SCIENCES
SORBONNE
UNIVERSITÉ

Project
« Higgs Boson Machine Learning Challenge »

Final Report

Abdelraouf KESKES
2019-2020

Exploratory Data Analysis

Introduction

In this project , we are going to tackle the challenge known as “Higgs Boson” taken up from this ***Kaggle Competition*** **.** Basically , It is a binary classification problem , we will use the simulated data with features characterizing events detected by ATLAS, we should classify examples (events) into "tau tau decay of a Higgs boson" encoded by ‘s’ (aka signal) versus "background" encoded by ‘b’ . Also , the target is represented by the column “Label” .
Since the competition deadlines are overdue . We will try to simulate this competitive environment internally . First , we will start with a preliminary exploratory analysis of the given raw data . We hopefully aim to find out some interesting features and figure out some patterns and regularities in the data which could be relevant and discriminant for the machine learning step , technically known as “ Feature engineering and features selection “ .
Afterwards , we will discuss deeply “the machine learning” step , through an entire process primarily focused on model selection , hyper-parameters tuning , different experiments and so forth .

Describing the data in numbers

- The number of Training examples : 250 000 .
- The number of Testing examples : 550 000 .
- The number of columns are 33 with :

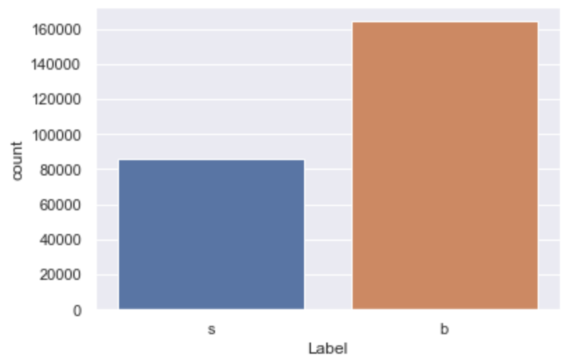
  ➔ The “EvenId” column which is an identifier ensuring the uniqueness of the examples .

  ➔ The missing data is encoded by the value -999 .

  ➔ All the 30 features are floats except “PRI_jet_num” which is an integer (actually a kind of categorical or ordinal) .

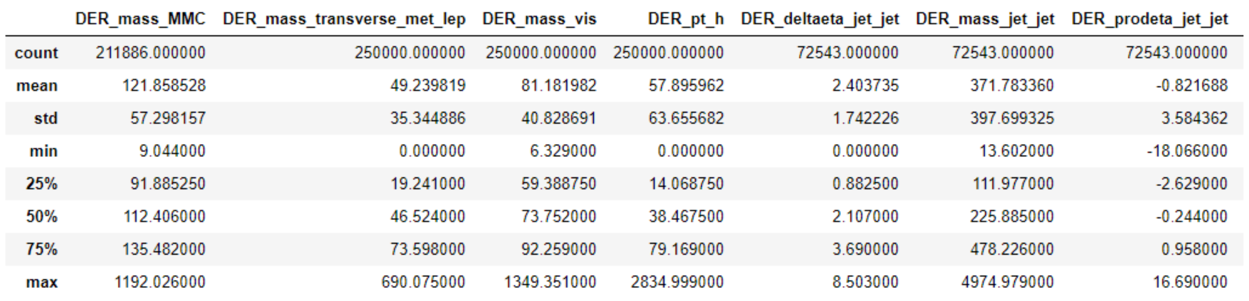
  ➔ The “Weight” column, which represents the weight of the example. It gives a rough idea of how significant a particular data point is (not all the data points are equally significant and important) .
  Note : this column is “missing” on the test set , thus we will gather it from CERN open data : http://opendata.cern.ch/record/328 **in order to estimate the metric of this challenge which is fundamentally based on the latter .**

- The “Label” column (target) consist of 2 values “s” and “b”. “s” denotes a “signal” event and “b” denotes a “background” event.
- 30 features which are basically the primitive and derived values computed by the physicists of ATLAS .
- We re dealing with a correctly unbalanced data as the following figure shows :



34 % of the class ‘s’ called signal event .
66 % of the class ‘b’ called background event .

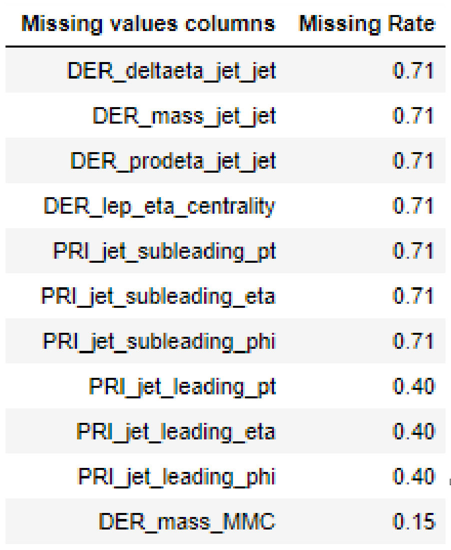
- Some basic statistics (first 7 columns to not overload the report) :



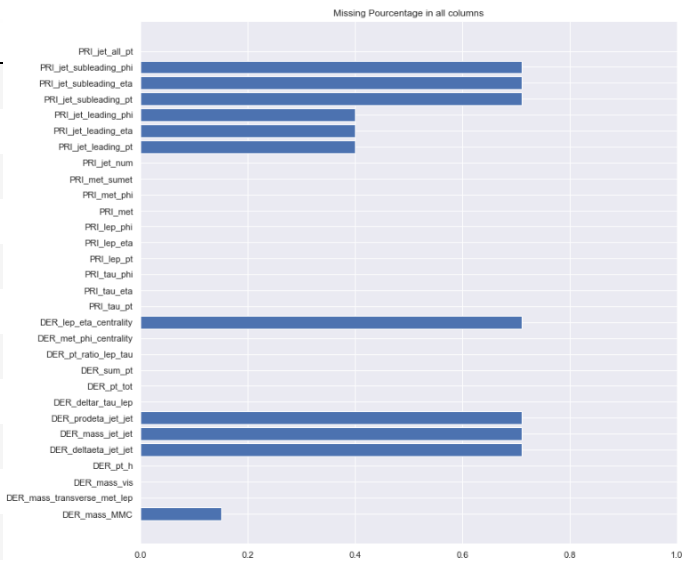
	DER_mass_MMC	DER_mass_transverse_met_lep	DER_mass_vis	DER_pt_h	DER_deltaeta_jet_jet	DER_mass_jet_jet	DER_prodeta_jet_jet
count	211886.000000	250000.000000	250000.000000	250000.000000	72543.000000	72543.000000	72543.000000
mean	121.858528	49.239819	81.181982	57.895962	2.403735	371.783360	-0.821688
std	57.298157	35.344886	40.828691	63.655682	1.742226	397.699325	3.584362
min	9.044000	0.000000	6.329000	0.000000	0.000000	13.602000	-18.066000
25%	91.885250	19.241000	59.388750	14.068750	0.882500	111.977000	-2.629000
50%	112.406000	46.524000	73.752000	38.467500	2.107000	225.885000	-0.244000
75%	135.482000	73.598000	92.259000	79.169000	3.690000	478.226000	0.958000
max	1192.026000	690.075000	1349.351000	2834.999000	8.503000	4974.979000	16.690000

- **The missing data percentage** :

We have 11 columns with missing values and 18 columns which are complete .



Missing values columns	Missing Rate
DER_deltaeta_jet_jet	0.71
DER_mass_jet_jet	0.71
DER_prodeta_jet_jet	0.71
DER_lep_eta_centrality	0.71
PRI_jet_subleading_pt	0.71
PRI_jet_subleading_eta	0.71
PRI_jet_subleading_phi	0.71
PRI_jet_leading_pt	0.40
PRI_jet_leading_eta	0.40
PRI_jet_leading_phi	0.40
DER_mass_MMC	0.15



Features Distributions & density estimation

In this section we will zoom in the features and explore their impact on the final target class within their different distributions .

Note :

In order to not overload the report with a lot of plots , we picked up few ones which could be relevant or more precisely representative of the global 30 distributions .

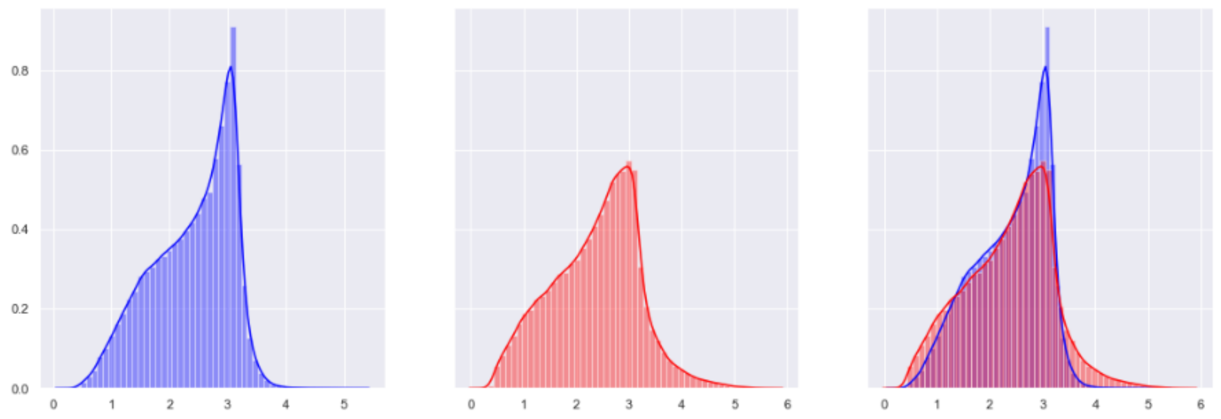
Sometimes , with some heavy-tailed distributions we consider the **log** estimated density rather than the density itself , to compresses the range of large numbers and expands the range of small numbers in order to make smoother visualizations .

With features accepting negative values we shift the distribution by | minimum | + Constant to ensure the positivity of the latter , and we plot the log density on the shifted distribution .

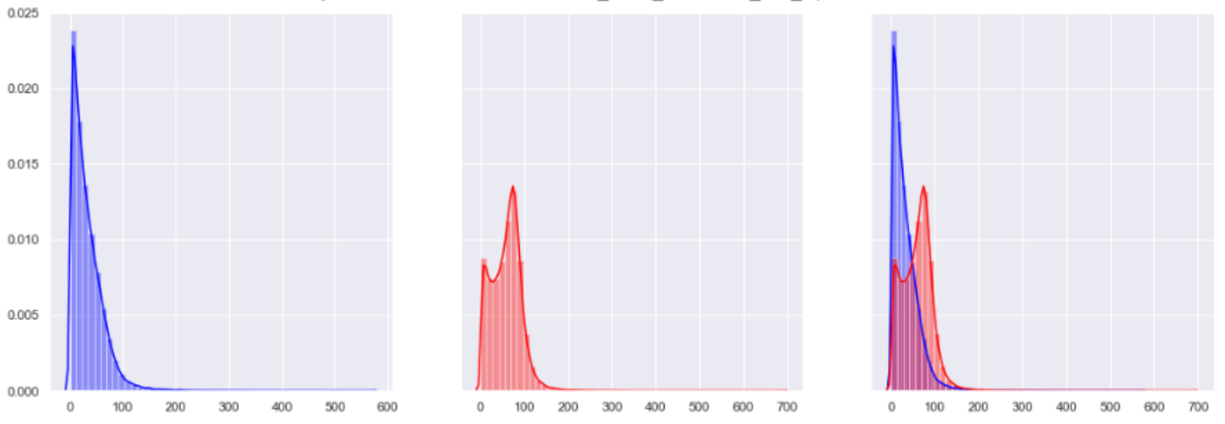
First , we will focus on the complete features (no missing values) and then we will explore the missing ones (having missing values) , where maybe we could gather a relevant information beyond this missing values .

