Boosting with AdaBoost and Gradient Boosting

Diogo Menezes Borges Follow

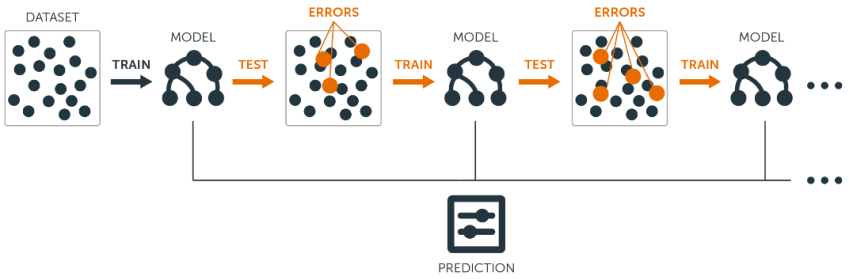
Sep 21, 2018 · 6 min read ★

Have you ever been or seen a Kaggle competition? Most of the prize winners do it by using boosting algorithms. Why is AdaBoost, GBM, and XGBoost the go-to algorithm of champions?

First of all, if you never heard of Ensemble Learning or Boosting check out my post *"Ensemble Learning: When everybody takes a guess…I guess!"* so you can understand better these algorithms.

> *More informed? Good, let's start!*

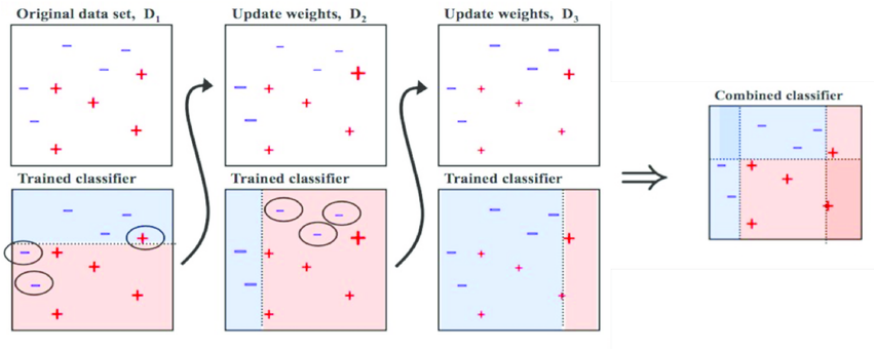
So, the idea of Boosting just as well any other ensemble algorithm is to combine several weak learners into a stronger one. The general idea of Boosting algorithms is to try predictors sequentially, where each subsequent model attempts to fix the errors of its predecessor.



Source

Adaptive Boosting

Adaptive Boosting, or most commonly known AdaBoost, is a Boosting algorithm. **Shocker!** The method this algorithm uses to correct its predecessor is by paying more attention to underfitted training instances by the previous model. Hence, at every new predictor the focus will be, each time, on the harder cases.



Source

Let’ts take the example of the image. To build a AdaBoost classifier, imagine that as a first base classifier we train a Decision Tree algorithm to make predictions on our training data. Now, following the methodology of AdaBoost, the weight of the misclassified training instances is increased. The second classifier is trained and acknowledges the updated weights and it repeats the procedure over and over again.

> *At the end of every model prediction we end up boosting the weights of the misclassified*

> *instances so that the next model does a better job on them, and so on.*

This sequential learning technique sounds a bit like Gradient Descent, except that instead of tweaking a single predictor's parameter to minimise the cost function, AdaBoost adds predictors to the ensemble gradually making it better. The great disadvantage of this algorithm is that the model cannot be parallelized since each predictor can only be trained after the previous one has been trained and evaluated.

Below are the steps for performing the AdaBoost algorithm:

1. Initially, all observations are given equal weights.
2. A model is built on a subset of data.
3. Using this model, predictions are made on the whole dataset.
4. Errors are calculated by comparing the predictions and actual values.
5. While creating the next model, higher weights are given to the data points which were predicted incorrectly.
6. Weights can be determined using the error value. For instance,the higher the error the more is the weight assigned to the observation.
7. This process is repeated until the error function does not change, or the maximum limit of the number of estimators is reached.

Hyperparameters

`base_estimators` : specifies the base type estimator, i.e. the algorithm to be used as base learner.

`n_estimators` **:** It defines the number of base estimators, where the default is 10 but you can increase it in order to obtain a better performance.

`learning_rate` **:** same impact as in gradient descent algorithm

`max_depth` **:** Maximum depth of the individual estimator

`n_jobs` : indicates the system how many processors it is allowed to use. Value of '-1' means there is no limit;

`random_state` : makes the model's output replicable. It will always produce the same results when you give it a fixed value as well as the same parameters and training data.

