Chapter 9
Machine Learning for Patient Stratification and Classification Part 1: Data Preparation and Analysis

Cátia M. Salgado and Susana M. Vieira

Abstract Machine Learning for Phenotyping is composed of three chapters and aims to introduce clinicians to machine learning (ML). It provides a guideline through the basic concepts underlying machine learning and the tools needed to easily implement it using the Python programming language and Jupyter notebook documents. It is divided into three main parts: part 1—data preparation and analysis; part 2—unsupervised learning for clustering, and part 3—supervised learning for classification.

Keywords Machine learning · Phenotyping · Data preparation · Data analysis · Unsupervised learning · Clustering · Supervised learning · Classification · Clinical informatics

9.1 Learning Objectives and Approach

It is recommended that you follow this chapter using the jupyter notebook provided in https://github.com/cmsalgado/book_chapter, so that you can experiment with the code on your own. Since we are not implementing most of the algorithms and models, you will be able to follow the notebook even if you have no prior experience with coding. We will cover a large number of well-known ML algorithms, starting with the simplest (logistic regression and k-nearest neighbors) and ending with more complex ones (random forest), without delving in much detail into the underlying mathematical theory. Instead, we will focus on explaining the intuitions behind algorithms so that you understand how they learn and how you can use and interpret them for any new problem.

A real-world clinical dataset will be used. It was constructed based on the code provided in https://github.com/YerevaNN/mimic3-benchmarks for the prediction of hospital mortality using data collected during the first two days in the ICU. The data was extracted from the MIMIC-III Clinical Database, which is a large,

C. M. Salgado (✉) · S. M. Vieira
IDMEC Instituto Superior Técnico, Universidade de Lisboa, Av. Rovisco Pais, Lisboa, Portugal
e-mail: catia.salgado@tecnico.ulisboa.pt

L. A. Celi et al. (eds.), *Leveraging Data Science for Global Health*,
https://doi.org/10.1007/978-3-030-47994-7_9

publicly-available database comprising de-identified electronic health records of approximately 60 000 ICU admissions. Patients stayed in the intensive care unit (ICU) of the Beth Israel Deaconess Medical Center between 2001 and 2012. MIMIC-III database is described in:

> Johnson AEW, Pollard TJ, Shen L, Lehman L, Feng M, Ghassemi M, Moody B, Szolovits P, Celi LA, Mark RG. MIMIC-III, a freely accessible critical care database. Scientific Data (2016).

Before dwelling into ML, this chapter presents the data preparation phase, which consists of a series of steps required to transform raw data from electronic health records into structured data that can be used as input to the learning algorithms. This is a crucial part of any ML project. First, it is important to understand what the algorithm is supposed to learn so that you can select appropriate information/variables to describe the problem. In other words, you want to avoid what is frequently referred to as "garbage in, garbage out". Second, since it is usually not handed in a format that is ready for straightaway ML, chances are that you will need to spend some time preprocessing and analyzing the data. You will see that in the end, your results are highly dependent on decisions made during this part. It does not matter to keep trying to optimize the ML algorithm if you have poor data, your algorithm will not be able to learn. In this scenario, you probably gain more by going back to the stages of data extraction and preprocessing and rethink your decisions; it is better to have good data and a simple algorithm than poor data and a very complex algorithm. In particular, the data preparation phase consists of:

- Exploratory data analysis
- Variable selection
- Identification and exclusion of outliers
- Data aggregation
- Inclusion criteria
- Feature construction
- Data partitioning.

The machine learning phase consists of:

- Patient stratification through unsupervised learning
  - k-means clustering
- Patient classification through supervised learning
  - Feature selection
  - Logistic regression
  - Decision trees
  - Random forest.

9.2 Prerequisites

In order to run the code provided in this Chapter, you should have Python and the following Python libraries installed:

- NumPy: fundamental package for scientific computing with Python. Provides a fast numerical array structure.
- Pandas: provides high-performance, easy-to-use data structures and data analysis tools.
- Scikit-learn: essential machine learning package in Python. Provides simple and efficient tools for data mining and data analysis.
- matplotlib: basic plotting library.
- seaborn: visualization library based on matplotlib. It provides a high-level interface for drawing statistical graphics.
- IPython: for interactive data visualization.
- Jupyter: for interactive computing.

It is highly recommended that you install Anaconda, which already has the above packages and more included. Additionally, in order to be able to visualize decision trees, you should install pydot and graphviz. Next, we will import the python libraries that we will need for now and MIMIC-III data.

9.2.1 Import Libraries and Data

The next example shows the Python code that imports Numpy, Pandas, Matplotlib and IPython libraries:

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from IPython import display
        import warnings


        plt.style.use('ggplot')
```

The next example shows the Python code that loads the data as a Pandas DataFrame, which is a tabular data structure that makes it easy to do all kinds of manipulations such as arithmetic operations and grouping. The unique ICU stay ID is used as the DataFrame index, for facilitating data manipulation. This is achived by setting the 'index_col' parameter to the name of the index column.

```
In [2]: data = pd.read_csv('/https://www.dropbox.com/s/3l20njo1sb5ui4g/mimic3_mortality.csv?raw=1_',
    index_col='icustay')
```

9.3 Exploratory Data Analysis

A quick preview of 'data' can be obtained using the 'head' function, which prints the first 5 rows of any given DataFrame:

```
In [3]: data.head()
```

```
Out[3]:              hours  diastolic BP  glasgow coma scale  glucose  heart rate  \
        icustay
        282372.0  0.066667          60.0                 NaN      NaN       139.0
        282372.0  0.150000          73.0                 NaN      NaN       128.0
        282372.0  0.233333          81.0                 NaN      NaN       127.0
        282372.0  0.316667          86.0                 NaN      NaN       132.0
        282372.0  0.400000          86.0                 NaN      NaN       138.0

                    mean BP  oxygen saturation  respiratory rate  systolic BP  \
        icustay
        282372.0   84.666702              100.0              20.0        134.0
        282372.0   93.000000              100.0              25.0        133.0
        282372.0   88.666702              100.0              22.0        104.0
        282372.0  100.000000              100.0              19.0        128.0
        282372.0  100.333000              100.0              21.0        129.0

                  temperature        age  gender  height  pH  weight  day  mortality
        icustay
        282372.0          NaN  48.682393     2.0     NaN NaN    59.0    1          1
        282372.0          NaN  48.682393     2.0     NaN NaN    59.0    1          1
        282372.0          NaN  48.682393     2.0     NaN NaN    59.0    1          1
        282372.0          NaN  48.682393     2.0     NaN NaN    59.0    1          1
        282372.0          NaN  48.682393     2.0     NaN NaN    59.0    1          1
```

It tells us that the dataset contains information regarding patient demographics: age, gender, weight, height, mortality; physiological vital signs: diastolic blood pressure, systolic blood pressure, mean blood pressure, temperature, respiratory rate; lab tests: glucose, pH; scores: glasgow coma scale. Each observation/row is associated with a time stamp (column 'hours'), indicating the number of hours since ICU admission where the observation was made. Each icustay has several observations for the same variable/column.

We can print the number of ICU stays by calculating the length of the unique indexes, number of missing data using the 'info' function and summary statistics using the 'describe' function:

```
In [4]: print('Number of ICU stays: ' + str(len(data.index.unique())))
        print('Number of survivors: ' + str(len(data[data['mortality']==0].index.unique())))
        print('Number of non-survivors: ' + str(len(data[data['mortality']==1].index.unique())))
        print('Mortality: ' + str(round(100*len(data[data['mortality']==1].index.unique()) /
        len(data.index.unique()),1)) + '%')
        print()
        display.display(data.info(null_counts=1))
        display.display(data.describe())
```

```
Number of ICU stays: 21139
Number of survivors: 18342
Number of non--survivors: 2797
Mortality: 13.2%
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 1461282 entries, 282372.0 to 245756.0
Data columns (total 18 columns):
hours                  1461282 non-null float64
diastolic BP           1015241 non-null float64
glasgow coma scale     159484 non-null float64
glucose                248460 non-null float64
heart rate             1057551 non-null float64
mean BP                1007986 non-null float64
oxygen saturation      1065406 non-null float64
respiratory rate       1066675 non-null float64
systolic BP            1015683 non-null float64
temperature            321762 non-null float64
age                    1461282 non-null float64
gender                 1461282 non-null float64
height                 455361 non-null float64
pH                     116414 non-null float64
weight                 1309930 non-null float64
day                    1461282 non-null int64
mortality              1461282 non-null int64
hour                   1461282 non-null float64
dtypes: float64(16), int64(2)
memory usage: 251.8 MB
```

	hours	diastolic BP	glasgow coma scale	glucose	\
count	1.461282e+06	1.015241e+06	159484.000000	248460.000000	
mean	2.182227e+01	6.031228e+01	11.600668	143.930140	
std	1.421245e+01	1.452069e+01	3.920855	67.442769	
min	0.000000e+00	-1.300000e+01	3.000000	0.000000	
25%	8.938958e+00	5.100000e+01	9.000000	106.000000	
50%	2.091528e+01	5.900000e+01	14.000000	129.000000	
75%	3.404250e+01	6.800000e+01	15.000000	162.000000	
max	4.800000e+01	2.980000e+02	15.000000	1748.000000	

	heart rate	mean BP	oxygen saturation	respiratory rate	\
count	1.057551e+06	1.007986e+06	1.065406e+06	1.066675e+06	
mean	8.697399e+01	7.831875e+01	9.678339e+01	1.920545e+01	
std	1.886481e+01	1.627105e+01	4.695637e+00	6.246538e+00	
min	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	
25%	7.400000e+01	6.700000e+01	9.600000e+01	1.500000e+01	
50%	8.500000e+01	7.600000e+01	9.800000e+01	1.900000e+01	
75%	9.900000e+01	8.700000e+01	1.000000e+02	2.300000e+01	
max	3.000000e+02	3.000000e+02	1.000000e+02	2.500000e+02	

	systolic BP	temperature	age	gender	height	\
count	1.015683e+06	321762.000000	1.461282e+06	1.461282e+06	455361.000000	
mean	1.197054e+02	36.986995	6.535809e+01	1.554603e+00	169.170224	
std	2.332119e+01	0.854016	1.663225e+01	4.970097e-01	14.552301	
min	0.000000e+00	24.700000	1.803403e+01	1.000000e+00	0.000000	
25%	1.030000e+02	36.444444	5.474257e+01	1.000000e+00	163.000000	
50%	1.170000e+02	37.000000	6.738069e+01	2.000000e+00	170.000000	
75%	1.340000e+02	37.500000	7.862746e+01	2.000000e+00	178.000000	
max	4.110000e+02	42.222222	9.000000e+01	2.000000e+00	445.000000	

	pH	weight	day	mortality	hour
count	116414.000000	1.309930e+06	1.461282e+06	1.461282e+06	1.461282e+06
mean	7.368808	8.280987e+01	1.439831e+00	1.505267e-01	2.132638e+01
std	0.086337	2.575886e+01	4.963666e-01	3.575871e-01	1.420751e+01
min	6.800000	0.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00

```
25%         7.320000  6.650000e+01  1.000000e+00  0.000000e+00  8.000000e+00
50%         7.380000  7.910000e+01  1.000000e+00  0.000000e+00  2.000000e+01
75%         7.420000  9.440000e+01  2.000000e+00  0.000000e+00  3.400000e+01
max         7.800000  9.312244e+02  2.000000e+00  1.000000e+00  4.700000e+01
```

The dataset consists of 21,139 unique ICU stays and 1,461,282 observations. All columns with the exception of 'hours', 'mortality' and 'day' have missing information. Looking at the maximum and minimum values it is possible to spot the presence of outliers (e.g. max glucose). Both missing data and outliers are very common in ICU databases and need to be taken into consideration before applying ML algorithms.

9.4 Variable Selection

In general, you should take into consideration the following criteria when deciding whether to include or exclude variables:

- adding more variables tends to decrease the sample size, because fewer patients are likely to have all of them collected at the same time;
- selecting a high number of variables might bias the dataset towards the selection of a specific group of patients whose characteristics required the measurement of those specific variables;
- variables should be independent with minimal correlation;
- the number of observations should be significantly higher than the number of variables, in order to avoid the curse of dimensionality.

Rejecting variables with an excessive number of missing values is usually a good rule of thumb. However, it might also lead to the reduction of predictive power and ability to detect significant differences. For these reasons, there should be a trade-off between the potential value of the variable in the model and the amount of data available. We already saw the amount of missing data for every column, but we still do not know how much information is missing at the patient level. In order to do so, we are going to aggregate data by ICU stay and look at the number of non-null values, using the 'groupby' function together with the 'mean' operator. This will give an indication of how many ICU stays have at least one observation for each variable.

Note that one patient might have multiple ICU stays. In this work, for the sake of simplicity, we will consider every ICU stay as an independent sample.

