Dipartimento di Automatica e Informatica
Master's degree in Data science and engineering

Politecnico di Torino

Analysis of the MAGIC Gamma Telescope Data Set

Tesina of Mathematics in Machine Learning

July 25, 2021

Simone PAPICCHIO

s281811@studenti.polito.it

July 25, 2021

Contents

1 Dataset Overview

The data present in this dataset is generated to simulate registration of high energy gamma particles in a ground-based atmospheric. Cherenkov gamma telescope (CTA) observes high energy gamma rays, taking advantage of the radiation emitted by charged particles produced inside the electromagnetic showers initiated by the gammas, and developing them in the atmosphere. More precisely, when the gamma rays reach the earth's atmosphere they interact with it, producing cascades of subatomic particles. These cascades are also known as air or particle showers. Light travels 0.03 percent slower in air. Thus, these ultra-high energy particles can travel faster than light in air, creating a blue flash of "Cherenkov light". Although the light is spread over a large area (250 m in diameter), the cascade only lasts a few billionths of a second. CTA's large mirrors and high-speed cameras will detect the flash of light and image the cascade generated by the gamma rays for further study of their cosmic sources allowing reconstruction of the shower parameters. The available information consists of pulses left by the incoming Cherenkov photons on the photomultiplier tubes. Depending on the energy of the primary gamma, a total of few hundreds to some 10000 Cherenkov photons get collected, in patterns (called the shower image), allowing to discriminate statistically those caused by primary gammas (signal) from the images of hadronic showers initiated by cosmic rays in the upper atmosphere (background).

Typically, the image of a shower after some pre-processing is an elongated cluster. Its long axis is oriented towards the camera center if the shower axis is parallel to the telescope's optical axis, i.e. if the telescope axis is directed towards a point source. A principal component analysis is performed in the camera plane, which results in a correlation axis and defines an ellipse. If the depositions were distributed as a bivariate Gaussian, this would be an equidensity ellipse. The characteristic parameters of this ellipse (often called Hillas parameters) are among the image parameters that can be used for discrimination. The energy depositions are typically asymmetric along the major axis, and this asymmetry can also be used in discrimination. There are, in addition, further discriminating characteristics, like the extent of the cluster in the image plane, or the total sum of depositions. The goal of the classification is to correctly classify the gamma signal (g) from hadron (h, background).

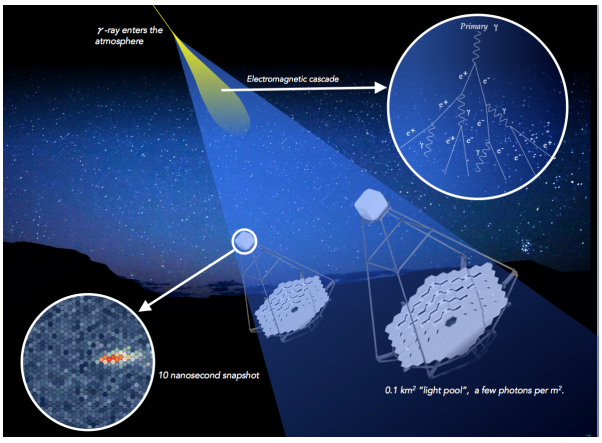


Figure 1: An example of how the Cherenkov gamma telescope (CTA) can detect the γ-rays in the atmosphere
Source: How CTA Detects Cherenkov Light

2 Dataset analysis

The dataset is composed of 19020 distinct records with 11 attributes, including the label:

- **fLength**: [continuous] major axis of ellipse
- **fWidth**: [continuous] minor axis of ellipse
- **fSize**: [continuous] 10-log of sum of content of all pixels
- **fConc**: [continuous] ratio of sum of two highest pixels over fSize
- **fConc1**: [continuous] ratio of highest pixel over fSize
- **fAsym**: [continuous] distance from highest pixel to center, projected onto major axis
- **fM3Long**: [continuous] 3rd root of third moment along major axis
- **fM3Trans**: [continuous] 3rd root of third moment along minor axis
- **fAlpha**: [continuous] angle of major axis with vector to origin
- **fDist**: [continuous] distance from origin to center of ellipse
- **class**: [categorical] g gamma (signal), h hadron (background)

The dataset does not contain missing values.
In Figure 2 the feature boxplots are compared. We can notice that the range of values for each feature is different. i.e **fDist** is around 200 instead **fSize**, **fConc** and **fConc1** have all values around 0. Therefore, in order to not have bias in the model towards higher/lower values, we need to scale the dataset.
In addition, some "distant" measurement are visible in the Fig. 2. However, we actually do not know if they are measurement errors or, instead, some possible outliers. For this reason the outlier removal step is not performed manually, but an algorithm is use (see later).

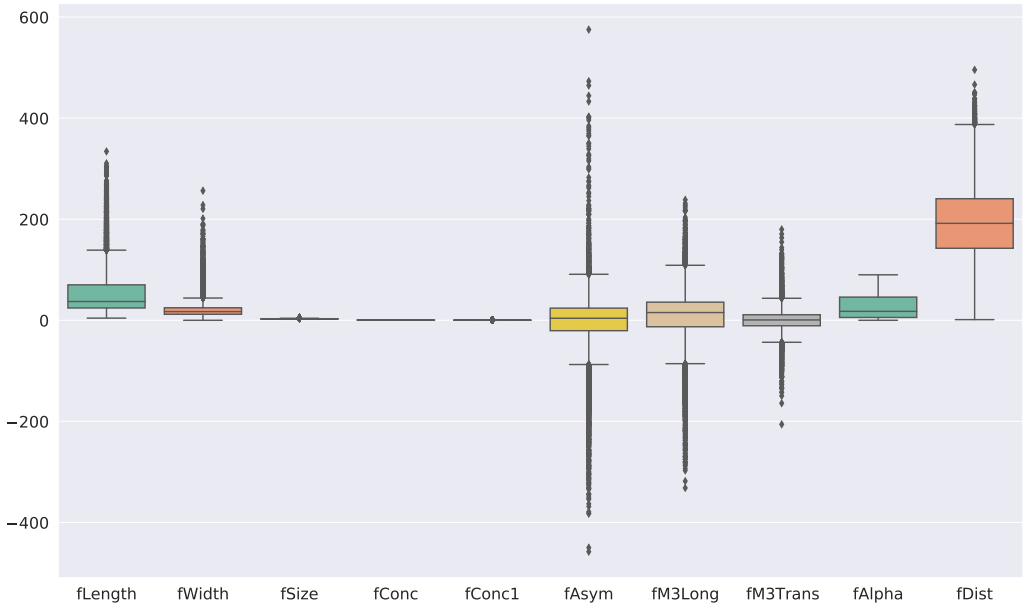


Figure 2: Boxplot of the features compared. On the x-axis there are the features, instead on the y-axis there are the possible value range for the features.
Source: made by me

Interesting deduction can also been made from the pairplot in Fig. 3. On the diagonal are plotted the kernel density estimate (KDE) figures. The KDE is a method for visualizing the distribution of observations in a dataset, analogous to a histogram but it represents the data using a continuous probability density curve in one or more dimensions. Almost all the features does not have a normal distributed KDE (*fM3Long*, *fM3Trans*, *fAlpha*, etc...). From the scatter plots, it is not possible to see any evident separation of classes. Moreover, in some scatter plots (for example *fSize-fAlpha*) the blue dots (Gamma) are either mixed with the orange dot (Hadron) or completely not visible.

