Variational Gradient Boost — Novel Gradient Boosting Method

Agnij Moitra

High School Student
Birla Vidya Niketan, Pushp Vihar, Sector - IV, New Delhi - 110017

October 31, 2022

Abstract

Purpose — Gradient Boosting is a compelling ensemble method, such as XGBoost. It uses Decision Trees as estimators to create a feed forward loop in order to minimize the residuals errors. Which is prone to overfitting due to irrelevant features, outliers and external noise. Thereby, it is proposed that a variational feed forward loop of a combination of many different estimators 3 should be implemented.

Method — Variation Gradient Boosting (VGBoost), makes an initial prediction as the mean of the dependent feature. Then it calculates the residuals and uses these residuals as the dependent feature for the next layer. In each subsequent layer besides just relying on Decision Trees, it simultaneously trains many linear and non-linear models and chooses the model with least validation mean squared error (mse). This feed forward loop is repeated until the mse value converges to zero or it has exhausted the iterations.

Results [4.1] — On Scikit-Learn's California housing price dataset, XGBoost and VGBoost had a training and validation mse of 0.584 and 0.575, and 0.767 and 0.725. On Scikit-Learn's Friedman1 dataset, XGBoost and VGBoost had a training and validation mse of 0.113 and 0.163, and 0.367 and 0.235. Further on Scikit-Learn's make_classification dataset, XGBoost and VGBoost had a training and validation F1 score of 0.966, and 0.888, and 0.841 and 0.847.

Conclusion — The above benchmarking proves that testing various different machine learning models and making a feed forward loop would combat overfitting and thereby also improve the overall metrics [1].

[1] Mean Squared Error for regression and F1 Score for classification

1 Introduction

Gradient Boosting is an ensemble learning algorithm which creates a sequential feed forward ensemble based on the previous residual errors. In conventional gradient boosting algorithms [6], first an initial prediction which is the arithmetic mean of the dependent features is made. Then the residuals errors values are calculated which are the difference between the true dependent values and the predicted dependent values which is scaled by a learning rate in order to improve the generalization accuracy [9]. For each subsequent layer a decision tree is used to calculate the current dependent features, this is iterated n number of times where n is a hyper-parameter. Although Decision Trees which are used as the base estimator in gradient boosting are highly prone to over-fitting and having a high variance [3]. As a result, gradient boosted like its base estimators, decision trees, have low bias and high variance [4].

2 Reviewing Current Literature

2.1 Reducing variance and over-fitting in Decision Trees

1. Adding noise filters such as ENN, RENN, and ALLKNN for instance reduction [2].
2. Further, by pre-pruning and post-pruning the branches of the decision trees [1].
3. Making minimum Surfeit and Inaccuracy algorithm based on Kolmogorov complexity and breadth search traversal [5].

2.2 Reducing variance and over-fitting in Gradient Boosting Algorithms

1. Tuning hyper parameters such as the learning rate, limiting max depth of base trees and by early-stopping.
2. Removing confusing samples and outliers empirically shows that it would reduce the generalization error [8].

Further as a general principle variance can be reduced by early-stopping, network-reduction, data-expansion and regularization [10].

3 Variational Gradient Boosting Algorithm

3.1 Overview

Depending on the dataset, algorithms like Linear Regression, Ridge Regressor, LARS Lasso, RANSAC Regressor et cetera, and non linear algorithms such as SVM, K-nearest Neighbor, Bayesian Ridge et cetera, may have better accuracy than a decision tree.

VGBoost makes the initial prediction as the arithmetic mean of the dependent features, and calculates the difference between the true values and predicted values as the residual errors. For each subsequent layer it uses multi-threading to simultaneously train various models and chooses the one with the least Mean Squared Error, as defined in Model Selection 3. It simultaneously keeps track of the models which will be later used for making predictions. Then it scales the current predictions by a learning rate α, and calculates the new residuals. This process is repeated till it has exhausted n estimators, or the residual error is zero, or due to early stopping. Classification — It can be directly used for binary classification problem, for rest add Scikit-Learn's [7] OneVsRestClassifier on VGBoost Classifier. The complete implementation of VGBoost is in (`github.com/Agnij-Moitra/variational-gradient-boosting`).

