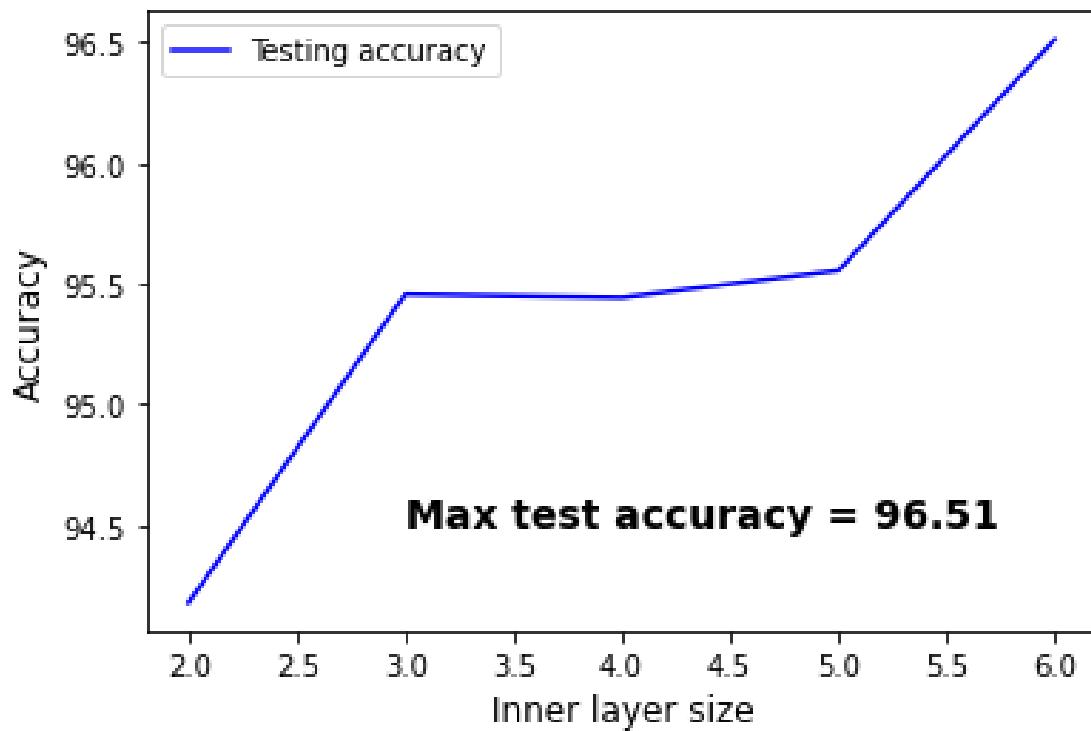


Figure 8.7: Results of the Neural Network on the photo data collection in terms of the size of the inner layers.

## 8.3 CNN [Supervised]

### 8.3.1 Mathematical derivation

[8][9][14][15] Before introducing the Convolutional Neural Networks, we have to introduce the concept of a convolution. A convolution is an operation between two matrices. Let's suppose we have a grey-scale square photo given by a matrix  $P$  of size  $N \times N$  pixels (I will call these matrices input matrices) and a matrix  $C$  of size  $3 \times 3$  (I will call these matrices convolution matrices for simplicity):

$$P = \begin{pmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,N} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ P_{N,1} & P_{N,2} & \cdots & P_{N,N} \end{pmatrix} \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} \quad (8.13)$$

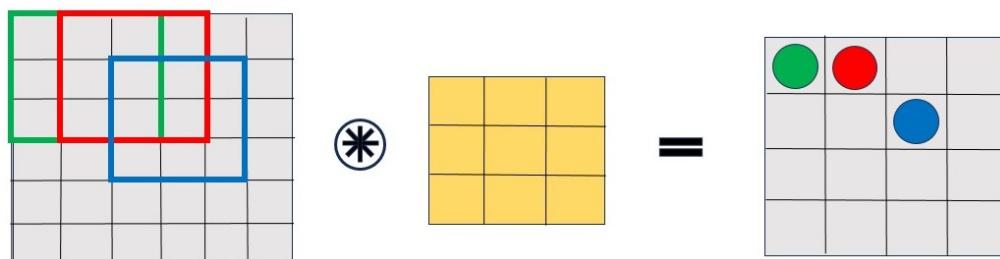
The result of the convolution operation  $*$  between the matrices  $P$  and  $C$  yields the matrix  $P'$ :

$$P' = P * C = \begin{pmatrix} P'_{1,1} & P'_{1,2} & \cdots & P'_{1,N-2} \\ P'_{2,1} & P'_{2,2} & \cdots & P'_{2,N-2} \\ \vdots & \vdots & \ddots & \vdots \\ P'_{N-2,1} & P'_{N,2} & \cdots & P'_{N-2,N-2} \end{pmatrix} \quad (8.14)$$

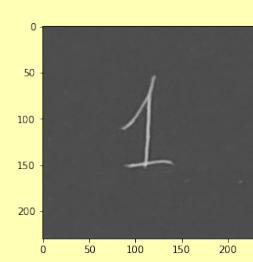
Where each matrix element is given by:

$$\begin{aligned} P'_{i,j} = & P_{i,j} C_{1,1} + P_{i,j+1} C_{1,2} + P_{i,j+2} C_{1,3} + \\ & P_{i+1,j} C_{2,1} + P_{i+1,j+1} C_{2,2} + P_{i+1,j+2} C_{2,3} + \\ & P_{i+2,j} C_{3,1} + P_{i+2,j+1} C_{3,2} + P_{i+2,j+2} C_{3,3} \end{aligned} \quad (8.15)$$

We have an illustration of the convolution operation on the graph below. The input matrix is divided into squares of the size of the convolution matrix  $C$ , and all the elements from the submatrices and  $C$  are multiplied and added up (scalar product).



To see why this operation can be useful, I will take as an example a photo of a one from the data collection, and we will consider the following convolution matrices:



$$L = \begin{pmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 0 & 1 & -1 \\ 0 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

$$T = \begin{pmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ -1 & -1 & -1 \end{pmatrix}$$

If we apply the convolution operation of the four different matrices, L, R, T, and B, on the photo, we obtain the results shown in the following graph. As we can see, each of these matrices picks up the left, right, top, and bottom edges of the photo, respectively. Now, we can understand why this mechanism is handy for pattern recognition. As the convolution gets more complex, this mechanism can pick up more complex patterns, such as eyes, particular objects, or any desired shape.

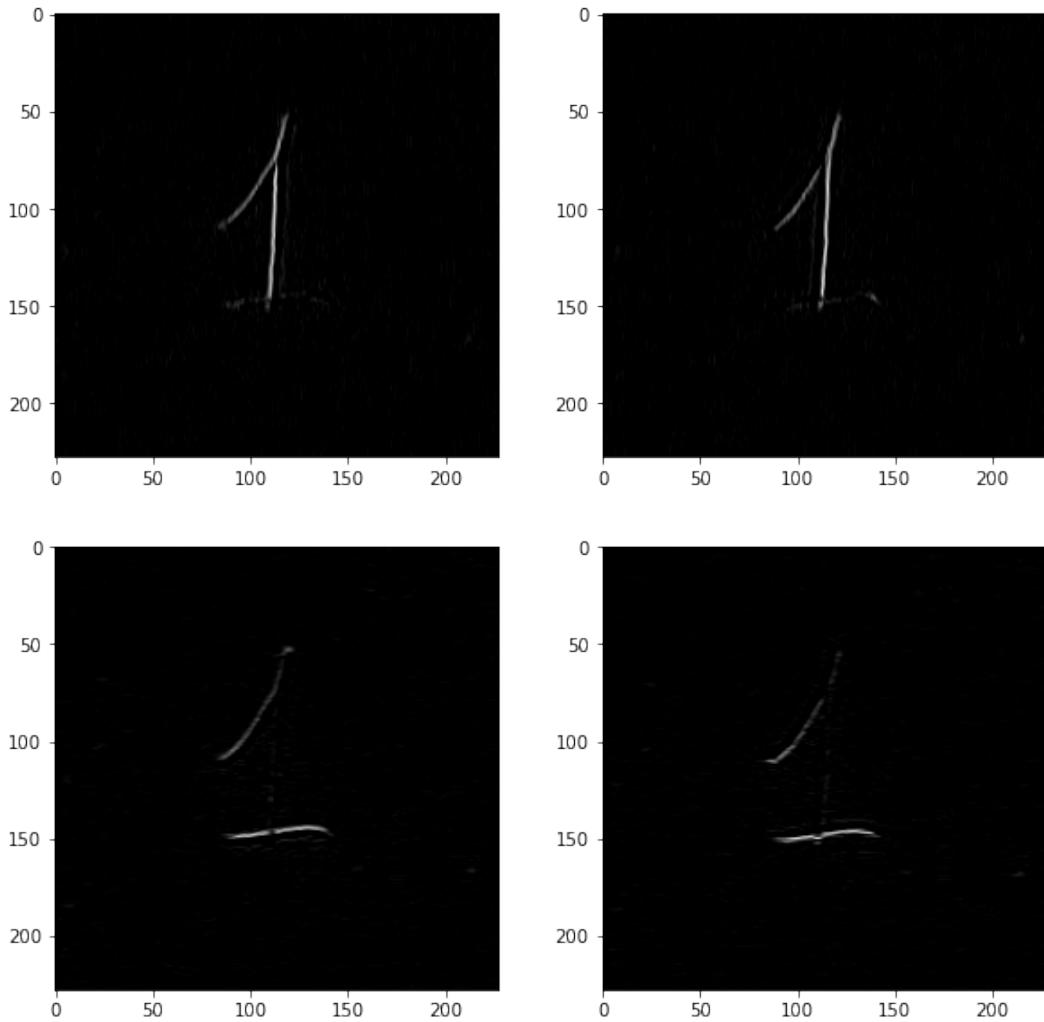
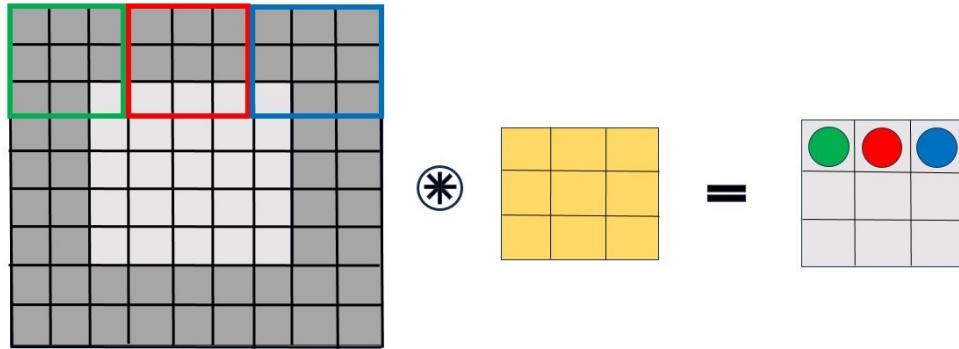


Figure 8.8: Results of the convolution product between the pixel values of the photo containing a one and the matrices L, R, T, and B.

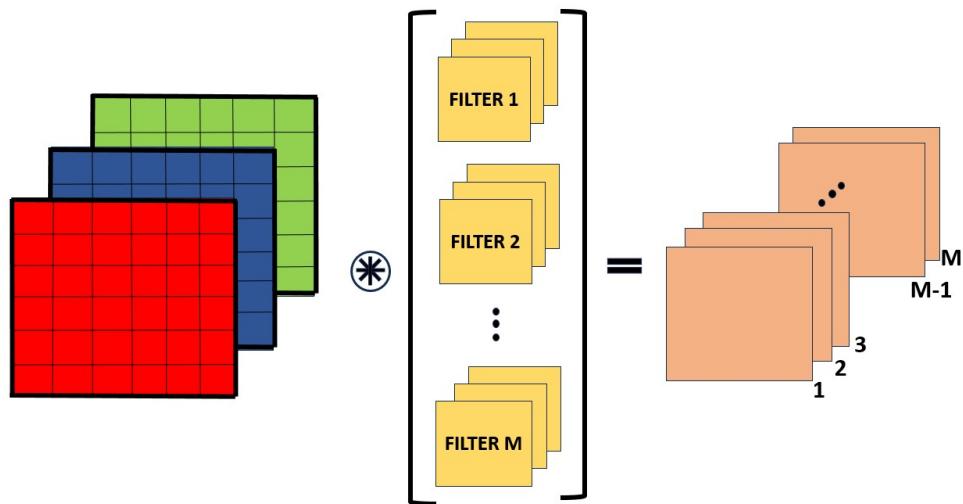
In general, a convolution operation depends on different parameters, such as the convolution matrix size ( $f$ ), the padding ( $p$ ), and the striding ( $s$ ).



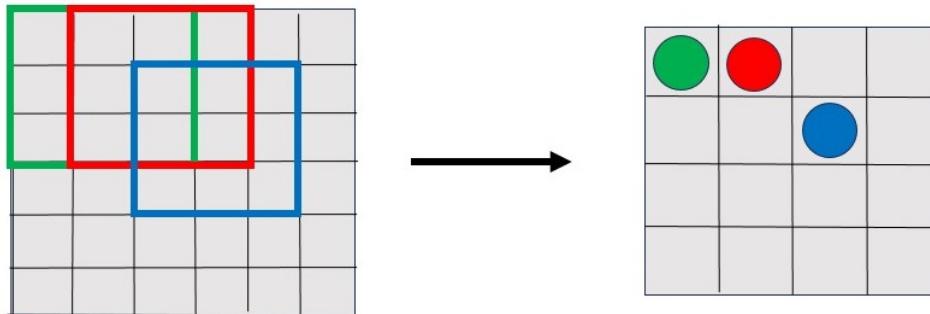
As we can see, this concrete example corresponds to  $p = 2$ ,  $s = 3$  and  $f = 3$ . The padding is introduced so that the information on the edges of a photo is preserved and taken into account on the computations. The size of the output of a convolution operation will be given by:

$$D = \frac{N + 2p - f}{s} + 1 \quad (8.16)$$

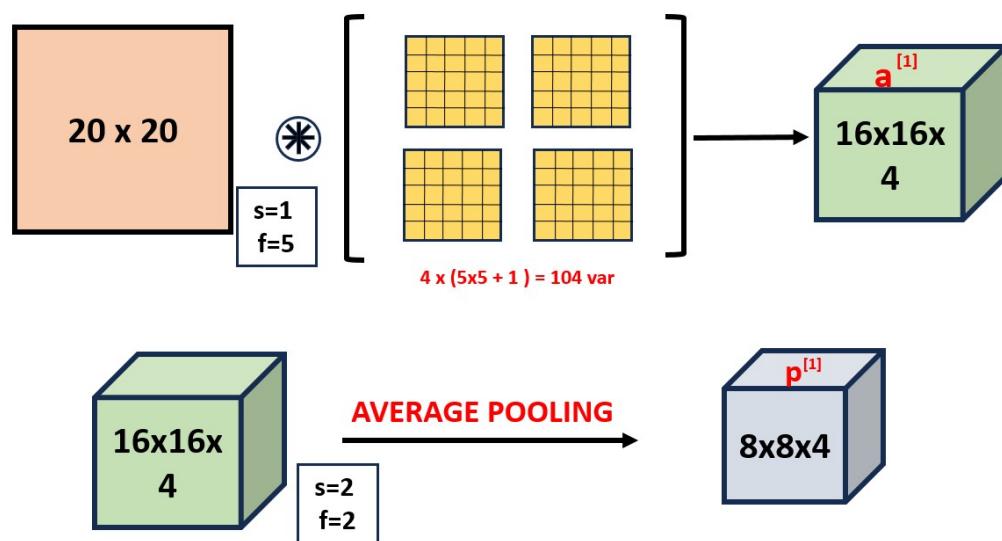
Where  $N$  is the size of the initial matrix without taking into account the paddling. There are a couple of things that must be added. First, the input of a convolution can be three-dimensional; one can imagine, for example, an RGB (Red-Green-Blue) squared photo, which will have  $N \times N \times 3$  dimensions. In this case, the convolution matrix will also be three-dimensional, with dimensions  $f \times f \times 3$  (where the width has to match the width of the input). We will call this third dimension *channel*, and the convolution matrix will be called a filter. One more thing that must be said is that a convolution operation might be composed of several filters.