Also , note that the Blue is for Y = 's' =1 and red for Y ='b'= 0

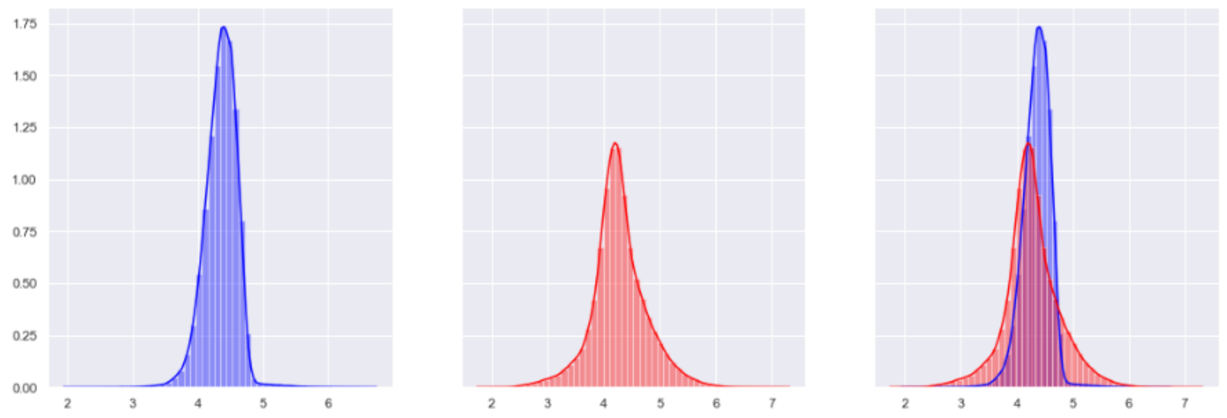
The density estimation of the variable "DER_deltar_tau_lep" on the two classes



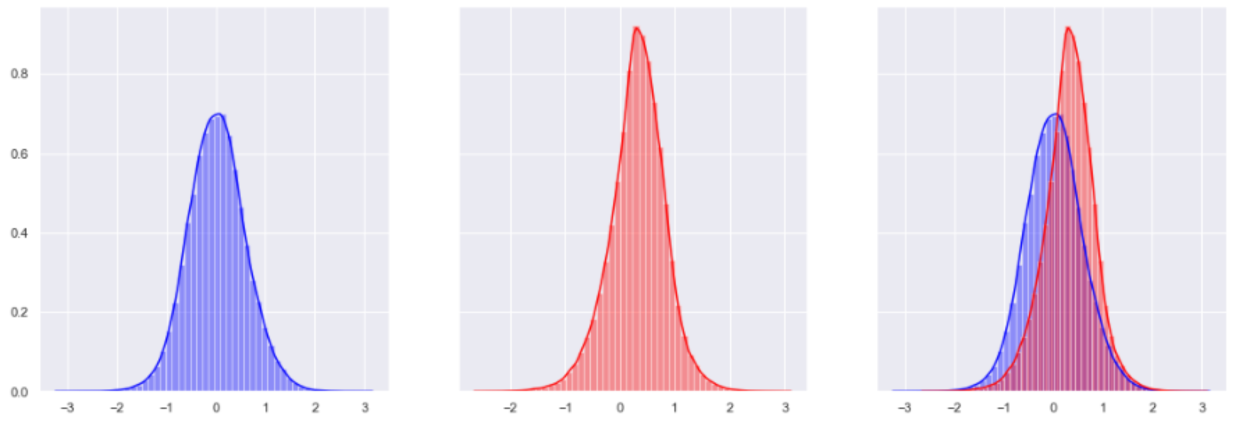
The density estimation of the variable "DER_mass_transverse_met_lep" on the two classes



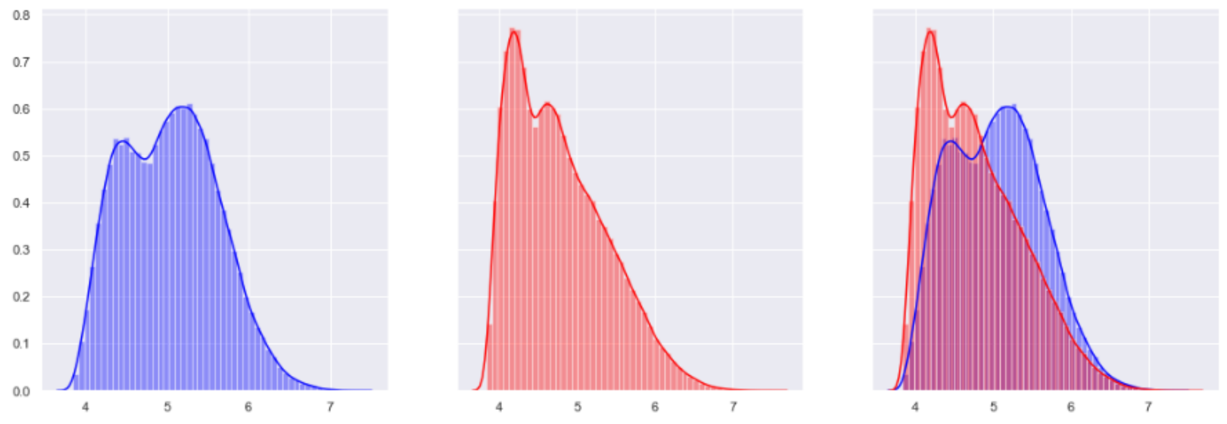
The Log density estimation of the variable "DER_mass_vis" on the two classes



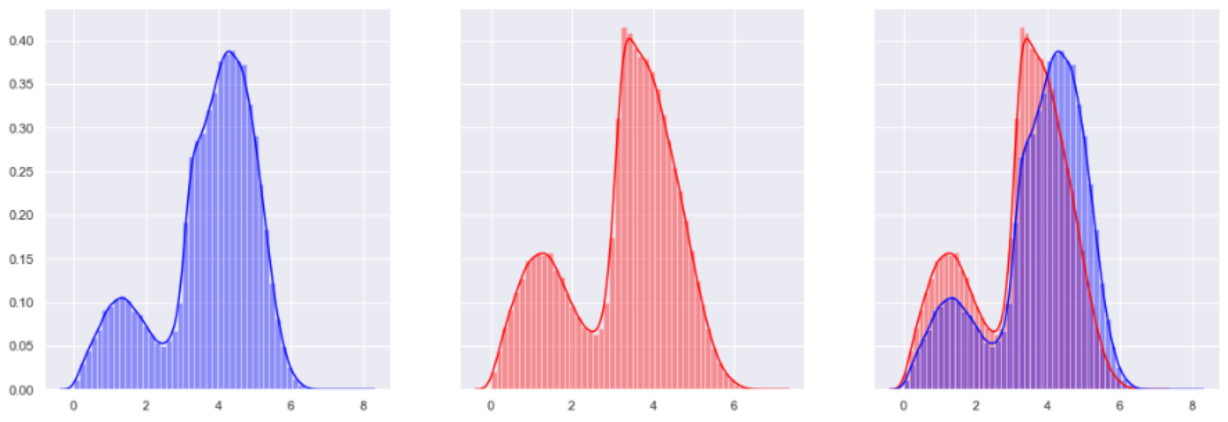
The Log density estimation of the variable "DER_pt_ratio_lep_tau" on the two classes



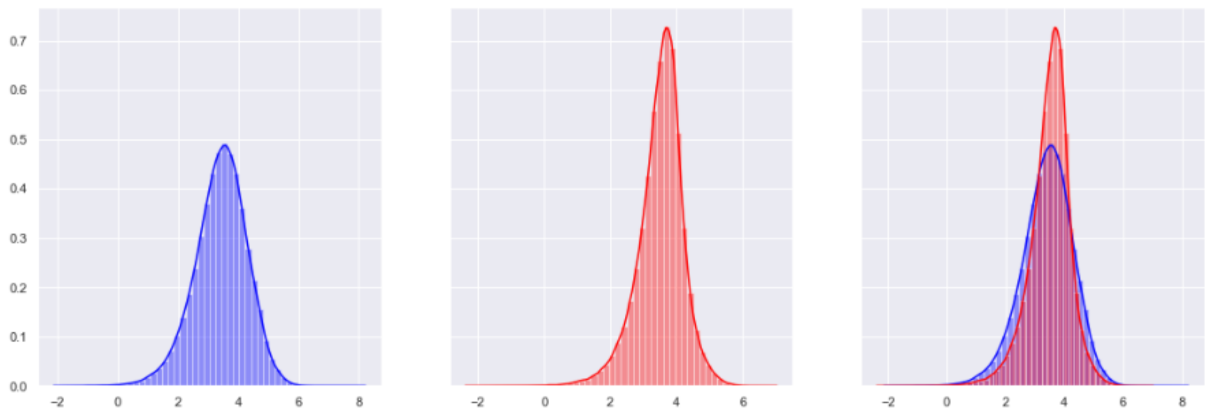
The log density estimation of the variable "DER_sum_pt" on the two classes



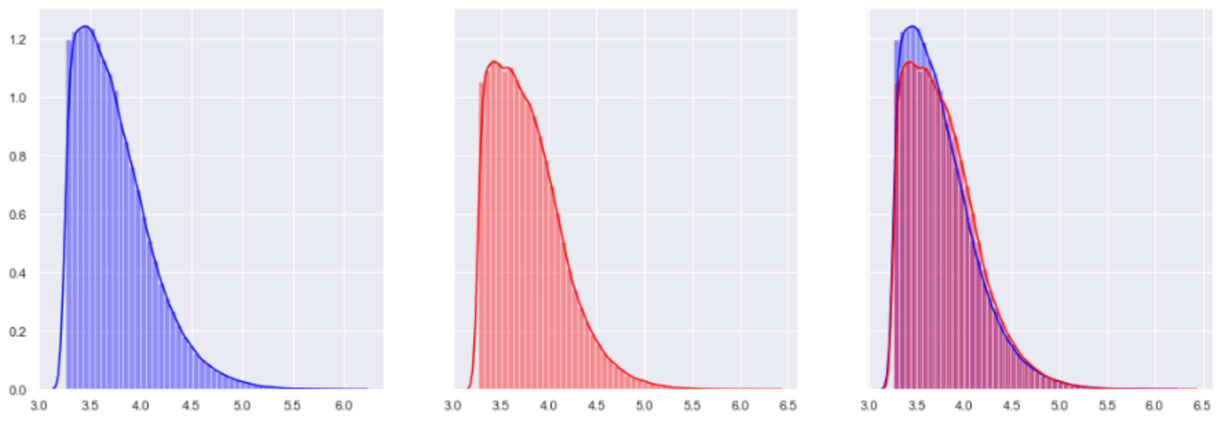
The shifted log density estimation of the variable "DER_pt_h" on the two classes



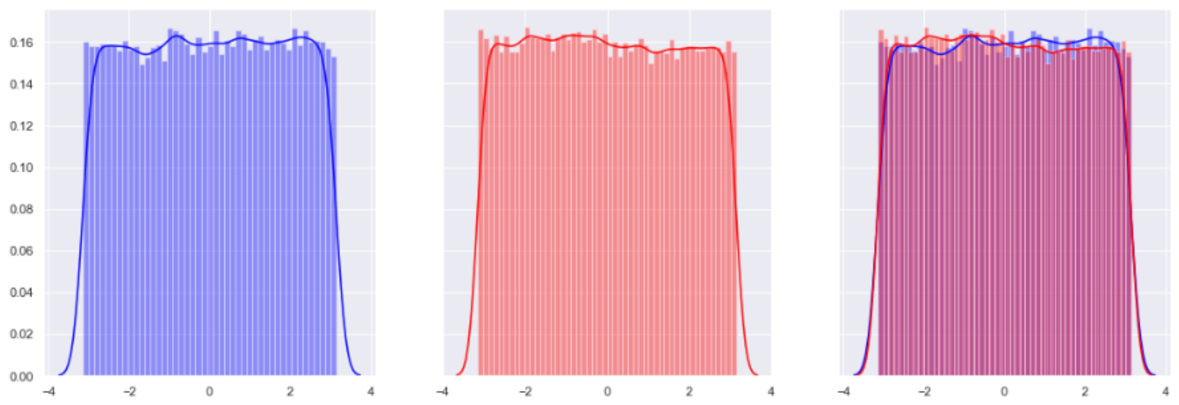
The density estimation of the variable "PRI_met" on the two classes



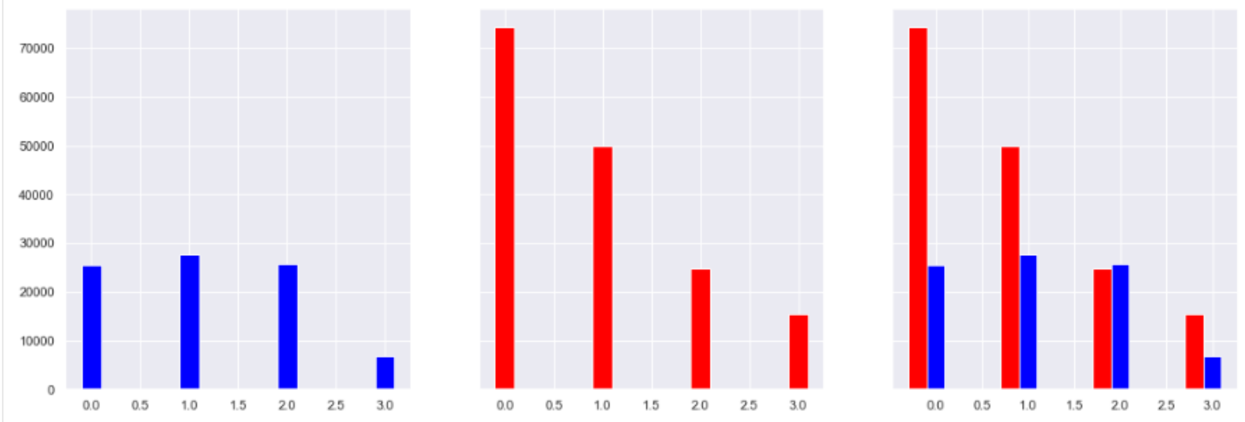
The density estimation of the variable "PRI_lep_pt" on the two classes



