POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica

Tesina di Data Spaces

Data Spaces: 01RLPOV

Professor:

Prof. Francesco Vaccarino

Student:

Pietro Basci
266004

ACADEMIC YEAR 2019/2020

Index

1. Introduction

The objective of this work is to build a classier that, starting from a set of measurements, is able to detect if a patient is affected by the Chronic Kidney Disease. Chronic kidney disease is a type of kidney disease in which there is gradual loss of kidney function over a period of months to years. Initially there are generally no symptoms; later, symptoms may include leg swelling, feeling tired, vomiting, loss of appetite, and confusion. Complications include an increased risk of heart disease, high blood pressure, bone disease and anemia.

In order to do that, we will start in analyzing the *Chronic_Kidney_Disease DataSet* available on *UCI Machine Learning Repository* and then, after knowing better the data, we will analyze the behavior of few classifiers in order to find the best performing on our data. All analysis will be done using Python and the complete code can be found in the Jupyter Notebook associated to this report.

2. Dataset

The dataset considered is the Chronic_Kidney_Disease DataSet available on UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease). It was created by L.Jerlin Rubini (Research Scholar), Alagappa University, India, in collaboration with Dr. P. Soundara Pandian (Senior Consultant Nephrologist), Apollo Hospitals, India.

The dataset was collected over a period of 2 month from Apollo Hospitals and consists of 400 instances and 25 attributes (11 numeric and 14 nominal) including the target class which can assume two values: Positive or Negative.

- *NUMERICAL ATTRIBUTES*

1. ***age*** - Age (numerical): age in years
2. ***bp*** - Blood Pressure (numerical): bp in mm/Hg
3. ***bgr*** - Blood Glucose Random (numerical): bgr in mgs/dl
4. ***bu*** - Blood Urea (numerical): bu in mgs/dl
5. ***sc*** - Serum Creatinine (numerical): sc in mgs/dl
6. ***sod*** - Sodium (numerical): sod in mEq/L
7. ***pot*** - Potassium (numerical): pot in mEq/L
8. ***hemo*** - Hemoglobin (numerical): hemo in gms
9. ***pcv*** - Packed Cell Volume (numerical)
10. ***wbcc*** - White Blood Cell Count(numerical): wbcc in cells/cumm
11. ***rbcc*** - Red Blood Cell Count(numerical): rbcc in millions/cmm

- *NOMINAL ATTRIBUTES*

1. ***sg*** - Specific Gravity (nominal): sg - (1.005, 1.010, 1.015, 1.020, 1.025)
2. ***al*** - Albumin (nominal): al - (0, 1, 2, 3, 4, 5)
3. ***su*** - Sugar (nominal): su - (0, 1, 2, 3, 4, 5)
4. ***rbc*** - Red Blood Cells (nominal): rbc - (normal, abnormal)
5. ***pc*** - Pus Cell (nominal): pc - (normal, abnormal)
6. ***pcc*** - Pus Cell clumps (nominal): pcc - (present, notpresent)
7. ***ba*** - Bacteria (nominal): ba - (present, notpresent)
8. ***htn*** - Hypertension (nominal): htn - (yes, no)
9. ***dm*** - Diabetes Mellitus (nominal): dm - (yes, no)
10. ***cad*** - Coronary Artery Disease (nominal): cad - (yes, no)
11. ***appet*** - Appetite (nominal): appet - (good, poor)
12. ***pe*** - Pedal Edema (nominal): pe - (yes, no)
13. ***ane*** - Anemia(nominal): ane - (yes, no)
14. ***class*** - Class (nominal): class - (ckd, notckd)

3. Exploratory Data Analysis

As first step we import some of the most popular library needed for the following analysis.

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

Figure 3.1. Python libraries imports

The original dataset was provided in *arff* format which is a kind of file format used by the *Weka* software. A typical characteristic of this file is that missing values are represented with the '?' character. So, it has been added to the list of symbols used by Pandas to recognize *NaN* values. After loading the data, some initial rows are showed to quickly test if data was correctly loaded.

```
# Make a list of missing value types
missing_values = ["n/a", "na", "--", "?", -1]

# Read the CSV file
df = pd.read_csv('Chronic_Kidney_Disease/chronic_kidney_disease.csv', na_values = missing_values)
df.head()
```

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc	rbcc	htn	dm	cad	appet	pe	ane	class
0	48.0	80.0	1.020	1.0	0.0	NaN	normal	notpresent	notpresent	121.0	...	44.0	7800.0	5.2	yes	yes	no	good	no	no	ckd
1	7.0	50.0	1.020	4.0	0.0	NaN	normal	notpresent	notpresent	NaN	...	38.0	6000.0	NaN	no	no	no	good	no	no	ckd
2	62.0	80.0	1.010	2.0	3.0	normal	normal	notpresent	notpresent	423.0	...	31.0	7500.0	NaN	no	yes	no	poor	no	yes	ckd
3	48.0	70.0	1.005	4.0	0.0	normal	abnormal	present	notpresent	117.0	...	32.0	6700.0	3.9	yes	no	no	poor	yes	yes	ckd
4	51.0	80.0	1.010	2.0	0.0	normal	normal	notpresent	notpresent	106.0	...	35.0	7300.0	4.6	no	no	no	good	no	no	ckd

Figure 3.2. Commands used to load dataset and a section of the dataset

Then we can retrieve some basic information of our dataset.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 25 columns):
age      391 non-null float64
bp       388 non-null float64
sg       353 non-null category
al       354 non-null category
su       351 non-null category
rbc      248 non-null category
pc       335 non-null category
pcc      396 non-null category
ba       396 non-null category
bgr      356 non-null float64
bu       381 non-null float64
sc       383 non-null float64
sod      313 non-null float64
pot      312 non-null float64
hemo     348 non-null float64
pcv      329 non-null float64
wbcc     294 non-null float64
rbcc     269 non-null float64
htn      398 non-null category
dm       398 non-null category
cad      398 non-null category
appet    399 non-null category
pe       399 non-null category
ane      399 non-null category
class    400 non-null category
dtypes: category(14), float64(11)
```

Figure 3.3. Dataset basic informations

From these information we can see that our dataset is composed by 400 entries and 25 columns of which 14 are Categorical and 11 are Numerical. At the same time, we can also notice the conspicuous presence of missing values. This aspect will be further analyzed in section 3.4.

3.1 Descriptive Statistics

As first analysis, the following command is used to obtain some interesting statistics for numerical features. These measures are particularly useful to understand the central tendency, dispersion and shape of our data.

The function returns for each feature:

- the count of the non-null values
- the mean
- the standard deviation
- the minimal value
- 1st quartile
- 2nd quartile (median)
- 3rd quartile
- the maximal value

```
df.describe()
```

	age	bp	bgr	bu	sc	sod	pot	hemo	pcv	wbcc	rbcc
count	391.000000	388.000000	356.000000	381.000000	383.000000	313.000000	312.000000	348.000000	329.000000	294.000000	269.000000
mean	51.483376	76.469072	148.036517	57.425722	3.072454	137.528754	4.627244	12.526437	38.884498	8406.122449	4.707435
std	17.169714	13.683637	79.281714	50.503006	5.741126	10.408752	3.193904	2.912587	8.990105	2944.474190	1.025323
min	2.000000	50.000000	22.000000	1.500000	0.400000	4.500000	2.500000	3.100000	9.000000	2200.000000	2.100000
25%	42.000000	70.000000	99.000000	27.000000	0.900000	135.000000	3.800000	10.300000	32.000000	6500.000000	3.900000
50%	55.000000	80.000000	121.000000	42.000000	1.300000	138.000000	4.400000	12.650000	40.000000	8000.000000	4.800000
75%	64.500000	80.000000	163.000000	66.000000	2.800000	142.000000	4.900000	15.000000	45.000000	9800.000000	5.400000
max	90.000000	180.000000	490.000000	391.000000	76.000000	163.000000	47.000000	17.800000	54.000000	26400.000000	8.000000

Figure 3.4. Numerical features statistics

Then, the same command is used for the categorical feature. In this case it returns:

- the count of the non-null values
- the number of distinct values for that attribute
- the most common value (mode)
- the most common value's frequency

