Machine Learning Project
Higgs Boson Machine Learning Challenge

LAURA MISRACHI
laura.misrachi@gmail.com

As mentioned in the description of this project on ***Kaggle***, "the goal of the Higgs Boson Machine Learning Challenge is to explore the potential of advanced machine learning methods to improve the discovery significance of the experiment. No knowledge of particle physics is required. Using simulated data with features characterizing events detected by ATLAS, [our] task is to classify events into "tau tau decay of a Higgs boson" versus "background"." In the following is highlighted my approach to this classification problem, ranging from the initial Exploratory data analysis down to the implementation of a subset of models along with the analysis of the obtained results.

Contents

List of Figures

List of Tables

I. Exploratory Data Analysis

I. Understanding our dataset

To get started, let us highlight a few points which will reveal very useful to understand our dataset:

- All variables are floating point, except PRI_jet_num which is integer (factor feature).
- Variables prefixed with PRI (for PRImitives) are "raw" quantities about the bunch collision as measured by the detector.
- Variables prefixed with DER (for DERived) are quantities computed from the primitive features, which were selected by the physicists of ATLAS
- It can happen that for some entries some variables are meaningless or cannot be computed; in this case, their value is ?999.0, which is outside the normal range of all variables. This value should therefore be understood as the encoding for 'NaN'.
- One should notice that the "Weight" feature is not present in the dataset meaning that it cannot be used for learning. We will try to understand later how it might be used for training.
- 31 variables are shared by the train and test set, 250000 samples are presented in the training set while the Kaggle test set is composed of 83332 samples.

II. Handling missing values

Before leading any further investigation on the features of our dataset, it is of primary importance to investigate the proportion of missing values. Indeed, it can be very tricky to deal with those, as 'NaN' values are usually not handled by most algorithms (except some ensembling techniques such as Gradient Boosting Decision Trees).

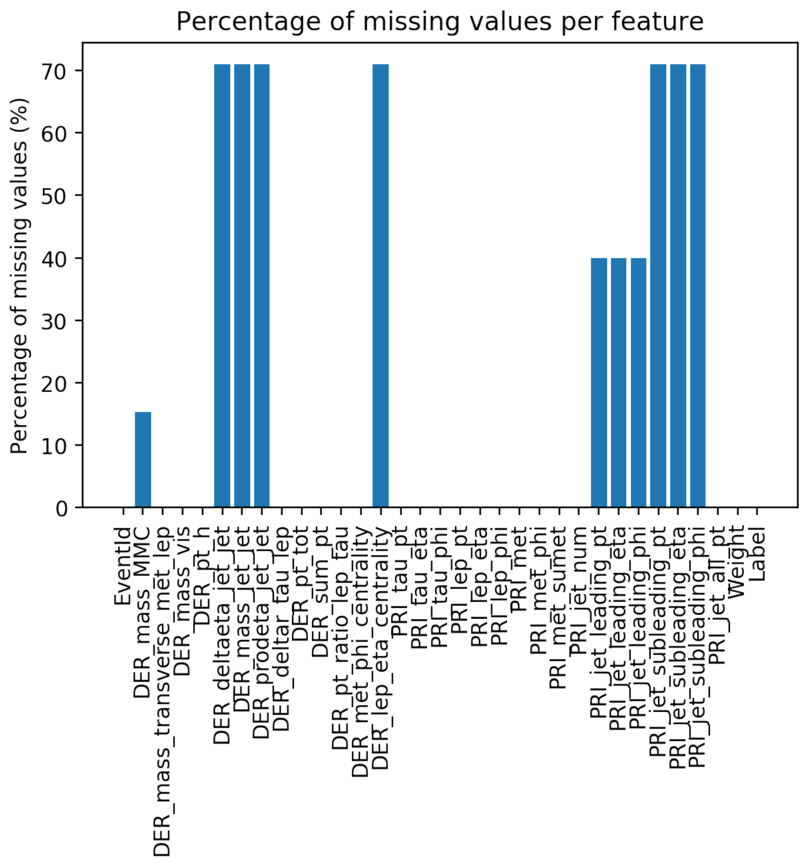

Figure 1: Proportion of missing values per feature

According to the previous plot, it seems that the proportion of missing values is substantially high for at least 6 features ($\approx 70\%$) and that the simple drop of the samples containing missing values is not acceptable. Some form of encoding can be performed for these missing values : mean encoding (replacement of NaNs by the mean value of each feature) or even max, min and zero encoding.
Besides, based on the description of the physical experiment and, let us be honest, on the strategies pointed out by others (see Kaggle : Discussion forum), the influence of the ***PRI_jet_num*** feature on the proportion of missing values - which is the only categorial feature in our data set - was further examined. Following these observations,