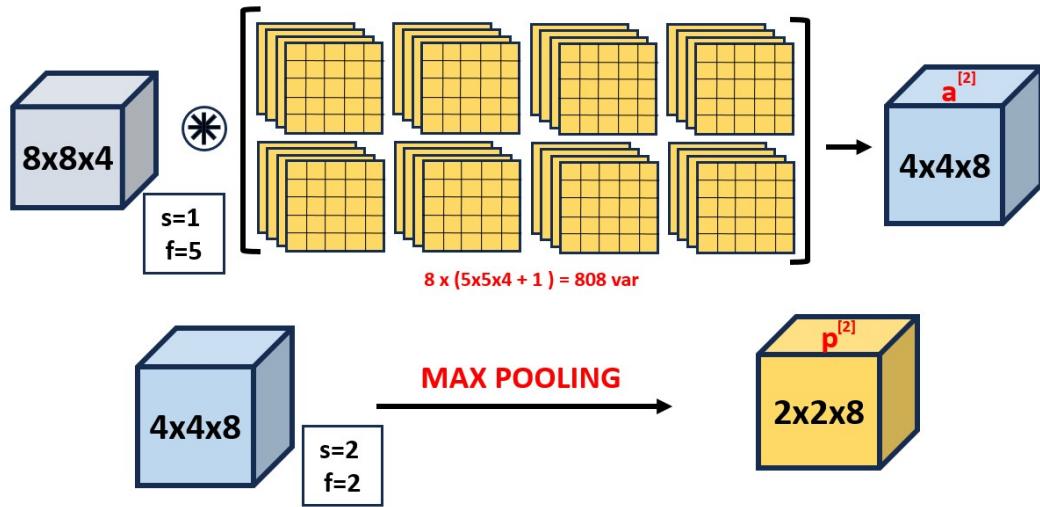
The convolution product between an input of dimensions:  $N \times N \times C$  and a filter of dimensions:  $f \times f \times C$  will result in a matrix of dimensions:  $D \times D$ . We compute the convolution between each channel of the input and filter and then sum the resulting matrices. If we have a layer with  $M$  filters, the convolution operation will result in a matrix of dimensions:  $D \times D \times M$ . Two other operations are frequent on the convolutional neural networks: the max-pooling and average pooling operations; they work similarly to the convolution operations:



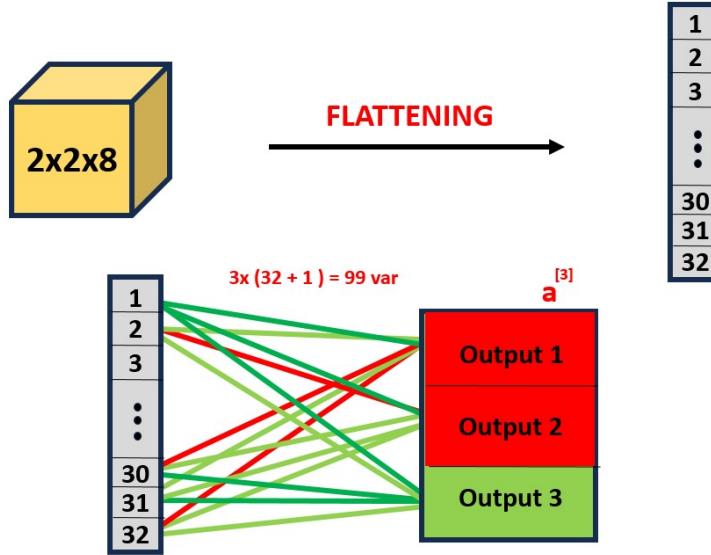
These two operations require only the input matrix. They divide the matrix into submatrices based on the  $s$ ,  $p$ , and  $f$  values. Then, either the maximum value of the submatrices or the average value is kept on the output matrix. The dimensions of the outcome will be the same as on the convolutions; it will depend on the picked sub-matrix size ( $f$ ), the padding ( $p$ ), and the striding ( $s$ ) in the same way that it does on convolutions. I will now show an example of a complete convolutional network that I will implement later. It takes 20x20 pixel grey photos as input (the same I used on the regular NN), and then the first layer is composed by a convolution of 4 filters and an average pooling:



The second layer is composed of a convolution of 8 filters and a max pooling operation:



Moreover, on the last layer, we flatten the data and then implement a regular neural network layer to produce the output:



We have about 1000 variables on this complex CNN, which is about the same as in the regular neural network, which has less complexity. This is one of the advantages of CNN over the regular NN: the same number of variables but a more complex and robust network.

Now, we have to describe the maths behind all this network. We start by defining the equations connecting the different steps.

$$Z^{[1]} = X * C^{[1]} + b^{[1]} \mid a^{[1]} = \text{ReLU}(Z^{[1]}) \mid p^{[1]} = \text{avgpool}(a^{[1]}) \quad (8.17)$$

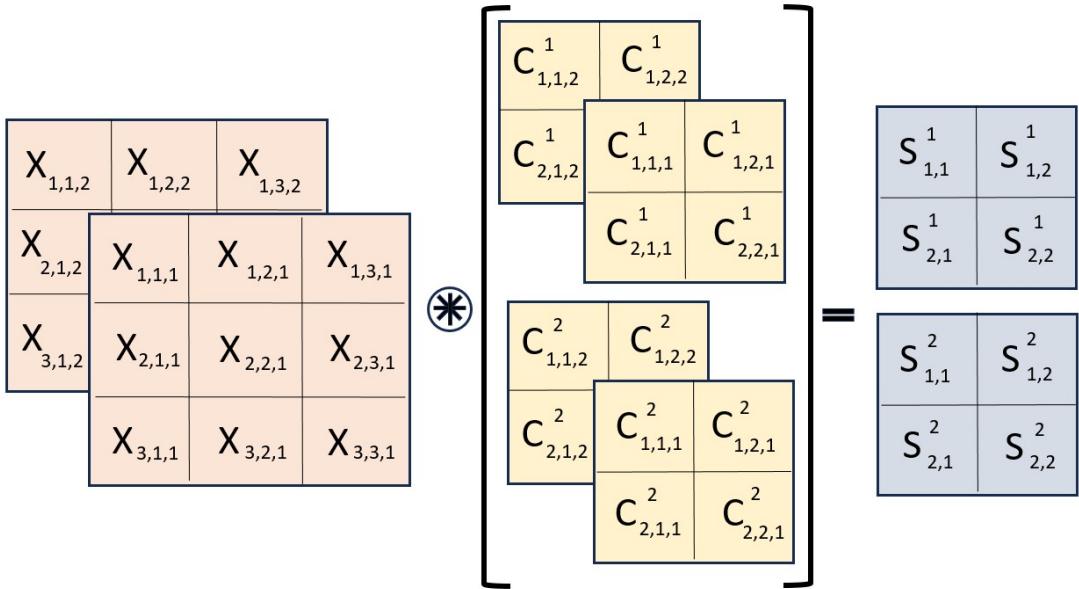
$$Z^{[2]} = p^{[1]} * C^{[2]} + b^{[2]} \mid a^{[2]} = \text{ReLU}(Z^{[2]}) \mid p^{[2]} = \text{maxpool}(a^{[2]}) \quad (8.18)$$

$$f^{[3]} = \text{flatten}(p^{[2]}) \mid Z^{[3]} = W^{[3]} f^{[3]} + b^{[3]} \mid a^{[3]} = \text{Sigmoid}(Z^{[3]}) \quad (8.19)$$

Where  $b^{[i]}$  are built so that there is only one parameter  $b$  per filter, on the first convolution,  $b^{[1]}$  will only have four terms. The sigmoid function is given by:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad (8.20)$$

Now, there is only one thing left to do; since this is a machine learning algorithm, the filters of the convolutions and all the variables involved will be initialized randomly, and we will use back-propagation to train the algorithm. Before computing these quantities, applying back-propagation on a simple convolution is an excellent exercise. Suppose the following convolution where the third lower index is the channel and the upper index refers to the different filters:



$$Z_{i,j}^k = S_{i,j}^k + b^k \quad (8.21)$$

$$\begin{aligned} Z_{1,1}^1 &= X_{1,1,1} C_{1,1,1}^1 + X_{1,2,1} C_{1,2,1}^1 + X_{2,1,1} C_{2,1,1}^1 + X_{2,2,1} C_{2,2,1}^1 \\ &\quad + X_{1,1,2} C_{1,1,2}^1 + X_{1,2,2} C_{1,2,2}^1 + X_{2,1,2} C_{2,1,2}^1 + X_{2,2,2} C_{2,2,2}^1 + b^1 \end{aligned}$$

$$Z_{1,2}^1 = X_{1,2,1} C_{1,1,1}^1 + X_{1,3,1} C_{1,2,1}^1 + X_{2,2,1} C_{2,1,1}^1 + X_{2,3,1} C_{2,2,1}^1 \\ + X_{1,2,2} C_{1,1,2}^1 + X_{1,3,2} C_{1,2,2}^1 + X_{2,2,2} C_{2,1,2}^1 + X_{2,3,2} C_{2,2,2}^1 + b^1$$

$$Z_{2,1}^1 = X_{2,1,1} C_{1,1,1}^1 + X_{2,2,1} C_{1,2,1}^1 + X_{3,1,1} C_{2,1,1}^1 + X_{3,2,1} C_{2,2,1}^1 \\ + X_{2,1,2} C_{1,1,2}^1 + X_{2,2,2} C_{1,2,2}^1 + X_{3,1,2} C_{2,1,2}^1 + X_{3,2,2} C_{2,2,2}^1 + b^1$$

$$Z_{2,2}^1 = X_{2,2,1} C_{1,1,1}^1 + X_{2,3,1} C_{1,2,1}^1 + X_{3,2,1} C_{2,1,1}^1 + X_{3,3,1} C_{2,2,1}^1 \\ + X_{2,2,2} C_{1,1,2}^1 + X_{2,3,2} C_{1,2,2}^1 + X_{3,2,2} C_{2,1,2}^1 + X_{3,3,2} C_{2,2,2}^1 + b^1$$

Four analogous equations for the second filter are obtained by changing the upper index from 1 to 2. Now, the idea is that we want to do back-propagation for a given sequence of operations:

$$X * C + b \rightarrow Z \rightarrow \mathbb{L} \quad (8.22)$$

That is, we have a convolution operation resulting in an output  $Z$ , and this will be the input for the following layer or operation, which will have as output  $\mathbb{L}$ . Since we are doing back-propagation, let us assume we have already computed  $\partial \mathbb{L} / \partial Z$  and the quantities we want to find are  $\partial \mathbb{L} / \partial C$ ,  $\partial \mathbb{L} / \partial b$  and  $\partial \mathbb{L} / \partial X$ . Remember that the last quantity might be needed in case there are previous layers that we want to compute back-propagation on. Let's compute a few terms using the chain rule for partial derivatives:

$$\frac{\partial \mathbb{L}}{\partial C_{1,1,1}^1} = \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} \frac{\partial Z_{1,1}^1}{\partial C_{1,1,1}^1} + \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} \frac{\partial Z_{1,2}^1}{\partial C_{1,1,1}^1} + \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} \frac{\partial Z_{2,1}^1}{\partial C_{1,1,1}^1} + \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \frac{\partial Z_{2,2}^1}{\partial C_{1,1,1}^1} = \\ X_{1,1,1} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{1,2,1} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{2,1,1} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{2,2,1} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.23)$$

$$\frac{\partial \mathbb{L}}{\partial C_{1,2,1}^1} = X_{1,2,1} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{1,3,1} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{2,2,1} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{2,3,1} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.24)$$

$$\frac{\partial \mathbb{L}}{\partial C_{2,1,1}^1} = X_{2,1,1} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{2,2,1} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{3,1,1} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{3,2,1} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.25)$$

$$\frac{\partial \mathbb{L}}{\partial C_{2,2,1}^1} = X_{2,2,1} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{2,3,1} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{3,2,1} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{3,3,1} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.26)$$

$$\frac{\partial \mathbb{L}}{\partial C_{1,1,2}^1} = X_{1,1,2} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{1,2,2} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{2,1,2} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{2,2,2} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.27)$$

$$\frac{\partial \mathbb{L}}{\partial C_{1,2,2}^1} = X_{1,2,2} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{1,3,2} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{2,2,2} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{2,3,2} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.28)$$

$$\frac{\partial \mathbb{L}}{\partial C_{2,1,2}^1} = X_{2,1,2} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{2,2,2} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{3,1,2} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{3,2,2} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.29)$$

$$\frac{\partial \mathbb{L}}{\partial C_{2,2,2}^1} = X_{2,2,2} \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + X_{2,3,2} \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + X_{3,2,2} \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + X_{3,3,2} \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \quad (8.30)$$

Once again, there are another eight analogous equations for the elements of the second filter, which can be easily obtained by changing the upper index from 1 to 2. One can see that these partial derivatives can be written as convolution products, as shown in the following graph:

$\frac{\partial L}{\partial C_{1,1,1}^1}$	$\frac{\partial L}{\partial C_{1,2,1}^1}$
$\frac{\partial L}{\partial C_{2,1,1}^1}$	$\frac{\partial L}{\partial C_{2,2,1}^1}$

=

X <sub>1,1,1</sub>	X <sub>1,2,1</sub>	X <sub>1,3,1</sub>
X <sub>2,1,1</sub>	X <sub>2,2,1</sub>	X <sub>2,3,1</sub>
X <sub>3,1,1</sub>	X <sub>3,2,1</sub>	X <sub>3,3,1</sub>

\*

$\frac{\partial L}{\partial Z_{1,1}^1}$	$\frac{\partial L}{\partial Z_{1,2}^1}$
$\frac{\partial L}{\partial Z_{2,1}^1}$	$\frac{\partial L}{\partial Z_{2,2}^1}$

$\frac{\partial L}{\partial C_{1,1,2}^1}$	$\frac{\partial L}{\partial C_{1,2,2}^1}$
$\frac{\partial L}{\partial C_{2,1,2}^1}$	$\frac{\partial L}{\partial C_{2,2,2}^1}$

=

X <sub>1,1,2</sub>	X <sub>1,2,2</sub>	X <sub>1,3,2</sub>
X <sub>2,1,2</sub>	X <sub>2,2,2</sub>	X <sub>2,3,2</sub>
X <sub>3,1,2</sub>	X <sub>3,2,2</sub>	X <sub>3,3,2</sub>

\*

$\frac{\partial L}{\partial Z_{1,1}^1}$	$\frac{\partial L}{\partial Z_{1,2}^1}$
$\frac{\partial L}{\partial Z_{2,1}^1}$	$\frac{\partial L}{\partial Z_{2,2}^1}$

$\frac{\partial L}{\partial C_{1,1,1}^2}$	$\frac{\partial L}{\partial C_{1,2,1}^2}$
$\frac{\partial L}{\partial C_{2,1,1}^2}$	$\frac{\partial L}{\partial C_{2,2,1}^2}$

=

X <sub>1,1,1</sub>	X <sub>1,2,1</sub>	X <sub>1,3,1</sub>
X <sub>2,1,1</sub>	X <sub>2,2,1</sub>	X <sub>2,3,1</sub>
X <sub>3,1,1</sub>	X <sub>3,2,1</sub>	X <sub>3,3,1</sub>

\*

$\frac{\partial L}{\partial Z_{1,1}^2}$	$\frac{\partial L}{\partial Z_{1,2}^2}$
$\frac{\partial L}{\partial Z_{2,1}^2}$	$\frac{\partial L}{\partial Z_{2,2}^2}$

$\frac{\partial L}{\partial C_{1,1,2}^2}$	$\frac{\partial L}{\partial C_{1,2,2}^2}$
$\frac{\partial L}{\partial C_{2,1,2}^2}$	$\frac{\partial L}{\partial C_{2,2,2}^2}$