The density estimation of the variable "PRI_met_phi" on the two classes

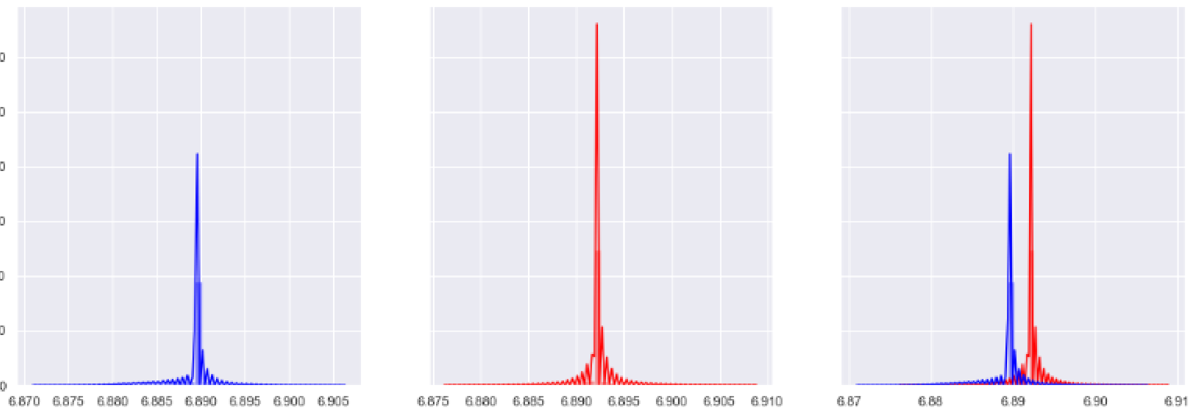


The distribution of the discrete variable "PRI_jet_num" on the two classes

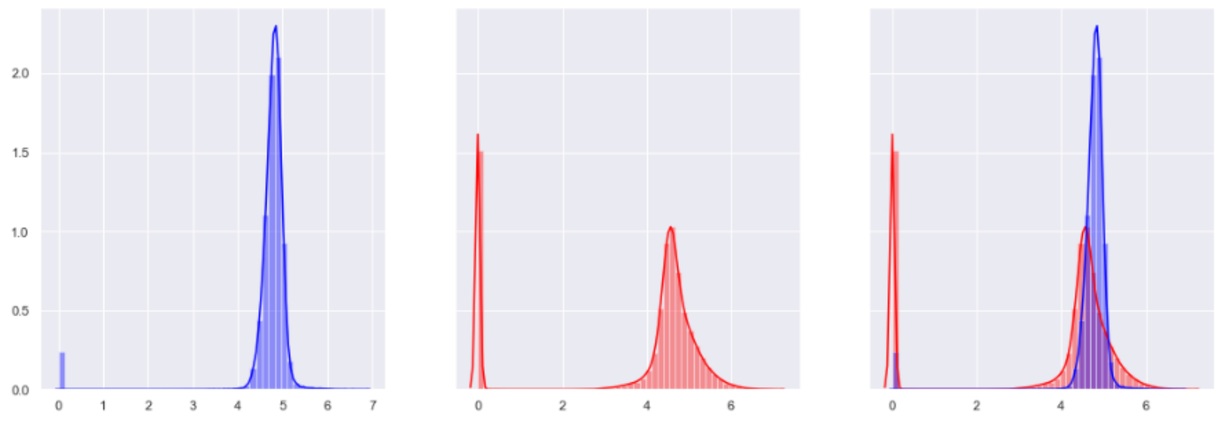


With some missing values features :

The shifted log density estimation of the variable "DER_prodeta_jet_jet" on the two classes



The shifted log density estimation of the variable "DER_mass_MMC" on the two classes

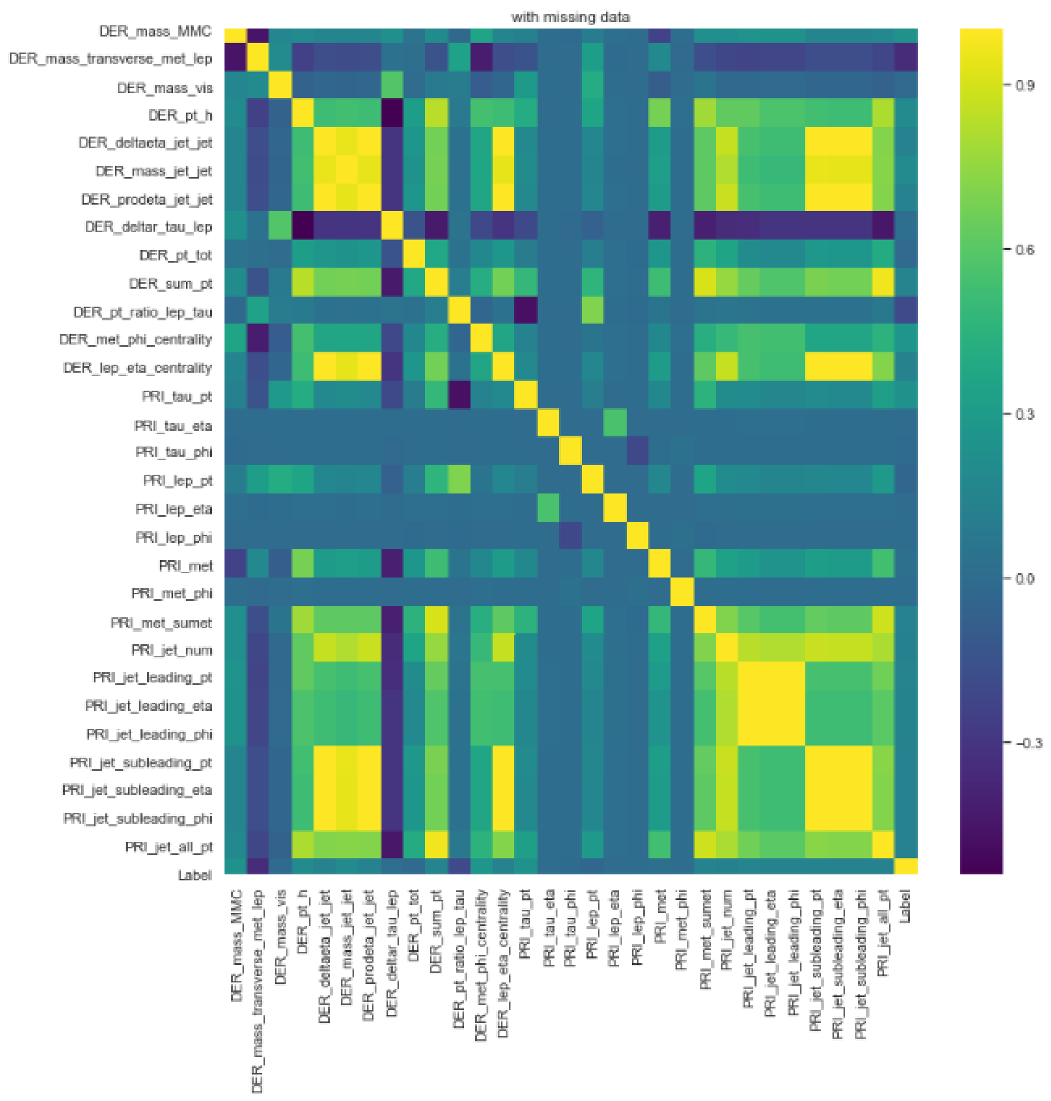


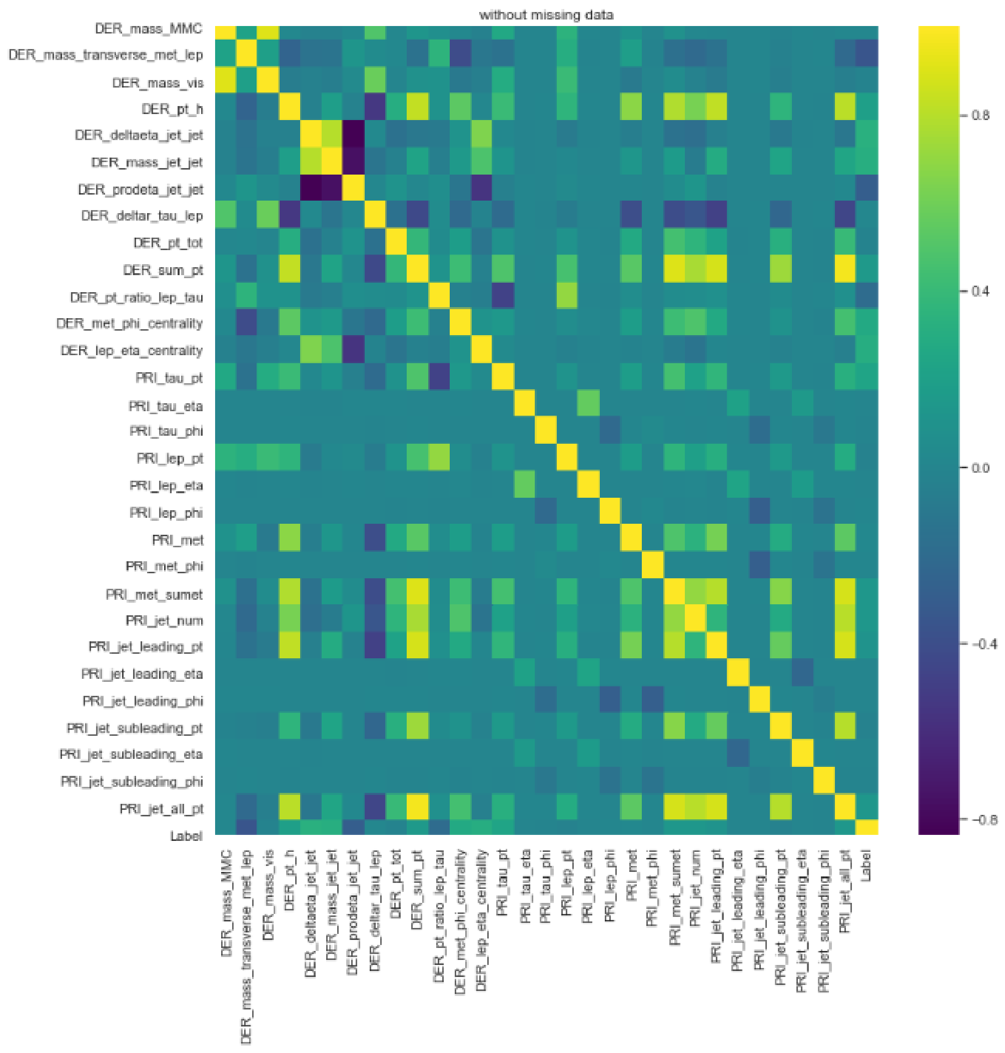
Assumption 1 :

- There are a lot of complete features which could be relevant for the machine learning and models step such as : “**DER_deltar_tau_lep**” , “**DER_mass_transverse_met_lep**” , “**DER_mass_vis**” ,” **DER_pt_h**” , “**DER_pt_ratio_lep_tau**” , “**DER_sum_pt**” , “**PRI_jet_all_pt**” , “**PRI_jet_num**” , “**PRI_lep_eta**” , “**PRI_met**” , “**PRI_met_sumet**” ,” **PRI_tau_eta**”
- Almost all the missing values columns have no relevant distributions , because of the high missing rate (40% , 71%) , except the “**DER_mass_MMC**” feature (15% missing rate) and “**DER_prodeta_jet_jet**” (71%)

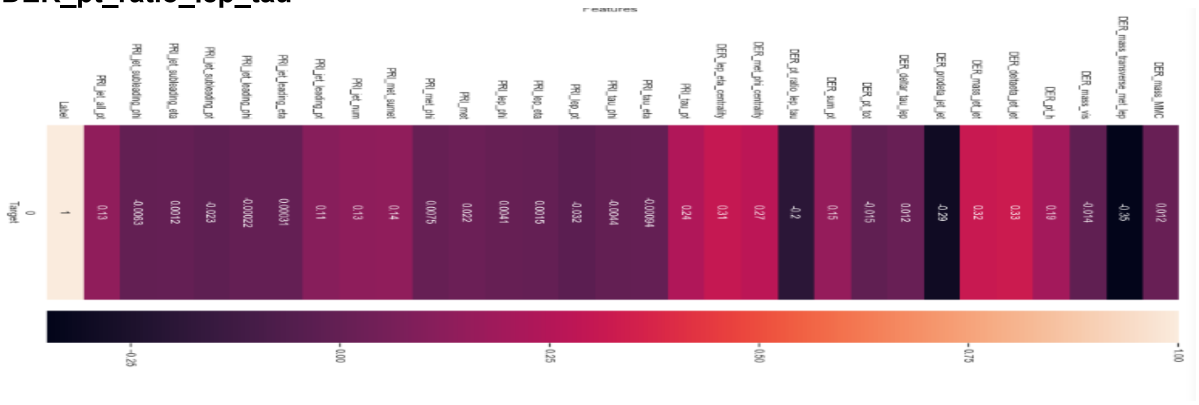
Diving into Features relationships & correlations

The following 2 figures shows the Features heatmap on the “raw” data when we take in consideration the missing value -999 **and** when we ignore it :


with missing data
DER_mass_MMC
DER_mass_transverse_met_lep
DER_mass_vis
DER_pt_h
DER_deltaeta_jet_jet
DER_mass_jet_jet
DER_prodeta_jet_jet
DER_deltar_tau_lep
DER_pt_tot
DER_sum_pt
DER_pt_ratio_lep_tau
DER_met_phi_centrality
DER_lep_eta_centrality
PRI_tau_pt
PRI_tau_eta
PRI_tau_phi
PRI_lep_pt
PRI_lep_eta
PRI_lep_phi
PRI_met
PRI_met_phi
PRI_met_sumet
PRI_jet_num
PRI_jet_leading_pt
PRI_jet_leading_eta
PRI_jet_leading_phi
PRI_jet_subleading_pt
PRI_jet_subleading_eta
PRI_jet_subleading_phi
PRI_jet_all_pt
Label
0.9
0.6
0.3
0.0
−0.3

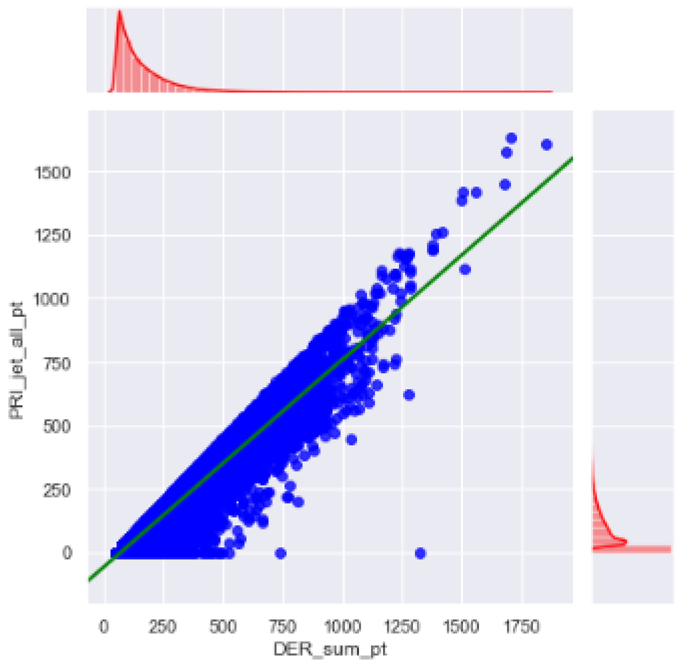


- We can intuitively notice that if we take in consideration the missing value , we will end up with “ misleading “ correlations especially between the highly missing rate features .
- If we zoom in “Label” line we can realize that the top 3 linearly impactful features are **« DER_mass_transverse_met_lep » , « DER_prodeta_jet_jet »(highly missing one) , “DER_pt_ratio_lep_tau”**