```
df.describe(exclude=[np.number])
```

	sg	al	su	rbc	pc	pcc	ba	htn	dm	cad	appet	pe	ane	class
count	353.00	354.0	351.0	248	335	396	396	398	398	398	399	399	399	400
unique	5.00	6.0	6.0	2	2	2	2	2	2	2	2	2	2	2
top	1.02	0.0	0.0	normal	normal	notpresent	notpresent	no	no	no	good	no	no	ckd
freq	106.00	199.0	290.0	201	259	354	374	251	261	364	317	323	339	250

Figure 3.5. Categorical features statistics

Finally, the distribution of class has been analyzed in order to detect any unbalance among classes.

As we can see from Figure 3.6, there is no relevant unbalance in our data:

- 250 CKD samples (62,5%)
- 150 NOTCKD samples (37,5%)

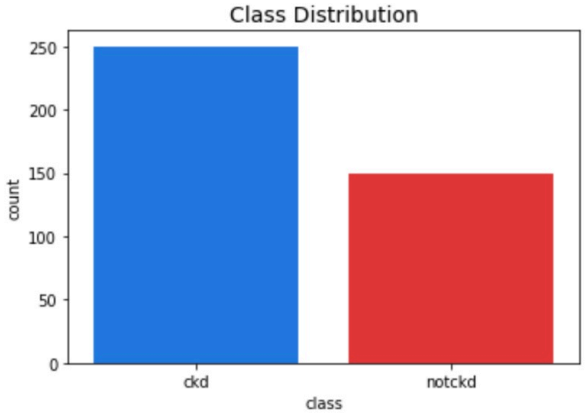


Figure 3.6. Original class distribution

3.2 Outliers Analysis

The next step is the analysis of outliers. The quicker way to locate outliers inside a distribution is by using boxplots. A boxplot is a kind of graphical representation of data distribution that gives a good indication of how the values in the data are spread out. It is based on five values: "minimum", first quartile (Q1), median (Q2), third quartile (Q3), and "maximum". The graph is composed by a box, whose height is equal to *Q3-Q1* (IQR), and two whiskers, whose length are given by "minimum" and "maximum" that are respectively given by the minimum value if it is greater than *Q1 - 1,5*IQR* or otherwise the smallest value greater than *Q1 - 1,5*IQR*, and similarly, the maximum value if it is lower than *Q3 + 1,5*IQR* or otherwise the greatest value lower than *Q3 + 1,5*IQR*.

Figure 3.7 shows that there are some data point which are very far from the other such as in the case of *Sodium* and *Potassium*. The normal ranges of Sodium and Potassium for instance are 135-145 mEq/L and 3.5-5.0 mEq/L respectively. But looking at boxplot we can see the presence of value near to 0 mEq/L in the case of Sodium and values greater than 40 mEq/L which are extremely far from typical values and so it is likely that these values are affected by errors.

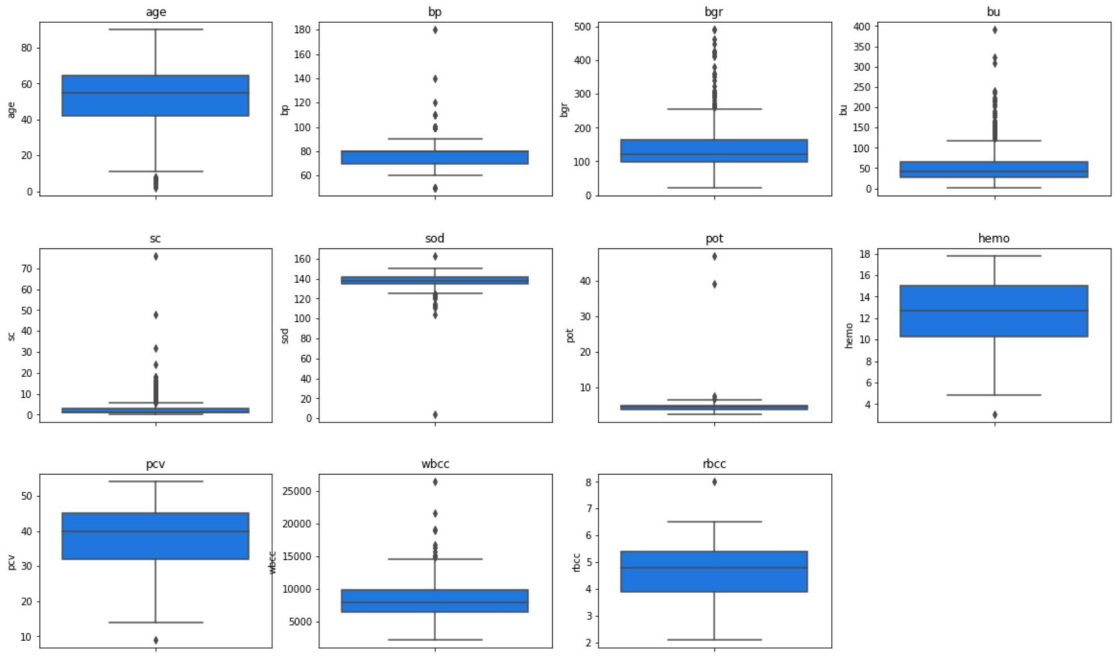


Figure 3.7. Graphical representation of numerical data distributions through boxplots

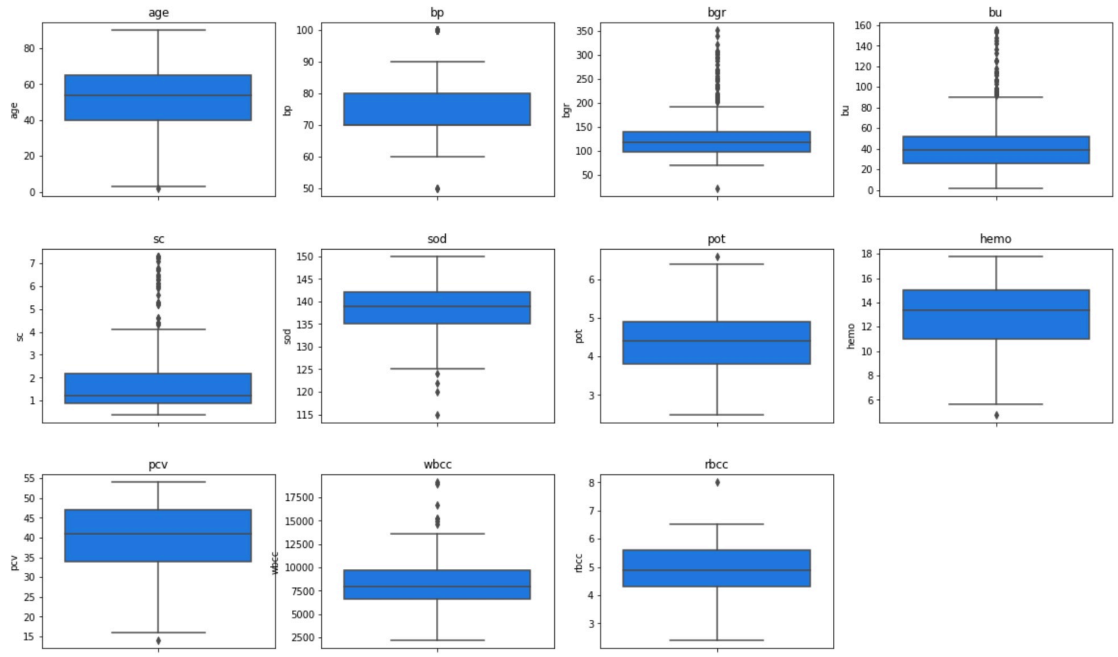


Figure 3.8. Graphical representation of numerical data distributions through boxplots after outliers removing

However there are also outliers that are closer to the rest of the data, are possible values and cannot be considered wrong. So, I decided to remove only 'extreme outliers', namely values that are more than 3xIQR below the 1st quartile or above the 3rd quartile:

$$x < Q1 - 3 \cdot IQR \quad \text{or} \quad x > Q3 + 3 \cdot IQR$$

After that extreme outliers have been removed, checking the status of our dataset, we see that 55 entries are lost, but we still have a lot of data. Then, we check the variations of boxplots (Figure 3.8).

After that, we take a look also to boxplots divided by class. In this way we can see how the values of these attributes changes for each class.

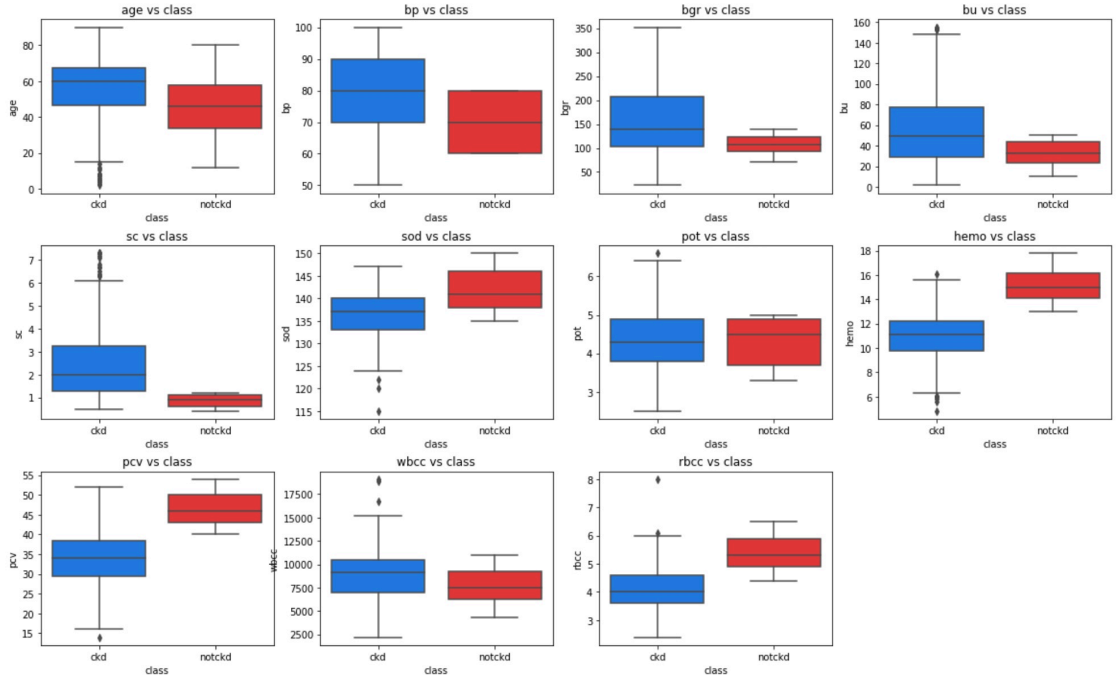


Figure 3.9. Graphical representation of numerical data distributions through boxplots divided by class

Looking at these plots, we can expect that there is a negative correlation among *class* and some attribute such as *hemo*, *pcv* and *rbcc*. Indeed, lower values of those attributes correspond to positive class (ckd, then referred as 1), while higher values correspond to negative class (notckd, then referred as 0). Similarly we can expect positive correlation among *class* and *sc*, while no correlation among *class* and *pot* as both distribution are almost aligned. Correlations will be further analyzed in section 3.5.

Finally, we check if there are wrong values in the categorical features.

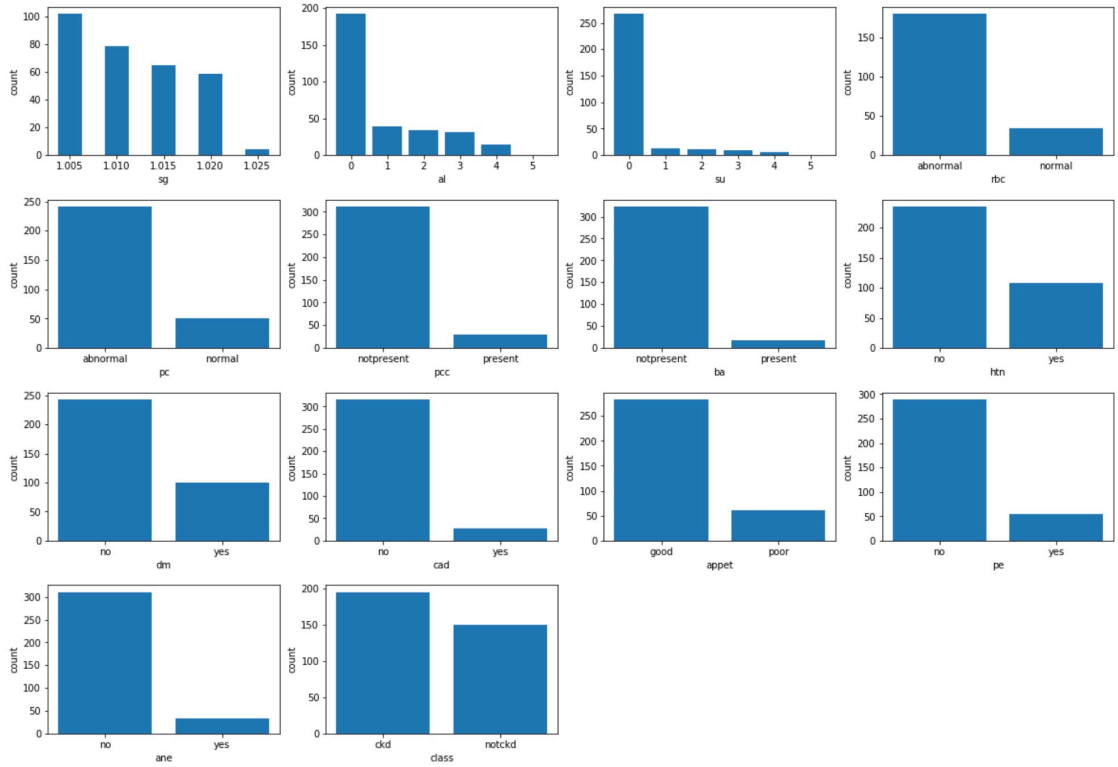


Figure 3.10. Graphical representation of categorical data distributions through histograms

From Figure 3.10, we see that all categorical features assume only values in the ranges specified in the dataset's description.

3.3 Categorical Data Encoding

Since we are going to use some techniques based on distances among data, we need to codify categorical data into numerical format. So, for some attributes such as *sg*, *al* and *su,* we can simply cast them to numerical as they already assume ordered numerical values. In the other cases, since predictors have only two possible values, we can simply replace each of them with a *dummy variable* that assumes two possible numerical values: 0 or 1.

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc	rbcc	htn	dm	cad	appet	pe	ane	class
0	48.0	80.0	1.020	1.0	0.0	NaN	0.0	0.0	0.0	121.0	...	44.0	7800.0	5.2	1.0	1.0	0.0	1.0	0.0	0.0	1
1	7.0	50.0	1.020	4.0	0.0	NaN	0.0	0.0	0.0	NaN	...	38.0	6000.0	NaN	0.0	0.0	0.0	1.0	0.0	0.0	1
4	51.0	80.0	1.010	2.0	0.0	0.0	0.0	0.0	0.0	106.0	...	35.0	7300.0	4.6	0.0	0.0	0.0	1.0	0.0	0.0	1
5	60.0	90.0	1.015	3.0	0.0	NaN	NaN	0.0	0.0	74.0	...	39.0	7800.0	4.4	1.0	1.0	0.0	1.0	1.0	0.0	1
8	52.0	100.0	1.015	3.0	0.0	0.0	1.0	1.0	0.0	138.0	...	33.0	9600.0	4.0	1.0	1.0	0.0	1.0	0.0	1.0	1

Figure 3.11. A section of the dataset after categorical data encoding

3.4 Missing Values Analysis

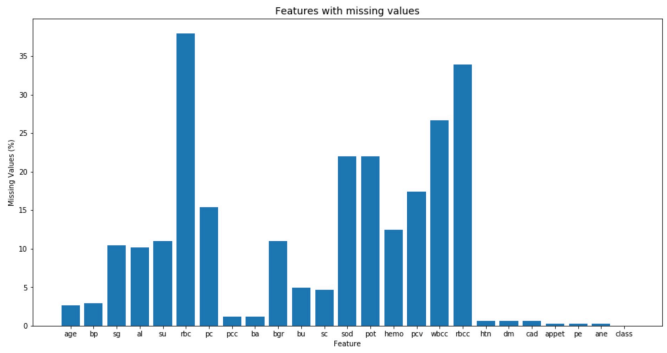
In this section we will address the problem of missing values. The dataset under consideration contains a significant number of missing values as confirmed by the command in Figure 3.12. In the next steps, we will analyze how those values are distributed inside the dataset. Specifically, we will consider the number of *NaN* values per feature and then the number of *NaN* values per instance.