=

X <sub>1,1,2</sub>	X <sub>1,2,2</sub>	X <sub>1,3,2</sub>
X <sub>2,1,2</sub>	X <sub>2,2,2</sub>	X <sub>2,3,2</sub>
X <sub>3,1,2</sub>	X <sub>3,2,2</sub>	X <sub>3,3,2</sub>

\*

$\frac{\partial L}{\partial Z_{1,1}^2}$	$\frac{\partial L}{\partial Z_{1,2}^2}$
$\frac{\partial L}{\partial Z_{2,1}^2}$	$\frac{\partial L}{\partial Z_{2,2}^2}$

Next, we compute the partial derivatives with respect to the b variables:

$$\frac{\partial \mathbb{L}}{\partial b^1} = \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} \frac{Z_{1,1}^1}{\partial b^1} + \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} \frac{Z_{1,2}^1}{\partial b^1} + \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} \frac{Z_{2,1}^1}{\partial b^1} + \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} \frac{Z_{2,2}^1}{\partial b^1} = \sum_{i,j} \frac{\partial \mathbb{L}}{\partial Z_{i,j}^1} \quad (8.31)$$

$$\frac{\partial \mathbb{L}}{\partial b^2} = \frac{\partial \mathbb{L}}{\partial Z_{1,1}^2} \frac{\partial Z_{1,1}^2}{\partial b^2} + \frac{\partial \mathbb{L}}{\partial Z_{1,2}^2} \frac{\partial Z_{1,2}^2}{\partial b^2} + \frac{\partial \mathbb{L}}{\partial Z_{2,1}^2} \frac{\partial Z_{2,1}^2}{\partial b^2} + \frac{\partial \mathbb{L}}{\partial Z_{2,2}^2} \frac{\partial Z_{2,2}^2}{\partial b^2} = \sum_{i,j} \frac{\partial \mathbb{L}}{\partial Z_{i,j}^2} \quad (8.32)$$

The last thing that we have to compute is the partial derivatives with respect to the input, which can be generalized using the chain rule by the following derivative product:

$$\frac{\partial \mathbb{L}}{\partial X} = \frac{\partial Z}{\partial X} \frac{\partial \mathbb{L}}{\partial Z} \quad (8.33)$$

If we compute all the individual elements:

$$\frac{\partial \mathbb{L}}{\partial X_{1,1,1}} = C_{1,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + C_{1,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^2} \quad (8.34)$$

$$\frac{\partial \mathbb{L}}{\partial X_{1,2,1}} = C_{1,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + C_{1,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + C_{1,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^2} + C_{1,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^2} \quad (8.35)$$

$$\frac{\partial \mathbb{L}}{\partial X_{1,3,1}} = C_{1,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + C_{1,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^2} \quad (8.36)$$

$$\frac{\partial \mathbb{L}}{\partial X_{2,1,1}} = C_{2,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + C_{1,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + C_{2,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^2} + C_{1,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^2} \quad (8.37)$$

$$\begin{aligned} \frac{\partial \mathbb{L}}{\partial X_{2,2,1}} &= C_{2,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^1} + C_{1,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + C_{2,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + C_{1,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} + \\ &C_{2,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,1}^2} + C_{1,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^2} + C_{2,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^2} + C_{1,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^2} \end{aligned} \quad (8.38)$$

$$\frac{\partial \mathbb{L}}{\partial X_{2,3,1}} = C_{2,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^1} + C_{1,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} + C_{2,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{1,2}^2} + C_{1,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^2} \quad (8.39)$$

$$\frac{\partial \mathbb{L}}{\partial X_{3,1,1}} = C_{2,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + C_{2,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^2} \quad (8.40)$$

$$\frac{\partial \mathbb{L}}{\partial X_{3,2,1}} = C_{2,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^1} + C_{2,1,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} + C_{2,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,1}^2} + C_{2,1,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^2} \quad (8.41)$$

$$\frac{\partial \mathbb{L}}{\partial X_{3,3,1}} = C_{2,2,1}^1 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^1} + C_{2,2,1}^2 \frac{\partial \mathbb{L}}{\partial Z_{2,2}^2} \quad (8.42)$$

Moreover, analogous equations are obtained by changing the third lower index from 1 to 2 on all the equations. All these 18 equations can be summarized with the following convolutions:

$$\begin{array}{|c|c|c|} \hline
\frac{\partial L}{\partial X_{1,1,1}} & \frac{\partial L}{\partial X_{1,2,1}} & \frac{\partial L}{\partial X_{1,3,1}} \\ \hline
\frac{\partial L}{\partial X_{2,1,1}} & \frac{\partial L}{\partial X_{2,2,1}} & \frac{\partial L}{\partial X_{2,3,1}} \\ \hline
\frac{\partial L}{\partial X_{3,1,1}} & \frac{\partial L}{\partial X_{3,2,1}} & \frac{\partial L}{\partial X_{3,3,1}} \\ \hline
\end{array} = 
\begin{array}{|c|c|c|c|} \hline
0 & 0 & 0 & 0 \\ \hline
0 & \frac{\partial L}{\partial Z_{1,1}^2} & \frac{\partial L}{\partial Z_{1,2}^2} & 0 \\ \hline
0 & \frac{\partial L}{\partial Z_{1,1}^1} & \frac{\partial L}{\partial Z_{1,2}^1} & 0 \\ \hline
0 & \frac{\partial L}{\partial Z_{2,1}^1} & \frac{\partial L}{\partial Z_{2,2}^1} & 0 \\ \hline
0 & 0 & 0 & 0 \\ \hline
\end{array} *
\begin{array}{|c|c|} \hline
C_{2,2,1}^2 & C_{2,1,1}^2 \\ \hline
C_{2,2,1}^1 & C_{2,1,1}^1 \\ \hline
C_{1,2,1}^1 & C_{1,1,1}^1 \\ \hline
\end{array}$$

$$\begin{array}{|c|c|c|} \hline
\frac{\partial L}{\partial X_{1,1,2}} & \frac{\partial L}{\partial X_{1,2,2}} & \frac{\partial L}{\partial X_{1,3,2}} \\ \hline
\frac{\partial L}{\partial X_{2,1,2}} & \frac{\partial L}{\partial X_{2,2,2}} & \frac{\partial L}{\partial X_{2,3,2}} \\ \hline
\frac{\partial L}{\partial X_{3,1,2}} & \frac{\partial L}{\partial X_{3,2,2}} & \frac{\partial L}{\partial X_{3,3,2}} \\ \hline
\end{array} = 
\begin{array}{|c|c|c|c|} \hline
0 & 0 & 0 & 0 \\ \hline
0 & \frac{\partial L}{\partial Z_{1,1}^2} & \frac{\partial L}{\partial Z_{1,2}^2} & 0 \\ \hline
0 & \frac{\partial L}{\partial Z_{1,1}^1} & \frac{\partial L}{\partial Z_{1,2}^1} & 0 \\ \hline
0 & \frac{\partial L}{\partial Z_{2,1}^1} & \frac{\partial L}{\partial Z_{2,2}^1} & 0 \\ \hline
0 & 0 & 0 & 0 \\ \hline
\end{array} *
\begin{array}{|c|c|} \hline
C_{2,2,2}^2 & C_{2,1,2}^2 \\ \hline
C_{2,2,2}^1 & C_{2,1,2}^1 \\ \hline
C_{1,2,2}^1 & C_{1,1,2}^1 \\ \hline
\end{array}$$

To summarize the results, given the quantities  $C_{i,j,c}^f$  where the upper index  $f$  refers to the filters and the index  $c$  refers to the different channels (the third input dimension), the results of the partial derivatives are:

$$\frac{\partial \mathbb{L}}{\partial \mathbb{C}_c^f} = \mathbb{X}_c * \frac{\partial \mathbb{L}}{\partial \mathbb{Z}^f} \quad (8.43)$$

$$\frac{\partial \mathbb{L}}{\partial b^f} = \sum_{i,j} \frac{\partial \mathbb{L}}{\partial Z_{i,j}^f} \quad (8.44)$$

$$\frac{\partial \mathbb{L}}{\partial \mathbb{X}_c} = \text{padded}\left(\frac{\partial \mathbb{L}}{\partial \mathbb{Z}}\right) * \mathbb{C}_c^{180^\circ} \quad (8.45)$$

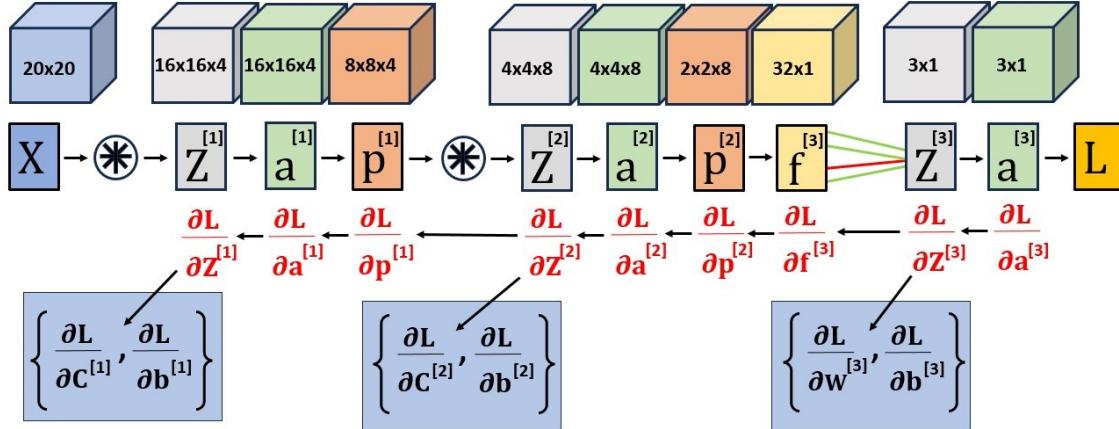
Where the notation I used is the following:

$$\mathbb{X}_c = \begin{pmatrix} X_{1,1,c} & X_{1,2,c} & \cdots & X_{1,N,c} \\ X_{2,1,c} & X_{2,2,c} & \cdots & X_{2,N,c} \\ \vdots & \vdots & \ddots & \vdots \\ X_{N,1,c} & X_{N,2,c} & \cdots & X_{N,N,c} \end{pmatrix} \quad (8.46)$$

$$\mathbb{C}_c^f = \begin{pmatrix} C_{1,1,c}^f & C_{1,2,c}^f & \cdots & C_{1,K,c}^f \\ C_{2,1,c}^f & C_{2,2,c}^f & \cdots & C_{2,K,c}^f \\ \vdots & \vdots & \ddots & \vdots \\ C_{K,1,c}^f & C_{K,2,c}^f & \cdots & C_{K,K,c}^f \end{pmatrix} \quad (8.47)$$

$$\mathbb{Z}^f = \begin{pmatrix} Z_{1,1}^f & Z_{1,2}^f & \cdots & Z_{1,M}^f \\ Z_{2,1}^f & Z_{2,2}^f & \cdots & Z_{2,M}^f \\ \vdots & \vdots & \ddots & \vdots \\ Z_{M,1}^f & Z_{M,2}^f & \cdots & Z_{M,M}^f \end{pmatrix} \quad (8.48)$$

Moreover,  $\mathbb{C}_c^{180^\circ}$  is the 3D matrix combination of all the different 2D matrices  $\mathbb{C}_c^f$  for all the possible  $f$  indexes and then rotated 180 degrees. Now that we have the results of the back-propagation for a convolution, we can do the process for a complete convolutional neural network; let's bring back a scheme of what our network looks like:



The only values we need to update are the derivatives with respect to the convolution matrices and the  $b$  parameters (derivatives on the blue boxes), but for this, we also need to compute the rest of the values, and hence, we will compute all of them one by one. The loss function that I will introduce (an alternative to the MSE) will be the following one:

$$L = - \sum_i [y_i \log(a_i^{[3]}) + (1 - y_i) \log(1 - a_i^{[3]})] \quad (8.49)$$

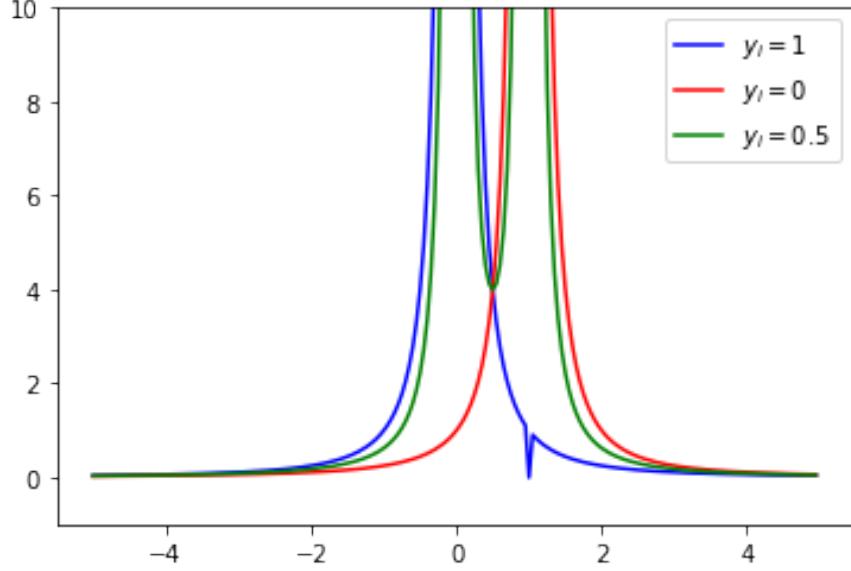
Let us see when this function is minimal by computing the derivative:

$$\frac{\partial L}{\partial a_i^{[3]}} = \frac{a_i^{[3]} - y_i}{a_i^{[3]}(1 - a_i^{[3]})} \quad (8.50)$$

The function is an extremal when  $a_i^{[3]} = y_i$ . If this turns out to be a minimum, it would mean that minimizing  $L$  would make  $a_i^{[3]}$  and the outcome converge, which is what we

are looking for. Now, we have to make sure this is a minimum, and for that, we compute the second derivative:

$$\frac{\partial^2 L}{\partial a_i^{[3]2}} = \frac{a_i^{[3]2} - 2a_i^{[3]}y_i + y_i}{a_i^{[3]2}(1 - a_i^{[3]})^2} \quad (8.51)$$



As we can see, it is bigger than zero for any given value, so we have a minimum. Now, to compute the first partial derivative, we bring back the equation connecting  $a^{[3]}$  and  $Z^{[3]}$ .

$$a_i^{[3]} = \frac{1}{1 + e^{-Z_i^{[3]}}} \quad (8.52)$$

Hence, the partial derivative reads:

$$\frac{\partial L}{\partial Z_i^{[3]}} = \sum_j \frac{\partial L}{\partial a_j^{[3]}} \frac{\partial a_j^{[3]}}{\partial Z_i^{[3]}} = \frac{\partial L}{\partial a_i^{[3]}} \frac{\partial a_i^{[3]}}{\partial Z_i^{[3]}} = \frac{a_i^{[3]} - y_i}{a_i^{[3]}(1 - a_i^{[3]})} \frac{e^{-Z_i^{[3]}}}{(1 + e^{-Z_i^{[3]}})^2} = \frac{(a_i^{[3]} - y_i)a_i^{[3]}(1 - a_i^{[3]})}{a_i^{[3]}(1 - a_i^{[3]})}$$

Moreover, we find the simplified expression:

$$\frac{\partial L}{\partial Z^{[3]}} = \mathbf{a}^{[3]} - \mathbf{y} \quad (\text{column vector}) \quad (8.53)$$