```
In [5]: print(data.groupby(['icustay']).mean().info(null_counts=1))
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 21139 entries, 200001.0 to 299995.0
Data columns (total 17 columns):
hours                21139 non-null float64
diastolic BP         19154 non-null float64
glasgow coma scale   10993 non-null float64
glucose              18745 non-null float64
heart rate           19282 non-null float64
mean BP              19123 non-null float64
```

```
oxygen saturation     19305 non-null float64
respiratory rate      19367 non-null float64
systolic BP           19154 non-null float64
temperature           18440 non-null float64
age                   21139 non-null float64
gender                21139 non-null float64
height                5370 non-null float64
pH                    15489 non-null float64
weight                18181 non-null float64
day                   21139 non-null float64
mortality             21139 non-null float64
dtypes: float64(17)
memory usage: 2.9 MB
None
```

Based on the previous information, some decisions are made:

- Height can be discarded due to the high amount of missing data;
- Weight and height are typically used in combination (body mass index), since individually they typically provide low predictive power. Therefore, weight was discarded;
- The other variables will be kept. Let us start analyzing time-variant variables and set aside age and gender for now:

```
In [6]: variables = ['diastolic BP', 'glasgow coma scale',
                     'glucose', 'heart rate', 'mean BP',
                     'oxygen saturation', 'respiratory rate', 'systolic BP',
                     'temperature', 'pH']
        variables_mort = variables.copy()
        variables_mort.append('mortality')
```

These decisions are specific for this case. Other criteria could have been used, for example, checking the correlation between all variables and excluding one variable out of every pair with high correlation in order to reduce redundancy.

9.5 Data Preprocessing

9.5.1 Outliers

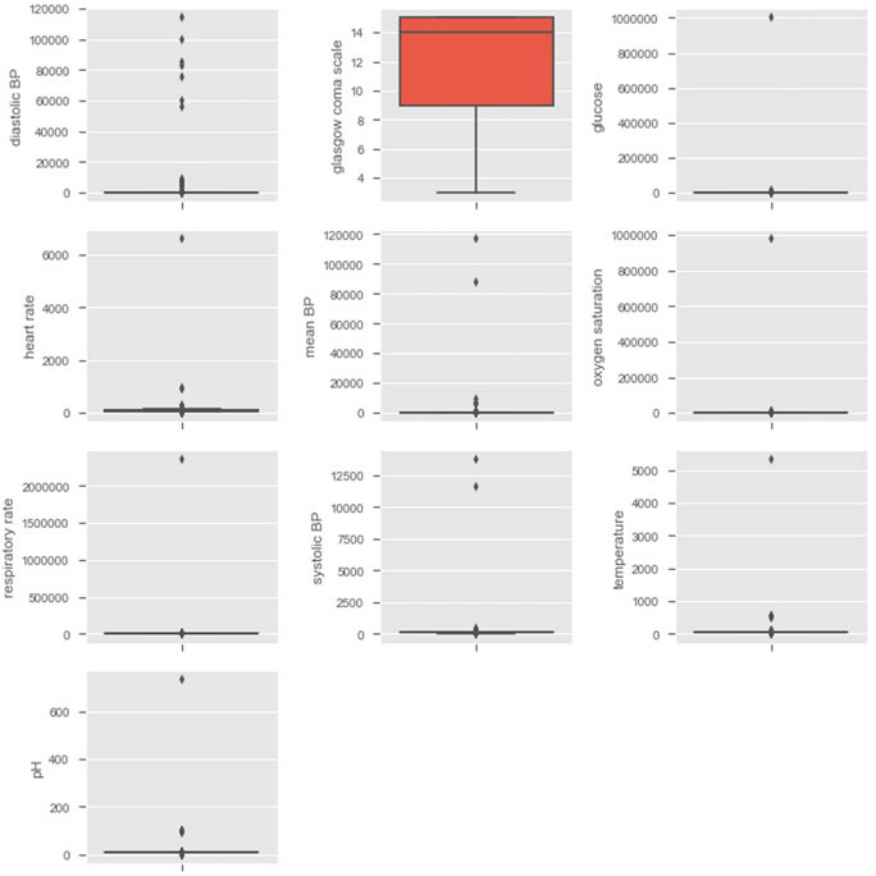
9.5.1.1 Data Visualization

We already saw that outliers are present in the dataset, but we need to take a closer look at data before deciding how to handle them. Using the 'seaborn' library and the 'boxplot' function we can easily create one boxplot for every variable. Seaborn is a visualization library based on matplotlib that provides a high-level interface for drawing statistical graphics.

```
In [7]: import seaborn as sns

        fig = plt.figure(figsize=(10,10))
        count = 0
        for variable in variables:
            count += 1
            plt.subplot(4, 3, count)
            ax = sns.boxplot(y=variable, data=data)

        fig.tight_layout()
        plt.show()
```



In some cases, the outliers are so deviant from the norm that it is not even possible to visualize the distribution of data (minimum, first quartile, median, third quartile, maximum) using boxplots. There are other plot types you can create to investigate the presence of outliers. Simply plotting all points using a scatter plot, or using violin plots are some of the options.

9.5.1.2 Exclusion

Ideally, we should keep extreme values related to the patients' poor health condition and exclude impossible values (such as negative temperature) and probable outliers (such as heart rate above 250 beats/min). In order to do so, values that fall outside boundaries defined by expert knowledge are excluded. This way, we avoid excluding extreme (but correct/possible) values.

