A Beginner's guide to XGBoost

This article will have trees.... lots of trees

George Seif May 29, 2019 · 6 min read ★

Trees... lots of them

> *I recently started a book-focused educational newsletter. Book Dives is a bi-weekly newsletter where for each new issue we dive into a non-fiction book. You’ll learn about the book’s core lessons and how to apply them in real life. You can* ***subscribe for it here****.*

XGBoost is an open source library providing a high-performance implementation of gradient boosted decision trees. An underlying C++ codebase combined with a Python interface sitting on top makes for an extremely powerful yet easy to implement package.

The performance of XGBoost is no joke — it’s become the go-to library for winning many Kaggle competitions. Its gradient boosting implementation is second to none and there’s only more to come as the library continues to garner praise.

In this post we’re going to go through the basics of the XGBoost library. We’ll start with a practical explanation of how gradient boosting actually works and then go through a Python example of how XGBoost makes it oh-so quick and easy to do it.

Boosting Trees

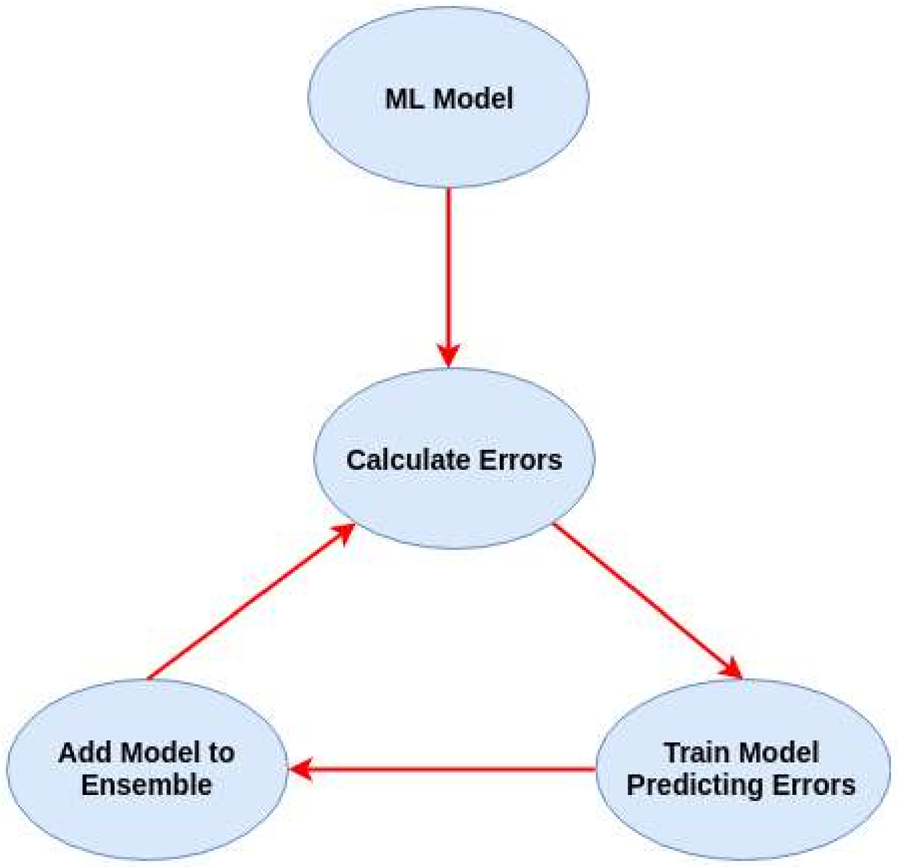
With a regular machine learning model, like a decision tree, we’d simply train a single model on our dataset and use that for prediction. We might play around with the parameters for a bit or augment the data, but in the end we are still using a single model. Even if we build an ensemble, all of the models are trained and applied to our data separately.

Boosting, on the other hand, takes a more *iterative* approach. It’s still technically an ensemble technique in that many models are combined together to perform the final one, but takes a more clever approach.

Rather than training all of the models in isolation of one another, boosting trains models in succession, with each new model being trained to correct the errors made by the previous ones. Models are added sequentially until no further improvements can be made.

The advantage of this iterative approach is that the new models being added are focused on correcting the mistakes which were caused by other models. In a standard ensemble method where models are trained in isolation, all of the models might simply end up making the same mistakes!

Gradient Boosting specifically is an approach where new models are trained to predict the residuals (i.e errors) of prior models. I've outlined the approach in the diagram below.



Getting started with XGBoost

Let's start using this beast of a library — XGBoost.