Next, we can compute the partial derivatives with respect to the first set of weights by remembering the equation:

$$Z_i^{[3]} = \sum_j W_{i,j}^{[3]} f_j^{[3]} + b_i^{[3]} \quad (8.54)$$

$$\frac{\partial L}{\partial W_{i,j}^{[3]}} = \frac{\partial L}{\partial Z_i^{[3]}} \frac{\partial Z_i^{[3]}}{\partial W_{i,j}^{[3]}} = (a_i^{[3]} - y_i)f_j^{[3]} \quad (8.55)$$

$$\frac{\partial L}{\partial \mathbf{W}^{[3]}} = \begin{pmatrix} a_1^{[3]} - y_1 \\ a_2^{[3]} - y_2 \\ \vdots \\ a_N^{[3]} - y_N \end{pmatrix} (f_1^{[3]}, f_2^{[3]} \dots f_M^{[3]}) = (\mathbf{a}^{[3]} - \mathbf{y}) \mathbf{f}^{[3]T} \quad (matrix) \quad (8.56)$$

$$\frac{\partial L}{\partial b_i^{[3]}} = \frac{\partial L}{\partial Z_i^{[3]}} \frac{\partial Z_i^{[3]}}{\partial b_i^{[3]}} = a_i^{[3]} - y_i \quad (8.57)$$

$$\frac{\partial L}{\partial \mathbf{b}^{[3]}} = \mathbf{a}^{[3]} - \mathbf{y} \quad (column \ vector) \quad (8.58)$$

Now, the last term of the third layer:

$$\frac{\partial L}{\partial f_j^{[3]}} = \sum_i \frac{\partial L}{\partial Z_i^{[3]}} \frac{\partial Z_i^{[3]}}{\partial f_j^{[3]}} = \sum_i (a_i^{[3]} - y_i) W_{i,j}^{[3]} \quad (8.59)$$

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{f}^{[3]}} &= \begin{pmatrix} W_{1,1}^{[3]} & W_{2,1}^{[3]} & \cdots & W_{N,1}^{[3]} \\ W_{1,2}^{[3]} & W_{2,2}^{[3]} & \cdots & W_{N,2}^{[3]} \\ \vdots & \vdots & \ddots & \vdots \\ W_{1,M}^{[3]} & W_{2,M}^{[3]} & \cdots & W_{N,M}^{[3]} \end{pmatrix} \begin{pmatrix} a_1^{[3]} - y_1 \\ a_2^{[3]} - y_2 \\ \vdots \\ a_N^{[3]} - y_N \end{pmatrix} = \\ &= (\mathbf{W}^{[3]})^T (\mathbf{a}^{[3]} - \mathbf{y}) \quad (column \ vector) \end{aligned} \quad (8.60)$$

Next, we can go onto the second layer; the first element is relatively straightforward since  $\mathbf{f}^{[3]} = flatten(\mathbf{P}^{[2]})$  and hence, we simply have to change shapes to compute the next term, i.e.:

$$\frac{\partial L}{\partial \mathbf{P}^{[2]}} = reshape\left(\frac{\partial L}{\partial \mathbf{f}^{[3]}}, \mathbf{P}^{[2]}\right) \quad (8.61)$$

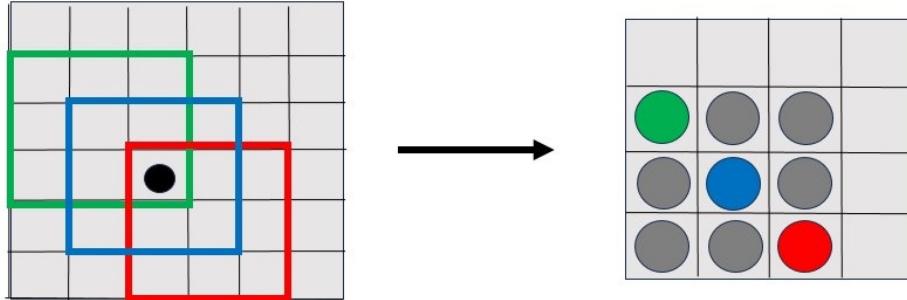
Where the function  $reshape(x, y)$  gives x the dimensions of y. For the next step, we need the equation:

$$\mathbf{P}^{[2]} = maxpool(\mathbf{a}^{[2]}) \quad (8.62)$$

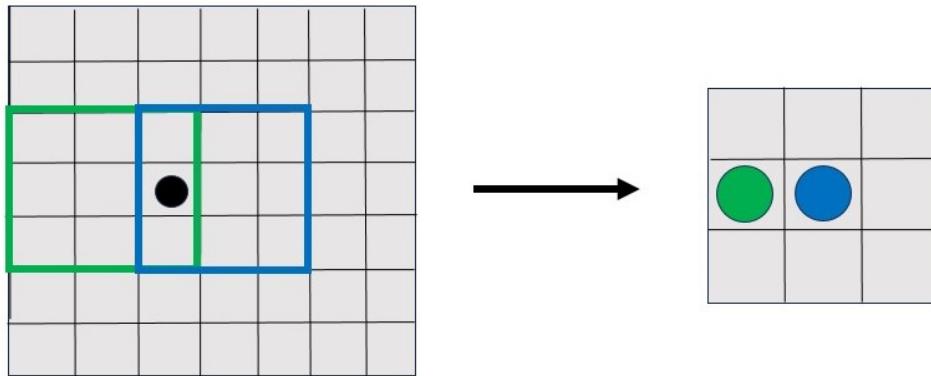
Moreover, the partial derivative reads:

$$\frac{\partial L}{\partial a_{m,n}^{l[2]}} = \sum_{i=0}^{f-1} \sum_{j=0}^{f-1} \begin{cases} \frac{\partial L}{\partial P_{\frac{m-i}{s}, \frac{n-j}{s}}^{l[2]}} & \text{if } a_{m,n}^{l[2]} = P_{\frac{m-i}{s}, \frac{n-j}{s}}^{l[2]} \\ 0 & \text{else} \end{cases} \quad (8.63)$$

Where s is the stride of the maxpool operation, and f is the size of the blocks of the maxpool operation. Let's look at the position on the image given by the fourth row and the third column ( $m,n = 3,2$ ) and consider  $s=1$  and  $f=3$ . We obtain the following indexes on the summation:  $m' = (m - i)/s = \{1, 2, 3\}$  and  $n' = (n - j)/s = \{0, 1, 2\}$ , producing precisely the nine index combinations of  $P^{[2]}$  that this term of  $a^{[2]}$  contributes to. The partial derivative of this term of  $a^{[2]}$  will be nonzero if  $a_{m,n}^{l[2]} = P_{\frac{m-i}{s}, \frac{n-j}{s}}^{l[2]}$  for at least one of the index combinations.



If we take the second picture where  $s=2$  and we take the same point, we obtain the following indexes on the summation:  $m' = \{1/2, 1, 3/2\}$  and  $n' = \{0, 1/2, 1\}$ . From all these indexes, only two combinations of integer numbers can be obtained, which can be seen in the right side figure of the graph. It goes without saying that if the indexes are higher than the length of the output matrix, smaller than zero, or noninteger, we get rid of them.



The next equation we need is  $a^{[2]} = \text{ReLU}(Z^{[2]})$ , and this one gives a straightforward answer to the following derivative:

$$\frac{\partial L}{\partial Z_{m,n}^{l[2]}} = \frac{\partial L}{\partial a_{m,n}^{l[2]}} \frac{\partial a_{m,n}^{l[2]}}{\partial Z_{m,n}^{l[2]}} = \begin{cases} \frac{\partial L}{\partial a_{m,n}^{l[2]}} & \text{if } Z_{m,n}^{l[2]} > 0 \\ a \frac{\partial L}{\partial a_{m,n}^{l[2]}} & \text{if } Z_{m,n}^{l[2]} < 0 \end{cases} \quad (8.64)$$

Where  $a \in [0, 1]$ , particularly I will choose  $a = 0.1$ . Next, we encounter the first back-propagation on a convolution, given that we have:

$$Z^{[2]} = p^{[1]} * C^{[2]} + b^{[2]} \quad (8.65)$$

Since we have already derived what the partial derivatives of all the elements involved are, we can just simply write:

$$\frac{\partial L}{\partial C_c^f} = P_c^{[1]} * \frac{\partial L}{\partial Z^{f[2]}} \quad (8.66)$$

$$\frac{\partial L}{\partial b^f} = \sum_{i,j} \frac{\partial L}{\partial Z_{i,j}^f} \quad (8.67)$$

$$\frac{\partial L}{\partial P_c^{[1]}} = padded(\frac{\partial L}{\partial Z^{[2]}}) * \mathbb{C}_c^{[2] 180^\circ} \quad (8.68)$$

With:

$$P_c^{[1]} = \begin{pmatrix} p_{1,1,c}^{[1]} & p_{1,2,c}^{[1]} & \cdots & p_{1,N,c}^{[1]} \\ p_{2,1,c}^{[1]} & p_{2,2,c}^{[1]} & \cdots & p_{2,N,c}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ p_{N,1,c}^{[1]} & p_{N,2,c}^{[1]} & \cdots & p_{N,N,c}^{[1]} \end{pmatrix} \quad (8.69)$$

$$\mathbb{C}_c^f [2] = \begin{pmatrix} C_{1,1,c}^f [2] & C_{1,2,c}^f [2] & \cdots & C_{1,K,c}^f [2] \\ C_{2,1,c}^f [2] & C_{2,2,c}^f [2] & \cdots & C_{2,K,c}^f [2] \\ \vdots & \vdots & \ddots & \vdots \\ C_{K,1,c}^f [2] & C_{K,2,c}^f [2] & \cdots & C_{K,K,c}^f [2] \end{pmatrix} \quad (8.70)$$

$$\mathbb{Z}^f [2] = \begin{pmatrix} Z_{1,1}^f [2] & Z_{1,2}^f [2] & \cdots & Z_{1,M}^f [2] \\ Z_{2,1}^f [2] & Z_{2,2}^f [2] & \cdots & Z_{2,M}^f [2] \\ \vdots & \vdots & \ddots & \vdots \\ Z_{M,1}^f [2] & Z_{M,2}^f [2] & \cdots & Z_{M,M}^f [2] \end{pmatrix} \quad (8.71)$$

Where  $\mathbb{C}_c^{[2]}$  is a 3D matrix result of combining  $\mathbb{C}_c^f [2]$  for all the f values. Moreover,  $\mathbb{C}_c^{[2] 180^\circ}$  is the result of rotating this matrix 180 degrees. Now, all that is left to do is to compute the remaining elements of the first layer; for that, we first bring the equation:

$$\mathbf{P}^{[1]} = avgpool(\mathbf{a}^{[1]}) \quad (8.72)$$

Hence, the partial derivative becomes:

$$\frac{\partial L}{\partial a_{m,n}^{l[1]}} = \sum_{i=0}^{f-1} \sum_{j=0}^{f-1} \frac{1}{f^2} \frac{\partial L}{\partial P_{\frac{m-i}{s}, \frac{n-j}{s}}^{l[1]}} \quad (8.73)$$

This equation is equivalent to the max-pooling backpropagation equation. However, now all the terms contribute averaged over and hence are divided by the number of elements in the submatrices, i.e.,  $f^2$ . The next set of equations is straightforward; it follows from the results of the previous layer:

$$\frac{\partial L}{\partial Z_{m,n}^{l[1]}} = \frac{\partial L}{\partial a_{m,n}^{l[1]}} \frac{\partial a_{m,n}^{l[1]}}{\partial Z_{m,n}^{l[1]}} = \begin{cases} \frac{\partial L}{\partial a_{m,n}^{l[1]}} & \text{if } Z_{m,n}^{l[1]} > 0 \\ a \frac{\partial L}{\partial a_{m,n}^{l[1]}} & \text{if } Z_{m,n}^{l[1]} < 0 \end{cases} \quad (8.74)$$

$$\frac{\partial L}{\partial \mathbb{C}^f [1]} = X * \frac{\partial L}{\partial \mathbb{Z}^f [1]} \quad (8.75)$$

$$\frac{\partial L}{\partial b^{f[1]}} = \sum_{i,j} \frac{\partial L}{\partial Z_{i,j}^{f[1]}} \quad (8.76)$$

Now that we have all the equations, we can wrap up the section by writing the steps of the CNN.

## STEPS

- Initialize  $C^{[m]}$  using He weights given by a Gaussian of zero mean and variance  $\sqrt{2/\text{input layer size}}$  and  $b_i^{[m]}$ , which is initialized to zero.
- We iterate until convergence is reached and do the following steps:
  - Compute  $Z^{[1]}, a^{[1]}, p^{[1]}, Z^{[2]}, a^{[2]}, p^{[2]}, f^{[3]}, Z^{[3]}, a^{[3]}$  for the different input photos on the same iteration.
  - Compute the partial derivatives of all the elements of the CNN for all the photos.
  - We average the partial derivatives  $\partial L / \partial C^{[1]}, \partial L / \partial b^{[1]}, \partial L / \partial C^{[2]}, \partial L / \partial b^{[2]}, \partial L / \partial W^{[3]}$  and  $\partial L / \partial b^{[3]}$  of all the photos and update  $C^{[1]}, b^{[1]}, C^{[2]}, b^{[2]}, W^{[3]}$  and  $b^{[3]}$  with an appropriate learning rate.

### 8.3.2 Algorithm implementation

The coding of the CNN was quite challenging as it was composed of many functions, and some of them were not trivial to implement.

```
[1]: import numpy as np
import copy as copy

class CNN():

    def __init__(self,X_train,Y_train):
        self.X_train = X_train
        self.Y_train = Y_train
        self.iterations = 3000
        self.lr = 0.1
```

The algorithm will take an array of filenames as input. This function will start reading the different files of the entry, then reshape them to a 3D matrix (1x 20 x 20) and append them to an empty array.

```
def construct_data(self,X):

    filenames = copy.deepcopy(X)
    X = []

    for pic in filenames:

        photo = cv2.imread(pic,cv2.IMREAD_GRAYSCALE)
        photo = np.reshape(photo,(1,20,20))
        X.append(photo)

    X = np.array(X)

    return(X)
```

Next, we have the core of the algorithm. First, it starts by initializing the convolution matrices, the  $b^{(m)}$  parameters, and the weights of the last layer  $W^{(3)}$ . It does this by using He weights, introduced in the previous section.

```
def main_part(self):

    self.X_train = self.construct_data(self.X_train)

    #Weight initialization
    C_1 = np.random.randn(4,1,5,5)*np.sqrt(2.0/25)
```

```

C_2 = np.random.randn(8,4,5,5)*np.sqrt(2.0/100)
W_3 = np.random.randn(3,32)*np.sqrt(2.0/32)

b_1 = np.zeros((4,16,16))
b_2 = np.zeros((8,4,4))
b_3 = np.zeros((3,1))

```

In the following step, we initialize the partial derivatives to zero on all the iterations. We will also run through all the training photos, and for each of these, we will call for the *update* function, which will return all the partial derivatives. Then, we will add the contribution of each photo to the derivatives divided by the number of training data points so that after iterating through all the training photos, we have an average correction for the parameters.

```

for it in range(self.iterations):

    delta_C_1 = np.zeros((np.shape(C_1)))
    delta_C_2 = np.zeros((np.shape(C_2)))
    delta_W_3 = np.zeros((np.shape(W_3)))
    delta_b_1 = np.zeros((np.shape(b_1)))
    delta_b_2 = np.zeros((np.shape(b_2)))
    delta_b_3 = np.zeros((np.shape(b_3)))

    count = 0

    for pic in range(len(self.X_train)):
        X = np.array(self.X_train[pic])/255
        Y = np.array(self.Y_train[pic])

        derivatives = self.update(X, Y, C_1, C_2,
                                  W_3, b_1 , b_2 , b_3)

        a_3 = derivatives[-1]

        if (abs(a_3[0]-Y[0])<=0.05 and \
            abs(a_3[1]-Y[1])<=0.05 and \
            abs(a_3[2]-Y[2])<=0.05):

            count = count + 1

    delta_C_1 = delta_C_1 + \
                np.array(derivatives[0])/len(self.X_train)
    delta_C_2 = delta_C_2 + \

```

```

        np.array(derivatives[1])/len(self.X_train)
delta_W_3 = delta_W_3 + \
            np.array(derivatives[2])/len(self.X_train)
delta_b_1 = delta_b_1 + \
            np.array(derivatives[3])/len(self.X_train)
delta_b_2 = delta_b_2 + \
            np.array(derivatives[4])/len(self.X_train)
delta_b_3 = delta_b_3 + \
            np.array(derivatives[5])/len(self.X_train)