```
# Total number of missing values
df.isnull().sum().sum()
```

```
864
```

Figure 3.12. The total number of missing values in the dataset

In the following graph, the percentage of missing values per feature is considered. As we can see, the features with the largest number of missing values are *rbc* and *rbcc* with a percentage of about 38% and 34% of data.

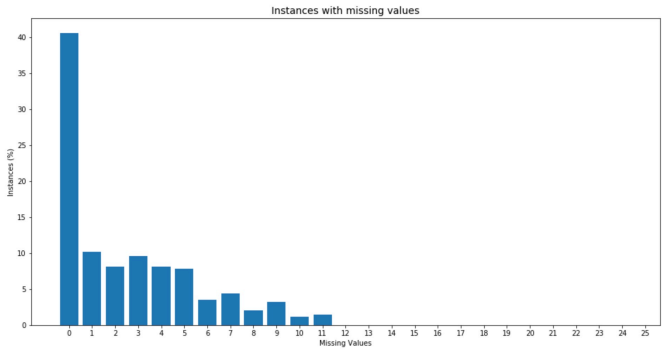



```
**Features with missing values**

age - missing values: 9  --  (2.61%)
bp - missing values: 10  --  (2.9%)
sg - missing values: 36  --  (10.43%)
al - missing values: 35  --  (10.14%)
su - missing values: 38  --  (11.01%)
rbc - missing values: 131  --  (37.97%)
pc - missing values: 53  --  (15.36%)
pcc - missing values: 4  --  (1.16%)
ba - missing values: 4  --  (1.16%)
bgr - missing values: 38  --  (11.01%)
bu - missing values: 17  --  (4.93%)
sc - missing values: 16  --  (4.64%)
sod - missing values: 76  --  (22.03%)
pot - missing values: 76  --  (22.03%)
hemo - missing values: 43  --  (12.46%)
pcv - missing values: 60  --  (17.39%)
wbcc - missing values: 92  --  (26.67%)
rbcc - missing values: 117  --  (33.91%)
htn - missing values: 2  --  (0.58%)
dm - missing values: 2  --  (0.58%)
cad - missing values: 2  --  (0.58%)
appet - missing values: 1  --  (0.29%)
pe - missing values: 1  --  (0.29%)
ane - missing values: 1  --  (0.29%)
class - missing values: 0  --  (0.0%)
```

Figure 3.13. Percentage of missing values per feature

Now, lets check the number of missing values per instance. From the graph below we can see that about 40% of instances do not contain *NaN* values while there are some instances with even 10-11 *NaN* values.



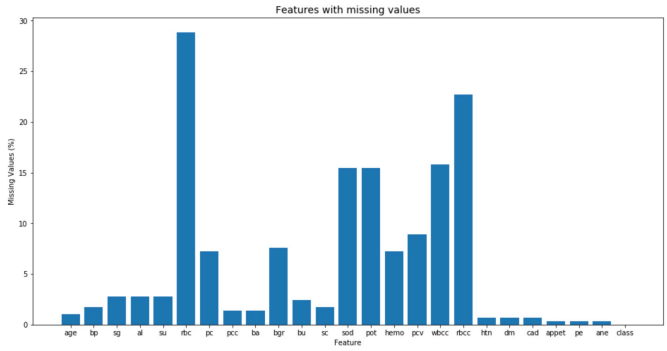

```
**Instances with missing values**

Instances with 0 missing values: 140  --  (40.58%)
Instances with 1 missing values: 35  --  (10.14%)
Instances with 2 missing values: 28  --  (8.12%)
Instances with 3 missing values: 33  --  (9.57%)
Instances with 4 missing values: 28  --  (8.12%)
Instances with 5 missing values: 27  --  (7.83%)
Instances with 6 missing values: 12  --  (3.48%)
Instances with 7 missing values: 15  --  (4.35%)
Instances with 8 missing values: 7  --  (2.03%)
Instances with 9 missing values: 11  --  (3.19%)
Instances with 10 missing values: 4  --  (1.16%)
Instances with 11 missing values: 5  --  (1.45%)
Instances with 12 missing values: 0  --  (0.0%)
Instances with 13 missing values: 0  --  (0.0%)
Instances with 14 missing values: 0  --  (0.0%)
Instances with 15 missing values: 0  --  (0.0%)
Instances with 16 missing values: 0  --  (0.0%)
Instances with 17 missing values: 0  --  (0.0%)
Instances with 18 missing values: 0  --  (0.0%)
Instances with 19 missing values: 0  --  (0.0%)
Instances with 20 missing values: 0  --  (0.0%)
Instances with 21 missing values: 0  --  (0.0%)
Instances with 22 missing values: 0  --  (0.0%)
Instances with 23 missing values: 0  --  (0.0%)
Instances with 24 missing values: 0  --  (0.0%)
```

Figure 3.14. Percentage of missing values per instance

From this analysis we discover that only 40% of instances are complete, while 60% of data has one missing value at least.

In order to not loose too much data and, at the same time, to not have the risk to introduce too much distortion in our data, a thread-off has been found: ignore instances with more than 5 *NaN* values and replace missing values in the other cases. To choose values that will be used for the replacing, for each feature some values will be sampled from the real distribution of the data. The precess will be explained more in detail later.



Features with missing values

age - missing values: 3 -- (1.03%)
bp - missing values: 5 -- (1.72%)
sg - missing values: 8 -- (2.75%)
al - missing values: 8 -- (2.75%)
su - missing values: 8 -- (2.75%)
rbc - missing values: 84 -- (28.87%)
pc - missing values: 21 -- (7.22%)
pcc - missing values: 4 -- (1.37%)
ba - missing values: 4 -- (1.37%)
bgr - missing values: 22 -- (7.56%)
bu - missing values: 7 -- (2.41%)
sc - missing values: 5 -- (1.72%)
sod - missing values: 45 -- (15.46%)
pot - missing values: 45 -- (15.46%)
hemo - missing values: 21 -- (7.22%)
pcv - missing values: 26 -- (8.93%)
wbcc - missing values: 46 -- (15.81%)
rbcc - missing values: 66 -- (22.68%)
htn - missing values: 2 -- (0.69%)
dm - missing values: 2 -- (0.69%)
cad - missing values: 2 -- (0.69%)
appet - missing values: 1 -- (0.34%)
pe - missing values: 1 -- (0.34%)
ane - missing values: 1 -- (0.34%)
class - missing values: 0 -- (0.0%)

Figure 3.15. Percentage of missing values per feature after removing instances with more than 5 *NaN*

So, as first step, we remove instances with more than 5 missing values. By doing that, we loose only 54 instances (13,5% of original data). After that, we check again missing values per feature to see if there are significant changes. The plot in Figure 3.15 shows a reduction of about 10% of *NaN* values for the features with the largest values.

Now, let's analyze in detail the process of sampling and replacing for one feature. The idea is to sample values from the real distribution and replace *NaN* values with these. In order to do that, we have to reconstruct that distribution starting from the original data. Once found it, we will use the *numpy.random.choice* function to sample values. This function needs the list of values from which samples will be taken and the probabilities associated with each entry in that list (we use the relative frequency of each value in our distribution).

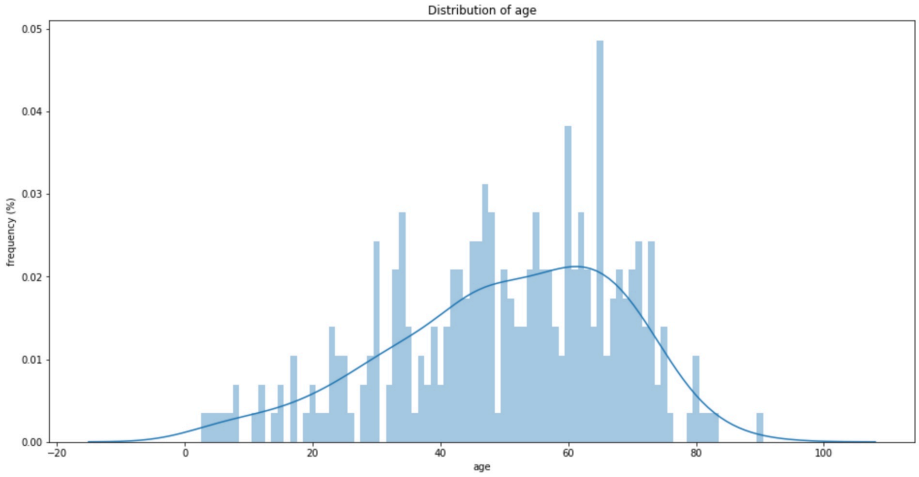


Figure 3.16. Distribution of age

Output:

