Machine Learning

Coursework 2

CID 01945214, Imperial College London Spring 2020

The questions 1 and 3 will be addressed in Python and the question 2 will be addressed in R. The two main Python libraries used are Scikit_learn and Tensorflow. All R and Python functions used will be quoted with an hyperlink containing the corresponding documentation.

1 Question 1

1.1 Data summary

We aim to achieve a classification task on chromosomes for which information have been collected by image processing. We have a data set with 28 features and 1093 rows, each of them is a chromosome observation from an individual. 11 features $V_2, .., V_{12}$ correspond to the size and shape of the chromosome, and the 17 others $V_{13}, .., V_{29}$ correspond to the banding pattern induced by staining.

We start by shuffling the data set and we separate it in 2 equally sized parts : the training set and the test set. This is done with the function sklearn.model_selection.train_test_split. The test set will only be used once, for the final out-of-sample performance comparison.

We need to handle the missing values presented in our data set. 51 out of 546 rows of our training data set have at least one missing value. This corresponds to around 10% of the total number of observations. We notice that each feature has between 0 and 4 missing values. The rows with missing values have actually a few number of features with missing value. To avoid the lost of this data, we choose to handle the missing values with median imputation. It means that we replace every missing value by the median calculated on the associated feature column. We replace the missing values in the test set using the same medians calculated on the training set.

Now that we have data without missing values, we can start having a look at it. The labels are well balanced : we have 263 observations with label $z = 0$ and 283 observations with label $z = 1$. The features have globally the same magnitude, with values between -500 and 500. They are all roughly centered in zero but the standard deviation varies a lot from one feature to another.

A Spearman correlation matrix can be used to investigate the dependence between features. Recall that the Spearman correlation assesses how well the relationship between two variables can be described using a monotonic function [1]. A pattern strikes us immediately : each feature with index higher than 20 is completely uncorrelated with the other features and with the label z (See Appendix A). Moreover, these features with index higher than 20 have very similar density for label $z = 0$ and $z = 1$ (figure 1), and p-value much larger than 0.05 for Mann–Whitney U test (See Appendix B). Therefore, they may not be very important in the decisions of our classification algorithms.

For most features with index lower than 20, which are the size features and some of pattern features, there is a noticeable distinction between their density depending on the label $z = 0$ or $z = 1$. Morevoer, most of these features have a statistically significant p-value for Mann–Whitney U test. For these reasons, they are likely to play a key role in the decisions of our classification algorithms. V_2 and V_3 have the lowest p-value for Mann–Whitney U test and have the highest absolute Spearman correlation with z. Consequently, they should be particularly important in the decision process. As these two features are very highly correlated ($\rho_{Spearman}(V_2, V_3) = 0.98$), Machine Learning models may pick up more information from one of these feature than from the other.

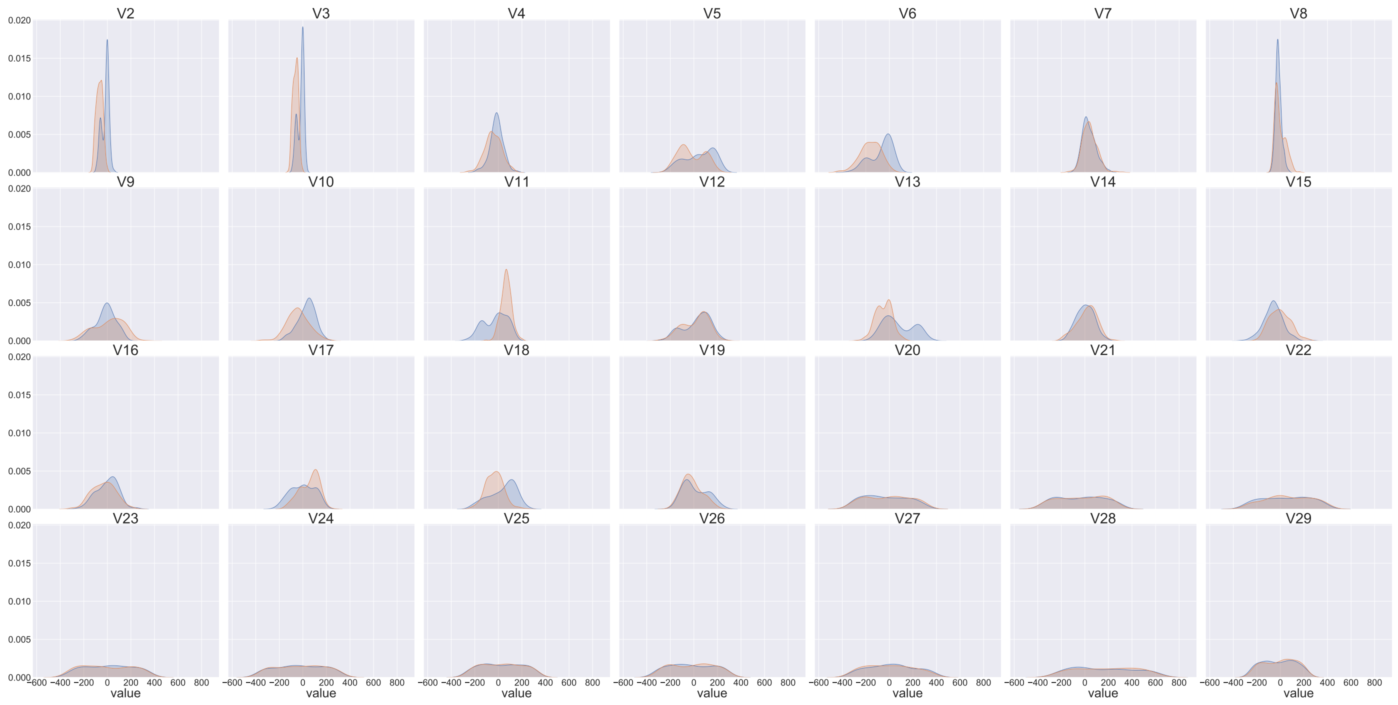


Figure 1: Density plot of the features for label $z = 0$ (blue) and $z = 1$ (orange)

1.2 Machine Learning models

1.2.1 Random Forests (RF)

Random Forests are an ensemble learning method that consists in constructing many decision tree classifiers on various sub-samples of the dataset. Predictions are made averaging the predictions made by each individual tree. We fit Random Forests to our data set with the function sklearn.ensemble.RandomForestClassifier.

To avoid overfitting, we put constraints on the decision trees constructed by the randoms forests method :

- When splitting a node, only a subset of features is considered.
- The depth of the tree is upper-bounded
- When splitting an internal node, a minimum number of samples is required in each leaf
- The number of leaf nodes is upper-bounded
- A minimum number of samples is required in a leaf node

Other hyper parameters that we must fix are the number of trees in the forest and the criterion used to split a node.

We use a randomized grid search to tune the hyperparameters of the model. The parameters of the Random Forests are optimized by cross-validated search over parameter settings. We create a grid with large range of possible values for hyperparameters. For 300 iterations, a combination of hyperparameters is sampled and the performances of the model with these hyperparameters are evaluated by 5-fold cross-validation. We then select the combination that obtained the lowest cross-validation error and report it in table 1. The final Random Forests classifier obtains an accuracy of 0.9872 on training set and 0.9141 on test set.

Hyperparameters	Value
n_estimators - Number of trees in the forest	2100
max_features - Number of features to consider when looking for the best split	25
max_depth - Maximum depth of the tree	133
min_samples_split - Minimum number of samples required to split an internal node	2
max_leaf_nodes - Maximum number of leaf nodes	15
min_samples_leaf - Minimum number of samples in a leaf node	1
criterion - Measure of the quality of a split	*entropy*

Table 1: Selected hyperparameters for the Random Forests classifier

Random Forests give a measure of the relative importance of each feature. We can see that our intuition that features higher than 20 are not important for Random Forests predictions is confirmed. All together, they represent less than

5% of Random Forests importance. In the contrary, size features seem to be particularly important for the decision process, and especially the feature V_3 which alone represents more than 40% of the total importance. It is also noticed that Random Forests identify V_{12} as an important feature which we were not able to do looking at the density pattern and Mann–Whitney U test p-values.

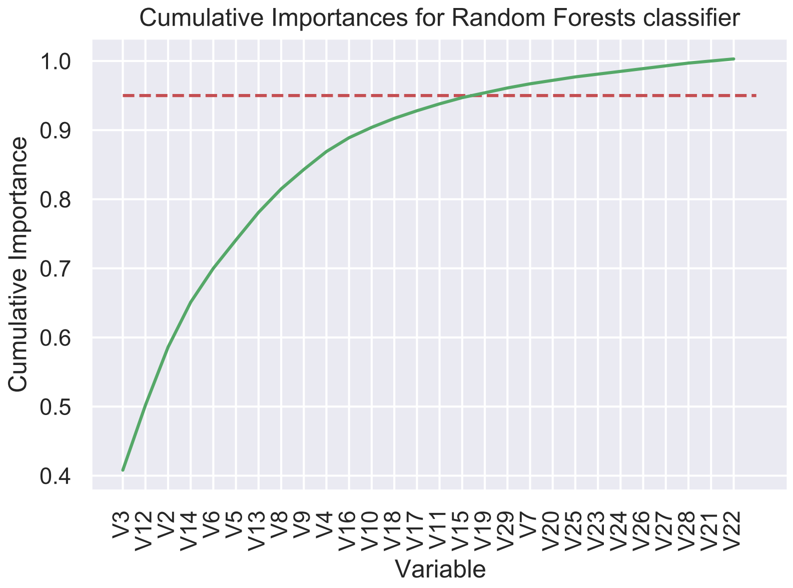


Figure 2: Features importance computed by Random Forests

1.2.2 Multilayer perceptrons (MLP)

We aim to fit a Multi-Layer Perceptron to our data set. We start by dropping 50 observations from the training set to create a validation set. Then, we scale the training set to make the training faster and reduce the chances of getting stuck in local optima :

$$X_{i,j} \leftarrow \frac{X_{i,j} - \bar{X}_j}{\sigma_j} \tag{1.1}$$

Where $\bar{X}_j$ and σ_j are respectively the mean and standard deviation of the column j. The same transformation is applied to the validation and test sets with the means and standard deviations computed on the training set.

To find the architecture of our MLP and tune the multiple hyperparameters, various (well selected) architectures are tried and the behaviour of the model is analyzed. The shape of the training and validation loss function, and the training/validation accuracy indicate if our model has a large bias and/or a large variance. This process gives us a final model that achieves good performances on both the training set and validation set.

As we aim to predict probabilities to belong to either the class $z = 1$ or the class $z = 0$, it is relevant to use a sigmoid output layer which has a range in $(0, 1)$. We use one hidden dense layers with 50 neurons, ie, with all neurons connected to every unit of the input. We choose a ReLU activation function for the hidden layer (see figure 3). The loss used for model training is the negative binary crossentropy defined by :

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{N}\left[z_i \log(p_i(\boldsymbol{\theta})) + (1 - z_i)\log(1 - p_i(\boldsymbol{\theta}))\right] \tag{1.2}$$

where $\boldsymbol{\theta}$ contains all the weights and biases of the neural network and $p_i(\boldsymbol{\theta}) = \mathbb{P}(\hat{z}_i = 1)$ is the MLP predicted probability that the i^{th} observation belongs to class $z = 1$

We use the following methods to reduce overfitting :

- callback "early stopping" : we stop the training of Neural Networks when the performance do not improve on the validation set after 10 successive epochs (figure 4).
- l2 penalty : we add a sum of squares penalty term to the loss term to discourage the weights from growing too large : $\mathcal{L}'(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \alpha_{l2}||\boldsymbol{\theta}||_2^2$ where $\alpha_{l2} > 0$ is an hyperparameter.
- Dropout : during the network training, each neuron has a probability *Dropout_rate* to have its weights set equal to 0 for one epoch. Any neuron in the network is no longer able to depend on any other specific neurons being present, and so each neuron learns features that are more robust, and generalises better.

A minimum of the lost function $\mathcal{L}$ is seeked using a stochastic gradient descent. Although the stochastic gradient descent is known to be slow to converge, our data set is small and this drawback is no longer an important issue. Several learning rates between 10^{-5} and 10^{-1} were tried, and their learning curves for 10 epochs each were compared using Tensorboard. The best validation accuracy was obtained for a learning rate equal to 5×10^{-3}.

We used the Sequential API tf.keras.Sequential to fit the MLP to our data. A summary of the model architecture with its associated hyperparameters is presented in table 2.

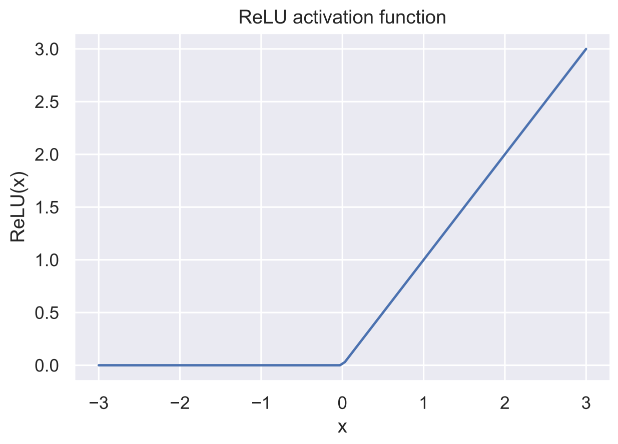


Figure 3: ReLu activation function

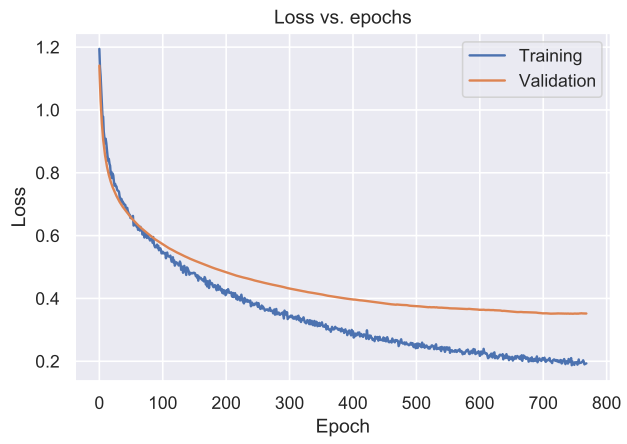


Figure 4: Training and validation loss during the MLP training. The callbacks "early stoppping" stops the training when the validation loss does not decrease (while the training loss is still decreasing)

output layer activation function	Sigmoid
number of hidden layers	1
number of neurons in dense layers	50
hidden layer activation function	ReLU
Optimizer	Stochastic Gradient Descent (SGD)
Rate SGD	5×10^{-3}
Dropout rate	0.3
l2 regularizer rate α_{l2}	10^{-2}

Table 2: Hyperparameters tuning of Random Forests classifier

The accuracy of the predictions made by our two classifiers are summarized in table 6. We can see that Random Forests classifier performs slightly better than MLP on both the training set and test set. A deeper comparison between the two models will be made in next section.

	Training set	Test set
Random Forests	0.9872	0.9141
MLP	0.9839	0.9122

Table 3: Accuracy on training set and test set for Random Forests and MLP

1.3 Model comparison - McNemar's test

Let n_{MLP} be the number of errors made by the MLP classifier and not by Random Forests, and n_{RF} the number of errors made by the Random Forests and not by the MLP. Then McNemar's test [2] with continuity correction refers

$$M = \frac{|n_{RF} - n_{MLP}| - 1}{\sqrt{n_{RF} + n_{MLP}}}$$

to a $\mathcal{N}(0, 1)$ distribution. The null hypothesis is that the predictions of Random Forests and MLP are the same. McNemar's test with continuity correction rejects for high value of the statistic M.