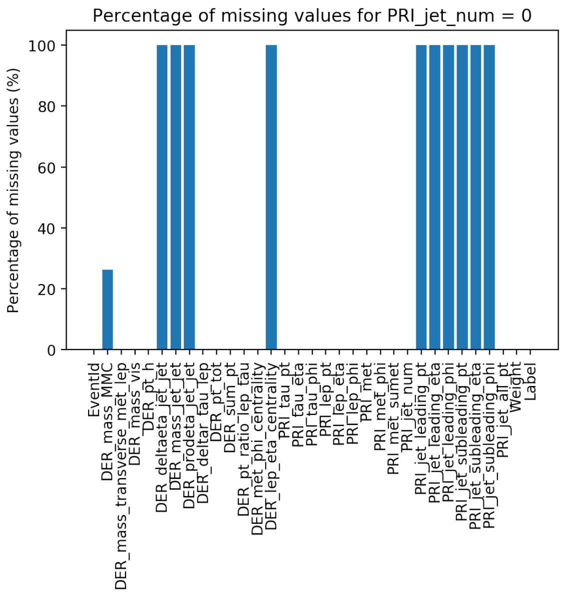

Figure 2: Missing values for PRI_jet_num = 0

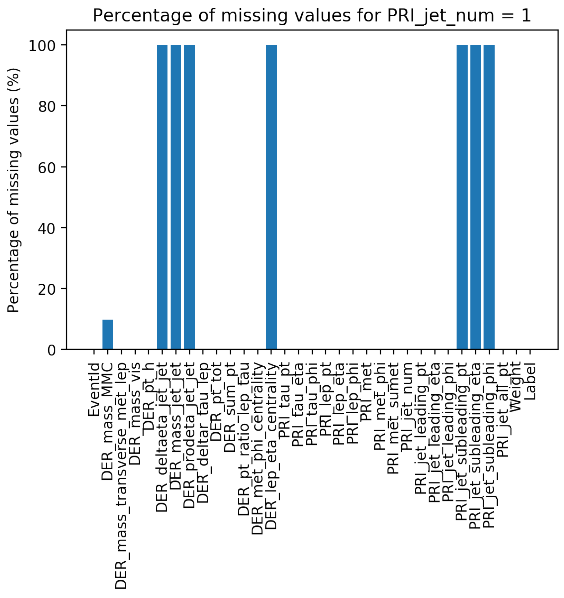

Figure 3: Missing values for PRI_jet_num = 1

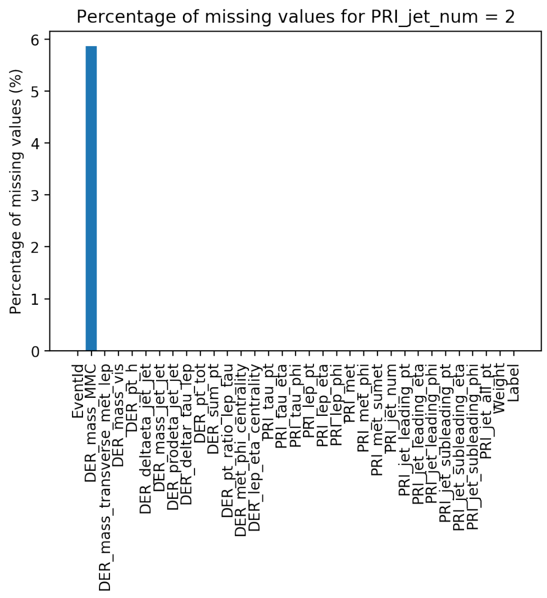

Figure 4: Missing values for PRI_jet_num = 2

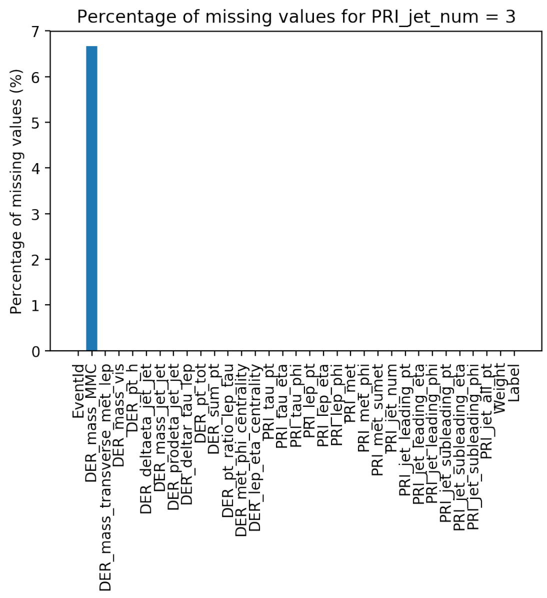

Figure 5: Missing values for PRI_jet_num = 3

we understand that the missing values (which are not the consequences of bad/wrong measurements, but are more linked to the inherent structures of the measures) can be linked with the values of the integer PRI_jet_num feature. Indeed, we observe that, for most features with missing values, these are completely missing or not missing at all, based on the values of the PRI_jet_num feature. This is true, except for the DER_mass_MMC feature for which there is, for each given value of the PRI_jet_num feature a certain percentage (> 0, < 100) of missing values. One of our idea was therefore to subset our training set (and our test set when testing our model of course) into four bins according to the value of the PRI_jet_num feature and to perform mean encoding for the remaining missing values of the DER_mass_MMC feature. To resume, two strategies were developed to handle missing values during this project:

- Direct mean encoding for all the missing values in the dataset
- Subset of our dataset into 4 bins corresponding to the value of the PRI_jet_num feature and train four classifiers on these different bins. Along with this: drop of the feature with 100% missing values, meanwhile performing mean encoding for the missing values of the DER_mass_MMC feature.

III. Label proportion

Now, it is of primary importance to investigate the repartition of the labels across our training set. It is shown in the following:

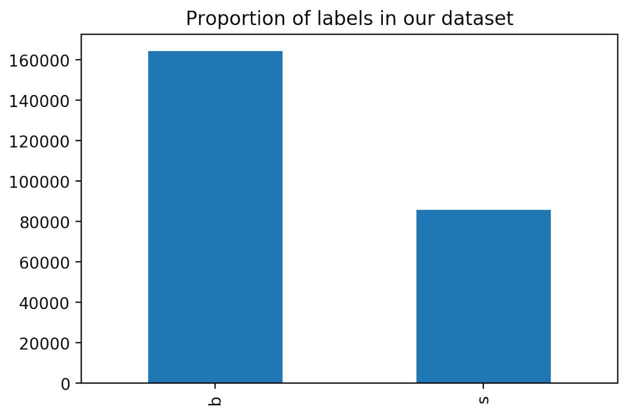

Figure 6: Repartition of the labels across our training set.

