GROUP 2408 – XGBoost Analysis : Model, dimensionality reduction and Neural Networks comparison

Cucchetto Filippo, Simioni Federico, Amorosetti Gabriel, and Quaglio Emanuele
(Dated: April 3, 2024)

The goal of this work is to study the performances and characteristics of XGBoost. We studied the model complexity and regularization by experimenting different parameters to find the simplest but most effective XGBoost model that keeps a good validation accuracy for our specific task (and assess the impact of regularization). After this, we studied the effect of using reduced data samples to evaluate the model's accuracy when the least important features are absent. Finally, we compared the validation accuracy of XGBoost with the one of a simple feed-forward neural network, to determine which one performs better at lower sample sizes.

INTRODUCTION

XGBoost is a gradient boosting library implementing machine learning algorithms. It is an ensemble learning method based on weak learners (i.e learners that are just slightly better than a random guessing), such as decision trees, in order to improve a prediction. A number of decision trees are trained on different subsets of the data, and the final prediction is a combination of each tree's prediction [1]. Compared to classical gradient boosting techniques, XGBoost have the advantages to include regularization parameters to keep generalization while improving the model performance, and uses a sequence of decision trees, where each tree tries to correct the errors of the precedent one. The idea is to associate to each element of the dataset a leaf of the tree according to a certain weight value, usually denoted with "w". The correct predictions are given a low weight whereas the incorrect ones are given a higher weight. XGBoost iteratively refines its predictive capabilities by sequentially adding decision trees to its ensemble: in this learning algorithm we can find the best weights that minimizes the cost function $C(\mathbf{x}, g_A)$. To achieve that we minimize $\Delta C_{t-1,t}$, at every iteration new weights are found and it continues until the algorithm converges or the maximum depth of a tree is reached. To deal with this model many parameters have to be taken into account, and we list and detail some of them, which were also used for our work [2] : `learning_rate` is the step size shrinkage used in update to prevent overfitting, `max_depth` is the maximum depth of a tree, `reg_lambda` is the $l2$ regularization term on weights, `n_estimators` is the number of trees to be built in the ensemble.

METHODS

Model complexity, parameters and regularization

To obtain a better performance from the XGBoost model, it's necessary to properly tune multiple hyperparameters, associated both to the model (hypothesis class) itself and to the learning algorithm and its optimization.

We explore the 5-dimensional space of the parameters listed above. We use the `GridSearchCV` class from the `sklearn` library, that tests the accuracy of any combination of parameters through a 4-fold cross-validation on the training set (75% of the original dataset). At first, the search is performed logarithmically, to span many orders of magnitude efficiently.

```
parameters0 = {
      'importance_type' : ['gain', 'weight', 'cover', 'total_gain', 'total_cover'],
      'learning_rate' : list(map(lambda x: 10**x, range(-5, 1, 1))),
      'reg_lambda' : list(map(lambda x: 10**x, range(-5, 1, 1))),
      'n_estimators' : range(50,300, 50)
          }
```

FIG. 1: Hyperparameters spanned by first XGBoost grid search.

Then, we focus on the most promising ranges, as obtained from the former coarse-grained search, and explore them linearly, introducing also a variable `max_depth`.

The parameters combination with the highest accuracy (0.94825) is: `best_params={'importance_type': 'gain', 'learning_rate': 0.1111111111111111, 'max_depth': 10, 'n_estimators': 200, 'reg_lambda': 0.0}`. We notice how the resulted values of all three `max_depth`, `n_estimators` and `reg_lambda` imply a highly complex model.

In order to look for the simplest yet effective and accurate model, we select the subset of the combinations of parameters evaluated above, that produced an accuracy higher than a threshold we deem sufficiently good (`good_accuracy=0.9`). Then, we look for the combinations that minimize the complexity, defined in a proper way.

Indeed, we define a `complexity_estimate` function that takes as arguments the max depth of the trees, the number of estimators and the $l2$-norm of the model parameters. We take inspiration from the form of the Ω-regularization defined in [3] (sum over estimators of number of leaves plus $l2$-norm):

- summing over trees corresponds to multiply by `n_estimators` in first approximation (we assume tree 'complexity' to be homogeneous);

- we assumed the number of leaves to be roughly $\sim 2^{\texttt{max\_depth}}$, but after some trials we find out that approximating the dependency as linear (with a weight `u`) leads to a more stable complexity estimation w.r.t. the `max_depth` parameter;
- finally we assume that the bigger the `reg_lambda` parameter is, the smaller the $l2$-norm will be kept by the learning algorithm. Thus we add to the complexity a term inversely proportional to `reg_lambda`, weighted by a `v` parameter.