The exact test refers n_{MLP} to a binomial $(n_{MLP} + n_{RF}, 1/2)$ distribution with a two-sided critical region.

To construct the test, we first construct the contingency table of our two classifiers. We use the function statsmodels.stats.contingency_tables.mcnemar to compute both the p-value of the McNemar's test with continuity correction and the exact test. As we can see in contingency table (table 4), both models make errors in much the same proportion. About a third of the errors are made on different instances of the test set. For both corrected and exact test, we have a high p-value and we fail to reject the null hypothesis at a 5% level. The conclusion drawn by the McNemar's test is that the predictions of Random Forests and MLP are the same.

	RF correct	RF incorrect
MLP correct	485	15
MLP incorrect	14	33

	p-value	statistics
McNemar test with correction	0.8551	0.03
Exact McNemar test	0.8560	14.00

Table 4: Contingency table (Right) and McNemar test (Left) for Random Forests and MLP

1.4 Model Selection

The conclusion of McNemar test is that the predictions of Random Forests and MLP are the same. Therefore, we should not have any preference between Random Forests classifier and MLP. Nevertheless, both the exact and corrected McNemar tests are known to be overly conservative [3]. Moreover, McNemar test statistic depends only on the different correct or incorrect predictions between the two models, but it does not give directly on the classification error rates. For these reasons, a deeper study of the models performances is needed to select our final classifier.

As we saw before, the Random Forests classifier obtains higher accuracy than MLP on both training and test set. We should also be interested to the capability of models to distinguish between classes $z = 0$ and $z = 1$. A good tool to measure this is the ROC curve and the associated Area Under the Curve (AUC). The ROC curve is a plot of the True positive rate (TPR) against the False Positive Rate (FPR) for various discrimination thresholds. TPR and FPR are defined by :

$$TPR = \frac{True\,Positive}{True\,Positive + False\,Negative} \qquad FPR = \frac{False\,Positive}{True\,Negative + False\,Positive} \tag{1.3}$$

Where we assume that the positive class is $z = 1$ and the negative class is $z = 0$. The Higher the AUC is, the better the model is at predicting $\hat{z} = 0$ an observation of class $z = 0$ and $\hat{z} = 1$ an observation of class $z = 1$. Both the classifiers obtain an AUC close to 1 which indicates a good capability to distinguish between the classes $z = 0$ and $z = 1$. The AUC is slightly higher for Random Forests than for MLP.

Finally, we could argue that Random Forests are able to give a measure of the relative importance of each feature, what a MLP cannot. In this sense, Random Forests are more interpretable than MLP.

The final model that we select is the Random Forests classifier with hyperparameters defined by table 1. This model obtains better performances in terms of accuracy and AUC than MLP, and it offers a better interpretability too.

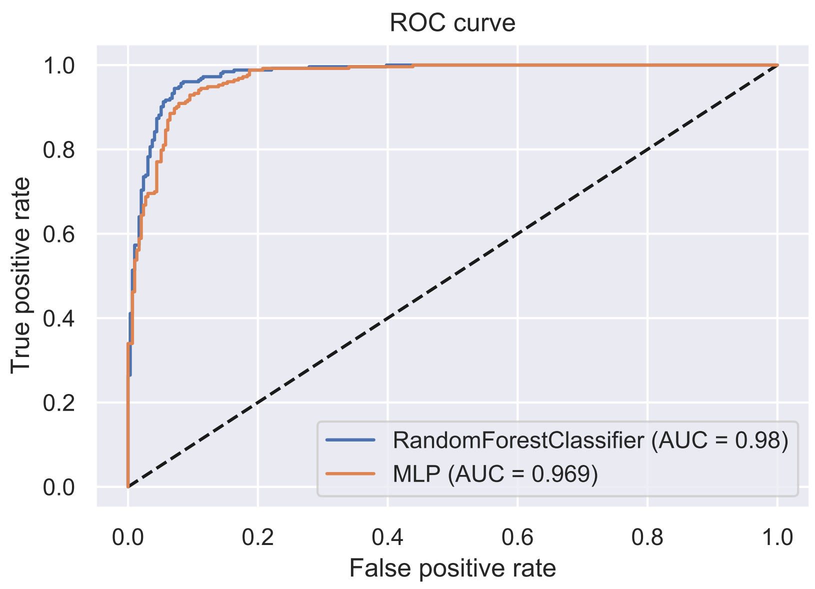


Figure 5: Empirical and true cumulative distribution functions

Note that we could have issues concerning our final model recommendation in regards to our relative preferences between $True\,Positive$ (TP) and $False\,Negative$ (FN). Without any additional information on the nature of class z, we considered that we do not have any preference between TP and FN. However, we could imagine situations where we absolutely want to avoid classification errors of $z = 0$ observations. For example, the genes of class $z = 0$

may have key information on patients that other genes do not have and then should be detected in priority. In this case, we should not compare our classifiers only on their accuracy, AUC or the outcome of McNemar test. In particular, we should pay attention to the confusion matrix obtained for each classifier. These latters can be computed with the function sklearn.metrics.confusion_matrix. The MLP is observed to make less classification errors for $z = 0$ observations (33 FP) than Random Forests (42 FP). Therefore, in the case where we want to avoid classification errors of $z = 0$ observations, it may be more relevant to select the MLP than Random Forests.

RF Prediction	$\hat{z}^{(RF)} = 0$	$\hat{z}^{(RF)} = 1$
True class $z = 0$	252	42
True class $z = 1$	5	248

MLP Prediction	$\hat{z}^{(MLP)} = 0$	$\hat{z}^{(MLP)} = 1$
True class $z = 0$	261	33
True class $z = 1$	15	238

Table 5: Confusion matrix for Random Forests and MLP

1.5 Model performance with reject option

We aim to implement a reject option for our final Random Forests classifier [4]. We partition the sample space into two complementary regions $\mathcal{R}$, a reject region, and $\mathcal{A}$, an acceptance region. For a given threshold t, these regions are defined by :

$$\mathcal{R} = \left\{ \mathbf{x} | 1 - \max_{i=0,1} p(\hat{z} = i|\mathbf{x}) > t \right\}$$
$$\mathcal{A} = \left\{ \mathbf{x} | 1 - \max_{i=0,1} p(\hat{z} = i|\mathbf{x}) \leq t \right\}$$

Using a threshold $t = 0.4$. The reject and acceptance regions can be written equivalently :

$$\mathcal{R} = \left\{ \mathbf{x} | \max_{i=0,1} p(\hat{z} = i|\mathbf{x}) < 0.6 \right\} = \{\mathbf{x} | 0.4 < p(\hat{z} = 1|\mathbf{x}) < 0.6\}$$
$$\mathcal{A} = \left\{ \mathbf{x} | \max_{i=0,1} p(\hat{z} = i|\mathbf{x}) \geq 0.6 \right\} = \{\mathbf{x} | p(\hat{z} = 1|\mathbf{x}) < 0.4 \text{ or } p(\hat{z} = 1|\mathbf{x}) > 0.6\}$$

We identify which test set observations fall in the acceptance region of our Random Forests classifier, and evaluate performance only on this subset. Among the 547 observations in the test set, 45 have a predicted probability to belong to class $z = 1$ between 0.4 and 0.6. These observations are identified as "rejected" and not considered for classification. The confusion matrix is given on the 502 remaining observations in the acceptance region :

RF Prediction	$\hat{z}^{(RF)} = 0$	$\hat{z}^{(RF)} = 1$
True class $z = 0$	234	24
True class $z = 1$	3	242

Table 6: Accuracy on training set and test set for Random Forests and MLP

Comparing this confusion matrix with the one implemented without reject option (table 5), we can notice that almost half False Negative observations have been rejected. The Random Forests classifier is uncertain of its prediction for several observations especially of class $z = 0$ in the sense that the predictive class probabilities are close to 0.5. For these observations Random Forests predictions are almost as bad as those of a classifier that picks up randomly a class. Deleting these observations leads to a significant improvement of Random Forests performances resulting in an increasing of the accuracy to 0.9463.

2 Question 2

2.1 Hierarchical Clustering

Hierarchical clustering is used to group observations in clusters based on their similarity. The similarity is derived from a linkage criteria which defines a distance between two sets of features A and B. The linkage criteria is based on a distance d defined on the features space $\mathbb{R}^{N_{obs}}$. One could choose to use an usual distance on this space like Euclidean distance or Manhattan distance. However, it may be convenient to define a distance which is more interpretable when comparing features. We make the choice to consider two features to be similar if they are correlated (in absolute value). A relevant distance would then be close to 0 for highly correlated features (in absolute value) and would be stricly positive for uncorrelated features. We define the following distance d that has the desired property :

$$d(V_i, V_j) = 1 - |\rho_{Spearman}(V_i, V_j)| \tag{2.1}$$

Where $\rho_{Spearman}$ is the Spearman correlation [1]. We will use the following linkage criteria based on the distance d:

- the maximum or complete-linkage clustering, which is the maximum value of all pairwise distances between the elements in cluster A and the elements in cluster B :

$$\max\{d(a,b),(a,b)\in A\times B\} \tag{2.2}$$

- the minimum or single-linkage clustering, which is the minimum value of all pairwise distances between the elements in cluster A and the elements in cluster B :

$$\min\{d(a,b),a,b\in A\times B\} \tag{2.3}$$

- the average linkage clustering, which is the average distance between the elements in cluster A and the elements in cluster B :

$$\frac{1}{|A||B|}\sum_{a\in A}\sum_{b\in B}d(a,b) \tag{2.4}$$

The clusters are built with an agglomerative approach. Each observation is first assigned to its own cluster. Then, the pair of clusters which obtains the lowest measure of dissimilarity is merged into one cluster. This process is made recursively until all the observation are grouped in one cluster. We plot the hierarchical clustering obtained with the three linkage criteria stated above in figure 6. This is done using the R function *hclust* from *stat* package.

The dendrograms obtained with single and average linkage are quite similar : they have one principal cluster composed principally of the size features, and multiple cluster composed of only one or two pattern feature(s). The dendrogram obtained with complete linkage presents more compact clusters. We also display the dendrograms obtained using the usual euclidean distance (after features normalization) instead of the distance based on Spearman correlation defined above (equation 2.1). The obtained clusters are observed to be quite different. However, we find globally the same patterns : one large cluster and multiple small clusters with single and average linkage, and more compact clusters with complete linkage.

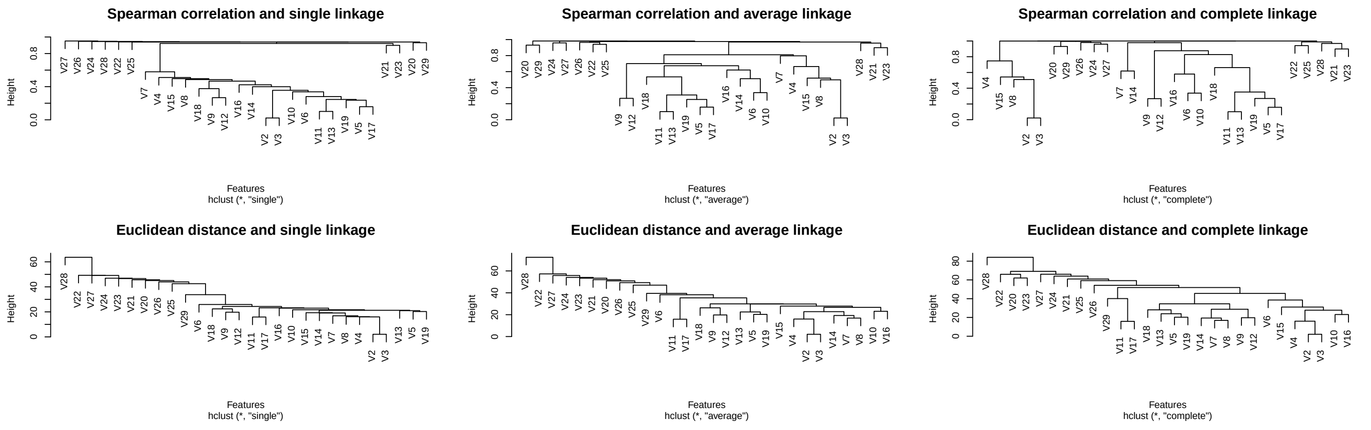


Figure 6: Hierarchical clustering with different linkage criteria

In the following, we use hierarchical clustering with the distance d defined in (2.1). The number of clusters K is an hyper-parameter. It can be chosen by maximizing the silhouette measure of the clustering.

$$s_{sil}(x)=\begin{cases}\frac{b(x)-a(x)}{\max(a(x),b(x))} & \text{if } |\mathcal{C}_k|>1\\ 0 & \text{if } |\mathcal{C}_k|=1 \qquad (x \text{ is alone in its own cluster })\end{cases} \tag{2.5}$$

Where $a(x)$ is the mean distance between x and all other data points in the same cluster, and $b(x)$ is the smallest mean distance of x to all points in any other cluster than $\mathcal{C}_k$:

$$a(x)=\frac{1}{|\mathcal{C}_k|-1}\sum_{y\in\mathcal{C}_k,y\neq x}d(x,y) \qquad b(x)=\min_{i\neq k}\frac{1}{|\mathcal{C}_i|}\sum_{y\in\mathcal{C}_i}d(x,y) \tag{2.6}$$

Let us explain why a high silhouette measure is an indicator of the relevancy of the assignment of an observation x to its cluster $\mathcal{C}_k$:

- When $s_{sil}(x)\simeq 1$, we have $a(x)<<b(x)$. Then, x is in average much closer to the points in its assigned cluster $\mathcal{C}_k$ than to the points in a different cluster. Therefore, we can say that in this case x is 'well-clustered'.

- When $s_{sil}(x)\simeq 0$, x, we have $a(x)\simeq b(x)$. Then, x is in average as close to the points in its assigned cluster than to the point of the cluster $\mathcal{C}_j$ with $j=\arg\min_{i\neq k}\frac{1}{|\mathcal{C}_i|}\sum_{y\in\mathcal{C}_i}d(x,y)$. In this case, there is not a big difference assigning x to either $\mathcal{C}_k$ or $\mathcal{C}_j$ as it lies approximately equally far away from both.

- When $s_{sil}(x) \simeq -1$, we have $a(x) >> b(x)$. Then, x is in average much closer to the points in the cluster $\mathcal{C}_j$ with $j = \arg\min_{i \neq k} \frac{1}{|\mathcal{C}_i|} \sum_{y \in \mathcal{C}_i} d(x, y)$ than to the points in its assigned cluster $\mathcal{C}_k$. Therefore x should have been assigned to $\mathcal{C}_j$ or another cluster it is close to.

The silhouette measure of our clustering is defined as the average silhouette index over all points in the data set. It constitutes an indicator of the goodness of our clustering. Therefore we can determine a good number of clusters K maximizing the silhouette measure.

For the hierarchical clustering of our features, we use the silhouette measure to identify the best choice of linkage criterion and a good number K of clusters. The R package *cluster* implements the silhouette index with the function *silhouette*. We plot the silhouette index for different choices of linkage criterion and number of clusters K (figure 7). We choose to select a number of clusters $K = 5$ which corresponds to a maximum of the silhouette index. The linkage criterion associated is the average linkage.