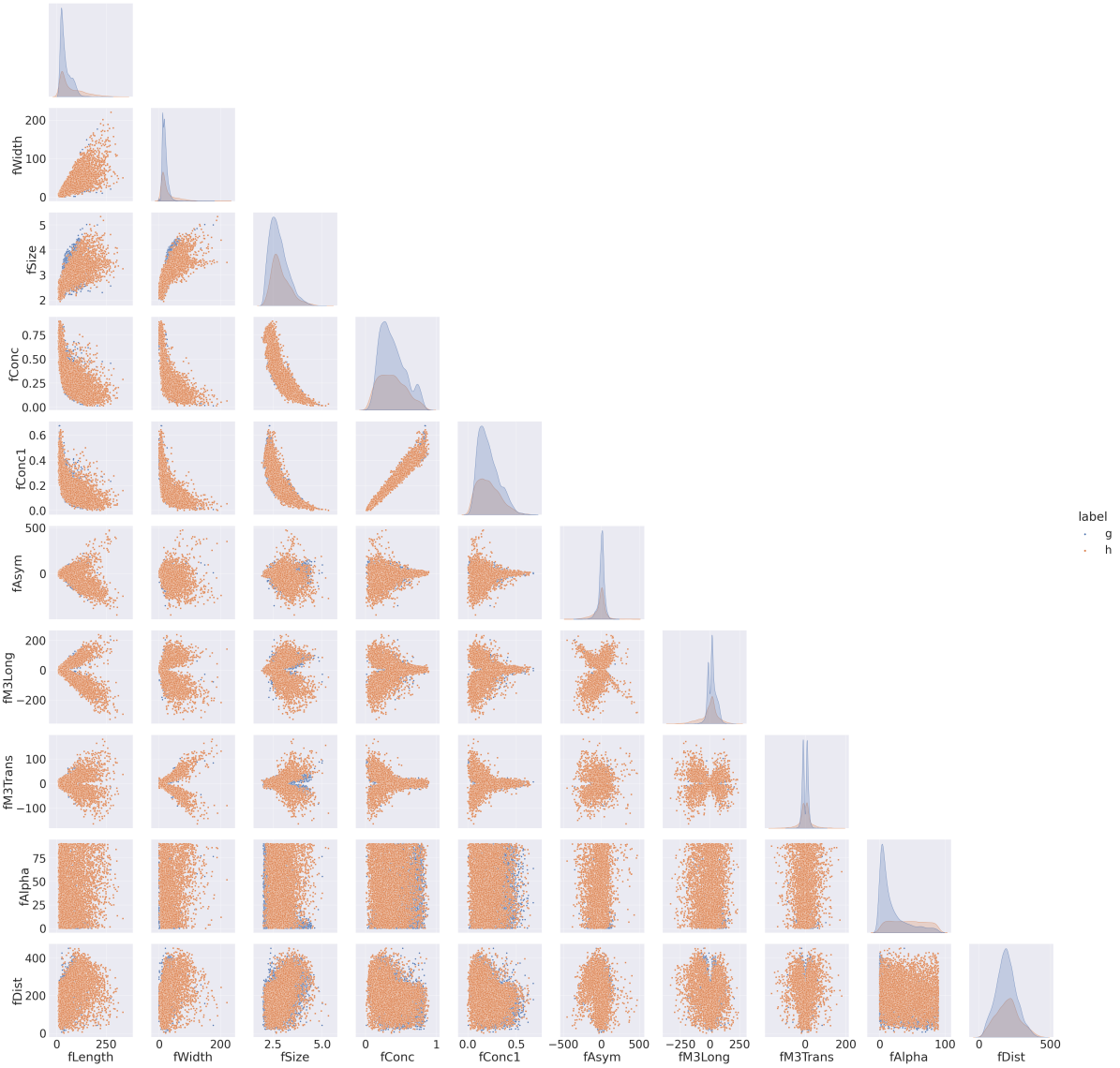


Figure 3: Pairplot between features. On the diagonal there are the KDEs plot. On the lower triangle there are the scatter plots between the selected two features. All the plots use the class label as semantic variable that is mapped to determine the color of plot elements.
Source: made by me

Useful for the analysis is also to understand if there is some correlation between features. The correlation measurement used is the Pearson correlation (value between -1 and 1). it is a measure of linear correlation between two sets of data. It is the ratio between the covariance of two variables and the product of their standard deviations:

$$Cov(x,y) = \frac{1}{n-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}) \qquad \Rightarrow \qquad PearsonCoefficient = \rho = \frac{Cov(x,y)}{S_x S_y}$$

High ρ correlation means that the two features are linearly correlated, that is, to be related in such a manner that their values form a straight line when plotted on a graph. In Figure 4 is shown, as heatmap, the correlation between features. The features more correlated are:

- **fConc**: [continuous] ratio of sum of two highest pixels over fSize

- **fConc1**: [continuous] ratio of highest pixel over fSize

We could reasonably expect this high ρ by their definitions. We can safely remove *fConc1*.

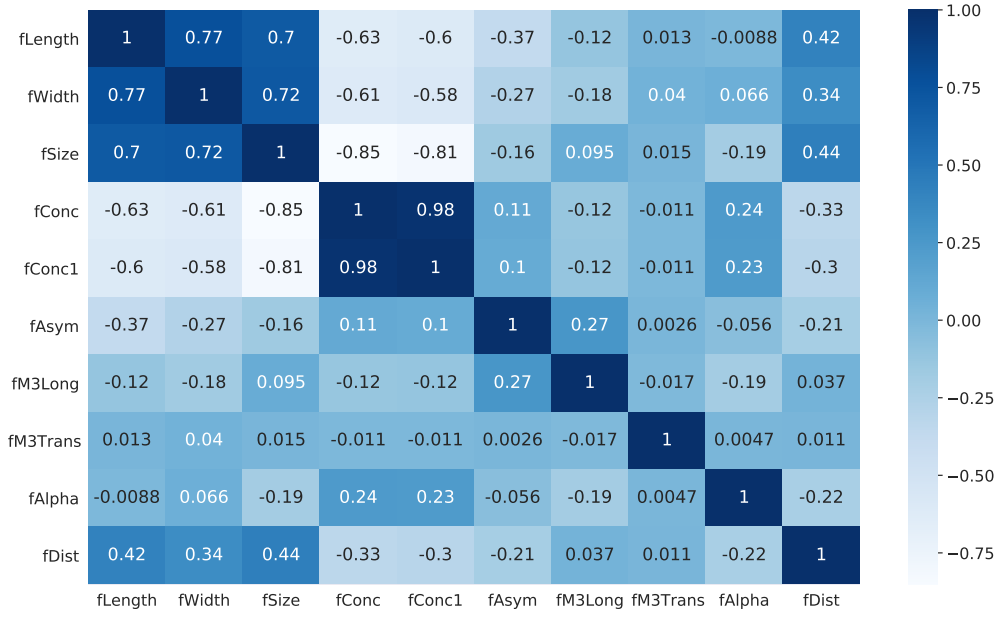


Figure 4: Heatmap of the correlation features using the Pearson correlation.
Source: made by me

Crucial is also to understand if the dataset that we are analyzing is balance or imbalance. It is "balanced" when it contains equal or almost equal number of samples from each class, "unbalanced", as in this case, if the samples from one of the classes outnumbers the other. Figure 5 clearly shows the imbalance of the dataset:

- g = gamma (signal): 12332 => 65%
- h = hadron (background): 6688 => 35%

This kind of skewness towards the label g ma cause problems during the training. Some classifiers, such as Random Forest, fail to address this kind of problem as they are sensitive to the proportions of classes, i.e they tend to favor the class with the largest proportion of observations (majority class).

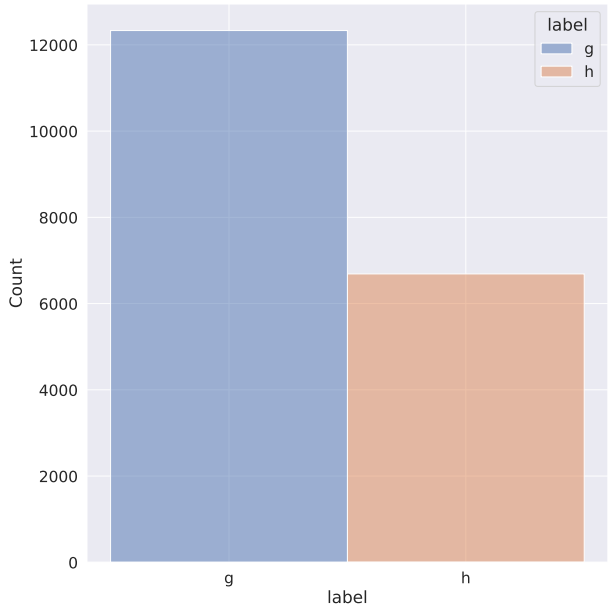


Figure 5: counts of the record having label='g' or label='h'.
Source: made by me

3 Data Preparation

3.1 Outliers removal: Local Outlier Factor

The first step of the preprocessing activity is to understand whether our dataset has outliers or not. From the Fig. 2 we can notice that there are some "distant" measures that may be measurement errors or relevant outliers. For this reason we cannot manually remove them. The *Local Outlier Factor* (LOF) algorithm is an unsupervised anomaly detection method which computes the local density deviation of a given data point with respect to its neighbors. It considers as outliers the samples that have a substantially lower density than their neighbors.

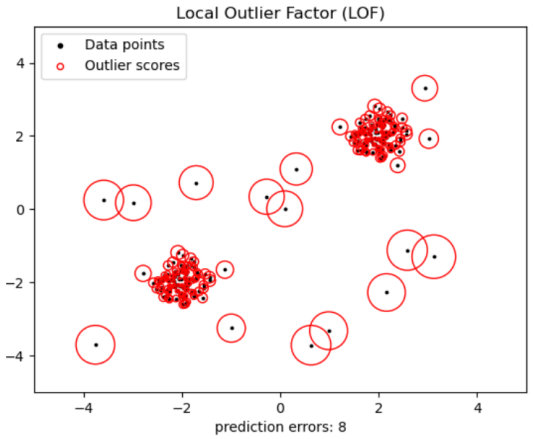


Figure 6: Example of calculation of the outlier scores by means of the Local Outlier Factor.
Source: Scikit-learn

In order to understand the algorithm we need to define:
k-distance of A $[dist_k(A)]$: it represents the distance between its k-nearest neighbors.
k-distance neighborhood of A $[N_k(A)]$: They are all the objects that are inside the sphere with radius equal to $k - dist_k(A)$. Mathematically speaking:

$$N_k(A) = \{A' | A' \in D, dist(A, A') \leq dist\_k(A)\}$$

Reachability distance from A to A' $[reachdist_k(A, A')]$: it is the true distance between A and A' if it is greater than the $dist_k(A)$:

$$reachdist_k(A, A') = max\{dist_k(A), dist(A, A')\}$$

Local reachability distance of A $[lrd_k(A)]$: It is the inverse of the average of the reachability distances of all the neighbors of A. if it is high it means that the neighbors of A are far away and A is in a low density region.

$$rd_k(A) = \frac{\left\| N_k(A) \right\|}{\sum_{A'}^{N_k} reachdist_k(A', A))}$$

Local Outlier Factor score $[LOF_k(A)]$: it is the average of the local reachability distance values of all the neighbors of A divided by the local reachability distance of A.

$$LOF_k(A) = \frac{\sum_{A'}^{N_k} lrd_k(A')}{lrd_k(A) \left\| N_k(A) \right\|}$$

The algorithm, after the calculation of the *Local Outlier Factor score*, simply define the object as outliers if the score is greater than 1, not an outlier otherwise. The outliers removed from LOF are measurement erros because 'distant' from all the other measurements.