```
In [8]: nulls_before = data.isnull().sum().sum()

        data.loc[data['diastolic BP']>300, 'diastolic BP'] = np.nan
        data.loc[data['glucose']>2000, 'glucose'] = np.nan
        data.loc[data['heart rate']>400, 'heart rate'] = np.nan
        data.loc[data['mean BP']>300, 'mean BP'] = np.nan
        data.loc[data['mean BP']<0, 'mean BP'] = np.nan
        data.loc[data['systolic BP']>10000,  'systolic BP'] = np.nan
        data.loc[data['temperature']>50, 'temperature'] = np.nan
        data.loc[data['temperature']<20, 'temperature'] = np.nan
        data.loc[data['pH']>7.8, 'pH'] = np.nan
        data.loc[data['pH']<6.8, 'pH'] = np.nan
        data.loc[data['respiratory rate']>300, 'respiratory rate'] = np.nan
        data.loc[data['oxygen saturation']>100, 'oxygen saturation'] = np.nan
        data.loc[data['oxygen saturation']<0, 'oxygen saturation'] = np.nan

        nulls_now = data.isnull().sum().sum()
        print('Number of observations removed: ' + str(nulls_now - nulls_before))
        print('Observations corresponding to outliers: ' + str(round((nulls_now -
        nulls_before)*100/data.shape[0],2)) + '%')
```

```
Number of observations removed: 7783
Observations corresponding to outliers: 0.53%
```

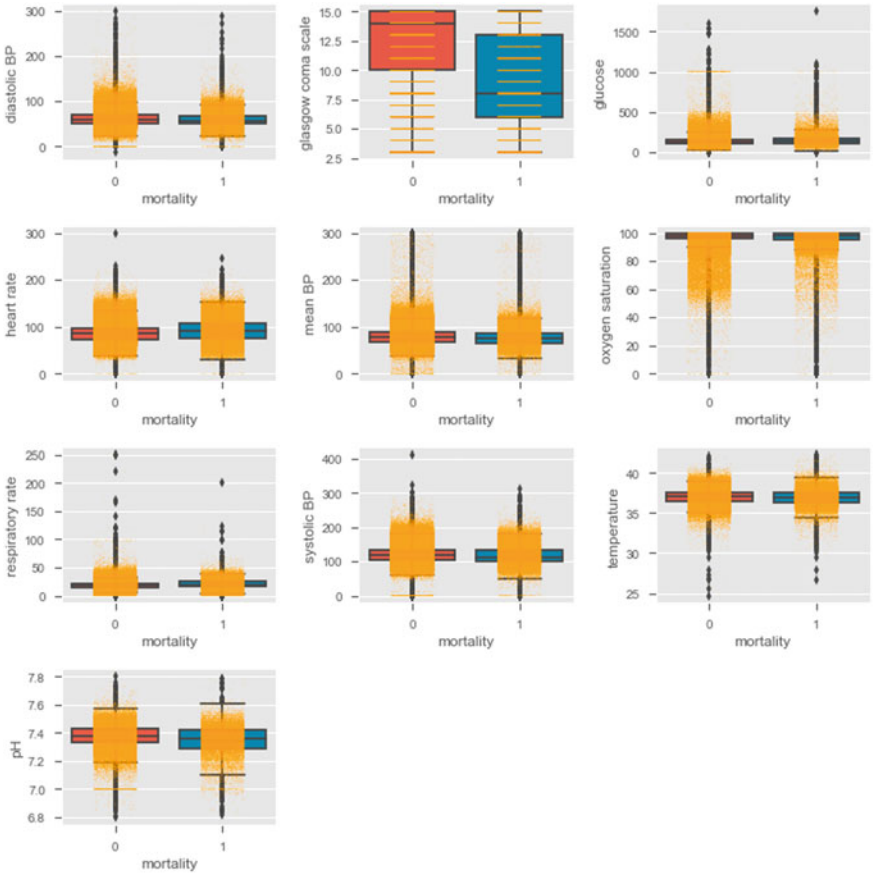
9.5.1.3 Data Visualization After Outliers Exclusion

The same code can be used to verify the data distribution after exclusion of outliers. The 'stripplot' function allows to visualize the underlying distribution and the number of observations. Setting x = 'mortality' shows the boxplots partitioned by outcome.

```
In [9]: fig = plt.figure(figsize=(10,10))
        count = 0
        for variable in variables:
            count += 1
            plt.subplot(4, 3, count)

            ax = sns.boxplot(x = 'mortality', y=variable, data=data)
            ax = sns.stripplot(x = 'mortality', y=variable, data=data, color="orange",
        jitter=0.2, size=0.5)

        fig.tight_layout()
        plt.show()
```



9.5.2 Aggregate Data by Hour

As mentioned before, the dataset contains information regarding the first 2 days in the ICU. Every observation is associated with a time stamp, indicating the number of hours elapsed between ICU admission and the time when the observation was collected (e.g., 0.63 h). To allow for ease of comparison, individual data is condensed into hourly observations by selecting the median value of the available observations within each hour. First, the 'floor' operator is applied in order to categorize the hours

in 48 bins (hour 0, hour 1, ..., hour 47). Then, the 'groupby' function with the 'median' operator is applied in order to get the median heart rate for each hour of each ICU stay:

```
In [10]: data['hour'] = data['hours'].apply(np.floor)

         # data goes until h = 48, change 48 to 47
         data.loc[data['hour'] == 48, 'hour'] = 47

         data_median_hour = data.groupby(['icustay','hour'])[variables_mort].median()
```

The 'groupby' will create as many indexes as groups defined. In order to facilitate the next operations, a single index is desirable. In the next example, the second index (column 'hour') is excluded and kept as a DataFrame column. Note that the first index corresponds to level 0 and second index to level 1. Therefore, in order to exclude the second index and keep it as a column, 'level' should be set to 1 and 'drop' to False.

```
In [11]: data_median_hour = data_median_hour.reset_index(level=(1), drop = False)
```

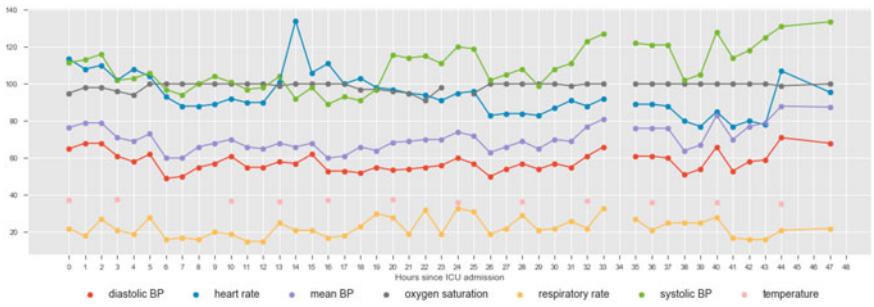
The next example shows the vital signs for a specific ICU stay (ID = 200001). Consecutive hourly observations are connected by line.