The result of hierarchical clustering with $K = 5$ clusters is given figure 8. The dendrogram is composed of 4 clusters with only two or three features which have index larger than 20. All the remaining features are in the last cluster. Therefore, we can see that the clustering manages to group all informative features $v_2, .., V_{19}$ in the same cluster. The uninformative features $V_{20}, .., V_{29}$ are disseminated into the remaining clusters.

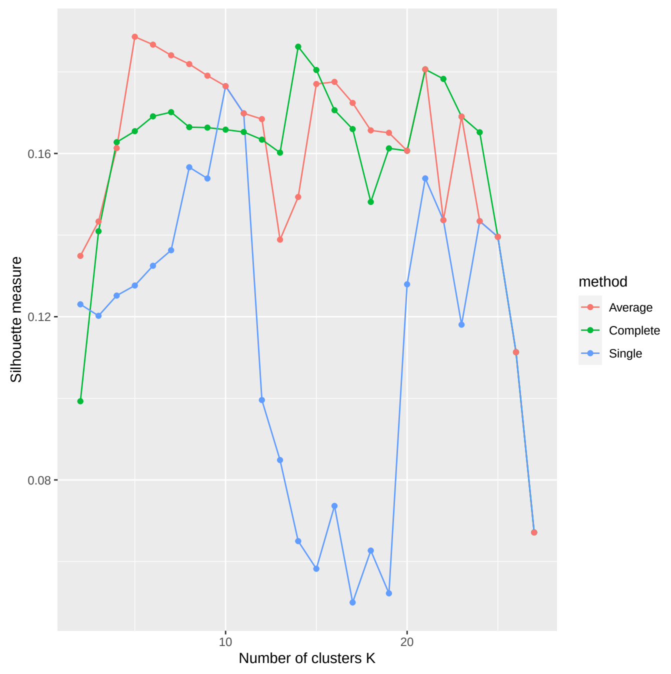


Figure 7: Silhouette index for different number of clusters K and different linkage criteria

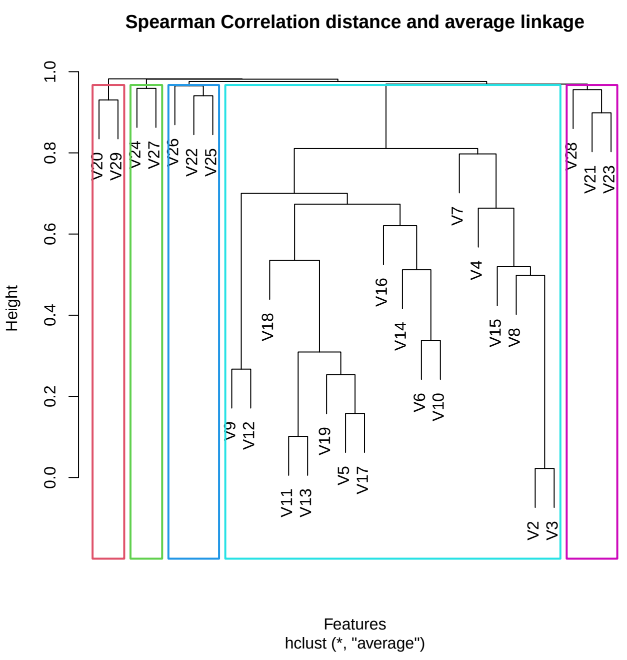


Figure 8: Hierarchical clustering with $K = 11$ clusters, average linkage and Manhattan distance

2.2 K-means

2.2.1 Method description

The K-means algorithm is a clustering method in which each observation belongs to the cluster with the nearest centroid. The objective is to partition the observations into K clusters $\mathcal{C}_1, .., \mathcal{C}_K$ so as to minimize the within-cluster sum of squares :

$$\mathcal{E} = \sum_{k=1}^{K} \sum_{x \in \mathcal{C}_k} ||x - m_k||^2 \tag{2.7}$$

The within-cluster sum of squares is the sum of the distances of each point from its associated cluster centroid defined by :

$$m_k = \frac{1}{\mathcal{C}_k} \sum_{x \in \mathcal{C}_k} x \tag{2.8}$$

To try minimizing the within-cluster sum of squares, the K-mean method proceeds as follows : all the points are assigned to the closest cluster centroid, then centroids are recomputed from newly formed clusters. This process is repeated until convergence, i.e when the clusters assignment is the same after one iteration of the process. A pseudo-code is given to detail the K-means procedure (Algorithm 1).

Algorithm 1 K-means algorithm

```
1: input: a data set $(s_1, .., s_n)$, a number of clusters $K$
2: Select randomly $K$ points from the data as centroids.
3: While not converged do :
4:     Assign each data points $s_i$ to a cluster: $x \to C_k$ with $k = \arg\max_{k \in [1..K]} ||s_i - m_k||$
5:     Update the cluster centroids : $m_k = \frac{1}{\mathcal{C}_k} \sum_{x \in \mathcal{C}_k} x$
```

We want to apply a K-mean procedure to the observations in our data set. Our goal is to assess the performance of the K-means algorithm in dividing the observations into the 2 classes of the z indicator. We start by removing the variable z from our data set. Then it is relevant to use a K-means procedure with $K = 2$ clusters in order to compare partitions composed of two subsets of our data.

As stated in equation (2.7), K-means method uses distances between observations. The distance calculation is sensitive to variations within the scales of the variables. The normalization of them will prevent outweighing features having a large number over features with smaller numbers [5]. We then apply the following normalization transformation to our observations :

$$X_{ij} \leftarrow \frac{X_{ij} - m_j}{\sigma_j} \qquad i = 1, , ., n \; j = 1, .., p$$

where $m_j = \frac{1}{n} \sum_{k=1}^{n} X_{kj}$ and $\sigma_j = \frac{1}{n} \sum_{k=1}^{n} (X_{kj} - m_j)^2$ are respectively the mean and standard deviation of the column j.

To verify that this choice of $K = 2$ is acceptable in terms of K-means goodness of fit we look at the silhouette measure computed with the R function *NbClust* from package *NbClust* for different numbers of clusters (figure 9). The highest silhouette measure is obtained with $K = 3$, but the silhouette measure obtained with $K = 2$ is very close and it is acceptable to keep $K = 2$.

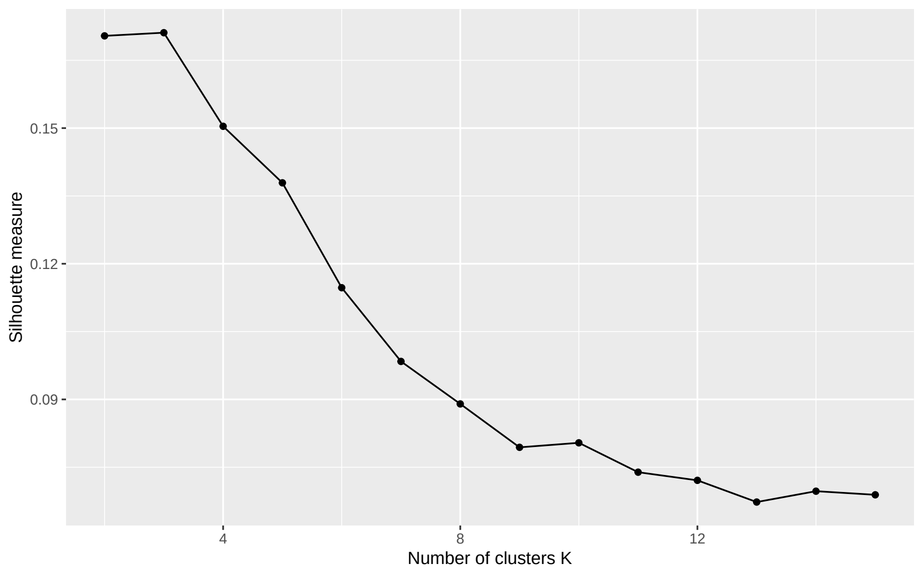


Figure 9: Silhouette index for different number of clusters K

We want the clusters to be compact and well separated, ie with a small within-cluster sum of squares and a large distance between centroids. Recall that the clusters created by K-mean method depend on the initial centroids chosen. We run 10,000 K-means procedures with the R function *kmeans* from *stats* package, and analyse the obtained clusters. The performances in terms of within-cluster sum of squares and distance between centroids are quite unstable (table 7). In particular, we can see that a few number of K-mean procedures perform poorly with a high within-cluster sum of squares and/or a small distance between centroids. We choose to keep the clustering with the lowest within-cluster sum of squares among the 10,000 K-means outcomes. It is noticed a posteriori that this clustering obtains a distance between centroids equal to 4.655 which is the minimum among the 10,000 K-means outcomes. We plot the final clustering in a 2D graph which are the two first principal components obtained by fitting a PCA procedure to our data set with R function *prcomp* from *stat* package (figure 10).

To avoid bad outcomes of K-means method, another solution is to use a different initialization procedure such as K-mean++ [6]. It has been done using the R function *kcca* from *flexclust* package. there was no sensible improvement of performances so we keep the final cluster obtained with our first method.

	Min.	1st Qu.	Median	3rd Qu.	Max.
Within-cluster sum of squares	25,135	25,135	25,135	25,135	27,993
Distance between centroids	3.953	4.655	4.655	4.655	4.655

Table 7: Within cluster Sum of squares (SS) for $N = 10^4$ K-means methods

2.2.2 dividing the observations into the classes of the “z” indicator with k-mean

We now assess the performance of the K-means algorithm in dividing the observations into the classes of the ”z” indicator by computing the Rand index. In our study, the Rand index is a measure of the similarity between two data clusterings : the clustering with respect to the z indicator and the clustering obtained with K-means. It is defined by [7] :

$$I_{Rand} = \frac{a+b}{\binom{n}{2}} \tag{2.9}$$

Where a is the number of observations in cluster 0 and with label $z = 0$, and b is the number of observations in cluster 1 and with label $z = 1$. Another approach consists in seeing the problem as a classification problem. We consider the z indicator as the real label and the predicted label of an observation is given by its assigned cluster by K-mean method with $K = 2$ clusters. The Rand index can then be written equivalently :

$$I_{Rand} = \frac{TP + TN}{\binom{n}{2}} \tag{2.10}$$

Where TP measures the proportion of positives ($z = 1$) that are correctly identified and TN measures the proportion of negatives ($z = 0$) that are correctly identified. We use the R function *rand.index* from *fossil* package.

$$I_{Rand}^{(obs)} = 0.7173 \tag{2.11}$$

When partitioning the data in 2 clusters, the Rand index is the accuracy computed for the K-mean method considered as a classification algorithm with respect to the label z. The obtained Rand index is very low compared to the accuracy obtained with the classification algorithms in question 1. The K-mean algorithm seen as a classifier performs poorly on our data set.

It can be noticed that K-mean method gives clusters almost linearly separable in the 2D plane formed by the two first principal components (figure 10). There is no similar pattern for the clustering based on z label. In particular, the observations in the top right corner are roughly equally distributed between the two classes $z = 0$ and $z = 1$. The K-means is not able to capture this nuance and classify all these observations in the class 0, resulting in a high classification error for these observations.

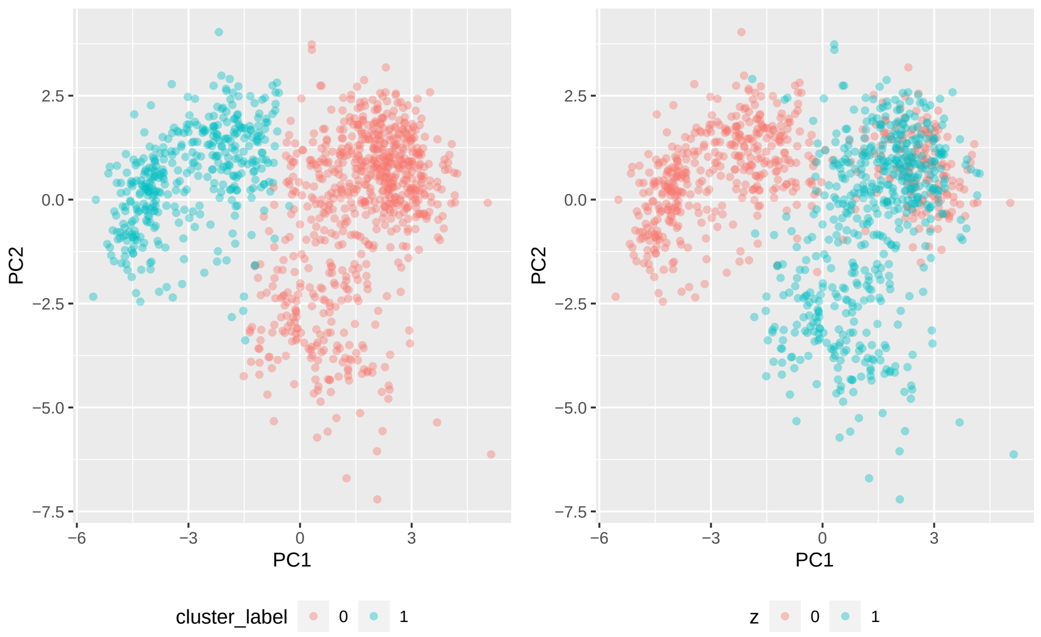


Figure 10: Clustering with K-mean (Left) and with respect to z class (Right)

3 Question 3

3.1 Data summary

We have a data set composed of $N = 200$ train observations and $N_{test} = 10$ test observations. The explicative variable is x and the response of interest is y. As we are in 2-dimensions, we can have a look at the train data with a scatter plot (figure 11). Both the variables x and y are continuous and lie in $\mathbb{R}$. This is a regression task we want to address using 2 machine learning methods : the Kernel Ridge Regression and the Gaussian process regressor.

