Using Bayesian optimization to select gradient boosting model for multilabel classification

***Abstract*—We use the xgboost implementation of gradient boosting principle to solve the problem. The xgboost model is carefully selected to maximum performance while control overfitting using Bayesian optimization approach**

***Keywords*—gradient boosting, bayesian optimization, multilabel classification**

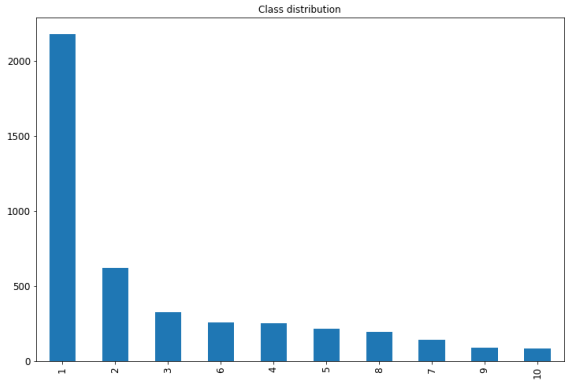

Fig. 1. Histogram showing class distribution

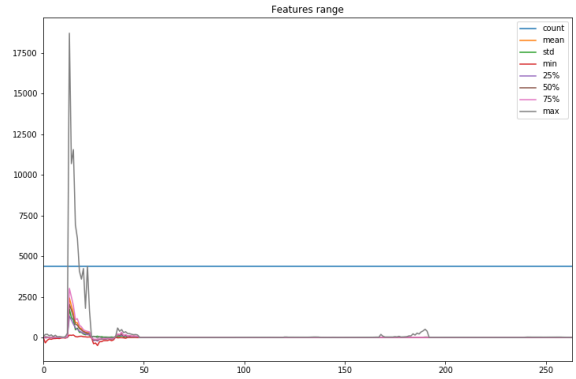

Fig. 2. Descriptive statistics of the data

I. Introduction

I expected to learn about applying multiple machine learning algorithm into a real problem

The question being addressed is: whether or not gradient boosting is viable in this problem and how to select the best model with hyperparameter tuning technique

Why this task is important ? * In order to find out the best model

* IN order to select the best model within the shortage amount of time

II. Data analysis

A. Class distribution

The data has unbalanced class distribution, with class 1 take majority of samples. Therefore, the classification will show heavy bias to class 1 if we don't have any appropriate method. Figure 1 show the unbalanced class distribution

B. Descriptive statistics

Figure 2 shows the descriptive statistics of the data. The horizontal axis contains all of the predictors, with their mean, min, max and quantiles values are shown by the vertical axis. We can see there is some outliers in the first 50 predictors of the dataset

Figure 3 shows the visualization of the data by the first three orthogonal components that explain the maximum amount of variance. It can be seen that the data seems to be clustered due to the effect of the outliers

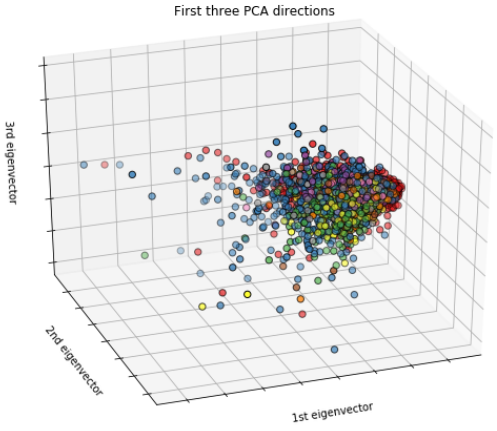

Fig. 3. Visualization of the data

III. Methods and experiments

A. Data preprocessing

Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected. In practice we often ignore the shape of the distribution and just

transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.

As mentioned before, our data contains many outliers (some predictors has over 25% data points as outliers). Scaling using the mean and variance of the data like above mentioned is likely to not work very well because outliers can often influence the sample mean / variance in a negative way. Therefore I use RobustScaler (sklearn.preprocessing.RobustScaler) to transform our dataset, using statistics that are robust to outliers, i.e. the median and the interquantile range. To be specific, this Scaler removes the median and scales the data according to the Interquartile Range - the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile). Centering and scaling happen independently on each feature (or each sample, depending on the axis argument) by computing the relevant statistics on the samples in the training set.

In my experience, scaling the dataset often improve the performance of ensemble classifiers with trees as base estimators (using boosting or bagging technique)

B. Ensemble model and Gradient boosting:

I decide to choose boosting as the main method for this problem because boosting is one of the most powerful learning idea introduced in the last twenty years. [1, chap 10, p338]

The main idea of boosting is based on ensemble learning. In an ensemble mode, a committee of trees (weak learners or base learners) are trained and developed. A single tree model will classify the data points into different leaves, with the score associated with each of the leaves. The final ensemble model is made by combining the prediction of all the base learners, thus become more powerful than a single model. Ensemble learning can be broken down into two tasks: developing a population of base learners from the training data, and then combining them to produce the final prediction.

A boosting model is characterized by how the weak learners are developed and combined: the weak learners evolve over time at each boosting step by additive training (sequentially apply the weak classification algorithm to repeatedly modified versions of the data).

