Xgboost Parameter Tuning

Noe Tepec Tinoco, CDMX, Mexico, 09/10/2020

Objective

- Creating a single function which will helps us to tune our model and gives us the best set of hyperparameters for it.
- Previously, we had been working on this by constructing individual functions for each hyperparameter. Now, we add all of them together and make extra steps to ensure this is the best combination of them.
- We named this new function as following.

```
def tune_params(train_X, train_Y, test_X, test_Y, test_sig_x, train_sig_x, test_b_x, train_b_x, show_plots=True):
```

- With it, we only need one single line to do it.

```
In [30]: tune_params(train_x, train_y, test_x, test_y, test_signal_x, train_signal_x, test_bkgnd_x, train_bkgnd_x, True)
```

Overall functionality

- Let's see now how it works…

In [29]:
```
def tune_params(train_X, train_Y, test_X, test_Y, test_sig_x, train_sig_x, test_b_x, train_b_x, show_plots=True):

    b_par={} #A void dictionary which will contain the best params in each iteration

    #TUNING TREES
    #INTIAL MODEL
    model=xgb.XGBClassifier(objective='binary:logistic',
                            learning_rate=0.2,
                            max_depth=7,
                            min_child_weight=5,
                            reg_lambda=1010,
                            n_estimators=150)

    #NUMBER OF TREES
    eval_set = [(train_X, train_Y), (test_X, test_Y)]
    eval_metric = ["auc","error"]
    %time model.fit(train_X, train_Y, eval_metric=eval_metric, eval_set=eval_set, verbose=True)

    # Evaluation results
    evaluation_results = model.evals_result_
    auc_train = evaluation_results["validation_0"]["auc"] # Train 'auc' metric
    auc_test = evaluation_results["validation_1"]["auc"] # Test 'auc' metric

    if show_plots:
        # Plotting 'XGBOOST Classification Error' and 'XGBOOST Classification AUC'
        plt.figure(figsize=(7,7))
        plt.plot(auc_train)
        plt.plot(auc_test)
        plt.xlabel('Number of trees (n_estimators)')
        plt.ylabel('auc')
        plt.legend(['Train', 'Test'])
```

```
param_test1 = {
          'n_estimators':[100,150,200]
         }

gsearch1 = GridSearchCV(estimator = XGBClassifier( objective= 'binary:logistic',
                                                   learning_rate=0.2,
                                                   max_depth=6,
                                                   min_child_weight=2,
                                                   reg_lambda=1010),

                                param_grid = param_test1, scoring='roc_auc',n_jobs=4,iid=False, cv=5)

gsearch1.fit(train_X,train_Y)

b_par['n_estimators']=gsearch1.best_params_['n_estimators']
```

```
#DEPTH AND WEIGHT TUNING
v1=0

param_test2 = {
          'max_depth':[5,6,7],
          'min_child_weight':[4,5,6]
         }

gsearch2 = GridSearchCV(estimator = XGBClassifier( objective= 'binary:logistic',
                                                   learning_rate=0.2, n_estimators=150, max_depth=6,
                                                   min_child_weight=2,
                                                   reg_lambda=1010),

                                param_grid = param_test2, scoring='roc_auc',n_jobs=4,iid=False, cv=5)

gsearch2.fit(train_X,train_Y)
#gsearch2.cv_results_, gsearch2.best_params_, gsearch2.best_score_
v1+=1
```

```
while gsearch2.best_score_ < gsearch1.best_score_ and v1!=3:

    param_test2 = {
         'max_depth':[5,6,7],
         'min_child_weight':[4,5,6]
        }

    gsearch2 = GridSearchCV(estimator = XGBClassifier( objective= 'binary:logistic',
                                                       learning_rate=0.2,
                                                       n_estimators=b_par['n_estimators'] ,
                                                       max_depth=6,
                                                       min_child_weight=2,
                                                       reg_lambda=1010),

                                param_grid = param_test2, scoring='roc_auc',n_jobs=4,iid=False, cv=5)

    gsearch2.fit(train_X,train_Y)
    #gsearch2.cv_results_, gsearch2.best_params_, gsearch2.best_score_
    v1+=1

if gsearch2.best_score_ < gsearch1.best_score_:
    b_par['max_depth']=6
    b_par['min_child_weight']=1
    gsearch2.best_score_ = gsearch1.best_score_
    print(b_par)
else:
    b_par['max_depth']= gsearch2.best_params_['max_depth']
    b_par['min_child_weight']= gsearch2.best_params_['min_child_weight']
```