```
Number of Nan: 3
1 Nan replaced with: 70.0
1 Nan replaced with: 56.0
1 Nan replaced with: 42.0
Number of Nan after the operation: 0
```

Results shows that values has been taken in the range 42-70 were there is the highest concentration of samples in the original distribution. The same operation has been done for all the other columns with missing values, so after this process there are no missing value in the resulting dataset.

Now, let's check if all these changes have affected statistics and class distribution to detect if we introduced any unbalance.

	age	bp	sg	al	su	rbc	pc	pcc	ba	bgr	...	pcv	wbcc
count	291.000000	291.000000	291.000000	291.000000	291.000000	291.000000	291.000000	291.000000	291.000000	291.000000	...	291.000000	291.000000
mean	50.350515	74.604811	1.018488	0.776632	0.281787	0.123711	0.171821	0.099656	0.054983	132.896907	...	40.955326	8378.006873
std	17.609583	11.541462	0.005441	1.212693	0.856805	0.329819	0.377875	0.300057	0.228339	55.639694	...	7.971159	2515.043870
min	3.000000	50.000000	1.005000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	22.000000	...	14.000000	2200.000000
25%	39.000000	70.000000	1.015000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	99.000000	...	35.500000	6700.000000
50%	53.000000	70.000000	1.020000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	118.000000	...	42.000000	8100.000000
75%	64.500000	80.000000	1.025000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	140.000000	...	47.500000	9800.000000
max	90.000000	100.000000	1.025000	4.000000	5.000000	1.000000	1.000000	1.000000	1.000000	352.000000	...	54.000000	19100.000000

Figure 3.17. Feature statistics after data cleaning operations

Deleting extreme outliers and some kind instances with missing values, leads to loose 109 instances in total. However, the process not only does not introduce unbalance in class distribution but, since almost all of those instances belong to class 0 (*notckd*), it leads to a dataset which has almost the same number of instances for both classes.

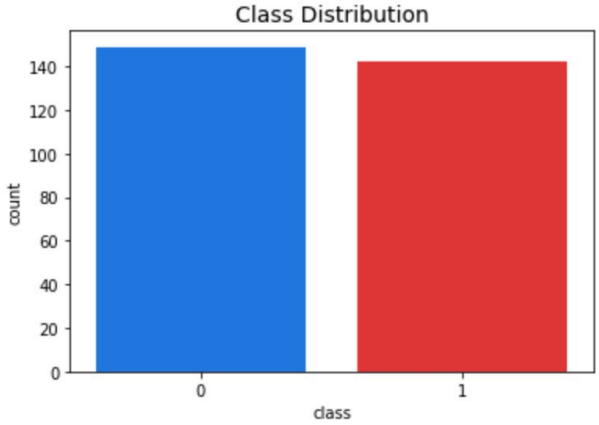


Figure 3.18. Class distribution after data cleaning operations

3.5 Further Analysis

In this section, we will perform an analysis of correlations among predictors and among predictors and class. The correlation matrix allows to know if there is any statistical relationship between two attributes, and it is particularly interesting to see if values assumed by attributes are related to Positive or Negative class. The correlation matrix is showed through the *heatmap* in Figure 3.19.

From it we can see that there are some attributes which are **positively correlated**, highlighted with red tones, such as *hemo-pcv*, *sc-bu* and *htn-dm*. This means that for these pairs of attributes the higher the first is, the higher the second is. This can be graphically verified with scatter plot in the top of Figure 3.20.

There are also some attributes that are **negatively correlated**, highlighted with blue tones, such as *hemo-sc, pcv-sc, hemo-htn*, *pcv-htn*. This means that for these pairs of attributes the higher the first is, the lower the second is, as showed by scatter plot in the bottom of Figure 3.20.

Considering correlations with the **class**, the most negatively correlated are *hemo*, *sg* and *pcv*. Hence, the lower these value are, the more likely the patient is Positive to CKD. The most positively correlated are *htn*, *al* and *dm*. So, the higher these value are, the more likely the patient is Positive to CKD. While there is no correlation with the value of *pot*. This confirms what we inferred during the analysis of boxplots divided by class.

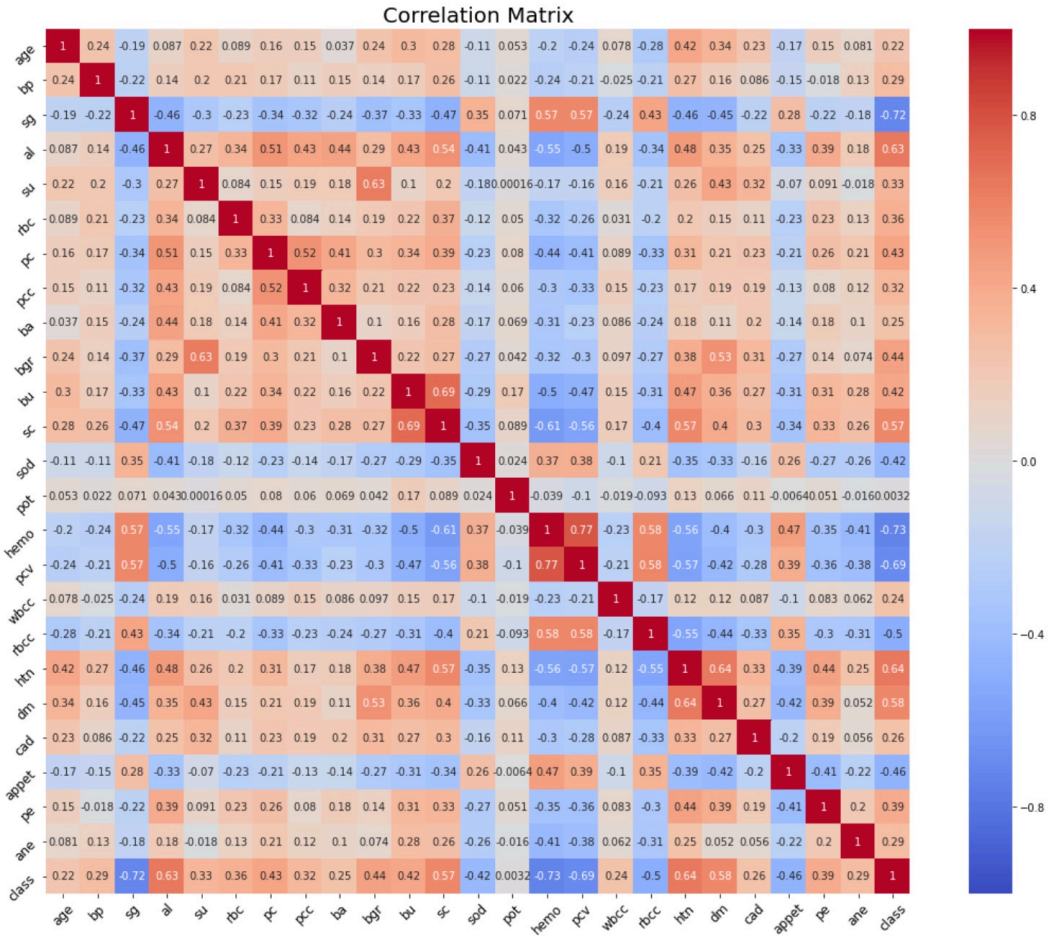


Figure 3.19. Heatmap showing the correlation matrix: red tones means positive correlation, blue tones negative correlation

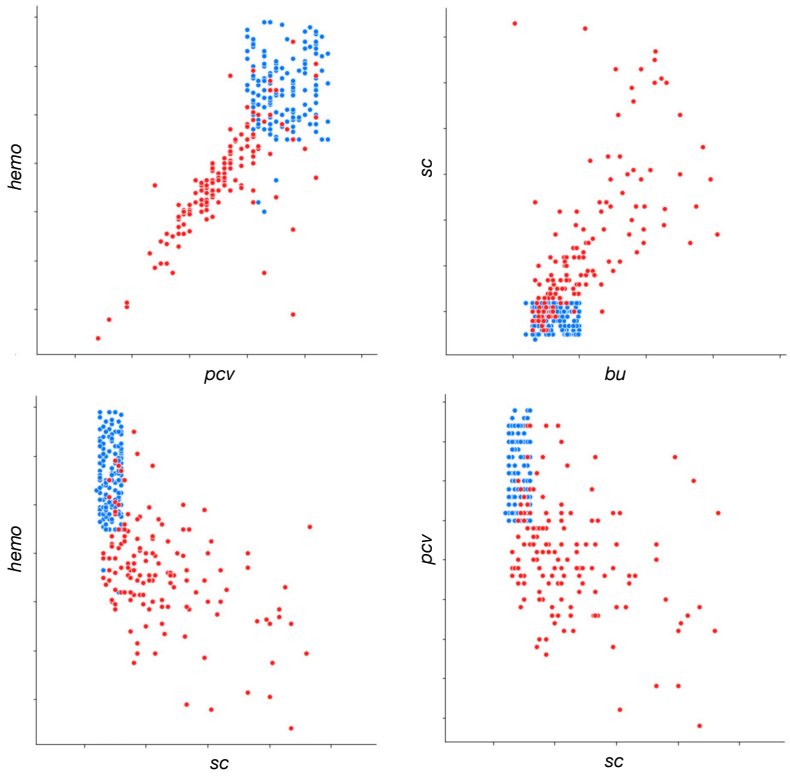


Figure 3.20. Top row: two pairs of the most positive correlated variables. Bottom row: two pairs of the most negative correlated variables

4. Principal Component Analysis

In this section, we will use Principal Component Analysis (PCA) to perform a dimensionality reduction of our data. PCA is an unsupervised approach (it involves only features and no target class) that allows to find a low-dimensional representation of a data set that captures as much of the information as possible. The idea is to find a smaller number of representative variables, known as Principal Components, that collectively explain most of the variability in the original set. Specifically, PCA seeks the directions along which original data are characterized by the maximum variance (the higher the variance the more data are separable, but not always) and projects data in a lower dimensional subspace (defined by these directions). Principal Components are computed with the constraint that each of them is orthogonal to the others, hence the new dimensions are uncorrelated.

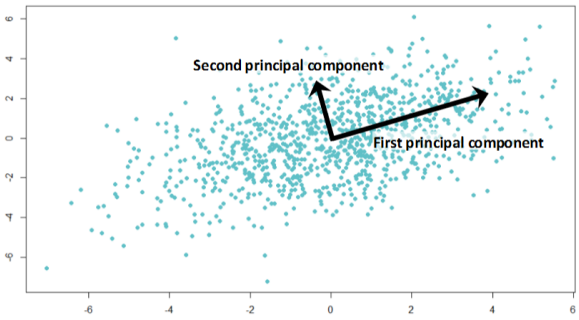


Figure 4.1. Directions of Principal Components

Given our data matrix of d samples and n features $\underset{dxn}{X} = \{x_1 \dots x_n\}$ where x_i are columns of d elements, we want to reduce dimensions from n to k using PCA. To do that, we can follow the steps below:

I. Subtract sample mean from the data $z_i = x_i - \hat{\mu}$, where $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$

II. Compute the Covariance matrix $\underset{nxn}{S} = Z^t Z$

III. Compute eigenvectors $\{e_1, e_2, \dots, e_k\}$ corresponding to the k largest eigenvalues λ of S , so we get the matrix $\underset{nxk}{E} = [e_1 \dots e_k]$, where e_i are columns and are ordered according to λ values which give measures of the explained variance

IV. The desired Y which is the closest approximation to X is $\underset{dxk}{Y} = \underset{dxn}{Z}\ \underset{nxk}{E}$

So, the first principal component of a set of features $\{z_1, z_2, \dots, z_n\}$ is the normalized linear combination of the features:

$$y_1 = e_{11}z_1 + e_{21}z_2 + \dots + e_{n1}z_n$$

with the constraint that $\sum_{j=1}^{n} e_{j1}^2 = 1$

where $e_{11}, \dots, e_{n1}$ are referred as loadings of the first principal component and $y_{11}, \dots, y_{n1}$ as scores of the first principal component.

The loading vector e_i defines the direction along which the data vary the most. The principal component scores $y_{11}, \dots, y_{n1}$ represent the projection of the initial data $x_{11}, \dots, x_{n1}$ onto this direction.

Often it is possible that features are measured in different units and since PCA uses variance to find the new axis, variable characterized by an higher variance will influence the process more than others. Hence, it is important to scale each variable to have standard deviation one so that all variable will have the same influence.

