Multivariate analysis of data of channel $D^0 \to \pi^-\pi^+$ collected during the Run-2 at LHCb, based on Boosted Decision trees training with a AdaBoost algorithm.

Luca Toscano

February 26, 2022

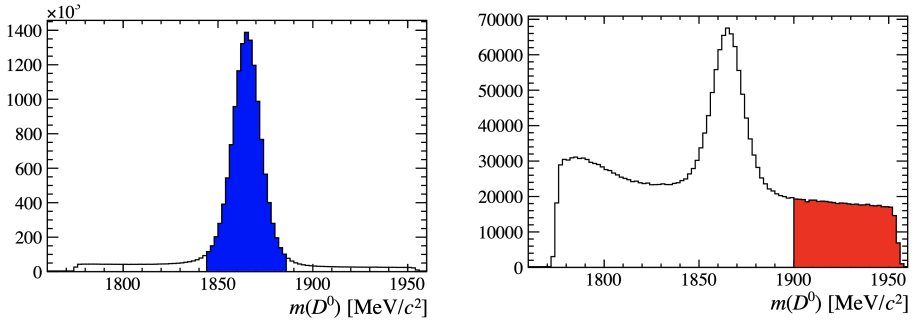
1 Introduction

Charm physics offers a unique opportunity to probe for physics beyond the Standard Model in the up-quark sector by searching indirect effects of unknown particles in loop processes. In particular, the CP violation in charm decays remains one of the main aspects of the Standard Model to be studied. The LHCb collaboration plays a fundamental rule in the charm sector research and the ntuple used in this works are collected during the Run-2 at LHCb.

Among all the possible charm decays, the $D^0 \to K^-K^+$ and $D^0 \to \pi^-\pi^+$ decays, are the most promising channels for searching for direct CP violation as, thanks to the relative large branching ratios, they give the opportunity to reach experimental precision below per mille level.

Before being ready for the analysis, the data require a selection. The first selection is an offline selection but after this a not negligible fraction of background events is still present. For this reason, to improve the signal significance a multivariate analysis is performed on the two channels $D^0 \to K^-K^+$ and $D^0 \to \pi^-\pi^+$. For both channels the $K\pi$ sample is used as signal proxy in the training, while invariant mass side-bands of the two decay modes are considered as background. In this project the selection will be performed only on the $D^0 \to \pi^-\pi^+$. The $K\pi$ sample is chosen because it includes a negligible amount of background. and it has kinematic properties very similar to the signal. Therefore, the training of the signal is performed using events lying in the $1844-1884$ MeV/c^2 invariant mass window of the $K\pi$. For the Background, the sideband $m(\pi\pi) > 1900$ MeV/c^2 is used. The windows of invariant mass chosen for the trainign of signal and background are shown in fig 2.In this window most of the events are purely combinatorial so can be considered as pure background events.

Both for signal and background sample, the dataset is divided in 80% for the training of the AdaBoost algorithm and the remaining part for the testing.



(a) (b)

Figure 1: The $m(D^0)$ distribution for the (left) $K\pi$ and (right) $\pi\pi$ samples. The signal region is shown in blue, the background regions in red. The amount of the background under the signal peak of the $K\pi$ sample is very small and for this reason the events around the peak are used as signal proxy in the training of the BDT.

The ntuple of LHCb contains millions of data, during the training and the test only 100000 events for the signal and 100000 for the background are used, both coming from the data collected in the 2018. After the training, the classification will be performed on a larger dataset of the channel $D^0 \to \pi^-\pi^+$ of the same year.

Ten kinematic variables are used for the training:

- DTF χ^2/ndf(B)
- $\log\{\chi^2[\mathrm{FD}(D^0)]\}$
- $\log\{\chi^2[\mathrm{IP}(h^+)]\}$
- $\log\{\chi^2[\mathrm{IP}(h^-)]\}$
- $P_T(h^+)$
- $P_T(h^-)$
- $(vtx_z(D^0) - vtx_z(B))/(\sqrt{\sigma^2(vtx_z(D^0)) - \sigma^2(vtx_z(B))})$
- $m(B)$
- $m_{corr}(B)$
- number of SPD hits

The label FD and IP stand for the flight distance and the impact parameter computed with respect to the PV associated to the B candidate, respectively.