```

Lastly, we update all the variables of the CNN with a learning rate of 0.1. There is also a mechanism to interrupt the iterations if the training data's outcome differs from the predictions  $a^{(3)}$  by a quantity smaller than 0.05.

```

C_1 = C_1 - self.lr * delta_C_1
C_2 = C_2 - self.lr * delta_C_2
W_3 = W_3 - self.lr * delta_W_3
b_1 = b_1 - self.lr * delta_b_1
b_2 = b_2 - self.lr * delta_b_2
b_3 = b_3 - self.lr * delta_b_3

if (count == len(self.X_train)):

    break

return(C_1,C_2,W_3,b_1,b_2,b_3)

```

The *update* function (which I could have also named backpropagation) is in charge of computing all the partial derivatives for a given training photo. First, we initialize the partial derivatives to zero and then compute all the different layers' quantities using Equations 8.17, 8.18, and 8.19.

```

def update(self,X, Y, C_1, C_2, W_3, b_1 , b_2 , b_3):
    Y=np.reshape(Y,(len(Y),1))

    delta_C_1 = np.zeros((np.shape(C_1)))
    delta_C_2 = np.zeros((np.shape(C_2)))
    delta_W_3 = np.zeros((np.shape(W_3)))
    delta_b_1 = np.zeros((np.shape(b_1)))
    delta_b_2 = np.zeros((np.shape(b_2)))
    delta_b_3 = np.zeros((np.shape(b_3)))

    #PHASE 1
    Z_1 = self.convolution(X,C_1,1) + b_1

```

```

a_1 = self.ReLU(Z_1)
p_1 = self.avg_pooling(a_1,2,2)
#PHASE 2
Z_2 = self.convolution(p_1,C_2,1) + b_2
a_2 = self.ReLU(Z_2)
p_2 = self.max_pooling(a_2,2,2)
#PHASE 3
f_3 = np.reshape(p_2,
                  (len(p_2)*len(p_2[0])*len(p_2[0][0]),1))
Z_3 = W_3.dot(f_3) + b_3
a_3 = self.sigmoid(Z_3)

```

Next, we have the backpropagation section, which uses some functions to compute the partial derivatives of all the quantities that appear in the CNN.

```

#BACKWARD PROPAGATION
#PHASE 3
delta_Z_3 = a_3 - Y
delta_W_3 = (a_3 - Y).dot(f_3.transpose())
delta_b_3 = a_3 - Y
delta_f_3 = W_3.transpose().dot(a_3 - Y)

#PHASE 2
delta_p_2 = np.reshape(delta_f_3, np.shape(p_2))
delta_a_2 = self.backward_maxpool(p_2,delta_p_2,a_2,2,2)
delta_Z_2 = delta_a_2 * self.ReLU_prime(Z_2)
delta_C_2 = np.zeros((np.shape(C_2)))
delta_b_2 = np.zeros((np.shape(b_2)))

for f in range(len(C_2)):
    for c in range(len(C_2[f])):

        delta_C_2[f][c] = self.conv_operation(p_1[c],
                                              delta_Z_2[f],1)
    delta_b_2[f] = np.sum(delta_Z_2[f])

padded_delta_Z_2 = np.zeros((8,12,12))
padded_delta_Z_2[0:8,4:8,4:8] = delta_Z_2
rotated_C_2 = self.rotate(C_2)

#PHASE 1
delta_p_1 = np.zeros((np.shape(p_1)))

for c in range(len(p_1)):
```

```

    for f in range(len(padded_delta_Z_2)):

        conv = self.conv_operation(padded_delta_Z_2[f] ,
                                   rotated_C_2[f][c], 1)
        delta_p_1[c] = delta_p_1[c] + conv

    delta_a_1 = self.backward_avgpool(delta_p_1, a_1, 2, 2)
    delta_Z_1 = delta_a_1 * self.ReLU_prime(Z_1)
    delta_C_1 = np.zeros((np.shape(C_1)))
    delta_b_1 = np.zeros((np.shape(b_1)))

    for f in range(len(C_1)):
        for c in range(len(C_1[f])):

            delta_C_1[f][0] = self.conv_operation(X[0] ,
                                                   delta_Z_1[f], 1)
            delta_b_1[f] = np.sum(delta_Z_1[f])

    return(delta_C_1, delta_C_2 , delta_W_3 ,
           delta_b_1 , delta_b_2 , delta_b_3,a_3)

```

The *predict* function is straightforward; it first scales down the testing data from a maximum value of 255 to one (this was also done for the training data). Then, it calls for the *main-part* function, which trains the algorithm on the training data. And last, for each testing photo, it computes all the quantities of the CNN to obtain the prediction  $a^{(3)}$ .

```

def predict(self,X_test):

    X_test = self.construct_data(X_test)
    X_test = np.array(X_test)/255

    C_1,C_2,W_3,b_1,b_2,b_3 = self.main_part()

    results = []

    for i in range(len(X_test)):

        #PHASE 1
        Z_1 = self.convolution(X_test[i],C_1,1) + b_1
        a_1 = self.ReLU(Z_1)
        p_1 = self.avg_pooling(a_1,2,2)

        #PHASE 2
        Z_2 = self.convolution(p_1,C_2,1) + b_2

```

```

a_2 = self.ReLU(Z_2)
p_2 = self.max_pooling(a_2, 2, 2)

#PHASE 3
f_3 = np.reshape(p_2,
                  (len(p_2)*len(p_2[0])*len(p_2[0][0]), 1))
Z_3 = W_3.dot(f_3) + b_3
a_3 = self.sigmoid(Z_3)

results.append(np.transpose(a_3)[0])

results = np.round(np.array(results), 2)

return(results)

```

Next, we have the ReLU function and its derivative, which are equivalent to the functions of the regular NN but implemented for 3D matrices, which will be present in this code.

```

def ReLU(self, x):
    output = np.zeros(np.shape(x))

    for a in range(len(x)):
        for b in range(len(x[a])):
            for c in range(len(x[a][b])):

                if (x[a][b][c] < 0):
                    output[a][b][c] = x[a][b][c] / 10
                else:
                    output[a][b][c] = x[a][b][c]
    return(output)

def ReLU_prime(self, x):
    output = np.zeros(np.shape(x))

    for a in range(len(x)):
        for b in range(len(x[a])):
            for c in range(len(x[a][b])):

                if (x[a][b][c] < 0):
                    output[a][b][c] = 0.1
                else:
                    output[a][b][c] = 1.0

```

```
    return(output)
```

The *convolution-operation* function computes the convolution explained in Equation 8.15. It takes two 2D matrices and gives the regular convolution as the output.

```
def conv_operation(self,X,Y,striding):  
  
    sizing = len(Y)  
    D = int((len(X[0])-sizing)/striding + 1)  
    conv = np.zeros((D,D))  
  
    for i in range(D):  
        for j in range(D):  
            strided_i = striding*i  
            strided_j = striding*j  
            conv[i][j] = np.sum(X[strided_i:strided_i \   
                +sizing,strided_j:strided_j+sizing]*Y)  
  
    return(conv)
```

This function will use the *convolution-operation* function to compute the convolution between matrices of higher dimensions. In particular, the convolution between a 3D matrix of dimensions ( $C \times N \times N$ ) and a 4D matrix of dimensions ( $f \times C \times M \times M$ ).

```
def convolution(self,X,Y,striding):  
    #X dimensions (c,n,n)  
    #Y dimensions (f,c,m,m)  
    # X*Y dimensions (f,D,D) with D = (n-f)/s + 1  
  
    sizing = len(Y[0][0])  
    D = int((len(X[0])-sizing)/striding + 1)  
    convolution_matrix=[]  
  
    for f in range(len(Y)):  
        total_conv = np.zeros((D,D))  
        for c in range(len(Y[0])):  
  
            conv = self.conv_operation(X[c,:,:],  
                Y[f,c,:,:],striding)  
            total_conv = total_conv + conv  
        convolution_matrix.append(total_conv)  
    convolution_matrix = np.array(convolution_matrix)  
  
    return(convolution_matrix)
```

The *avg-pooling* and *max-pooling* functions are similar. They take as input a 3D matrix of dimensions ( $C \times N \times N$ ), the striding  $s$ , and the size  $f$  of the submatrices. Then, it iterates over the channel index and generates the corresponding output matrices, computing either the maximum or average value of the submatrices determined by  $f$  and  $s$ .

```

def avg_pooling(self,X,striding,sizing):
    #X dimensions (c,n,n)
    AP_X = []
    for c in range(len(X)):

        X_c = X[c]
        D = int((len(X[0])-sizing)/striding + 1)
        AP_X_c = np.zeros((D,D))

        for i in range(D):
            for j in range(D):

                strided_i = striding*i
                strided_j = striding*j
                AP_X_c[i][j] = np.sum(X_c[strided_i:strided_i\
                    +sizing,strided_j:strided_j+sizing]\n                    sizing])/ (sizing**2.0)

        AP_X.append(AP_X_c)
    AP_X = np.array(AP_X)
    return(AP_X)

def max_pooling(self,X,striding,sizing):
    #X dimensions (c,n,n)
    AP_X = []
    for c in range(len(X)):

        X_c = X[c]
        D = int((len(X[0])-sizing)/striding + 1)
        AP_X_c = np.zeros((D,D))

        for i in range(D):
            for j in range(D):

                strided_i = striding*i
                strided_j = striding*j
                AP_X_c[i][j] = np.max(X_c[strided_i:strided_i\
                    +sizing,strided_j:strided_j+sizing])

```

```

AP_X.append(AP_X_c)
AP_X = np.array(AP_X)
return(AP_X)

```

The *Sigmoid* function will be used to transition from  $Z^{(3)}$  to  $a^{(3)}$ . It uses the function of Equation 8.20.

```

def sigmoid(self,X):

    sigmoid_X = np.zeros((np.shape(X)))

    for i in range(len(X)):
        for j in range(len(X[0])):

            sigmoid_X[i][j] = 1/(1+np.exp(-X[i][j]))
    return(sigmoid_X)

```

The *backward-maxpooling* function will be used to perform backpropagation on the max-pooling step, which transitions from a 3D matrix of dimensions  $(N \times N \times C) a^{(2)}$  to another 3D matrix of dimensions  $(M \times M \times C) p^{(2)}$ . It runs through all the elements of  $a_{m,n}^c$ ; then it generates all the indexes  $m' = (m - i)/s$  and  $n' = (n - j)/s$  where  $i$  and  $j$  runs from zero to the sizing value minus one. Then, it checks if the indexes  $m'$  and  $n'$  are within the matrix range and if they are integers. Lastly, it checks if  $a_{m,n}^c$  is equal to  $p_{m',n'}^c$  and if this is the case, it adds the contribution to the partial derivative.

```

def backward_maxpool(self,p,delta_p,a,s,f):
    delta_a = np.zeros((np.shape(a)))
    for c in range(len(a)):
        for m in range(len(a[c])):
            for n in range(len(a[c][m])):
                for i in range(f):
                    for j in range(f):

                        m_p = (m-i)/s
                        n_p = (n-j)/s

                        if m_p >= 0 and n_p >= 0 \
                        and m_p < len(delta_p[c]) \
                        and n_p < len(delta_p[c]) \
                        and (int(m_p)-m_p)==0 \
                        and (int(n_p)-n_p)==0 \
                        and a[c][m][n]==p[c][int(m_p)][int(n_p)]:

                            delta_a[c][m][n] = delta_a[c][m][n] \

```

```

        + delta_p[c][int(m_p)][int(n_p)]
return(delta_a)

```

The *backward-avgpooling* function is similar to the *backward-maxpooling* function but it adds all the contributions divided by number of elements in the submatrix, i.e.,  $f^2$ .

```

def backward_avgpool(self,delta_p,a,s,f):
    delta_a = np.zeros((np.shape(a)))
    for c in range(len(a)):
        for m in range(len(a[c])):
            for n in range(len(a[c][m])):
                for i in range(f):
                    for j in range(f):

                        m_p = (m-i)/s
                        n_p = (n-j)/s

                        if m_p >= 0 and n_p >= 0 \
                        and m_p < len(delta_p[c])\
                        and n_p <len(delta_p[c])\
                        and (int(m_p)-m_p)==0 \
                        and (int(n_p)-n_p)==0 :

                            delta_a[c][m][n] = delta_a[c][m][n]\ 
                            + delta_p[c][int(m_p)][int(n_p)]/(f**2.0)

    return(delta_a)

```

Lastly, we have the function that rotates the matrix by 180 degrees, which will be needed to compute some partial derivatives such as  $\frac{\partial L}{\partial P_c^{[1]}}$ .

```

def rotate(self,X):
    output = np.zeros((np.shape(X)))
    for f in range(len(X)):
        for c in range(len(X[f])):
            submatrix = X[f][c]
            M = len(submatrix)
            N = len(submatrix[0])
            for i in range(M):
                for j in range(N):
                    output[f][c][M-1-i][j]=submatrix[i][N-1-j]

    return(output)

```

After implementing the class of the CNN, we load the photos and their outcomes into a variable pair: X and Y.

```
[2]: import cv2
import numpy as np

X = ["1_1.png", "1_2.png", "1_3.png", "1_4.png", "1_5.png",
     "1_6.png", "1_7.png", "1_8.png", "1_9.png", "1_10.png",
     "1_11.png", "1_12.png", "1_13.png", "1_14.png",
     "2_1.png", "2_2.png", "2_3.png", "2_4.png", "2_5.png",
     "2_6.png", "2_7.png", "2_8.png", "2_9.png", "2_10.png",
     "2_11.png", "2_12.png", "2_13.png", "2_14.png",
     "3_1.png", "3_2.png", "3_3.png", "3_4.png", "3_5.png",
     "3_6.png", "3_7.png", "3_8.png", "3_9.png", "3_10.png",
     "3_11.png", "3_12.png", "3_13.png", "3_14.png"]

Y = np.array([[1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
              [1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
              [1,0,0],[1,0,0],[1,0,0],[1,0,0],[1,0,0],
              [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
              [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
              [0,1,0],[0,1,0],[0,1,0],[0,1,0],[0,1,0],
              [0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],
              [0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1],
              [0,0,1],[0,0,1],[0,0,1],[0,0,1]])
```

The cross-validation function of the CNN model is identical to the function of the regular Neural Network.

```
[3]: from sklearn.model_selection import KFold

def K_fold_cross_validation_KNN(X,Y,iterations):

    test_accuracy = []

    for randomstates in range(iterations):

        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])
            X_test = np.array([X[i] for i in test_index])
```

```

Y_test = np.array([Y[i] for i in test_index])

reg = CNN(X_train,Y_train)

# PREDICTIONS
Y_pred = reg.predict(X_test)

Y_test = np.array([np.where(Y_test[i] == \
                           np.max(Y_test[i]))[0][0] \
                           for i in range(len(Y_test))])
Y_pred = np.array([np.where(Y_pred[i] == \
                           np.max(Y_pred[i]))[0][0] \
                           for i in range(len(Y_pred))])

# ERRORS
error = Y_test - Y_pred

# CORRECT PREDICTIONS
correct_pred = np.count_nonzero(error == 0)

test_accuracy.append(100*correct_pred/len(test_index))

return(np.mean(test_accuracy))

```

If we use the cross-validation function for the first 20 random states, we find an accuracy on the testing data of 98.93%, which is way higher than the accuracy of the regular Neural Network or the Eigenfaces model.

[4]: `print("The accuracy of the CNN algorithm is:",  
 K_fold_cross_validation_KNN(X,Y,20))`

The accuracy of the CNN algorithm is: 98.93055555555554

Remember that in the introduction of the CNN, we showed a few C matrices and learned that they pick on different patterns of the photos, such as edges. Hence, I decided to visualize the patterns that the implemented Network picks up. I did this by obtaining the variables of the Network:  $C^{[1]}$ ,  $C^{[2]}$ , and so on, and computing the different transformations of the Network for a photo of a one, two, and three. Remember that these are not matrices we chose; they are obtained with backpropagation by the model with a randomized starting point. Hence, as shown in the graph below, there are no clear patterns for the human eye, especially in the later steps of the Neural Network.