- Initially, each training observations are given an equal weight w_i and the classifiers (tree learners) is trained on the data in the usual manner
- For each successive iteration, the observation weights are individually modified so that misclassified data receive more influence. Each successive classifier is thereby forced to concentrate on those training observations that are missed by previous ones in the sequence. The scores are also calculated for each classifier so that we have the contribution of each classifier in the final model
- Finally, each of the base classifier casts a weighted vote in the final ensemble model to get the final prediction.

1) numerical optimization via gradient boosting:

C. XGBoost models

Xgboost (https://github.com/dmlc/xgboost) is chosen because of its high performance in many data science problems in a fast and accurate way.

According to its introduction, Xgboost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. XGboost is still based on gradient boosting framework but it use a more regularized model formalization to control over-fitting, which gives it better performance. It also utilized computational resources better for boosted tree algorithm

A xgboost model has multiple parameters (around 20 to 30) that need to be selected carefully in order to get the best performance while also regularize the overfitting.

Choose the right size for each tree

Regularlization

To control overfitting, we need to control the parameters "max_depth", min_child_weight and gamma. We can also add noise to the model by control the parameters "subsample" and "colsample_bytree " The problem of unbalanced dataset can be addressed by setting weight for each data point when training the xgboost model

D. Evaluation: stratified K-fold

Scoring accury and log loss

E. Tuning hyperparameters - Bayesian optimization vs Randomsearch

1) Randomsearch:

2) Bayesian optimization: Traditionally, hyperparameter tuning are done by grid search. However, it takes a lot of time to cover a wide range of parameters. Therefore we use Bayesian optimization (https://github.com/fmfn/BayesianOptimization)

This is a constrained global optimization package built upon bayesian inference and gaussian process, that attempts to find the maximum value of an unknown function in as few iterations as possible. This technique is particularly suited for optimization of high cost functions, situations where the balance between exploration and exploitation is important.

F. Experiments

IV. Results

A. performance measures

B. performance on Kaggle

After carrying out the bayesian optimization, we get the result of the best parameters here:

```
objective = 'reg:logistic',
learning_rate= 0.1,
n_estimators = 50,
gamma=0.1,
max_depth = 12,
min_child_weight = 20,
max_delta_step= 50,
```

```
subsample = 0.9,
colsample_bytree = 0.7,
reg_lambda = 0.1,
reg_alpha = 0.1,
```

Training xgboost with those parameter on the training set, then make prediction on the test set, we get the following score for test set:

* Accuracy: 0.64654 (position 95) - over the benchmark
* Log-loss: 0.17534 (position 58) - over the benchmark

V. Conclusion

Xgboost with gradient boosting works really well in this case

Bayesian opt let us to go over 20 runs (40 mins in total) to select the best xgboost model over a wide range of hyper-parameter. IF we use gridsearch, it is expected to spend 10 hours to get the same results

VI. Appendices

Code for tuning the xgboost with Bayes optimization

References

[1] Trevor Hastie, Robert Tibshirani, Jerome Friedman *The Elements of Statistical Learning*, 2nd edition, 2008
[2] Tianqi Chen *Introduction to Boosted Trees*, http://xgboost.readthedocs.io/en/latest/model.html
[3] Tianqi Chen *XgBoost Parameters*, http://xgboost.readthedocs.io/en/latest/parameter.html

Fig. 4. Source code

```
def xgbcv(learning_rate, n_estimators, gamma, max_depth, min_child_weight,
 ↪ max_delta_step, subsample, colsample_bytree, reg_lambda, reg_alpha):
        xgb_model =  xgb.XGBClassifier(
                objective = 'multi:softmax',
                learning_rate= max(min(learning_rate, 1), 0),
                n_estimators = int(n_estimators),
                gamma=max(gamma, 0),
                max_depth = int(max_depth),
                min_child_weight = int(min_child_weight),
                max_delta_step= int(max_delta_step),
                subsample = max(min(subsample, 1), 0),
colsample_bytree = max(min(colsample_bytree, 1), 0),
reg_lambda = max(reg_lambda, 0),
reg_alpha = max(reg_alpha, 0),
random_state=12345)
val = cross_val_score(xgb_model,
train_data, train_labels, scoring='accuracy',
cv=StratifiedKFold(n_splits=5, random_state=12345, shuffle=False)
).mean()
return val
gp_params = {"alpha": 1e-5}
xgbBO = BayesianOptimization(xgbcv,
{'learning_rate': (0.001, 1),
'n_estimators': (1, 100),
'gamma': (0.001, 100),
'max_depth': (1, 15),
'min_child_weight': (1, 20),
'max_delta_step': (1, 100),
'subsample': (0.3, 1),
'colsample_bytree': (0.001, 1),
'reg_lambda': (0.001, 100),
'reg_alpha': (0.001, 100)}
)
xgbBO.explore({'learning_rate': [0.001, 0.01, 0.1],
'n_estimators': [10, 20, 50],
'gamma': [0.001, 0.01, 0.1],
'max_depth': [4, 8, 12],
'min_child_weight': [5, 10, 20],
'max_delta_step': [10, 20, 50],
'subsample': [0.1, 0.45, 0.9],
'colsample_bytree': [0.2, 0.5, 0.7],
'reg_lambda': [0.001, 0.01, 0.1],
'reg_alpha': [0.001, 0.01, 0.1]})
xgbBO.maximize(n_iter=40, **gp_params)
print('-' * 53)
print('Final Results')
print('XGB: %f' % xgbBO.res['max']['max_val'])
```