After the training, the AdaBoost algorithm assigns to each event a label equal to -1 or +1, if the event is classified as background or signal respectively.

2 Description of the scripts

The project is composed by four scripts: AdaBoost.py, get_features.py, the training.py and main.py. In this section, the idea behind the training algorithm and all the passages of the scripts will be explained in details.

2.1 AdaBoost.py

Adaptive Boosting or AdaBoost is a statistical classification meta-algorithm formulated by Yoav Freund and Robert Schapire. The idea behind this method of classification is to combine different decision stumps in order to create a more complex classifier. In the decision trees language, a stump is a tree with only one node and two leaves. An events which cross a stump is classified in the $+1$ leave or in the -1 leave depending on whether the condition in the node has been satisfied or not. In this analysis the conditions are threshold values of one of the 10 kinematic variables described in the introduction. These variables are called "features" of the dataset, in the algorithm.

AdaBoost.py file contains the classes `DecisionStumps` and `AdaBoost Algorithm`. The class of the DecisionStumps contain two methods: `__init__(self)` and `predict(self, X)`. In the initialization method there are four instance attributes

```
def __init__(self):
        self.polarity = 1
        self.feature_index = None
        self.threshold = None
        self.alpha = None
```

The `feature_index` and `threshold` are respectively the kinematic variable and the threshold value chosen for the node condition. The `polarity` indicate how the events are classified in function of the threshold. If the polarity is 1 (default value) the events belong to the class $+1$ if their value of the chosen feature is higher then the threshold and to -1 if it's lower. If the polarity is -1 the classification works in the opposite way. The `alpha` attribute is the "amount of say" of the stump. It is a value which quantifies how much the stumps classifies well the events and it's defined as

$$\alpha = \frac{1}{2} ln\left(\frac{1 - TotalError}{TotalError}\right). \tag{1}$$

The $TotalError$ will be defined later. If α has an high positive value, the stump is a good classifier, if α is close to 0, the stumps classifies the events in a casual way and if α has a negative value, the stumps classifies the events very bad.

The method `predict(self, X)` takes as argument the stumps itself and the dataset used later to train the algorithm. It returns the array of the prediction of the dataset, after have classified each event in function of the polarity, the threshold and the feature, as explained before.

The second class defined in this file is the `class AdaBoost Algorithm` which contain the core of the algorithm. This class has three method `__init__(self, n_of_classifier = 5)`, `fit(self, X, y)` and `predict(self, X)`. When an object of the this class is created, the users can set the number of classifiers , first attribute of the initialization method, which will compone the array pf the classifier, the second attribute.

The method

```
def fit(self, X, y):
```

performs the training of the algorithm. It takes as arguments the training dataset, both the columns of the features `X` and the column of the label. The number of samples and the number of features (10 in our case) is taken directly from the dataset. The second line of `fit` is the definition of the weight. The weight is a parameter assigned at each sample in the dataset, used by the algorithm to be more careful to the events "harder to classify". The events start all with the same weight equal to $1/n_o f_s ample$ because the weight are always normalize to 1. Later the value will be modified depending on the right or wrong classification of the event.

