University of Coimbra

Department of Informatics Engineering

Masters Degree in Informatics Engineering

Pattern Recognition

Project Assignment
Final Delivery

Author:
João Ricardo Lourenço
2011151194

Author:
Joaquim Leitão
2011150072

May 31, 2015

CONTENTS

1 INTRODUCTION

This document addresses the work developed by *João Ricardo Lourenço* and *Joaquim Leitão* as part of the first and final delivery of the final assignment of the *Master's Degree in Informatics Engineering Pattern Recognition* course.

The proposed work is based on the *Kaggle Higgs Boson Machine Learning Challenge*[1]. This challenge aims to discover new particles and processes using *head-on* collisions of protons of extraordinarily high energy, exploring the potential of *machine learning methods*. Using simulated data with features characterising collision events previously detected and collected, the task of this challenge is to classify events into *"tau tau decay of a Higgs Boson"* or *"background"*.

Therefore, the assignment's task is to develop a classifier for a given set of features extracted from simulated *LHC collision events*, labelling them as being related to a *Higgs Boson* or a background event. To achieve this goal one has to consider a series of steps executed by a *Pattern Recognition System*:

1. The data collected must be *pre-processed*, filtering its invalid and missing values
2. A subset of the collected data should also be selected, removing irrelevant or less discriminative features of the data that skew the classifier's performance. This process is also called *Feature Selection*
3. In some cases it has also been proved to be effective to *map* the features into a low dimensionality feature space. This process is also called *Feature Extraction*
4. The classifier should be trained and tested

In the present document we will only address the first three steps of the presented list, as required for the assignment's first delivery report. The classifier system will be developed in the *MATLAB*[1] programming language, along with a series of *toolboxes* available for this language, namely *MATLAB's Statistical Pattern Recognition Toolbox*[2].

The remaining of this document is structured as follows: In chapter 2 we will perform a brief description of the dataset provided with the assignment and how we integrated it with the developed code for the system. In chapter 3 a series of methods and algorithms for dealing with missing and invalid values in the dataset. In chapter 4 a series of methods to select the most discriminative and less redundant features from the dataset will be addresses. In chapter 5 we will comment on methods to extract and combine features from the

1 `http://www.mathworks.com/products/matlab/index-b.html`

dataset. In chapter 7 we will perform a detailed explanation of the classifiers selected and implemented to classify the provided events. In chapter 8 we will present the *Graphical User Interface* developed in the course of our work. Finally, chapter 10 presents some additional considerations on the problem solving (data normalization, data set loading, etc), which is followed by concluding remarks in chapter 11.

2

DATASET DESCRIPTION

The dataset provided with the project's statement corresponds to a subset of the official training dataset of the mentioned challenge[1]. In it we can find 200000 events gathering information regarding the event's ID, 30 *floating-point* feature values and a label (integer value, either 1 or 2), discriminating the two types of events to classify.

In the full, official contest dataset, however, data about 250000 events was collected, with one additional feature: a weight column, used for result evaluation in the contest, and which will not be used in the given assignment.

More detailed information about the features of the dataset can be found in [3].

2.1 DATASET CONTENTS

From a brief analysis of the given dataset one can easily find two major problems that need to be considered during the development and execution of our work.

The first one is related to the number of events of each category/label present in the dataset: About 66% of the data are related to *background* events while only 34% are *"tau tau decay of a Higgs Boson"* events. This is an important detail to retain while developing the classifier, and the data must be balanced to prevent skewing of the classifier. In particular, given that we have a higher number of background events, a regular classifier will have a natural balance towards having higher accuracy in background event correct classification.

The second identified problem is related to missing values. The impossibility of accurately recording all the values during the simulations resulted in the registration of meaningless feature values (when it was possible to actually collect some data). In these situations, the related feature(s) assumed the value -999.0, outside the normal range for all variables, and the value was considered to be missing. In the next chapter we will present and briefly discuss the proposed and implemented methods to fix the *missing values* found in the dataset.

2.2 INTEGRATION WITH MATLAB

After being loaded into our programming environment (*MATLAB*) the information stored in the dataset is converted into a data structure compatible with the *STPR Toolbox*[2], so that we can use the functions available in that library[1]. Thus, the data was divided into two sets: the *feature set* and the *label set*.

The *feature set* was stored in the *X* field of the data structure while the *label set* was stored in the *y* field. The *IDs* of the events were discarded from the dataset since they provide no additional relevant information about the events being analysed.

1 To make this conversion easier and available at any moment in the code's execution we developed a small script in *MATLAB* to handle this conversion. The script can be found in the file *convert_to_sprt_data.m*

3

MISSING VALUES

Real world data isn't perfect. It doesn't always follow perfectly a desired mathematical model, it can be inaccurate, noisy or even nonexistent. Preprocessing collected data is a critical and remarkably demanding task common to all *machine learning* exercises.

One of the most common imperfections in data regards the absence of certain values, also known as missing values. There has been extensive research in this area, searching the best methods for computing accurate and plausible estimations for the missing values in any kind of dataset[1], minimising both the estimation error and the final results and conclusions drawn from a posterior data analysis.

As covered in chapter 2, *missing values* are more than present in the dataset provided: Quite a large number of *LHC collision events* were simulated, with some features being extracted from the simulated events. Therefore, one can expect a small error associated with the collection of these features, as well as moments in time where the sensors (or any other equipment/device used to acquire the data) are simply unable to collect any data at all[2].

In order to perform a proper filtering of the data, and hopefully improving our final classifier's performance, we implemented a series of techniques for dealing with such missing values, which we present in the sections of this chapter.

3.1 DISCARD TECHNIQUE

The *Discard Technique* was the first technique implemented to fix the missing values present in the data set and is the simplest. In the following sections we look at it from a more theoretical background and present our MATLAB implementation.

3.1.1 *Theory*

According to this technique all the events containing at least one invalid or missing value are discarded from the dataset. It has been proven to not introduce any bias if the data are missing at random[4].

1 Or, in some cases, proving that the best replacement is the discarding of instances that contain missing values.
2 Originating the presence of *missing data* in the dataset provided with the project's assignment

3.1.2 *MATLAB Implementation*

Our MATLAB implementation is present in the *fill_missing_values_discard.m* file, which is passed a full matrix of the data and replaces and removes any missing values. First, the rows with missing values are found, and then they are removed.

Listing 3.1: fill_missing_values_discard.m

```
function [ outdata ] = fill_missing_values_discard( indata )
    line_idx = sum(indata' == -999.0) == 0;
    outdata = indata(line_idx,:);
end
```

3.2 MEAN

Replacing all the invalid and missing values of any given dataset with its average value is one of the most popular methods for filling missing values. In the following sections we look at this technique from a more theoretical background and present our MATLAB implementation.

3.2.1 *Theory*

Missing Value imputation using the mean value stems from the idea that if the source distribution is random, then replacing missing values with the average of that feature will not introduce any bias[4].

3.2.2 *MATLAB Implementation*

In our scenario, we considered each feature individually, and computed its average value for all non-missing values. Then, all the invalid or missing values of that given feature were replaced by this average value. The code is implemented in the MATLAB file *fill_missing_values_mean.m* .

Listing 3.2: fill_missing_values_mean.m

```
function [ outdata ] = fill_missing_values_mean( indata )
    num_features = size(indata,2);
    outdata = indata;
    for i=1:num_features-1
            feat = indata(:,i);
            feat(feat == -999) = mean(feat(feat ~= -999));
            outdata(:,i) = feat;
    end
end
```

3.3 MEDIAN

The median method is similar to the mean method and intends to provide outlier robustness. In the following sections we look at it from a more theoretical background and present our MATLAB implementation.

3.3.1 *Theory*

While in the previous method we considered the average value of each feature in the dataset as an estimate for all the invalid or missing values for that given feature, in the *Median technique* instead of replacing the invalid and missing values with the feature's average value, we compute its median value. This method is more robust, i. e., it has more resistance to outliers in the original data.

3.3.2 *MATLAB Implementation*

The implementation is trivial and can be found in *fill_missing_values_median.m*.

Listing 3.3: fill_missing_values_median.m

```
function [ outdata ] = fill_missing_values_median( indata )
    num_features = size(indata,2);
    outdata = indata;
    for i=1:num_features-1
            feat = indata(:,i);
            feat(feat == -999) = median(feat(feat ~= -999));
            outdata(:,i) = feat;
    end
end
```

3.4 K-NEAREST NEIGHBOURS

The KNN method is so widely used that MATLAB provides us with a thorough implementation for missing value imputation. In the following sections we look at it from a more theoretical background and present our MATLAB implementation.

3.4.1 *Theory*

While the *K-Nearest Neighbours* algorithm is in its nature a classification algorithm, it can be adapted to serve as a missing value imputation method.

If one uses it as a classification method, for each instance, we locate the K nearest instances and use their label to "vote" the label of the instance to be labeled. The distance metric, typically euclidean, can be any other, such as the *Mahalanobis Distance*.

The same principle can be applied to fill in missing values[5]. If a given feature is missing for an instance, we can use the remaining features to find the K closest instances (according to a given distance metric) and use their value for that feature to estimate the missing value for the current instance. While this can be done using any estimation method, one typically uses a weighted mean where weights are inversely proportional to the distances of the points. In summary, this method finds the K closest instances (using non-missing features) and averages their values for the missing feature in the original instance.

3.4.2 *MATLAB Implementation*

MATLAB already implements this method for us in configurable fashion. As such, our implementation is trivial and can be seen in *fill_missing_values_knn.m*.

Listing 3.4: fill_missing_values_knn.m

```
function outdata=fill_missing_values_knn(indata,k_neighbours,distance_method )
    indata(indata == -999) = NaN;
    outdata = knnimpute(indata, k_neighbours, 'Distance', distance_method);
end
```

3.5 OBSERVATION-DESIGN MATRIX IMPUTATION

There are a set of techniques which we implemented that use the same concept and only change a couple of steps. We call these techniques "Observation-Design matrix imputation techniques". In particular we implemented Multivariate Regression and Neural Network estimation for the imputation.

3.5.1 *Theory*

The general idea of Observation-Design matrix imputation is that a model m will be built from a subset S of the dataset containing training data (A *Design Matrix*) from features without any missing values, and target data (extracted from *an Observation Matrix*) corresponding to features with missing values.

Consider the following dataset M, comprised of 7 instances and 5 features. For convenience, we split features into the $G1$ group (no missing values) and the $G2$ group (missing values).

$$M = \begin{array}{c} \\ E1 \\ E2 \\ E3 \\ E4 \\ E5 \\ E6 \\ E7 \end{array} \begin{array}{c} \begin{array}{ccccc} G1 & G1 & G1 & G2 & G2 \end{array} \\ \begin{bmatrix} 46.463 & 2.545 & 250.65 & NaN & 26.429 \\ 59.414 & 3.533 & 222.73 & 2.157 & NaN \\ 20.47 & 3.654 & 251.76 & NaN & 30.456 \\ 21.554 & 4.234 & 240.58 & 3.254 & 35.458 \\ 22.632 & 5.123 & 250.47 & 2.894 & 27.254 \\ 19.458 & 4.898 & 265.84 & 3.454 & 36.56 \\ 19.758 & 4.498 & 255.84 & 2.454 & 40.56 \end{bmatrix} \end{array}$$

Features 1, 2 and 3 are placed in the first group since they do not possess any missing value (*NaN*), while features 4 and 5 (each with two missing values) are placed in the second group.

The method works by looking only at instances where there are no missing values and creating a *Design Magtrix (DM)* and an *Observation Matrix (OM)*. The *design matrix* will contain all the features belonging to the first group, while the *observation matrix* contains all the features belonging to the second.