We tune `u` and `v` manually. The goal is to quantify *a priori* (i.e., before training) the complexity of the model as well as possible.

```
import pandas as pd
from itertools import product as CartProd
print('best parameters combination of global grid search, with score associated:\n ', GSCV.best_params_,GSCV.best_score_ )
#print(clf.param_grid)
param_combs=list(CartProd(*clf.param_grid.values()))
df=pd.DataFrame(param_combs, columns=['importance_type', 'learning_rate', 'reg_lambda', 'n_estimators', 'max_depth'])
#print(df)
df['mean_accuracy']=GSCV.cv_results_['mean_test_score']
good_accuracy=0.9
df_good=df[df['mean_accuracy']>good_accuracy]
def complexity_estimate(n_estimators, reg_lambda, max_depth, accuracy):
    u,v=0.021,1
    return n_estimators*(u*max_depth+v/reg_lambda)*(1-accuracy)
df_good['complexity']=complexity_estimate(df_good['n_estimators'], df_good['reg_lambda'], df_good['max_depth'], df_good['mean_accuracy'])
# Finding the row with the minimum value in column 'A'
min_row_index = df_good['complexity'].idxmin()
# Retrieving the row corresponding to the minimum value in column 'A'
min_row = df_good.loc[min_row_index]
print('LESS COMPLEX YET SUFFICIENTLY GOOD MODEL:\n', min_row)
```

FIG. 2: Selection of the 'simplest yet sufficiently good' XGBoost model from the spanned hyperparameters combinations.

In order to incentivize small increase in complexity as far as they would bring to big accuracy improvement, we also multiply the complexity function described above by a $1 - \texttt{accuracy}$ factor.

Given this procedure, the selected parameters are the following:

```
simple_good_params= {'importance_type':
'gain', 'learning_rate':  0.222222,
'reg_lambda':  1.0, 'max_depth':  4,
'n_estimators':  50}
```

Dimensionality reduction

We analyze the dataset in order to detect and select the most important features, which allows us to reduce the dimensionality of the data.

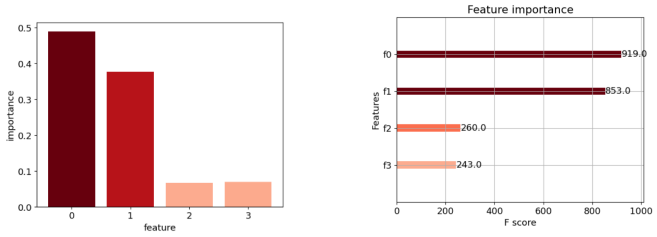


FIG. 3: Feature Importance

The histogram in Fig. 3, representing features' importance [4], suggests to focus our attention on features zero and one [5].

Then we repeat the training process on these reduced dataset. The results of the classification obtained with this dataset can be seen in Fig. 8 and Table II.

We also experiment with increasing the dimensionality of our dataset in a sensible way, based on visual interpretation. Accordingly, we consider both the 0-th and 1-st features and some quadratic functions applied on them. Their exact definition is shown in the code in Fig. 4.

```
x=np.vstack((x[:,0], x[:,1], (x[:,1]+5)**2, (x[:,0]+20)**2+(x[:,1]+10)**2, (x[:,0]-20)**2+(x[:,1]-5)**2))
x=x.T
N,L = len(x), len(x[0])
N_train = int(0.75*N)
x_train,y_train = x[:N_train],y[:N_train]
x_test,y_test = x[N_train:],y[N_train:]
```

FIG. 4: Transformed dataset

We plot (Fig. 5) four different pairs of features belonging to the new 5-dimensional dataset space, to visualize how the above defined transformations can help the classifier.

We train the default XGBoost model on the augmented dataset, and obtain a much improved accuracy (0.98), visually confirmed by the plot of the model prediction in Fig. 8.

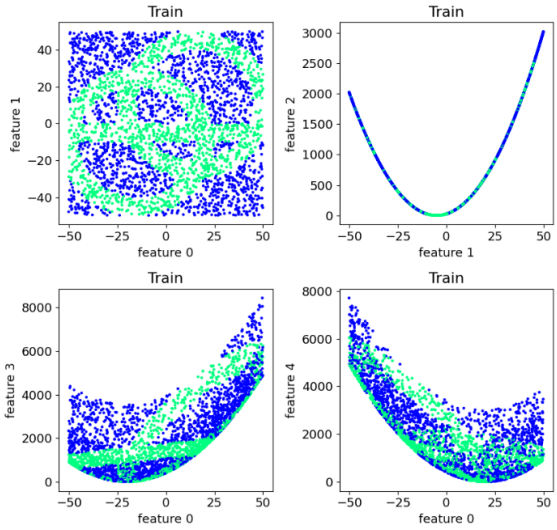


FIG. 5: Higher dimension plots

Neural Networks comparison

Finally, to compare with other models suitable for the purposes described earlier, we decided to contrast the validation accuracy of XGBoost with that obtained from a Feed-Forward Neural Network (FFNN). To achieve this, we have experimented with various hyperparameter values, as outlined in the Fig. 6

Utilizing `KerasClassifier` and `GridSearchCV`, we have arrived at identifying a single combination (see Results paragraph).