Input: X_train: iterable
Input: y_train: iterable
▷ expects binary feature for classification
Output: None, creates ensemble
global init_preds = mean(X_train)
residuals[0] = y_train - init_preds
global ensemble = list() ▷ sequential list of models in each layer
for $i \in n\_estimators$ **do**
    $y[i] = residuals[i-1]$
    do in parallel
        $train\_test\_split(X\_train, y[i])$
        train all models
        model = get model with least Mean Squared Error
    $residuals[i] = y[i]$ - $model.predict(X_train) * \alpha$
    global $ensemble.append(model)$
    if $warm\_start$ **then**
        $X\_train[i] = y[i-1]$
    if *early_stopping or residuals[i] == 0* **then**
        break

Algorithm 1: VGBoost Fit Method

```
Input: X_test: iterable
Output: y_preds: numpy array
preds[0] = global init_preds
for i in range(len( global ensemble)) do
    preds[i] = ensemble[i].predict(X_test)
if classification then
    row_preds = row_wise_sum(preds)    ▷ add prediction of each layer
    final_preds = quantize(row_preds)    ▷ quantize value to 0 or 1
    return final_preds
else
    final_preds = row_wise_sum(preds)    ▷ add prediction of each layer
    return final_preds
```

Algorithm 2: VGBoost Predict Method

3.2 Model Selection

Based on the parameters provided, for each subsequent layer, VGBoost uses multi-threading to train the following model and chooses the one with least validation mean squared error [2], to change the time complexity of the algorithms used:

[2] It uses `train_test_split(*, train_size = 0.7)` in for this evaluation.

```
Input: light = True
Input: complexity = False
Input: custom_models = None
Output: None, sets model list
if light then
    models                                          ▷ Least Complexity (Default)
        ↪ LGBMRegressor, ExtraTreesRegressor,BaggingRegressor,
        RANSACRegressor, LassoLarsIC, BayesianRidge
if complexity then
    models                                          ▷ Maximum Complexity
        ↪ DecisionTreeRegressor, LinearRegression, BayesianRidge,
        KNeighborsRegressor, HistGradientBoostingRegressor, ElasticNet,
        LassoLars, Lasso, GradientBoostingRegressor,
        ExtraTreesRegressor, BaggingRegressor, NuSVR, XGBRegressor,
        SGDRegressor, KernelRidge, MLPRegressor, LGBMRegressor,
        Ridge, ARDRegression, RANSACRegressor, HuberRegressor,
        TheilSenRegressor, LassoLarsIC
else
    if custom_models then
        models                                      ▷ Set Custom Models
            ↪ custom_models
    else
        models
            ↪ DecisionTreeRegressor, LinearRegression,
            BayesianRidge, KNeighborsRegressor, LGBMRegressor,
            ElasticNet, LassoLars, Lasso, SGDRegressor,
            BaggingRegressor, ExtraTreesRegressor, Ridge,
            ARDRegression, RANSACRegressor, LassoLarsIC
```

Algorithm 3: Model Selection

3.3 Optimization

It was found that on any given dataset only a few models were finally chosen in each layer, thereby the execution time can be reduced if we reduce the number of available training models. This was done as follows:

1. Randomly chose n number of models for each iteration, to generate the model list.

2. Chose the n_1 number of models with least errors in each layer. Then repeat this over n_2 layers, and chose n_3 models to generate the final model list.

Input: n_random_models (int, defaults to 0): Number of random samples to be used for model selection
Input: n_models (int, defaults to 5): Number of models models to be used in each layer
Input: n_iter_models (int, defaults to 5): Number of iterations before selecting the final model list
Input: model_list: The default model list
Output: Model List

```
for i in range( global n_estimator) do
    if n_random_models > 0 then
        models
            ↪ random_sample(model_list, n_models = n_random_models)
    else
        if n_iter_models > −1 then
            models.append (n_modelswithleasterrors)
        else
            model_histogram = get frequency of each model in models
            model_lst
                ↪ n_models with greatest frequency in model_histogram
return model_lst
```

Algorithm 4: Optimizing model selection

4 Results and Interpretation

4.1 Testing

VGBoost was then tested on the following datasets[3]:

1. `X, y` ← `make_regression (n_samples = 20000, noise = 0.2)` [7]
2. `X, y` ← `make_friedman1(n_samples = 20000, noise = 0.2)` [7]
3. `X, y` ← `make_friedman2(n_samples = 20000, noise = 0.2)` [7]
4. `X, y` ← `make_friedma3(n_samples = 20000, noise = 0.2)` [7]
5. `X, y` ← `make_classification(n_samples = 20000, flip_y = 0.1)` [7]
6. `X, y` ← `fetch_california_housing(as_frame = True)["data"]` [7]

The results of the above datasets are as follow [4]:

[3] The noise factors were added in order to simulate real world data
[4] Results can be found in (https : //bit.ly/3FuZPIf) and (https : //bit.ly/3DmuCoa)

Model	Validation MSE	Training MSE	Time
VGBRegressor (light=True)	0.0397	.0403	12min 43s
VGBRegressor (light=False)	**0.0394**	0.0405	12min 17s
VGBRegressor (complexity=True)	0.0396	0.0403	1h 4min 17s
XGBRegressor	1347.7776	269.4160	11.8 s
GradientBoostingRegressor	1352.8389	974.4383	1min 40s
ExtraTreesRegressor	3116.4577	6.4210×10^{-26}	1min
RandomForestRegressor	3987.9605	576.5084	2min 51s
BaggingRegressor	4973.3768	1006.1820	17.1 s
AdaBoostRegressor	5888.4974	5522.6421	32.3 s
DecisionTreeRegressor	11298.4037	0.00	**2.39 s**