Applying these notions to our example, we temporarily discard instances $E1$, $E2$ and $E3$ (which had missing values), obtaining the matrix DM by selecting the first three columns (that is, the first three features, belonging to the first feature group, *G1*) of the four last instances. The matrix OM is obtained by considering the same events as in DM, but only selecting the last two columns (that is, the last two features, belonging to the second feature group, *G2*).

$$DM = \begin{array}{c} \begin{array}{ccc} G1 & G1 & G1 \end{array} \\ \begin{bmatrix} 21.554 & 4.234 & 240.58 \\ 22.632 & 5.123 & 250.47 \\ 19.458 & 4.898 & 265.84 \\ 19.758 & 4.498 & 255.84 \end{bmatrix} \end{array}$$

The Design Matrix, then, contains the values belonging to features where there are no missing values, and which can be used to perform the estimation of any feature where values are missing. The Observation Matrix is given by:

$$OM = \begin{array}{c} \begin{array}{cc} G2 & G2 \end{array} \\ \begin{bmatrix} 3.254 & 35.458 \\ 2.894 & 27.254 \\ 3.454 & 36.56 \\ 2.454 & 40.56 \end{bmatrix} \end{array}$$

If we now take each column OM_i of OM ($n \times k$, $i = 1, \ldots, k$), one by one, we can create models $m_1, \ldots, m_k$ such that[3]

$$m_i(DM_j) = OM_i, \quad i = 1, \ldots, k \quad j = 1, \ldots n$$

3 We use notation DM_j to represent a line in the design matrix. Note that DM_j is $1 \times p$, where p is the number of features without missing values, whereas OM_i is $n \times k$, where n is the number of instances and k is the number of features with missing values.

(Which is to say that feature i should be perfectly mapped to features $1, \ldots, p$ by model m_i)

In general, however, such perfect models don't exist, so approximations are found using any technique. We use two techniques in our work: Multivariate Regression and Neural Networks.

$$m_i(DM_j) \approx OM_i, \quad i = 1, \ldots, k \quad j = 1, \ldots n$$

With each of the k models built, they can now be used to extrapolate their values to missing data, assuming no over-fitting took place. That is to say that each instance E_i ($i = 1, \ldots, n$) with missing value in feature i can have its missing value replaced by the model approximation[4]:

$$E_{ji} = m_i(E_j)$$

Any method of finding and computing the model m_i can be used, and we now present two techniques that were implemented.

3.5.1.1 *Multivariate Regression*

The Multivariate Regression builds model m_i by performing a multivariate regression. It computes the coefficient estimates $\beta = [\beta_1, ..., \beta_p]^T$ such that

$$DM \times \beta \approx OM_i \tag{1}$$

Once we have the β values we can easily compute the estimations for any event originally discarded in the algorithm: We identify the event for which we intend to estimate the missing values of the features, collect the values of its its *G1* features and simply multiply them by the β vector computed, obtaining the estimations for all the features in the *G2* group. Then we simply select the estimations for the values missing.

Again, applying this to the given example, to compute an estimation for the $4-th$ feature of the event $E1$ we first need to obtain the β vector by finding the least squares solution to the system

$$DM \times \beta = OM_1 \tag{2}$$

and then obtaining estimations for[5] $M(1, 4:5)$ by computing:

$$M(1, 1:3) \times \beta \tag{3}$$

4 We use E_{ji} to note the value of feature i on instance j, and E_j the row vector containing values for features in instance j for features without missing values.

5 We use matlab index notation to note subsets of matrices. In particular, for the original matrix M. In other words, $M(i, j:k)$ means the vector containing the elements in the $i-th$ row of matrix M and columns j to k.

3.5.1.2 *Neural Network*

The Neural Network approach builds a Neural Network with p inputs and 1 output. It is trained with the Design Matrix to output the values in the Observation Matrix of the corresponding lines. Any neural network can be used, but we used a cascade forward network with configurable parameters. Further networks can be added at a point later in time.

3.5.2 *MATLAB Implementation*

The general form of the algorithm, where DM and OM are found, can be found in *fill_missing_values_design.m*. This method then feeds the DM and OM to the two specific implementations. This portion of the code can be seen below:

Listing 3.5: fill_missing_values_design.m portion related to DM and OM

```
function outdata = fill_missing_values_design(indata, method,varargin)
% FILL_MISSING_VALUES_DESIGN Replaces missing values with estimations
% computed using either a multivariate regression or a neural network.
%   indata is a matrix MxN, with M the number of events and N the number of
%          features of each event, containing missing or invalid values
%          (represented by the value -999.0)
%   method is the desired method to used when computing the estimations for
%          the missing values
    outdata = indata;

    %Get cols without missing values
    indexes_without_missing = features_without_missing_values(indata);
    %Get cols with missing values
    indexes_missing = features_with_missing_values(indata);

    %Select only the columns without missing data
    design_matrix = indata(:,indexes_without_missing);

    for j=indexes_missing

        %For each collumn with missing values we have to determine which
        %lines have missing values and which lines have not, discarding the
        %lines with missing values
        current_lines_missing = find(indata(:,j)==-999);
        current_design_matrix = design_matrix;
        current_design_matrix(current_lines_missing,:) = [];
        current_observation_matrix = indata(:,j);
        current_observation_matrix(current_lines_missing) = [];

        % [ METHOD SPECIFIC CODE HERE ]
    end
end
```

The method specific code is as follows:

Listing 3.6: fill_missing_values_design.m portion related to method specific code

```
if strcmp(method,'mvregress')
```

```
            [beta,~,~,~,~] = mvregress(current_design_matrix, \
            current_observation_matrix);
            outdata(current_lines_missing, j) = indata(current_lines_missing, \
            indexes_without_missing) * beta;
        elseif strcmp(method,'neuralnetwork')
            outdata(current_lines_missing, j) = \
            fill_missing_values_neuralnetwork(current_design_matrix, \
            current_observation_matrix, indata(current_lines_missing, \
            indexes_without_missing), varargin{:});
        end
```

The fill_missing_values_neuralnetwork was developed because different neural networks can be used in the future – it allows for portability. Its code is shown below:

Listing 3.7: fill_missing_values_neuralnetwwork.m

```
function [ out ] = fill_missing_values_neuralnetwork( current_design_matrix, \
current_observation_matrix, indata, varargin )
            method = varargin(1);

            if strcmp(method, 'cascadeforward')
                [~,neuronsPerLayer,hiddenLayers,maxEpochs,goal, \
                max_fail,performanceFnc,learningFnc] = \
                args_with_default_values(varargin,[],30,1,100,1E-5,50, \
                'mse','trainlm');
                hLayers = repmat(neuronsPerLayer,1,hiddenLayers);
                net = cascadeforwardnet(hLayers,learningFnc);
                net.trainParam.epochs = maxEpochs;
                net.trainParam.goal = goal;
                net.performFcn = performanceFnc;
                net.trainParam.max_fail = max_fail;
                net=train(net, current_design_matrix', current_observation_matrix');
                out = sim(net, indata')';
            end
end
```

In summary, the observation-design matrix imputation method implementation builds the observation and design matrixes, as shown in the theoretical background section, and then uses either the MATLAB mvregress method or builds a neural network using the Neural Network toolbox.

4

FEATURE SELECTION

Feature Selection can be described as the process of analysing a set of features in a given dataset, determining which are more relevant and helpful to distinguish the elements in that dataset, discarding all the remainder of the features, considered irrelevant to the classification process, reducing the problem's *feature space*.

This definition is based on the assumption that the datasets used in these applications contain *redundant* or *irrelevant features*, which do not provide extra, new or useful information about the problem.

In fact, by reducing the problem's *feature space* we are also reducing the risk of experiencing a popular phenomena among *machine learning* experts: The *"Curse of Dimensionality"*. This is a problem that usually arises when analysing data with a lot of features (typically hundreds or thousands of features), due to lack of scalability of the classifier algorithm, and that are not present in low-dimensional feature spaces.

Therefore, by removing redundant and irrelevant features we can improve our classifiers' and predictors' performance, not only in terms of correct classifications or predictions, but also in terms of execution time and resources reserved to execute those operations.

In the present chapter we will present the methods implemented so far to reduce the number of features to analyse in our dataset, aiming to select the most relevant features that allow us to distinguish the different simulated events with a minimum error.

4.1 FILTER METHODS

In their essence, *filter methods* are a class of *feature selection* methods independent from the classifier being used, attempting to identify and select in any given dataset the features with a higher discriminative power. To achieve this, *filter methods* rank each feature according to a given univariate metric, which characterises the method being applied, selecting the features with the highest ranking.

However, these methods often assume an independence between features of the same dataset, resulting in the selection of redundant variables with high metric values. This is, obviously, undesirable for this type of operation, since our goal is exactly the opposite: To eliminate redundant features, gathering only relevant, non-redundant features that enable us to produce an accurate classification of the elements in the given dataset.

4.1.1 *Fisher Method*

The *Fisher* method aims to identify and select the features with higher discriminative power by computing the *Fisher Score* for each feature in the dataset, selecting the top K features, with K being the desired number of relevant features to consider in the dataset.

4.1.1.1 *Theory*

Assuming the data in our dataset can only be classified in one of two possible classes, the *Fisher Score* for each feature i in the dataset will be computed as follows:

$$F(i) = \frac{(\mu_{w_1}(i) - \mu_{w_2}(i))^2}{\sigma_{w_1}^2 + \sigma_{w_2}^2} \tag{4}$$

where $\mu_{w_j}(i)$ refers to the average feature value for elements of class j, considering only their values for feature i and σ_{w_j} refers to the standard deviation of the feature values of the elements of class j.

In our classification process we are interested in finding compact features (low variance) with very far apart means, which will separate the elements of each class in the dataset, allowing for a better classification. Therefore, given the *Fisher Scores* for the features in the dataset we select the ones with higher value, corresponding to the more easily separable features.

4.1.1.2 *MATLAB Implementation*

The implementation of this *Feature Selection* method can be found in the file *fisher.m*. As previously described, in this algorithm's implementation we mainly compute the *Fisher Score* for each feature in the dataset, selecting the top K features.

In the listing that follows we present the algorithm implementation in *MATLAB*.

Listing 4.1: fisher.m

```matlab
function [ data, indexes ] = fisher(data, target_number_of_features)
%FIHSER Implementation of the Fisher Filter method for feature selection
%   data is a stprtool data type, containing the features/observed data and
%        the respective classes, in the format (data.X, data.y)
%   target_number_of_features is the desired number of features to select

    X = data.X';
    classes = data.y;
    number_features = size(X, 2);
    scores = zeros(1,number_features);
    classes_unique = unique(classes);

    for i=1:number_features
        % Rank each feature according to a given metric -- Fisher Score
        mean_feature_classes = 0;
        std_squared_feature_classes = 0;

        %Compute the mean and std for each class, for this feature
        for k=1:length(classes_unique)
           j = classes_unique(k);
           indexes_current_class = find(classes==j);
           if (k == 1)%First class
               mean_feature_classes = mean(X(indexes_current_class,i));
           else
               mean_feature_classes = mean_feature_classes -
                                      mean(X(indexes_current_class, i));
           end
           std_squared_feature_classes = std_squared_feature_classes +
                                         (std(X(indexes_current_class,i))^2);
        end

        %Compute the fisher score
        scores(i) = (mean_feature_classes^2) / std_squared_feature_classes;
    end

    %Sort the scores in descending order (good features first) and return
    %their original indexes
    [~, indexes] = sort(scores, 'descend');

     indexes = indexes(1:target_number_of_features);
     data.X = X(:,indexes)';
end
```

4.1.2 *Kruskal-Wallis*

4.1.2.1 *Theory*

The *Kruskal-Wallis Test* is a non-parametric method that tests if two or more separate sets of samples belong to the same distribution, by determining if they have the same median, under a certain level of confidence.