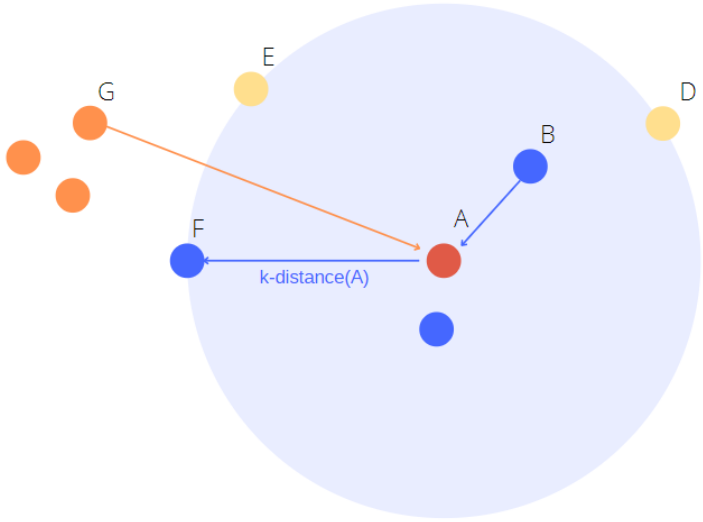


Figure 7: Example used to visually explain the LOF algorithm. The blue point represent the 3-nearest neighbor of A. The k-distance(A) is the distance between A and the third neighbor. The $reachdist_k(A,G)$ is the actual distance between point G and A. The $reachdist_k(A,B)$, instead, is the k-distance(A). $N_k(A)$ with k=3 contains all the blue and yellow circles (even if E, D are not the nearest neighbors).
Source: made by me

3.2 Divide the dataset in Training and Test parts

Before starting the effective preprocessing of our dataset we need to split it in two parts. The training part will be used for learning the parameters of the preprocessing methods (for example the mean and standard deviation used in RobustScaler) and for training and validate the dataset. The test set instead will be used only for testing our model. The dataset is splitted in this early stage in order to create a model as general as possible. If the dataset is splitted later, our model could learn some statistics about the distribution of the test set during the training ('leaking' information).

Usually, the training-test split is performed by randomly sample records from the dataset. However, in this case this will cause a generalization error because we could obtain a test dataset which does not have the real proportion between gamma and handron (65% and 35% respectively). For this reason it is performed a stratified sampling, i.e. the returned sets contain approximately the same percentage of samples of each target class as the complete set.

3.3 Feature scaling: RobustScaler

Many machine learning estimators work better with standardized dataset. Typically this is performed by removing the mean and scaling to unit variance. However, with the presence of the outliers the sample mean / variance can be influenced in a negative way. In such cases, the median and the interquartile range often give better results.

The RobustScaler removes the median and scales the data according to the Interquartile Range (IQR). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Centering and scaling is applied independently on each feature by computing the appropriate statistics on the samples in the training set. Median and interquartile range are then stored to be

used on the Test set.

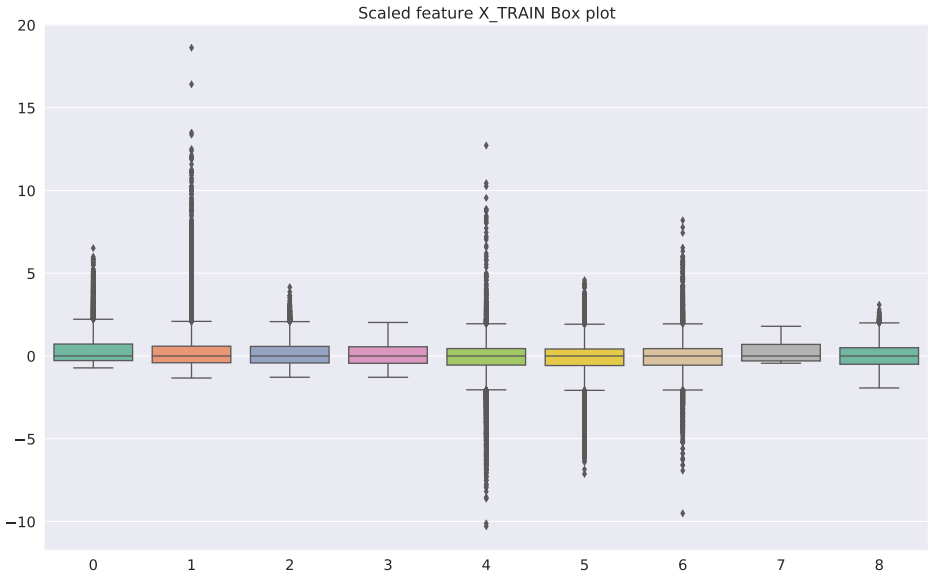


Figure 8: Boxplot of the features in the training set after the application of RobustScaler. *Source: made by me*

3.4 Dimensionality reduction: PCA

PCA is an unsupervised technique commonly used for dimensionality reduction and the reduction is performed by means of a linear transformation, i.e. $x \mapsto Wx$ where $x \in \mathbb{R}^d$ and $Wx \in \mathbb{R}^n$ with $n < d$. There are different definition of PCA, the following sees the PCA as an autoencoder (compression and recovery). The goal of this definition is to minimize the differences between the recovered vector and the initial vector in the least square sense (best projection).
Given:

- m is the total number of sample in the dataset
- $x_1, x_2, \dots, x_m$ where $x_i \in \mathbb{R}^d$ being the sample vectors
- $W \in \mathbb{R}^{n,d}$ with $n < d$ the compression matrix
- $Wx_i \in \mathbb{R}^n$ the compression of x_i

Then a second matrix, $U \in \mathbb{R}^{d,n}$ can be used to *approximately* recover x, producing $\widehat{x}$:

$$y = Wx \quad \Rightarrow \quad \widehat{x} = Uy$$

The PCA tries to find the compression matrix W and the recovering matrix U so that:

$$\underset{W \in \mathbb{R}^{n,d}, U \in \mathbb{R}^{d,n}}{argmin} \sum_{i=1}^{m} \left\| x_i - UWx_i \right\|_2^2$$

in other words, the total squared distance between the original and the recovered vector is minimal. The solution of this optimization problem is given by a *Lemma* which says that the solution respect these two conditions:

1. the **columns** of **U** are **orthonormal** $\Rightarrow U^T U = I \in \mathbb{R}^{n,n}$
2. $W = U^T$

Given this lemma, we can rewrite the optimization problem by changing W with U^T:

$$\underset{U\in\mathbb{R}^{d,n}, U^TU=I}{argmin} \sum_{i=1}^{m} \left\| x_i - UU^T x_i \right\|_2^2$$

With simply algebraic operation we find that:

$$\left\| x - UU^T x \right\|^2 = \left\| x \right\|^2 - trace(U^T x x^T U)$$

where the trace of a matrix is the sum of its diagonal entries. Since the trace is a linear operator, we can rewrite the minimization problem as a maximization problem on the trace:

$$\underset{U\in\mathbb{R}^{d,n}, U^TU=I}{argmax} \; trace(U^T \sum_{i=1}^{m} x_i x_i^T U)$$

At this point we can define the **scatter matrix** as $A = \sum_{i=1}^{m} x_i x_i^T$ and since this matrix is symmetrix, it can be written using its spectral decomposition:

$$A = VDV^T \;\; where \; D \; is \; diagonal \; and \; V^T V = VV^T = I$$

The elements of D are the eigenvalues of A instead the columns of V are the respective eigenvectors. We can safely assume that the diagonal elements are sorted as $D_{1,1} \geq D_{2,2} \geq ... \geq D_{d,d}$. It also holds that that $D_{d,d} > 0$ because A is positive semidefinite. Now that we have all the elements, we can claim the solution of the optimization problem which is given by the followin Theorem:
Let:

- $x_1, x_2, ..., x_m$ *where* $x_i \in \mathbb{R}^d$ *be arbitrary vecotrs*
- $A = \sum_{i=1}^{m} x_i x_i^T$ *be the scatter matrix*
- $u_1 \geq u_2 \geq, ..., \geq u_n$ *be the n largest eigenvectors of the matrix A*

Then the solution to the PCA optimization problem is to set U to be the matrix whose columns are $u_1, u_2, ..., u_n$ *and to set* $W = U^T$.