[5]: `reg = CNN(X,Y)  
C_1,C_2,W_3,b_1,b_2,b_3 = reg.main_part()`

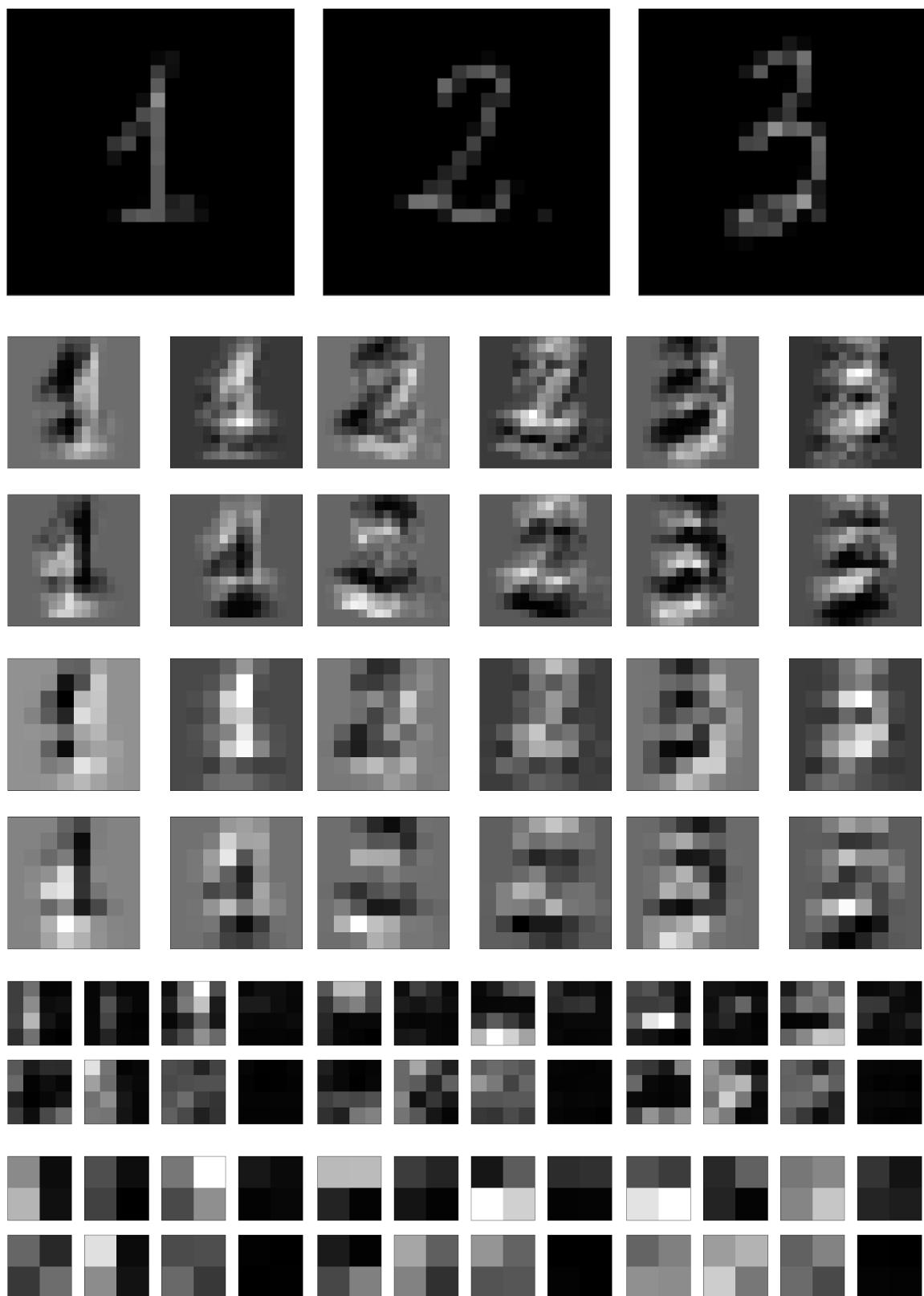


Figure 8.9: Visualization of the transformations  $a^{[1]}, p^{[1]}, a^{[2]}, p^{[2]}$  for a photo of a one, two and three.

We can see the initial photos of a one, two, and three on the top. Then, the result of applying the four filters  $C^{[1]}$  to each photo. Next, the average pooling of the previous step reduces the dimensions of the images from  $16 \times 16$  to  $8 \times 8$ . Moreover, the last two parts of the graph result from applying the eight filters  $C^{[2]}$  and then the max-pooling operation on these results. The visualization of the last layer is not shown as it is composed of a regular NN, and there will not be any interpretability for the human eye. In that last layer, the algorithm takes those  $8 \times 4$  pixels on the last row of images, flattens them, and produces an outcome given by an array of three elements for each photo. It is worth mentioning that the plots in Figure 8.9 are made with a gray colormap where the minimum and maximum values correspond to the minimum and maximum values of the matrix, which means that the color black could represent a value smaller than zero and the white color a value higher than one.

# Chapter 9

## Evaluation metrics

[35][47] To finish the book's contents, I included a small chapter discussing the evaluation metrics that might be useful when working with an ML model. We only worked with a few different data collections, so we could not explore the different metrics in previous sections, and we limited ourselves to the Mean Squared Error for regression and the accuracy and the log-loss function for classification. Remember that metrics can be used to assess a model's performance and also to build a minimization problem in order to derive the model's equations.

### 9.1 Regression metrics

#### 9.1.1 Mean Squared Error (MSE)

We will start with the MSE metric for regression, which we are most familiar with since it was used for all the regression algorithms of previous chapters. The expression for the MSE reads:

$$MSE = \frac{\sum_{j=1}^M (\hat{y}_j - y_j)^2}{M} \quad (9.1)$$

- **Advantages:** Since the MSE squares the deviation between the outcome and the prediction, if an outlier's error (a point far from the data cluster) is too large, the MSE becomes sensitive to it. Hence, when testing different hyperparameters or models, the MSE could help identify if the model takes outliers into account or discards them when training, based on the impact the prediction of outliers have on the MSE.
- **Disadvantages:** The negative side to the advantage we just mentioned is that a single testing point located out of the data cluster can increase the MSE's value, removing clarity on the performance of our model.

In the following plot, we can see a visualization of why outliers lead to a higher error, which is further increased when we compute the squared error on the MSE.

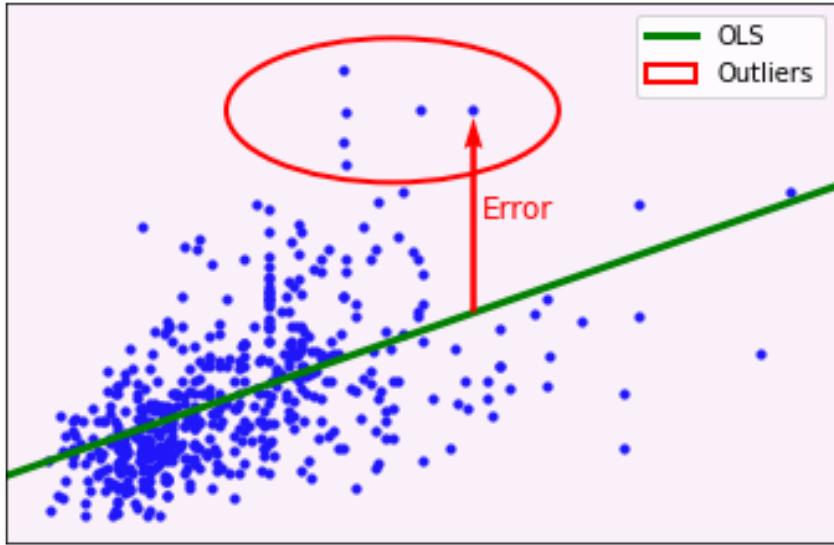


Figure 9.1: Visualization of outliers on a data collection.

### 9.1.2 Root Mean Squared Error (RMSE)

The Root Mean Squared Error is often used over the MSE as it has the same drawbacks but offers a significant advantage regarding the interpretability of the results. The expression for the RMSE is the following:

$$RMSE = \sqrt{\frac{\sum_{j=1}^M (\hat{y}_j - y_j)^2}{M}} \quad (9.2)$$

- **Advantages:** The RMSE has the same advantages as the MSE but offers one additional advantage over the latter. The RMSE has the same units as the outcome and is the equivalent of the standard deviation, making it easier to interpret the results as we can read how spread the data is.
- **Disadvantages:** The drawback of the RMSE is that, similarly to the MSE, it squares the errors and is sensitive to the outliers.

### 9.1.3 Mean Absolute Error (MAE)

The following classification metric is the MAE, which will not have squared errors in contrast to the MSE or RMSE. The expression for this metric reads:

$$MAE = \frac{\sum_{j=1}^M |\hat{y}_j - y_j|}{M} \quad (9.3)$$

- **Advantages:** The advantage of the MAE is that it is not sensitive to the outliers, and hence, a few outliers do not influence this quantity since the errors are not squared and all the contributions have the same weight. Among the advantages, it is also worth mentioning that MAE has the same units as the outcome.

- **Disadvantages:** The negative side to this metric is that, as we have seen on the Lasso model, including the absolute value of a magnitude often leads to difficulties when minimizing it and sometimes requires applying gradient descent instead of having an analytical solution.

### 9.1.4 Root Mean Squared Logarithmic Error (RMSLE)

Next, we have the Root Mean Squared Logarithmic Error, which has the following expression:

$$RMSLE = \sqrt{\frac{1}{M} \sum_{j=1}^M \left[ \log\left(\frac{y_j + 1}{\hat{y}_j + 1}\right) \right]^2} \quad (9.4)$$

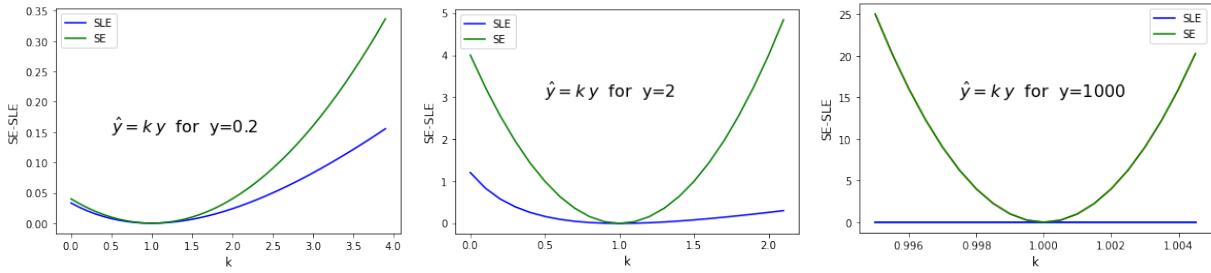


Figure 9.2: Squared Error and Squared Logarithmic Error vs the proportionality constant between the outcome and the prediction of a point.

On the previous graph I have plotted the quantities  $SE = (y - \hat{y})^2$  and  $SLE = \left[ \log\left(\frac{y+1}{\hat{y}+1}\right) \right]^2$ . As we can see, the RMSLE is a lower bound of the RMSE.

- **Advantages:** The advantage of this metric is that it treats minor differences between outcomes and predictions the same way, no matter if the magnitude of the outcomes is high or small. This does not happen with the RMSE. If we have a slight deviation of 1% between the outcome and the prediction and both these values are large, the RMSE will grow rapidly while the RMSLE is negligible due to the log coefficient converging to 1.
- **Disadvantages:** As we can see on the second graph, one of the disadvantages of this metric is that it penalizes underestimates more than overestimates. For  $k = 1.5$  we have that  $\hat{y} = 1.5 y$  and hence  $|Error| = 0.5 y$ . For  $k = 0.5$  we have that  $\hat{y} = 0.5 y$  and hence  $|Error| = 0.5 y$  as well. However, we can see that the RMSLE is greater for the latter.

### 9.1.5 R-Squared factor ( $R^2$ )

[17] The R-squared factor, also denoted as  $R^2$ , is given by the residuals (or errors on the prediction) divided by a quantity proportional to the variance of the outcomes.  $R^2$  is often computed on linear regressions to assess if the model is unbiased (i.e., a good balance between overfitting and underfitting).

$$R^2 = 1 - \frac{\sum_{j=1}^M (y_j - \hat{y}_j)^2}{\sum_{j=1}^M (y_j - \langle y \rangle)^2} \quad (9.5)$$

$\langle y \rangle$  is the average of the actual outcomes. One interpretation of the R-squared value is that it tells us the percentage of the dependent variable's (outcome) variance that can be explained by the independent variables. Whenever the predictions match the outcomes completely, we will get  $R^2 = 1$ . If the prediction is equal to the average of the outcomes, we will find  $R^2 = 0$ , which we will take as the baseline performance. In some cases, we can find  $R^2 < 0$  for a model with higher residuals than the baseline. For an unbiased model,  $R^2$  will generally be between 0 and 1.

- **Advantages:** The main advantage of the R-squared factor is that it will usually be on the interval  $[0,1]$  in contrast to other evaluation metrics that are not bounded. We also have a baseline value of the  $R^2$  to compare with our results.
- **Disadvantages:** The drawback of this metric is that it is hard to interpret if a model is a good fit or not based on the  $R^2$  value only, as this highly depends on the data collection. A value of the R-squared factor of 0.9 might sometimes represent a good fit, while in other cases, a value of 0.5 might be the best configuration. On the other hand, this metric tends to grow higher as we increase the number of variables of the data collection, which eclipses the interpretation of the results even more, as we can increase  $R^2$  by including linearly dependent variables.

### 9.1.6 Adjusted R-Squared factor ( $\bar{R}^2$ )

The adjusted R-squared factor corrects the regular R-squared value. It includes a dependency on the variables to account for the increase on  $R^2$  as we include variables in the data collection.

$$\bar{R}^2 = 1 - (1 - R^2) \frac{N - 1}{N - D - 1} \quad (9.6)$$

- **Advantages:**  $\bar{R}^2$  will have  $R^2$  as the upper bound. The extra advantage of the adjusted R-squared factor over  $R^2$  is that it does not increase when including more variables. If it increases, it is only due to the variable adding vital information to the regression.
- **Disadvantages:**  $\bar{R}^2$  shares most of the drawbacks of the regular R-squared factor, excluding the tendency to increase when including variables.

## 9.2 Classification metrics

### 9.2.1 Accuracy (ACC)

We are already familiar with this classification metric as we have used it in all the classification algorithms we have worked with. The expression reads:

$$Acc = \frac{\text{Nº of correct predictions}}{\text{Total predictions}} = \frac{TP + TN}{TP + TN + FP + FN} \quad (9.7)$$

We have the following definitions where one can define an outcome being positive if it equals one and negative if it equals zero:

- TP: The number of *true positive* values, i.e., values where both the outcome and the prediction are equal to one.
- TN: The number of *true negative* values, i.e., values where both the outcome and the prediction are equal to zero.
- FP: The number of *false positive* values, i.e., values where the outcome is negative (zero) and the prediction is positive (one).
- FN: The number of *false negative* values, i.e., values where the outcome is positive (one) and the prediction is negative (zero).

Next, we will see the main advantages and disadvantages of the accuracy metric, which we did not have to worry about in previous chapters as the iris data collection is well-balanced.

- **Advantages:** The main advantage of the accuracy is that it is easy to interpret as it gives the percentage of points that have been correctly predicted.
- **Disadvantages:** The disadvantage of this evaluation metric for classification is that it does not consider the size of the different classes. Suppose we have a data collection with 99 data points of class A and 1 point of class B. Most models will predict all the points to belong to the first class, achieving 99% accuracy. In general, this is not a good measurement when the different classes in a data collection are not balanced.

### 9.2.2 Logarithmic Loss/ Cross-Entropy Loss (LL)

[28] The logarithmic loss is another function we are familiar with, as we have seen it in several algorithms. The equation for this reads:

$$LL = \sum_{i=1}^N -[y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)] \quad (9.8)$$

As shown in the following graph, this function is minimal when the outcomes  $y_i$  and the predictions  $p_i$  match and grow as the difference between these two quantities increases.