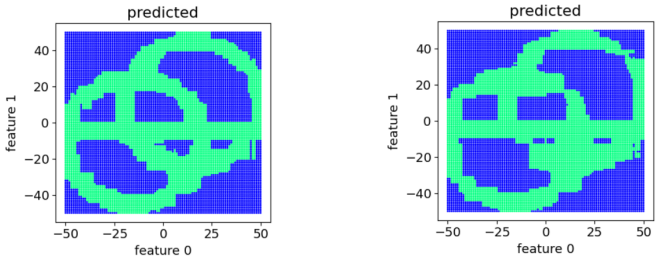
```
param_grid = {
    'optimizer': ['rmsprop', 'adam', 'sgd'],
    'init': ['glorot_uniform', 'normal', 'uniform'],
    'activation': ['softmax', 'relu', 'sigmoid'],
    'neurons': [8, 12, 16, 20],
    'epochs': [50, 100],
    'batch_size': [5, 10, 20],
    'dropout_rate': [0.0, 0.1, 0.3],
    'learning_rate': [0.001, 0.01, 0.1],
    'momentum': [0.0, 0.2, 0.4]
}
```

FIG. 6: Parameters chosen.

With this model, in order to establish a comparison with the previously discussed XGBoost model, we first examined the trends of validation accuracy as a function of the fraction of samples used for training relative to the total samples. Then, in order to proceed further with cross-validation for all cases, we did the same thing just varying the number of folds of the k-fold validation process.

RESULTS

The classifier resulting from the model simplification procedure described in *Model complexity, parameters and regularization* scores a sufficiently high accuracy (0.902) on the test set, and the number of estimators and the depth are reasonably low (50 and 4 respectively). The *l2*-regularization term is 1, contributing to keep the model simple. In Fig. 7 we plot the result of the prediction of the model on a square grid of points, and compare it to the default XGBoost model. In Fig. 5 we show the expected labels of the points of the training dataset. We consider the 0-th and 1-st features only. We can confirm visually that the performance of the model is satisfying. To confirm our success at reasonably limiting the model



(a) Default XGBoost `xgb_model` (b) Simplified XGBoost `simple_good_model`

FIG. 7: Prediction comparison of simplified vs default XGBoost model

complexity, we verify that, despite the good performance of the simplified model, still the training time is 37% faster than the default XGBoost model (39.9 s and 62.9 s respectively).

Model	Accuracy	Error
Gradient Boosting model	80.50%	19.50%
XGB model	94.80%	5.20%
Simple Good Model	90.20%	9.80%

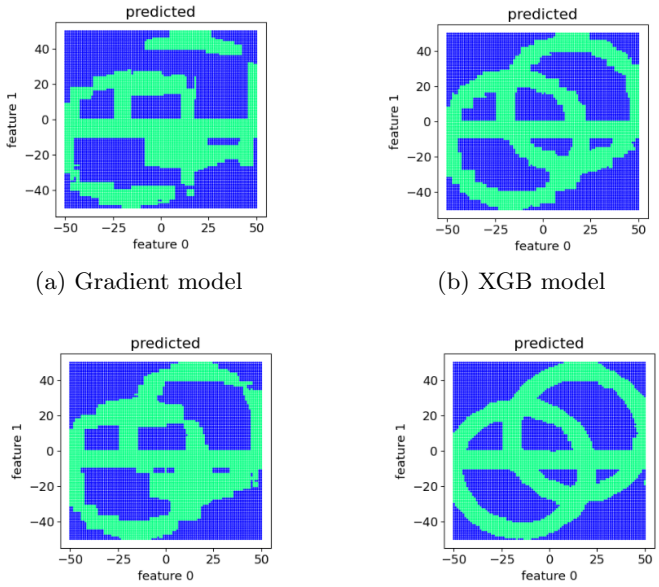
TABLE I: Accuracy and Error Rates of Different Models with original dataset.

In Table II we report the results obtained with various model, discussed in the section dimensionality reduction (`Gradient Boosting model`, `Xgb model`, `Simple good model` , `Higher dimensionality model`)

Model	Accuracy	Error
Gradient Boosting model	83.70%	16.30%
XGB model	95.50%	4.50%
Simple Good Model	90.20%	9.80%
Higher Dimensionality Model	98.00%	2.00%

TABLE II: Accuracy and Error Rates of Different Models with reduced dataset.