It is also possible to define PCA by means of the variance. Namely, we project the data to n vectors in a way that the direction of the first vector (first principal component) is equal to the direction of the largest variance of the data. The following vector (second principal component) is

orthogonal to the first one and represent the largest variance of the remaining subspace and so on so forth for the others. At the end, the principal components form an orthonormal basis in which the dimensions are uncorrelated. The optimization problem can be written as follows:

$$\begin{gathered} find\left\{Pc_1, Pc_2, ..., Pc_d\right\} \subset \mathbb{R}^d \text{ such that:} \\ max(Var(Pc_1)) \text{ and} \\ max(Var(Pc_2)) \text{ and } Cov(Pc_1, Pc_2) = 0 \\ ... \\ max(Var(Pc_k)) \text{ and } \underset{\forall j<k}{Cov(Pc_k, Pc_j)} = 0 \end{gathered}$$

For the sake of brevity, it is not reported the entire solution of this optimization problem but only the first maximization is solved.
Let:

- $x_1, x_2, \dots, x_m$ where $x_i \in \mathbb{R}^d$ be arbitrary vecotrs centered ($\mu = 0$)
- $A = \sum_{i=1}^{m} x_i x_i^T$ be the scatter matrix
- a_i the direction of Pc_i ($Pc_i = a_{i,1} p_1 + a_{i,2} p_2 + ... + a_{i,d} p_d$)

Then we want to find $max(Var(Pc_1))$, mathematically speaking:

$$\begin{cases} \max\limits_{a_1} \quad a_1 A a_1^T \\ a_1^T a_1 = 1 \quad (\|a_1\| = 1) \end{cases}$$

The second equation is needed otherwise, since we are looking for the max, the function will grow just by increasing $\|a_1\|$, instead we are interested in the direction of the component.
For solving it we will use the Lagrangian relaxation:

$$L(a_1, \lambda) = a_1 A a_1^T - \lambda(a_1^T a_1 - 1)$$

Calculating the derivatives:

$$\begin{cases} \frac{\partial}{\partial a_1} L(a_1, \lambda) = 2Aa_1 - 2\lambda a_1 \\ \frac{\partial}{\partial \lambda} L(a_1, \lambda) = a_1^T a_1 - 1 \end{cases}$$

Putting the derivatives equal to zero, we obtain:

$$\begin{cases} Aa_1 = \lambda a_1 \\ a_1^T = a_1 \end{cases}$$

So at the end, the Lagrangian coefficient λ is the maximum eigenvalue of the scatter matrix A and a_1 is the respective eigenvector. For the other passages of the solution we need to set, in the optimization problem, also: $a_j^T A a_i = 0$ with $j > i$ (a_j and a_i are uncorrelated).

At the end we obtain the same solution of the previous definition ($\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_d > 0$) but now the λ_is have different meaning: they represent the explained variance by the principal components. These different definitions leads to different approaches when calculating the principal components. In the first definition, we select n (the num. of components) at the beginning and then we find the best projection. In the second one, instead, we first compute all the eigenvalues of the scatter matrix, and then we select the number of components according to the proportion of variance we want to explain [Fig. 9].

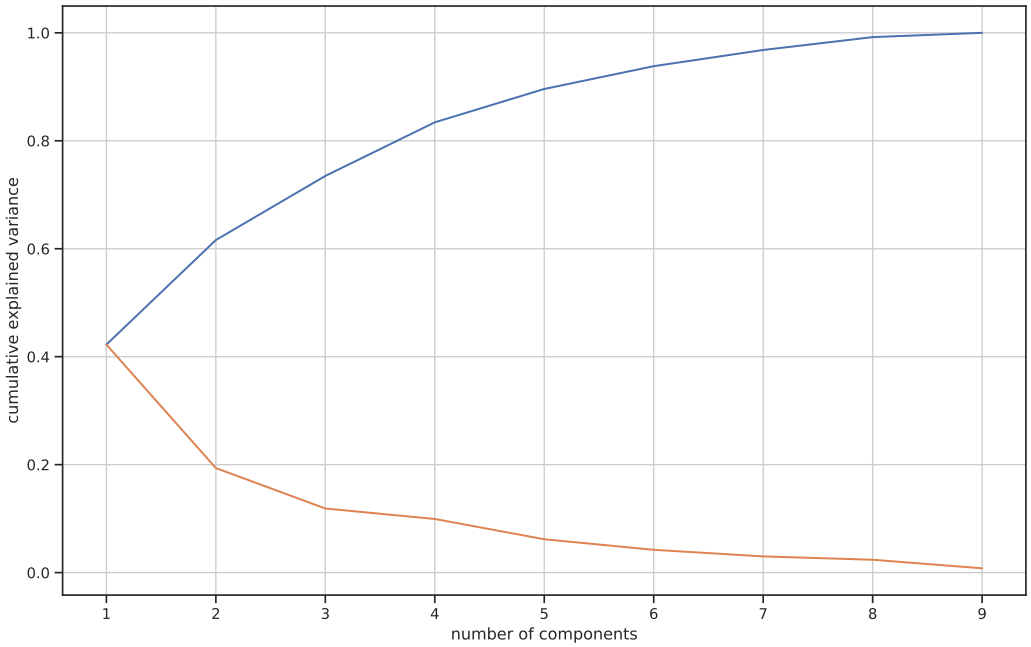


Figure 9: Proportion of variance explained by each component (Orange line). Cumulative variance (Blue Line). With only 5 components we can explain more than 80% of the variance inside the dataset
Source: made by me

3.5 Balancing the dataset : Synthetic Minority Oversampling Technique

We have seen that the dataset is imbalanced towards the class Hadron (65% g and 35%h). There are three possibilities to address this kind of skewness:

- Undersampling the majority class (Gamma)
- Oversampling the minority class (Hadron)
- Leave the dataset imbalanced

Undersampling the majority class in this case it is nor recommended because the goal of the classification is not only to correctly classify Gamma but also minimizing the error on Hadron since classifying a background event (h) as signal (g) is worse than classifying a signal event as background. For this reason, it is better to keep as many records as possible with label (Gamma).
For the same reason of above, the third solution has to be excluded since leaving the dataset imbalanced towards g will tend to favor this class respect to h, i.e. leading to more wrong prediction for h.
Oversampling the majority class, instead, seems to be the correct choice for this dataset. The selected algorithm for oversampling is the Synthetic Minority Oversampling Technique (SMOTE). Conceptually the algorithm is quite simple:

- For each object of the minority class, finds its k-nearest neighbors
- Connects the object and its neighbors with high-dimensional lines
- Creates new samples on these high-dimensional lines $X_{new} = x_i + \alpha(x_j - x_i)$ with $i \neq j$, x_j is a k-nearest neighbor of x_i and $\alpha \in [0,1]$

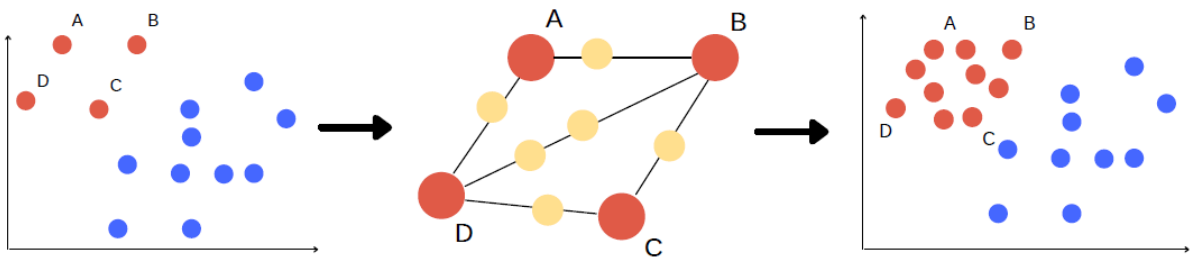


Figure 10: The figure on the left shows an example of imbalance dataset. In the central figure is shown as SMOTE works (the yellow circles are the new samples). The third figure instead represent the balanced dataset after using SMOTE.
Source: made by me

There is still a detail that is important to make clear when using technique for balancing the dataset. After choosing the correct solution for your dataset, you must apply it only on the training set because otherwise you are changing the real proportion of the classes and, therefore, the model will almost surely fail in real scenario.

4 Model selection

4.1 Empirical Risk Minimization

In this analysis, we will explore two type of models, the one based on the Empirical Risk Minimization (Random Forest, SVM) and the statistical model (Logistic Regression). In this section is explained the ERM paradigm.
Let:

- X be the set of objects
- Y be the set of labels
- D the **unknown** distribution where X is sampled
- $f(\cdot) : X \mapsto Y$ be the labelling function
- $S = ((x_1, y_1), (x_2, y_2), ...(x_m, y_m))$ be the **finite** sequence of training input composed of m elements
- $h : X \mapsto Y$ be the predictor rule

then we can defined the **true error** as:

$$L_{D,f}(h) = \underset{x\sim D}{P}[h(x) \neq f(x)] = D\left(\{x : h(x) \neq f(x)\}\right)$$

However, since we do not know neither D nor $f(\cdot)$ we cannot use this formula.
Since the learner can use only S, it makes sense to search for a solution that works well on the training set, the training error is defined as follow:

$$L_S(h) \;=\; \frac{\left|\{\; i \in [m] \;:\; h(x_i) \neq y_i \;\}\right|}{m}$$

where $[m] = \{1, 2, ..., m\}$

This learning paradigm (find a predictor h_S that minimizes $L_S(h)$) is called **Empirical Risk Minimization**. We fix in advance $H :=$ set of all possible h:

$$h_S = \underset{H}{ERM}(S) = \underset{h\in H}{argmin}\; L_S(h)$$

4.2 No-Free-Lunch Theorem

Why should we choose a specific learner for a specific problem? Does not exist an universal learner? Unfortunately, this universal learner does not exist and this is stated by the **No-Free-Lunch** theorem: Let:

- A be any learning algorithm for a **binary** classification with respect to the 0-1 loss over a domain X
- m be any number smaller than $\frac{|X|}{2}$ representing a training set size

Then, there exist a distribution D over $X \times \{0,1\}$ such that

- there exists a function $f : X \times \{0,1\}$ with $L_D(f) = 0$
- With probability of at least $1/7$ over the choice of $S \sim D^m$ we have that $L_D(A(S)) \geq \frac{1}{8}$

This theorem states that for every learner, there exists at least one task on which it fails, even if there exists another learner f which successfully solve that task ($L_D(f) = 0$).
Now the question is: how can we prevent to select the wrong learner for the task?
We can try to bypass the NFL theorem by using our prior knowledge about a specific learning task. This prior knowledge can be expressed by restricting the hypothesis class H. However, we cannot restrict H to few predictor rules because we could be discarding a possible valid h_S for the learning task. This trade-off is discussed in the next section.