```
In [12]: vitals = ['diastolic BP', 'heart rate', 'mean BP', 'oxygen saturation',
                   'respiratory rate', 'systolic BP','temperature']
         ICUstayID = 200001.0

         fig, ax = plt.subplots(figsize=(20,6))

         # scatter plot
         for col in vitals:
             ax.scatter(data_median_hour.loc[ICUstayID,'hour'],data_median_hour.loc[ICUstayID,col])
             plt.legend(loc=9, bbox_to_anchor=(0.5, -0.1), ncol=len(vitals), prop={'size': 14})
             plt.xticks(np.arange(0, 49, step=1))
             plt.xlabel('Hours since ICU admission')

         # connect consecutive points by line
         for col in vitals:
             ax.plot(data_median_hour.loc[ICUstayID,'hour'], data_median_hour.loc[ICUstayID,col])
```



9.5.3 Select Minimum Number of Observations

We decided to keep all time-variant variables available. However, and as you can see in the previous example, since not all variables have a hourly sampling rate, a lot of information is missing (coded as NaN). In order to train ML algorithms it is important to decide how to handle the missing information. Two options are: to replace the missing information with some value or to exclude the missing information. In this work, we will avoid introducing bias resultant from replacing missing values with estimated values (which is not the same as saying that this is not a good option in some situations). Instead, we will focus on a complete case analysis, i.e., we will include in our analysis only those patients who have complete information.

Depending on how we will create the feature set, complete information can have different meanings. For example, if we want to use one observation for every hour, complete information is to have no missing data for every t = 0 to t = 47, which would lead to the exclusion of the majority of data. In order to reduce the size of the feature space, one common approach is to use only some portions of the time series. This is the strategy that will be followed in this work. Summary statistics, including the mean, maximum, minimum and standard deviation will be used to extract relevant information from the time series. In this case, it is important to define the minimum length of the time series before starting to select portions of it. One possible approach is to use all patients who have at least one observation per variable. Since, the summary statistics have little meaning if only one observation is available, a threshold of two observations is used.

In the following function, setting 'min_num_meas = 2' means that we are selecting ICU stays where each variable was recorded at least once at two different hours. Again, we are using the 'groupby' function to aggregate data by ICU stay, and the 'count' operator to count the number of observations for each variable. We then excluded ICU stays where some variable was recorded less than 2 times. Section 9.7 will show how to extract features from the time series.

```
In [13]: min_num_meas = 2

         def extr_min_num_meas(data_median_hour, min_num_meas):
             """ Select ICU stays where there are at least 'min_num_meas' observations
             and print the resulting DataFrame size"""
             data_count = data_median_hour.groupby(['icustay'])[variables_mort].count()

             for col in data_count:
                 data_count[col] = data_count[col].apply(lambda x: np.nan if x < min_num_meas
         else x)

             data_count = data_count.dropna(axis=0, how='any')
             print('Number of ICU stays: ' + str(data_count.shape[0]))
             print('Number of features: ' + str(data_count.shape[1]))
             unique_stays = data_count.index.unique()

             data_median_hour = data_median_hour.loc[unique_stays]

             return data_median_hour

         data_median_hour = extr_min_num_meas(data_median_hour, min_num_meas)
```

```
Number of ICU stays: 6931
Number of features: 11
```

It is always important to keep track of the size of data while making decisions about inclusion/exclusion criteria. We started with a database of around 60,000 ICU stays, imported a fraction of those that satisfied some criteria, in a total of 21,140 ICU stays, and are now looking at 6,931 ICU stays.

9.6 Data Analysis

9.6.1 Pairwise Ploting

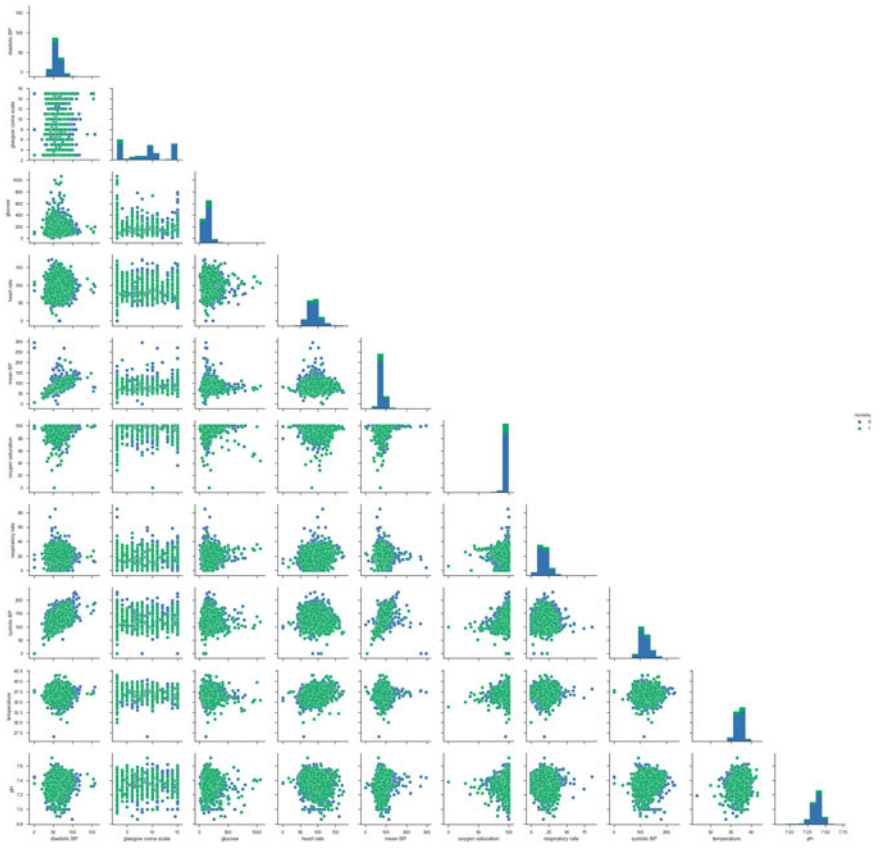
One of the most used techniques in exploratory data analysis is the pairs plot (also called scatterplot). This technique allows to see both the distribution of single variables as well as relationships between every two variables. It is easily implemented in Python using the 'seaborn' library. The next example shows how to plot the pairwise relationships between variables and the histograms of single variables partitioned by outcome (survival vs non-survival). The argument 'vars' is used to indicate the set of variables to plot and 'hue' to indicate the use of different markers for each level of the 'hue' variable. A subset of data is used: 'dropna(axis = 0, how = 'any')' excludes all rows containing missing information.