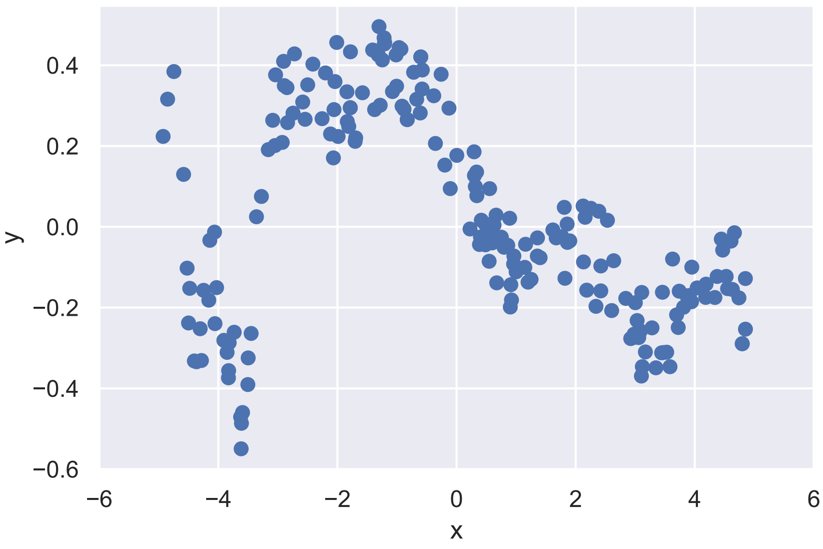


Figure 11: Scatter plot of the train data

3.2 Kernel Ridge Regression

3.2.1 Model Description

Let assume that the data is generated according to the following model :

$$y = f(x, \boldsymbol{\theta}) + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

We consider a regression function of the form :

$$f(x, \boldsymbol{\theta}) = \sum_{m=0}^{p} \theta_m \phi_m(x)$$

where p is an hyper-parameter which corresponds to the degree of the polynomial kernel. In the particular case of the polynomial kernel, we know the expression of the basis functions :

$$\phi_m(x) = \sqrt{c_{nm}} x^{m-1} \qquad m = 0, .., M \tag{3.1}$$

where $c_{nm} = \binom{n}{m} = \frac{n!}{m!(n-m)!}$. We introduce the design matrix :

$$\boldsymbol{\phi}_x = [\phi_j(x_i)]_{(i,j) \in [1..n] \times [0..p]} \tag{3.2}$$

For the kernel Ridge regression, the loss function that we use is :

$$\mathcal{L}(\boldsymbol{\theta}) = (\mathbf{y} - \boldsymbol{\phi}_x \boldsymbol{\theta})^T (\mathbf{y} - \boldsymbol{\phi}_x \boldsymbol{\theta}) + \alpha ||\boldsymbol{\theta}||_2^2 \tag{3.3}$$

with $\alpha > 0$. Setting $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = (-\mathbf{Z}^T \mathbf{y} + \mathbf{Z}^T \mathbf{Z} \boldsymbol{\theta}) + 2\alpha \boldsymbol{\theta} = 0$, we obtain the solution to the regularised least squares problem :

$$\theta^* = (\boldsymbol{\phi}_x^T \boldsymbol{\phi}_x + \alpha \mathbf{I}_M)^{-1} \boldsymbol{\phi}_x^T \mathbf{y} \tag{3.4}$$

which can be done because the matrix $\boldsymbol{\phi}_x^T \boldsymbol{\phi}_x + \alpha \mathbf{I}_M$ is always invertible as it has only striclty positive eigen values for $\alpha > 0$. We now introduce the polynomial kernel function :

$$K(x_i, x_j) = (x_i x_j + 1)^p \tag{3.5}$$

And the associated kernel matrix :

$$\mathbf{K}(\mathbf{x},\mathbf{x}) = [K(x_i,x_j)]_{(i,j)\in[1..n]\times[1..n]} \tag{3.6}$$

We can then make prediction for a new observation x^* with the formula :

$$y^* = \sum_{i=1}^{N} K(x^*, x_i)c_i \tag{3.7}$$

where $\mathbf{c} = (\mathbf{K}(\mathbf{x},\mathbf{x}) + \alpha \mathbf{I}_n)^{-1}\mathbf{y}$.

3.2.2 Implementation in Python

We fit a Kernel Ridge Regression to our data with the package sklearn.kernel_ridge.

We first use the Kernel Ridge Regression model with the default hyper-parameters $p = 3$ and $\alpha = 1$. We observe that this model has an important bias and does not fit well the train data (Figure 12). To try reducing the bias, we perform a hyper-parameter tuning with the package sklearn.model_selection.RandomizedSearchCV. We create a range of hyperparameters with p going from 1 to 20 and α going from 10^{-3} to 10. The randomized grid search will draw randomly 10000 pairs of hyper-parameters (p, α), and perform a 5-fold cross validation with each pair. The model that obtained the lowest cross-validation mean squared error will be selected as the final model. With this method, we obtain a couple of hyper-parameter :

$$\hat{p} = 11 \qquad\qquad \hat{\alpha} = 1.4623 \tag{3.8}$$

The final model is observed to fit much better the train data than the default model (figure 12). The Mean squared error on the train data is much lower for the final model (Table 8).

We now make predictions on the unseen data contained in the test set. The final model with $\hat{p} = 11$ perform very poorly on the test set (figure 13). The final model overfits the training data : The observations on the test set are slightly bigger than the observations and the final model is not able to predict them well. In the contrary, The default model with $p = 3$ predicts the test data not too bad. It has a higher bias but a lower variance than the final model.

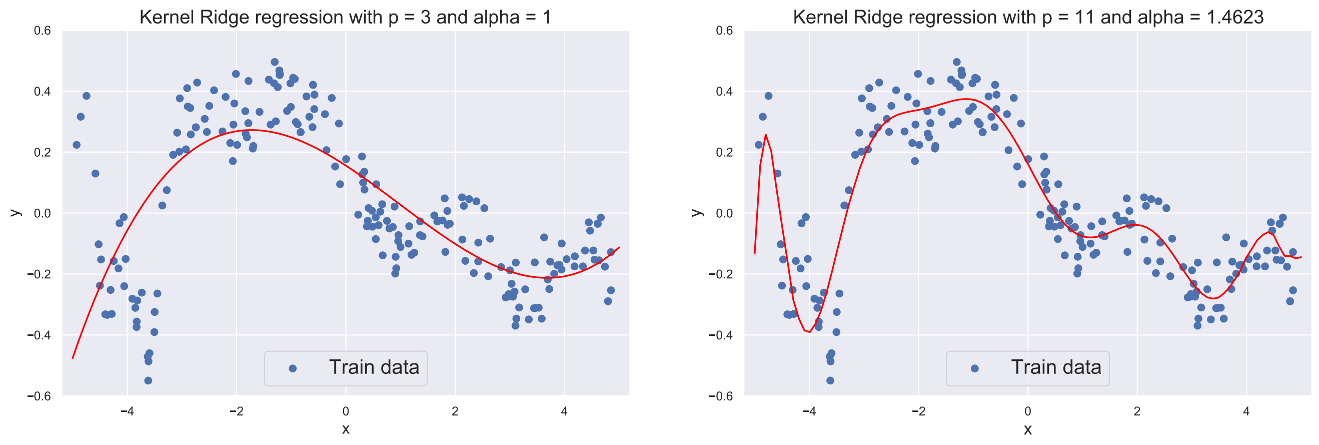


Figure 12: Predictions of the Kernel Ridge Regression on the train data

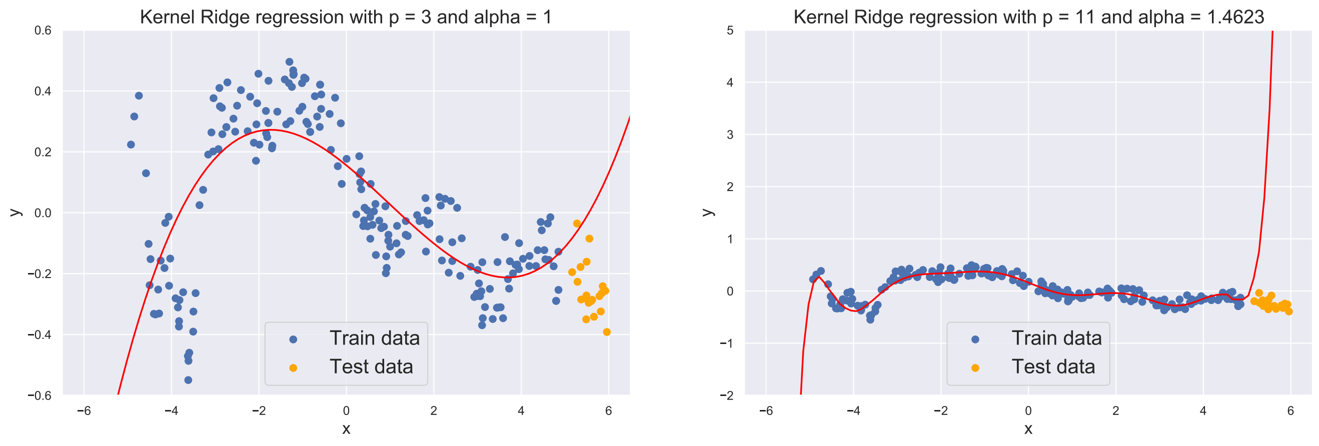


Figure 13: Predictions of the Kernel Ridge Regression on the test data

	MSE on training set	MSE on test set
Default KRR	3.0839×10^{-2}	8.4984×10^{-2}
Final KRR	9.7059×10^{-3}	89.4858

Table 8: MSE calculated for Kernel Ridge Regression with and without hyper-parameters tuning

In conclusion, we have not managed to fit a Kernel Ridge regression model that achieves a good tradeoff between bias and variance. It leads us to study another class of models, the Gaussian Process regressors.

3.3 Gaussian Process Regressor

3.3.1 Model description

Let assume that the data is generated according to the following model :

$$y = f(x, \theta) + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{3.9}$$

In a Bayesian approach, we consider the function f as a Gaussian Process $\{f_x = f(x),\, x \in \mathbb{R}\}$. It means that for any subset $(x_1, .., x_d) \in \mathbb{R}^d$ for some $d \in \mathbb{N}$, the random vector $(f_1, .., f_d) = (f(x_1), .., f(x_d))$ follows a multivariate Gaussian distribution.

A Gaussian process is entirely specified by its mean function $m(.)$ and its covariance function $k(., .)$. In our Bayesian approach, we choose a prior distribution on the functions f by specifying a mean function m and a kernel k. We then use Bayes theorem to obtain the posterior distribution over the functions f :

$$p(f(.)|X, y) = \frac{p(y|f(.), X)p(f(.)|X)}{p(y|X)} \tag{3.10}$$

Using the properties of Gaussian distribution, we obtain the posterior :

$$f(.)|X, y \sim GP(m_{post}(.), k_{post}) \tag{3.11}$$

where :

$$m_{post}(.) = m(.) + k(., X)(K + \sigma^2 I)^{-1}(\mathbf{y} - m(X)) \tag{3.12}$$

$$k_{post}(.) = k(., .) + k(., X)(K + \sigma^2 I)^{-1}K(X, .) \tag{3.13}$$

We can then compute the predicted distribution f_* for a new instance X^* :

$$f_*|X, y, X^* \sim \mathcal{N}\left(m_{post}(X_*), k_{post}(X_*, X_*)\right) \tag{3.14}$$

The Gaussian process regressor has the following hyperparameters that we need to determine : the parameters of the mean function $m(.)$, the parameters of the kernel $k(., .)$ and the noise variance σ_ϵ. In the following, we choose an a priori mean function equal to everywhere.

We create a vector $\boldsymbol{\theta}$ which contains all the hyperparameters cited previously. In our Bayesian approach, we place a prior distribution $p(\boldsymbol{\theta})$ on the hyperparameters and the posterior distribution of $\boldsymbol{\theta}$ is given by Bayes' theorem :

$$p(\boldsymbol{\theta}|X, y) = \frac{p(\boldsymbol{\theta})p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})}{p(\mathbf{y}|\mathbf{X})} \tag{3.15}$$

We aim to maximize the posterior distribution, or equivalently the logarithm of the posterior distribution :

$$\boldsymbol{\theta}^* \in \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}) + \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) - \log p(\mathbf{y}|\mathbf{X}) \tag{3.16}$$

Removing the last term not dependent of $\boldsymbol{\theta}$ and choosing a uniform prior distribution $p(\boldsymbol{\theta})$, this is equivalent to the optimization problem :

$$\boldsymbol{\theta}^* \in \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) \tag{3.17}$$

We then aim to maximize the marginal log-likelihood, which can be written :

$$\log p(y|X, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{y}^T \mathbf{K}_{\boldsymbol{\theta}}^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{K}_{\boldsymbol{\theta}}| + C^{ste} \tag{3.18}$$

With $\mathbf{K}_{\boldsymbol{\theta}} = K + \sigma_\epsilon I$. To maximize this function, we can use a gradient-based optimisation thanks to the following closed form :

$$\frac{\partial \log p(y|X, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}_i} = \frac{1}{2}Tr\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^T - \mathbf{K}_{\boldsymbol{\theta}}^{-1})\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}_i}\right) \tag{3.19}$$

with $\boldsymbol{\alpha} = \mathbf{K}_{\boldsymbol{\theta}}{}^{-1}\mathbf{y}$. We present three different kernels, their hyperparameters and the marginal likelihood maximization associated.

3.3.2 Gaussian Kernel

The Gaussian Covariance function is given by :

$$k_{gauss} = \sigma_f^2 \exp\left(-\frac{(x_i - x_j)^2}{2l^2}\right) \tag{3.20}$$

3 hyper-parameters need to be determined for this kernel : the noise variance σ_ϵ, the amplitude of the latent function σ_f and the length-scale l. Let introduce $\boldsymbol{\theta} = [\sigma_\epsilon \quad \sigma_f \quad l]^T$, we compute the partial derivatives $\frac{\partial K_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}_i}$ from (3.19) in order to implement a gradient-based optimizer for the Gaussian kernel :

$$\begin{aligned}
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \sigma_\epsilon}(x_i, x_j) &= 2\sigma_\epsilon \delta_{ij} \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \sigma_f}(x_i, x_j) &= 2\sigma_f \exp\left(-\frac{(x_i - x_j)^2}{2l^2}\right) \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial l}(x_i, x_j) &= \frac{(x_i - x_j)^2}{l^3} \sigma_f^2 \exp\left(-\frac{(x_i - x_j)^2}{2l^2}\right)
\end{aligned}$$

Where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. The Gaussian kernel is implemented in Python in the class sklearn.gaussian_process.kernels

3.3.3 Matérn kernel

The Matérn Covariance function is given by :

$$k_{Mat}(x_i, x_j) = \sigma_f^2 \left(1 + \frac{\sqrt{3}||x_i - x_j||}{l}\right) \exp\left(-\frac{\sqrt{3}||x_i - x_j||}{l}\right) \tag{3.21}$$

3 hyper-parameters need to be determined for this kernel : the noise variance σ_ϵ, the amplitude of the latent function σ_f and the length-scale l. Let introduce $\boldsymbol{\theta} = [\sigma_\epsilon \quad \sigma_f \quad l]^T$, we compute the partial derivatives $\frac{\partial K_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}_i}$ from (3.19) in order to implement a gradient-based optimizer for the Matérn kernel :

$$\begin{aligned}
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \sigma_\epsilon}(x_i, x_j) &= 2\sigma_\epsilon \delta_{ij} \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \sigma_f}(x_i, x_j) &= 2\sigma_f \left(1 + \frac{\sqrt{3}||x_i - x_j||}{l}\right) \exp\left(-\frac{\sqrt{3}||x_i - x_j||}{l}\right) \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial l}(x_i, x_j) &= \sigma_f^2 \frac{3(x_i - x_j)^2}{l^3} \exp\left(-\frac{\sqrt{3}||x_i - x_j||}{l}\right)
\end{aligned}$$

The Matérn kernel is implemented in Python in the class sklearn.gaussian_process.kernels.Matern.

3.3.4 Rational Quadratic Kernel

The Rational Quadratic Covariance function is given by [8] :

$$k_{RQ}(x_i, x_j) = \sigma_f^2 \left(1 + \frac{(x_i - x_j)^2}{2\alpha l^2}\right)^{-\alpha} \tag{3.22}$$

4 hyper-parameters need to be determined for this Kernel : the noise variance σ_ϵ, the amplitude of the latent function σ_f, the length-scale l and α is the scale mixture parameter. Let introduce $\boldsymbol{\theta} = [\sigma_\epsilon \quad \sigma_f \quad l \quad \alpha]^T$, we compute the partial derivative $\frac{\partial K_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}_i}$ from (3.19) in order to implement a gradient-based optimizer for the Rational Quadratic kernel :

$$\begin{aligned}
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \sigma_\epsilon}(x_i, x_j) &= 2\sigma_\epsilon \delta_{ij} \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \sigma_f}(x_i, x_j) &= 2\sigma_f \left(1 + \frac{(x_i - x_j)^2}{2\alpha l^2}\right)^{-\alpha} \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial l}(x_i, x_j) &= \sigma_f^2 \frac{(x_i - x_j)^2}{l^3} \left(1 + \frac{(x_i - x_j)^2}{2\alpha l^2}\right)^{-\alpha-1} \\
\frac{\partial \mathbf{K}_{\boldsymbol{\theta}}}{\partial \alpha}(x_i, x_j) &= \sigma_f^2 \left(-\log\left(1 + \frac{(x_i - x_j)^2}{2\alpha l^2}\right) + \frac{(x_i - x_j)^2}{(x_i - x_j)^2 + 2\alpha l^2}\right) \left(1 + \frac{(x_i - x_j)^2}{2\alpha l^2}\right)^{-\alpha}
\end{aligned}$$

The Rational Quadratic kernel is implemented in Python in the class sklearn.gaussian_process.kernels.RationalQuadratic.