4.3 Bias Complexity Trade-off

It is possible to decompose the true error in the following way:

$$L_D(h_S) = \varepsilon_{app} + \varepsilon_{est} \text{ where } \varepsilon_{app} = \min_{h \in H} L_D(h), \quad \varepsilon_{est} = L_D(h_S) - \varepsilon_{app}$$

Where ε_{app} is the approximation error, ε_{est} is the estimation error and h_S an ERM_H hypothesis.

The ε_{app} represents the minimum risk achievable by a predictor in the hypothesis class H. It represents the risk that we have because we restrict our learning to a specific H (how much **inductive bias** we have). Enlarging H can reduce the approximation error (small H might lead to *underfitting*).

The ε_{est} is the difference between the approximation error and the error achieved by h_S. This kind of error is present because we minimize according to the training error which is an estimate of the true risk. The quality of this error depends on the size of the training set (more samples we use, more accurate is the estimation of the true error) and on the size of the hypothesis class (richer H may lead to *overfitting*).

At the end, our goal is to minimize the total risk, but we have just seen that it is not possible to have low values for both errors (enlarging H produces less ε_{est} but more ε_{app} and this is true also on the other way around). This kind of trade-off is called **bias-complexity trade-off**.

4.4 Cross Validation: stratified cross validation

In the bias complexity trade-off, to define the estimation error is used the true error ($L_D(h_S)$). However, the distribution D is unknown (it is the distribution that we are trying to estimate with our learner). For this reason some re-sampling methods are used to provide an estimate of the true error. We have already splitted our dataset in training and test set in order to have the test set as representative as possible of the reality.
Nevertheless, during the training is used a stratified cross validation which is used to have an idea of the bias-complexity trade-off during the training and for studying the solutions in case of underfitting/overfitting.

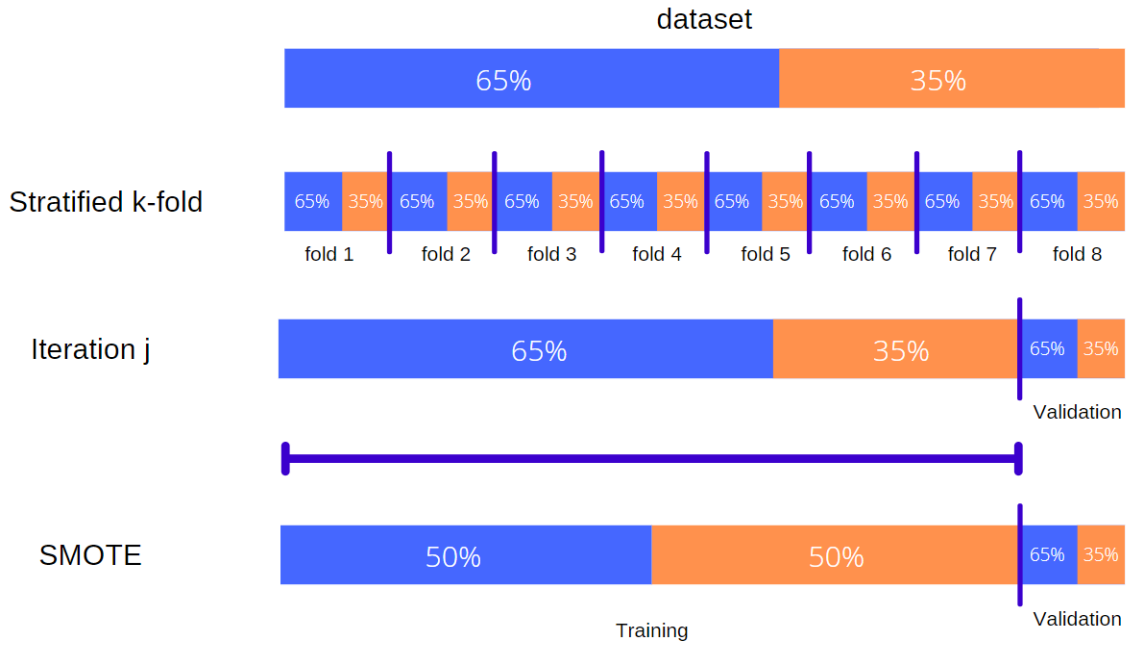


Figure 11: The first line represent this dataset (65% gamma and 35% hadron). The second line is the application of the stratified k-fold with k = 8 (eight stratified subsets are created). The third line represents the j-th iteration when, during the training, the stratified CV object is called. Only the training set is oversampled (using SMOTE), instead the validation one is untouched in order to have a correct estimation of the true error.
Source: made by me

In the Fig. 11 is visually explained how stratified cross validation works and, more importantly, where SMOTE is applied. For each iteration one fold is used for validation and the others for training. The same fold is used only one time for validating. In the figure, for simplicity, it is only reported one iteration.
At the end the set that perform better than the others is used to print the best metric score.
Setting the parameter k is crucial because, as is possible to see from the image, an high value of k will produce a small validation dataset that will lead to a poor estimation of the true error. The other way around instead will create a small training dataset that may lead to overfitting. For this analysis k has been set to 11.

4.5 Random Forest

Random forests or random decision forests are an ensemble learning method it operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees.

The random forest starts from the idea of combining prediction functions trained on multiple datasets in order to improve the overall prediction accuracy. Given a regression tree and B independent and identically distributed (iid) copies $\tau_1, \tau_2, ..., \tau_B$ of a training set τ, we can train B separate regression models, obtaining $g_{\tau_1}, g_{\tau_2}, ..., g_{\tau_B}$ prediction functions. By the law of large numbers, as $B \to \infty$ the average prediction function converges to the **expected prediction function**:

$$g_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B} g_{\tau_b}(x) \quad \underset{B\to\infty}{\to} \quad g^{\tau} := E_{g_\tau}$$

This lead to the theorem : **Expected squared-error generalization risk**

$$E(Y - g_\tau(X))^2 \geq E(Y - g^\tau(X))^2$$

Where Y, X are random variables. This theorem states that the expected error for a prediction function $(g_\tau(\cdot))$ cannot be less than the expected error of the expected prediction function $(g^\tau(\cdot))$. Unfortunately, the assumption for building $(g^\tau(\cdot))$ are almost never satisfies because it is not always possible to find multiple iid copies of the same dataset.
In order to bypass this problem we can use **bagging**. Namely, we can obtain random training sets by **sampling** from a **single** training set with **replacement** until they have the same size of the original dataset. For each resulting set we can obtain $g_{\tau_1^*}, g_{\tau_2^*}, ..., g_{\tau_B^*}$. The average prediction function then is calculated as follow:

$$g_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} g_{\tau_b^*}(x) \quad \underset{B\to\infty}{\to} \quad g^{\tau^*} := E_{g_\tau^*}$$

g_{bag} is an **approximation** of g_{avg} as a consequence, for stable predictors, g_{bag} may perform worse than g_τ.

However, bagging introduce another type of problem that was not present with iid training sets. More precisely if we set (for brevity) $Z_b = g_{\tau_b}(x), b = 1, ..., B$ with $Var(Z_b = \sigma^2)$, we can calculate the variance of the average prediction functions $g_{avg}(x)$ (calculated on B iid training sets):

$$Var\left[\frac{1}{B}\sum_{b=1}^{B} Z_b\right] \;=\; \frac{1}{B^2}Var(Z_b)\sum_{b=1}^{B} 1 \;=\; \frac{1}{B}Var(Z_b) \;=\; \frac{\sigma^2}{B}$$

However, in Bootstrapped datasets (τ_b^*), Z_b^* will be correlated:

$$Var(Z_b) = \rho\sigma^2 + \sigma^2\frac{1-\rho}{B}$$

where ρ is the pairwise correlation. This happens because the $\tau_b^*, b = 1, ..., B$ are identically distributed but not independent.

This problem is particularly relevant with decision trees, if there exists a feature that provides a good split, it will be selected by all the decision trees, ending up with an **highly correlated predictions** ($\rho >> 0$).
The solution then is to **decorrelate** the trees by including only a subset of features.

So at the end, the random forest creates a certain number of trees and then it exploits the bagging technique to train each tree on a different bootstrapped set. In addition, for building uncorrelated tree, for each split only m randomly parameters are used (usually $m = \sqrt{p}$ or $m = \frac{p}{3}$ where p is the total number of predictors).