```
In [14]: import seaborn as sns

         sns.set(style="ticks")
         g = sns.pairplot(data_median_hour[variables_mort].dropna(axis=0, how='any'), vars =
         variables, hue = 'mortality')

         # hide the upper triangle
         for i, j in zip(*np.triu_indices_from(g.axes, 1)):
             g.axes[i, j].set_visible(False)
         plt.show()

         # change back to our preferred style
         plt.style.use('ggplot')
```



Unfortunately, this type of plot only allows to see relationships in a 2D space, which in most cases is not enough to find any patterns or trends. Nonetheless, it is still able to tell us important aspects of data; if not for showing promising directions for data analysis, to provide a means to check data's integrity. Things to highlight are:

- Hypoxic patients generally have lower SBPs;
- SBP correlates with MAP, which is a nice test of the data's integrity;
- Fever correlates with increasing tachycardia, also as expected.

9.6.2 Time Series Ploting

In order to investigate time trends, it is useful to visualize the mean HR partitioned by outcome from t = 0 to t = 47. In order to easily perform this task, the DataFrame needs to be restructured.

The next function takes as input a pandas DataFrame and the name of the variable and transposes/pivots the DataFrame in order to have columns corresponding to time (t = 0 to t = 47) and rows corresponding to ICU stays. If 'filldata' is set to 1, the function will fill missing information using the forward fill method, where NaNs are replaced by the value preceding it. In case no value is available for forward fill, NaNs are replaced by the next value in the time series. The function 'fillna' with the method argument set to 'ffill' and 'bfill', allows us to easily perform these two actions. If 'filldata' is set to 0 no missing data imputation is performed.

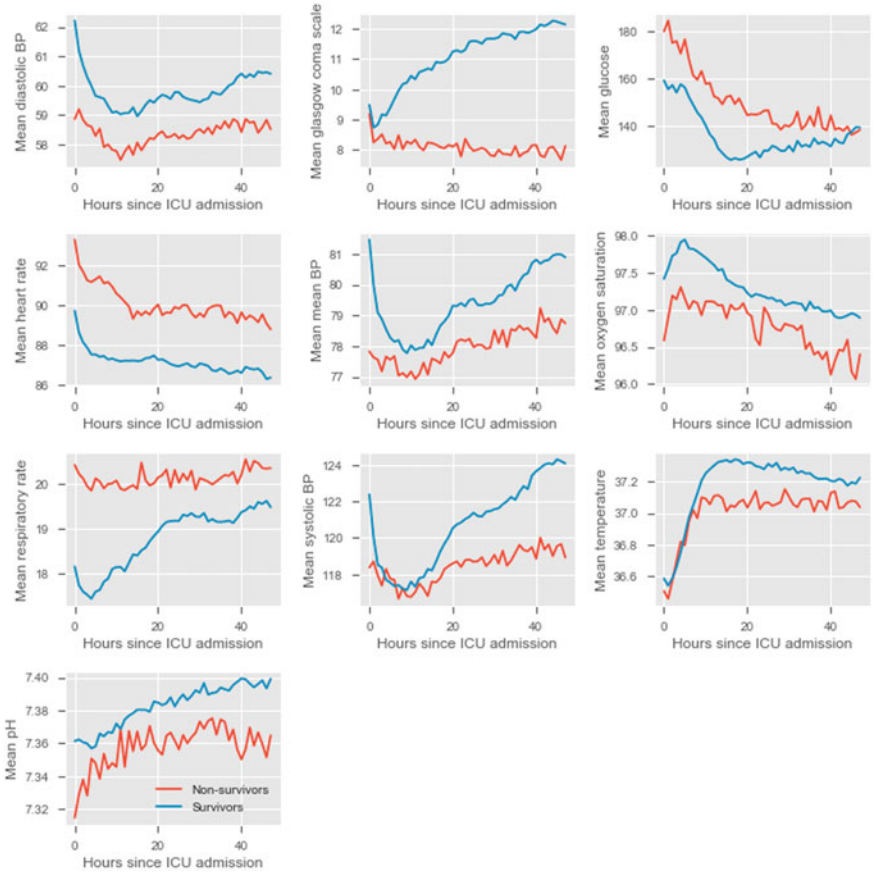
```
In [15]: def timeseries_data(data_median_hour, variable, filldata = 1):
             """Return matrix of time series data for clustering,
             with rows corresponding to unique observations (ICU stays) and
             columns corresponding to time since ICU admission"""
             data4clustering = data_median_hour.pivot(columns='hour', values=variable)
             if filldata == 1:
                 # first forward fill
                 data4clustering = data4clustering.fillna(method='ffill', axis=1)
                 # next backward fill
                 data4clustering = data4clustering.fillna(method='bfill', axis=1)
                 data4clustering = data4clustering.dropna(axis=0, how='all')
             return data4clustering
```

The next script plots the average HR for every variable in the dataset. At this point, 'filldata' is set to 0. In the section named Time Series Clustering of Part II of this tutorial, 'filldata' will be set to 1 in order to perform clustering of time series.