We report the plots of features 0 and 1 for these four models



(a) Gradient model (b) XGB model

(c) Simple good model (d) Higher dim. model

FIG. 8: Comparison of Different Models

As we can see the "Higher Dimensionality Model" give us the best accuracy and it minimizes the error.

Finally, regarding the results obtained in the last section, those concerning the comparison with Feedforward Neural Networks (FFNN), we proceed considering just the best combination, whose parameters are reported here:

`best_params={'activation': 'softmax',`

```
'batch_size': 10, 'dropout_rate': 0.0,
'epochs': 100, 'init': 'normal',
'learning_rate: 0.001, 'momentum': 0.0,
'neurons': 20, 'optimizer': 'rmsprop'}
```

Concerning the comparison between validation accuracies as a function of the fraction of train samples (so N' w.r.t. N, or total ones), the graphs depicting the respective trends for XGBoost and the FFNN are presented in Fig. 9

We observe a similar shape in the XGBoost accuracy

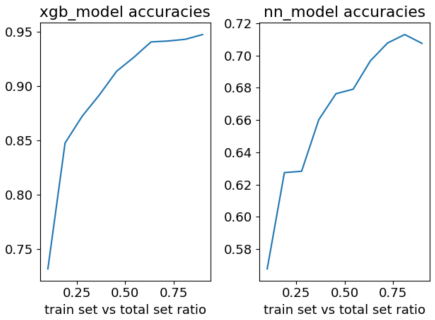


FIG. 9: Validation accuracy w.r.t. N'/N.

and neural network accuracy vs training ratio, but the XGBoost model performs way better, with a $> 20\%$ difference in accuracy. The main difference we observe is a decrease in the accuracy for very high values of training ratio for the neural network model. Compared to the non-optimized model, the trend of this NN is much more stable and less oscillating. Furthermore, exploring the lower range of N' we observe that the progression of the accuracy for the XGBoost is much more stable than out simple NN model (see Fig. 10). In particular, we observe that the accuracy is higher for the xgb_model even for small size training set. The main difference in trend is in the steadily and quickly increasing accuracy of the XGBoost model w.r.t. the NN model.

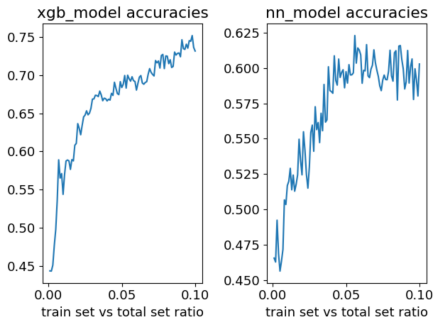


FIG. 10: Validation accuracy for low N'/N.

And then, finally, for what concerns cross validation we obtained what reported in Fig. 11, where the xgb_model shows as well in this case a steadily higher accuracy value w.r.t. the NN model.

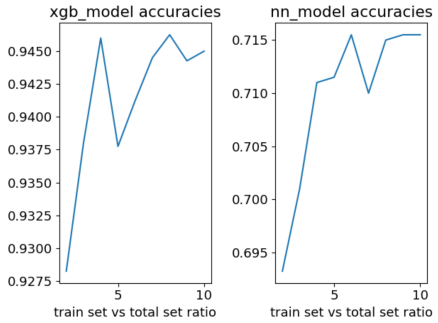


FIG. 11: Cross validation.

CONCLUSIONS

In the first part we have looked for the best hyperparameters to tune the XGBoost model, taking into account both the accuracy and the complexity of our model. We have managed to define a simple enough yet well-performing model. In part two we have observed how the accuracy of the models has improved by reducing the dataset dimension, but in fact the highest accuracy has been obtained by training the model on an augmented dataset, enriched with ad-hoc transformed features. In the final part, we have compared the test and (cross-) validation accuracy of the XGBoost model and that of a Feedforward Neural Network (FFNN). As evident from the results presented, XGBoost proved to be superior, although some enhancements could be applied to the neural network. For instance, performing a better hyperparameter optimization might strongly improve the FFNN model performance. [6]

[1] Chen Tianqi, Carlos Guestrin, "XGBoost: A Scalable Tree Boosting System." Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016.
[2] XGBoost documentation: https://xgboost.readthedocs.io/en/stable/
[3] Mehta *et al.*, A high-bias, low-variance introduction to Machine Learning for physicists, Physics Reports **810**, 1–124 (2019). doi:10.1016/j.physrep.2019.03.001
[4] Https://machinelearningmastery.com/feature-importance-and-feature-selection-with-xgboost-in-python/.
[5] We see that the standard PCA is uncapable of distinguish them instead, since the variance of the 2 and 3 coordinates is as much as high, despite them being essentially noise.
[6] Each step of the researching, coding and writing process behind this paper was completed cohesively and cooperatively by all four group members.