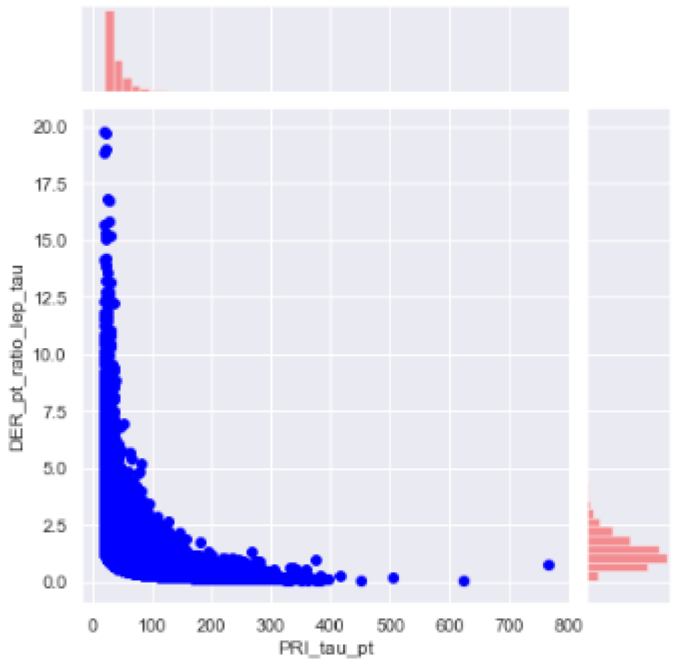


- There also some highly correlated features such as :

  - **PRI_jet_all_pt** vs **DER_sum_pt**

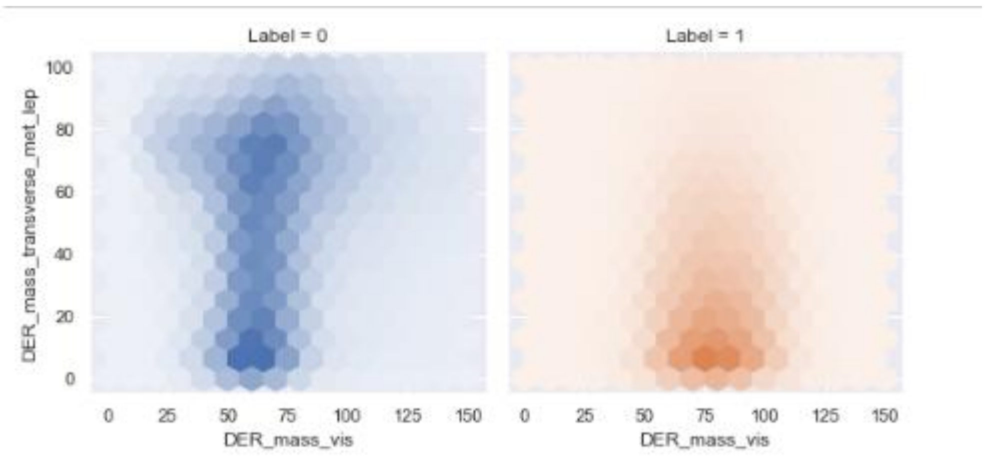


  - **DER_pt_ratio_lep_tau** vs **PRI_tau_pt**

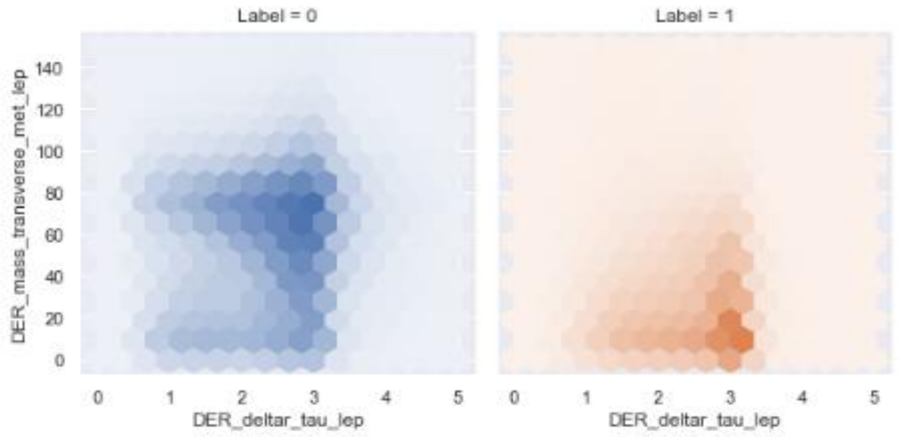


- Some dependencies are made by construction (by definition of the feature) like the "**PRI_jet _all_pt**" which is The scalar sum of the transverse momentum of all the jets of the events , or **DER_pt_ratio_lep_tau** vs **Pri_Tau_pt** which are based on transverse momentum and hadronic tau. . This dependencies are totally natural .

  - **DER_mass_vis tau** vs **DER_mass_transverse_met_lep**



  - **DER_deltar_tau_lep** vs **DER_mass_transverse_met_lep**



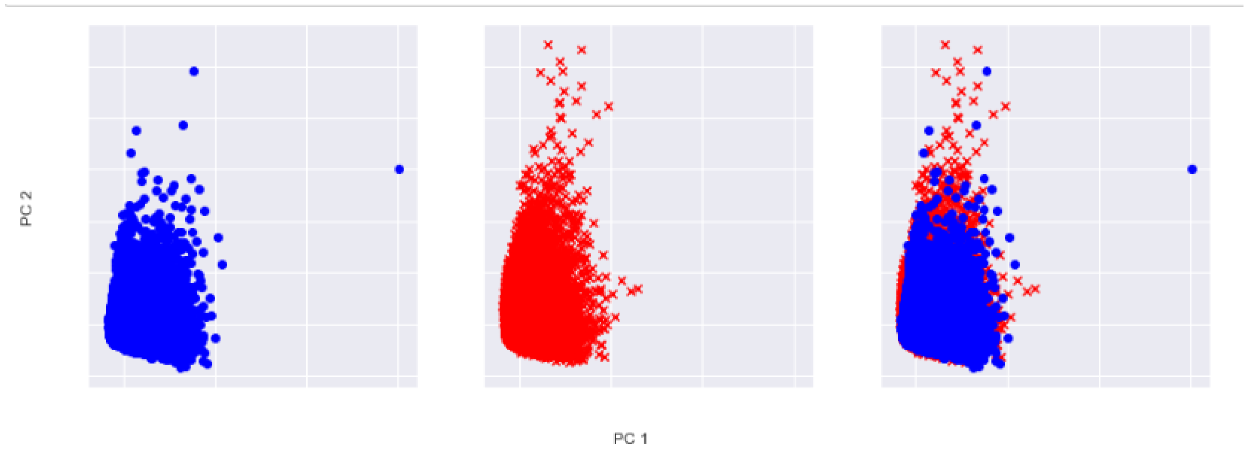
- We can see that by combining 2 features , we can recognize some dense patterns which could lead to a better discrimination between the two target classes .

- While tackling the machine learning step , we can select one variable from each two dependent variables which could drastically reduce the complexity of the model while training and therefore it will speed up the training/tuning, decrease the variance and reduce the overfitting of the model .

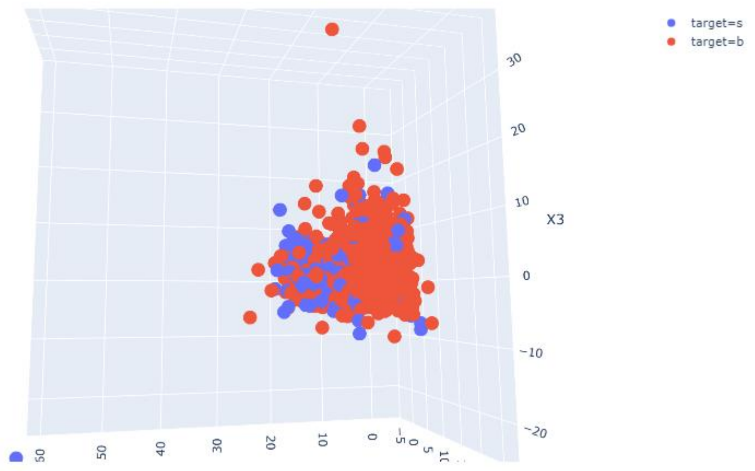
We can for example delete **pri_jet_all** and keep the basic features , or keep only the **pri_jet_all** and delete the “premise” features , or even maybe keep both to decrease the bias and increase the variance .

PCA

Applying a PCA on the complete features (19 features with no missing value) results on the following figure :



3D

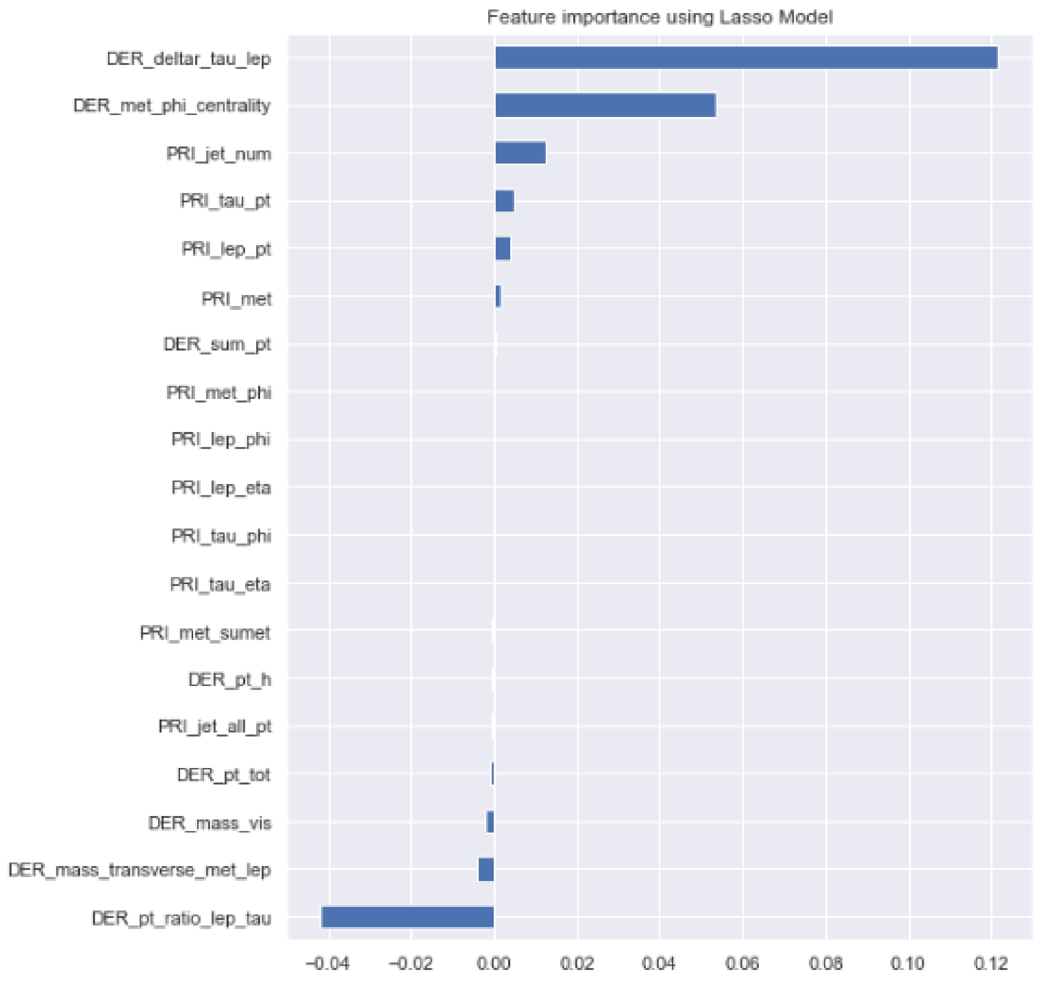


By Definition a PCA is a linear projection of the data. This method can be powerful, but often miss important non-linear structure in the data . Thus , our case is a non-linear subspace called “manifolds” which explains the bad reduction using a PCA .

Conclusion of the first part

After this manual Feature engineering , it turns out that we re dealing with a hard problem with real dense and compact data , Actually , we could choose the Top-5 independent impactful features from the complete features and the two relevant missing features which are : **DER_mass_MMC , DER_prodeta_jet_jet .**

We also have to include the machine models and try some “wrappers method“ to get the best relevant features according to the latter’s .
 We could also try some “ embedded methods “ like LASSO and others to retrieve the most relevant features . The following figure is a “very quick” method showing LASSO features selection on complete features .



The Experimental Protocol

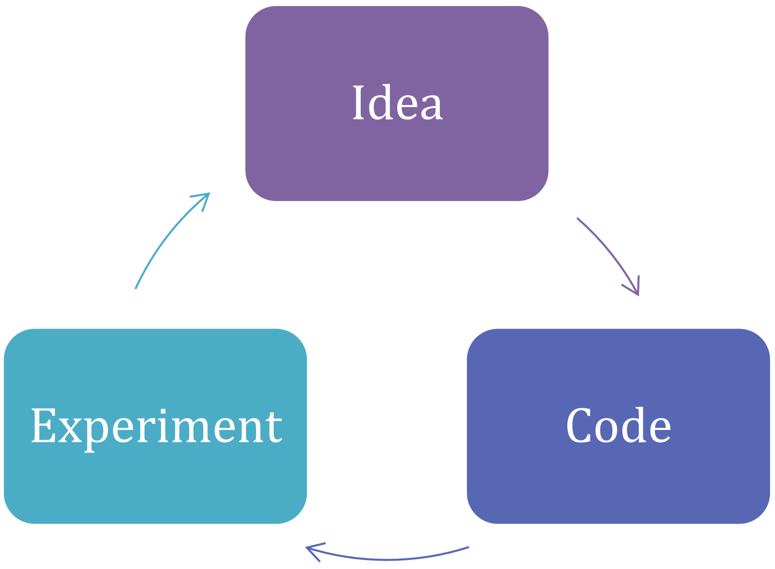
In this second part , we will train different models from different perspectives and then fine-tune & evaluate them . We will start with simple classical machine learning models like KNN , Logistic regression and so on . Afterwards , we will experience some Bagging/Boosting Models and see what are the outcomes .

