Gradient Boosting Classification explained through Python

towardsdatascience.com/gradient-boosting-classification-explained-through-python-60cc980eeb3d

October 7, 2020

Top highlight

Photo by on

In my previous article, I discussed and went through a working python example of Gradient Boosting for Regression. In this article, I would like to discuss how Gradient Boosting works for Classification. If you did not read that article, it's all right because I will reiterate what I discussed in the previous article anyway. So, let's get started!

Ensemble Methods

Usually, you may have a few good predictor, and you would like to use them all, instead of painfully choosing one because it has a 0.0001 accuracy increase. In comes *Ensemble Learning.* In Ensemble Learning, instead of using a single predictor, multiple predictors and training in the data and their results are aggregated, usually giving a better score than using a single model. A Random Forest, for instance, is simply an ensemble of bagged(or pasted) Decision Trees.

Photo by on

You can think of Ensemble Methods as an orchestra; instead of just having one person play an instrument, multiple people play different instruments, and by combining all the musical groups, the music sound generally better than it would if it was played by a single person.

While Gradient Boosting is an Ensemble Learning method, it is more specifically a *Boosting* Technique. So, what's Boosting?

Boosting

Boosting is a special type of Ensemble Learning technique that works by combining several *weak learners(*predictors with poor accuracy*)* into a strong learner(a model with strong accuracy). This works by each model paying attention to its predecessor's mistakes.

The two most popular boosting methods are:

- Adaptive Boosting(you can read my article about it)
- Gradient Boosting

We will be discussing Gradient Boosting.

Gradient Boosting

In Gradient Boosting, each predictor tries to improve on its predecessor by reducing the errors. But the fascinating idea behind Gradient Boosting is that instead of fitting a predictor on the data at each iteration, it actually fits a new predictor t*o the residual errors made by the previous predictor.* Let's gothrough a step by step example of how Gradient Boosting Classification Works:

Photo by Lindsay Henwood on Unsplash

1. In order to make initial predictions on the data, the algorithm will get the *log of the odds* of the target feature. This is usually the number of True values(values equal to 1) divided by the number of False values(values equal to 0).

So, if we had a breast cancer dataset of 6 instances, with 4 examples of people who have breast cancer(4 target values = 1) and 2 examples of people who do not have breast cancer(2 target values = 0), then the log(odds) = log(4/2) ~ 0.7. This is our base estimator.

1. Once it has the log(odds), we convert that value to a probability by using a logistic function in order to make predictions. If we are continuing with our previous example of a log(odds) value of 0.7, then the logistic function would equate to around 0.7 too.

Since this value is *greater than 0.5,* the algorithm will predict 0.7 for every instance as its base estimation. The formula for converting the log(odds) into a probability is the following:

```
e * log(odds) / (1 + e * log(odds))
```

1. For every instance in the training set, it calculates the *residuals* for that instance, or, in other words, the observed value minus the predicted value.

2. Once it has done this, it build a new Decision Tree that actually tries to predict the residuals that was previously calculated. However, this is where it gets slightly tricky in comparison with Gradient Boosting Regression.

When building a Decision Tree, there is a set number of leaves allowed. This can be set as a parameter by a user, and it is usually between 8 and 32. This leads to two of the possible outcomes:

- Multiple instances fall into the same leaf
- A single instance has its own leaf

Unlike Gradient Boosting for Regression, where we could simply average the instance values to get an output value, and leave the single instance as a leaf of its own, we have to transform these values using a formula:

$$\frac{\sum Residual}{\sum[PreviousProb * (1 - PreviousProb)]}$$

Credit: Blogspace

The Σ sign means “sum of”, and *PreviousProb* refers to our previously calculated probability(in our example, being 0.7). We apply this transformation for every leaf in the tree. Why do we do this? Because remember our base estimator is a log(odds), and our tree was actually built on a probability, so we cannot simply add them because they come from two different sources.