Kruskal-Wallis is often used in *One-Way ANOVA Tests*, to test the null hypothesis that independent samples from two or more groups come from distributions with equal medians, returning a *p-value* for the given test, containing the probability of the given groups coming from the same distribution.

4.1.2.2 *MATLAB Implementation*

In our application context, we are interested in determining the features that follow different distributions when compared to the distribution of the labels of the events, that is, we want to find features with discriminative information regarding the class of the events collected. In other words, we are interested in finding the features which, when subjected to the *Kruskal-Wallis Test* result in small *p-values*.

Thus, our implementation of this algorithm can be summarised to the computation of the *Kruskal-Wallis Test* for each feature in the dataset, selecting the desired number of features with smaller *p-values*. Our implementation of this method was highly based on this method's implementation available at `http://featureselection.asu.edu/software.php`, and can be found in the file *fsKruskalWallis.m*.

Listing 4.2: fsKruskalWallis.m

```
function [ data, indexes ] = fsKruskalWallis(data, target_number_of_features)
%FSKRUSKALWALLIS Implementation of the Kruskal-Wallis test for feature
%selection
%Taken and adapted from http://featureselection.asu.edu/software.php
%   data is the higgins data (features of all the entries in the dataset)
%   target_number_of_features is the desired number of features to select

    X = data.X';
    Y = data.y;
    [~, n] = size(X);
    W = zeros(n,1);

    for i=1:n
        W(i) = -kruskalwallis(vertcat(X(:,i)', Y'),{},'off');
    end

    [~, fList] = sort(W, 'descend');

    indexes = fList(1:target_number_of_features);
    data.X = X(:,indexes)';
end
```

4.1.3 *Area Under Curve*

This method consists in, for each feature in the dataset, computing the area under the *Receiver Operating Characteristic Curve (ROC Curve)*, selecting the features with higher values, according to this metric.

4.1.3.1 *Theory*

The *Receiver Operating Characteristic Curve* consists in a graphical representation of the performance of a given classifier while its decision threshold is being changed. In this representation we plot the true positive rate (also called *sensitivity*) and the false positive rate (also called *fall-out*, computed as $1 - specificity$) computed for various threshold values.

In the figure that follows we present an example of a *ROC Curve*:

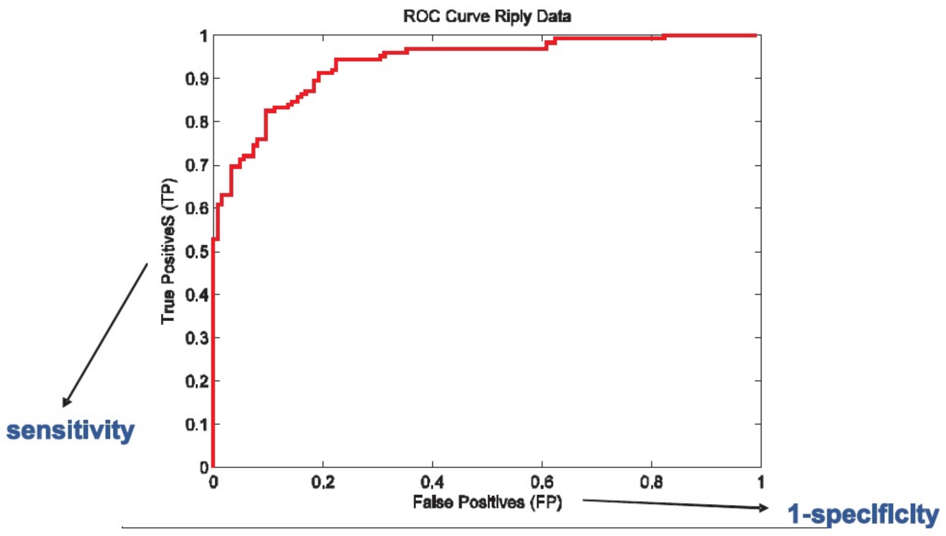


Figure 1: ROC Curve example. Taken from [6]

The *Area Under Curve (AUC) Method* presented in this chapter aims to compute the *ROC Curve* for each feature in the dataset, computing its area underneath and selecting the features with higher *Area Under the ROC Curve*. Features with a corresponding high *AUC* will have an higher number of true positive (and consequently smaller number of false positives), meaning a good linear classification performance.

4.1.3.2 *MATLAB Implementation*

In the file *area_under_curve.m* we can find our implementation of the mentioned algorithm in *MATLAB*, also presented in the listing that follows.

Listing 4.3: area_under_curve.m

```matlab
function [data, indexes] = area_under_curve(data, target_number_of_features)
%AREA_UNDER_CURVE Implementation of the Area Under Curve filter for feature
%selection
%   data is the higgins data (features of all the entries in the dataset)
%   target_number_of_features is the desired number of features to select

    X = data.X';
    Y = data.y;
    number_features = size(X, 2);
    auc_scores = zeros(1, number_features);

    for i=1:number_features
        [~,~,~,auc] = perfcurve(Y, X(:, i), 1);
        auc_scores(i) = auc;
    end

    [~, indexes] = sort(auc_scores, 'descend');
    indexes = indexes(1:target_number_of_features);
    data.X = X(:,indexes)';
end
```

As we can see, the implementation of this method is very simple, relying in *MATLAB's perfcurve* function, which computes the *ROC Curve* and the area underneath that *ROC Curve* for a given classifier's output.

Thus, by simply computing the *AUC* for each feature in the dataset, considering the correspondent events' labels as the classifier output, we can select the features with higher *AUC* value, which will be the features with higher relationship with the events' classes, that is, the ones we are interested in.

4.1.4 *Maximise Relevance Minimise Redundancy*

The *Maximise Relevance Minimise Redundancy (mRMR)* method for *Feature Selection* selects the features of interest in a given dataset, according to a *multi-objective optimisation* problem formulation, attempting to find a set of features that maximises the relevance to the final classification results, while minimising the redundancy among the selected features.

4.1.4.1 *Theory*

The problem can be defined as:

$$max \quad V_F; \quad VF = \left(\frac{1}{|S|} \sum_{i \in S} I(i,h) \right), \quad min \quad W_C; \quad W_C = \left(\frac{1}{|S|^2} \sum_{i,j \in S} |I(i,j)| \right) \tag{5}$$

where S is the set of features in the dataset, $c(i,j)$ is the correlation between features i and j and $I(i,h)$ is the mutual information between feature i and j, computed as follows:

$$I(f1,f2) = \sum_{i,j} p\left(f1, f2_j\right) log \left(\frac{p(f1_i, f2_j)}{p(f1_i, f2_i)} \right) \tag{6}$$

Among the literature there are two popular techniques to combine the objective functions. They are:

- *Additive Combination:* $J = max_{i \in \Omega_S} \left(I(i,h) - \frac{1}{|S|} \sum_{j \in S} |I(i,j)| \right)$
- *Multiplicative Combination:* $J = max_{i \in \Omega_S} \left(\frac{I(i,h)}{\frac{1}{|S|} \sum_{j \in S} |I(i,j)|} \right)$

4.1.4.2 *MATLAB Implementation*

In our work we made use of implementations of the *mRMR* algorithm using the two mentioned techniques for combination of the objective functions.

The *MATLAB* implementations used were developed by *Hanchuan Peng*, are available for download available at `http://www.mathworks.com/matlabcentral/fileexchange/14608-mrmr-feature-selection--using-mutual-information-computation-`. These implementations are also included in the source code provided with this document, namely in the files *mrmr_aditive.m* and *mrmr_multiplicative.m*.

In the listing that follows we present the two *MATLAB* implementations of the algorithm.

Listing 4.4: mrmr_aditive.m

```
function [ data, indexes ] = mrmr_aditive( data, target_number_of_features )
%mRMR ADITIVE The MID scheme of minimum redundancy maximal relevance (mRMR)
%              feature selection
%Taken and adapted from
%http://www.mathworks.com/matlabcentral/fileexchange/14608-mrmr-feature-
%selection--using-mutual-information-computation-
%
% Parameters:
%   data is a stprtool data type, containing the features/observed data and
%        the respective classes, in the format (data.X, data.y)
%   target_number_of_features is the desired number of features to select
%
% The data parameter is converted in the old "d" and "f" parameters while
% "target_number_of_features" is directly converted into the old "K"
% parameter (see next paragraph with Original Parameters information)
%
% Original Parameters:
%  d - a N*M matrix, indicating N samples, each having M dimensions.
%      Must be integers.
%  f - a N*1 matrix (vector), indicating the class/category of the N samples.
%      Must be categorical.
%  K - the number of features need to be selected
%
% Note: This version only supports discretized data, thus if you have continuous
%       data in "d", you will need to discretize them first. This function needs
%       the mutual information computation toolbox written by the same author,
%       downloadable at the Matlab source codes exchange site.
%       Also There are multiple newer versions on the Hanchuan Peng's web site
%       (http://research.janelia.org/peng/proj/mRMR/index.htm).
%
% More information can be found in the following papers.
%
% H. Peng, F. Long, and C. Ding,
%   "Feature selection based on mutual information: criteria
%    of max-dependency, max-relevance, and min-redundancy,"
%   IEEE Transactions on Pattern Analysis and Machine Intelligence,
%   Vol. 27, No. 8, pp.1226-1238, 2005.
%
% C. Ding, and H. Peng,
%   "Minimum redundancy feature selection from microarray gene
%    expression data,"
%    Journal of Bioinformatics and Computational Biology,
%   Vol. 3, No. 2, pp.185-205, 2005.
%
% C. Ding, and H. Peng,
%   "Minimum redundancy feature selection from microarray gene
%    expression data,"
%   Proc. 2nd IEEE Computational Systems Bioinformatics Conference (CSB 2003),
%   pp.523-528, Stanford, CA, Aug, 2003.
%
%
% By Hanchuan Peng (hanchuan.peng@gmail.com)
% April 16, 2003
%
    d = data.X';
    f = data.y;
    K = target_number_of_features;
    bdisp=0;
    nd = size(d,2);
    nc = size(d,1);

    t1=cputime;
    for i=1:nd,
       t(i) = mutualinfo(d(:,i), f);
    end;

    [tmp, idxs] = sort(-t);
    fea_base = idxs(1:K);
    fea(1) = idxs(1);
    KMAX = min(1000,nd); %500
    idxleft = idxs(2:KMAX);
    k=1;

    for k=2:K,
       t1=cputime;
       ncand = length(idxleft);
       curlastfea = length(fea);
       for i=1:ncand,
          t_mi(i) = mutualinfo(d(:,idxleft(i)), f);
          mi_array(idxleft(i),curlastfea) = getmultimi(d(:,fea(curlastfea)),
                                                       d(:,idxleft(i)));
          c_mi(i) = mean(mi_array(idxleft(i), :));
       end;

       [tmp, fea(k)] = max(t_mi(1:ncand) - c_mi(1:ncand));

       tmpidx = fea(k); fea(k) = idxleft(tmpidx); idxleft(tmpidx) = [];
    end;
    indexes = fea;
    data.X = data.X(indexes,:);

    return;
end

%======================================
function c = getmultimi(da, dt)
    for i=1:size(da,2),
       c(i) = mutualinfo(da(:,i), dt);
    end;
end
```