3.3.5 Implementation in Python

We fit a Gaussian Process regressor to our data with the package sklearn.gaussian_process.GaussianProcessRegressor.

Recall that the optimization problem (3.17) requires a uniform prior on $\boldsymbol{\theta}$. All the hyper-parameters evocated above lie in $(0, +\infty)$, we then need to restrict ourselves to a compact subset of $(0, +\infty)$ to ensure a proper uniform distribution. Consequently, we add the constraint $\boldsymbol{\theta}_i \in (10^{-5}, 10^5)$ in the optimization problem (3.17). The hyperparameters are tuned with a gradient-based optimizer as described in section 3.3.1. Gradient-based optimization methods might suffer from multiple local optima. To counter this concern, we iterate the gradient-based optimization method with 50 different initial values. We compare the local minima obtained and we keep the hyper-parameters that obtain the highest marginal log-likelihood. The final hyper-parameters are reported in table 9.

Parameter	MLE
σ_ϵ	5.2416×10^{-3}
σ_f	5.0034×10^{-2}
l	3.1229×10^{-1}

Parameter	MLE
σ_ϵ	5.1692×10^{-3}
σ_f	5.4460×10^{-2}
l	5.9516×10^{-1}

Parameter	MLE
σ_ϵ	5.2220×10^{-3}
σ_f	5.4531×10^{-2}
l	3.8654×10^{-1}
α	7.0195×10^{-1}

Table 9: Maximum Likelihood Estimators for parameters of the Gaussian Process regressor with Gaussian kernel (Left), Matérn kernel (Center) and Rational Quadratic kernel (Right)

The 3 models with different kernels fit the data very well and obtain similar performances in terms of Cross-validation MSE and Marginal log-likelihood (table 10). We introduce the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC) to help us selecting a model :

$$AIC = 2k - 2\log(\hat{L}) \qquad BIC = k\log(n) - 2\log(\hat{L})$$

where $\hat{L}$ is the maximized value of the marginal log-likelihood, k is the number of parameters to be estimated (dimensionality of θ) and $n = 200$ is the number of observations in the training set. AIC and BIC deal with the trade-off between the goodness of fit of the model and its simplicity. We select the Gaussian Process regressor with Matérn kernel as our final model because it obtains the lowest AIC and BIC (table 10). It can be observed that the Gaussian Process Regressor tends to make predictions close to the a priori mean equal to 0 when the training set does not carry information in the neighborhood of a new unseen point. As a result, the Gaussian Process regressor performs much better as it avoids to give exploding predictions for unexplored subsets of the observations space like did the Kernel Ridge regressor (table 11).

	Cross-validation MSE	AIC	BIC	Marginal log-likelihood
k_{Gauss}	6.9437×10^{-3}	-364.6211	-354.7262	185.3106
$k_{Matérn}$	7.1174×10^{-3}	-365.7772	-355.8823	185.8886
k_{Quad}	6.9900×10^{-3}	-365.4262	-354.7262	186.7131

Table 10: Performances of Gaussian Process regressors with different kernels

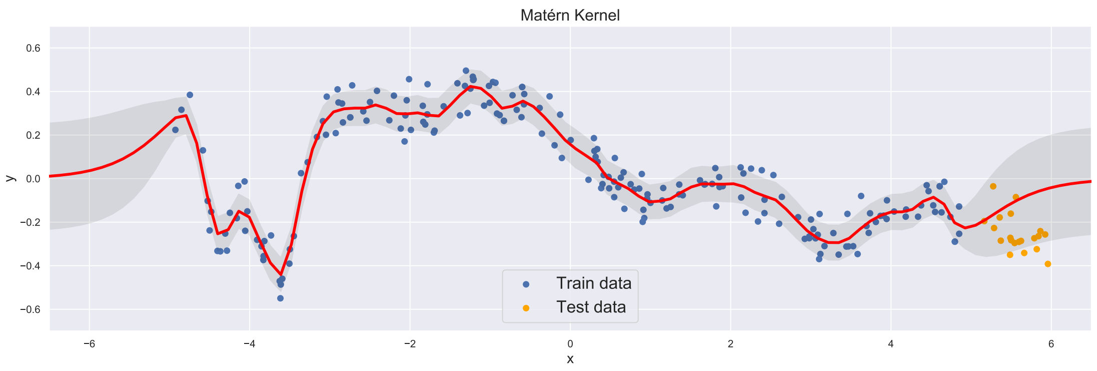


Figure 14: Predictions of the Gaussian Process regressor with Matérn kernel on the test data

	MSE on training set	MSE on test set
Final KRR	9.7059×10^{-3}	89.4858
Gaussian Process Regressor	4.1595×10^{-3}	34.1952×10^{-3}

Table 11: MSE calculated for Kernel Ridge regressor(KRR) and Gaussian Process regressor with Matérn kernel on the training and test set

References

[1] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[2] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[3] Morten Fagerland, Stian Lydersen, and Petter Laake. The mcnemar test for binary matched-pairs data: Mid-p and asymptotic are better than exact conditional. *BMC medical research methodology*, 13:91, 07 2013.

[4] A.R. Webb. *Statistical Pattern Recognition*. Wiley InterScience electronic collection. Wiley, 2003.

[5] Ismail Mohamad and Dauda Usman. Standardization and its effects on k-means clustering algorithm. *Research Journal of Applied Sciences, Engineering and Technology*, 6:3299–3303, 09 2013.

[6] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. volume 8, pages 1027–1035, 01 2007.

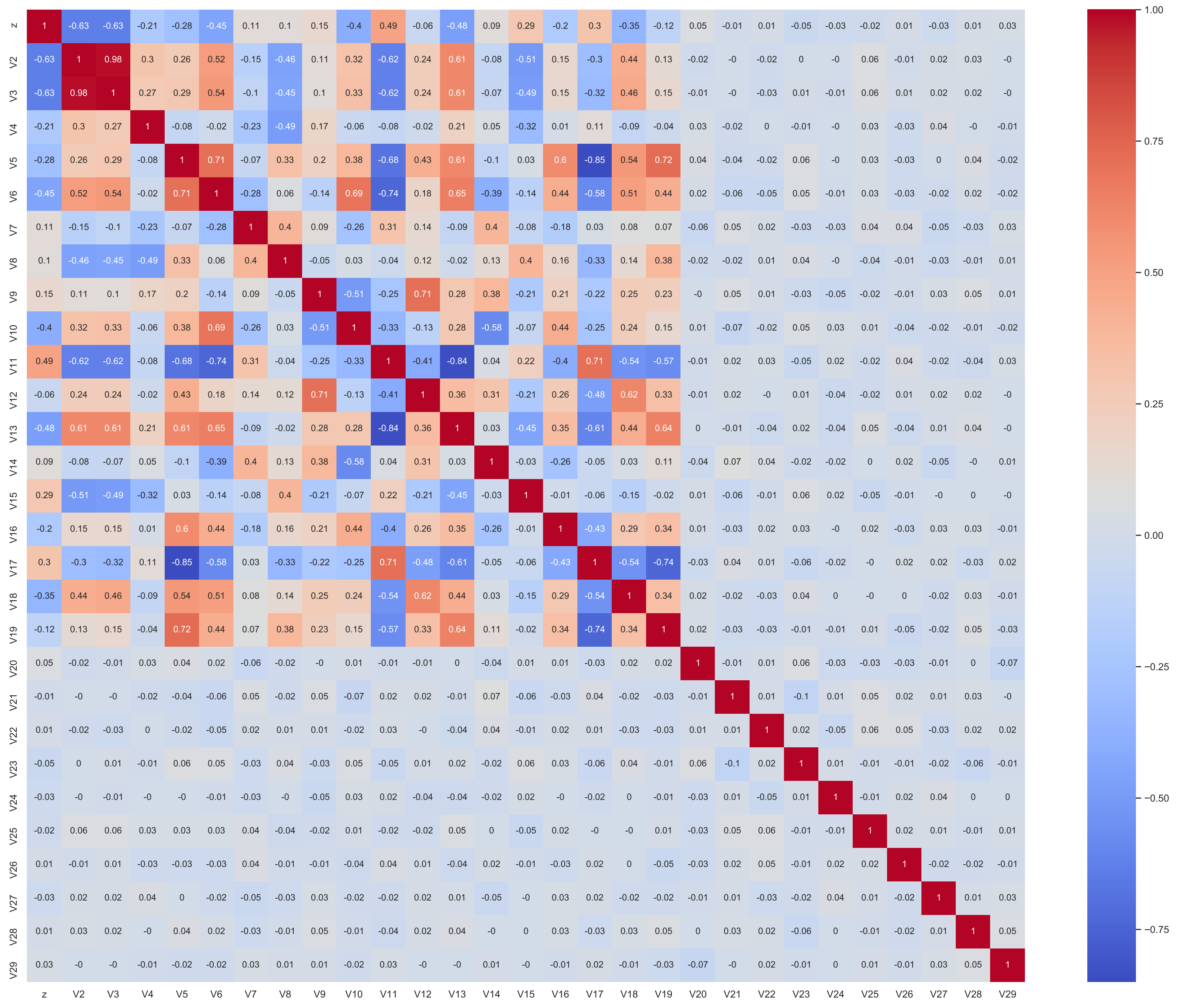
[7] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[8] David Duvenaud. The kernel cookbook, `https://www.cs.toronto.edu/~duvenaud/cookbook/`.

[9] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50 – 60, 1947.

[10] John W. Pratt. Robustness of some procedures for the two-sample location problem. *Journal of the American Statistical Association*, 59(307):665–680, 1964.

A Correlation Matrix



	z	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28	V29
z	1	-0.63	-0.63	-0.21	-0.28	-0.45	0.11	0.1	0.15	-0.4	0.49	-0.06	-0.48	0.09	0.29	-0.2	0.3	-0.35	-0.12	0.05	-0.01	0.01	-0.05	-0.03	-0.02	0.01	-0.03	0.01	0.03
V2	-0.63	1	0.98	0.3	0.26	0.52	-0.15	-0.46	0.11	0.32	-0.62	0.24	0.61	-0.08	-0.51	0.15	-0.3	0.44	0.13	-0.02	-0	-0.02	0	-0	0.06	-0.01	0.02	0.03	-0
V3	-0.63	0.98	1	0.27	0.29	0.54	-0.1	-0.45	0.1	0.33	-0.62	0.24	0.61	-0.07	-0.49	0.15	-0.32	0.46	0.15	-0.01	-0	-0.03	0.01	-0.01	0.06	0.01	0.02	0.02	-0
V4	-0.21	0.3	0.27	1	-0.08	-0.02	-0.23	-0.49	0.17	-0.06	-0.08	-0.02	0.21	0.05	-0.32	0.01	0.11	-0.09	-0.04	0.03	-0.02	0	-0.01	-0	0.03	-0.03	0.04	-0	-0.01
V5	-0.28	0.26	0.29	-0.08	1	0.71	-0.07	0.33	0.2	0.38	-0.68	0.43	0.61	-0.1	0.03	0.6	-0.85	0.54	0.72	0.04	-0.04	-0.02	0.06	-0	0.03	-0.03	0	0.04	-0.02
V6	-0.45	0.52	0.54	-0.02	0.71	1	-0.28	0.06	-0.14	0.69	-0.74	0.18	0.65	-0.39	-0.14	0.44	-0.58	0.51	0.44	0.02	-0.06	-0.05	0.05	-0.01	0.03	-0.03	-0.02	0.02	-0.02
V7	0.11	-0.15	-0.1	-0.23	-0.07	-0.28	1	0.4	0.09	-0.26	0.31	0.14	-0.09	0.4	-0.08	-0.18	0.03	0.08	0.07	-0.06	0.05	0.02	-0.03	-0.03	0.04	0.04	-0.05	-0.03	0.03
V8	0.1	-0.46	-0.45	-0.49	0.33	0.06	0.4	1	-0.05	0.03	-0.04	0.12	-0.02	0.13	0.4	0.16	-0.33	0.14	0.38	-0.02	-0.02	0.01	0.04	-0	-0.04	-0.01	-0.03	-0.01	0.01
V9	0.15	0.11	0.1	0.17	0.2	-0.14	0.09	-0.05	1	-0.51	-0.25	0.71	0.28	0.38	-0.21	0.21	-0.22	0.25	0.23	-0	0.05	0.01	-0.03	-0.05	-0.02	-0.01	0.03	0.05	0.01
V10	-0.4	0.32	0.33	-0.06	0.38	0.69	-0.26	0.03	-0.51	1	-0.33	-0.13	0.28	-0.58	-0.07	0.44	-0.25	0.24	0.15	0.01	-0.07	-0.02	0.05	0.03	0.01	-0.04	-0.02	-0.01	-0.02
V11	0.49	-0.62	-0.62	-0.08	-0.68	-0.74	0.31	-0.04	-0.25	-0.33	1	-0.41	-0.84	0.04	0.22	-0.4	0.71	-0.54	-0.57	-0.01	0.02	0.03	-0.05	0.02	-0.02	0.04	-0.02	-0.04	0.03
V12	-0.06	0.24	0.24	-0.02	0.43	0.18	0.14	0.12	0.71	-0.13	-0.41	1	0.36	0.31	-0.21	0.26	-0.48	0.62	0.33	-0.01	0.02	-0	0.01	-0.04	-0.02	0.01	0.02	0.02	-0
V13	-0.48	0.61	0.61	0.21	0.61	0.65	-0.09	-0.02	0.28	0.28	-0.84	0.36	1	0.03	-0.45	0.35	-0.61	0.44	0.64	0	-0.01	-0.04	0.02	-0.04	0.05	-0.04	0.01	0.04	-0
V14	0.09	-0.08	-0.07	0.05	-0.1	-0.39	0.4	0.13	0.38	-0.58	0.04	0.31	0.03	1	-0.03	-0.26	-0.05	0.03	0.11	-0.04	0.07	0.04	-0.02	-0.02	0	0.02	-0.05	-0	0.01
V15	0.29	-0.51	-0.49	-0.32	0.03	-0.14	-0.08	0.4	-0.21	-0.07	0.22	-0.21	-0.45	-0.03	1	-0.01	-0.06	-0.15	-0.02	0.01	-0.06	-0.01	0.06	0.02	-0.05	-0.01	-0	0	-0
V16	-0.2	0.15	0.15	0.01	0.6	0.44	-0.18	0.16	0.21	0.44	-0.4	0.26	0.35	-0.26	-0.01	1	-0.43	0.29	0.34	0.01	-0.03	0.02	0.03	-0	0.02	-0.03	0.03	0.03	-0.01
V17	0.3	-0.3	-0.32	0.11	-0.85	-0.58	0.03	-0.33	-0.22	-0.25	0.71	-0.48	-0.61	-0.05	-0.06	-0.43	1	-0.54	-0.74	-0.03	0.04	0.01	-0.06	-0.02	-0	0.02	0.02	-0.03	0.02
V18	-0.35	0.44	0.46	-0.09	0.54	0.51	0.08	0.14	0.25	0.24	-0.54	0.62	0.44	0.03	-0.15	0.29	-0.54	1	0.34	0.02	-0.02	-0.03	0.04	0	-0	0	-0.02	0.03	-0.01
V19	-0.12	0.13	0.15	-0.04	0.72	0.44	0.07	0.38	0.23	0.15	-0.57	0.33	0.64	0.11	-0.02	0.34	-0.74	0.34	1	0.02	-0.03	-0.03	-0.01	-0.01	0.01	-0.05	-0.02	0.05	-0.03
V20	0.05	-0.02	-0.01	0.03	0.04	0.02	-0.06	-0.02	-0	0.01	-0.01	-0.01	0	-0.04	0.01	0.01	-0.03	0.02	0.02	1	-0.01	0.01	0.06	-0.03	-0.03	-0.03	-0.01	0	-0.07
V21	-0.01	-0	-0	-0.02	-0.04	-0.06	0.05	-0.02	0.05	-0.07	0.02	0.02	-0.01	0.07	-0.06	-0.03	0.04	-0.02	-0.03	-0.01	1	0.01	-0.1	0.01	0.05	0.02	0.01	0.03	-0
V22	0.01	-0.02	-0.03	0	-0.02	-0.05	0.02	0.01	0.01	-0.02	0.03	-0	-0.04	0.04	-0.01	0.02	0.01	-0.03	-0.03	0.01	0.01	1	0.02	-0.05	0.06	0.05	-0.03	0.02	0.02
V23	-0.05	0	0.01	-0.01	0.06	0.05	-0.03	0.04	-0.03	0.05	-0.05	0.01	0.02	-0.02	0.06	0.03	-0.06	0.04	-0.01	0.06	-0.1	0.02	1	0.01	-0.01	-0.01	-0.02	-0.06	-0.01
V24	-0.03	-0	-0.01	-0	-0	-0.01	-0.03	-0	-0.05	0.03	0.02	-0.04	-0.04	-0.02	0.02	-0	-0.02	0	-0.01	-0.03	0.01	-0.05	0.01	1	-0.01	0.02	0.04	0	0
V25	-0.02	0.06	0.06	0.03	0.03	0.03	0.04	-0.04	-0.02	0.01	-0.02	-0.02	0.05	0	-0.05	0.02	-0	-0	0.01	-0.03	0.05	0.06	-0.01	-0.01	1	0.02	0.01	-0.01	0.01
V26	0.01	-0.01	0.01	-0.03	-0.03	-0.03	0.04	-0.01	-0.01	-0.04	0.04	0.01	-0.04	0.02	-0.01	-0.03	0.02	0	-0.05	-0.03	0.02	0.05	-0.01	0.02	0.02	1	-0.02	-0.02	-0.01
V27	-0.03	0.02	0.02	0.04	0	-0.02	-0.05	-0.03	0.03	-0.02	-0.02	0.02	0.01	-0.05	-0	0.03	0.02	-0.02	-0.02	-0.01	0.01	-0.03	-0.02	0.04	0.01	-0.02	1	0.01	0.03
V28	0.01	0.03	0.02	-0	0.04	0.02	-0.03	-0.01	0.05	-0.01	-0.04	0.02	0.04	-0	0	0.03	-0.03	0.03	0.05	0	0.03	0.02	-0.06	0	-0.01	-0.02	0.01	1	0.05
V29	0.03	-0	-0	-0.01	-0.02	-0.02	0.03	0.01	0.01	-0.02	0.03	-0	-0	0.01	-0	-0.01	0.02	-0.01	-0.03	-0.07	-0	0.02	-0.01	0	0.01	-0.01	0.03	0.05	1