Approach

To make a challenging and correct baseline , we will take the raw data without any features selection or preprocessing and feed it to different models in order to figure out which kind of data we are dealing with and more accurately how models conceive this raw data (Models interpretability) .

The chosen evaluation metric will be a familiar one called accuracy as a warming up metric , before diving into Precision/Recall/F-measure and obviously the proposed metric called AMS and test it on different contexts

The following figure depicts the global picture of our methodology :



Baseline & classical models

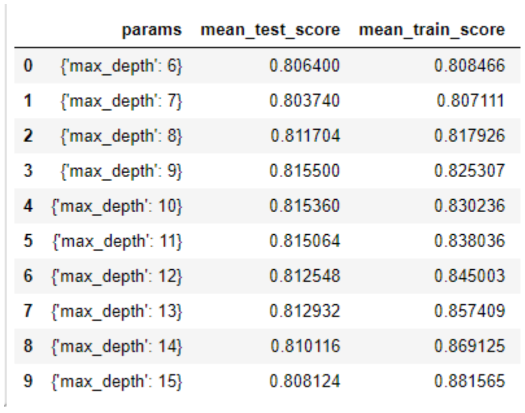
Due to the huge number of available training data (250K) , and as a rule of thumb we split the training data into 90 % : training set and 10% : Validation set (25 000 examples) .
The following table illustrates some baselines without any feature selection / engineering .

Note : We did some tuning to reduce overfitting with a cross-validated grid search selection .

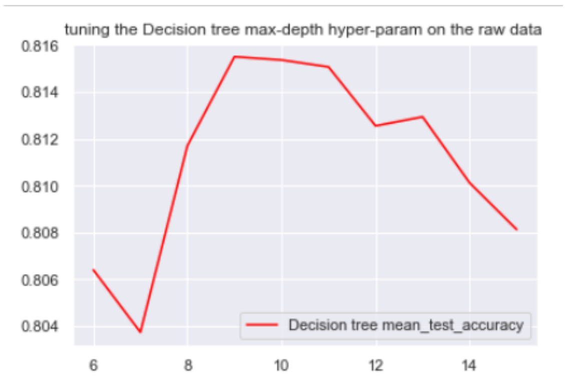
	KNN	**Logistic Regression**	**SVM**	**Decision Tree**	**Random Forests**	**Adaboost**	**XGBoost**	**LightGBM**
Train	0.84	0.74	Very long time (crash)	0.82	0.84	0.81	0.83	0.85
Validation	0.80	0.74		0.81	0.83	0.81	0.83	0.84

As expected , linear models like Logistic regression and SVM are not well suited for this problem because the data as we've seen before is not linearly separable .Even though we could use kernels with SVMs , but it was drastically slow , therefore we skipped them all .
We could also confirm this assumption by the fact that non-linear (Decision-Tree oriented models in our case) outperforms the others .

As we mentioned before , we did some grid search cross-validation (cv=5) hyper-parameters tuning to reduce overfitting in order to tease out the best parameters . you could check the figures below for more details :

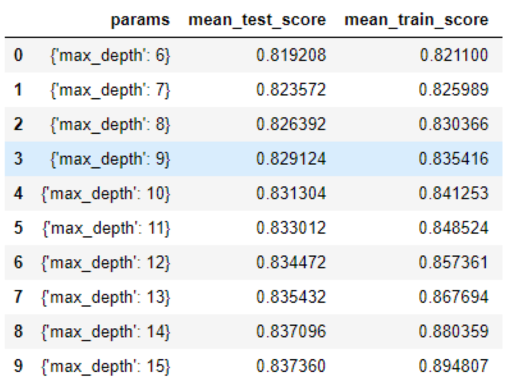


	params	mean_test_score	mean_train_score
0	{'max_depth': 6}	0.806400	0.808466
1	{'max_depth': 7}	0.803740	0.807111
2	{'max_depth': 8}	0.811704	0.817926
3	{'max_depth': 9}	0.815500	0.825307
4	{'max_depth': 10}	0.815360	0.830236
5	{'max_depth': 11}	0.815064	0.838036
6	{'max_depth': 12}	0.812548	0.845003
7	{'max_depth': 13}	0.812932	0.857409
8	{'max_depth': 14}	0.810116	0.869125
9	{'max_depth': 15}	0.808124	0.881565

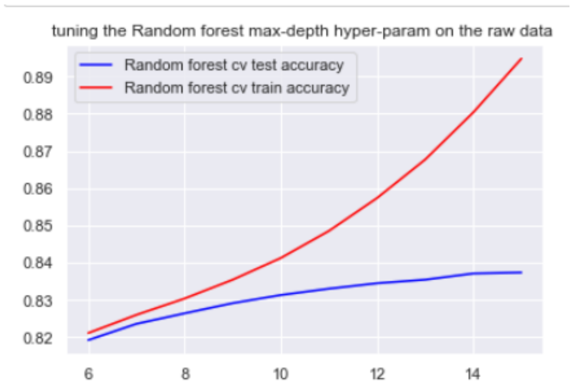


Tuning the hyper-parameter " max-depth " for a Decision Tree Model

We can see the limitations of a single decision tree which is prone to overfitting quickly when we aim to make it strong , we ve selected max-depth = 9 as the best fitting .



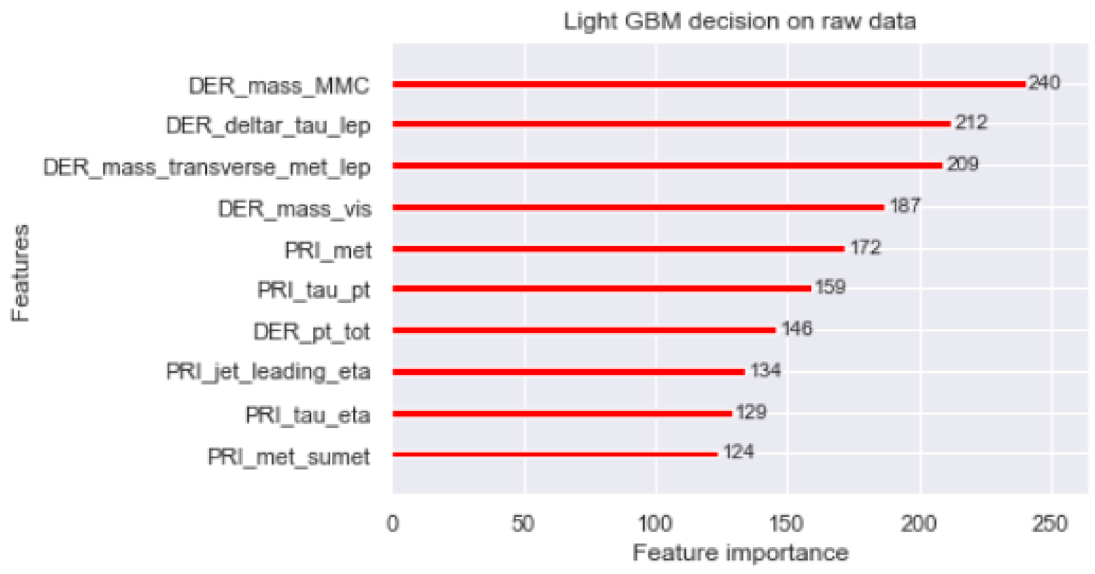
	params	mean_test_score	mean_train_score
0	{'max_depth': 6}	0.819208	0.821100
1	{'max_depth': 7}	0.823572	0.825989
2	{'max_depth': 8}	0.826392	0.830366
3	{'max_depth': 9}	0.829124	0.835416
4	{'max_depth': 10}	0.831304	0.841253
5	{'max_depth': 11}	0.833012	0.848524
6	{'max_depth': 12}	0.834472	0.857361
7	{'max_depth': 13}	0.835432	0.867694
8	{'max_depth': 14}	0.837096	0.880359
9	{'max_depth': 15}	0.837360	0.894807



Tuning the hyper-parameter “ max-depth “ for a Random Forest

We can realize that the gap between Train / Valid sets accuracy is drastically diverging , which illustrate the high sensitivity of Bagging Models to the complexity of weak classifiers .

Before switching to Gradient Boosting models , We just want to point out **the incredible computational efficiency** of their implementations such as XGBoost and Light GBM libraries .
Additionally , “the icing of the cake “ was the fact that they re adaptable and flexible with missing data (encoded with a negative number) .
After training a light GBM on our training set and zooming in its TOP 10 features descrimination importance , it turns out that DER_mass_MMC (which has 15% missing values) is the best one.

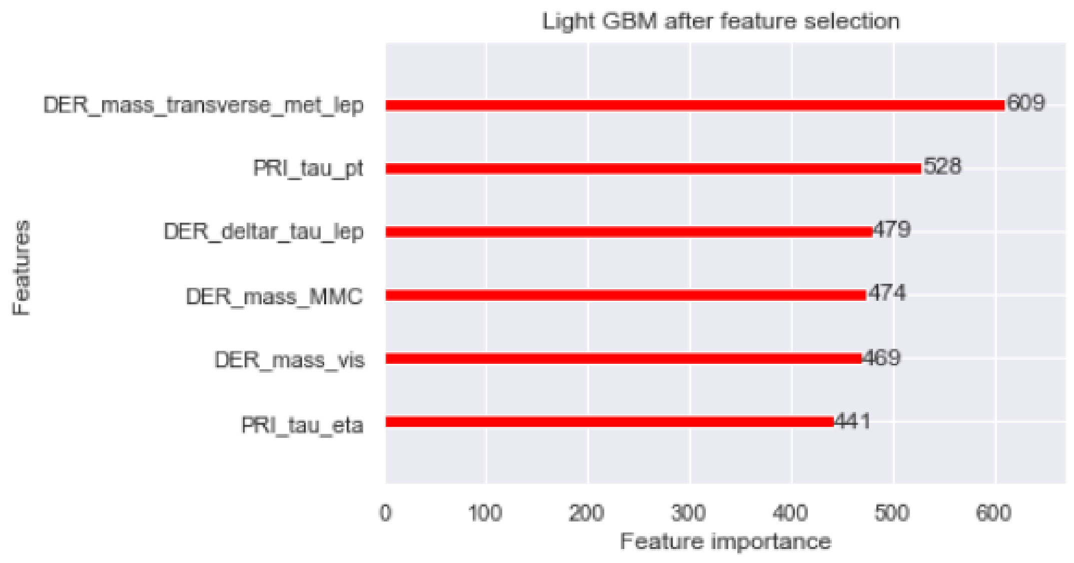


After doing some features selection:
["DER_mass_MMC","DER_deltar_tau_lep","DER_mass_transverse_met_lep","DER_mass_vis" , "PRI_tau_pt"] and applying it on LightGBM (and also on Decision Trees) , the performances have decreased to :

- Train From **0.85** to **0.82**
- Test : From **0.84** to **0.82**

These results fit exactly with the nature of Decision-Tree oriented Models , that do an inner native features selection while building its model .Thus , most of time , we will focus on features engineering (scaling , normalizing , …) rather than a time-consuming task such as features selection which could be used on tuning hyper-parameters when we deal with a DT oriented models .

What s remarkable too is the fact that after doing this features selection , the features importance order has changed , so for example The first one “DER_mass_MMC” has leaped down to the 4 th place , “DER_deltar_tau_lep” loosed one place , “PRI_tau_pt” has jumped up to the second place , and so forth .



These findings approve some speculations that we ve been doing since the first part around features relevance . There are some features which are weak alone but make a strong descrimination combined with another (other) feature(s) and vice-versa .

Actually , we will focus on the most relevant model according to these baselines “LightGBM” which seems to be slightly distinguished for this task .Therefore , we’ve **greedily** chosen our model .We will tackle the hardest and the most time consuming part (aka tuning hyper parameters) trying different configurations which hopefully will increase our metrics .

As cited above , the accuracy was not very informative , it was almost misleading (biased by the true positives of catching the ‘backgrounds‘) .Therefore , in this week , we will use first of all Precision/Recall/F1-measure based on the class ‘signal’ encoded by ‘1’ ., and then we will quickly move on and put our whole effort on the challenge metric known as ‘AMS’ .
We remind that the test data used here is the data marked as ‘ b‘ (b for leaderboard) will occur only at the end to get an approximation of our model with **unseen data** ,We will also use it to compare ourselves with those who are displayed on Kaggle leaderboard .

LightGBM

As mentioned on its documentation . LightGBM is a **gradient boosting framework** that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages: Faster training speed and higher efficiency (20 x than others frameworks like XGBoost …) . Lower memory usage. Better accuracy. Support of parallel and GPU learning. Capable of handling large-scale data.
Briefly , we will try to give you the intuition behind this framework , inspired from the original paper . For more theoretical details I highly recommend the official paper of this magic tool : https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

As we all know , the main problem of "Decision Trees Oriented Models " is the fact that the splitting decision is a very time-consuming and computationally completely inefficient and not scalable . The major problem comes from the necessity of scanning all data instances for each feature to estimate the gain . Thus , many frameworks have tried to tackle this problem in order to break the complexity and figure out an efficient way to select a representative relevant fraction of data , which will approximatively yields to the best estimation of the real "Gain" while pslitting . These techniques of fractioning are inspired from Random forests which actually performs a kind of dropouts (Neural nets) , so it's also a kind of regularization to avoid overfitting .
Most of relative work which has been done to tackle this kind of problem was based on a "Weight column" for each sample to boost the training , and what makes LightGBM outstanding is the fact that it's designed for the purpose of not needing weights column by definition , obviously it will leads to a drastically better performance if this column is available.