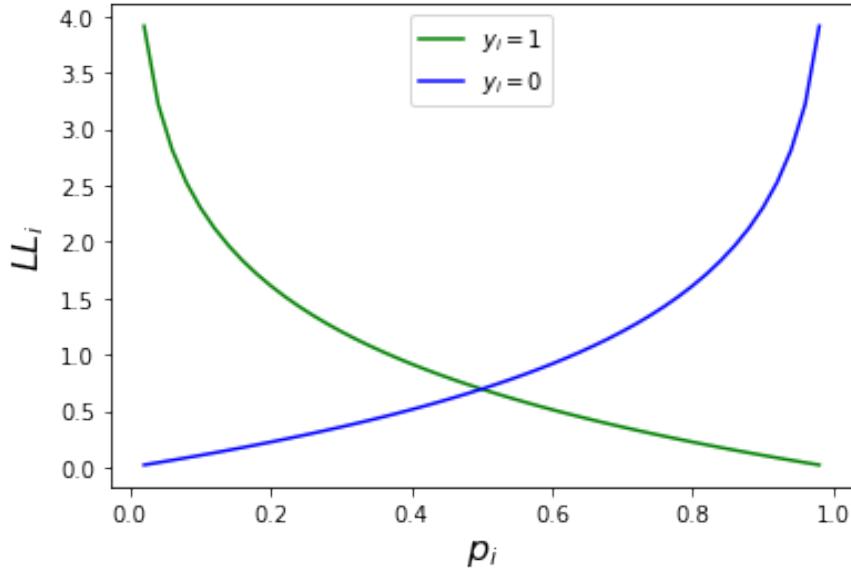


Figure 9.3: Visualization of the logarithmic loss function.

- **Advantages:** One of the advantages of the log-loss function is that it is easy to implement. For example, as we saw on the CNN model, it leads to a straightforward equation for the partial derivative. In general, it is also true that gradient descent behaves well with log-loss as it is a continuous and smooth convex function. Another positive side is that it quantifies the error in the predictions. For example, if a point has an outcome of one and its predictive probability is 0.95, that 0.05 contributes to increasing the log-loss function. So, we can, for example, analyze the difference between two models with the same accuracy, as one may have a lower log loss than the other.
- **Disadvantages:** The main drawback of this metric is that it cannot be used alone to determine the performance of a model. We can imagine, for example, a model that predicts all the testing points correctly, having an accuracy of 100%, but it predicts all the points with  $|y_i - p_i| = 0.4$ . The log-loss function would be substantial despite having perfect accuracy. However, the log-loss function is an excellent complementary metric to assess a model's performance.

### 9.2.3 ROC-AUC

[7] Despite not having used this metric, it is one of the most common and popular metrics to measure the performance of a model. First, we will define the following quantities:

- True positive rate:  $TPR = TP / (TP + FN)$ . It gives us the fraction of positive points that are correctly classified.
- False positive rate:  $FPR = FP / (FP + TN)$ . It gives us the fraction of negative points that are incorrectly classified.

The idea behind the ROC curve is that we will plot TPR vs FPR for different classification thresholds. What this means is that instead of classifying a point as positive whenever its probability is greater than 0.5 and as negative otherwise. We will do this for different thresholds other than 0.5, going from zero to one.

Then, we have the AUC factor, which stands for *Area Under the Curve* of the ROC curve. The AUC factor will tell us how good a model is; the closer the value is to one, the better our model is. The AUC value represents the probability of a random positive point having a higher predictive probability than a random negative point. So, if we have  $AUC = 1$ , there is a 100% probability that a random positive point has a higher predictive probability than a random negative point.

AUC	[0.5,0.6]	[0.6,0.7]	[0.7,0.8]	[0.8,0.9]	[0.9,1.0]
Rating	Bad	Poor	Decent	Good	Excellent

Table 9.1: Rating of a model based on the value of the AUC.

This metric for classification has many advantages, some of which are the following: First, it does not require us to set a threshold as it takes all the thresholds into account. Secondly, it performs well when there is a class imbalance in the data collection.

Next, we will see all these concepts with an example, but first, we need to build some functions. We will build a function to merge cross-validation for the Logistic regression and the ROC metric. There are also a couple of auxiliary functions: the *integral* function, which computes the integral of the curve given by a variable and outcome pair X, Y. And another function that assigns a given probability  $p_i$  the value zero or one given a certain threshold.

```
[1]: from sklearn.model_selection import KFold

def integral(X,Y):
    Y = [x for _,x in sorted(zip(X,Y))]
    X = np.sort(X)

    integral = 0

    for i in range(len(X)-1):
        integral = integral + (X[i+1]-X[i])*np.mean(Y[i:i+2])

    return(integral)
```

```
[1]: def threshold_function(x,threshold):
    if (x < threshold):
        return(0.0)
    else:
        return(1.0)

def ROC_Logistic(X,Y,iterations,threshold):
    TPR_values = []
    FPR_values = []

    for randomstates in range(iterations):
        KF = KFold(n_splits=5,shuffle=True,random_state=randomstates)
        KF.get_n_splits(X)

        for i, (train_index, test_index) in enumerate(KF.split(X)):

            X_train = np.array([X[i] for i in train_index])
            Y_train = np.array([Y[i] for i in train_index])
            X_test = np.array([X[i] for i in test_index])
            Y_test = np.array([Y[i] for i in test_index])

            reg = logistic(X_train,Y_train,200)
            ypred = reg.predict(X_test)
            ypred = np.array([[threshold_function(ypred[i],threshold)] \
                             for i in range(len(ypred))])

            # Compute TP, TN, FP and FN
            TP = np.count_nonzero((ypred == 1) & (Y_test == 1))
            TN = np.count_nonzero((ypred == 0) & (Y_test == 0))
            FP = np.count_nonzero((ypred == 1) & (Y_test == 0))
            FN = np.count_nonzero((ypred == 0) & (Y_test == 1))

            # Compute TPR and FPR
            TPR = TP/(TP + FN)
            FPR = FP/(FP + TN)

            TPR_values.append(TPR)
            FPR_values.append(FPR)
```

```
    return(np.mean(TPR_values),np.mean(FPR_values))
```

If we load the one-dimensional breast cancer dataset and use the *ROC-logistic* function for different threshold values and then compute the integral on the results, we obtain the following AUC and plot.

```
[2]: threshold = np.arange(0,1.05,0.05)

FPR_values = []
TPR_values = []

for t in threshold:

    TPR,FPR = ROC_Logistic(X,Y,10,t)

    FPR_values.append(FPR)
    TPR_values.append(TPR)

print("The area under the ROC curve (AUC) is ",
      integral(FPR_values,TPR_values))
```

```
The area under the ROC curve (AUC) is 0.9328438904414458
```

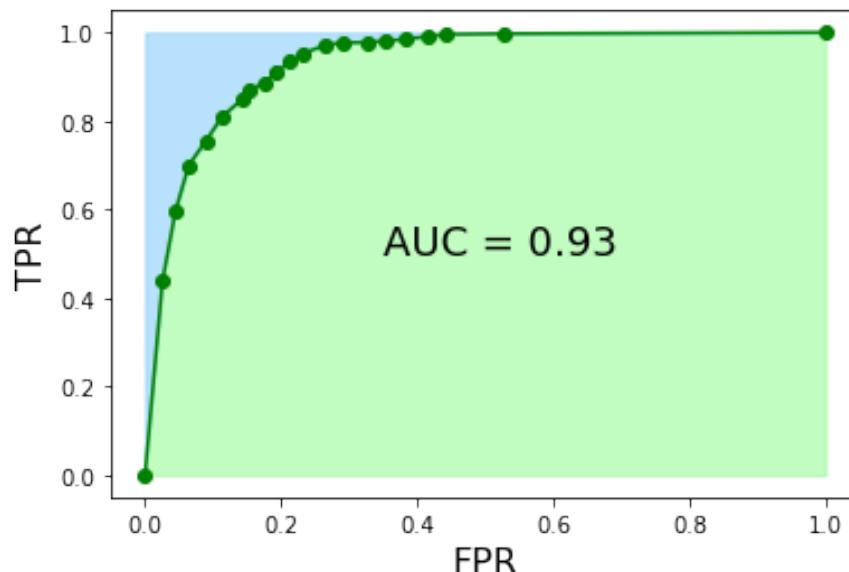
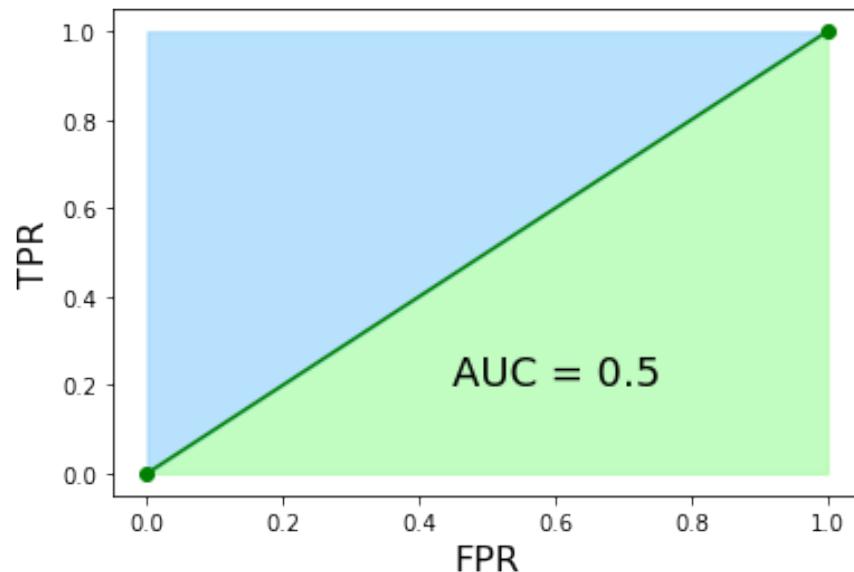


Figure 9.4: ROC curve after testing the Logistic regression model with cross-validation on the first variable of the Breast cancer dataset.

As we can see, we obtain an AUC of 0.93, which indicates that there is a 93% chance that a random point with an outcome equal to one has a higher predictive probability than a random point with an outcome of zero, which is a good model.

Next, let us see an example of the second advantage of the ROC-AUC metric. Suppose we have a data collection composed of 100 data points. Of these points, 96 have an outcome of one, and the remaining 4 have an outcome of zero. Let us suppose a model that gives all the points a predictive probability of 0.99, i.e., the model classifies all the points as a one. This would mean an accuracy of 96%, but there is obviously a class imbalance, and classifying this model as a good model would be a mistake. If we consider different thresholds, ranging from zero to one, on steps of size 0.01, we will find  $\text{FPR} = \text{TPR} = 1$  for all the thresholds except for one value corresponding to the threshold being equal to one. Since none of the predictive probabilities is greater than one, for this value, it would classify all the data points as negatives and hence yield  $\text{FPR} = \text{TPR} = 0$ . We would obtain the following ROC curve:



Since we only have two points, we obtain the following curve and an AUC of 0.5. This means that we have a horrible model. If we pick a random positive point, there is a 50% chance that its predictive probability is higher than a random negative point, making sense as all the predictive probabilities are the same.

### 9.2.4 Classification report (CR)

[21][24] Last, we have the classification report, given by a collection of parameters that determine the model's performance. These parameters are the *precision*, *recall*, *TNR* and the *F1 score*. It is worth mentioning that the TP, FP, TN, and FN values are often presented in matrix form. We define the *Confusion matrix* by:

$$CM = \begin{pmatrix} TP & FP \\ FN & TN \end{pmatrix} \quad (9.9)$$

Next, we define the quantities that are often included in a classification report:

$$Precision = \frac{TP}{TP + FP} \quad (9.10)$$

$$Recall = \frac{TP}{TP + FN} \quad (9.11)$$

$$TNR = \frac{TN}{TN + FP} \quad (9.12)$$

Moreover, the *F1 score*, which is a Harmonic mean between the precision and the recall.

$$F_1 = 2 \frac{1}{\frac{1}{precision} + \frac{1}{recall}} \quad (9.13)$$

The  $F_1$  score is contained on the interval [0,1]. The precision tracks the false positives; when the number of false positives increases, the precision drops. On the other hand, the recall tracks the False negatives, as it drops when this number increases. Whenever we do not have false negatives or positives,  $F_1$  equals one, yielding a good model. All these quantities together provide enough information to judge the performance of a model. However, often, the  $F_1$  score is used alone to determine the performance of a model. Hence, we will talk about its advantages and disadvantages.

- **Advantages:** It is a combination of the *precision* and *recall* metrics, which helps us capture much information about the performance of the model. Another advantage is that it works well with imbalanced classes.
- **Disadvantages:** The  $F_1$  score does not consider the TNR factor, which leads to the metric missing some information about the model's performance. Also, it gives the same weight to the *precision* and *recall*, which might not always be the best ratio to test a model. There might be cases where having false negatives will not have the same impact as having false positives. An example of this is the medical field, where every false negative is an illness we overlook, which is more severe than a false positive, an illness we misdiagnose.

## 9.3 Multiclass classification

A few algorithms, such as the Logistic regression or the Support Vector Machines, are only implemented for binary data collections, i.e., data with two possible outcomes. When this happens, and we have some data with more than two classes, we need some way to extend these algorithms. The main ways to address this issue are the *One vs All* and *One vs One* methods.

### 9.3.1 One vs all

Suppose we have data with three classes, and we are implementing an algorithm that can only work with two classes. We will pick each class and group the remaining classes under the same outcome or label. First, we will pick class A and assign to it the positive outcome (one) and the negative outcome (zero) to classes B and C; we will train the algorithm with this data and yield predictive probabilities labeled as  $P_{A \text{ vs } B,C}$ . Then, we will pick class B, assign it the positive outcome, and assign the negative outcome to classes A and C; we will train the algorithm with this data and yield predictive probabilities labeled as  $P_{B \text{ vs } A,C}$ . Lastly, the process for the remaining class C leads to a predictive probability labeled as  $P_{C \text{ vs } A,B}$ . Whenever we want to predict a testing point, we compute  $P_{A \text{ vs } B,C}$ ,  $P_{B \text{ vs } A,C}$  and  $P_{C \text{ vs } A,B}$  for that testing point and assign it the outcome associated to the highest value (the outcome of class A if  $P_{A \text{ vs } B,C}$  is the greatest value for example).

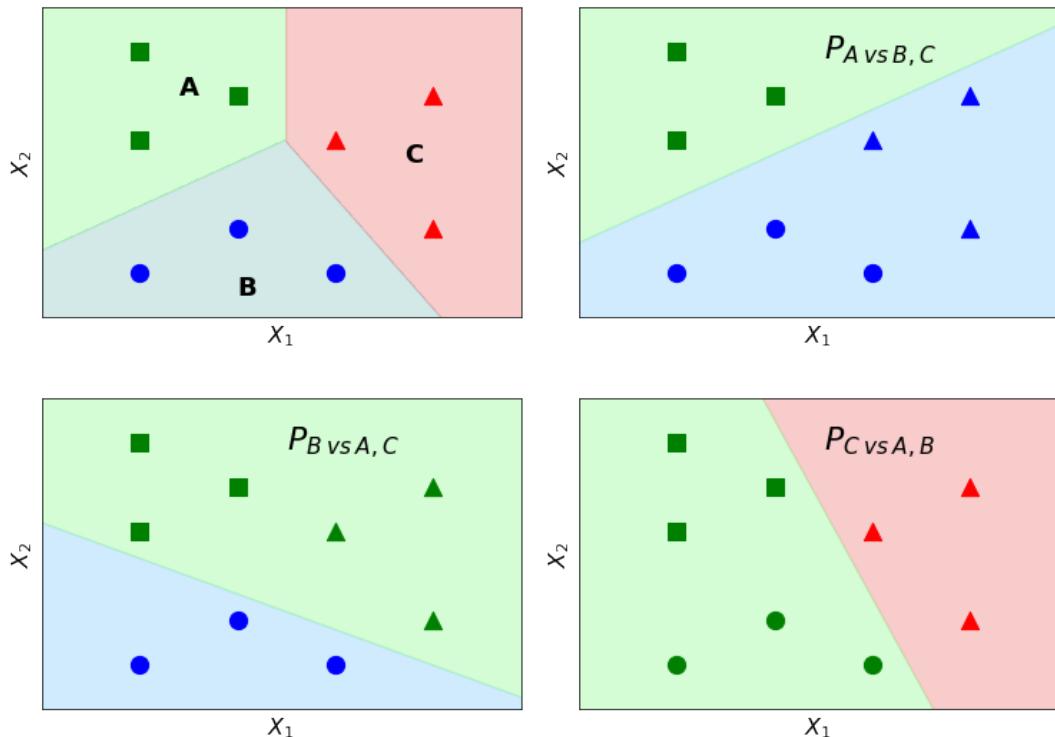


Figure 9.5: Representation of the One vs all method for multiclass classification.