Listing 4.5: mrmr_multiplicative.m

```
function [ data, indexes ] = mrmr_multiplicative( data, target_number_of_features )
%mRMR MULTIPLICATIVE The MIQ scheme of minimum redundancy maximal relevance (mRMR)
%                    feature selection
```

```
%Taken and adapted from http://www.mathworks.com/matlabcentral/fileexchange/14608-
%mrmr-feature-selection--using-mutual-information-computation-
%
% Parameters:
%   data is a stprtool data type, containing the features/observed data and
%        the respective classes, in the format (data.X, data.y)
%   target_number_of_features is the desired number of features to select
%
% The data parameter is converted in the old "d" and "f" parameters while
% "target_number_of_features" is directly converted into the old "K"
% parameter (see next paragraph with Original Parameters information)
%
% Original Parameters:
%  d - a N*M matrix, indicating N samples, each having M dimensions.
%      Must be integers.
%  f - a N*1 matrix (vector), indicating the class/category of the N samples.
%      Must be categorical.
%  K - the number of features need to be selected
%
% Note: This version only supports discretized data, thus if you have continuous
%       data in "d", you will need to discretize them first. This function needs
%       the mutual information computation toolbox written by the same author,
%       downloadable at the Matlab source codes exchange site.
%       Also There are multiple newer versions on the Hanchuan Peng's web site
%       (http://research.janelia.org/peng/proj/mRMR/index.htm).
%
% More information can be found in the following papers.
%
% H. Peng, F. Long, and C. Ding,
%   "Feature selection based on mutual information: criteria
%    of max-dependency, max-relevance, and min-redundancy,"
%   IEEE Transactions on Pattern Analysis and Machine Intelligence,
%   Vol. 27, No. 8, pp.1226-1238, 2005.
%
% C. Ding, and H. Peng,
%   "Minimum redundancy feature selection from microarray gene
%    expression data,"
%    Journal of Bioinformatics and Computational Biology,
%   Vol. 3, No. 2, pp.185-205, 2005.
%
% C. Ding, and H. Peng,
%   "Minimum redundancy feature selection from microarray gene
%    expression data,"
%   Proc. 2nd IEEE Computational Systems Bioinformatics Conference (CSB 2003),
%   pp.523-528, Stanford, CA, Aug, 2003.
%
%
%
% By Hanchuan Peng (hanchuan.peng@gmail.com)
% April 16, 2003
%
    d = data.X';
    f = data.y;
    K = target_number_of_features;
    bdisp=0;
    nd = size(d,2);
    nc = size(d,1);

    t1=cputime;
    for i=1:nd,
        t(i) = mutualinfo(d(:,i), f);
    end;

    [tmp, idxs] = sort(-t);
    fea_base = idxs(1:K);
    fea(1) = idxs(1);
    KMAX = min(1000,nd); %500 %20000
    idxleft = idxs(2:KMAX);
    k=1;

    for k=2:K,
       t1=cputime;
       ncand = length(idxleft);
       curlastfea = length(fea);
       for i=1:ncand,
          t_mi(i) = mutualinfo(d(:,idxleft(i)), f);
          mi_array(idxleft(i),curlastfea) = getmultimi(d(:,fea(curlastfea)),
                                                     d(:,idxleft(i)));
          c_mi(i) = mean(mi_array(idxleft(i), :));
       end;
       [tmp, fea(k)] = max(t_mi(1:ncand) ./ (c_mi(1:ncand) + 0.01));

       tmpidx = fea(k); fea(k) = idxleft(tmpidx); idxleft(tmpidx) = [];
    end;

    indexes = fea;
    data.X = data.X(indexes,:);

    return;
end

%======================================
function c = getmultimi(da, dt)
    for i=1:size(da,2),
       c(i) = mutualinfo(da(:,i), dt);
    end;
end
```

4.2 WRAPPER METHODS

Unlike *filter methods*, which operate independently of any given classifier, *Wrapper methods* are a class of *feature selection methods* dependent on the classifier being used.

Wrapper methods make use of a feature subset selection algorithm[7], which evaluates subsets of features and searches for possible interactions between them, making use of the classifier's classification algorithm and its performance to evaluate the given subset, in an attempt to identify the optimum subset which improves the classifier's performance.

For this reason we can say that in these methods the feature selection algorithm and the classification algorithm are shared (since the same algorithm is used for this two situations with different purposes).

4.2.1 *Sequential Feature Selection*

Sequential Feature Selection approaches aim to obtain a subset of features that best predict the classes of the elements in a given dataset by sequentially selecting features of the dataset until there is no improvement in the classifier's prediction.

4.2.1.1 *Theory*

Usually, these approaches tend to have two main variants:

- *Sequential Forward Selection (SFS)*, in which features are sequentially added to an empty candidate set until the classifier's prediction does not show any improvement in the classification process.
- *Sequential Backward selection (SBS)*, in which features are sequentially removed from a full candidate set until there is an increase in the classifier's prediction.

FIXME: VALE A PENA FALAR UM POUCO MAIS SOBRE AS VARIANTES AQUI? NOMEADAMENTE MENCIONAR EVENTUAIS VANTAGENS/DESVANTAGENS DE UMA EM RELAÇÃO À OUTRA?

4.2.1.2 *MATLAB Implementation*

In the listing that follows we present the implementation of the algorithm in *MATLAB*, in its two variants. Thanks to *MATLAB's sequentialfs* function, which implements this algorithm in its two variants, we only needed to implement a small function to handle this function call.

Listing 4.6: sequentialFs.m

```
function [ data, indexes ] = sequentialFs(data, target_number_of_features, forward,
                                          varargin)
%SEQUENTIALFS Implementation of a Sequential Feature Selection approach for
%             feature selection
%   data is a stprtool data type, containing the features/observed data and
%        the respective classes, in the format (data.X, data.y)
%   target_number_of_features is the desired number of features to select
%   forward is a parameter indicating the use of forward selection or
%          backward selection

    if (forward)
        direction = 'forward';
    else
        direction = 'backward';
    end

    if (iscell(varargin))
        function_name = varargin(1);
    else
        function_name = 'my_fitlm';
    end

    opt = statset('UseParallel', true);
    indexes = sequentialfs(str2func(function_name), data.X', data.y,...
                           'cv', 'none', 'nullmodel', true,...
                           'direction', direction, 'options', opt,...
                           'nfeatures', target_number_of_features);
    data.X = data.X(indexes,:);
end

function dev = critfun(X,Y)
    [~,dev] = glmfit(X,Y,'binomial');
end

function error = my_fitlm(X, Y)
    model = fitlm(X, Y);
    error = model.RMSE;
end
```

4.3 OTHER METHODS

4.3.1 *Correlation Coefficients*

A simpler approach, yet not necessarily less powerful, can be to use the correlation coefficients of the features to do feature selection. We present two such methods which we implement

4.3.1.1 *Theory*

The correlation matrix of the data allows one to understand how features relate to one another. In that sense, if features are found to be highly correlated, it is safe to discard one of these features, as it will only increase the dimensionality problem and add data replication. Thus, one way of using the correlation matrix is to find features with values above a threshold and remove them. We call this the 'exclude_high_correlation' method.

Another way of using the correlation matrix consists of finding features highly correlated with the labelling feature, although this should only be beneficial for cases where a linear classifier would be applicable[1]. We call this method the 'include_high_correlation_with_class' method.

4.3.1.2 *MATLAB Implementation*

In the listing that follows we present the algorithm implementation in *MATLAB*, implemented in the file *feature_selection.m*.

1 Something which, from an analysis of the dataset, we are sure that is not possible in our case.

Listing 4.7: Correlation Coefficients

```
[corrcoef_type, threshold] = args_with_default_values(varargin, \
                                                     'exclude_high_correlation',\
                                                      0.9);
    if strcmp(corrcoef_type, 'exclude_high_correlation')
        % Exclude highly correlated features (leave only one of them)
        % Note that we ignore target_number_of_features. This step only eliminates
        % redundancy.
        c = corrcoef(data.X');
        idx = find( abs(c) > threshold  & c ~= 1.0 );
        [~,idx]=ind2sub(size(c),idx);
        indexes = setdiff(1:data.dim, idx);

        %Remember that data.X is transposed!
        data.X(idx,:)=[];
    elseif strcmp(corrcoef_type, 'include_high_correlation_with_class')

        %FIXME: Implement: if threshold == -1, then use the
        %target_number_of_features most discriminating features (even
        %if they are not good. Currently we ignore target_number_of_features

        % Only include features with high class correlation
        d = [data.X' data.y];
        c = corrcoef(d);
        c = c(end,1:end-1);
        indexes= find( abs(c) > threshold );

        %Remember that data.X is transposed!
        data.X = data.X(indexes,:);
        %^Note that indexes from d and c are still valid in data.X
        %because only the last column is different (hence size changes
        %are irrelevant)
    end
```

We note that this step might prove useful as an extra step after the previous feature selection steps.

5

FEATURE EXTRACTION

Feature extraction aims to to map existing features to a set of new features which prove better for classification and allow for dimensionality reduction. Figure 2 gives an example of how dimensionality reduction can be achieved through feature extraction.

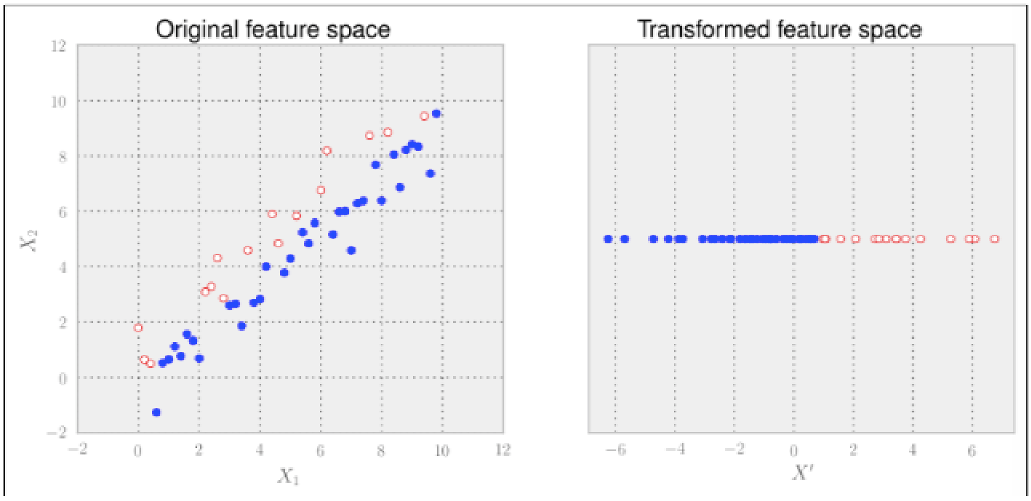


Figure 2: Example of feature extraction (LDA) used for dimensionality reduction. Source: `http://zhangjunhd.github.io/assets/2014-10-07-dimensionality-reduction/8.png`

There are several researched methods for Feature Extraction, each with their advantages and disadvantages. In the following sections we will look at those implemented in our work.

5.1 PCA

Principal Component Analysis, or PCA, is a popular feature extraction method which aims to find the principal components of the data, giving room for elimination of the dimensions/components with the least contribution to it. The principal components are linearly uncorrelated variables.

The general idea behind PCA is that of finding an orthogonal transformation that can find the direction of maximum variation in the data. The transformation should be such that the new coordinate system's first coordinate (called the first principal component) has the greatest variation, with following coordinates having decreasing orders of variation. Intuitively, one can understand that directions with little to no contribution to the overall variance of the data can be discarded as being of little relevance.