```
In [16]: fig = plt.figure(figsize=(10,10))
         count = 0
         for variable in variables:
             count += 1
             data4clustering = timeseries_data(data_median_hour, variable, filldata = 0)
             print('Plotting ' + str(data4clustering.count().sum()) + ' observations from ' +
         str(data4clustering.shape[0]) + ' ICU stays' + ' - ' + variable)
             class1 = data4clustering.loc[data[data['mortality']==1].index.unique()].mean()
             class0 = data4clustering.loc[data[data['mortality']==0].index.unique()].mean()
             plt.subplot(4, 3, count)
             plt.plot(class1)
             plt.plot(class0)
             plt.xlabel('Hours since ICU admission')
             plt.ylabel('Mean ' + variable)

         fig.tight_layout()
         plt.legend(['Non-survivors', 'Survivors'])
         plt.show()
```

```
Plotting 308913 observations from 6931 ICU stays - diastolic BP
Plotting 107073 observations from 6931 ICU stays - glasgow coma scale
Plotting 99885 observations from 6931 ICU stays - glucose
Plotting 310930 observations from 6931 ICU stays - heart rate
Plotting 306827 observations from 6931 ICU stays - mean BP
Plotting 302563 observations from 6931 ICU stays - oxygen saturation
Plotting 305363 observations from 6931 ICU stays - respiratory rate
Plotting 308942 observations from 6931 ICU stays - systolic BP
Plotting 138124 observations from 6931 ICU stays - temperature
Plotting 60375 observations from 6931 ICU stays - pH
```



The physiological deterioration or improvement over time is very different between survivors and non-survivors. While using the pairwise plot we could not see any differences between the groups, this type of plot reveals very clear differences. Several observations can be made:

- **Diastolic BP**
  - higher in the survival group;
  - rapidly decreasing during the first 10 h, especially in the survival group, and increasing at a lower rate thereafter;
- **Glasgow coma scale**
  - higher in the survival group, increasing over time;
  - steady around 8 in the non-survival group;
  - similar between both groups at admission, but diverging thereafter;
- **Glucose**
  - decreasing over time in both groups;

- **Heart rate**
  - lower in the survival group;
- **Mean BP** - similar to diastolic BP;
- **Oxygen saturation**
  - higher in the survival group;
  - low variation from t = 0 to t = 48 h;
- **Respiratory rate**
  - lower in the survival group, slowly increasing over time;
  - steady around 20 in the non-survival group;
- **Systolic BP**—similar to diastolic and mean BP;
- **Temperature**
  - low variation from t = 0 to t = 48 h;
  - slightly increasing during the first 10 h;
- **pH**
  - Increasing over time in both groups;
  - pH < 7.35 (associated with metabolic acidosis) during the first 10 h in the non-survival group.

Most of these graphs have fairly interesting trends, but we would not consider the oxygen saturation or temperature graphs to be clinically relevant.

9.7 Feature Construction

The next step before ML is to extract relevant features from the time series. As already mentioned, the complete time series could be used for ML, however, missing information would have to be filled or excluded. Also, using the complete time series would result in $48\text{h} \times 11\text{variables} = 528$ features, which would make the models difficult to interpret and could lead to overfitting. There is a simpler solution, which is to use only a portion of the information available, ideally the most relevant information for the prediction task.

Feature construction addresses the problem of finding the transformation of variables containing the greatest amount of useful information. In this chapter, simple operations will be used to construct/extract important features from the time series:

- Maximum
- Minimum
- Standard deviation
- Mean

Other summary statistics or time-series snapshots could have been used, for example, the median, time elapsed between maximum and minimum, time elapsed between baseline and maximum, difference between baseline and minimum, and others. Alternatively, other techniques can be used for dimensionality reduction, such as principal component analysis (PCA) and autoencoders described in the previous Chapter.

The maximum, minimum, standard deviation and mean summarize the worst, best, variation and average patient' condition from t = 0 to t = 47h. In the proposed exercises you will do this for each day separately, which will increase the dataset dimensionality but hopefully will allow the extraction of more useful information. Using the 'groupby' function to aggregate data by ICU stay, together with the 'max', 'min', 'std' and 'mean' operators, these features can be easily extracted:

```
In [17]: def feat_transf(data):
             data_max = data.groupby(['icustay'])[variables].max()
             data_max.columns = ['max ' + str(col) for col in data_max.columns]

             data_min = data.groupby(['icustay'])[variables].min()
             data_min.columns = ['min ' + str(col) for col in data_min.columns]

             data_sd = data.groupby(['icustay'])[variables].std()
             data_sd.columns = ['sd ' + str(col) for col in data_sd.columns]

             data_mean = data.groupby(['icustay'])[variables].mean()
             data_mean.columns = ['mean ' + str(col) for col in data_mean.columns]

             data_agg = pd.concat([data_min,data_max,data_sd,data_mean],axis=1)

             return data_agg

         data_transf = feat_transf(data_median_hour).dropna(axis=0)

         print('Extracted features: ')
         display.display(data_transf.columns)
         print('')
         print('Number of ICU stays: ' + str(data_transf.shape[0]))
         print('Number of features: ' + str(data_transf.shape[1]))
```

```
Extracted features:
```

```
Index(['min diastolic BP', 'min glasgow coma scale', 'min glucose',
       'min heart rate', 'min mean BP', 'min oxygen saturation',
       'min respiratory rate', 'min systolic BP', 'min temperature', 'min pH',
       'max diastolic BP', 'max glasgow coma scale', 'max glucose',
       'max heart rate', 'max mean BP', 'max oxygen saturation',
       'max respiratory rate', 'max systolic BP', 'max temperature', 'max pH',
       'sd diastolic BP', 'sd glasgow coma scale', 'sd glucose',
       'sd heart rate', 'sd mean BP', 'sd oxygen saturation',
       'sd respiratory rate', 'sd systolic BP', 'sd temperature', 'sd pH',
       'mean diastolic BP', 'mean glasgow coma scale', 'mean glucose',
       'mean heart rate', 'mean mean BP', 'mean oxygen saturation',
       'mean respiratory rate', 'mean systolic BP', 'mean temperature',
       'mean pH'],
      dtype='object')
```

```
Number of ICU stays: 6931
Number of features: 40
```

A DataFrame containing one row per ICU stay was obtained, where each column corresponds to one feature. We are one step closer to building the models. Next, we are going to add the time invariant information—age and gender—to the dataset.