In order to preserve some data for the final testing of models that will be selected in the next section, the dataset has been randomly split into two sets: Train Set (60%) and Test Set (40%). The reasons of this operation will be better explained in the next section.

After dividing training set and test set, a feature scaling with **Z-score** normalization has been performed on the data using the *StandardScaler* provided by *scikit-learn*. It standardizes features by removing the mean and scaling to unit variance. This operation is important since it is possible that features are measured in different units.

The standard score of a sample x is calculated as

$$z = \frac{x - \mu}{\sigma}$$

where μ and σ are the mean and the standard deviation of the training samples respectively.

The fit method has been called on the train set in order to compute mean and the standard deviation on this set, and then use these values for later scaling.

To have an idea of how much of the information is lost by projecting the observation in a lower dimensional space, we analyze the **Proportion of Variance Explained** (PVE) by each principal component. The total variance of the dataset (variable centered) is

$$\sum_{j=1}^{n} Var(X_j) = \sum_{j=1}^{n} \frac{1}{d} \sum_{i=1}^{d} x_{ij}^2$$

and the variance explained by the m^{th} principal component is

$$\frac{1}{d} \sum_{j=1}^{d} y_{im}^2 = \frac{1}{d} \sum_{i=1}^{d} \left(\sum_{j=1}^{n} e_{jm} x_{ij} \right)^2$$

therefore, the PVE of the m^{th} principal component is

$$\frac{\sum_{i=1}^{d} \left(\sum_{j=1}^{n} e_{jm} x_{ij} \right)^2}{\sum_{j=1}^{n} \sum_{i=1}^{d} x_{ij}^2}$$

The plot in Figure 4.2 shows the PVE of each principal component and the Cumulative PVE. From it we can see that the first principal component explains 31% of the variance in the data, while all the others explain less than 8% each.

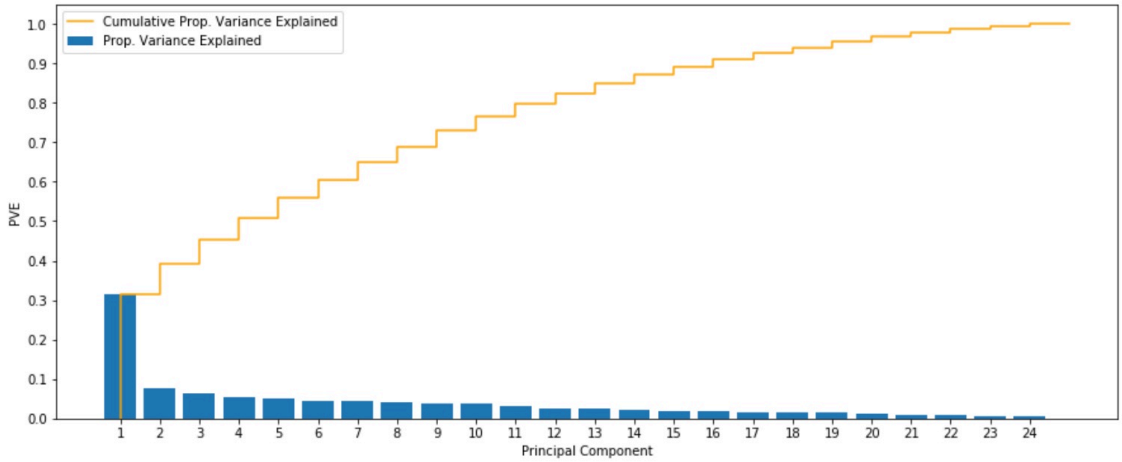


Figure 4.2. Proportion of variance explained

For the next analysis we will try to use the first two principal component that will allows us to visualize data in two dimensional graph. Despite together they explain only about 40% of variance, by using only these two as predictors will allows us to obtain good classifiers as discussed in the next section.

Figure 4.3. Pairplot considering the first 6 Principal Components that explain about 60% of variance

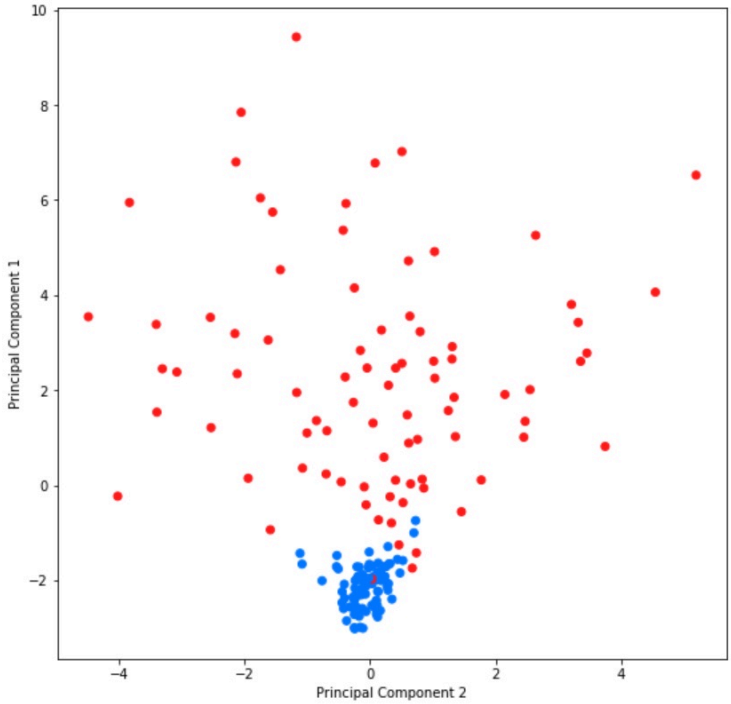


Figure 4.4. Scatter plot considering the first 2 Principal Components

So considering these two dimensions, the classification problem consists in finding the classifier that allows to 'better' separate blu (notCKD) from red (CKD) observations showed in the scatter plot below, and then test how it performs on new data.

5. Classification

In this section, we will build some classification models, tune some hyper-parameters and evaluate the performance changes in order to find the best classifier for our data. Specifically, we will analyze the behavior of 5 different classification algorithms:

- k-Nearest Neighbor
- Logistic Regression
- SVM
- Decision Tree
- Random Forest

As introduced above, the initial dataset has been split in two sets (Train and Test) in order to preserve some data for the final evaluation. We perform all the operations of hyper-parameters tuning without considering them, so that the choice of hyper-parameters values is not affected by these data, and then we can see how the selected model works on new data i.e. how good it is to generalize. Therefore, models are trained and evaluated using the *K-Fold Cross Validation* on the train set, and once found the best configuration, the final evaluation can be done on the test set.

The **K-Fold Cross Validation** is a validation technique which consists in randomly dividing the set of observations into *k* groups, known as folds, and use the one fold as validation set and the other *k-1* folds as training set. The accuracy is computed on the current validation set. The process is repeated *k* times, considering each time a different split for the validation, so at the end we will have *k* values of accuracy. The final evaluation is obtained by computing the average of these values.

Figure 5.1. 10-Fold Cross Validation

In our experiment we will use the Stratified 10-Fold Cross Validation. This is a variation of 10-Fold Cross Validation that returns stratified folds: the folds are made by preserving the percentage of samples for each class.

5.1 K-Nearest Neighbors

The K-Nearest Neighbors is a very simple algorithm which is based on the idea that an element can be classified according to his neighbors. The principle behind this method indeed is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number K of samples is defined by the user while the distance can be any metric measure such as the standard Euclidean distance:

$$dist = \sqrt{\sum_{k=1}^{n} (p_k - q_k)^2}$$

Where n is the number of dimensions (attributes) and p_k and q_k are, respectively, the k^{th} attributes (components) of data objects p and q.

The nearest neighbor algorithm is particularly sensitive to outliers so that a single mislabeled example can dramatically change the prediction. Therefore, the choice of K is extremely important for a correct classification: if the value of K is very low, classification could be affected by outliers while, if it is too large, everything tend to be classified as the most probable class.

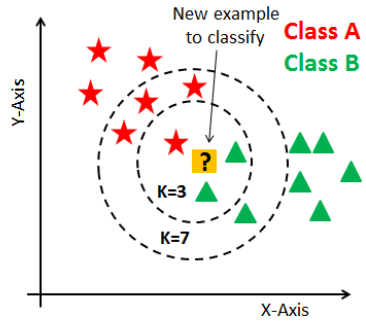


Figure 5.2. K-Nearest Neighbors

The K-NN classifiers are 'memory based': they do not attempt to construct a general internal model, but simply store instances of the training data. This allows to have a training time of 0 but introduce another inconvenient. When a new observation has to be classified, the distances between this element and those in the train data have to be computed and then it is classified using a simple majority vote among the K nearest neighbors. Therefore, it can be inefficient when the train set is very huge and so inconvenient in real time applications where it is very important to have a prediction as fast as possible.

Despite its simplicity, nearest neighbors has been successful in a large number of classification: it is often successful in classification situations where the decision boundaries are very irregular.

In order to find the best value of *K*, a *Grid Search* has been performed. Since we are dealing with a binary classification problem, we choose only odd values of *K*. The 10-Fold Cross Validation will be used as validation strategy and the value that allows to achieve the highest accuracy will be selected.

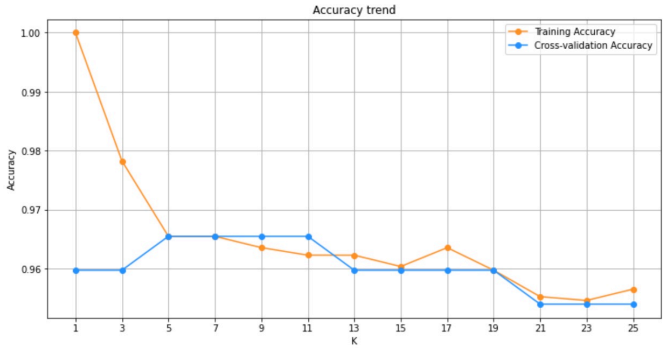


Figure 5.3. Accuracy trend for different values of K

As we can see, the best value of *K* is 5 which allows to achieve a very high accuracy (about 96,5% using cross validation). Higher values of K instead lead to a constant decrease of performance.