PCA is dependent on the scaling of the input features. If they are not previously normalized, then PCA might give skewed results. There has been work in producing a scale-invariant version of PCA, but we did not consider it for our project.

An example of PCA can be seen in Figure 3. In this figure, the two axis represent the first and second principal components of the data.

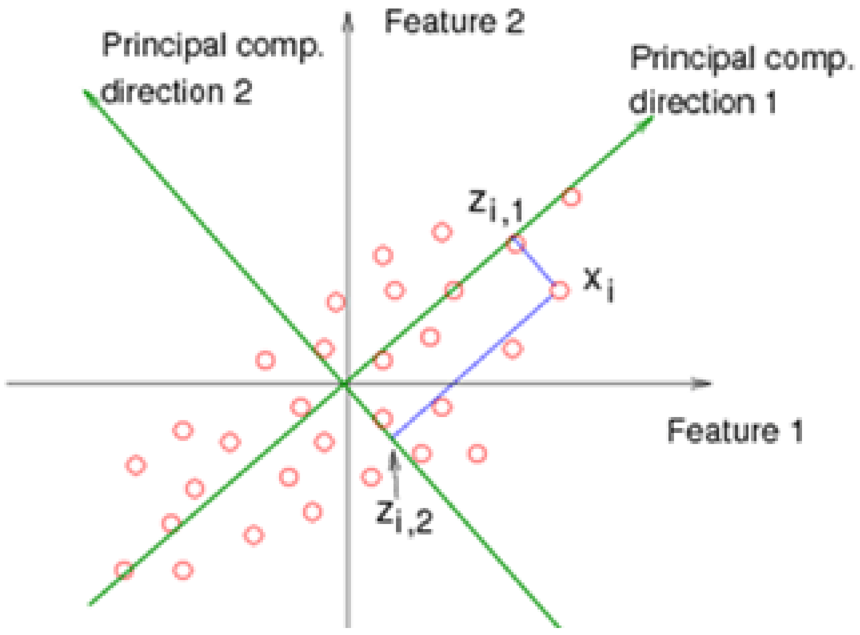


Figure 3: PCA example. Source: `https://onlinecourses.science.psu.edu/stat857/sites/onlinecourses.science.psu.edu.stat857/files/lesson05/PCA_plot_02.gif`

It should be noted that PCA does not take labels into account, but merely looks at data variance. In this sense, improper usage of PCA might lead to bad results.

5.2 LDA

Linear Discriminant Analysis, also known as LDA, is similar to PCA in the sense that it also looks for a linear combination of variable which best explain the data. However, unlike PCA, LDA takes the class of the data points into account, attempting to separate them. In that sense, LDA can be used as a somewhat naive classifier.

The general idea of LDA is that one should look for a transformation which would maximize the variance between classes and minimize the variance within them. Intuitively, maximizing the variance between classes makes them more easily separable, whereas minimizing their within variance also reduces their overlap.

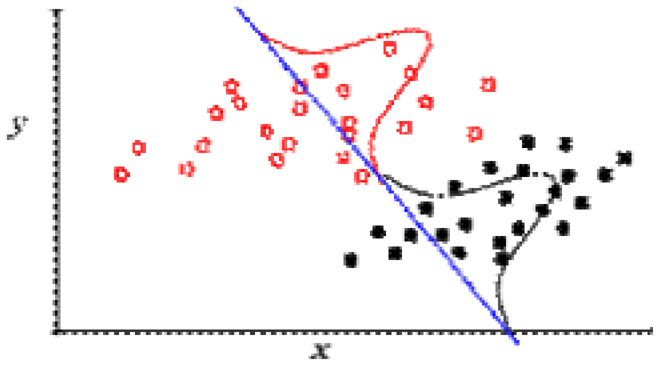


Figure 4: LDA example. Source: http://www.mu-sigma.com/analytics/images/cafe_cerebral/DiscriminantAnalysis\OT1\textendashtwovariables.gif

An example of LDA can be seen in Figure 4. In this figure, the line represents the axis which which, when destination of a projection of the data, allows for maximum separability between classes (maximizes between class variance and minimizes within class variance).

The differences between LDA and PCA can be accurately seen in Figure 5, where both methods are compared. Note that if dimensionality reduction is performed, PCA renders classification impossible, whereas LDA does not.

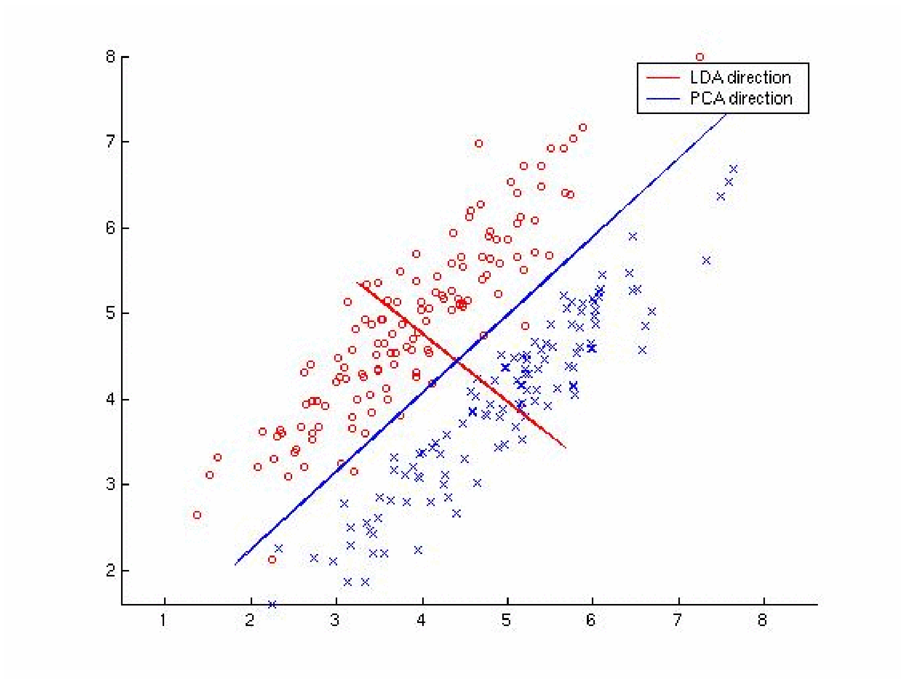


Figure 5: PCA and LDA example differences. Source: http://cmp.felk.cvut.cz/cmp/software/stprtool/examples/ldapca_example1.gif

5.3 KPCA

Kernel PCA, or KPCA, is a variant of PCA where the Kernel trick is used to extend the limits of linear PCA. Operations are, in theory, performed in a mapped kernel space, although in practice such transformation is never explicitly performed. The best utility of KPCA is that it allows for separability of data points in non linear fashion – i. e., they are lineaer in the kernel feature space, not in the original one. The kernel is a mapping function which maps the original space to a higher dimensionality one, thus allowing for the more trivial existance of a separating hyperplane in said high dimensionality space, which eventually is projected back to the original space as a non-linear entity. In KPCA, one leaps in dimensionality to a higher dimensional space ("in theory") so that a low dimensionality subset from that high dimensionality space can be chosen. In this sense, there is an apparent increase in dimensionality to allow for a decrease. Figure 6 shows the differences between the two methods.

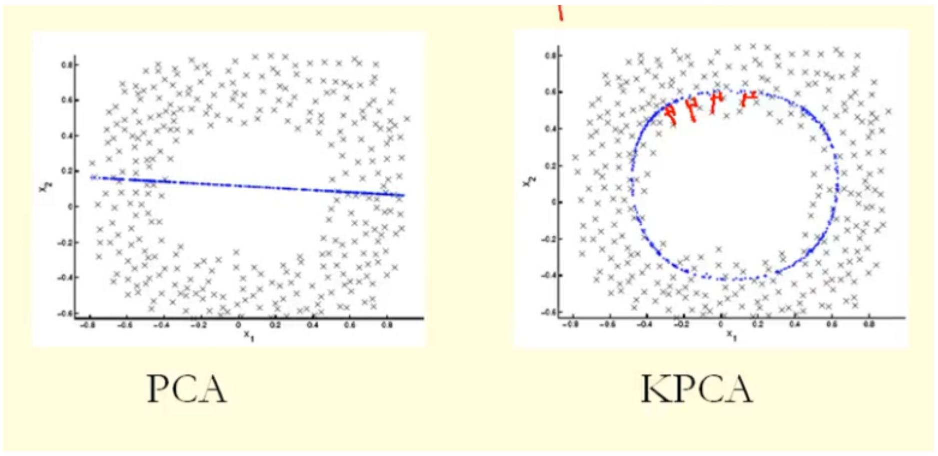


Figure 6: PCA vs KPCA. Source: `http://stats.stackexchange.com/questions/94463/what-are-the-advantages-of-kernel-pca-over-standard-pca`

5.4 GDA

The same principle that extends PCA with KPCA can be applied to LDA. The input feature space can be mapped to a higher dimensionality space and the LDA applied in such space. This originates the Generalized Discriminant Analysis presented by [8]. An example is shown in Figure 7 where data is separated by a GDA classifier.

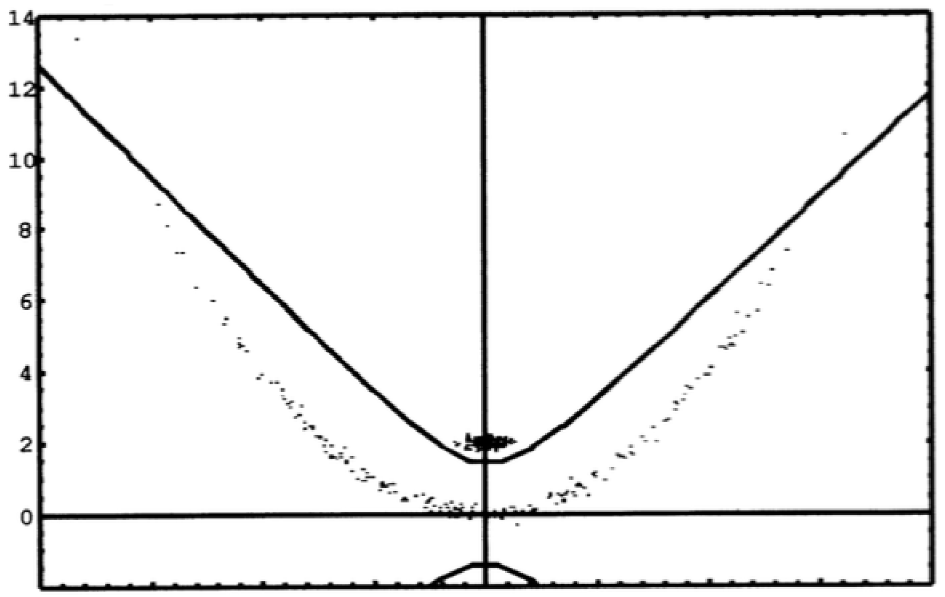


Figure 7: GDA example[8]

5.5 IMPLEMENTATION

The SPRT tool already implements all of the aforementioned methods, and, so, our implementation merely proxies between our data structures and the tool. It can be found in *feature_extraction.m*.