4.6 SVM

Support Vector Machine is a supervised method that solves an optimization problem: Find the hyperplane that maximizes the margin between two classes in the feature space. Where the margin of an hyperplane with respect to a training set is the minimal distance between a point and the hyperplane (large margin leads to better predictions).

The **Hard SVM** is the ERM hyperplane with the largest possible margin. The distance between a point x and an hyperplane (W, b) with $\|W\| = 1$ is defined as:

$$|< w, x > +b| = \frac{|w_1 x_1 + w_2 x_2 + ... + w_n x_n + b|}{\sqrt{w_1^2 + w_2^2 + ... + w_n^2}}$$

the closest point in the training set to (w, b) is then:

$$\min_{i \in [m]} |< w, x_i > +b|$$

the Hard SVM optimization problem is defined as:

$$\begin{aligned} &\operatorname*{argmax}_{(w,b)|\|w\|=1} \min_{i \in [m]} |< w, x_i > +b| \\ &s.t. \;\; y_i(< w, x_i > +b) > 0 \end{aligned}.$$

this optimization problem tries to find hyperplane that linearly separates the dataset and it has the maximum margin.
In the **separable case** there exists a solution otherwise the constraint is not satisfied.
The SVM can also be described as a **quadratic optimization problem** (the objective is a convex quadratic functions and the constraints are linear inequalities):

$$(w_0, b_0) = \operatorname*{argmin}_{(w,b)} \|w\|^2 \quad \text{s.t.} \quad y_i(< w, x_i > +b) \geq 1, \forall i$$

There is a *Lemma* that states that the solution of this quadratic optimization problem is indeed the separating hyperplane with the largest margin. The difference with the previous definition is that we are enforcing the margin to be 1, but now the units in which we measure the margin scale with the norm of w.

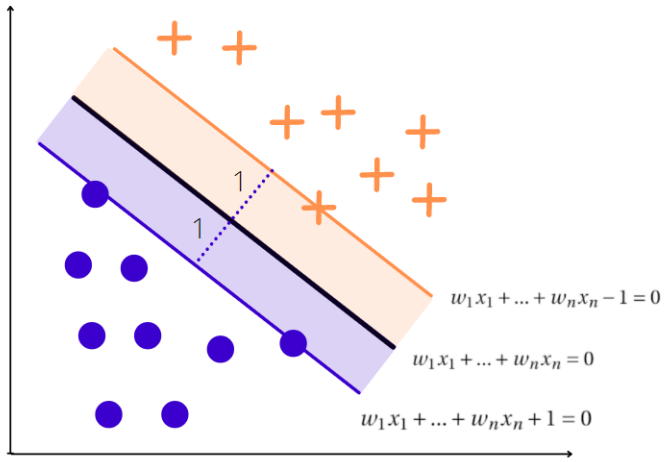


Figure 12: Figure used to visually explain the HARD SVM. *Source: made by me*

Hard SVM assumes that the dataset is linearly separable but this is almost never the case.
Soft SVM is a relaxation of Hard SVM: $y_i(< w, x_i > +b) \geq 1 - \xi_i$ where ξ_i is a *slack variable* and measures how much $y_i(< w, x_i > +b) \geq 1$ may be violated. The Soft SVM rule is defined as:

$$\begin{gathered} \underset{w,b,\xi}{argmin}\left(\lambda \|w\|^2 + \frac{1}{m}\sum_{i=1}^{m}\xi_i\right) \\ s.t \;\; y_i(< w, x_i > +b) \geq 1 - \xi_i \\ \xi_i \geq 0 \end{gathered} \cdot$$

We want to minimize both the norm of the vector and the average of the slack variables, the trade-off is controlled by λ.
The Soft SVM can be applied even if the training set is not linearly separable. We can rewrite this optimization problem as a regularize loss minimization problem.
Let:

$$l^{hinge}((w,b),(x,y)) = \max\{0, 1 - y(< w, x > +b)\}$$

$$L_S^{hinge}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\max\{0, 1 - y(< w, x_i > +b)\}$$

be respectively the hinge loss and the averaged hinge loss on a training set S. Then we can rewrite the optimization function as a regularized loss minimization problem:

$$\min_{w,b} \; \lambda \|w\|^2 + L_S^{hinge}(w,b)$$

This can be proven by considering the minimization problem over the slack variables.

The name *"Support Vector Machine"* is based on the fact that the solution of hard-SVM (w_0) is supported by the examples that are at distance $\frac{1}{\|w_0\|}$ from the separating hyperplane (see Fig. 12). These vectors are called *Support Vectors*. This is states by the **Fritz John Optimality Conditions**:

Let w_0 be the solution of the regularized loss minimization problem and let $I = i : | < w_0, x_i > | = 1$. (we are considering the homogeneous case)

Then there exist coefficients $\alpha_1, \alpha_2, ..., \alpha_m$ *such that:*

$$w_0 = \sum_{i \in I} \alpha_i x_i$$

The examples $\{x_i : i \in I\}$ *are called support vectors.* (the proof is based on applying the Fritz John lemma to the loss minimization problem)

Before starting explaining how the kernels work for the SVM, it is needed to define the duality problem (Lagrangian relaxation). We can define the dual problem on the homogeneous case without loss of generality:

$$(w_0, b_0) = \underset{(w,b)}{min} \|w\|^2 \quad s.t. \;\; y_i(< w, x_i >) \geq 1, \forall i$$

For simplicity, we can define an auxiliary function:

$$g(w) = \max_{\alpha \in \mathbb{R}^m : \alpha \geq 0} \sum_{i=1}^{m} \alpha_i (1 - y_i < w, x_i >) = \begin{cases} 0 \;\; if \;\; \forall i, y_i < w, x_i > \geq 1 \\ \infty otherwise \end{cases}$$

Now we can relax the constraint by moving it in the objective function:

$$(w_0, b_0) = \underset{(w,b)}{min} \|w\|^2 + g(w)$$

We have added the constraint because $g(w) \geq 0$ but we are minimizing, the software will try to put it to zero.
If we flip the order of min and max we have the *weak duality* however in this case also the *strong duality* is satisfied. At the end we have:

$$\underset{\alpha \in \mathbb{R}^m : \alpha \geq 0}{max} \;\; \underset{w}{min} \left(\frac{1}{2} \|w\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y_i < w, x_i >) \right)$$

When α is fixed, the optimization problem to w is *unconstrained* and the objective is an hyperpal-aboloid (then differentiable). To find the optimum we can put the gradient equal to zero:

$$\vec{\nabla} \left(\frac{1}{2} \|w\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y_i < w, x_i >) \right)$$

$$\Rightarrow w_0 - \sum_{i=1}^{m} \alpha_i y_i x_i = 0$$

$$\Rightarrow w_0 = \sum_{i=1}^{m} \alpha_i y_i x_i$$

which says that the optimal w must be in the linear span of the examples (the smallest linear subspace that contains the set).
We can now substitute the just found optimum inside the dual problem:

$$\underset{\alpha \in \mathbb{R}^m : \alpha \geq 0}{max} \;\; \left(\frac{1}{2} \left\| \sum_{i=1}^{m} \alpha_i y_i x_i \right\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y_i < \sum_{i=1}^{m} \alpha_i y_i x_i, x_i >) \right)$$

which can be rewrite as:

$$\max_{\alpha \in \mathbb{R}^m : \alpha \geq 0} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j \; y_i y_j < x_j, x_i >$$

So at the end we have a parametric problem (we want to find the parameters α) on an hyperpal-aboloid ($a_i a_j$) wich contains the scalar product of the examples.

Thanks to the scalar product, we can use the **kernel trick**. The idea is to map the feature space in an higher dimension where the problem is linearly separable, more precisely:

- Given X, choose $\psi : X \mapsto F$
- $S = (x_1, y_1), ..., (x_m, y_m) \Rightarrow \hat{S} = (\psi(x_1), y_1), ..., (\psi(x_m), y_m)$
- Train a linear predictor h over $\hat{S}$
- Predict the label of a test point x_{test} as $h(\psi(x_{test})$

This paradigm works only if we select the correct mapping function ψ. In addition, passing to an higher space can be too costly to handle. Fortunately we can use the Kernel Trick. Given an embedding ψ of some domain space X into the *Hilbert space* then the Kernel function is defined as:

$$k(x, x^{'}) = < \psi(x), \psi(x^{'}) >$$

The kernel functions are used to implement linear separators in high dimensional feature space, without having to specify points in that space or expressing the embedding ψ explicitly.
Therefore, for solving an optimization problem where there is a scalar product and it is applied ψ, we do not need any direct access to the elements in the feature space, we only need to know how to calculate the inner products in the mapped feature space, i.e. how to calculate the kernel function. With the kernel function, the size of the problem does no longer depends on the high dimensional space, but it depends on m (the size of the dataset). This bottleneck is then bypassed by deep learning.