One immediately understands that our classes are not equally spread in the dataset, meaning that if we use our samples without penalizing the most frequent class, our model will be very likely to predict the most frequent class and thus achieve a poor precision score, even though the accuracy score could be quite acceptable. For the sake of clarity, we do not present the distribution of the labels for our 4 bins subset of the training set as introduced before, but one should know that the class imbalance is preserved for most of the bins. To tackle this potential issue, several solutions are available:

- If many data are available, an option could be to use less data with a more balanced (50 %) proportion of available examples. This means that many data are not exploited.
- If few data are available, an idea could be to duplicate some rows for the less frequent class. One should be careful when testing the model in that case (a duplicate row could end up in the test set accidentally...)
- A more advanced method consist in penalizing the examples from the most frequent class with a certain coefficient in the loss function. Taking the example of logistic regression, we would replace the following loss:

$$L = \frac{1}{n}\sum_{i=0}^{n}\Big(\log(1+e^{\langle x,x_i\rangle})1_{y_i=1} + \log(1+e^{-\langle x,x_i\rangle})1_{y_i=0}\Big) \tag{1}$$

  by:

$$L = \frac{1}{n}\sum_{i=0}^{n}\Big(q_1\log(1+e^{\langle x,x_i\rangle})1_{y_i=1} + q_0\log(1+e^{-\langle x,x_i\rangle})1_{y_i=0}\Big) \tag{2}$$

  with $q_0 = \frac{n}{\{\#i \quad y_i=0\}}$ and $q_1 = \frac{n}{\{\#i \quad y_i=1\}}$ and where x_i is the i th example/sample, x the minimizer of the loss function, y_i the training set label of the i th example and n the total number of examples in the training set.

Doing so, the gradient steps are larger for the less frequent class than for the other one. The latter option was chosen, as it is implemented in Scikit-learn.

IV. Univariate distributions of the features and feature engineering

Before doing some feature engineering, we thaught of plotting their univariate distribution. They are presented in the following plot, where the -999 values in the dataset have been replaced by "NaN" which are not taken into account in the distributions.