```
In [18]: mortality = data.loc[data_transf.index]['mortality'].groupby(['icustay']).mean()
         age = data.loc[data_transf.index]['age'].groupby(['icustay']).mean()
         gender = data.loc[data_transf.index]['gender'].groupby(['icustay']).mean()

         data_transf_inv = pd.concat([data_transf, age, gender, mortality],
         axis=1).dropna(axis=0)
         print('Number of ICU stays: ' + str(data_transf_inv.shape[0]))
         print('Number of features: ' + str(data_transf_inv.shape[1]))
```

```
Number of ICU stays: 6931
Number of features: 43
```

9.8 Data Partitioning

In order to assess the performance of the models, data can be divided into training, test and validation sets as exemplified in Fig. 9.1. This is known as the holdout validation method. The training set is used to train/build the learning algorithm; the validation (or development) set is used to tune parameters, select features, and make other decisions regarding the learning algorithm and the test set is used to evaluate the performance of the algorithm, but not to make any decisions regarding the learning algorithm architecture or parameters.

For simplicity, data is divided into two sets, one for training and another for testing. Later, when performing feature selection, the training set will be divided into two sets, for training and validation.

Scikit-learn is the essential machine learning package in Python. It provides simple and efficient tools for data mining and data analysis. The next example shows how to use the 'train_test_split' function from 'sklearn' library to randomly assign observations to each set. The size of the sets can be controlled using the 'test_size' parameter, which defines the size of the test set and which in this case is set to 20%. When using the 'train_test_split' function, it is important to set the 'random_state' parameter so that later the same results can be reproduced.

```
In [23]: from sklearn.cross_validation import train_test_split

         # set the % of observations in the test set
         test_size = 0.2

         # Divide the data intro training and test sets
         X_train, X_test, y_train, y_test = train_test_split(data_transf_inv,
         data_transf_inv[['mortality']], test_size = test_size, random_state = 10)
```

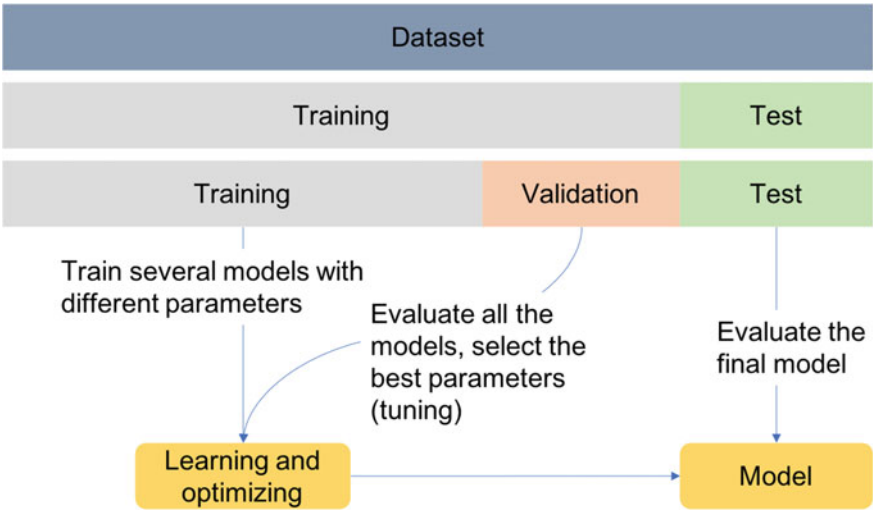


Fig. 9.1 Illustrative scheme of the holdout validation method

It is useful to create a function that prints the size of data in each set:

```
In [24]: def print_size(y_train, y_test):
             print(str(len(y_train[y_train['mortality']==1])) + '(' +
         str(round(len(y_train[y_train['mortality']==1])/len(y_train)*100,1)) + '%)' + ' non-
         survivors in training set')
             print(str(len(y_train[y_train['mortality']==0])) + '(' +
         str(round(len(y_train[y_train['mortality']==0])/len(y_train)*100,1)) + '%)' + '
         survivors in training set')
             print(str(len(y_test[y_test['mortality']==1])) + '(' +
         str(round(len(y_test[y_test['mortality']==1])/len(y_test)*100,1)) + '%)' + ' non-
         survivors in test set')
             print(str(len(y_test[y_test['mortality']==0])) + '(' +
         str(round(len(y_test[y_test['mortality']==0])/len(y_test)*100,1)) + '%)' + ' survivors
         in test set')
```

In cases where the data is highly imbalanced, it might be a good option to force an oversampling of the minority class, or an undersampling of the majority class so that the model is not biased towards the majority class. This should be performed on the training set, whereas the test set should maintain the class imbalance found on the original data, so that when evaluating the final model a true representation of data is used.

For the purpose of facilitating clustering interpretability, undersampling is used. However, as a general rule of thumb, and unless the dataset contains a huge number of observations, oversampling is preferred over undersampling because it allows keeping all the information in the training set. In any case, selecting learning algorithms that account for class imbalance might be a better choice.

The next example shows how to undersample the majority class, given a desired size of the minority class, controlled by the parameter 'perc_class1'. If 'perc_class1'>0, undersampling is performed in order to have a balanced training set. If 'perc_class1' = 0, no balancing is performed.

```
In [25]: # set the % of class 1 samples to be present in the training set.
         perc_class1 = 0.4

         print('Before balancing')
         print_size(y_train, y_test)

         if perc_class1 > 0:

             # Find the indices of class 0 and class 1 samples
             class0_indices = y_train[y_train['mortality'] == 0].index
             class1_indices = y_train[y_train['mortality'] == 1].index

             # Calculate the number of samples for the majority class (survivors)
             class0_size = round(np.int((len(y_train[y_train['mortality'] == 1])*(1 -
         perc_class1)) / perc_class1),0)

             # Set the random seed generator for reproducibility
             np.random.seed(10)

             # Random sample majority class indices
             random_indices = np.random.choice(class0_indices, class0_size, replace=False)

             # Concat class 0 with class 1 indices
             X_train = pd.concat([X_train.loc[random_indices],X_train.loc[class1_indices]])
             y_train = pd.concat([y_train.loc[random_indices],y_train.loc[class1_indices]])

             print('After balancing')
             print_size(y_train, y_test)

         # Exclude output from input data
         X_train = X_train.drop(columns = 'mortality')
         X_test = X_test.drop(columns = 'mortality')
```

```
Before balancing
941(17.0%) non--survivors in training set
4603(83.0%) survivors in training set
246(17.7%) non--survivors in test set
1141(82.3%) survivors in test set
After balancing
941(40.0%) non--survivors in training set
1411(60.0%) survivors in training set
246(17.7%) non--survivors in test set
1141(82.3%) survivors in test set
```

In the following "Part 2 - Unsupervised Learning with Clustering", clustering will be used to identify patterns in the dataset.

Acknowledgements This work was supported by the Portuguese Foundation for Science and Technology, through IDMEC, under LAETA, project UID/EMS/50022/2019 and LISBOA-01-0145-FEDER-031474 supported by Programa Operacional Regional de Lisboa by FEDER and FCT.