Figure 15: Spearman Correlation matrix for question 1 data set

B Mann–Whitney U test

The Mann-Whitney U test is a nonparametric test that compare two groups without making the assumption that their values are normally distributed [9]. This test is made for every feature, the first group is composed by the feature values for label $z = 0$ and the second group is composed by the feature values for label $z = 1$. Some features have not the density of a normal distribution when conditioning on label (figure 1) so it is particularly relevant to use a test not making the assumption of normality. For a given feature V_i, Mann-Whitney U test is based on the following hypothesis [10]:

$$\begin{cases} H_0: & \text{the distributions of } V_i|z=0 \text{ and } V_i|z=1 \text{ are equal} \\ H_1: & \text{the distributions of } V_i|z=0 \text{ and } V_i|z=1 \text{ are not equal} \end{cases}$$

Mann-Whitney U test require that all the observations from both groups are independent of each other and that the populations are ordinal. These two statements are clearly true for all our features which are real number measures from independent gene observations.

Let $(V_{i1}^{(0)}, .., V_{in_0}^{(0)})$ and $((V_{i1}^{(1)}, .., V_{in_1}^{(1)})$ be the values of feature V_i with label $z = 0$ and $z = 1$ respectively. The test is based on the statistic U defined by :

$$U = \sum_{k=1}^{n_0} \sum_{j=1}^{n_1} S(V_{ik}^{(0)}, V_{ij}^{(1)})$$

where S is defined for $(x, y) \in \mathbb{R}^2$ by :

$$S(x,y) = \begin{cases} 1, & \text{if } y < x, \\ \frac{1}{2}, & \text{if } y = x, \\ 0, & \text{if } y > x. \end{cases}$$

Our samples sizes are much higher than 20 and we can compare the statistics U to a normal distribution to compute p-values. The Mann-Whitney U test is implemented in Python with scipy.stats.mannwhitneyu. The sorted p-values calculated for each feature are reported in table 12.

Feature	p-value
V_2	2.5638×10^{-51}
V_3	3.7059×10^{-49}
V_{13}	5.3444×10^{-30}
V_{11}	1.3898×10^{-27}
V_6	3.4973×10^{-27}
V_{10}	8.8514×10^{-22}
V_{18}	5.8864×10^{-16}
V_{15}	2.3435×10^{-12}
V_5	6.5087×10^{-11}
V_{17}	2.0316×10^{-10}
V_4	3.1599×10^{-7}
V_{16}	1.2830×10^{-4}
V_9	3.6188×10^{-4}
V_{19}	3.5396×10^{-3}
V_{14}	7.9001×10^{-3}
V_8	3.6015×10^{-2}
V_7	7.0351×10^{-2}
V_{20}	9.6430×10^{-2}
V_{29}	9.8440×10^{-2}
V_{12}	0.1207
V_{23}	0.1223
V_{22}	0.1329
V_{27}	0.1749
V_{28}	0.2464
V_{26}	0.2898
V_{24}	0.3263
V_{21}	0.4014
V_{25}	0.4485

Table 12: p-values of Mann-Whitney U test

C Python code question 1

```
#!/usr/bin/env python
# coding: utf-8

# # Coursework 2 - Machine Learning - Question 1

# ## libraries

# In[3]:


import os

import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
get_ipython().run_line_magic('matplotlib', 'inline')

import pandas as pd

import random

import numpy as np
import numpy.random as rd

from scipy.stats import reciprocal
from scipy.stats import spearmanr
from scipy.stats import mannwhitneyu

from statsmodels.stats.contingency_tables import mcnemar

from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_roc_curve
from sklearn.metrics import roc_curve
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

import tensorflow as tf
from tensorflow import keras


# In[4]:


os.chdir(r"C:\Users\robin\Dropbox\Applications\Overleaf\Coursework_2_ML")
os.getcwd()

random.seed(42)


# ## Import data

# In[5]:


data = pd.read_csv("CID1945214.csv")


# In[6]:


```

```
data.shape


# In[7]:


y = data['z']
X = data.iloc[:,1:]
X.shape, y.shape


# In[8]:


n = data.shape[0]
p = data.shape[1]
n, p


# ## Create training set and testing set

# In[9]:


X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.5,
                                                    random_state=42,
                                                   shuffle = True)

print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)


# # exploratory data analysis

# In[10]:


data_train = pd.concat([pd.DataFrame(y_train), X_train], axis = 1)


# In[11]:


data_train.info()


# In[12]:


data_train.describe()


# In[13]:


df = pd.melt(data_train, data_train.columns[0], data_train.columns[1:])

g = sns.FacetGrid(df, col="variable", hue="z", col_wrap=7, height = 15)
g.map(sns.kdeplot, "value", shade=True)
g.fig.subplots_adjust( top = 0.85 )
plt.subplots_adjust(hspace=0.1, wspace=0.05)
n = 0
for ax, title in zip(g.axes.flat, data_train.columns[1:]):
    ax.set_title(title, size = 80)
    ax.yaxis.set_tick_params(labelsize=50)
    if n > 20 :
        ax.set_xlabel("value", size = 70)
        ax.xaxis.set_tick_params(labelsize=50)
    n += 1
```

```
plt.show()
g.savefig("data_summary.pdf", bbox_inches='tight', dpi = 300)


# In[14]:


data_size = data_train.iloc[:,range(12)]
sns.pairplot(data_size, hue='z', height=2.5)


# In[15]:


data_pattern = data_train.iloc[:,[0] + list(range(12,p))]
sns.pairplot(data_pattern, hue='z', height=2.5)


# In[16]:


interesting_features = ['z','V2', 'V3', 'V5', 'V6', 'V11', 'V13', 'V17', 'V18', 'V20', 'V25'
    ]
data_int = data[interesting_features]


# In[17]:


sns.pairplot(data_int, hue='z', height=2.5)


# In[22]:


correlation_matrix = data.corr(method='spearman').round(2)
# annot = True to print the values inside the square
plt.subplots(figsize=(25,20))
sns.heatmap(data=correlation_matrix, annot=True, cmap= 'coolwarm')
plt.savefig("corr_matrix.pdf", bbox_inches='tight')


# In[10]:


list_corr = []
for i in range(p-1):
    list_corr.append(("V"+str(i+2), abs(spearmanr(X_train.to_numpy()[:,i], y_train, nan_
        policy = 'omit')[0])))
def correl(M):
    return M[1]
sorted_corr = sorted(list_corr, key=correl, reverse = True)
sorted_corr


# In[11]:


label0 = y_train == 0
label1 = y_train == 1

X_train0 = X_train[label0].to_numpy()
X_train1 = X_train[label1].to_numpy()

list_pval = []
for i in range(p-1):
    list_pval.append(("V"+str(i+2), mannwhitneyu(X_train0[:,i], X_train1[:,i])[1]))
def correl(M):
    return M[1]
sorted_pval = sorted(list_pval, key=correl)
```

```
sorted_pval


# ## NA preprocessing

# In[19]:


# number of columns with NA

print('NaN occurrences in Columns:')
cols_with_na = data_train.isna().sum(axis = 0)>=1
n_cols_with_na = cols_with_na.sum()
print(n_cols_with_na)

# number of rows with NA

print('NaN occurrences in Rows:')
rows_with_na = data_train.isna().sum(axis = 1)>=1
n_rows_with_na = rows_with_na.sum()
print(n_rows_with_na)


# In[20]:


# Mean imputation

imp = SimpleImputer(missing_values=np.nan, strategy='median')
X_train = imp.fit_transform(X_train)
X_test = imp.transform(X_test)


# In[21]:


sum(y_train == 0), sum(y_train == 1)


# ## Random Forests

# In[22]:


rf_clf_basic = RandomForestClassifier(random_state = 42)
rf_clf_basic.fit(X_train, y_train)
y_pred_rf_basic = rf_clf_basic.predict(X_test)
y_pred_train_rf_basic = rf_clf_basic.predict(X_train)

train_acc_rf_basic = accuracy_score(y_pred_train_rf_basic, y_train)
test_acc_rf_basic = accuracy_score(y_pred_rf_basic, y_test)
train_acc_rf_basic.round(4), test_acc_rf_basic.round(4)


# ## Random Forests with parameters tuning

# In[92]:


# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 100, stop = 2100, num = 10)]

# Number of features to consider at every split
max_features = [3, 5, 7, 10, 15, 20, 25]

# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(5, 150, num = 10)]
max_depth.append(None)

# Minimum number of samples required to split a node
```

```
min_samples_split = [2, 5, 10]

# Minimum number of samples required at each leaf node
max_leaf_nodes = [1, 2, 5, 8, 11, 15]

# criterion
criterion = ['gini', 'entropy']

# min_samples_leaf
min_samples_leaf = [1, 2, 4, 8]


# In[93]:


param_rf = {'n_estimators': n_estimators,
            'max_features': max_features,
            'max_depth': max_depth,
            'min_samples_split': min_samples_split,
            'max_leaf_nodes' : max_leaf_nodes,
            'criterion' : criterion,
            'min_samples_leaf' : min_samples_leaf}


# In[94]:


rf_clf = RandomForestClassifier(random_state = 42)
grid_search = RandomizedSearchCV(estimator = rf_clf,
                                 param_distributions = param_rf,
                                 n_iter = 300,
                                 cv = 4,
                                 random_state=42,
                                 n_jobs = -1,
                                scoring = "accuracy")

grid_search.fit(X_train, y_train)


# In[95]:


grid_search.best_params_


# In[96]:


rf_clf_final = grid_search.best_estimator_


# In[97]:


rf_clf_final.fit(X_train, y_train)


# In[98]:


y_pred_rf_final = rf_clf_final.predict(X_test)
y_pred_train_rf_final = rf_clf_final.predict(X_train)


# In[99]:


train_acc_rf_final = accuracy_score(y_pred_train_rf_final, y_train)
test_acc_rf_final = accuracy_score(y_pred_rf_final, y_test)
train_acc_rf_final.round(4), test_acc_rf_final.round(4)
```

```
# ### Features importance - Random Forest

# In[31]:


feature_list = list(data.iloc[:,1:].columns)
importances = list(rf_clf_final.feature_importances_)
# List of tuples with variable and importance
feature_importances = [(feature, round(importance, 3)) for feature, importance in zip(
    feature_list, importances)]
# Sort the feature importances by most important first
feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)
# Print out the feature and importances
feature_importances


# In[32]:


x_values = list(range(len(importances)))

# List of features sorted from most to least important
sorted_importances = [importance[1] for importance in feature_importances]
sorted_features = [importance[0] for importance in feature_importances]
# Cumulative importances
cumulative_importances = np.cumsum(sorted_importances)
# Make a line graph
plt.plot(x_values, cumulative_importances, 'g-')
# Draw line at 95% of importance retained
plt.hlines(y = 0.95, xmin=0, xmax=len(sorted_importances), color = 'r', linestyles = 'dashed
    ')
# Format x ticks and labels
plt.xticks(x_values, sorted_features, rotation = 'vertical')
# Axis labels and title
plt.xlabel('Variable'); plt.ylabel('Cumulative Importance'); plt.title('Cumulative
    Importances for Random Forests classifier')
plt.savefig("importance_variables_RF1.pdf", bbox_inches='tight')


# In[33]:


print('Number of features for 95% importance:', np.where(cumulative_importances > 0.95)
    [0][0] + 1)


# # MLP

# ## Create features MLP

# In[34]:


random.seed(42)


# In[35]:


scaler = StandardScaler()
X_train_mlp = scaler.fit_transform(X_train)
X_test_mlp = scaler.transform(X_test)
y_train_mlp = np.array(y_train).reshape(-1,1)


# ## MLP

# In[36]:
```

```
def build_model(n_hidden = 0, n_neurons = 30, learning_rate = 3e-2, input_shape = [p-1]) :
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape = input_shape))
    for layer in range(n_hidden) :
        model.add(keras.layers.Dense(n_neurons))
    model.add(keras.layers.Dense(1, activation = "tanh"))
    optimizer = keras.optimizers.SGD(lr = learning_rate)
    model.compile(loss = "binary_crossentropy", optimizer = optimizer, metrics = "accuracy")
    return model


# In[37]:


keras_clf = keras.wrappers.scikit_learn.KerasClassifier(build_model)


# In[38]:


tf.random.set_seed(42)

keras_clf.fit(X_train_mlp[50:],
              y_train_mlp[50:],
              validation_data=(X_train_mlp[:50], y_train_mlp[:50]),
              epochs=1000,
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])


# In[39]:


y_pred_mlp = keras_clf.predict(X_test_mlp)
y_pred_train_mlp = keras_clf.predict(X_train_mlp)

train_acc_mlp = accuracy_score(y_pred_train_mlp, y_train_mlp)
test_acc_mlp = accuracy_score(y_pred_mlp, y_test)
train_acc_mlp, test_acc_mlp


# ## MLP with parameters tuning

# In[40]:


def auroc(y_true, y_pred):
    return tf.py_func(roc_auc_score, (y_true, y_pred), tf.double)


# In[41]:


def relu(x):
    return tf.keras.activations.relu(x)
x = np.linspace(-3,3,100)
ax = plt.gca()  # or any other way to get an axis object
ax.plot(x, relu(x))
ax.set_xlabel("x")
ax.set_ylabel("ReLU(x)")
ax.set_title("ReLU activation function")
plt.savefig("ReLU.pdf", bbox_inches='tight')


# In[42]:


def build_model_tuned(n_hidden = 1, n_neurons = 50, dropout_rate = 0.3, l2_coeff = 1e-2,
    input_shape = [p-1]) :
```

```
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape = input_shape))
    for layer in range(n_hidden) :
        model.add(keras.layers.Dense(n_neurons, kernel_regularizer = tf.keras.regularizers.
            l2(l2_coeff), activation = 'relu'))
        model.add(keras.layers.Dropout(rate = dropout_rate))
    model.add(keras.layers.Dense(1, activation = "sigmoid"))
    return model


# In[43]:


tf.random.set_seed(42)
mlp_tuned = build_model_tuned()
mlp_tuned.compile(loss = "BinaryCrossentropy", optimizer = tf.keras.optimizers.SGD(learning_
    rate = 0.005), metrics = "accuracy")


# In[44]:


X_valid_mlp, y_valid_mlp = X_train_mlp[:50,], y_train_mlp[:50,]
X_train_mlp, y_train_mlp = X_train_mlp[50:,], y_train_mlp[50:,]
X_train_mlp.shape, X_valid_mlp.shape


# In[45]:


history = mlp_tuned.fit(X_train_mlp,
              y_train_mlp,
              validation_data=(X_valid_mlp, y_valid_mlp),
              epochs=5000,
              callbacks=[keras.callbacks.EarlyStopping(patience=50, monitor = 'val_loss')])


# In[46]:


plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.savefig("loss_MLP.pdf", bbox_inches='tight')


# In[47]:


y_pred_train_mlp_final = mlp_tuned.predict_classes(X_train_mlp)
train_acc_mlp = accuracy_score(y_pred_train_mlp_final, y_train_mlp)
train_acc_mlp.round(4)


# In[48]:


y_pred_valid_mlp_final = mlp_tuned.predict_classes(X_valid_mlp)
valid_acc_mlp = accuracy_score(y_pred_valid_mlp_final, y_valid_mlp)
valid_acc_mlp


# In[49]:


y_pred_mlp_final = mlp_tuned.predict_classes(X_test_mlp)
test_acc_mlp = accuracy_score(y_pred_mlp_final, y_test)
```

```
test_acc_mlp.round(4)


# # Comparison Random Forest and MLP

# In[50]:


y_pred_rf = y_pred_rf_final
y_pred_mlp = y_pred_mlp_final.reshape(547,)


# In[51]:


confusion_matrix(y_test, y_pred_rf)


# In[52]:


confusion_matrix(y_test, y_pred_mlp)


# # McNemars test

# In[53]:


rf_cor = (y_pred_rf == y_test)
rf_err = (y_pred_rf != y_test)
mlp_cor = (y_pred_mlp == y_test)
mlp_err = (y_pred_mlp != y_test)


# In[54]:


rf_cor_mlp_cor = np.logical_and(rf_cor,mlp_cor).sum()
rf_cor_mlp_err = np.logical_and(rf_cor,mlp_err).sum()
rf_err_mlp_cor = np.logical_and(rf_err,mlp_cor).sum()
rf_err_mlp_err = np.logical_and(rf_err,mlp_err).sum()


# In[55]:


table = [[rf_cor_mlp_cor, rf_cor_mlp_err],
         [rf_err_mlp_cor, rf_err_mlp_err]]


# In[56]:


print(np.array(table))


# ### Test with continuity correction

# In[57]:


mcnemar_cont = mcnemar(table, exact=False, correction=True)


# In[58]:


print('statistic=%.2f, p-value=%.7f' % (mcnemar_cont.statistic, mcnemar_cont.pvalue))
# interpret the p-value
```

```
alpha = 0.05
if mcnemar_cont.pvalue > alpha:
    print('Same proportions of errors (fail to reject H0)')
else:
    print('Different proportions of errors (reject H0)')


# ### Exact test

# In[59]:


mcnemar_exact = mcnemar(table, exact=True)


# In[60]:


print('statistic=%.3f, p-value=%.3f' % (mcnemar_exact.statistic, mcnemar_exact.pvalue))
# interpret the p-value
alpha = 0.05
if mcnemar_exact.pvalue > alpha:
    print('Same proportions of errors (fail to reject H0)')
else:
    print('Different proportions of errors (reject H0)')


# ### ROC

# In[61]:


y_pred_mlp_roc = mlp_tuned.predict(X_test_mlp)
fpr_keras, tpr_keras, thresholds_keras = roc_curve(y_test, y_pred_mlp_roc)


# In[62]:


from sklearn.metrics import auc
auc_keras = auc(fpr_keras, tpr_keras)


# In[63]:


plt.figure(1)
plot_roc_curve(rf_clf_final, X_test, y_test)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_keras, tpr_keras, label='MLP (AUC = {:.3f})'.format(auc_keras))
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.savefig("roc_curve.pdf", bbox_inches='tight')


# # Reject option

# ## Random forest

# In[64]:


y_proba_rf = rf_clf_final.predict_proba(X_test)
A_rf = np.where(y_proba_rf >= 0.6)[0]


# In[65]:
```

```
print('number of test observations :', X_test.shape[0])
print('number of test observations accepted :', A_rf.shape[0])
print('number of test observations rejected :', X_test.shape[0] - A_rf.shape[0])


# In[66]:


X_test_A_rf = X_test[A_rf,:]
y_test_A_rf = y_test.values.reshape(-1,1)[A_rf,:]
X_test_A_rf.shape, y_test_A_rf.shape


# In[67]:


y_pred_rf_A = rf_clf_final.predict(X_test_A_rf)


# In[68]:


accuracy_score(y_pred_rf_A, y_test_A_rf).round(4)


# In[69]:


confusion_matrix(y_test_A_rf, y_pred_rf_A), confusion_matrix(y_test, y_pred_rf)


# ## MLP

# In[70]:


y_proba_mlp = mlp_tuned.predict(X_test_mlp)
A_mlp = np.where(abs(y_proba_mlp - 0.5) >= 0.1)[0]


# In[71]:


print('number of test observations :', X_test_mlp.shape[0])
print('number of test observations accepted :', A_mlp.shape[0])
print('number of test observations rejected :', X_test_mlp.shape[0] - A_mlp.shape[0])


# In[72]:


X_test_A_mlp = X_test_mlp[A_mlp,:]
y_test_A_mlp = y_test.values.reshape(-1,1)[A_mlp,:]
X_test_A_mlp.shape, y_test_A_mlp.shape


# In[73]:


y_pred_mlp_A = mlp_tuned.predict_classes(X_test_A_mlp)


# In[74]:


accuracy_score(y_pred_mlp_A, y_test_A_mlp)


# In[75]:
```

```
confusion_matrix(y_test_A_mlp, y_pred_mlp_A), confusion_matrix(y_test, y_pred_mlp)
```

D R code question 2

```
########## Machine Learning - Coursework 2 ##########

setwd("C:/Users/robin/Dropbox/Applications/Overleaf/Coursework_2_ML")
width=16
height=10

library(caret)
library(ggplot2)
library(tidyr)
library(matrixStats)
library(purrr)
library(NbClust)
library(graphics)
library(dendextend)
library(GGally)
library(pracma)
library(fossil)
library(gridExtra)
library(amap)
library(cluster)

set.seed(42)

### load data ###

data = read.csv("CID1945214.csv")

head(data)
summary(data)

n = dim(data)[1]
p = dim(data)[2]

head(data)
summary(data)

### NA ###

sum(sapply(1:n, function(i) sum(is.na(data[i,]))))
lapply(1:p, function(j) data[is.na(data[,j]),j] <<- mean(data[,j], na.rm = TRUE))
summary(data)


####### Hierarchical clustering #########

features = data[,-1]
data_t = t(data[,-1])
data_t_scale = scale(data_t)


diss_matrix_spear = 1 - abs(cor(features, method = "pearson"))
diss_matrix_spear = as.dist(diss_matrix_spear)

# plot the resulting dendrograms

pdf("dends.pdf", width = width, height = height/2)

par(mfrow = c(2,3))

dend_sin_cor = hclust(diss_matrix_spear, method="single")
dend_avg_cor = hclust(diss_matrix_spear, method="average")
dend_comp_cor = hclust(diss_matrix_spear, method="complete")

plot(dend_sin_cor,
     main = "Spearman correlation and single linkage",
      cex.main = 1.5,
      xlab="Features")

plot(dend_avg_cor,
      main = "Spearman correlation and average linkage",
      cex.main = 1.5,
      xlab="Features")

plot(dend_comp_cor,
      main = "Spearman correlation and complete linkage",
      cex.main = 1.5,
      xlab="Features")

dend_sin_euc = hclust(dist(data_t_scale, method = "euclidean"), method="single")
dend_avg_euc = hclust(dist(data_t_scale, method = "euclidean"), method="average")
dend_comp_euc = hclust(dist(data_t_scale, method = "euclidean"), method="complete")



plot(dend_sin_euc,
      main = "Euclidean distance and single linkage",
      cex.main = 1.5,
      xlab="Features")

plot(dend_avg_euc,
      main = "Euclidean distance and average linkage",
      cex.main = 1.5,
      xlab="Features")

plot(dend_comp_euc,
      main = "Euclidean distance and complete linkage",
      cex.main = 1.5,
      xlab="Features")

dev.off()


##### Seek appropriate number of clusters #####

nbc_comp <- NbClust(data_t_scale,
                     min.nc=2,
                     max.nc=27,
                     method="complete",
                     diss = diss_matrix_spear,
                     distance = NULL,
                     index="silhouette")$All.index

nbc_sin <- NbClust(data_t_scale,
                    min.nc=2,
                    max.nc=27,
                    method="single",
                    diss = diss_matrix_spear,
                    distance = NULL,
                    index="silhouette")$All.index


nbc_avg <- NbClust(data_t_scale,
                    min.nc=2,
                    max.nc=27,
                    method="average",
                    diss = diss_matrix_spear,
                    distance = NULL,
                    index="silhouette")$All.index


sil_sin_cor <- sapply(2:27, function(i) {summary(silhouette(cutree(dend_sin_cor, k = i),
                                                  diss_matrix_spear, full = FALSE))
                                                    $avg.width})

sil_comp_cor <- sapply(2:27, function(i) {summary(silhouette(cutree(dend_comp_cor, k = i),
                                                            diss_matrix_spear, full = FALSE)
                                                               )$avg.width})

sil_avg_cor <- sapply(2:27, function(i) {summary(silhouette(cutree(dend_avg_cor, k = i),
                                                             diss_matrix_spear, full = FALSE
                                                                ))$avg.width})

pdf(file = "silhouette_hier.pdf")

data.frame(n_clusters=2:27, Complete = sil_comp_cor,
           Single = sil_sin_cor, Average = sil_avg_cor) %>%
  pivot_longer(-n_clusters, names_to="method", values_to="index_value") %>%
  ggplot(aes(x=n_clusters, y=index_value, colour=method)) + geom_line() + geom_point() +
  labs(x = "Number_of_clusters_K", y = "Silhouette_measure")

dev.off()

# Average + Absolute correlation

pdf(file = "dendrogram.pdf")

par(mfrow = c(1,1))

dend = hclust(diss_matrix_spear, method="average")

plot(dend,
     main="Spearman_Correlation_distance_and_average_linkage",
     xlab="Features")
rect.dendrogram(as.dendrogram(dend),
                k = 5,
                border = c(2:9, "navyblue", "orange"),
                lwd = 2,
                lower_rect = -0.2)
dev.off()

########## k-mean ##########

library("LICORS")

data_scale = scale(data)
data_scale = data_scale[,-1]

pdf("silhouette_k_mean.pdf", width = width/2, height = height/2)
set.seed(42)
nbc_scale <- NbClust(data_scale, min.nc=2, max.nc=15, method="kmeans", index="silhouette")
data.frame(n_clusters=2:15, nbc_scale$All.index) %>%
  pivot_longer(-n_clusters, names_to="method", values_to="index_value") %>%
  ggplot(aes(x=n_clusters, y=index_value), col="red") + geom_line() + geom_point() +
  labs(x = "Number_of_clusters_K", y = "Silhouette_measure")
dev.off()

set.seed(42)
km_rep = replicate(10000, {km <- kmeans(data_scale, centers = 2, iter.max = 10)
c(km$tot.withinss, dist(km$centers))})

with_rep = km_rep[1,]
dist_centroids = km_rep[2,]

summary(with_rep)
summary(dist_centroids)

best_tot_withinss = Inf
n_iter = 10000
i = 0
for (i in 1:n_iter){
  km = kmeans(data_scale, centers = 2, iter.max = 10)
  if (km$tot.withinss < best_tot_withinss){
    best_dist_centroids = km$tot.withinss
    km_final = km
  }
```

```
}

km_final$tot.withinss
dist(km_final$centers)

pca = prcomp(data_scale, retx = TRUE, rank. = 4)

df <- as.data.frame(cbind(pca$x, km_final$cluster))
colnames(df)[[5]] <- "cluster_label"
df$cluster_label <- as.factor(df$cluster_label)

ggpairs(df, columns=1:4, aes(colour=cluster_label), progress=FALSE)


##### Rand index #####

round(rand.index(km_final$cluster, data$z), digits = 4)

df <- as.data.frame(cbind(pca$x, km_final$cluster))
colnames(df)[[5]] <- "cluster_label"
df$cluster_label <- as.factor(- df$cluster_label + 2)
df$z = as.factor(data$z)

pdf("pca_z.pdf", width = width/2, height = height/2)

g1 = ggplot(df, aes(x = PC1, y = PC2, colour=cluster_label), progress=FALSE) +
  geom_point(alpha = 0.4) + theme(legend.position="bottom")
g2 = ggplot(df, aes(x = PC1, y = PC2, colour=z), progress=FALSE) +
  geom_point(alpha = 0.4) + theme(legend.position="bottom")
grid.arrange(g1, g2, ncol = 2)

dev.off()




##### k-mean ++ #####

library("flexclust")

kpp = kcca(data_scale, k = 2, family = kccaFamily("kmeans"), control=list(initcent="kmeanspp
    "),
            simple = FALSE, save.data = TRUE)

info(kpp, which = "distsum")  # sum withinss
info(kpp, which = "av_dist")
cluster_label_pp = predict(kpp)
predict(kpp)

head(ddata_kpp)

sum(cluster_label_pp != km_final$cluster)

# final km 2D

df_pp <- as.data.frame(cbind(pca$x, cluster_label_pp))
colnames(df_pp)[[5]] <- "cluster_label"
df_pp$cluster_label <- as.factor(df_pp$cluster_label)

ggplot(df_pp, aes(x = PC1, y = PC2, colour=cluster_label_pp), progress=FALSE) +
  geom_point()
```