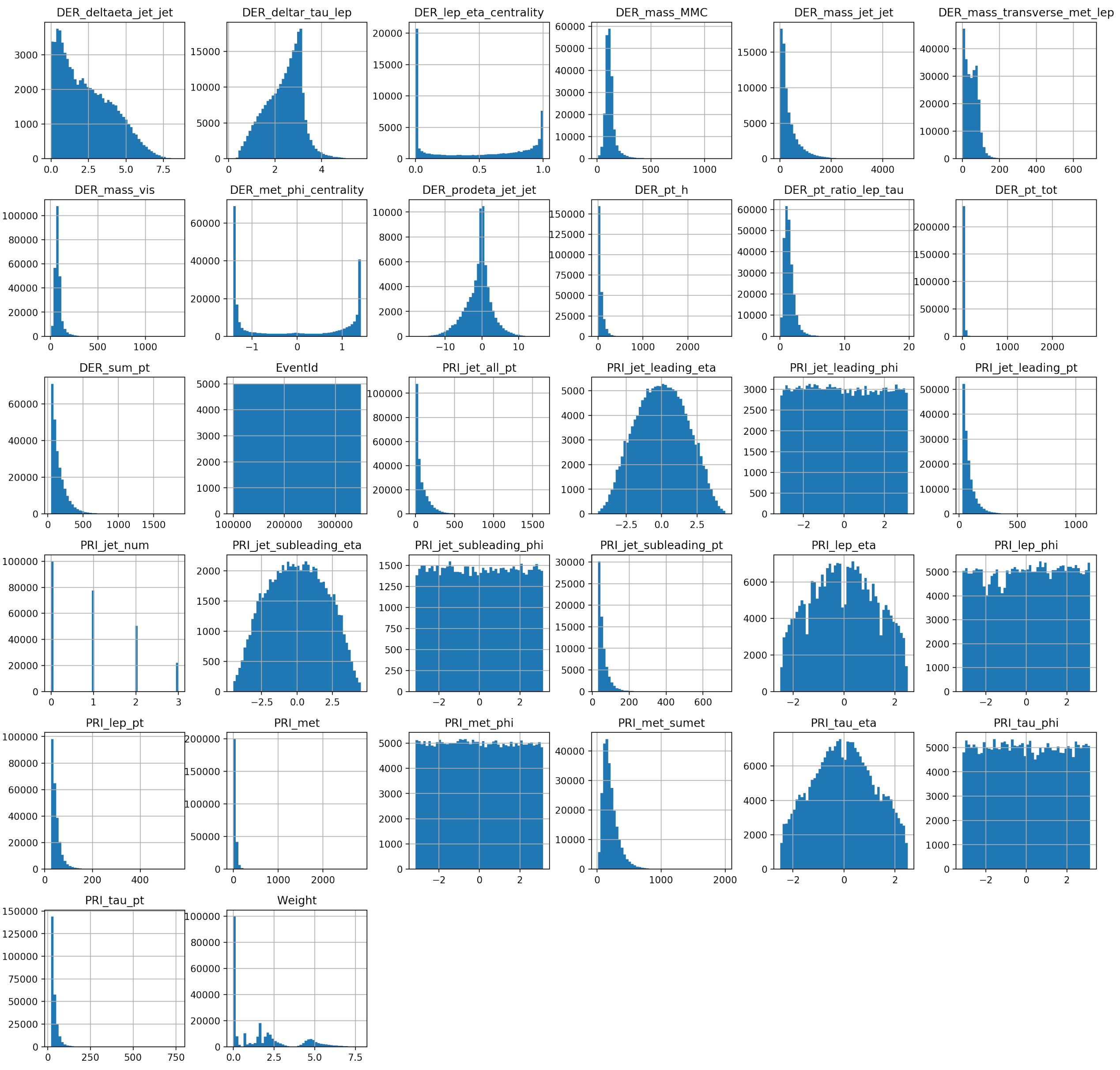

Figure 7: Univariate distribution of the features.

Several interesting observations can be made from these plots:

- All the "phi features" (PRI_jet_leading_phi, PRI_jet_subleading_phi, PRI_lep_phi, PRI_met_phi and PRI_tau_phi) have relatively uniform distributions, and are therefore not very likely to help the classifier discriminate in any way. Hence, we have decided to get rid off these features.
- Many variables have skewed distributions and for those who are positive we performed inverse log-transformation to recenter their distribution.
- Finally, for non-tree based methods, we have standard-scaled our data. It is not necessary to do it for tree-based method, but it is never harmful to the model, so this approach was followed as well for these models.

V. Principal Component Analysis and Dimensionality reduction

Our dataset is composed of 30 features, which is quite a lot. For this reason, it can be of interest to reduce the dimensions of our model, especially for particular algorithms such as the K-Neirest-Neighbors (or SVM), which could result in a high algorithmic complexity and a very long running time without dimensionality reduction. Principal Component Analysis is a statistical technique based on Singular Value Decomposition which aims at converting a set of "observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components"[1]. In this transformation, the first principal component has a higher variance, i.e. it accounts for as much of the variability in the data as possible, than the second one, etc... The plot below shows the ratio of variance explained by the 25th principal components of our model (the "PHI features" were dropped as previously explained).

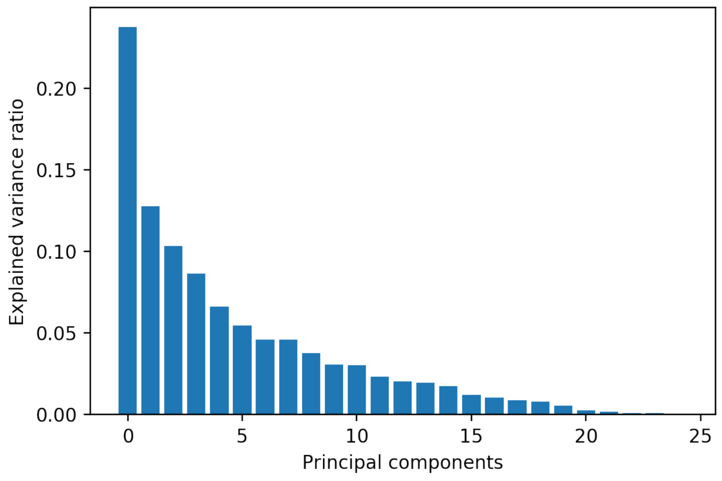

Figure 8: Variance ratio explained by the principal components of our PCA analysis.

From this plot, one understands that 72% of the variance in our dataset is explained by the first 15th principal components. For insight, we also plot the individuals (samples) in the first plane (first principal component / second principal component):

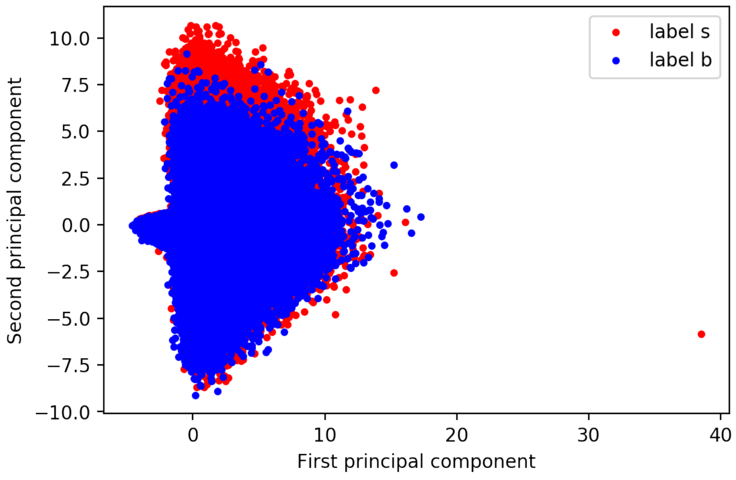

Figure 9: Individuals representation in the first plane of the principal components.

From the previous plot, it seems that for both labels ('s' for signal or 'b' for background), the distribution of our samples in the first axis-second axis plane is quite similar.

II. Models

Following the exploration of our data, several models were fitted to achieve the prediction of the type of event : 's' or 'b' from an unseen test set. Our strategy was to start with fairly simple models to observe what could be achieved in terms of performance and to subsequently broaden our approach with more complex models (tree-based models, neural networks...).