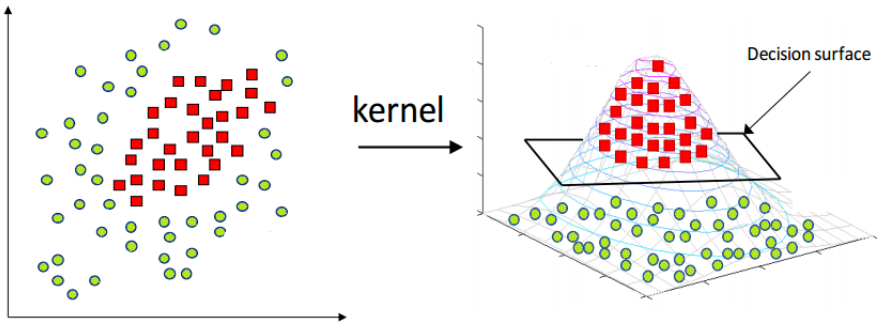


Figure 13: Example of how kernel works.
Source: *Source: What is the kernel trick? Why is it important?*

4.7 Logistic Regression

Logistic regression is a method in statistics and machine learning that returns the probability of an outcome, it is more used for classification although many more complex extensions exist.
The algorithm predicts the probability (outcome between 0 and 1) by means of a logistic function:

$$g(z) = \frac{1}{1+e^{-z}}$$

The function itself is not a classifier but can be used for classification task. More precisely, if we select a threshold (common choice $th = 0.5$), we can classify the input with probability greater than it as class one and the other way around as the other class.
For explaining the logistic function we use the *log-odds*:

$$l(x) = \log \frac{x}{1-x}$$

then for the sake of simplicity, only two binary input variables (x_1, x_2) and one binary response variable (y) are considered. Subsequently, is defined $p = P(y = 1)$ and we assume linear relationship between the predictor and the log-odds of p:

$$\log \frac{p}{1-p} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \text{ where } \theta_i \text{ is a parameter of the model}$$

We can rewrite the equation obtaining:

$$p(x) = g(\theta^T x) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2)}}$$

the coefficients θ_i of the model are calculated by means of the *Maximum Likelihood Estimation* (MLE).
During the training, the loss function that is used to tune the parameter is (usually) the *binary cross entropy loss*:

$$CE = -\frac{1}{m} \sum_{i=1}^{m} + (y_i \log p(x_i)) + (1 - y_i) \log (1 - p(x_i))$$

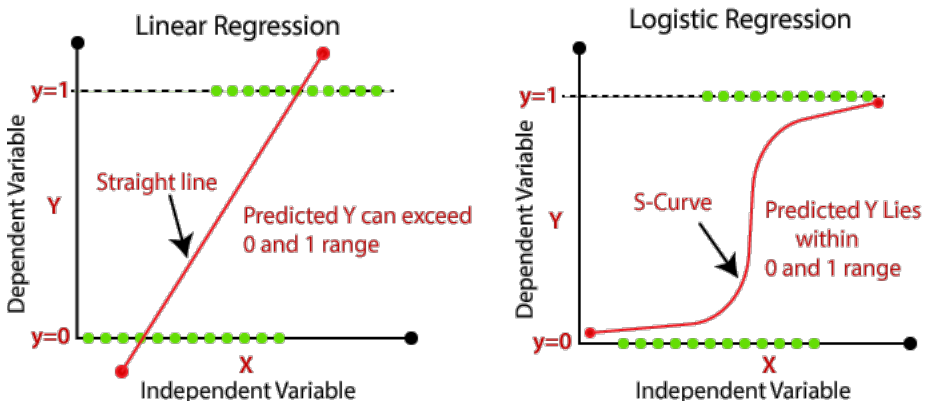


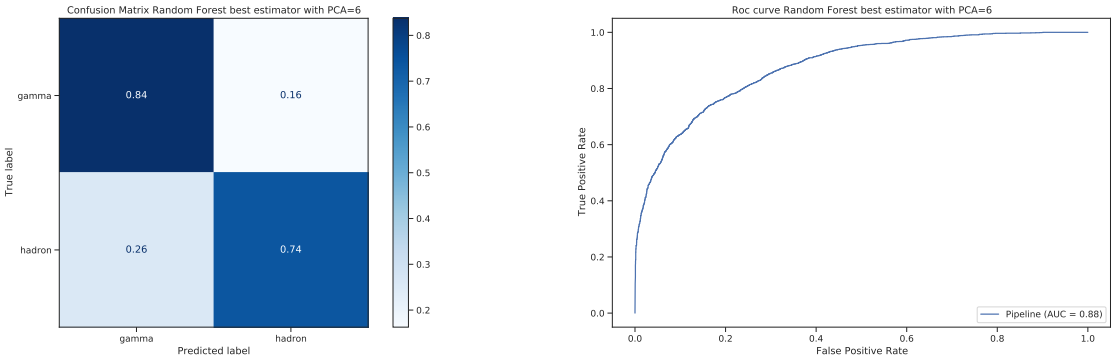
Figure 14: Linear regression vs Logistic regression
Source: *JavaPoint*

5 Results

In these section are presented the selected parameters, for each model, that produce the better performances. In addition, the confusion matrix and the RoC are also reported.
We can notice that there is a slightly improvement when the PCA is not used but this is a reasonable expectation since with six principal components (value choosen for the analysis) it is explained the 0.94% of the total variance.
The best Area under the curve is given by the Random Forest (94%) followed by the SVM with the Gaussian kernel (92%). The other way around instead we find the Polynomial SVM. Despite the high AUC, from the confusion matrix is clear that it fails the classification task.

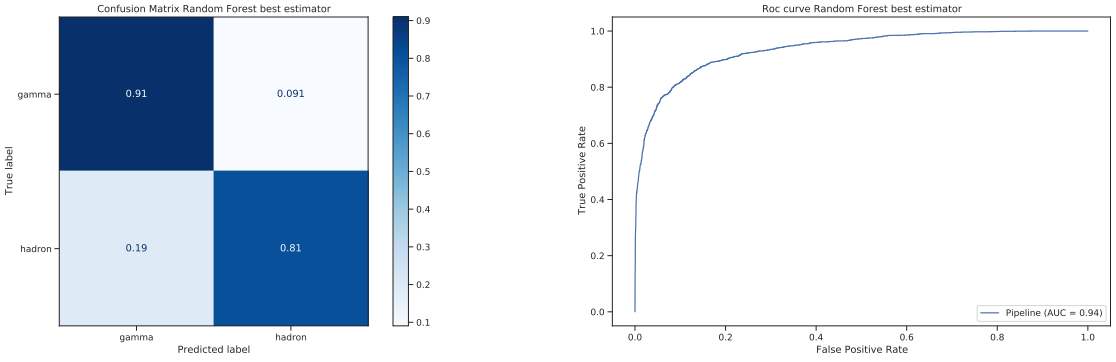
Model	Selected parameter	AUC of the Roc curve
Random Forest with PCA	*n_estimator: 150* *max_depth:15* *criterion: entropy*	0.88
Random Forest without PCA	***n_estimator: 100*** ***max_depth:20*** ***criterion: gini***	**0.94**
Linear SVM with PCA	*C: 1.5*	0.82
Linear SVM without PCA	*C: 1.2*	0.85
Polynomial SVM with PCA	*degree: 3* *C: 1.*	0.82
Polynomial SVM without PCA	*degree: 3* *C: 1.2*	0.89
RBF SVM with PCA	*C : 1.2*	0.87
RBF SVM without PCA	*C : 1.2*	0.92
Logistic Regression with PCA	*penalty: l1* *C: 0.8*	0.82
Logistic Regression without PCA	*penalty: l1* *C: 1.5*	0.85