The main problem with this method is that we will classify unbalanced classes, as we have one class versus all the other classes grouped. This may cause some deviations in the probabilities.

### 9.3.2 One vs One

An alternative to the One vs All method is the One vs One method. Now, we will build different classification problems, picking all the classes in pairs. In the following example, we will pick first A and B, yielding a probability labeled as  $P_{A,B}$  where it is irrelevant whether we assign A the positive or the negative outcome. If we give A the positive outcome then  $P_A = P_{A,B}$  and  $P_B = 1 - P_{A,B}$ . If we give A the negative outcome,  $P_A = 1 - P_{A,B}$  and  $P_B = P_{A,B}$ . We do the same process for the remaining pairs, and then we have two ways to proceed when predicting a point:

- We can make a prediction for a point on each of the three configurations and then pick the most predicted class for this testing point.
- We can also add each configuration's contribution to the probability of each class. Let us suppose, for example, that when picking A vs B, we assign A the positive outcome, and when picking A vs C, we assign A the negative outcome. Then the total probability of class A would be given by  $P_A = P_{A,B} + (1 - P_{A,C})$ . We do the same for the different classes and classify a point based on the highest value.

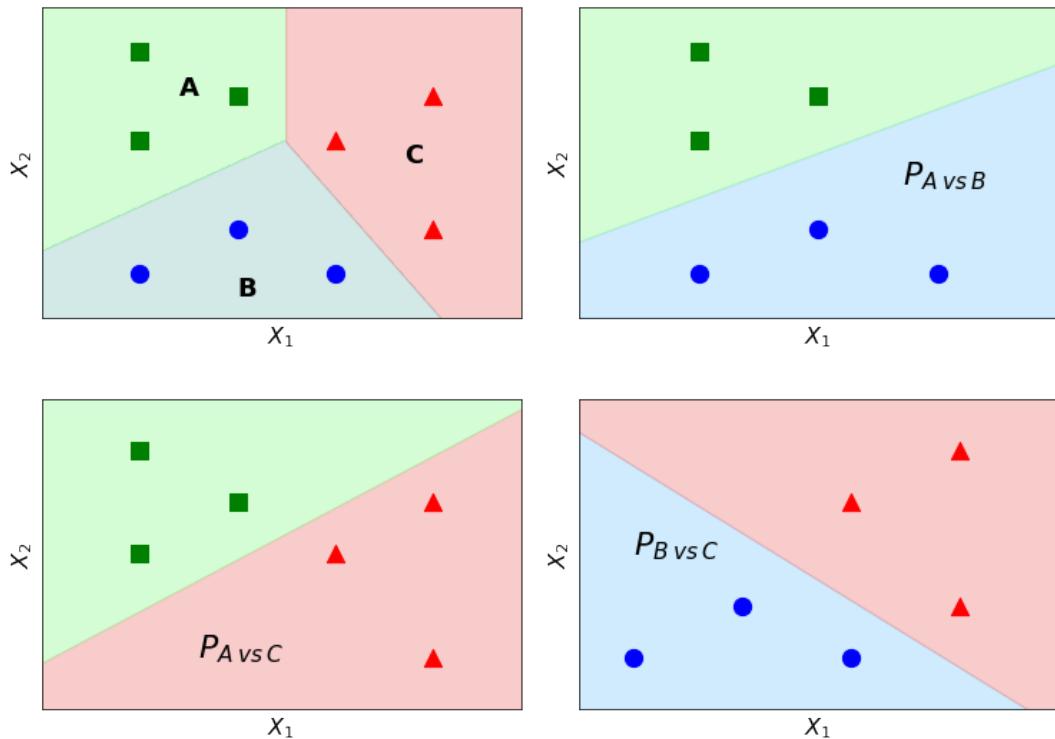


Figure 9.6: Representation of the One vs One method for multiclass classification.

This method is often used before one vs all since there is no class imbalance. The only disadvantage of this method is that it is computationally more expensive as there are more class pair combinations than class combinations on the One vs All method (this statement is not true for data with 3 labels).

# Bibliography

- [1] Addagatla, A. (2022, January 6). Maximum likelihood estimation in logistic regression. Medium. <https://arunaddagatla.medium.com/maximum-likelihood-estimation-in-logistic-regression-f86ff1627b67>
- [2] AMES Housing Dataset. (2018, September 10). Kaggle. <https://www.kaggle.com/datasets/prevek18/ames-housing-dataset>
- [3] Andrew Zisserman. (2015). Lecture 3: SVM dual, kernels and regression. <https://www.robots.ox.ac.uk/~az/lectures/ml/lect3.pdf>
- [4] Breast Cancer Dataset (n.d.). Scikit-learn. [https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load\\_breast\\_cancer.html](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html)
- [5] Brownlee, J. (2021, February 7). Weight initialization for deep learning neural networks. MachineLearningMastery.com. <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/>
- [6] Chiramana, S. (2021, December 12). SVM DUAL FORMULATION - sathvik chiramana - Medium. Medium. <https://medium.com/@sathvikchiramana/svm-dual-formulation-7535caa84f17>
- [7] Classification: ROC Curve and AUC. (n.d.). Google for Developers. <https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc?hl=en>
- [8] Coding Lane. (2021a, November 1). Backpropagation in CNN - Part 1 [Video]. YouTube. <https://www.youtube.com/watch?v=Pn7RK7tofPg>
- [9] Coding Lane. (2021b, November 6). Backpropagation in CNN - PART 2 [Video]. YouTube. <https://www.youtube.com/watch?v=vUbUozbkMhI0>
- [10] Correlation and regression | The BMJ. (2020, October 28). The BMJ | the BMJ: Leading General Medical Journal. Research. Education. Comment. <https://www.bmjjournals.org/about-bmj/resources-readers/publications/statistics-square-one/11-correlation-and-regression>

- [11] Dash, S. (2023, November 3). Decision Trees explained — entropy, information gain, Gini index, CCP pruning. Medium. <https://towardsdatascience.com/decision-trees-explained-entropy-information-gain-gini-index-ccp-pruning-4d78070db36c>
- [12] David Page. (n.d.). SVM by Sequential Minimal Optimization (SMO). <https://pages.cs.wisc.edu/~dpage/cs760/SMOlecture.pdf>
- [13] David Sontag. (n.d.). Support Vector Machines & Kernels Lecture 5. <https://people.csail.mit.edu/dsontag/courses/ml13/slides/lecture5.pdf>
- [14] DeepLearningAI. (2017). Convolutional Neural Networks (Course 4 of the Deep Learning Specialization) [Video]. YouTube. <https://www.youtube.com/playlist?list=PLkDae6sCZn6Gl29AoE31iwdVwSG-KnDzF>
- [15] Deeplizard. (2017, December 9). Convolutional Neural Networks (CNNs) explained [Video]. YouTube. [https://www.youtube.com/watch?v=YRhx dVk\\_sIs](https://www.youtube.com/watch?v=YRhx dVk_sIs)
- [16] Deisenroth, M. P., Faisal, A. A., & Ong, C. S. (2020). Mathematics for machine learning. <https://doi.org/10.1017/978108679930>
- [17] Fernando, J. (2023, December 13). R-Squared: Definition, Calculation Formula, uses, and Limitations. Investopedia. <https://www.investopedia.com/terms/r/rsquared.asp>
- [18] GeeksforGeeks. (2023, January 7). Learning Vector Quantization. <https://www.geeksforgeeks.org/learning-vector-quantization/>
- [19] Giba, B. (2021). Lasso regression explained, step by step. Machine Learning Compass. <https://machinelearningcompass.com/machine-learning-models/lasso-regression/>
- [20] Housing Dataset. (2019, March 7). Kaggle. <https://www.kaggle.com/datasets/ashydu/housing-dataset>
- [21] Kanstrén, T. (2023, September 27). A look at precision, recall, and F1-Score - towards data science. Medium. <https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec>
- [22] Kim, J. H. (2019). Multicollinearity and misleading statistical results. Korean Journal of Anesthesiology, 72(6), 558–569. <https://doi.org/10.4097/kja.19087>
- [23] Kurama, V. (2023, October 25). Gradient boosting for classification | Paperspace blog. Paperspace Blog. <https://blog.paper-space.com/gradient-boosting-for-classification/>
- [24] Machine Learning. (2023, September 27). What are the advantages and disadvantages of using F1 score for ANN performance evaluation? www.linkedin.com. <https://www.linkedin.com/advice/3/what-advantages-disadvantages-using-f1-score-ann/?lang=en>

- [25] Mahesh Huddar. (2020, June 4). 1. Association Rule Mining – Apriori Algorithm - Numerical example solved by Mahesh Huddar [Video]. YouTube. <https://www.youtube.com/watch?v=43CMKRHdH30>
- [26] Mazen Ahmed. (2022, August 2). Gradient boosted trees for classification explained [Video]. YouTube. [https://www.youtube.com/watch?v=hjxgouJ\\_va8](https://www.youtube.com/watch?v=hjxgouJ_va8)
- [27] Multivariate Adaptive Regression SPLines · UC Business Analytics R Programming Guide. (n.d.). <https://uc-r.github.io/mars>
- [28] Neural Networks. (2023, April 18). What are the advantages and disadvantages of using cross-entropy loss for classification tasks? www.linkedin.com. <https://www.linkedin.com/advice/0/what-advantages-disadvantages-using-cross-entropy>
- [29] Normalized Nerd. (2021, April 21). Random Forest algorithm clearly explained! [Video]. YouTube. <https://www.youtube.com/watch?v=v6VJ2R066Ag>
- [30] Popkes, A. (n.d.). Bayesian linear regression. [https://alpopkes.com/posts/machine\\_learning/bayesian\\_linear\\_regression/](https://alpopkes.com/posts/machine_learning/bayesian_linear_regression/)
- [31] R. Berwick. (n.d.). An Idiot's guide to Support vector machines (SVMs). <https://web.mit.edu/6.034/wwwbob/svm.pdf>
- [32] Roberto Odorico. (1997). Learning Vector Quantization with Training Count (LVQTC). <https://www.sciencedirect.com/science/article/pii/S089360809700129>
- [33] Rosebrock, A. (2022, December 29). OpenCV Eigenfaces for face recognition - PyImageSearch. PyImageSearch. <https://pyimagesearch.com/2021/05/10/opencv-eigenfaces-for-face-recognition/>
- [34] Saeed, M. (2022). Method of Lagrange Multipliers: The Theory behind support vector Machines (Part 1: The Separable case). MachineLearningMastery.com. <https://machinelearningmastery.com/method-of-lagrange-multipliers-theory-behind-support-vector-machines-part-1-the-separable-case/>
- [35] Seif, G. (2022, February 11). Understanding the 3 most common loss functions for Machine Learning Regression. Medium. <https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3>
- [36] Sicotte, X. B. (2018a, May 24). Locally Weighted Linear Regression (Loess). <https://xavierbourretsicotte.github.io/loess.html>
- [37] Sicotte, X. B. (2018b, June 13). Lasso regression: derivation of the coordinate descent update rule — Data Blog. [https://xavierbourretsicotte.github.io/lasso\\_derivation.html](https://xavierbourretsicotte.github.io/lasso_derivation.html)

- [38] Singh, S. (2023b, June 28). Understanding the Bias-Variance tradeoff - towards data science. Medium. <https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229>
- [39] Sklearn. (n.d.). Linear-model : Lasso. Scikit-learn. [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.Lasso.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html)
- [40] Stanford University. (Autumn 2009). The Simplified SMO Algorithm. <https://cs229.stanford.edu/materials smo.pdf>
- [41] StatQuest with Josh Starmer. (2019, November 4). Support vector Machines Part 3: The radial (RBF) kernel (Part 3 of 3) [Video]. YouTube. [https://www.youtube.com/watch?v=Qc5IyLW\\_hns](https://www.youtube.com/watch?v=Qc5IyLW_hns)
- [42] StatQuest with Josh Starmer. (2019a, April 22). Gradient Boost Part 4 (of 4): Classification details [Video]. YouTube. <https://www.youtube.com/watch?v=StWY5QWMXCw>
- [43] Team, C. (2023). Ridge. Corporate Finance Institute. <https://corporatefinanceinstitute.com/resources/data-science/ridge/>
- [44] The Iris Dataset. (n.d.). Scikit-learn. [https://scikit-learn.org/stable/auto\\_examples/datasets/plot\\_iris\\_dataset.html](https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html)
- [45] U.S. Bureau of labor statistics. (n.d.). The Employment Situation-December 2022. [https://www.bls.gov/news.release/archives/empsit\\_01062023.pdf](https://www.bls.gov/news.release/archives/empsit_01062023.pdf)
- [46] UW-Madison. (n.d.). SVM by Sequential Minimal Optimization (SMO). <https://pages.cs.wisc.edu/~dpage/cs760/SMOlecture.pdf>
- [47] Vipulgandhi. (2020, January 6). How to choose right metric for evaluating ML Model. Kaggle. <https://www.kaggle.com/code/vipulgandhi/how-to-choose-right-metric-for-evaluating-ml-model>
- [48] Visually Explained. (2021, September 29). Principal Component Analysis (PCA) [Video]. YouTube. <https://www.youtube.com/watch?v=FD4DeN810DY>
- [49] Wikipedia contributors. (2023a). Multicollinearity. Wikipedia. <https://en.wikipedia.org/wiki/Multicollinearity>
- [50] Wikipedia contributors. (2023b). Multivariate adaptive regression spline. Wikipedia. [https://en.wikipedia.org/wiki/Multivariate\\_adaptive\\_regression\\_spline](https://en.wikipedia.org/wiki/Multivariate_adaptive_regression_spline)
- [51] Wikipedia contributors. (2023c). Ordinary least squares. Wikipedia. [https://en.wikipedia.org/wiki/Ordinary\\_least\\_squares](https://en.wikipedia.org/wiki/Ordinary_least_squares)
- [52] Wikipedia contributors. (2023d). Ridge regression. Wikipedia. [https://en.wikipedia.org/wiki/Ridge\\_regression](https://en.wikipedia.org/wiki/Ridge_regression)

- [53] Yadav, A. (2018, October 22). SUPPORT VECTOR MACHINES(SVM) - towards data science. Medium. <https://towardsdatascience.com/support-vector-machines-svm-c9ef22815589>
- [54] Yuh-Jye Lee. (2018, May 2). Sequential Minimal Optimization (SMO). [https://milaab.github.io/Yuh-Jye-Lee/assets/file/teaching/2017\\_machine\\_learning/SMO\\_algorithm.pdf](https://milaab.github.io/Yuh-Jye-Lee/assets/file/teaching/2017_machine_learning/SMO_algorithm.pdf)
- [55] 3Blue1Brown. (2017a, October 5). But what is a neural network? | Chapter 1, Deep learning [Video]. YouTube. <https://www.youtube.com/watch?v=aircAruvnKk>
- [56] 3Blue1Brown. (2017b, October 16). Gradient descent, how neural networks learn | Chapter 2, Deep learning [Video]. YouTube. <https://www.youtube.com/watch?v=IHZwWFHWa-w>
- [57] 3Blue1Brown. (2017c, November 3). What is backpropagation really doing? | Chapter 3, Deep learning [Video]. YouTube. <https://www.youtube.com/watch?v=Ilg3gGewQ5U>
- [58] 3Blue1Brown. (2017d, November 3). Backpropagation calculus | Chapter 4, Deep learning [Video]. YouTube. <https://www.youtube.com/watch?v=tIeHlnjs5U8>