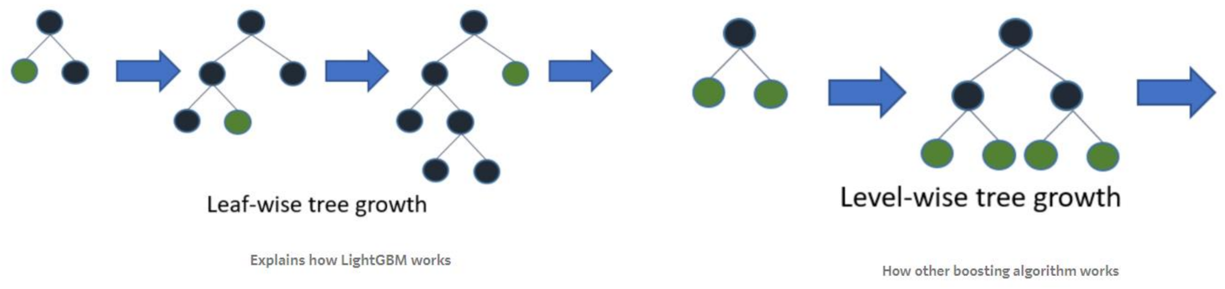
Therefore , to solve this problem LightGBM researchers propose two sampling techniques while splitting .The first one is called GOSS (stands for Gradient-based One-Side Sampling) and the second one is called EFB (stands for Exclusive Feature Bundling) .

- **GOSS** : it's a horizontal approach (i.e., it's instance based , it fractions data instances and not features) .
  They noticed that data instances plays different roles on gain's computation , thus the under-trained (with high gradient) instances will be very impactful and conversely those with lower gradients have small impact on the gain computation .Thus they decided to take all the impactful data instances and sample randomly upon the less impactful ones , and it gives a very accurate estimation of the gain , especially when the value of information gain has a large range .
- **EFB** : it s a vertical approach (i.e., it's features based , it fractions features and not instances) .
  They bundled mutually exclusive features (i.e., they rarely take nonzero values simultaneously) through a **greedy** algorithm yielding to very accurate estimations , because the optimal bundling (combinatory problems) is NP-hard .

To this end, they designed an efficient algorithm by reducing the optimal bundling problem to a graph coloring problem (by taking features as vertices and adding edges for every two features if they are not mutually exclusive), and solving it by a greedy algorithm with a constant approximation ratio .

Consequently , a hybrid fractioning method combining both GOSS & EFB is called **LightGBM**.

LightbGBM also grows trees vertically (leaf-wise) comparing to the other boosting algorithms which grows trees horizontally (level-wise) as the following comparative figure illustrates :


Leaf-wise tree growth
Explains how LightGBM works
Level-wise tree growth
How other boosting algorithm works

On the Left is LightGBM growth methodology and on the right other algorithms .

Experimental process

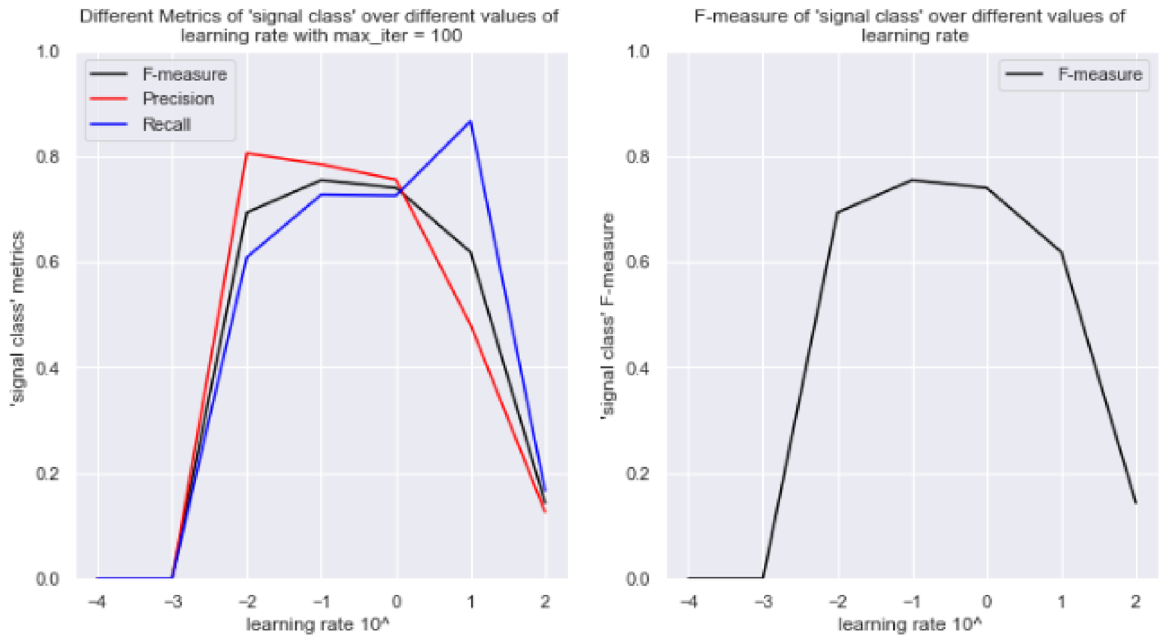
We just want to mention that we have tried different features selections and features engineering , such as :

- Selecting subsets of features
- Normalizing the data
- Replacing the missing values with Most frequent , Mean .
- Log transformations
- And many others …

These approaches was always falling down and did not lead to an improvement , most of times we've achieved lower results than with raw data , and it turns out that our lightGBM prefers the raw data as they are , and by definition it does the appropriate feature selection regarding it s own Loss function .

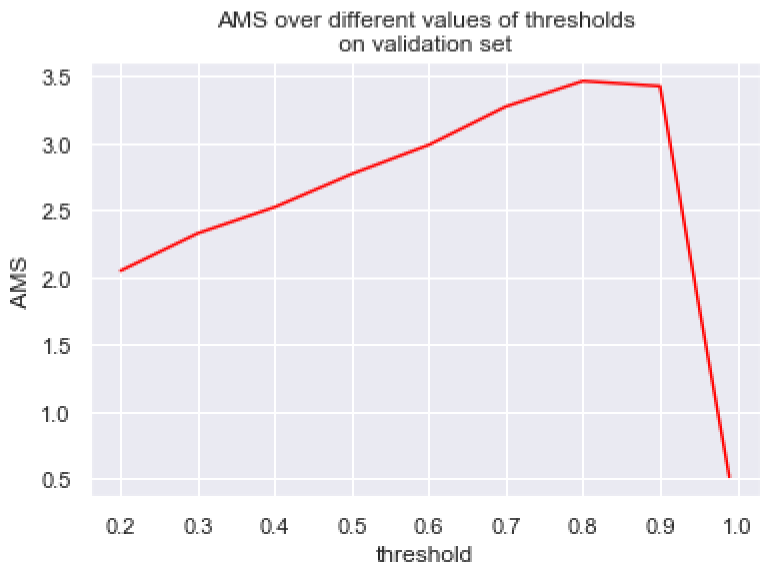
We could not do a tremendous grid search with millions of possibilities even over a sample of parameters due to deadlines date and also feasability . As a result , we will follow a greedy straightforward search , which means hyper-tuning one parameter after another and keep in mind to recycle the whole pipeline changing one or 2 parameters , it s a kind of random greedy manual grid search based on intuition and top potentially relevant hyper parameters , so for example rather than grid searching on all parameters over all values we could select a small range TOP-3 learning rates values , TOP-3 thresholds and so on , which were selected independently and do some combination tests to see !

We started with a classical hyper-parameter known as “the learning rate” , the following figure illustrates the range of values used to hyper-tune the latter .

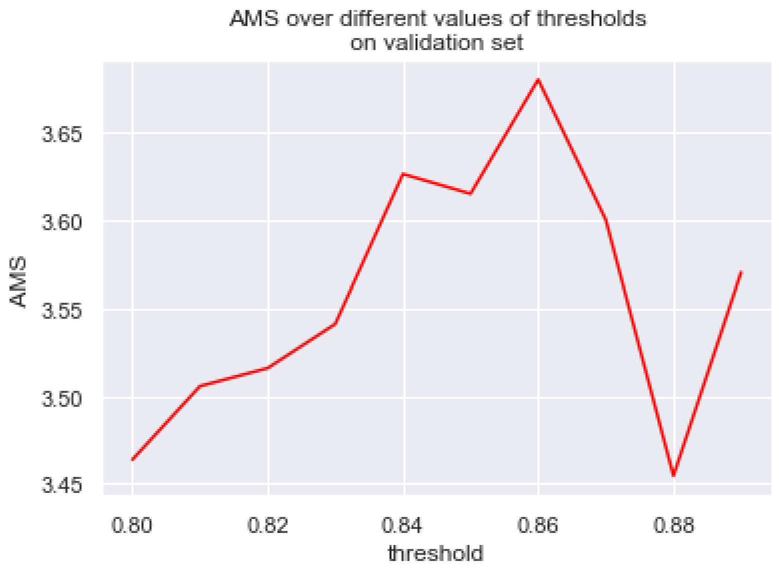


As we can see according to the left plot which combines the 3 metrics the most appropriate value between precision / recall and F1 is **10^0=1.** However , as the F1-measure is a harmonic mean between precision and recall by definition , it could be better to take **0.1** (the best value according to F1) or at least to give it a try .
After doing some experiments over the whole next pipeline , it turns out that **learning rate = 0.1** was widely better than 1 .

Then , We decided to explore the threshold parameter that cuts off the 2 classes , usually the default value is 0.5 , but after trying a range of values between 0.2 and 0.99 with a step of 0.1 except for the last step which was 0.09 .The figure bellow shows the findings :

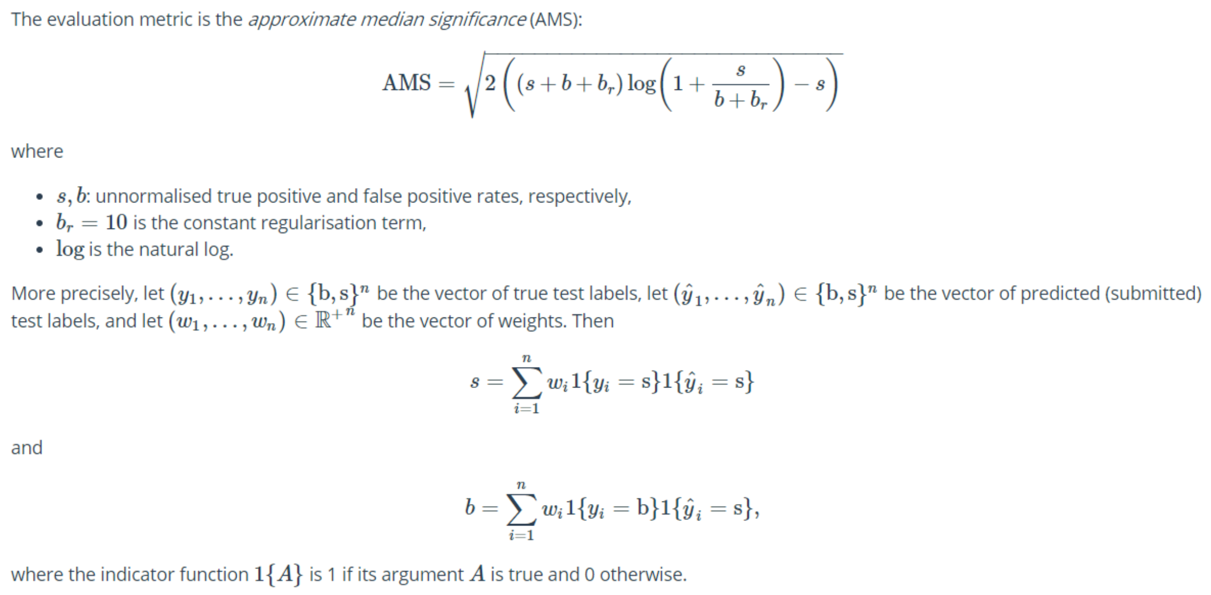


It seems that the interesting region to zoom in is between 0.8 and 0.9 , and that's what I exactly did , as the following figure depicts :



Hence , the best threshold is actually 0.86 , however we kept in mind the second best value which is depicted by a less higher peek and which s equal to 0.84 .

Afterwards , I decided to focus more on the AMS metric which is defined and explained as the following :



The evaluation metric is the *approximate median significance* (AMS):

$$\text{AMS} = \sqrt{2\left((s+b+b_r)\log\left(1+\frac{s}{b+b_r}\right)-s\right)}$$

where

- s, b: unnormalised true positive and false positive rates, respectively,
- $b_r = 10$ is the constant regularisation term,
- $\log$ is the natural log.