Table 1: make_regression Results

Model	Validation MSE	Training MSE	Time
VGBRegressor (light=True)	**0.0419**	0.0341	33.4 s
VGBRegressor (light=False)	0.0422	0.0343	33.5 s
VGBRegressor (complexity=True)	0.0420	0.0365	8min 35s
XGBRegressor	0.0439	0.0249	734 ms
GradientBoostingRegressor	0.0421	0.0384	2.14 s
RandomForestRegressor	0.0443	0.0060	6.75 s
ExtraTreesRegressor	3987.9605	3.2095×10^{-30}	3.45 s
BaggingRegressor	0.0476	0.0082	727 ms
AdaBoostRegressor	0.0472	0.0447	1.02 s
DecisionTreeRegressor	0.0866	0.00	**94 ms**

Table 2: make_friedman3 Results

Model	Validation MSE	Training MSE	Time
VGBRegressor (light=True)	**9.0275**	2.3473	51 s
VGBRegressor (light=False)	9.6481	2.4882	55.4 s
VGBRegressor (complexity=True)	10.2357	2.5552	10min 31s
XGBRegressor	62.0922	28.2265	732 ms
GradientBoostingRegressor	198.4483	176.8633	2.05 s
RandomForestRegressor	19.9297	2.9441	5 s
ExtraTreesRegressor	7.4181	7.1130×10^{-25}	3.31 s
BaggingRegressor	31.8670	7.9816	535 ms
AdaBoostRegressor	3994.4126	4012.8412	1.09 s
DecisionTreeRegressor	104.0727	0.00	**81 ms**

Table 3: make_friedman2 Results

Model	Validation MSE	Training MSE	Time
VGBRegressor (light=True)	0.2359	0.1636	56.6 s
VGBRegressor (light=False)	0.2325	0.1642	1min 3s
VGBRegressor (complexity=True)	**0.0755**	0.0696	12min 38s
XGBRegressor	0.3678	0.1132	1.25 s
GradientBoostingRegressor	0.6261	0.5429	4.87 s
RandomForestRegressor	0.9364	0.1385	11.3 s
ExtraTreesRegressor	0.6838	4.4569×10^{-28}	5.06 s
BaggingRegressor	1.1862	0.2404	1.17 s
AdaBoostRegressor	4.7566	4.6560	2.08 s
DecisionTreeRegressor	2.7734	0.00	**81 ms**

Table 4: make_friedman1 Results

Model	Validation f1_score	Training f1_score	Time
VGBClassifier (light=True)	0.8471	0.8882	1min 42s
VGBClassifier (light=False)	0.8466	0.9625	2min 23s
VGBClassifier (complexity=True)	**0.8499**	0.8564	38min 22s
XGBClassifier	0.8414	0.9669	1.94 s
GaussianNB	0.7887	0.7834	**75 ms**
GradientBoostingClassifier	0.8503	0.8575	11.4 s
RandomForestClassifier	0.8482	1.0	7.66 s
ExtraTreesClassifier	0.8495	1.0	2.49 s
BaggingClassifier	0.8290	0.9883	3.74 s
AdaBoostClassifier	0.8456	0.8479	3.23 s
DecisionTreeClassifier	0.7756	1.0	547 ms

Table 5: make_classification Results

Model	Validation MSE	Training MSE	Time
VGBRegressor (light=False)	0.7250	0.5750	1min 23s
VGBRegressor (complexity=True)	0.7890	0.5974	18min 16s
XGBRegressor	0.7670	0.5845	1.47 s

Table 6: California Housing Results

4.2 Optimization

This section containes the results obtained on training the models on:

1. $\texttt{X}, \texttt{y} \leftarrow \texttt{make\_friedman1}(\texttt{n\_samples} = 10000, \texttt{noise} = 0.2)$ [7]
2. $\texttt{X}, \texttt{y} \leftarrow \texttt{make\_regression}(\texttt{n\_samples} = 10000, \texttt{noise} = 0.2)$ [7]
3. $\texttt{X}, \texttt{y} \leftarrow \texttt{make\_classification}(\texttt{n\_samples} = 10000, \texttt{flip\_y} = 0.1)$ [7]