Table 1: Best parameters for each model



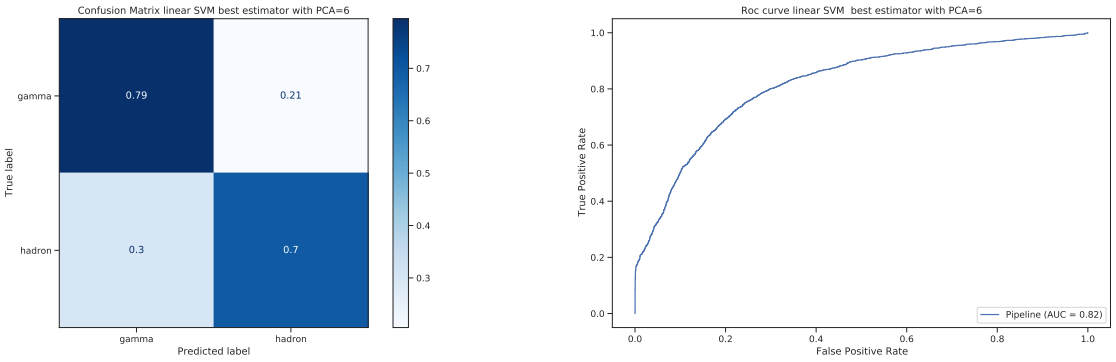
(a) Confusion matrix **(b)** ROC curve

Figure 15: Random Forest with PCA



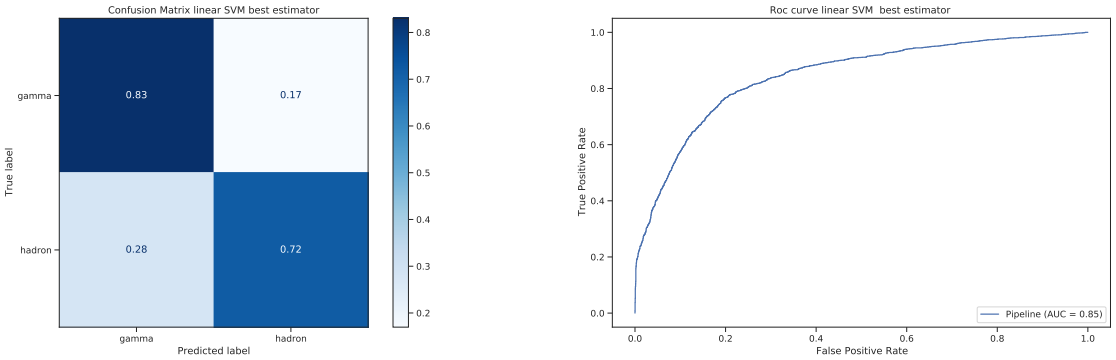
(a) Confusion matrix **(b)** ROC curve

Figure 16: Random Forest without PCA



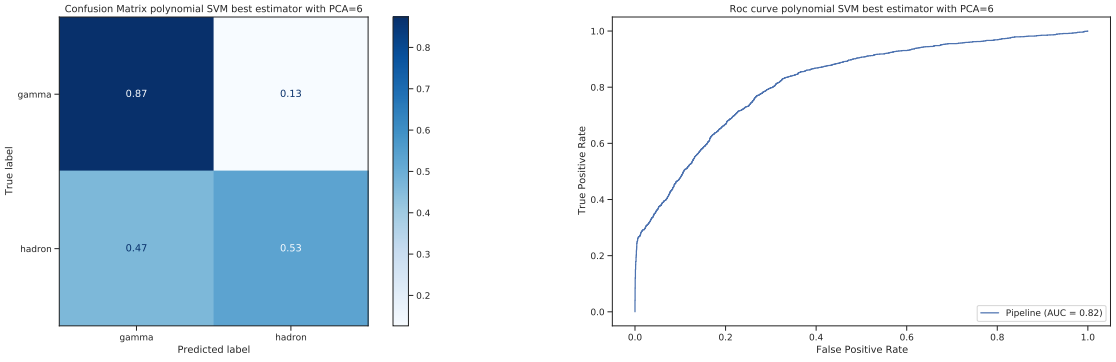
(a) Confusion matrix **(b)** ROC curve

Figure 17: Linear SVM with PCA



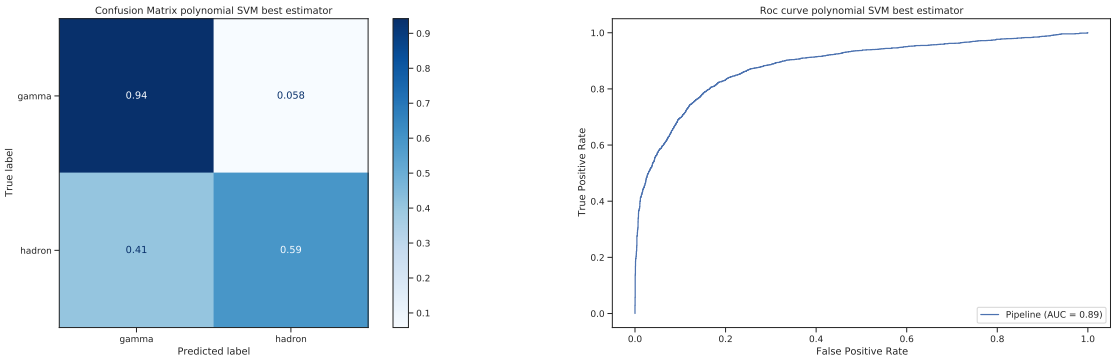
(a) Confusion matrix **(b)** ROC curve

Figure 18: Linear SVM without PCA



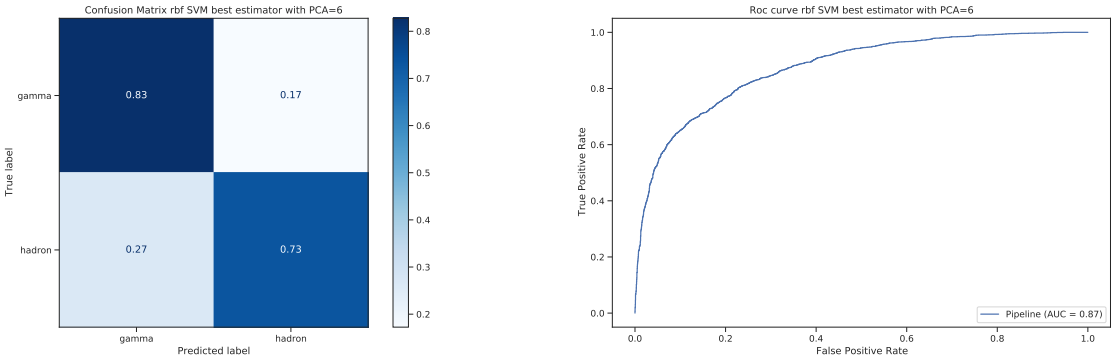
(a) Confusion matrix **(b)** ROC curve

Figure 19: Polynomial SVM with PCA



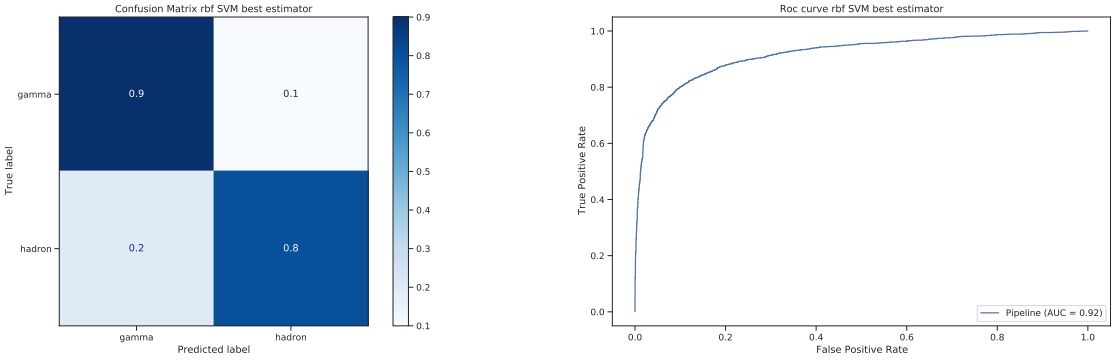
(a) Confusion matrix **(b)** ROC curve

Figure 20: Polynomial SVM without PCA



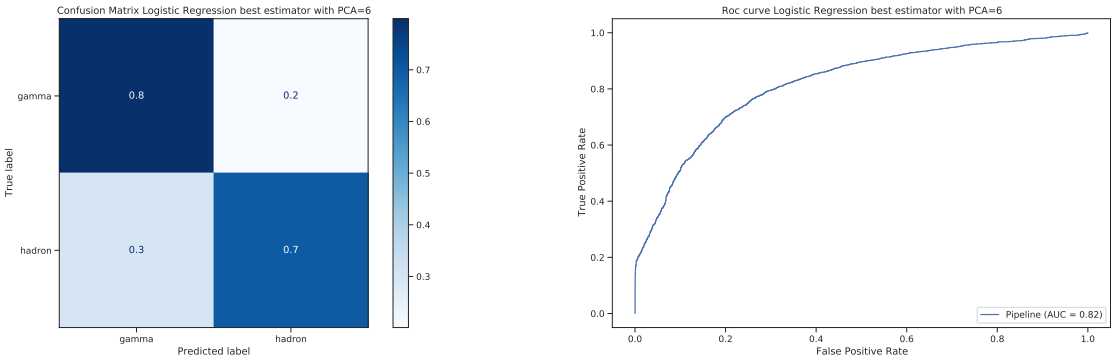
(a) Confusion matrix **(b)** ROC curve

Figure 21: RBF SVM with PCA



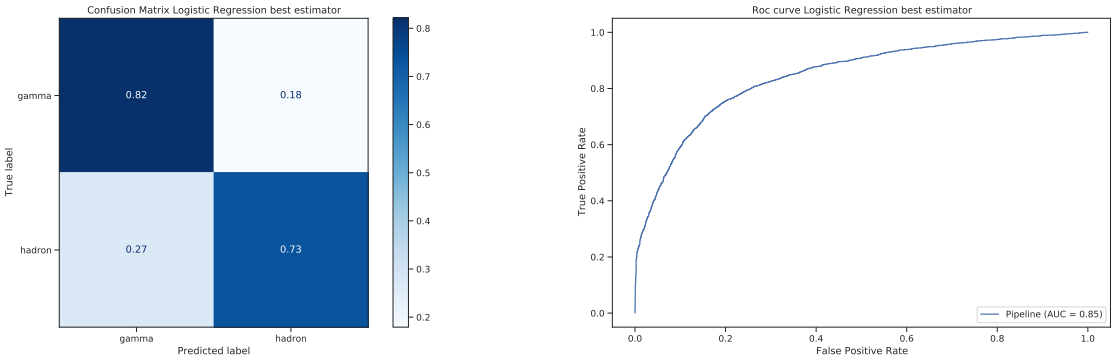
(a) Confusion matrix **(b)** ROC curve

Figure 22: RBF SVM without PCA



(a) Confusion matrix **(b)** ROC curve

Figure 23: Logistic Regression with PCA



(a) Confusion matrix **(b)** ROC curve

Figure 24: Logistic Regression without PCA