Making Predictions

Now, to make new predictions, we do 2 things:

1. get the log(odds) prediction for each instance in the training set
2. convert that prediction into a probability

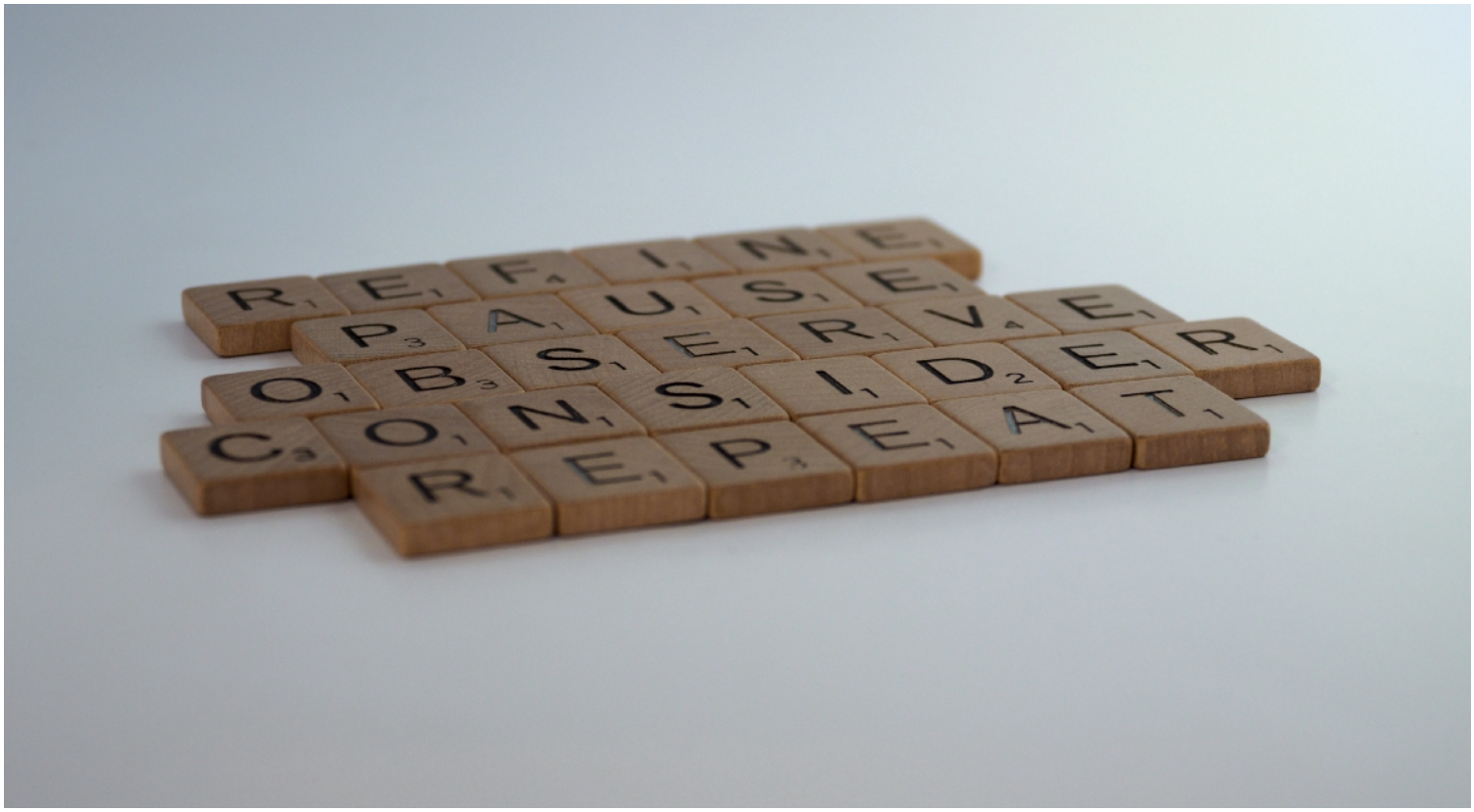
For each instance in the training set, the formula for making predictions would be the following:

```
base_log_odds + (learning_rate * predicted residual value)
```

The *learning_rate* is a hyperparameter that is used to scale each trees contribution, sacrificing bias for better variance. In other words, we multiply this number by the predicted value so that we do not overfit the data.