Let's check now the **Confusion Matrix** to analyze the wrong predictions and compute some other performance metrics such as Precision, Recall and F1-score.

Confusion Matrix is a performance measurement for machine learning classification problem. It consists in a table with 4 different combinations of predicted and actual values known as True Positive, True Negative, False Positive and False Negative.

- **True Positive**: number of samples correctly predicted as positive
- **True Negative**: number of samples correctly predicted as negative
- **False Positive**: number of samples incorrectly predicted as positive
- **False Negative**: number of samples incorrectly predicted as negative

From these values it is possible to compute some other metrics such as:

- **Accuracy** $= \dfrac{TP + TN}{TP + TN + FP + FN}$

  which is the ratio of correct predictions to total predictions made.

- **Precision** $= \dfrac{TP}{TP + FP}$

  which is proportion of positive identifications that was actually correct.

- **Recall** $= \dfrac{TP}{TP + FN}$

  which is proportion of actual positives that was identified correctly.

- **f1-score** $= 2 \cdot \dfrac{Precision \cdot Recall}{Precision + Recall}$

  which combines the previous two.

In our case, we may be interested in a Recall near to 1 (i.e. FN near to 0), since our is a preliminary disease screening of patients so we want to find all patients who actually have the disease and we can accept a lower Precision (i.e. more FP) if the cost of the follow-up examination is not significant.

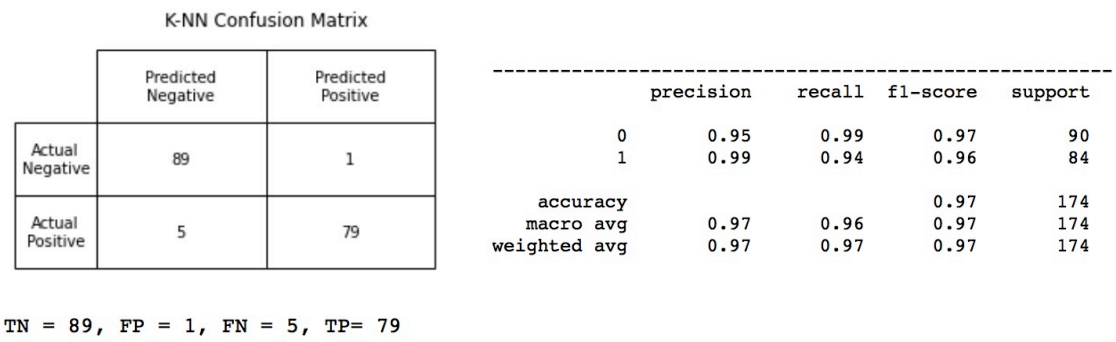


Figure 5.4. K-Nearest Neighbor confusion matrix

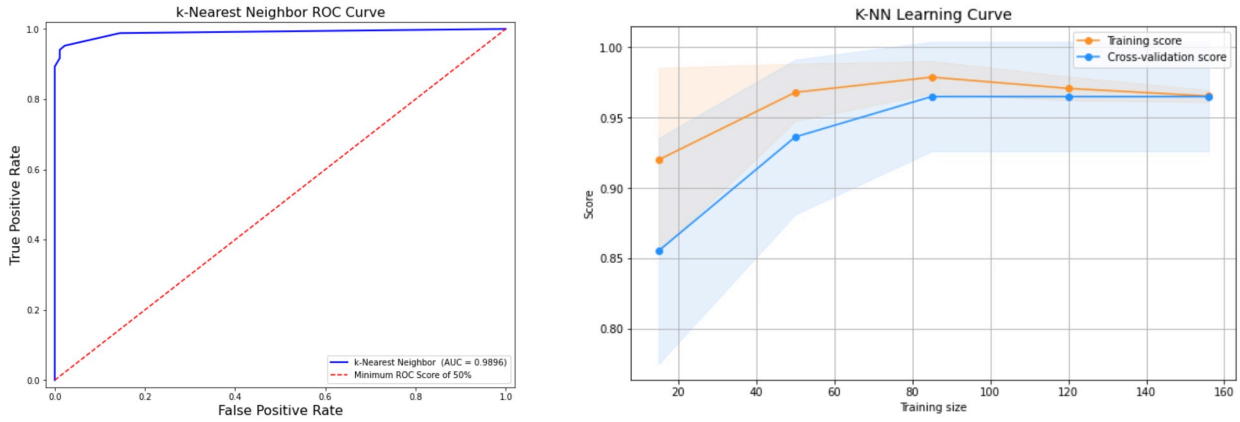


Figure 5.5. Left: K-Nearest Neighbor ROC curve. Right: K-Nearest Neighbor Learning curve

The **ROC** (Receiver Operating Characteristic) **Curve** displays the *True Positive Rate* (TPR/Recall/ Sensitivity) on the y-axis, and on the x-axis the *False Positive Rate* defined as

$$FPR = 1 - Specificity$$

where $Specificity = \frac{TN}{TN + FP}$ is proportion of actual negative that was identified correctly.

Considering the class probability, we classify an observation as Positive if

$$\hat{Pr}(Y = Positive \mid X) \geq threshold$$

so, we can change these two metrics by changing the threshold in [0,1]. The top left corner of the plot is the "ideal" case with a false positive rate of zero, and a true positive rate of one. The *AUC* (area under the curve) allows to summarize the overall performance. A larger AUC is usually better.

The **Learning Curve** plot shows the performances of the model on both Training and Validation set. It allows to make important analysis about our model: for instance, if we have high training score and low validation score at the same time it means that the model has overfit the data, or if the learning curve is still raising at the given training set size, it is likely that error might still decrease when providing more data to the model, so we should increase the training set.

5.2 Logistic Regression

Logistic Regression models the probability that Y belongs to a particular category, rather than directly the Y response.

$$p(X) = Pr(Y = 1 \mid X)$$

In general, when the response variable assumes only values in the [0,1] interval, may be inappropriate the use of linear regression that can produce output in the range $]-\infty, +\infty[$. By definition $Y = a + bX$, so it might produce probabilities less than zero or bigger than one.

Logistic Regression uses the **logistic function**:

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

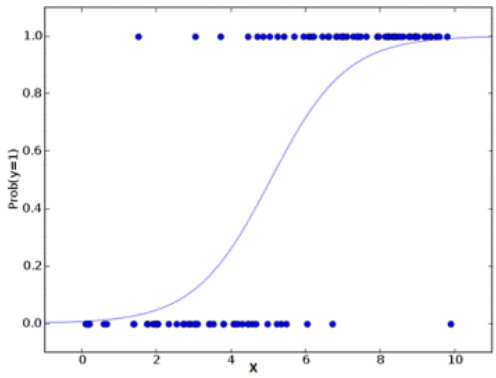


Figure 5.6. Predicted probabilities using Logistic Regression

The logistic function always produces an S-shaped curve in the [0,1] interval, and so regardless of the value of X, it ensures that our estimate for $p(X)$ lies between 0 and 1.

After some manipulation, we obtain:

$$\frac{p(X)}{1-p(X)} = e^{\beta_0+\beta_1 X}$$

where $\frac{p(X)}{1-p(X)}$ is called *odds*: given a value between 0 and ∞, it returns a value between 0 and 1.

After taking the logarithm of both sides, we obtain:

$$log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X$$

the element on the left side is called *log-odds* or *logit*. So, logistic regression has a logit that is linear in X.

In this case, results obtained with Logistic Regression are a little worse with respect to those obtained with the previous model.

Log_Reg Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	88	2
Actual Positive	5	79

	precision	recall	f1-score	support
0	0.95	0.98	0.96	90
1	0.98	0.94	0.96	84
accuracy			0.96	174
macro avg	0.96	0.96	0.96	174
weighted avg	0.96	0.96	0.96	174

TN = 88, FP = 2, FN = 5, TP= 79

Figure 5.7. Logistic Regression confusion matrix

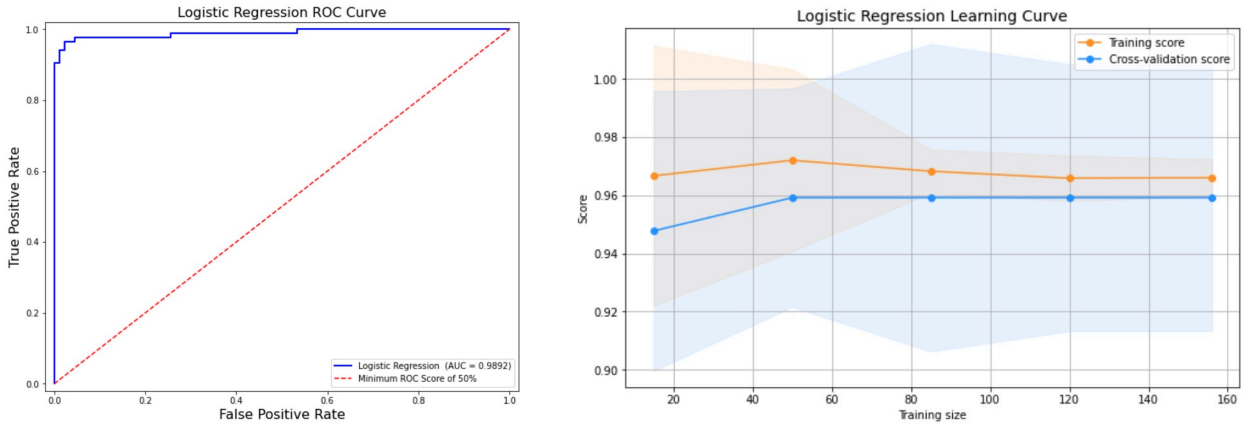


Figure 5.8. Left: Logistic Regression ROC curve. Right: Logistic Regression Learning curve

5.3 Support Vector Machine

The Support Vector Machine (SVM) is a generalization of the Maximal Margin Classifier which is based on the concept of *hyperplane*: given a p-dimensional space, an hyperplane is a flat subspace of dimension p-1 and is defined as:

$$f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0$$

All points X that satisfy this equation lie on the hyperplane.

The idea behind these classifiers is that it is possible to construct a hyperplane that separates the training data according to their class labels, and than classify a new element $x^* = \left(x_1^* \ldots x_p^*\right)^T$ according to which side of the hyperplane it is located:

if $f(x^*) > 0$ we assign the test observation to class 1 else if $f(x^*) < 0$ we assign it to class -1.

The **Maximal Margin Classifier** choose the maximal margin hyperplane which is the separating hyperplane that is farthest from the training observations. This can be done by solving the following optimization problem:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,M}{maximize} M$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M \quad \forall i = 1,\ldots,n$$

However, in many cases data can belong to two classes that are not separable by a hyperplane, and so there is no maximal margin classifier. So, a 'soft' margin approach must be considered: the **Support Vector Classifier** tries to find an hyperplane that 'almost' separates the classes, allowing some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane. This remove also the effect that a single observation can have on a maximal margin hyperplane and that can lead to a tiny margin. So, the optimization problem becomes:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n,M}{maximize} \quad M$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M(1 - \epsilon_i),$$

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C$$