I. Validation strategy

To evaluate the performance of our various models, a clear and precise validation strategy has to be set up. Given that our training set is highly unbalanced, we have decided to perform a stratified cross-validation for each of our model to estimate several metrics : accuracy, precision, recall and ROC auc. This type of cross-validation ensures that each folds maintain the distribution of the labels within the training and test set. More over, the unbalance characteristic of our training set makes it very important to analyse a wide range of metrics to state on the performance of our model. Accuracy is typically the worse metric to trust with an imbalance data set as it is very likely to reach satisfying values, even though our model is poorly performing on the less frequent class. Precision and recall are providing more insight on the performance of our model in this case. Finally, the ROC auc metric (measuring the area under the plot of the True Positive rate versus the False Positive rate for different values of the threshold) is a good measure of the classification on the whole.

II. Logistic regression

The first model we implemented was the logistic regression. A stratified cross-validation with 5 folds was used to simultaneously tune the regularisation parameter C and estimate the performance of our model. As a quick reminder, the constant C is introduced in the loss function as follows:

$$L = \frac{1}{n}\sum_{i=1}^{n} l(y_i, \langle x_i, w\rangle + b) + \frac{1}{C} pen(w) \tag{3}$$

where pen and l respectively refer to the type of penalisation and local loss used. x_i refers to the i th example of the training set, y_i to its related label, w to the weights of the model and b to the bias term. Here, the norm $l-2$ penalisation was used. From the previous equation, one understands that the regularisation parameter C is used to prevent our model from overfitting, as, with low values for C, it will "force" our model to choose less complex weight configurations. For the logistic regression only, we have decided to investigate the influence of our two missing values strategies. Hence, we highlight the result of our cross-validation for :

- the case where -999 values were mean encoded for each feature.
- the case where our training set was subset into 4 bins based on the value of the PRI_jet_num feature.

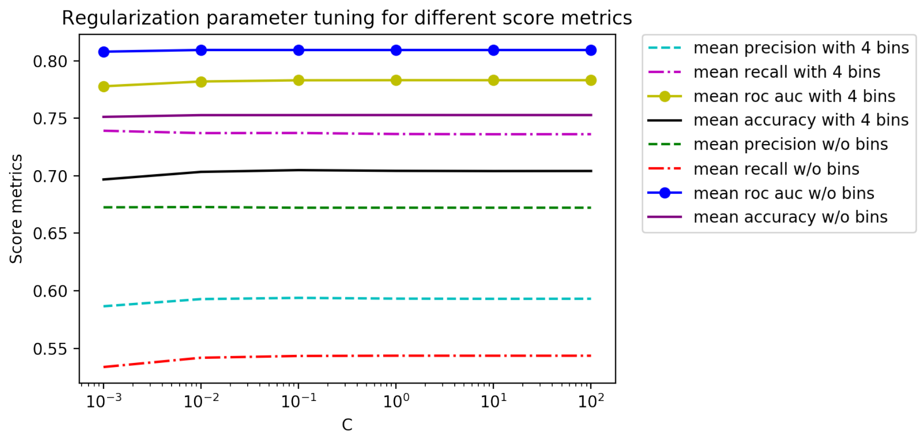

Figure 10: Regularisation parameter tuning for logistic regression with 5 fold stratified cross-validation.

We should insist on the fact that all the metrics presented in the above plot are the mean test set metrics across the five splits of the cross-validation for each regularisation parameter C. The performance of the model seems quite acceptable for a first implementation. We do obtain a quite good ROC auc and an acceptable recall and accuracy for our classifier. The precision is quite bad, even though we used the 'balanced' mode to penalize more the examples with label 1 in our loss. We are also able to come to the conclusion that the regularisation parameter has no influence on the results. This is a little bit surprising. It means that we do not have too many features, as we do not seem to be overfitting to the train set. Definetely, between the approach with recoding the NaN with mean value and the approach with 4 bins to reduce drastically the amount of missing values, the first approach seems to give sligthly better results.

As we have seen before, it looks like the regularisation parameter has very little influence on the performance of our model. Still, our model is quite good in terms of ROC auc. One of our subsequent idea was therefore to pick the threshold that provides us with this best TP and FP rate and select this parameter for our final Logistic Regression model, while choosing the "best" parameter for regularisation which was here obtained for $C \approx 0.01$. Here, the ROC curve for $C = 0.01$:

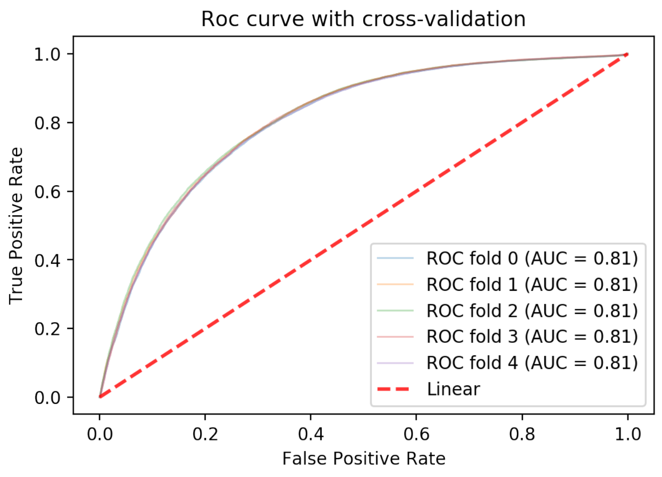

Figure 11: ROC curve for each fold of the cross-validation : logistic regression.