The first thing we want to do is install the library which is most easily done via pip. It can also be safer to do this in a Python virtual environment.

```
pip install xgboost
```

Setting up our data with XGBoost

For the rest of our tutorial we're going to be using the iris flowers dataset. We can use Scikit Learn to get that loaded up in Python. At the same time, we'll also import our newly installed XGBoost library.

```
from sklearn import datasets
import xgboost as xgb

iris = datasets.load_iris()
X = iris.data
y = iris.target
```

Let's get all of our data set up. We'll start off by creating a train-test split so we can see just how well XGBoost performs. We'll go with an 80%-20% split this time.

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, y,
test_size=0.2)
```

In order for XGBoost to be able to use our data, we'll need to transform it into a specific format that XGBoost can handle. That format is called **DMatrix**. It's a very simple one-linear to transform a numpy array of data to DMatrix format:

```
D_train = xgb.DMatrix(X_train, label=Y_train)
D_test = xgb.DMatrix(X_test, label=Y_test)
```

Defining an XGBoost model

Now that our data is all loaded up, we can define the parameters of our gradient boosting ensemble. We've set up some of the most important ones below to get us started. For more complicated tasks and models, the full list of possible parameters is available on the official XGBoost website.

```
param = {
    'eta': 0.3,
    'max_depth': 3,
    'objective': 'multi:softprob',
    'num_class': 3}

steps = 20  # The number of training iterations
```

The simplest parameters are the *max_depth* (maximum depth of the decision trees being trained), *objective* (the loss function being used), and *num_class* (the number of classes in the dataset). The *eta* algorithm requires special attention.

From our theory, Gradient Boosting involves creating and adding decision trees to an ensemble model sequentially. New trees are created to correct the residual errors in the predictions from the existing ensemble.

Due to the nature of an ensemble, i.e having several models put together to form what is essentially a very large complicated one, makes this technique prone to overfitting. The **eta** parameter gives us a chance to prevent this overfitting

The eta can be thought of more intuitively as a *learning rate*. Rather than simply adding the predictions of new trees to the ensemble with full weight, the eta will be multiplied by the residuals being adding to reduce their weight. This effectively reduces the complexity of the overall model.

It is common to have small values in the range of 0.1 to 0.3. The smaller weighting of these residuals will still help us train a powerful model, but won't let that model run away into deep complexity where overfitting is more likely to happen.

Training and Testing

We can finally train our model similar to how we do so with Scikit Learn:

```
model = xgb.train(param, D_train, steps)
```

Let's now run an evaluation. Again the process is very similar to that of training models in Scikit Learn:

```
import numpy as np
from sklearn.metrics import precision_score, recall_score, 
accuracy_score

preds = model.predict(D_test)
best_preds = np.asarray([np.argmax(line) for line in preds])

print("Precision = {}".format(precision_score(Y_test, best_preds,
average='macro')))
print("Recall = {}".format(recall_score(Y_test, best_preds,
average='macro')))
print("Accuracy = {}".format(accuracy_score(Y_test, best_preds)))
```

Awesome!

If you've followed all the steps up to this point, you should get at least 90% accuracy!

Further Exploration with XGBoost

That just about sums up the basics of XGBoost. But there are some more cool features that'll help you get the most out of your models.

- The **gamma** parameter can also help with controlling overfitting. It specifies the minimum reduction in the loss required to make a further partition on a leaf node of the tree. I.e if creating a new node doesn't reduce the loss by a certain amount, then we won't create it at all.
- The **booster** parameter allows you to set the type of model you will use when building the ensemble. The default is *gbtree* which builds an ensemble of decision trees. If your data isn't too complicated, you can go with the faster and simpler *gblinear* option which builds an ensemble of linear models.
- Setting the optimal hyperparameters of any ML model can be a challenge. So why not let Scikit Learn do it for you? We can combine Scikit Learn's grid search with an XGBoost classifier quite easily:

```
from sklearn.model_selection import GridSearchCV

clf = xgb.XGBClassifier()
parameters = {
     "eta"    : [0.05, 0.10, 0.15, 0.20, 0.25, 0.30 ] ,
     "max_depth"        : [ 3, 4, 5, 6, 8, 10, 12, 15],
     "min_child_weight" : [ 1, 3, 5, 7 ],
     "gamma"            : [ 0.0, 0.1, 0.2 , 0.3, 0.4 ],
     "colsample_bytree" : [ 0.3, 0.4, 0.5 , 0.7 ]
     }

grid = GridSearchCV(clf,
                    parameters, n_jobs=4,
                    scoring="neg_log_loss",
                        cv=3)

grid.fit(X_train, Y_train)
```

Only do that on a big dataset if you have time to kill — doing a grid search is essentially training an ensemble of decision trees many times over!

- Once your XGBoost model is trained, you can dump a human readable description of it into a text file:

```
model.dump_model('dump.raw.txt')
```

That's a wrap!

Like to learn?

Follow me on twitter where I post all about the latest and greatest AI, Technology, and Science! Connect with me on LinkedIn too!

Machine Learning Data Science Artificial Intelligence Technology Education