C is a tuning parameter that determines the number and severity of the violations to the margin that we will tolerate. So, when C is small we obtain a narrow margin that is rarely violated and this lead to a classifier that is highly fit to the data. As the value of C increase, the margin become wider (we allow more violations to it) and so we obtain a classifier that is less and less fit to the data.

When classes are not-linearly separable, it is possible to enlarge the feature space, in order to accommodate a non-linear boundary between the classes, using ***kernel*** functions such as the *Radial Basis Function*, which takes the form:

$$K(x_i, x_{i'}) = exp\left(-\gamma \sum_{j=1}^{p} \left(x_{ij} - x_{i'j}\right)^2\right)$$

When the support vector classifier is combined with a non-linear kernel, the resulting classifier is known as a **Support Vector Machine**. Instead, if we choose a *linear* kernel defined as:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

we simply get back the Support Vector Classifier.

Also this time, we will perform a *Grid Search* in order to find the best hyper-parameters. The hyper-parameters that we consider are:

- 'kernel' among the two defined above (linear or rbf);
- 'gamma' coefficient for the rbf kernel ('auto' = 1/n_features and 'scale' = 1/(n_features * X.var()));
- 'C' in scikit-learn, the parameter C of the SVC learner is the penalty for misclassifying a data point. It allows to specify the degree of tolerance we want to give when finding the decision boundary for the SVM. The bigger the C, the more penalty SVM gets when it makes misclassification and therefore, we will obtain narrow margins that are rarely violated. On the other hand, when C is small, the margin will be wider and we allow more violations to it.

SVC Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	89	1
Actual Positive	4	80

	precision	recall	f1-score	support
0	0.96	0.99	0.97	90
1	0.99	0.95	0.97	84
accuracy			0.97	174
macro avg	0.97	0.97	0.97	174
weighted avg	0.97	0.97	0.97	174

TN = 89, FP = 1, FN = 4, TP= 80

Figure 5.9. SVM confusion matrix

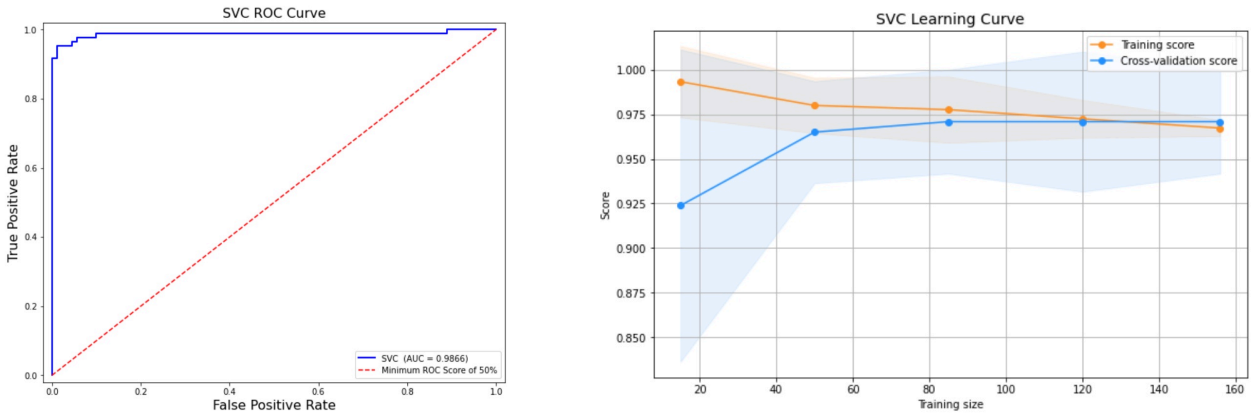


Figure 5.10. Left: SVM ROC curve. Right: SVM Learning curve

5.4 Decision Tree

The Decision Tree is one of the easiest classification method to explain. It is based on a tree structure where internal nodes are decision nodes and endpoint nodes (leaves) provide the final outcome (class). Decision nodes decide which branch to use inside the tree according to values of features.

The construction of a Classification Tree consists in dividing the predictor space $X_1, X_2, \ldots, X_p$ into J distinct and non-overlapping regions $R_1, R_2, \ldots, R_j$. Then, each observation that falls into a region R_j, is simply classified as the most commonly occurring class of training observations in that region.

In order to build those regions, a top-down, greedy approach (also known as recursive binary splitting) is taken. The recursive approach is top-down because it begins at the top of the tree (at which point all observations belong to a single region) and then successively splits the predictor space; each split is indicated via two new branches further down on the tree. It is greedy because at each step of the tree-building process, the best split is made at that particular step. The choose of the predictor X_j and the cutpoint s is made according to a criterion for making the binary splits. The most common used are:

- Gini index defined by

$$gini = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

where p_{mk} represents the proportion of training observations in the m^{th} region that are from the k^{th} class. Gini index provides a measure of total variance across the K classes and assumes small value if all of $\hat{p}_{mk}$'s are close to 0 or 1.

- Entropy given by

$$entropy = -\sum_{k=1}^{K} \hat{p}_{mk} log\, \hat{p}_{mk}$$

also in this case, entropy assumes a value near zero if the $\hat{p}_{mk}$'s are all near zero or near one.

Both criterions are referred as measure of node purity: a small value indicates that a node contains predominantly observations from a single class (the node is pure). Therefore, at each step the choice of the predictor and the cutpoint is made so that it minimize this value. The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than a certain number of observations or the tree reaches a certain depth.

Decision Tree Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	87	3
Actual Positive	6	78

	precision	recall	f1-score	support
0	0.94	0.97	0.95	90
1	0.96	0.93	0.95	84
accuracy			0.95	174
macro avg	0.95	0.95	0.95	174
weighted avg	0.95	0.95	0.95	174

TN = 87, FP = 3, FN = 6, TP= 78

Figure 5.11. Decision Tree confusion matrix

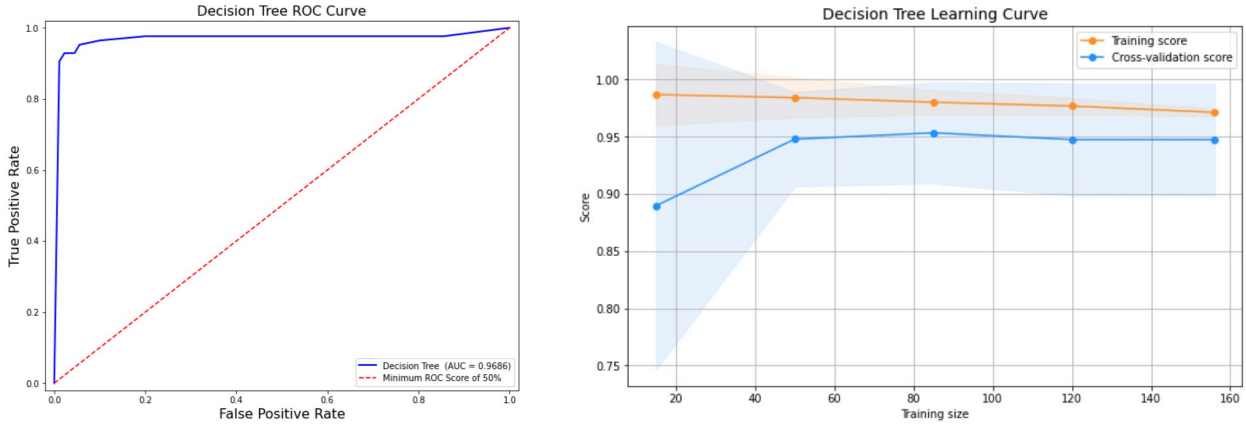


Figure 5.11. Left: Decision Tree ROC curve. Right: Decision Tree Learning curve

The structure of the resulting tree is showed below.

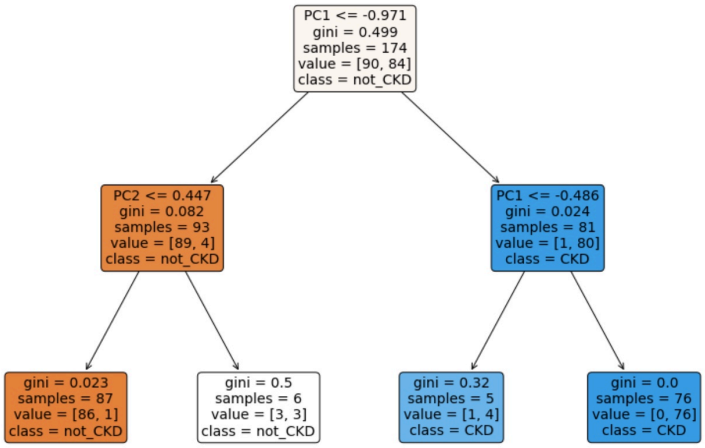


Figure 5.12. Decision Tree structure

5.5 Random Forest

Usually Decision Tree do not have the same level of accuracy as other classifier. Despite in our case we also achieve good performance due to data easily separable, it is still the worst classifier. Decision Trees suffer from high variance, therefore it is possible to use some techniques that can help to improve predictive performance.

Bootstrap aggregation, or **Bagging**, is a general-purpose procedure for reducing the variance of a statistical learning method: given a set of n independent observation $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean of the observation is given by σ^2/n (i.e. averaging a set of observations reduces variance). The idea is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions. Since it is not practical because we generally do not have access to multiple training set, we can use the *bootstrap* technique (take repeated samples with replacement from the initial dataset) to generate B different bootstrapped training sets. Then, we train a Tree on each of the B sets in order to get the prediction, and finally average all the predictions to obtain the final result:

$$f_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} f_b(x)$$

These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. In the case of classification problem, the strategy of *majority vote* is used: the final prediction is the most commonly predicted class by the B Trees.

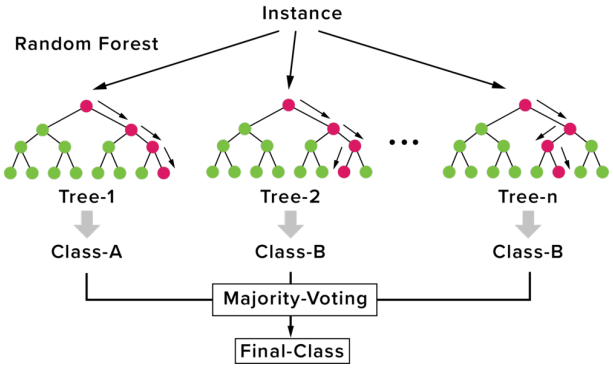


Figure 5.13. Random Forest

Random Forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees. As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$ that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors. By considering a new subset of predictors at each split, allows Random Forest to obtain trees that are not too much similar to each other. Hence, the correlation among predictions from these trees decreases (averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities).

Random Forest Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	88	2
Actual Positive	4	80