One can notice that our cross-validation score is very stable as the ROC curve are extremely similar from one fold to the other (five folds on the whole). This is encouraging regarding our strategy to estimate the performances of our model. Now, the influence of the threshold on our predictions is investigated:

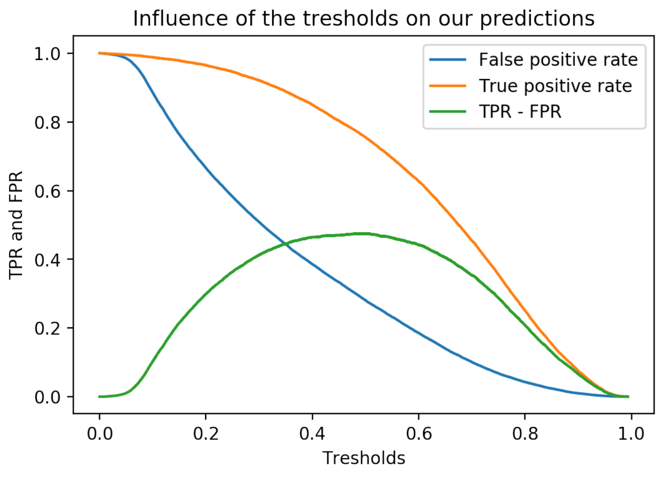

Figure 12: Influence of the threshold on the performance of the logistic regression.

The green curve refers to the value, for each threshold of the TPR (True Positive Rate) - FPR (False Positive

Rate). We actually want to maximize this quantity. Based on this, the optimal threshold was found to be 0.496. At this optimal threshold the mean test metrics are given in the following table:

Mean test set accuracy	*Mean test set precision*	*Mean test set recall*
0.74	0.73	0.76

Table 1: Score metrics at the optimal threshold for logistic regression.

Let us note that, as we have chosen the 'balanced' mode to account for an imbalanced dataset in Scikit-learn, it seems quite reassuring to find a optimal threshold around 0.5, which is the value used by default to discriminate between two classes when performing a logistic regression.

III. K-Neirest-Neighbors

As explained before, this algorithm can result in very high running times when used in high dimension. Therefore, we have decided to run a PCA and to keep the first 7th principal components to avoid long running times. These seven components account for about 50% of the variance in the data : this is quite low and would never be used in practice, but it was done so as to allow us to try the KNN algorithm on our dataset only. We then ran a cross-validation on our KNN algorithm with only 10000 samples (using a stratified split which keeps the proportion of classes within these new train and test set) to select the adequate number of neighbors (k). Again, the whole dataset was not used to avoid long simulations. The results are presented in the following:

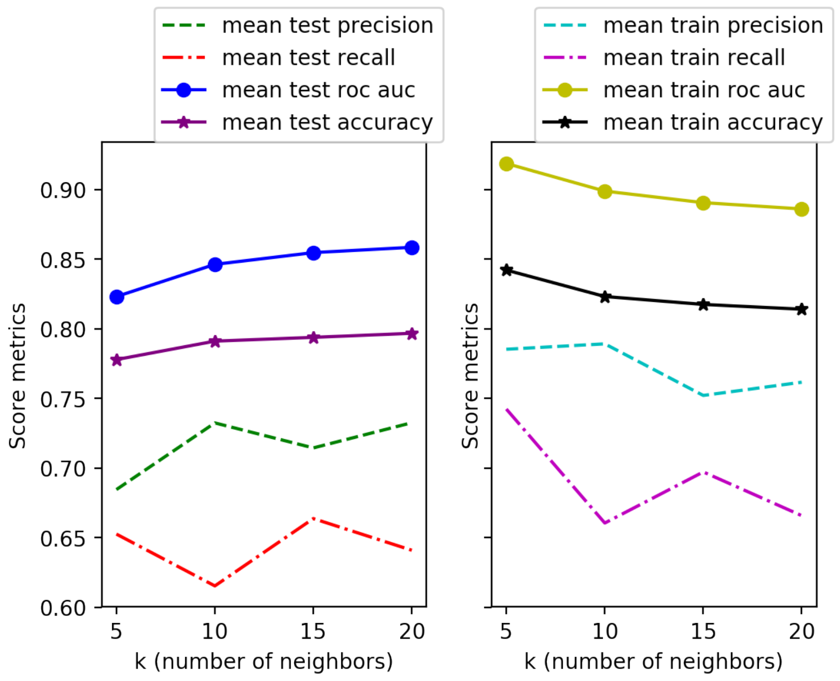

Figure 13: Knn cross-validation results with only the first seven principal components and 10000 samples.

The KNN algorithm seems to perform quite well, especially as it is here working on a small subset of our dataset (even though the proportion of labels in the train set was preserved in comparison with the total train set). Only 10000 samples were used for training here and only the first 7 principal components were considered (53% of variance only explained...) , whereas the total training set has 250000 examples. Based on the previous plot, the model is clearly not overfitting to the train set as all the metrics have similar scores on the train and test set. The main issue of the KNN method is that it is very slow to converge. I think it would perform quite well if we use the total data set, and let say 15 variables (explaining 73% of the variance), but this is very likely to be completed in at least 6-7 hours. Hence, this type of model is not realistic.

IV. Random forest

Tree-based methods are known to be very powerful and I wanted to test one of it on this dataset : random forest. This consist in building several trees with a certain degree of randomness and average over all of them to compute the output. This technique, which is referred to as "ensembling" is very valuable as it helps decrease the variance in the predictions. As we have not seen this algorithm in class, we are going to highlight its core points in more details. The algorithm is basically the following:

1. For $b = 1...B$, the number of trees in our forest
   (a) Create a new train sample by random sample on:
      i. the observations : selecting only a certain fraction of the observations (bootstrapping)
      ii. the features : selecting only samples with $card(m)$ (with $m < \sqrt(n_{tot})$, n_{tot} being the total number of features
   (b) Grow a random forest tree T_b to the bootstrapped data by recursively repeating the following steps, for each terminal node of the tree, until the minimal node size is reached (parameter of the random forest)
      i. pick the best value/split point among the m variables. This is done by choosing the variable/the split that maximizes the information gain (using expected entropy function ...)
      ii. split node into two daughter nodes
2. Output the ensemble of trees and average the predictions using a voting system.