Gradient Boosting

This is another very popular Boosting algorithm whose work basis is just like what we've seen for AdaBoost. Gradient Boosting works by sequentially adding the previous predictors underfitted predictions to the ensemble, ensuring the erros made previously are corrected.

The difference lies in what it does with the underfitted values of its predecessor. Contrary to AdaBoost, which tweaks the instance weights at every interaction, this method **tries to fit the new predictor to the *residual errors* made by the previous predictor.**

So that you can understand Gradient Boosting it is important to understand Gradient Descent first.

Below are the steps for performing the Gradien Boosting algorithm:

1. A model is built on a subset of data.
2. Using this model, predictions are made on the whole dataset.
3. Errors are calculated by comparing the predictions and actual values.
4. A new model is created using the errors calculated as target variable. Our objective is to find the best split to minimise the error.
5. The predictions made by this new model are combined with the predictions of the previous.
6. New errors are calculated using this predicted value and actual value.

7. This process is repeated until the error function does not change, or the maximum limit of the number of estimators is reached.

Hyperparameters

`min_samples_split:` Minimum number of observation which is required in a node to be considered for splitting. It is used to control overfitting.

`min_samples_leaf` : Minimum samples required in a terminal or leaf node. Lower values should be chosen for imbalanced class problems since the regions in which the minority class will be in the majority will be very small.

`min_weight_fraction_leaf` **:** similar to the previous but defines a fraction of the total number of observations instead of an integer.

`max_depth` : maximum depth of a tree. Used to control overfitting.

`max_lead_nodes` : maximum number of terminal leaves in a tree. If this is defined `max_depth` is ignored.

`max_features` : number of features it should consider while searching for the best split.

XGBoost

Extreme Gradient Boosting is an advanced implementation of the Gradient Boosting. This algorithm has high predictive power and is ten times faster than any other gradient boosting techniques. Moreover, includes a variety of regularisation which reduces overfitting and improves overall performance.

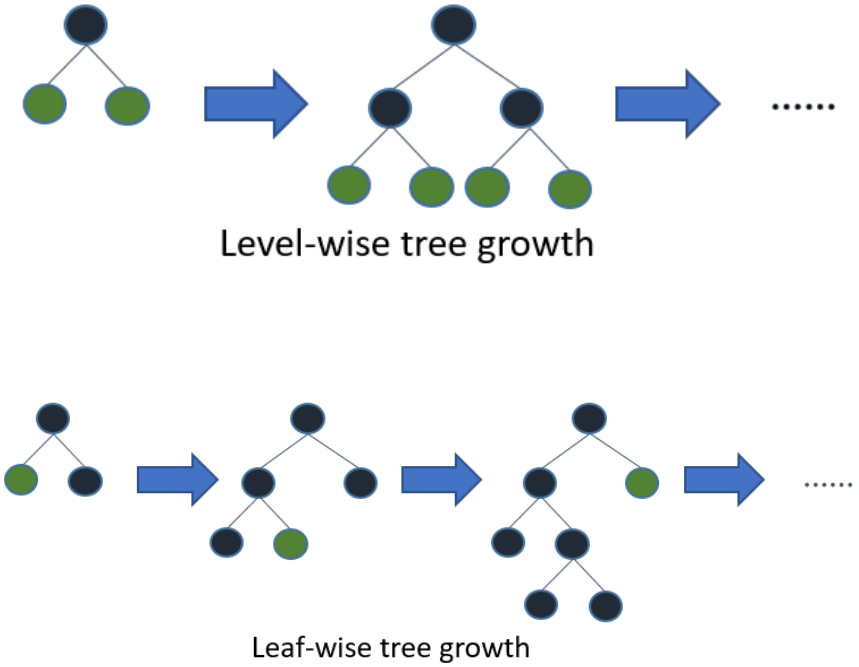
Advantages

- Implements regularisation helping reduce overfit (GB does not have);
- Implements parallel processing being much faster than GB;
- Allows users to define custom optimisation objectives and evaluation criteria adding a whole new dimension to the model;

- XGBoost has an in-built routine to handle missing values;
- XGBoost makes splits up to the `max_depth` specified and then starts pruning the tree backwards and removes splits beyond which there is no positive gain;
- XGBoost allows a user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run.

Light GB

For datasets which are extremely large **Light Gradient Boosting** is the best, compared to all of the other, since it takes less time to run.



Source

This algorithm is based on leaf-wise tree growth contrary to others which work in a level-wise approach pattern. You can see comparision between XGBoost and Light GB here.

If you liked it, follow me for more publications and don't forget, please, give it an applause!

You the mighty reader applauding!

Resources :

- *Hands-on Machine Learning with Scikit-Learn & TensorFlow* by Aurélien Géron, Chapter 7
- Analytics Vidhya, *A Comprehensive Guide to Ensemble Learning (with Python codes)*