More precisely, let $(y_1, \ldots, y_n) \in \{\text{b}, \text{s}\}^n$ be the vector of true test labels, let $(\hat{y}_1, \ldots, \hat{y}_n) \in \{\text{b}, \text{s}\}^n$ be the vector of predicted (submitted) test labels, and let $(w_1, \ldots, w_n) \in \mathbb{R}^{+n}$ be the vector of weights. Then

$$s = \sum_{i=1}^{n} w_i 1\{y_i = \text{s}\} 1\{\hat{y}_i = \text{s}\}$$

and

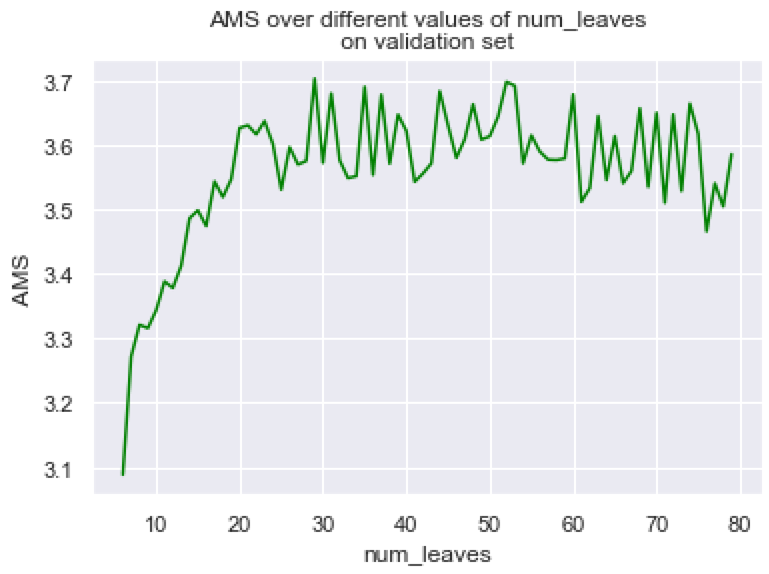
$$b = \sum_{i=1}^{n} w_i 1\{y_i = \text{b}\} 1\{\hat{y}_i = \text{s}\},$$

where the indicator function $1\{A\}$ is 1 if its argument A is true and 0 otherwise.

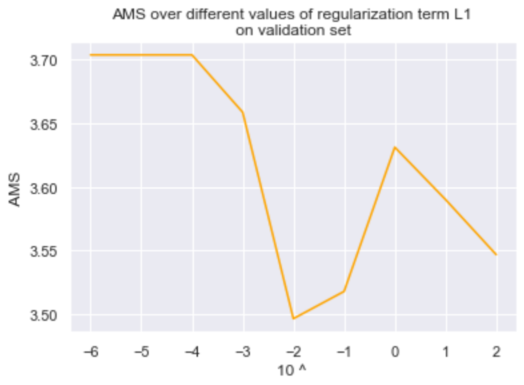
Reference : http://opendata.cern.ch/record/328

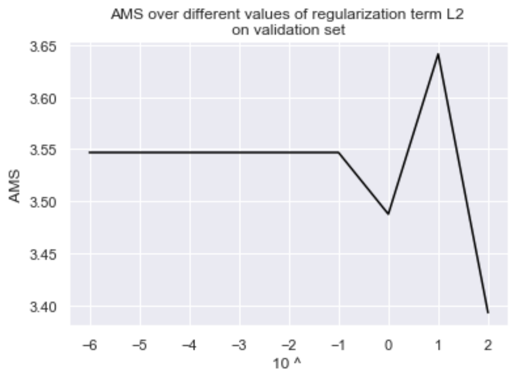
I tried the following non-exhaustive list of hyper-parameters :

- **Num_leaves** :
  I tested a range of (6 , 80) with a step = 1 .
  As you can see in the figure below , the best num_leaves value fetched is **29 ,** also we could see that it's not stable and keeps oscillating , thus we prefer take Top-3 first peeks and the experiments has shown that the best value was literally the first peek and best peek with num_leaves = 29 .



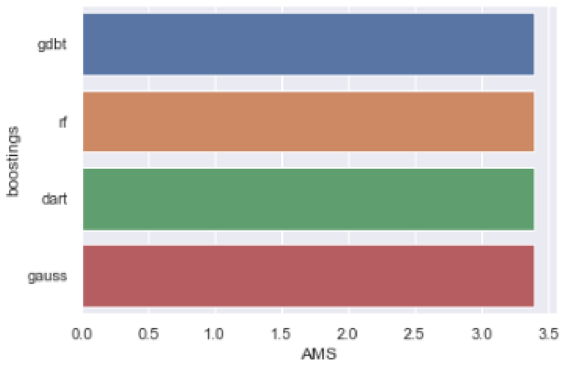
- **L1 regularization + L2 regularization :**

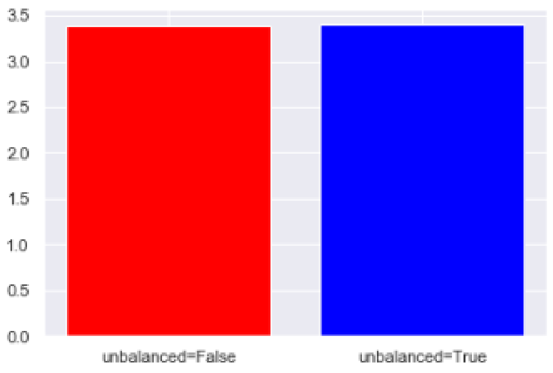




We could realize that both the regularization L1 and L2 here has no effect because We did attain 3.7 on our Validation set before without it .Thus , our model actually does not really suffer from an overfitting . We think that , it’s due to the nature of LightGBM as detailed before . the sub-sampling reduces drastically the risk of overfitting .

- **Boosting types + is_unbalanced :**
  We have tried to set the balancing weights on training samples , and we tested also several boosting types :
  Gbdt : traditional Gradient Boosting Decision Tree
  Rf : Random Forest
  Dart :Dropouts meet Multiple Additive Regression Trees
  Goss : Gradient-based One-Side Sampling

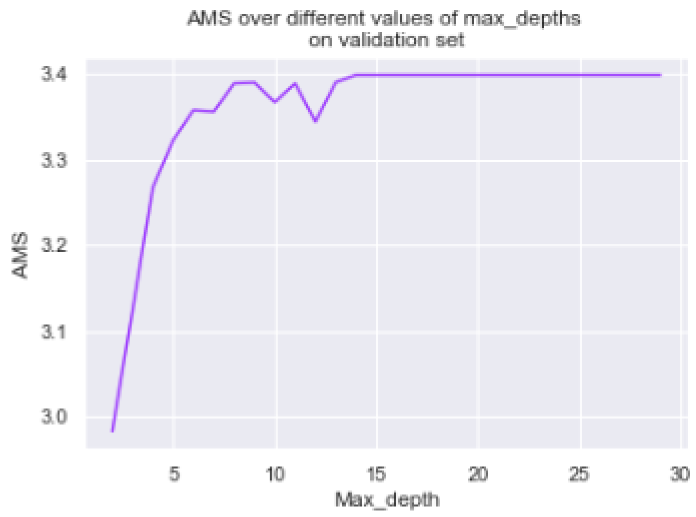




The results remain the same , None of them has improved the AMS
We continue all the way with other hyper parameters .

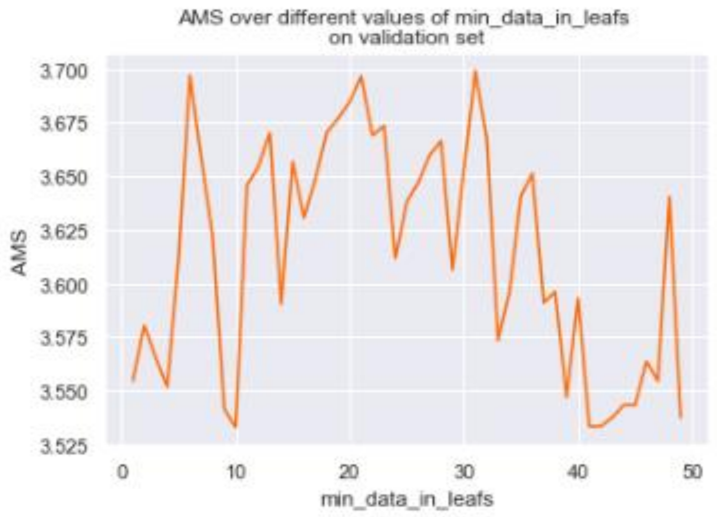
- **Max_depth :**

max_depth of the tree is a crucial hyper-parameter in Decision Trees based models. Even though theoretically, we have the guarantee of convergence whether we choose a depth of 2 , in practice we opt to efficiency and speed by fixing a maximum_depth/num_iterations trade off .



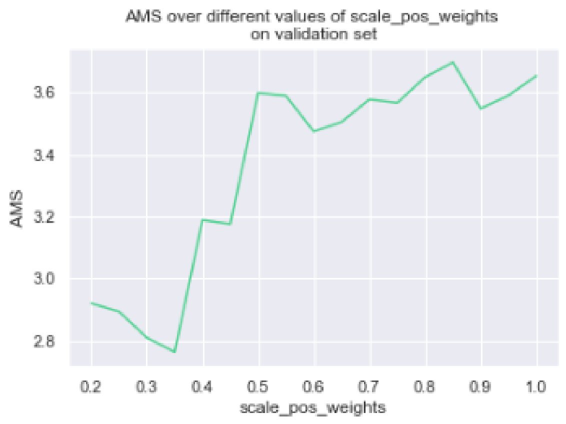
This is the type of curves that we love dealing with in machine learning , it ensures stability and for this we are very grateful !
We could pick up the best value which is **max_depth = 14 .**

- **Min_data_in_leaf :**
  We followed the same methodology as previously , but this parameter is actually very instable , thus we picked up the top 3 peeks , and after different experiments , it seems that the best one min_data_in_leaf =31 was the real best one .

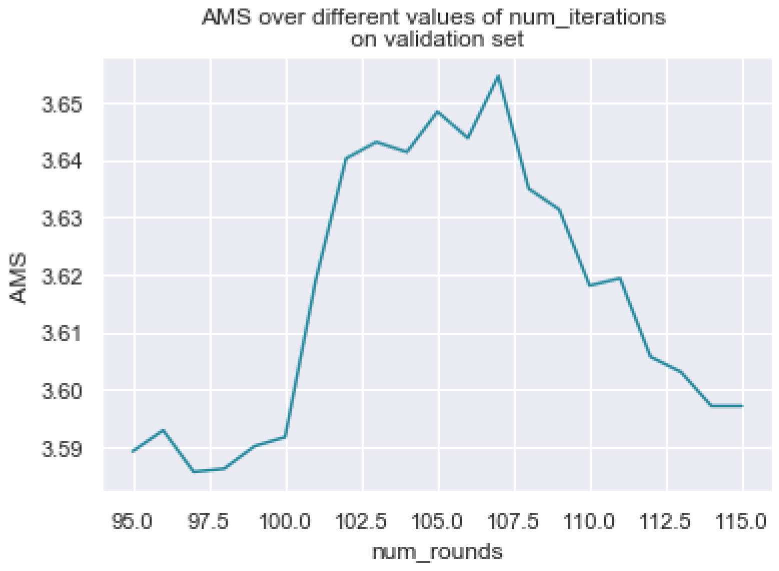


- **Scale_pos_weights :**

weight of labels with positive class , a hyper-parameter for adjusting the positive class penalizing , and in our case it seems to be very crucial , because we want our model to never miss signals .So, after selecting the top region and trying different configurations , we conclude that the best value was **0.96** .



We also checked out the **num_rounds** parameter , leading to this figure



We selected the best value with the higher peek mountain value , it was 107 rounds !

We now end up with a fine-tuned model with the following hyper parameters :

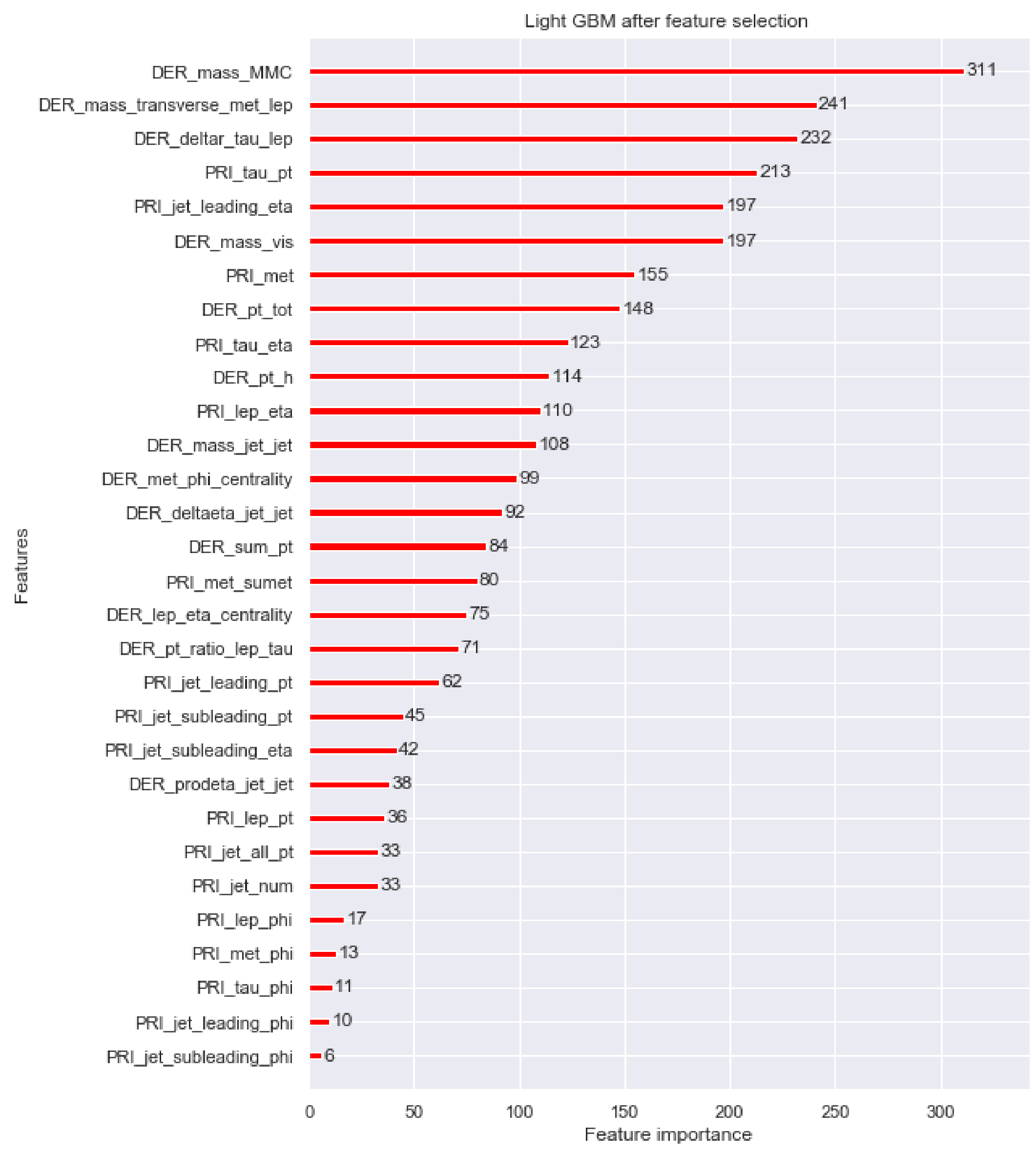
```
params['early_stopping_round '] = 5
params['learning_rate'] = 0.1
params["num_leaves"] = 29
```

params['max_depth'] = 14
params["min_data_in_leaf"] = 31
params["scale_pos_weight"] = 0.96

num_round = 107
threshold = 0.84

the features importance according to our model is in the figure bellow :

```
Plotting feature importances...
```