Once we have calculated the log(odds) prediction, we now must convert it into a probability using the previous formula for converting log(odds) values into probabilities.

Repetition & making predictions on unseen data


REFINE
PAUSE
OBSERVE
CONSIDER
REPEAT

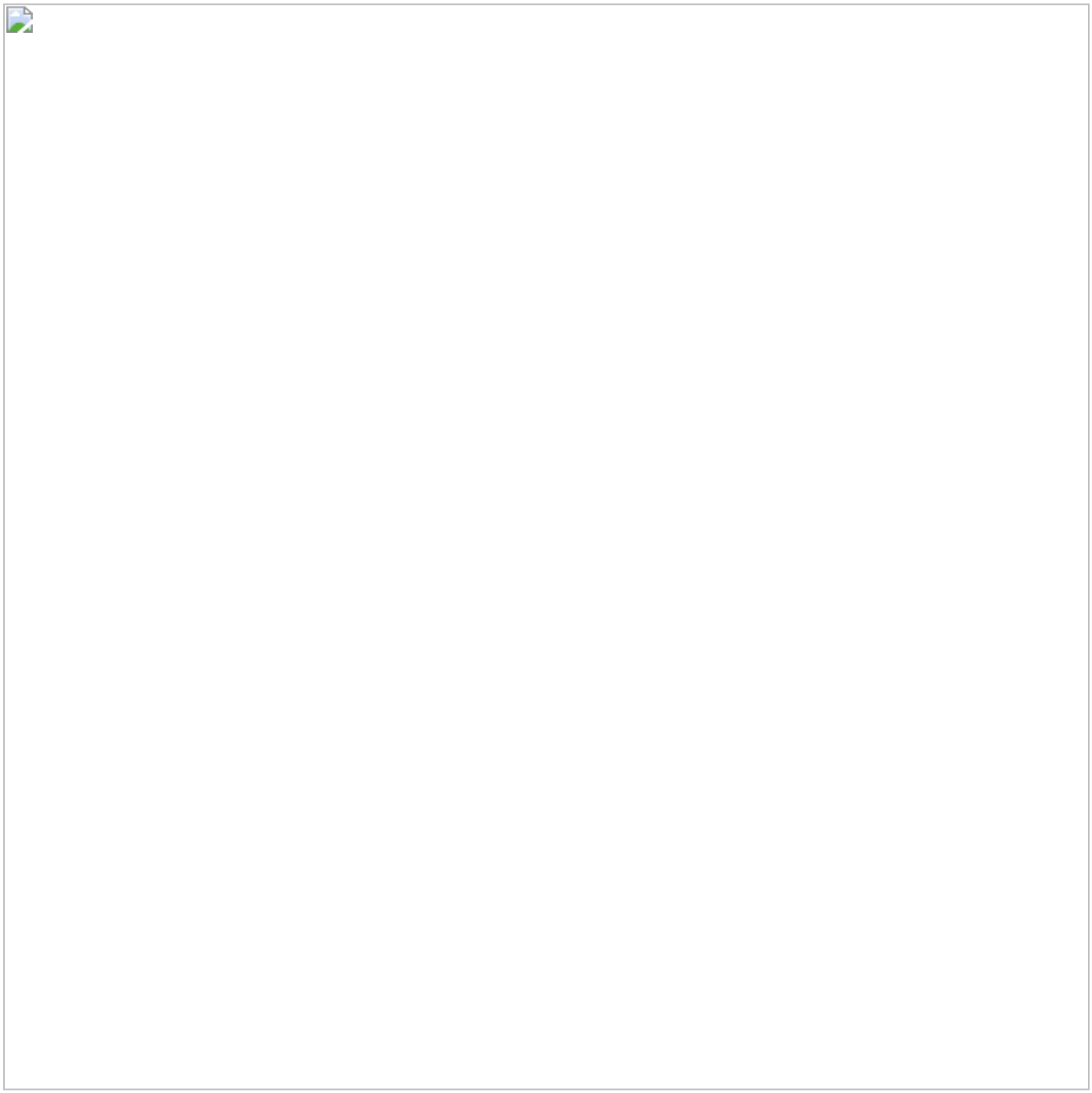


Photo by Brett Jordan on Unsplash

After we have have done this process, we calculate the new residuals of the tree and create a new tree to fit the new residuals. Again, the process is repeated until a certain predefined threshold is reached, or the residuals are negligible.

If we had training 6 trees, and we wanted to make a new prediction on an unseen instance, the pseudo-code for that would be:

```
X_test_prediction = base_log_odds + (learning_rate * tree1_scaled_output_value) +
(learning_rate * tree2_scaled_output_value) +
(learning_rate * tree3_scaled_output_value) +
(learning_rate * tree4_scaled_output_value) +
(learning_rate * tree5_scaled_output_value) +
(learning_rate * tree6_scaled_output_value) +prediction_probability =
e*X_test_prediction / (1 + e*X_test_prediction)
```

Ok, so now you should have somewhat of an understanding of the underlying mechanics of Gradient Boosting for Classification, let’s begin coding to cement that knowledge!