The algorithm is composed by three cycles:

```
for classifier_i in range(self.n_of_classifier)
for feature_i in range(n_of_features)
for threshold_i in thresholds
```

The training is done over all the features and all the possible threshold for each feature and it is redone as many times as the number of classifiers requested by the user. At the beginning of the first cycle, the classifier is defined as an object of the class `DecisionStumps` and the `min_error` is defined. Then the second cycle begins. Here the array of possible threshold is extracted from the dataset, by looking for all possible values of the current feature. In the last cycle, for each threshold the array of the prediction label is created by looking if each event has the value of the current feature higher or lower than the threshold. The error of the current classifier is computed by the sum of the weight of the wrong classified events. The cycle end with two if:

```
if error > 0.5:
error= 1 - error
pol = -1
if error < min_error:
classifier.polarity = pol
classifier.threshold = threshold_i
classifier.feature_index = feature_i
  min_error = error
```

The first change the polarity, initially equal to 1, if there are more wrong classification events than right classification events, so the label -1 will be assigned to the events higher than the threshold and viceversa. The second if change the

classifier with the current, if the error of the current classifier is smaller then all the previous classifiers. Once the two inner cycles are finished, the first classifier of the algorithm is chosen. Thus, the amount of say of the classifier is computed with the formula 1 and the weight are updated with the information coming from the first classifier training. The formula for the new weights is

$$w = exp(-\alpha\ y\ h(x)), \tag{2}$$

where the y is the known label of the event in the dataset and $h(x)$ is the label predicted by the training. When the prediction coincide with the true label, the exponent is negative and the weight of the event decrease respect to the initial weight, while if y and $h(x)$ are discordant, the exponent is positive and the weight will be higher than before. This mechanism allowed to the algorithm to pay more attention to the events classified wrongly in the next run of the cycle of the classifier.

The last method, verb?predict(self, X)?, uses the array of the classifiers, learned in the `fit` method, to make prediction on the new dataset passed as argument. The formula for the predicted label of each event of the tested dataset is

$$y_{pred} = sign\left(\sum_{c}^{N} \alpha_c\ h(x)_c\right), \tag{3}$$

where $\sum_c^N$ is the sum of all the classifiers composing the algorithm, α_c is the amount of say and $h(x)_c$ is the prediction label of the classifier.

2.2 get_features.py

The data used for the analysis contain the variables used to compute the ten kinematic variables listed in section 1. This script contains only one function `get_features(signal, bkg)`. It receives as two arguments the data sample for the training of the signal and the data sample for the training of the background. The function uses the pandas library firstly to modify the sample in order to contain only float variables and then to compute the kinematic variables required for the training. At the end, the two arguments are returned as pandas object `data_frame_signal, data_frame_bkg` having as rows all the events of the sample and as columns the ten kinematic variables which will be the features used in the AdaBoost algorithm.

2.3 training.py

In this script the training of the algorithm is done and evaluated. The script is divided in four functions: `get_dataset(csv_signal,csv_bkg)`, `accurancy (label_true, label_predicted)`, `cutter (array_to_cut, first_row, last_row)` and the `training (csv_signal,csv_bkg, n_classifier= 20, cut_low = 0, cut_high = 100000)`.

The first function takes as arguments the two .csv files containing the dataset for the training of the signal and the background. The dataset are transformed

in pandas object `dataset_signal`, `dataset_bkg` having as rows all the events of the sample and as columns the raw variables of the of the Root Trees containing the original data. Then the function $get_features()$, from the get_features.py file, is called and it takes the two dataset as arguments. The pandas object returned by the funciton `data_frame_signal`, `data_frame_bkg` are transformed in numpy array (`array_signal`, `array_bkg`) in order to use methods of the sklearn library.

The column of the labels, 1 for the signal training dataset and -1 for the background training dataset, are created and merged together by using the method `np.concatenate`. Even the `array_signal`, `array_bkg` are merged. Finally with the method `train_test_split`, imported from `sklearn.model_selection`, the two dataset array are divided in

- `X_train` = columns of the features used for the training;
- `X_validation` = columns of the features used for the evaluation of the algorithm accurancy;
- `y_train` = column of the label used for the training;
- `y_validation` = column of the label used for the evaluation of the algorithm accurancy;

The dataset is splitted randomically (`random_state=1`) in the 80% for the training and 20% for the evaluation test.

The second function is

```
def accurancy (label_true, label_predicted):

    accurancy = np.sum(label_true == label_predicted) / len(label_true)
    return accurancy
```

This is a simple function used to test the well operation of the training. It returns the normalized sum of all the predicted labels of the dataset which are equal to the true label, in our case, `y_validation`.