Model	Validation MSE	Training MSE	Time
XGBRegressor	0.5006	0.0865	1.1 s
VGBRegressor(light=True)	0.3496	0.1964	58.7 s
VGBRegressor(light=False)	0.3367	0.1796	1min 5s
VGBRegressor(complexity=True)	0.1944	0.1982	6min 10s
VGBRegressor(light=True, freeze_models=True)	0.3567	0.2049	58.9 s
VGBRegressor(light=True, n_random_models=5)	0.3450	0.1923	49.1 s
VGBRegressor(light=False, freeze_models=True)	0.3573	0.1958	53.9 s
VGBRegressor(light=False, n_random_models=5)	0.3492	0.2036	32.7 s
VGBRegressor(complexity=True, freeze_models=True)	0.2580	0.2891	3min 38s
VGBRegressor(complexity=True, n_random_models=5)	0.2083	0.2278	1min 31s

Table 7: make_friedman1 optimized results

Model	Validation MSE	Training MSE	Time
XGBRegressor	3877	279	6.88 s
VGBRegressor(light=True)	0.04037	0.04003	3min 26s
VGBRegressor(light=False)	0.04045	0.03988	3min
VGBRegressor(complexity=True)	0.04084	0.03959	45min 57s
VGBRegressor(light=True, freeze_models=True)	0.04095	0.03958	2min 49s
VGBRegressor(light=True, n_random_models=5)	0.04045	0.03999	2min 39s
VGBRegressor(light=False, freeze_models=True)	0.04016	0.04002	1min 16s
VGBRegressor(light=False, n_random_models=5)	0.04015	0.04004	1min 15s
VGBRegressor(complexity=True, freeze_models=True)	0.04146	0.04028	35min 19s
VGBRegressor(complexity=True, n_random_models=5)	0.04095	0.03961	9min 7s

Table 8: make_regression optimized results

Model	Validation F1	Training F1	Time
XGBClassifier	0.8767	0.9997	1.2 s
VGBClassifier(light=True)	0.8888	0.9313	1min 6s
VGBClassifier(light=False)	0.8868	0.9717	1min 31s
VGBClassifier(complexity=True)	0.8885	0.8993	14min 29s
VGBClassifier(light=True, freeze_models=True)	0.8843	0.9375	1min 5s
VGBClassifier(light=True, n_random_models=5)	0.8820	0.9348	1min 11s
VGBClassifier(light=False, freeze_models=True)	0.8849	0.9758	1min 6s
VGBClassifier(light=False, n_random_models=5)	0.8875	0.04004	1min 15s
VGBClassifier(complexity=True, freeze_models=True)	0.8810	0.9326	4min 35s
VGBClassifier(complexity=True, n_random_models=5)	0.8862	0.9351	4min 4s

Table 9: make_classification optimized results

4.3 Rationale

Thereby, it can be claimed that, using a Variational Gradient Boosting can yield better metrics [5] than GradientBoost and XGBoost. This is because VGBoost

[5] Refer footnote 1

simultaneously trains many models 3 which individually have a better metrics [6] than regular Decision Trees.

Hyper-parameters — based on the above results it can be claimed that having a greater complexity does not necessarily provide better metrics [7]. Although, setting `complexity` = `True` did give better results in friedman1 4, as which yields a more complex dataset than the rest. Further by `light` = `True` and `light` = `False` also had similar results, thereby one may also consider the training time of the algorithms.

Model Selection Optimization — on the basis of the optimized results it is claimed that using less models as defined in 3.3, the execution time of the models can be reduced. Further this is theoretically proven as if less models are trained in the first place, then it would take less execution time.

Drawbacks and further prospects — Compared to GradientBoost and XGBoost, VGBoost has a greater time and memory complexity, as observed in testing 4.1. Thereby, hardware level optimization and low-level programming language implementation, for example in PyPy or Cython. Maybe used in order to optimize VGBoost.

References

[1] *Avoiding Overfitting of Decision Trees*, pages 119–134. Springer London, London, 2007.

[2] Asma Faris Amro, Mousa T. Al-Akhras, Khalil M. El Hindi, Mohamed Habib, and Bayan Abu Shawar. Instance reduction for avoiding overfitting in decision trees. *Journal of Intelligent Systems*, 30:438 – 459, 2021.

[3] IBM. What is a decision tree — ibm, 2022.

[4] IBM. What is boosting — ibm, 2022.

[5] Rafael García Leiva, Antonio Fernández Anta, Vincenzo Mancuso, and Paolo Casari. A novel hyperparameter-free approach to decision tree construction that avoids overfitting by design. *IEEE Access*, 7:99978–99987, 2019.

[6] Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in Neurorobotics*, 7, 2013.

[6] Refer footnote 1

[7] Refer footnote 1

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Alexander Vezhnevets and Olga Barinova. Avoiding boosting overfitting by removing confusing samples. In *ECML*, 2007.

[9] D.R. Wilson and Tony R. Martinez. The need for small learning rates on large problems. *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, 1:115–119 vol.1, 2001.

[10] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168, 2019.