```
                    precision    recall  f1-score   support

                 0       0.96      0.98      0.97        90
                 1       0.98      0.95      0.96        84

          accuracy                           0.97       174
         macro avg       0.97      0.97      0.97       174
      weighted avg       0.97      0.97      0.97       174
```

```
TN = 88, FP = 2, FN = 4, TP= 80
```

Figure 5.14. Random Forest confusion matrix

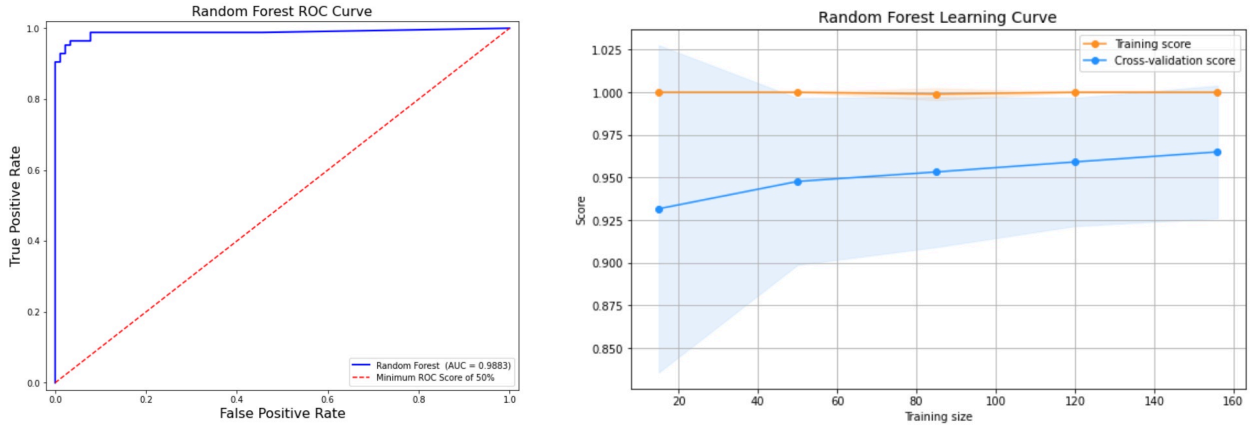


Figure 5.15. Left: Random Forest ROC curve. Right: Random Forest Learning curve

6. Comparing results and Conclusions

Comparing the results obtained, we can see that all classifiers analyzed allows to obtain very good performances with an accuracy greater than 0.95 in all cases, using 10-Fold Cross Validation as validation strategy. The best classifier is SVM with rbf kernel that, according to confusion matrix, allows to obtain only 4 False negative and 1 False positive with an overall accuracy of 0.97. Similar results are achieved by Random Forest with only 1 more False positive with respect to SVM. The worst classifier result as expected the Decision Tree with 6 False negative and 3 False positive. Despite the scope of improvement is small, the use of technique to improve Decision Tree performance (i.e. Random Forest) works fine. The other two classifiers also allow to achieve quite good results.

Looking at the ROC Curves, we see that in all case the curve is very near to the top-left corner. This mean that all classifiers analyzed behave similar to 'perfect' classifier. The good performance can be also confirmed by looking at the AUC that is in all case higher than 0.9686.

Considering the Learning Curves, in all cases the curve takes an horizontal trend in the final part of the plot. This means that the size of the dataset is enough and so providing more data should not lead to a further reduction of the error. The only exception seems to be the case of Random Forest where the cure is slightly growing, so in this case we could expect some improvements. Moreover, in all cases we can see that both training and validation curves are quite near, and this means that we are not overfitting our training data.

After that the best configurations have been found, all classifiers have been trained on the whole train set (60% of the initial dataset) and finally tested on the test set (40%) using the splits that we randomly generated at the beginning of the analysis.

K-NN Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	59	0
Actual Positive	0	58

Log_Reg Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	59	0
Actual Positive	0	58

SVC Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	58	1
Actual Positive	0	58

Decision Tree Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	59	0
Actual Positive	0	58

Random Forest Confusion Matrix

	Predicted Negative	Predicted Positive
Actual Negative	58	1
Actual Positive	0	58

Figure 6.1. Confusion matrices obtained on Test Set

Figure 6.1 shows the resulting confusion matrices obtained. From these we can see that almost all classifiers predict correctly the class for all samples and only SVM and Random Forest make a mistake on 1 case. However these results depends on the particular splits that we obtain during the random generation: performing multiple times these analysis with different splits we can obtain results that are slightly different but always extremely accurate.

Figure 6.2 shows the resulting decision boundaries for all classifiers. From there it is particularly interesting to notice the non-linear decision boundary in the case of SVM due to the use of the rbf kernel. Logistic Regression instead generates a linear decision boundary while K-Nearest Neighbor and Random Forest produce more wiggled boundaries. Decision Tree substantially divides the area in only two region, so we have a vertical split.

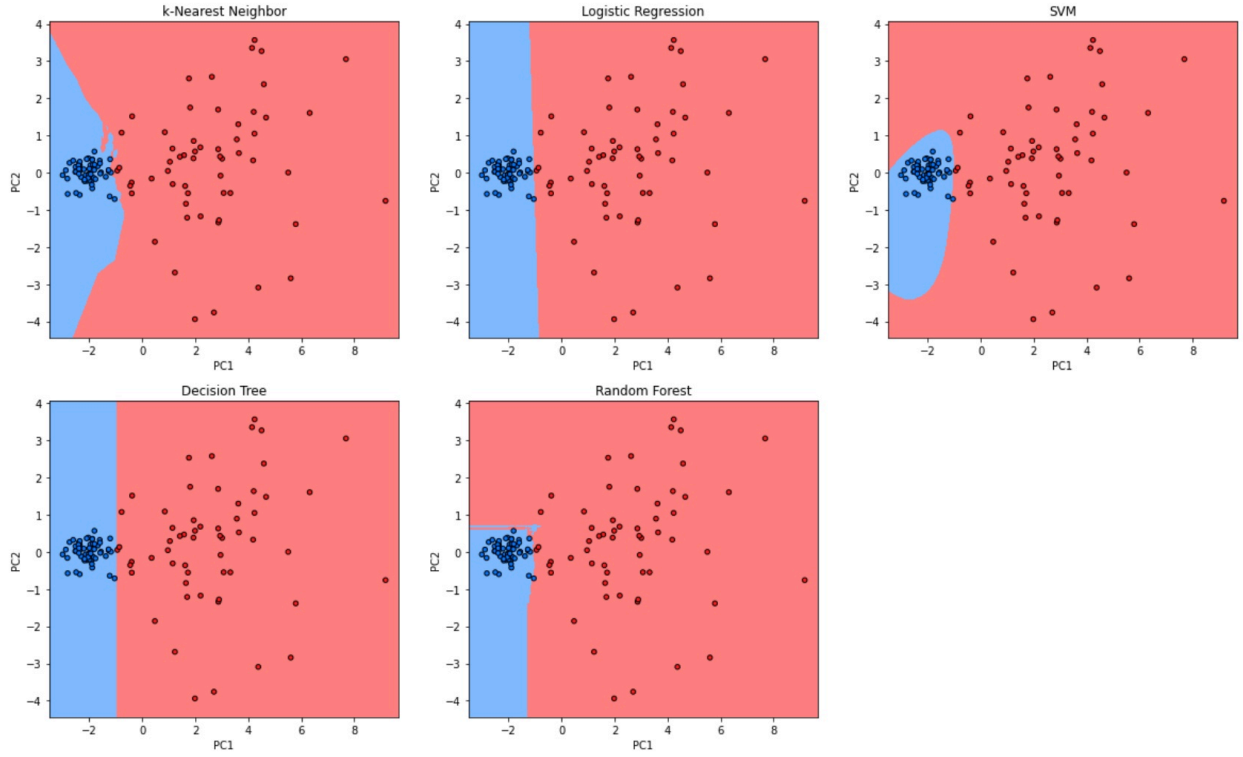


Figure 6.2. Decision Boundaries and Scatter plot of Test Set samples