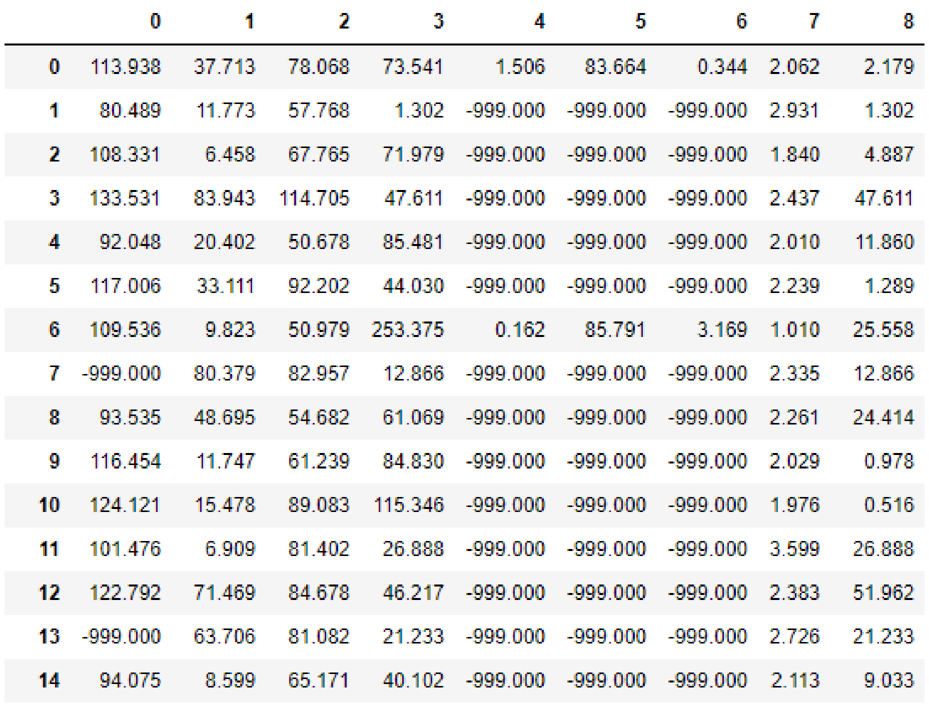


As it was expected the top feature is DER_mass_MMC followed by some relevant features!

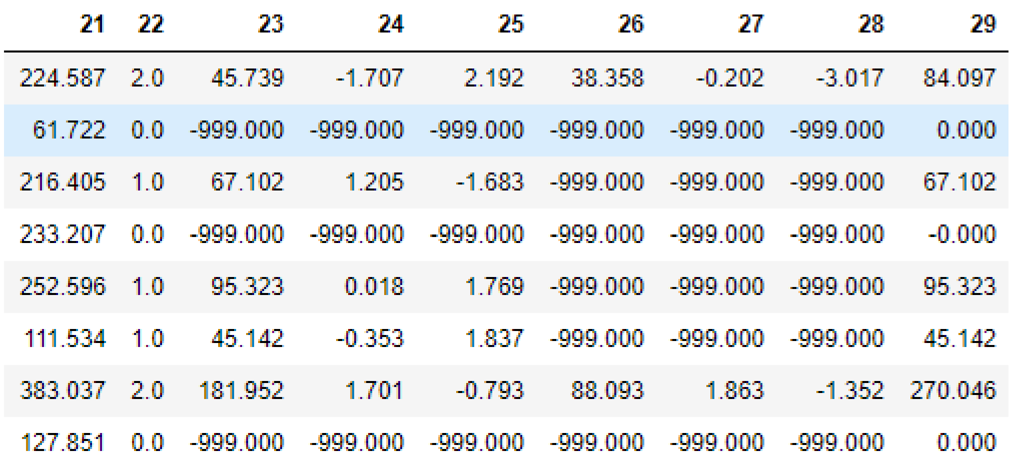
Analyzing the Error :

Now we will try to analyze the error trying to figure out the weaknesses of our models and why it is not able to predict correctly ! After analyzing the **False Negatives** (where we expect our model to detect the signal and it s no the case) , we realized that most of the data where he made mistakes was basically full of missing values from all missing features .
Whose columns are 4,5,6,12,23,24,25,26,27,28

As the following portion of data shows :



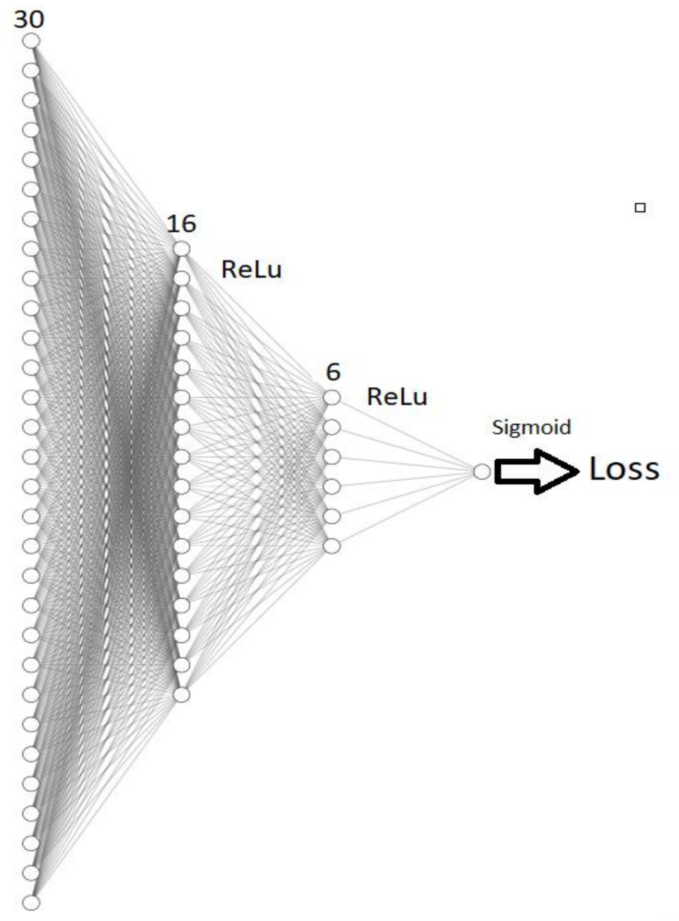
	0	1	2	3	4	5	6	7	8
0	113.938	37.713	78.068	73.541	1.506	83.664	0.344	2.062	2.179
1	80.489	11.773	57.768	1.302	-999.000	-999.000	-999.000	2.931	1.302
2	108.331	6.458	67.765	71.979	-999.000	-999.000	-999.000	1.840	4.887
3	133.531	83.943	114.705	47.611	-999.000	-999.000	-999.000	2.437	47.611
4	92.048	20.402	50.678	85.481	-999.000	-999.000	-999.000	2.010	11.860
5	117.006	33.111	92.202	44.030	-999.000	-999.000	-999.000	2.239	1.289
6	109.536	9.823	50.979	253.375	0.162	85.791	3.169	1.010	25.558
7	-999.000	80.379	82.957	12.866	-999.000	-999.000	-999.000	2.335	12.866
8	93.535	48.695	54.682	61.069	-999.000	-999.000	-999.000	2.261	24.414
9	116.454	11.747	61.239	84.830	-999.000	-999.000	-999.000	2.029	0.978
10	124.121	15.478	89.083	115.346	-999.000	-999.000	-999.000	1.976	0.516
11	101.476	6.909	81.402	26.888	-999.000	-999.000	-999.000	3.599	26.888
12	122.792	71.469	84.678	46.217	-999.000	-999.000	-999.000	2.383	51.962
13	-999.000	63.706	81.082	21.233	-999.000	-999.000	-999.000	2.726	21.233
14	94.075	8.599	65.171	40.102	-999.000	-999.000	-999.000	2.113	9.033



21	22	23	24	25	26	27	28	29
224.587	2.0	45.739	-1.707	2.192	38.358	-0.202	-3.017	84.097
61.722	0.0	-999.000	-999.000	-999.000	-999.000	-999.000	-999.000	0.000
216.405	1.0	67.102	1.205	-1.683	-999.000	-999.000	-999.000	67.102
233.207	0.0	-999.000	-999.000	-999.000	-999.000	-999.000	-999.000	-0.000
252.596	1.0	95.323	0.018	1.769	-999.000	-999.000	-999.000	95.323
111.534	1.0	45.142	-0.353	1.837	-999.000	-999.000	-999.000	45.142
383.037	2.0	181.952	1.701	-0.793	88.093	1.863	-1.352	270.046
127.851	0.0	-999.000	-999.000	-999.000	-999.000	-999.000	-999.000	0.000

First of all , we tried to fix these examples on the valid set **manually** to see the improvement that we could achieve at cutting edge , and it turns out that it was slightly improving the AMS with 0.03 or 0.04 which is not significant , and thus not very important .

Deep learning
We have implemented a simple feed forward network of 2 hidden layers , an input layer , and an output one , the figure below depicts the utilized architecture and



We 've tried different learning rates , batch_size , optimizers (SGD , Adam , AdaDelta , NAG , …) , num_epochs , and We could not make it work and learn something , the loss curve was weird .

Unseen Data (100 000 examples : public leaderboard)

We remind you that the test data "unseen" is the data marked as "leaderboard" .
Testing our fine-tuned LightGBM model gives us **an AMS of 3.5427**

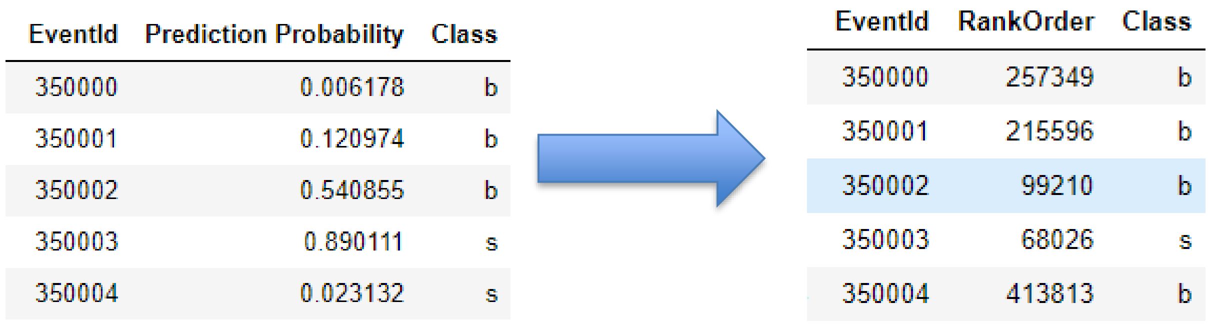
Then , we got a kind of whim to peruse this project and we decided to keep the same model but this time we will not train it only on the training_data (90% of the given training data) but we will recombine train and validation sets and train our model with the same hyper parameters and surprisingly we got **an AMS of 16.61** **,** the following table illustrates Precision / Recall and F1-measure after training on this whole given training data

Based on \ Metric	Precision	Recall	F-measure
Signal	0.837	1	0.91
background	1	0.62	0.76

Unseen Data (550 000 : public + private leaderboard)

Testing our fine-tuned LightGBM model on the huge test data set results **with an AMS of 5.02846.**

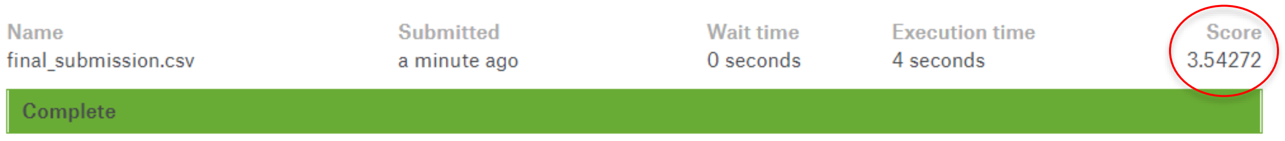
After submitting on Kaggle to get a real result , We've been realizing that we should also add a ranking column which rank the submitted examples based on their probabilities (How confident the model is to do such a prediction) , and it seems to hurt peformances .
the following figure illustrate this :



EventId	Prediction Probability	Class
350000	0.006178	b
350001	0.120974	b
350002	0.540855	b
350003	0.890111	s
350004	0.023132	s

EventId	RankOrder	Class
350000	257349	b
350001	215596	b
350002	99210	b
350003	68026	s
350004	413813	b

The final fresult was **3.54272** :



Name	Submitted	Wait time	Execution time	Score
final_submission.csv	a minute ago	0 seconds	4 seconds	3.54272

Complete