Let us emphasize on the fact that the size of the tree should be limited by cross-validation to avoid overfitting. Many parameters of the random forest should be determined using cross-validation : this is very likely to be time-consuming. Hence, we have decided to tune the following parameters:

1. The **maximum depth of the tree**, max_depth. Indeed, setting an upper bound value for this parameter helps reducing the complexity of the trees and leads to less chances of overfitting.
2. The **minimal samples required in terminal (leaf) nodes**, min_samples_leaf. Indeed, setting a lower bound value for this parameter helps reducing the possibility to catch some noise.

The number of features to consider when looking for the best split could also be optimized. But for running time concerns, we have not done so. In Scikit-learn, the default value is $\sqrt(n_{tot})$, n_{tot} being the maximum value of features present in our dataset. Finally, the number of estimators, i.e. the number of trees in our forest should be as big as our computing material can handle it in terms of memory. To avoid too long running times as well, we have set this value to 150. As done before, we performed a 5-fold stratified cross-validation to simultaneously tune these two parameters on our random forest. The results are highlighted in the following:

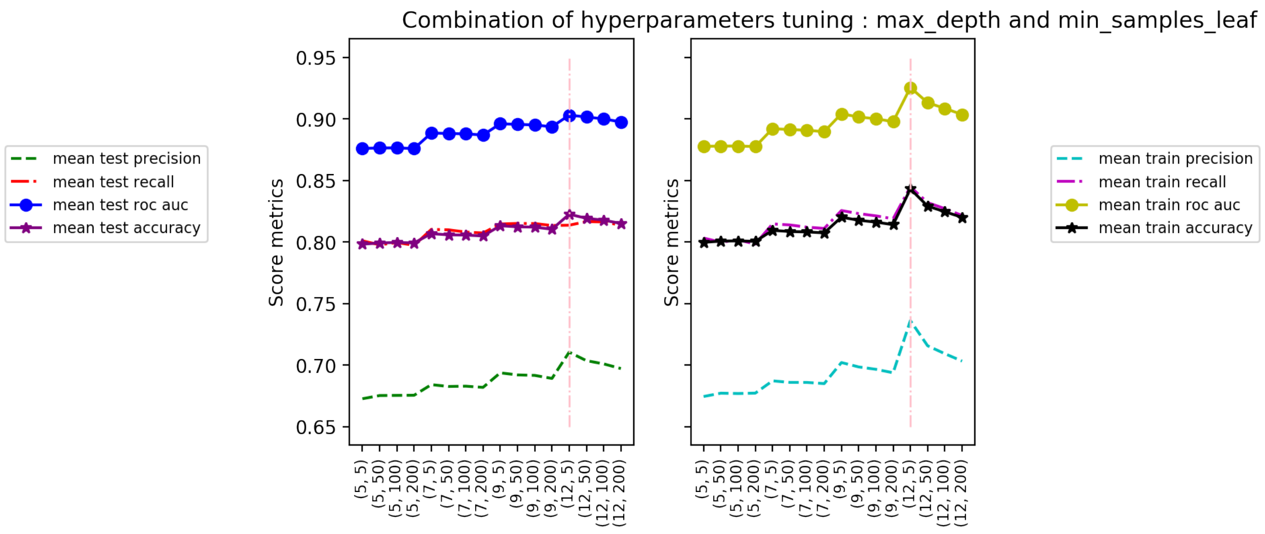

Figure 14: Random forest tuning parameters with 5-fold stratified cross-validation.

The optimal combination of the two parameters is highlighted in pink on the previous plots: $(max\_depth, min\_samples\_leaf$ $(12, 5)$. As we dit for the logistic regression, the ROC curve of the cross-validation and the influence of the threshold on the performance of our model are investigated in the following plots.

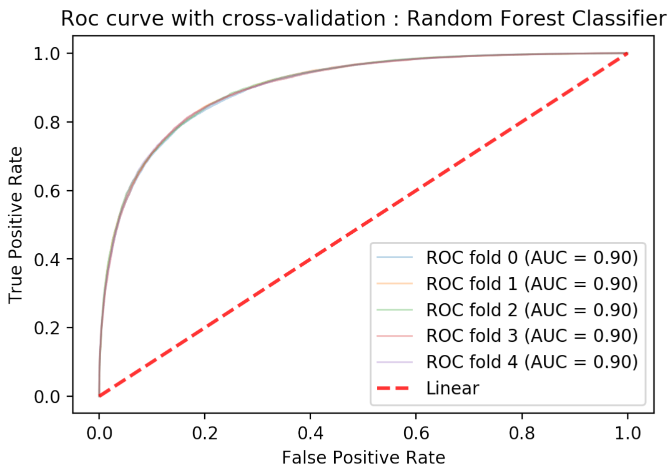

Figure 15: ROC curves for the 5-fold cross-validation for the random forest.