Output

- As you can see, the function gives us a dictionary with the best hyperparameters we can use when constructing our model.
- With that we construct an XGBoost Model.

```
{'n_estimators': 200, 'max_depth': 7, 'min_child_weight': 4, 'gamma': 0.05, 'subsample': 1, 'colsample_bytree': 1, 'reg_alph
a': 1e-05, 'reg_lambda': 800}
```

```
model = xgb.XGBClassifier(objective="binary:logistic",
                          learning_rate=0.2,
                          n_estimators=b_par['n_estimators'],
                          reg_lambda=b_par['reg_lambda'],
                          reg_alpha=b_par['reg_alpha'],
                          max_depth=b_par['max_depth'],
                          min_child_weight=b_par['min_child_weight'],
                          gamma=b_par['gamma'],
                          subsample=b_par['subsample'],
                          colsample_bytree=b_par['colsample_bytree'])
model.fit(train_X, train_Y)
```

An extra step to make

- After all this process, our last step is to verify the overfitting and AUC score of the ROC curve. If it satisfies the conditions related to ChiSquare in overfitting, and AUC score in the ROC curve, then we can assure that our list of hyperparameter is ideal.

```
#OVERFITTING AND ROC CURVE
if (chi2_signal <= 100 and chi2_bkgnd <= 100) and (auc_score_train >=0.90 and auc_score_test >= 0.90):
    print(b_par)
else:
    print("You should consider repeating this procedure.")
```

Evaluating the results

- Signal overfitting and ChiSquare

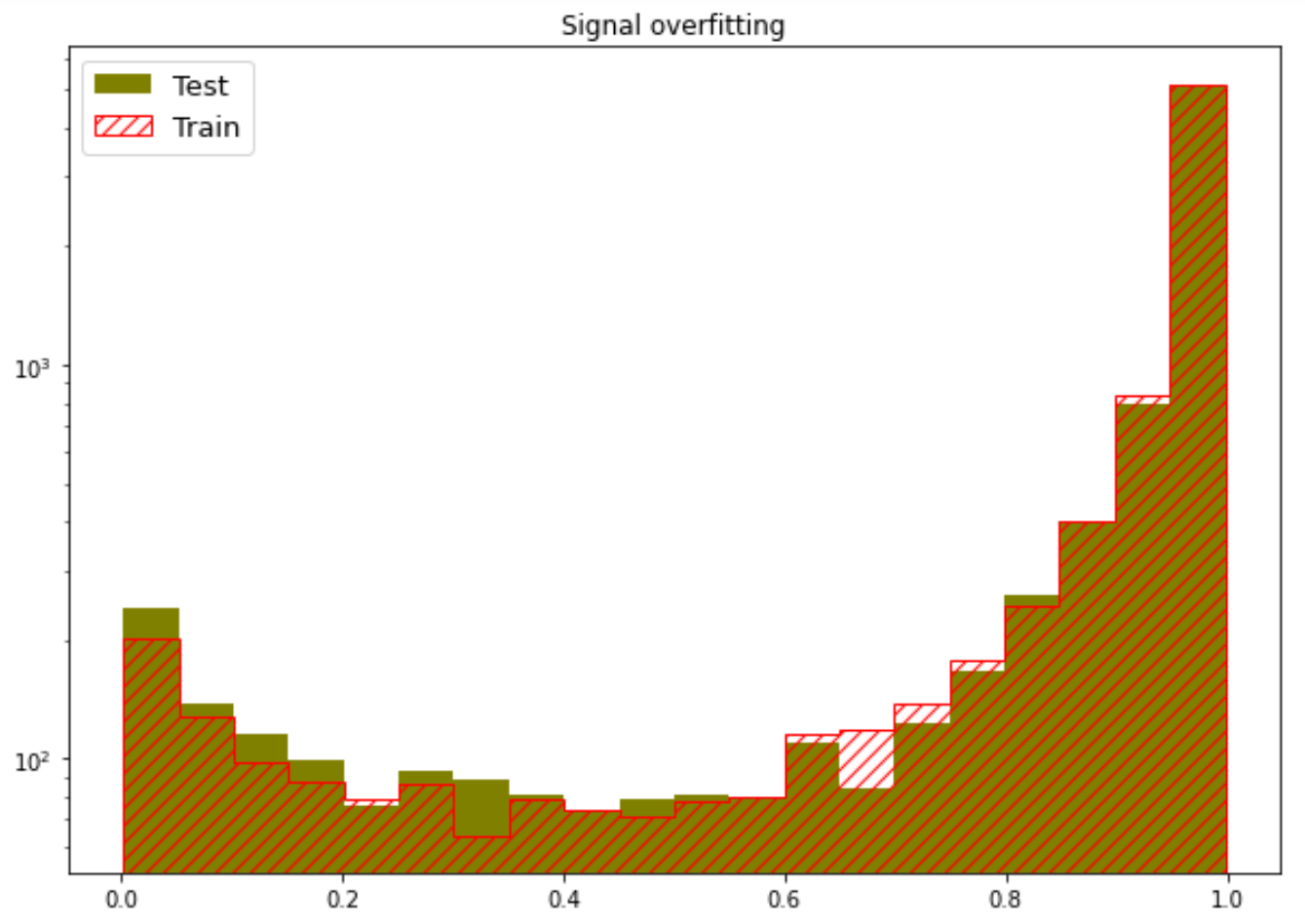
If the ChiSquare is less than a certain value, let's say 5, then we can assure that the overfitting in the model is not enough that we have to repeat this procedure all over again. In this case, it works.

```
chi2_signal=0
lista=[]

for i in range(len(m[0])):
        num=((m[0][i]-n[0][i])**2)/n[0][i]
        chi2_signal=chi2_signal+num

chi2_signal=chi2_signal/20
print(chi2_signal)
```

```
2.077164928060373
```