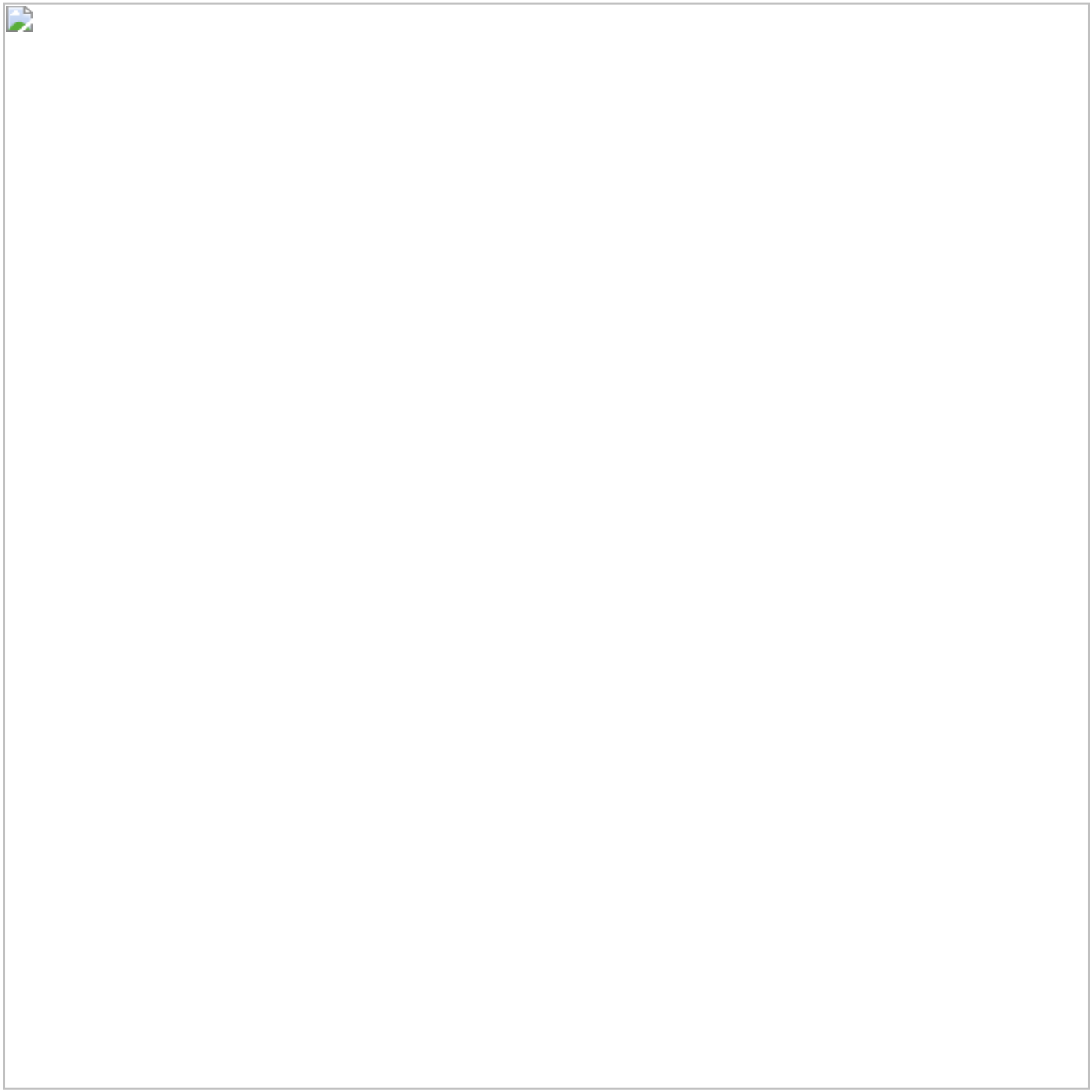


Photo by Rob Sarmiento on Unsplash

Gradient Boosting Classification with Scikit-Learn

We will be using the breast cancer dataset that is prebuilt into scikit-learn to use as example data. First off, let's get some imports out of the way:

```
import pandas as pd
import numpy as npfrom sklearn.metrics import classification_report
from sklearn.model_selection import KFold
from sklearn.datasets import load_breast_cancerfrom sklearn.ensemble import
GradientBoostingClassifier
```

Here, we are just importing pandas, numpy, our model, and a metric to evaluate our model's performance.

```
df = pd.DataFrame(load_breast_cancer()['data'],
columns=load_breast_cancer()['feature_names'])df['y'] = load_breast_cancer()['target']
df.head(5)
```

For convenience sake, we will convert the data into a DataFrame because it's easier to manipulate that way. Feel free to skip this step.

```
X,y = df.drop('y',axis=1),df.ykf = KFold(n_splits=5,random_state=42,shuffle=True)for
train_index,val_index in kf.split(X):
    X_train,X_val = X.iloc[train_index],X.iloc[val_index],
    y_train,y_val = y.iloc[train_index],y.iloc[val_index],
```

Here, we define our features and our label, and split ur data into a train and validation using 5 Fold cross validation.

```
gradient_booster = GradientBoostingClassifier(learning_rate=0.1)
gradient_booster.get_params()OUT:{'ccp_alpha': 0.0,
 'criterion': 'friedman_mse',
 'init': None,
 'learning_rate': 0.1,
 'loss': 'deviance',
 'max_depth': 3,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_iter_no_change': None,
 'presort': 'deprecated',
 'random_state': None,
 'subsample': 1.0,
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

There is a lot of parameters here, so I will only cover the most important:

- **Criterion**: The loss function used to find the optimal feature and threshold to split the data
- **learning_rate**: this parameter scales the contribution of each tree
- **max_depth**: the maximum depth of each tree
- **n_estimators**: the number of trees to construct

- **init: the initial estimator.** By default, it is the log(odds) converted to a probability(like we discussed before)

```
gradient_booster.fit(X_train,y_train)print(classification_report(y_val,gradient_booster.predict(X_
    recall  f1-score   support0       0.98      0.93      0.96        46
1       0.96      0.99      0.97        67    accuracy                           0.96       113
   macro avg       0.97      0.96      0.96       113
weighted avg       0.96      0.96      0.96       113
```

All right, 96% accuracy!

I hope this article has helped you understand Gradient Boosting Classification(in some shape or form). I wish you all the best in your ML endeavours and remember; One who knows few things but knows them well is better than one who knows many things but knows them all poorly!

Photo by Kelly Sikkema on Unsplash