E Python code question 3

```
#!/usr/bin/env python
# coding: utf-8

# # Coursework 2 - Machine Learning - Question 3

# Code realised by Robin Mathelier for the final coursework of Machine Learning module in
    MSc Statistics Imperial College London

# In[1]:


import os

import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
get_ipython().run_line_magic('matplotlib', 'inline')

import pandas as pd

import random

import numpy as np

from math import *

import scipy.optimize

from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import ConstantKernel, RBF, Matern, WhiteKernel,
    ExpSineSquared, RationalQuadratic
from sklearn.metrics import make_scorer


# In[2]:


os.chdir(r"C:\Users\robin\Dropbox\Applications\Overleaf\Coursework_2_ML")
os.getcwd()


# ## Import data

# In[3]:


data_train = pd.read_csv("train_dataQ3.csv")
data_test = pd.read_csv("test_dataQ3.csv")


# In[4]:


random.seed(1945214)

s = [random.randint(0, 2000) for i in range(200)]
data_train = data_train.iloc[s,:]

X_train = np.array(data_train["x"]).reshape(-1,1)
y_train = np.array(data_train["y"]).reshape(-1,1)

X_test = np.array(data_test["x"]).reshape(-1,1)
y_test = np.array(data_test["y"]).reshape(-1,1)
```

```
n = X_train.shape[0]
p = X_train.shape[1]

X_train.shape, y_train.shape, X_test.shape, y_test.shape


# In[5]:


fig = plt.figure()
plt.scatter(X_train, y_train)
plt.xlim(-6,6)
plt.xlabel("x", fontsize = 12)
plt.ylabel("y", fontsize = 12)
plt.savefig('scatter.pdf', bbox_inches='tight')


# ## Kernel Ridge Regression

# In[6]:


krr = KernelRidge(kernel = "poly", degree = 3)
krr.fit(X_train, y_train)
krr.get_params()


# In[7]:


y_pred_train_krr = krr.predict(X_train)
y_pred_test_krr = krr.predict(X_test)


# In[8]:


x = np.linspace(-6,6,100).reshape(-1,1)
y = krr.predict(x)
plt.scatter(X_train, y_train)
plt.plot(x,y,color='red')
plt.xlim(-5.2, 5.2)
plt.ylim(-0.6, 0.6)
plt.title("Kernel Ridge regression with p = 3 and alpha = 1", fontsize = 15)
plt.xlabel("x", fontsize = 12)
plt.ylabel("y", fontsize = 12)
plt.savefig('krr_3_1.pdf', bbox_inches='tight')


# In[9]:


mean_squared_error(y_train, y_pred_train_krr).round(6), mean_squared_error(y_test, y_pred_
    test_krr).round(6)


# ## Kernel Ridge Regression with parameters tuning (degree tuned)

# In[10]:


grid_krr = {"alpha" : np.linspace(0.001, 10, num = 1000),
            "degree" : range(30)}


# In[11]:


krr_ = KernelRidge(kernel = "poly")
```

```
grid_search = RandomizedSearchCV(estimator = krr_,
                                 param_distributions = grid_krr,
                                 n_iter = 10000,
                                 cv = 5,
                                 random_state=42,
                                 n_jobs = -1,
                                scoring = "neg_mean_squared_error")
grid_search.fit(X_train, y_train)


# In[12]:


grid_search.best_params_


# In[13]:


krr_final = grid_search.best_estimator_


# In[14]:


y_pred_train_krr_final = krr_final.predict(X_train)
y_pred_test_krr_final = krr_final.predict(X_test)

mean_squared_error(y_train, y_pred_train_krr_final).round(7), mean_squared_error(y_test, y_
    pred_test_krr_final).round(4)


# In[15]:


x = np.linspace(-5.5, 5.5, 100).reshape(-1,1)
y = krr_final.predict(x)
plt.scatter(X_train, y_train)
plt.plot(x,y,color='red')
plt.xlim(-5.2, 5.2)
plt.ylim(-0.6, 0.6)
plt.title("Kernel Ridge regression with p = 3 and alpha = 1", fontsize = 15)
plt.xlabel("x", fontsize = 12)
plt.ylabel("y", fontsize = 12)
plt.savefig('krr_3_1.pdf', bbox_inches='tight')


# In[16]:


X_ = np.linspace(-6, 6, 100)
fig, axs = plt.subplots(1,2, figsize=(20, 6))

x = np.linspace(-5, 5, 100).reshape(-1,1)
y = krr.predict(x)
l = axs[0].scatter(X_train, y_train)
axs[0].plot(x,y,color='red')
axs[0].set_xlim(-5.2, 5.2)
axs[0].set_ylim(-0.6, 0.6)
axs[0].set_title("Kernel Ridge regression with p = 3 and alpha = 1", fontsize = 17)
axs[0].set_xlabel("x", fontsize = 14)
axs[0].set_ylabel("y", fontsize = 14)
axs[0].legend([l],["Train data"], loc = "lower center", prop={'size':18})

x = np.linspace(-5, 5, 100).reshape(-1,1)
y = krr_final.predict(x)
l1 = axs[1].scatter(X_train, y_train)
axs[1].plot(x,y,color='red')
axs[1].set_xlim(-5.2, 5.2)
axs[1].set_ylim(-0.6, 0.6)
```

```
axs[1].set_title("Kernel Ridge regression with p = 11 and alpha = 1.4623", fontsize = 17)
axs[1].set_xlabel("x", fontsize = 15)
axs[1].set_ylabel("y", fontsize = 15)
axs[1].legend([l],["Train data"], loc = "lower center", prop={'size':18})

plt.savefig('train_Ridge.pdf', bbox_inches='tight')


# In[17]:


X_ = np.linspace(-6, 6, 100)
fig, axs = plt.subplots(1,2, figsize=(20, 6))

x = np.linspace(-7, 7, 100).reshape(-1,1)
y = krr.predict(x)
l1 = axs[0].scatter(X_train, y_train)
l2 = axs[0].scatter(X_test, y_test, color = 'orange')
axs[0].plot(x,y,color='red')
axs[0].set_xlim(-6.5, 6.5)
axs[0].set_ylim(-0.6, 0.6)
axs[0].set_title("Kernel Ridge regression with p = 3 and alpha = 1", fontsize = 17)
axs[0].set_xlabel("x", fontsize = 15)
axs[0].set_ylabel("y", fontsize = 15)
axs[0].legend([l1, l2],["Train data", "Test data"], loc = "lower center", prop={'size':18})

x = np.linspace(-6, 6, 100).reshape(-1,1)
y = krr_final.predict(x)
l1 = axs[1].scatter(X_train, y_train)
l2 = axs[1].scatter(X_test, y_test, color = 'orange')
axs[1].plot(x,y,color='red')
axs[1].set_xlim(-6.5, 6.5)
axs[1].set_ylim(-2, 5)
axs[1].set_title("Kernel Ridge regression with p = 11 and alpha = 1.4623", fontsize = 17)
axs[1].set_xlabel("x", fontsize = 15)
axs[1].set_ylabel("y", fontsize = 15)
axs[1].legend([l1, l2],["Train data", "Test data"], loc = "lower center", prop={'size':18})

plt.savefig('test_Ridge.pdf', bbox_inches='tight')


# # Gaussian Process

# In[18]:


gpc_rbf = GaussianProcessRegressor(kernel= 1 * RBF() + WhiteKernel(),
                                random_state=42, n_restarts_optimizer = 50).fit(X_train, y_
                                    train)

gpc_mat = GaussianProcessRegressor(kernel= 1 * Matern() +  WhiteKernel(),
                                random_state=42, n_restarts_optimizer = 50).fit(X_train, y_
                                    train)

gpc_quad = GaussianProcessRegressor(kernel = 1 * RationalQuadratic() + WhiteKernel(),
                                random_state=42, n_restarts_optimizer = 50).fit(X_train, y_
                                    train)


# In[19]:


X_ = np.linspace(-6.5, 6.5, 100)
fig, axs = plt.subplots(3,1, figsize=(20, 20))

l1 = axs[0].scatter(X_train, y_train)
l2 = axs[0].scatter(X_test, y_test, color = 'orange')
y_mean_rbf, y_std_rbf = gpc_rbf.predict(X_[:, np.newaxis], return_std=True)
y_mean_rbf = y_mean_rbf.reshape(-1,)
axs[0].fill_between(X_, y_mean_rbf - y_std_rbf, y_mean_rbf + y_std_rbf,
                     alpha=0.1, color='k')
axs[0].plot(X_, y_mean_rbf, 'k', lw=3, zorder=9, color = 'red')
axs[0].set_xlabel("x", fontsize = 15)
axs[0].set_ylabel("y", fontsize = 15)
axs[0].set_title('RBF Kernel', fontsize = 17)
axs[0].set_xlim(-6.5,6.5)
axs[0].set_ylim(-0.7,0.7)
axs[0].legend([l1, l2],["Train data", "Test data"], loc = "lower center", prop={'size':15})

l1 = axs[1].scatter(X_train, y_train)
l2 = axs[1].scatter(X_test, y_test, color = 'orange')
y_mean_mat, y_std_mat = gpc_mat.predict(X_[:, np.newaxis], return_std=True)
y_mean_mat = y_mean_mat.reshape(-1,)
axs[1].fill_between(X_, y_mean_mat - y_std_mat, y_mean_mat + y_std_mat,
                     alpha=0.1, color='k')
axs[1].plot(X_, y_mean_mat, 'k', lw=3, zorder=9, color = 'red')
axs[1].set_xlabel("x", fontsize = 15)
axs[1].set_ylabel("y", fontsize = 15)
axs[1].set_title('Matern Kernel', fontsize = 17)
axs[1].set_xlim(-6.5,6.5)
axs[1].set_ylim(-0.7,0.7)
axs[1].legend([l1, l2],["Train data", "Test data"], loc = "lower center", prop={'size':15})

l1 = axs[2].scatter(X_train, y_train)
l2 = axs[2].scatter(X_test, y_test, color = 'orange')
y_mean_quad, y_std_quad = gpc_quad.predict(X_[:, np.newaxis], return_std=True)
y_mean_quad = y_mean_quad.reshape(-1,)
axs[2].fill_between(X_, y_mean_quad - y_std_quad, y_mean_quad + y_std_quad,
                     alpha=0.1, color='k')
axs[2].plot(X_, y_mean_quad, 'k', lw=3, zorder=9, color = 'red')
axs[2].set_xlabel("x", fontsize = 15)
axs[2].set_ylabel("y", fontsize = 15)
axs[2].set_title('Rational Quadratic Kernel', fontsize = 17)
axs[2].set_xlim(-6.5,6.5)
axs[2].set_ylim(-0.7,0.7)
axs[2].legend([l1, l2],["Train data", "Test data"], loc = "lower center", prop={'size':15})


plt.savefig("Kernels_fitted.pdf", bbox_inches='tight')


# In[20]:


y_pred_train_gfb = gpc_rbf.predict(X_train)
y_pred_test_gbf = gpc_rbf.predict(X_test)

mean_squared_error(y_train, y_pred_train_gfb).round(7), mean_squared_error(y_test, y_pred_
    test_gbf).round(7)


# In[21]:


y_pred_train_mat = gpc_mat.predict(X_train)
y_pred_test_mat = gpc_mat.predict(X_test)

mean_squared_error(y_train, y_pred_train_mat).round(7), mean_squared_error(y_test, y_pred_
    test_mat).round(7)


# In[22]:


y_pred_train_quad = gpc_quad.predict(X_train)
y_pred_test_quad = gpc_quad.predict(X_test)

mean_squared_error(y_train, y_pred_train_quad).round(7), mean_squared_error(y_test, y_pred_
    test_quad).round(7)
```

```
# In[23]:


gpc_rbf.kernel_.get_params()


# In[24]:


gpc_mat.kernel_.get_params()


# In[25]:


gpc_quad.kernel_.get_params()


# # Gaussian Process - Model Selection

# ### Cross Validation

# In[26]:


mse_cv = cross_val_score(gpc_rbf, X_train, y_train, cv=5, scoring = 'neg_mean_squared_error'
    )
-np.mean(mse_cv).round(7)


# In[27]:


mse_cv = cross_val_score(gpc_mat, X_train, y_train, cv=5, scoring = 'neg_mean_squared_error'
    )
-np.mean(mse_cv).round(7)


# In[28]:


mse_cv = cross_val_score(gpc_quad, X_train, y_train, cv=5, scoring = 'neg_mean_squared_error
    ')
-np.mean(mse_cv).round(7)


# ### BIC - AIC

# In[29]:


ndim_rbf = gpc_rbf.kernel_.n_dims
ndim_mat = gpc_mat.kernel_.n_dims
ndim_quad = gpc_quad.kernel_.n_dims

ndim_rbf, ndim_mat, ndim_quad


# In[30]:


AIC_rbf = 2 * ndim_rbf - 2 * gpc_rbf.log_marginal_likelihood()
AIC_mat = 2 * ndim_mat - 2 * gpc_mat.log_marginal_likelihood()
AIC_quad = 2 * ndim_quad - 2 * gpc_quad.log_marginal_likelihood()

AIC_rbf.round(4), AIC_mat.round(4), AIC_quad.round(4)


# In[31]:
```

```
BIC_rbf = log(n) * ndim_rbf - 2 * gpc_rbf.log_marginal_likelihood()
BIC_mat = log(n) * ndim_mat - 2 * gpc_mat.log_marginal_likelihood()
BIC_quad = log(n) * ndim_quad - 2 * gpc_quad.log_marginal_likelihood()

BIC_rbf.round(4), BIC_mat.round(4), BIC_quad.round(4)


# ### Marginal likelihood values

# In[32]:


gpc_rbf.log_marginal_likelihood().round(4)


# In[33]:


gpc_mat.log_marginal_likelihood().round(4)


# In[34]:


gpc_quad.log_marginal_likelihood().round(4)


# # Final Gaussian Process

# In[35]:


X_ = np.linspace(-6.5, 6.5, 100)
fig, axs = plt.subplots(1, 1, figsize=(20, 6))

axs.scatter(X_train, y_train)
axs.scatter(X_test, y_test, color = 'orange')
y_mean, y_std = gpc_mat.predict(X_[:, np.newaxis], return_std = True)
y_mean = y_mean.reshape(-1,)
axs.fill_between(X_, y_mean - y_std, y_mean + y_std,
                 alpha=0.1, color='k')
axs.plot(X_, y_mean, 'k', lw=3, zorder=9, color = 'red')
axs.set_xlabel("x", fontsize = 15)
axs.set_ylabel("y", fontsize = 15)
axs.set_title('', fontsize = 17)
axs.set_title('Mat rn_Kernel', fontsize = 17)
axs.set_xlim(-6.5, 6.5)
axs.set_ylim(-0.7, 0.7)

plt.savefig("GBF_MAT.pdf", bbox_inches='tight')
```