Listing 5.1: feature_extraction.m

```
function data=feature_extraction(data,method, target_number_of_features, varargin)
    if strcmp(method, 'pca')
        m = pca(data.X, target_number_of_features);
        data = linproj(data, m);
    elseif strcmp(method, 'lda')
        m = lda(data, target_number_of_features);
        data = linproj(data, m);
    elseif strcmp(method, 'kpca')
        [kernel_type, kernel_argument]=args_with_default_values(varargin,'rbf',1);
        options.ker = kernel_type;
        options.arg = kernel_argument;
        options.new_dim = target_number_of_features;
        %m = greedykpca(data.X, options);
        m = kpca(data.X, options);
        data = kernelproj(data, m);
    elseif strcmp(method, 'gda')
        [kernel_type, kernel_argument]=args_with_default_values(varargin,'rbf',1);
        options.ker = kernel_type;
        options.arg = kernel_argument;
        options.new_dim = target_number_of_features;
        m = gda(data, options);
        data = kernelproj( data, m );
    end
    data.num_data = size(data.X, 2);
    data.name = 'Higgs Data';
end
```

6 DATASET BALANCING

The experimental dataset is unbalanced, meaning there is an unequal number of elements of each class. This may skew the classifier training and induce unwanted bias. In order to counter this, some form of dataset balancing must be applied. We now present three such methods, two of which we successfully implemented.

6.1 UNDERSAMPLING

The undersampling technique randomly discards instances with the majority label. The idea is that by discarding instances, we are only losing information, not creating new, possible fake and bias-inducing information. Of course, this method might drastically reduce the dataset size if the imbalancement is large.

This technique is particularly useful if one is also interested in reducing the training time, since the effective number of samples is highly reduced. It also has the advantage that it does not involve modifying the classifiers themselves.

6.2 OVERSAMPLING

In contrast to the undersampling technique, the oversampling technique selects random instances from the minority label and doubles them until the data are evenly balanced. This method may have a negative effect in the classifier by introducing bias. Duplicating instances makes them twice as relevant, thus introducing bias.

The oversampling technique, however, might be beneficial in the sense that it does not discard large amounts of data, and might be particularly useful if the minority samples are similar to one another, where duplicating them does not introduce very high bias. Like the undersampling technique, it doesnot involve modifying the classifiers.

6.3 ADJUSTED ACCURACY

Another technique involves modifying the classifiers to reflect the *adjusted accuracy* instead of accuracy in its validation methods. The *adjusted accuracy* is given by

$$A = 0.5 \times C_1 + 0.5 \times C_2$$

Where C_1 and C_2 are the number of correctly labeled instances for labels one and two, respectively.

This method has the main drawback of involving a modification of the classification algorithm.

6.4 MATLAB IMPLEMENTATION

In our work, the undersampling and oversampling techniques are both implemented. We used an auxiliary shuffle method, which we developed, and which shuffles an array.

Listing 6.1: balance_dataset.m

```
function [ balanced_dataset ] = balance_dataset( sprt_data, method )

    fprintf('....Balancing dataset with %s\n', method);
    classes = unique(sprt_data.y);
    %FIXME: Should make sure there are only two classes
    indexes_1 = find(sprt_data.y == classes(1));
    indexes_2 = find(sprt_data.y == classes(2));
    indexes_1 = shuffle(indexes_1);
    indexes_2 = shuffle(indexes_2);

    if strcmp(method,'undersample')
        max_idx = min(length(indexes_1),length(indexes_2));
        balanced_dataset_idx = [ indexes_1(1:max_idx) ; indexes_2(1:max_idx) ];
    elseif strcmp(method,'oversample')
        diff = length(indexes_1) - length(indexes_2);
        if diff > 0
            indexes_2 = [indexes_2 ; indexes_2(1:abs(diff))];
        elseif diff < 0
            indexes_1 = [indexes_1 ; indexes_1(1:abs(diff))];
        end
        balanced_dataset_idx = [indexes_1 ; indexes_2];
    end

    if strcmp(method,'undersample') || strcmp(method,'oversample')
        data_x = sprt_data.X(:,balanced_dataset_idx)';
        data_y = sprt_data.y(balanced_dataset_idx);
        data = [data_x data_y];
        balanced_dataset = convert_to_sprt_data(data);
    end
end
```

7 CLASSIFIER

The present chapter is reserved to detailing about the different *Classifiers* implemented in the course of the project assignment.

Since our work focused in building an application capable of classifying a given set of events into one of two possible categories, one can understand the importance of the classifier in our work: It is the central and most important component of the application built, responsible for learning the training data, its patterns and properties and identifying them in other independent datasets of the same nature.

7.1 SVM

SVM (or *Support Vector Machine*) is a very popular machine learning solution[1], consisting in the creation of a *supervised learning model* focusing on the analysis and recognition of patterns in datasets.

These supervised models consist in a representation of the mapping of the training examples in space, in such a way that there is a clear gap, as wide as possible, that separates the different categories of the training data. Once this model is created, new examples can be mapped into that same space, and based on that mapping their category can be predicted.

In the current work we made use of *MATLAB's* native *SVM* implementation.

7.2 LIBSVM

LIBSVM is an integrated software for support vector classification, regression and distribution estimation, supporting multi-class classification. It is a very popular and well-regarded *cross-platform* library for *Support Vector Machines*.

Theoretically, its operation mode is similar to what was described in the previous section, even though its implementation differs at some points from the implementation available in *MATLAB*.

1 Even considered the *State of the art* in machine learning and pattern recognition

The main difference between these two *SVM* implementations is probably the execution time, which is shorter in the *LIBSVM* implementation. In the tests performed in the course of this practical assignment we got the impression that, in general, the *LIBSVM* implementation produced more accurate classifiers.

7.3 NAIVE BAYES

In machine learning, *Naive Bayes Classifiers* are simple probabilistic classifiers, assuming strong independence between the features, that apply *Bayes' Theorem* to perform their classifications.

The *Bayes' Theorem* describes the probability of a given event, based on conditions possibly related to it. In fact, the *Bayes' Theorem* allows us to mathematically describe that conditional probability based on the probabilities of the events in question[9]:

$$P(A|B) = \frac{P(A \bigcap B)}{P(B)} = \frac{P(B|A) \times P(A)}{P(B)}$$

where $P(A|B)$ denotes the conditional probability.

Therefore, the main idea behind any *Bayesian Classifier* is the following: Given an instance represented by a vector $x = (x_1, \ldots, x_n$ and a set of K possible classes that can be assigned to that instance, the classifier determines the probabilities of the given instance belonging to each of the K classes and selects the class with higher probability. This probability can be expressed as:

$$p(C_k|x_1, \ldots, x_n)$$

for each of the K possible class labels.

There is no single unified algorithm for training such classifiers. Instead, a wide family of algorithms can be found in the literature, all based on the common assumption that the feature values are independent.

The main problem with the formulation presented above is that if the number of features of the instances (n) is large enough, or if a feature can take on a large number of values, it is infeasible to base this model on probability tables.

Therefore, with some mathematical manipulations[9], and under the feature independence assumptions, we can reformulate the problem and so that the conditional probability distribution of a given instance over each possible class C_k is:

$$p(C_k|x_1, \ldots, x_n) = \frac{1}{Z} p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

where Z is a scaling factor.

A *Naive Bayes Classifier* combines the model presented with a simple decision rule: For any given instance, select the most probable class label assigned to it. Therefore, our classifier can be described as:

$$\underset{k\in\{1,\ldots,K\}}{\operatorname{argmax}}\ p(C_k)\prod_{i=1}^{n} p(x_i|C_k)$$

7.4 binary regression decision tree

The *Binary Regression Decision Tree Classifier* makes use of a *Binary Decision Tree* to select the class to assign to a given instance.

We can define a *Decision Tree* as a *flow-chart-like* tree structure where each internal node (that is, each non-leaf node) denotes a test on an attribute, each branch represents the outcome of a test on one of the input variables, and each leaf holds a class label. A *Binary Decision Tree* is a *Decision Tree* holding, for each node, a binary test for a given attribute(For example: ″If attribute A ¿ 2″ against ″If attribute A ¡= 2″)

In figure 8 we present an example of a binary decision tree.

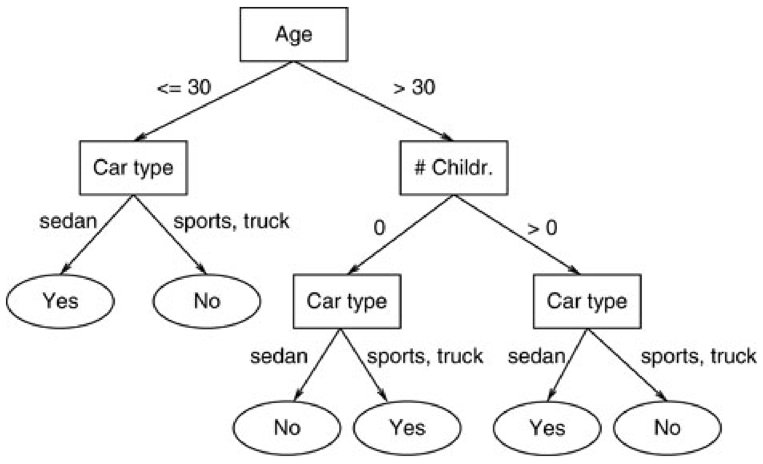


Figure 8: Binary Decision Tree Example

Therefore, a *Binary Regression Decision Tree* can be defined as a *Binary Tree* holding a linear regression performed over the attributes of the instances in the training dataset.

7.5 classification decision tree

A *Classification Decision Tree* is very similar to a *Binary Decision Tree* with the main difference that it represents a model where the target variable can take a finite set of values.

The remainder operation mode of such trees is very similar to what was already described in the previous section of this chapter.

7.6 k-nearest neighbours

As described in chapter 3 the main idea of the *K-Nearest Neighbours* algorithm is, for a given instance, to select its *K* closest neighbours considering their class labels to determine which class label to assign to the mentioned instance.

This algorithm is usually addressed as a *Lazy Learner*, since it does not require any specific training. Nevertheless, one of its advantages is related to that property, which makes it a simple and easy-to-use algorithm. However it usually increases its computational time and makes developers determine an optimum value for the number of neighbours to select.

8

GRAPHICAL USER INTERFACE

As requested in the project's assignment, we also developed a simple *Graphical User Interface* to allow any user to test our application and their features described in the previous chapters.

Upon launching the interface the user can see that it is divided in tabs, each representing a distinct phase of our application: Input data selection and pre-processing; Classifier selection and specification of the training parameters; And results presentation and analysis.

We called these phases *Pre-Processing*, *Train & Classification* and *Results*, respectively.

We now proceed to specify in more detail each phase of our application.

8.1 PRE-PROCESSING

In this phase of the application the user can specify the input file to use in the training and/or classification phases of the execution. A default file is selected, but by deselecting the *"Use Default File"* checkbox the user can then upload a file of his/hers choice to be processed.

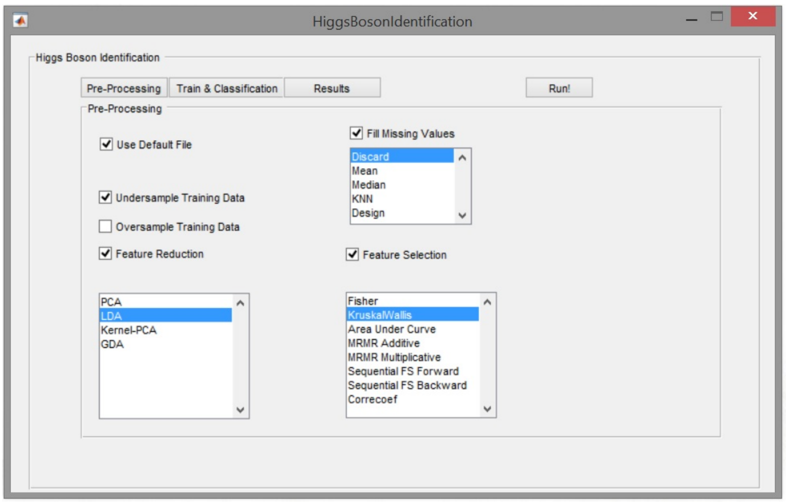
Once the user has selected the main to be processed he/she can specify a series of operations to be performed to that file:

- In the presence of any missing values stored in the uploaded file[1] the user can specify a method to deal with such values (If that is his/hers wish). The methods available in the application for this purpose were previously described in chapter 3. If the user does not desire to deal with missing values then he/she must deselect the *"Fill Missing Values"* option.
- The user can also select *Feature Selection* and *Feature Reduction* methods to apply to the dataset present in the previously selected file. In a way similar to the one presented for dealing with missing values, if the user does not want to apply one of these methods (or both) then he/she must deselect the given option (either *"Feature Reduction"* or *"Feature Selection"*). In both cases the available methods were described and presented in chapters 4 and 5.