Evaluating the results

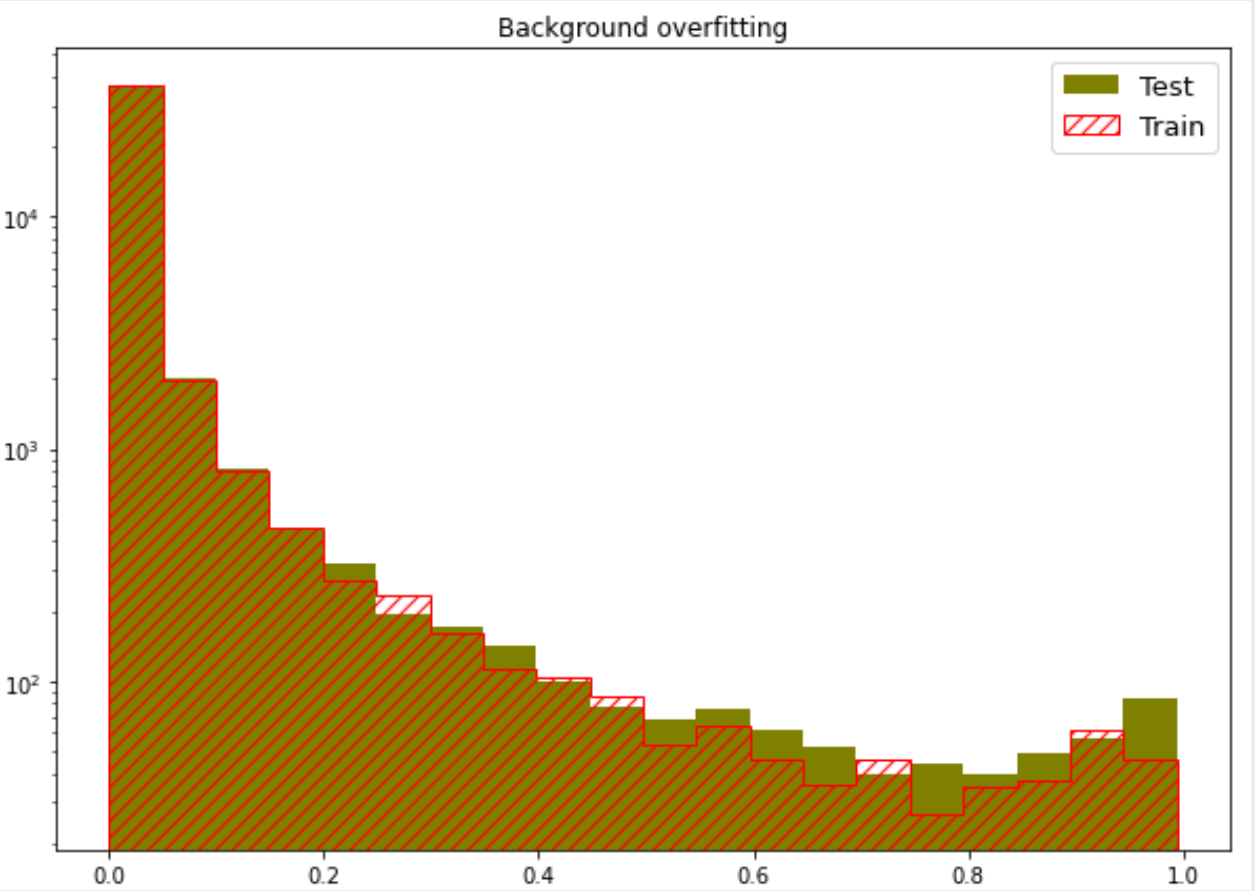
- Background overfitting and ChiSquare

In a similar way, we see now that it also works, but with a greater error.

```
chi2_bkgnd=0
for i in range(len(m[0])):
    num=((m2[0][i]-n2[0][i])**2)/n2[0][i]
    chi2_bkgnd=chi2_bkgnd+num

chi2_bkgnd=chi2_bkgnd/20
print(chi2_bkgnd)
```

```
4.7893111154165675
```