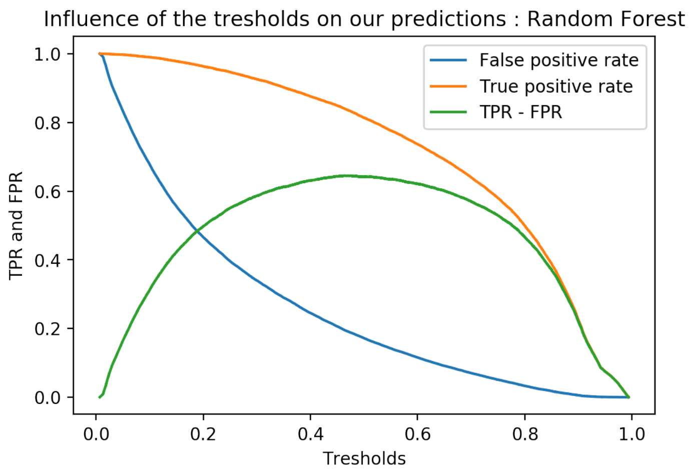

Figure 16: Influence of the threshold on the performance of the random forest.

Based on this, the optimal threshold was found to be 0.476. At this optimal threshold the mean test metrics are given in the following table:

Mean test set accuracy	*Mean test set precision*	*Mean test set recall*
0.82	0.82	0.83

Table 2: Score metrics at the optimal threshold for our tuned random forest.

The shape of the False positive rate and True positive rate with the threshold is better than for the Logistic Regression as one can notice that the False positive rate is more widely decreasing with the threshold than it used to. This model is clearly better, as it provides us with an AUC of around 0.90 which is clearly an improvement compared to the 0.81 score that was obtained with a Logistic regression.

V. Neural network

In a final attempt to this project, we fitted a neural network to our data. Neural networks are not straightforward at all to build, as there are no clear rules to follow to pick the exact number of hidden layers and their subsequent number of nodes. Hence, we decided to create our neural network with one hidden layer, containing 50 nodes. Several parameters had to be set:

- the **batch-size**, which consist in the number of training examples in one forward/backward pass. The higher the batch size, the more memory space one needs. The batch-size was set to 50.
- the number of **epochs**, which consist in one forward pass and one backward pass of all the training examples. The number of epochs was set to 20.

Regarding the activation function, we used a $tanh$ function for the input layer to the hidden layer. Usually the $ReLu$ function is used, however our choice was motivated by the fact that our dataset is filled with negative values (due to the inverse log transformation). Indeed, the $ReLu$ function provides a zero gradient for negative inputs which will not help our backpropagation algorithm, whereas the $tanh$ function has the advantage to center our data and provide a non-null gradient for negative inputs. As for the output layer, the classic sigmoid function was used, as it helps map the output to the $[0, 1]$ range which is satisfying as we wish to predict a probability. A 3-Fold Stratified cross-validation was set up to evaluate the performances of our model. The results are presented in the following.

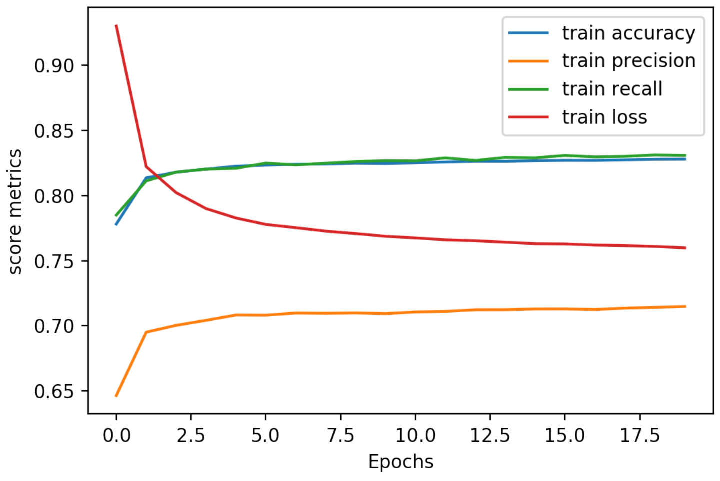

Figure 17: Training set performances of the neural network training.

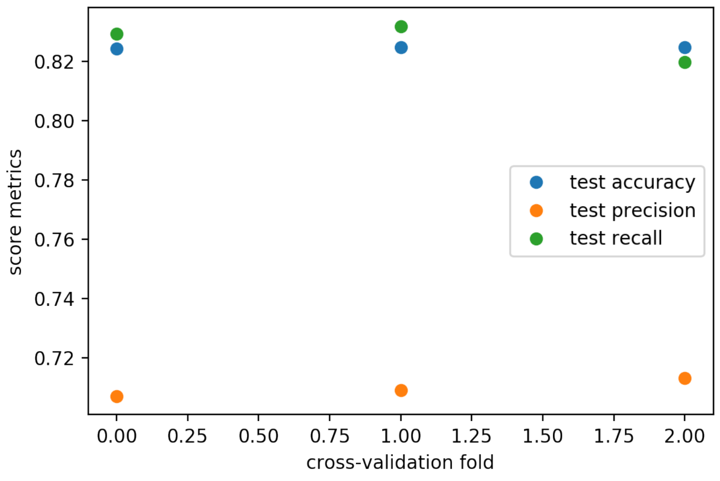

Figure 18: Validation set performances of the neural network. The results are averaged over all the epochs.

Let us note that it would have been better to monitor the metrics score of the test across the epochs. Here, we only present the mean values across all the epochs for these test set metrics.

III. Tips for understanding the notebook

Some sections of the notebook have very high running times. These are the following sections:

- K-Neirest-Neighbors
- Random Forest

For these reasons, the code related to these two sections is ***commented*** in the notebook.