1 The reader must note that, according to the assignment's initial requirements, any value is considered a missing value if equal to -999.0

- Since most datasets of this nature do not contain a balanced number of instances of each class to be identified by a classifier, and given the fact that unbalanced data have a negative effect in the classifier's performance, there is a real need in such applications to balance the training data, so that the number of instances of each class is the same. To do so, the user must select one of two possible balancing methods: Either the specified dataset for training is undersampled, meaning that some of its instances are removed, or it is oversampled, meaning that some of its instances are doubled.

In figure 9 we show an example of our application in the Pre-Processing phase:



[✓] Use Default File
[✓] Fill Missing Values: [✓ Discard] Mean Median KNN Design
[✓] Undersample Training Data
[ ] Oversample Training Data
[✓] Feature Reduction: PCA [✓ LDA] Kernel-PCA GDA
[✓] Feature Selection: Fisher [✓ KruskalWallis] Area Under Curve MRMR Additive MRMR Multiplicative Sequential FS Forward Sequential FS Backward Correcoef

Figure 9: Example of the interface in the pre-processing phase of the application

8.2 TRAIN & CLASSIFICATION

As stated before, this phase is reserved to the specification of the training parameters and to the selection of the classifier to be used by our application. This is also the application's main phase, where the pre-processing, training and classification actions take place.

In this section, as well as in the entire application, we can identify three different and independent flows of the execution, mapped to the interface through the three checkboxes displayed: *"Load Classifier"*, *"Use Previous File For Training and Test"* and *"Use Previous Dataset only For Training"*. We will now proceed to detail each of these different execution flows.

When the user selects the *"Load Classifier"* option he/she must provide a previously trained and saved classifier. In this case, there is no need to specify the training and testing percentages of the uploaded dataset, as this as already been made. To specify the desired classifier, the user must click on the *"Browse File"* button in the *"Select Classifier"* panel and select the desired classifier to upload. In this scenario the user must also specify the target dataset where the uploaded classifier should be tested, also by clicking on the *"Browse File"* button in the *"Select File For Testing"* panel and selecting the desired testing file to upload. Once these two steps have been completed successfully, all there is left to do is clicking on the *"Run!"* button to start the execution.

At this point, when the user presses the *"Run!"* button the application loads the uploaded classifier and the testing file, gathering the specified test set of that file, which is then presented to the classifier for classification. Once the classification ends its results are presented to the user in the *"Results"* tab.

If, on the other hand, the user does not have any trained classifier, or if he/she desires to train one, then he/she can do so by two different procedures: The user either uses the entire data in the previously uploaded dataset to train the classifier, or selects a subset of that data to perform the classifier's training and the remainder of the data for testing.

In the first case, the user must click on the *"Use Previous Dataset Only For Training"* option. Upon performing this selection, the user must select the testing dataset (in a similar process to the one adopted when loading a previously trained classifier) and the classifier to train. In this last step, the user must select the desired classifier from the available list in the *"Classifier"* panel, specifying any classifier's parameters if needed. Once these two steps have been completed successfully, all there is left to do is clicking on the *"Run!"* button to start the execution.

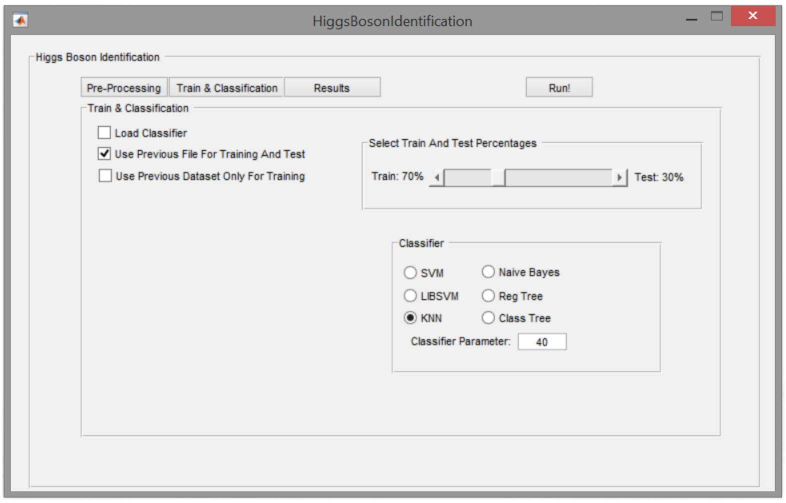
At this point, when the user presses the button to start the execution, the selected training dataset is loaded and preprocessed according to what was specified by the user in the first phase and the selected classifier is trained with the data in that file. Then, once the training is complete, the test file is loaded and its dataset is extracted from it, being then presented to the classifier for further classification. Again, once the classification has finished, its results are presented to the user in the *"Results"* tab.

In the second case, the user must click on the *"Use Previous File For Training and Test"* option. Upon performing this selection, given the fact that we are using the same file for training and testing the classifier, the user only needs to select the desired classifier (as previously described) and select the percentage of data to consider for the selected classifier's training, with the remaining data being used for testing.

We warn the reader to the fact that, in this step, when selecting the training data for the classifier we also take into account the sampling method selected by the user in the previous phase.

Once these two steps have been completed successfully, all there is left to do is clicking on the *"Run!"* button to start the execution. After clicking in that button, the training and testing file is loaded. The application then performs the specified preprocessing methods to the data in that file, extracting the training and testing datasets, according to the user's specifications. The selected classifier is then trained with the training dataset and, when its training is complete, it classifies the data in the classification dataset, with the classification results being presented to the user in the *"Results"* tab.

Whenever an execution finishes, that is whenever a classification process ends, the user can save the selected and trained classifier for future use, simply by clicking on the *"Save Classifier"* button and selecting the target output file (Where the classifier will be saved).


HiggsBosonIdentification

Higgs Boson Identification

Pre-Processing | Train & Classification | Results | Run!

Train & Classification

- [ ] Load Classifier
- [✓] Use Previous File For Training And Test
- [ ] Use Previous Dataset Only For Training

Select Train And Test Percentages

Train: 70% Test: 30%

Classifier

○ SVM ○ Naive Bayes
○ LIBSVM ○ Reg Tree
● KNN ○ Class Tree

Classifier Parameter: 40

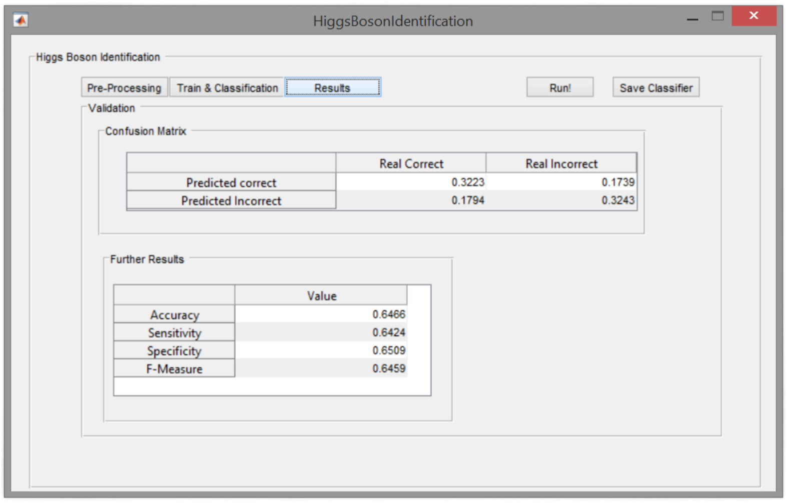
Figure 10: Example of the interface in the classifier selection and specification of the training parameters phase of the application

8.3 RESULTS

In the *Results* phase, as the reader may have already figured out, the results of the latest classification operation are presented to the user.

These results mainly consist in presenting a series of metrics used for evaluating the classifier's performance, like:

- A *Confusion Matrix*, containing the number of *True Positives*, *False Positives*, *False Negatives* and *True Negatives* classifications made by the classifier.
- A table where we present the classifier's *Accuracy*, *Sensitivity*, *Specificity* and the *F-Measure*.


HiggsBosonIdentification

Higgs Boson Identification

Pre-Processing | Train & Classification | Results | Run! | Save Classifier

Validation

Confusion Matrix

	Real Correct	Real Incorrect
Predicted correct	0.3223	0.1739
Predicted Incorrect	0.1794	0.3243

Further Results

	Value
Accuracy	0.6466
Sensitivity	0.6424
Specificity	0.6509
F-Measure	0.6459

Figure 11: Example of the interface in the results analysis phase of the application

9 EXPERIMENTS AND RESULTS

In order to assess the quality of our classifier and to develop the best classifier available, we conducted a series of experiments and recorded their results. In this section, we document these experiments and their results

9.1 EXPERIMENTAL SETUP

Our tests were conducted on a 2.5 GHz Intel Core i7 Macbook Pro Mid-2014, with 16 GB 1600 MHz DDR3. They were mostly conducted using the GUI provided with the project (in cases where it wasn't used it will be explicitly noticed). No test was allowed to run (i.e. train and classify) for longer than 1 hour.

9.2 EXPERIMENTS AND PROCEDURE

There were several different experiments performed throughout the development of this work. There are several parameters to take into account:

- The choice of feature extraction method (including none)
- The choice of feature selection method (including none)
- The choice of missing value imputation method
- The choice of dataset balancing technique (undersample and oversample)
- The use of normalization or not
- The training and classification method
- In some classifiers (e.g. KNN), classifier parameters

Since it was unfeasible to test all combinations of these parameters, most tests were ran in "ad-hoc" fashion, allowing us to develop a "feel" for the best classifiers and parameter configurations. There were, however, some automated experiments, which we will briefly explain.

All of the experiments were executed using a 70%-30% training-testing split. Performance was measured with a particular focus on accuracy and the $F1 - measure$. The best classifiers were furthermore tested on the whole dataset. Depending on the parameter configuration, this could produce significantly worse results. For instance, if the oversampling technique was used during the $70 - 30$ training phase, then using 100% of the dataset meant that a highly unbalanced dataset was now being tested, and that the trained classifier was trained on repeated instances of the minority class. This, in particular, meant that some classifiers that performed with 90% accuracy dropped to about 70% accuracy in some tests.

9.2.1 *Optimal K parameter for KNN classifier under specific circumstances*

During the development of the application we used LDA (target number of features $= 14$) as a feature extraction technique and the Fisher feature selection method (target number of features = 7). With this configuration, we looked for the optimal number of K for the KNN algorithm, searching from $K = 5$ to $K = 150$ with a step of 5. The results showed that $K = 40$ was the best value, and can be seen in Figure 12.