The third function is

```
def cutter (array_to_cut, first_row, last_row):

    array_cutted=np.delete(array_to_cut, slice(first_row, last_row), axis=0)
    print('Size of cutted array: {} \n'.format(array_cutted.shape))
    return array_cutted
```

It is used to select the length of the dataset used for the training. The last function is

```
def training (csv_signal,csv_bkg, n_classifier = 20, cut_low = 0, cut_high = 100000):

    X_train, X_validation, y_train, y_validation = get_dataset(csv_signal,csv_bkg)
```

```
classification = Adaboost_Algorithm(n_classifier)

print('The training is starting ...')

classification.fit(X_train, y_train)
y_pred = classification.predict(X_validation)
acc=accurancy(y_validation,y_pred)
print('The accurancy of the algorithm is : ',acc)

# save the model to disk
filename = 'Ada_classification.sav'
pickle.dump(classification, open(filename, 'wb'))
```

You can give as argument of the function, the .csv files you want to use for training the algorithm,the number of classifiers you want your AdaBoost algorithm has, and the value of first and last row of the datasets in the files, you want to use in the training. Some default values are setted. The function will perform the training and the evaluation test automatically. The final model will be salved in Ada_classification.sav. Thus you will be able to use the classification model without redo every time the training.

2.4 main.py

The main script can be used to classify the dataset selected by the user with a classification model. `def Classification (csv_dataset, model):` takes as argument the .csv file containing the dataset and the model used for the classification in a .sav format. The `csv_dataset` must contain the 14 variables used to compute the 10 kinematic variables used in the training (features) plus the column of the D^0 invariant mass, used to visualize the final results. The script is pretty simple and easy to understand and modifiy, even with the helps of the comments.

3 Results

An example of classification performed with a model trained by the AdaBoost algorithm implemented in this project is reported here. 200000 events of $\pi\pi$ decay detected at LHCb in 2018 are classified. The distribution of the invariant mass of D^0 before (blue) and after (orange) the classification is reported. The model is trained with only 10000 events of signal taken from the region under the peak $K\pi$ samples, and 10000 events of background taken from a region far from the signal peak of $\pi\pi$ samples, as explained in Section 1. The accurancy of the classification of the model used is the 81.5%. By using larger datasets fo the training it sould be possible to reach even the $90 - 95\%$.

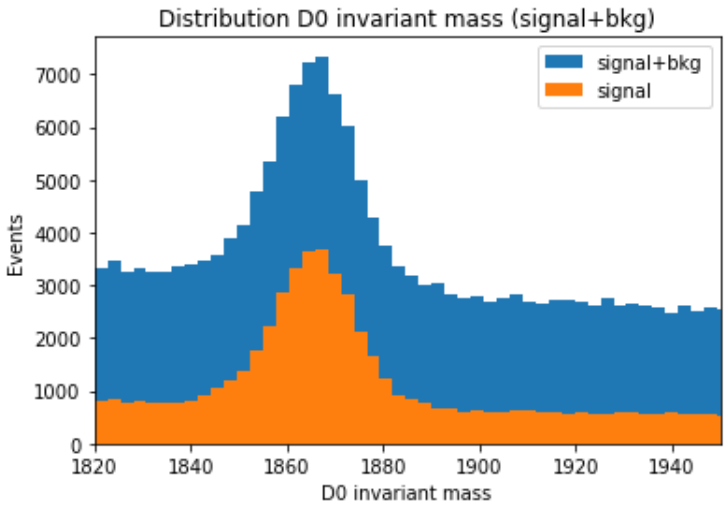


Figure 2: The $m(D^0)$ distribution for the (left) $K\pi$ and (right) $\pi\pi$ samples. The signal region is shown in blue, the background regions in red. The amount of the background under the signal peak of the $K\pi$ sample is very small and for this reason the events around the peak are used as signal proxy in the training of the BDT.