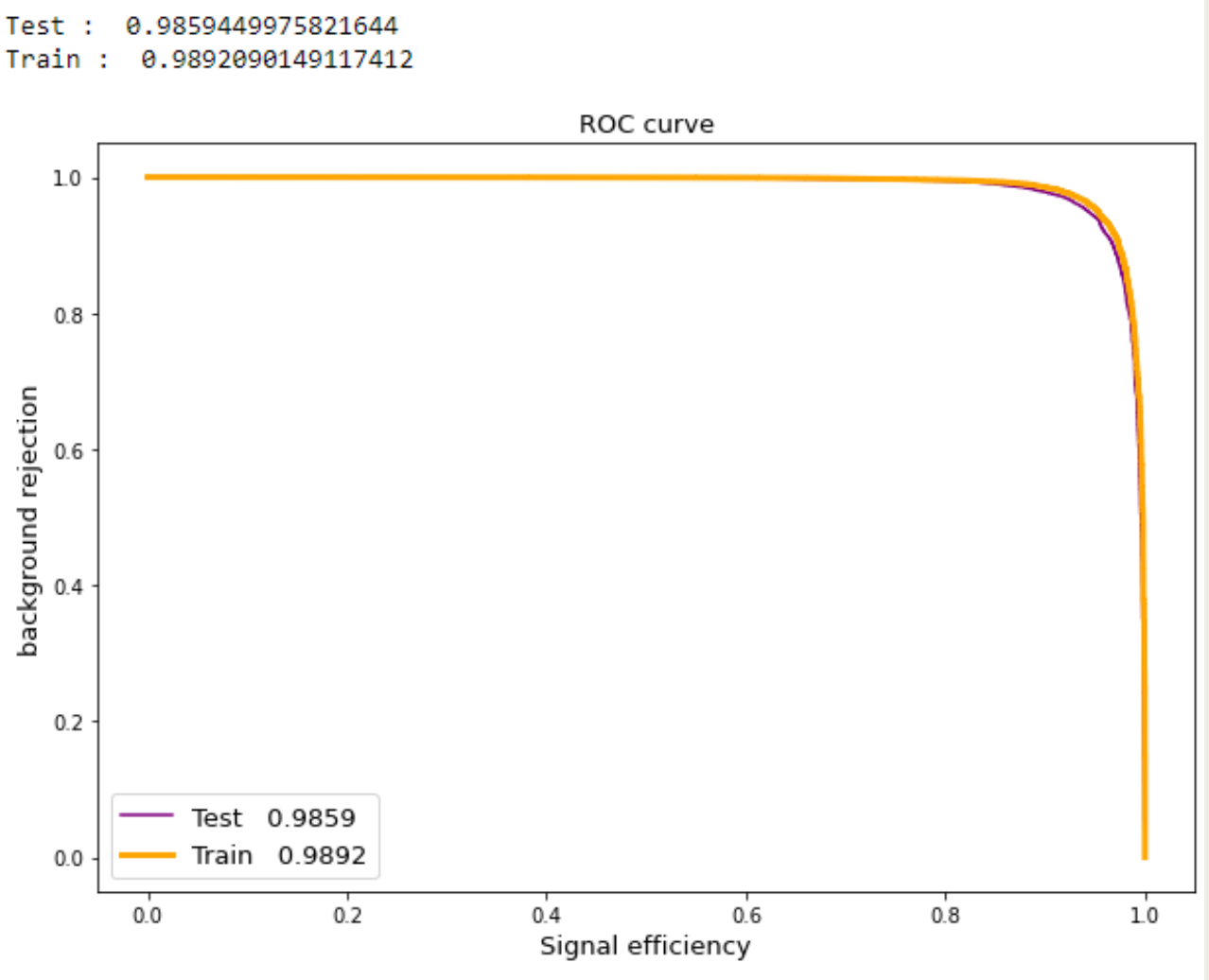
Evaluating the results

- ROC Curve

With the original program that Israel gave us, we previously had, for each set:

Test: 0.9827
Train: 0.9865

Success! As we can see, this method is currently working, but there is still work to do.



What's next? The final steps

- To include this last independent parts of the code into the general function.
- (Optional) Implementing the Kolmogorov-Smirnoff test to really ensure that the overfitting is not to great in the background and signal plots.
- To transform this

```
def tune_params(train_X, train_Y, test_X, test_Y, test_sig_x, train_sig_x, test_b_x, train_b_x, show_plots=True):
```

- Into this

```
def tune_params(train_X, train_Y, test_X, test_Y, show_plots=True):
```

- Integrating the parameter tuning into Daniel's set of functions to obtain the final program for XGBoost.