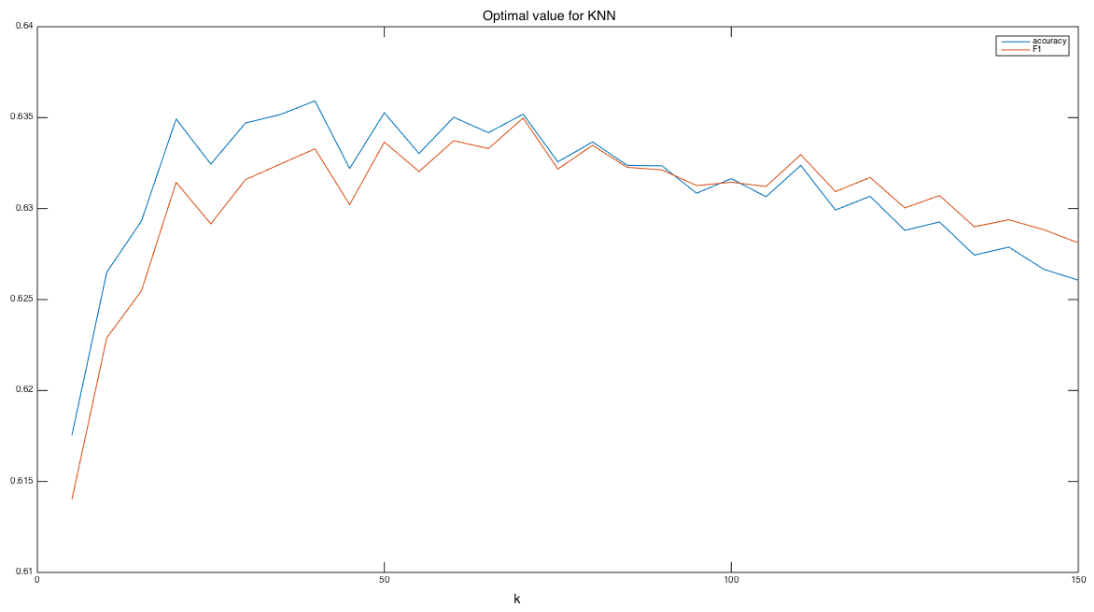


Figure 12: Results of looking for the best value of K.

9.2.2 *Results and conclusions*

As the result of our "ad-hoc" search for the best classifier, a couple of conclusions were clear, which we now list:

- Normalizing the data produced at best the same results. In general, normalizing the data worsened the classifier performances, with drops from 70% accuracy to 40% accuracy being the norm (for any classifier)

- PCA produces worse results than LDA, but they do not generally differ in more than 5% performance difference when all other parameters are kept constant
- The *KNN* missing value imputation technique produces “good results”. By good, we mean that it is better than all other techniques, with the exception of the design technique.
- The design technique (multivariate regression) missing value imputation, athough not the fastest to execute, produces the best results. All of our best classifiers were found when using the multivariate regression missing value imputation technique
- The Naive Bayes classifier is very fast and produces results with accuracy between 50% and 70%, regardless of the remaining parameter configuration.
- The *KNN* classifier mostly achieves performance between 68% and 75%, with optimal values of K tending towards $K = 40$ (as experimentally seen in a previous section). There are some instances of *KNN* with better performance, as we further discuss.
- The classification decision tree is fast and produces the best results. Most of our best classifiers were found using the classification decision tree and allowed for fast training and classification
- The oversampling data balance technique introduces bias. This is particularly visible when 100% of the dataset is used for testing (with no balancing). This also means that the undersampling data balance technique is the best.
- Both SVM implementations are slow (the slowest algorithms) and do not produce results better than 80% accuracy. In our tests, no SVM could perform better, although we note that we are limited in the amount of tests
- The best classifier found achieves 92% **accuracy** when using $70 - 30$ split and 95% **accuracy** when used on the full 100% dataset. The parameters for this dataset were:
  - Training balancing method: undersample
  - 70-30 split
  - No data normalization
  - Feature extraction: none
  - Feature selection: none
  - Missing value imputation method: Multi variate regression model
  - Classifier: Classification Tree trained with the classification decision tree algorithm
- The second best classifier achieves 90% accuracy both when using $70 - 30$ split and when tested on the full 100% dataset and differs from the first only in the classification algorithm: *KNN* was used with $K = 40$.

These conclusions should be taken with care, as they are indeed the result of “ad-hoc” testing, due to our limited time. However, we are quite sure that the SVM parameters could and should have been tuned to obtain better results and that the decision trees, atlhough the best result, might be a case of overfitting.

10

ADDITIONAL MATLAB CODE

In this chapter we present additional MATLAB code that we developed and use throuought the project.

10.1 DATA SET LOADING

To load the dataset, we developed two functions. The first loads the dataset itself and returns a matrix where the last column has the class. The second method constructs an SPRT-compatible structure from a matrix in the format returned by the former. We now present both of these functions.

10.1.1 *load_dataset*

This function loads the dataset and returns said dataset, excluding the ID column and also returning column names. The dataset can now be used for missing value imputation.

Listing 10.1: load_dataset.m

```
function [ column_names, higgs_data ] = load_dataset()
    load('dataset/higgs_data.mat');
    higgs_data=higgs_data_for_optimization(:,2:end);
end
```

10.1.2 *convert_to_sprt_data*

This function converts a matrix returned by *load_dataset* and converts it to an SPRT compatible format. Note that it can and should be called for feature extraction and selection. Lastly, note that it can be used with data matrixes of any dimension, and the only assumption is that instances are rows, features are columns and the last column is the class column.

Listing 10.2: convert_to_sprt_data.m

```
function [ sprt_data ] = convert_to_sprt_data( data )
    sprt_data.X = data(:, 1:end-1)';
    sprt_data.y = data(:,end);
    sprt_data.dim = size(sprt_data.X, 1);
    sprt_data.num_data = size(sprt_data.X, 2);
    sprt_data.name = 'Higgs Data';
end
```

10.2 DATA NORMALIZATION

Some methods require data to be normalized, and it is generally a good idea to do so. To this end, we developed two methods: one normalizes one feature, and the other normalizes the entire dataset, assuming it is given data in the *load_dataset* format.

10.2.1 *scalestd*

This method normalizes one individual feature. It is implemented in *scalestd.m*

Listing 10.3: scalestd.m

```
function [ Xout ] = scalestd( Xin )
    Xout=Xin-mean(Xin(:));
    Xout=Xout/std(Xout(:));
end
```

10.2.2 *normalize_data*

This method normalizes data in the form returned by *load_dataset*. It is implemented in *normalize_data.m*

Listing 10.4: normalize_data.m

```
function normalized_data = normalize_data(features)
    normalized_data = features;

    %Don't normalize the last column (it's the label)
    for i=1:size(normalized_data,2)-1
        normalized_data(:,i) = scalestd(normalized_data(:,i));
    end
end
```

10.3 DEFAULT ARGUMENT PARSING

Since many of our methods take a variable number of arguments, some of which have optional values, we developed a method of dealing with default values. Using a combination of vargin and knowledge of the number of expected parameters, we can use the user supplied value or a default one in just one line of matlab code. For instance, if the user could supply two arguments, a and b (both optional and with default values 16 and 10), we could extract them using *[a,b] = args_with_default_values(varargin,16,10);*

10.3.1 *arg_with_default_value*

This method is supplied with the *varargin* of a method, an index and a default value. It then returns the value at that index in varargin, or the default value if it is not found.

Listing 10.5: arg_with_default_value.m

```
function [ret] = arg_with_default_value (args, index, default_value)
    nargs = length(args);
    if nargs >= index
        ret = cell2mat(args(index));
    else
        ret = default_value;
    end
end
```

10.3.2 *args_with_default_values*

This method uses the above method. It receives the *varargin* of the caller method and a variable list of default values. It will then return a list of arguments where they are either the value supplied by the user, or the default value. The default values are supposed to appear in order.

Listing 10.6: args_with_default_values.m

```
function [varargout] = args_with_default_values (args,varargin)
    nargs = length(varargin);
    varargout = cell(size(varargin));
    for i=1:nargs
        varargout{i}=arg_with_default_value(args,i,cell2mat(varargin(i)));
    end
end
```

11

CONCLUSION

In this document we presented our progress and work done in the course's project. We studied, developed and applied a series of documented methods which will allow us[1] to extract meaningful information from the provided dataset, useful for the next stages of the project, namely the construction of the classifier and further assessment.

Since the first day of work in this project we kept the development of the different methods as generic as possible, keeping in mind the possibility of porting them to different datasets in the future, with minimum changes to the code.

In the course of our work we tested a wide range of classifiers and preprocess operations made to the input data. As stated in previous chapters, we gathered a series of conclusions:

For starters, the first major conclusion gathered was related to the normalisation of the input data. In the experiments conducted, normalising the data lead to, at most, the best same result.

The *LDA* method for *Feature Reduction* proved to be more effective than *PCA*, even though their difference is not very significant (we only registered an improvement of about 5%). A lot of other classifiers and methods for data preprocessing enabled us to achieve good results (like the *KNN*, the *Naive Bayes* and the *Design Technique* for missing values replacement) without needing too much time to run. The general conclusion of our work is that we managed to develop a set of classifiers that consistently had an F-measure and accuracy above 70%, which we consider good and, also, two classifiers which, in particularly, achieved accuracy (and f-measure) above 90%. For us, this is a success.

There are many parameters and configurations which could and should be studied in future work and that, regrettably, we did not have time to test. These include the parameters of SVM, a study of the best value of K in the KNN classifier when no feature selection or extraction was performed. More feature selection techniques could be used, and overall the feature extraction process could be improved. We would have wished to also use K-cross validation but found that it would add too much complexity to our application development.

The results of our experiments, however, also strike us with oddness. The fact that feature selection and extraction produce worse results tends to indicate that our methods were not the most appropriate, or that the set of features provided by CERN (which come in

1 At least we hope so

raw and derived fashion) were already "the" best features for this problem. Nevertheless, we would have liked to explore this issue more in depth.

In summary, although we recognise that more time would have been useful to explore many different parameters and research questions, the accuracy of our classifiers satisfies us and we believe that it shows that our work is not sub-par.

BIBLIOGRAPHY

[1] (2014) Higgs boson machine learning challenge. [Online]. Available: https://www.kaggle.com/c/higgs-boson/

[2] (2008) Statistical pattern recognition toolbox. [Online]. Available: http://cmp.felk.cvut.cz/cmp/software/stprtool/

[3] C. G. I. G. B. D. R. Claire Adam-Bourdarios, Glen Cowan, "Learnign to discover: The higgs boson machine learning challenge," 2014, http://higgsml.lal.in2p3.fr/files/2014/04/documentation_v1.8.pdf.

[4] A. C. Acock, "Working with missing values," *Journal of Marriage and Family*, vol. 67, no. 4, pp. 1012–1028, 2005.

[5] T. Speed, *Statistical analysis of gene expression microarray data*. CRC Press, 2004.

[6] B. Ribeiro, "Pattern recognition: 6 - non-parametric methods." Pattern Recognition Class Presentation. DEI-FCTUC, University of Coimbra, 2015.

[7] R. K. G. John and K. Pfleger, "Irrelevant features and the subset selection problem." Proceedings Fifth International Conference on Machine Learning, 1994.

[8] G. Baudat and F. Anouar, "Generalized discriminant analysis using a kernel approach," *Neural computation*, vol. 12, no. 10, pp. 2385–2404, 2000.

[9] (2015) Naive bayes classifier. [Online]. Available: http://en.wikipedia.org/wiki/Naive_Bayes_classifier