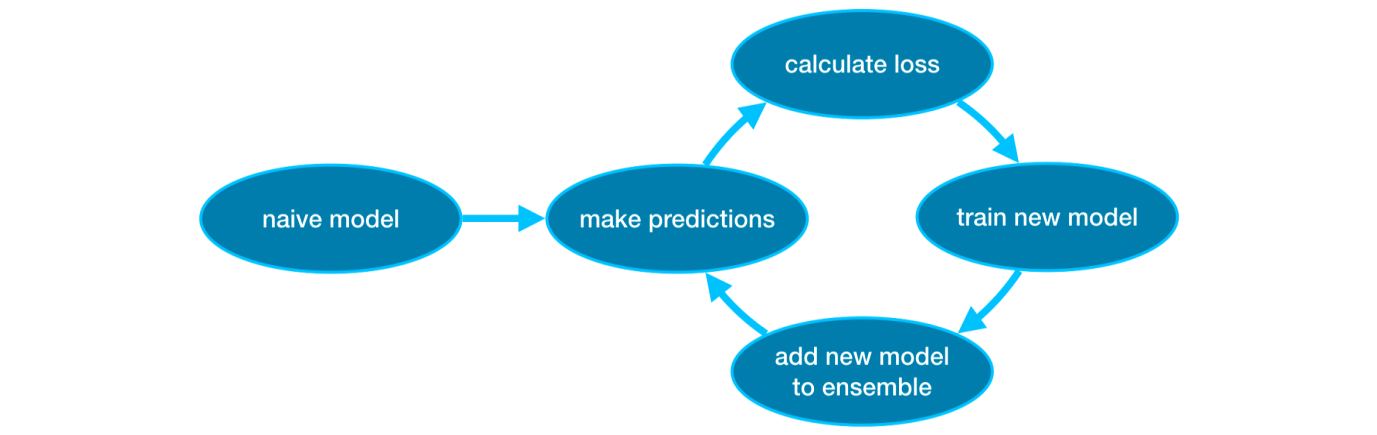
Intermediate Machine Learning: 6th lesson – XGBoost

Gradient boosting:

A method that goes through cycles to iteratively add models into an ensemble. It begins by initializing the ensemble with a single model, whose predictions can be pretty naïve (even if its predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors). Then, the user starts the cycle:

* First, use the current ensemble to generate predictions for each observation in the dataset. To make a prediction, add the predictions from all models in the ensemble.
* These predictions are used to calculate a loss function (like mean squared error, for instance).
* Then, use the loss function to fit a new model that will be added to the ensemble. Specifically, determine model parameters so that adding this new model to the ensemble will reduce the loss (the "gradient" in "gradient boosting" refers to the fact that will use gradient descent on the loss function to determine the parameters in this new model).
* Finally, we add the new model to ensemble, and ... repeat!



XGBoost (eXtreme Gradient Boosting):

* An open-source software library which provides a regularizing gradient boosting for framework including C++, Java, Python, R, Julia, Practical Extraction and Reporting Language (Perl), and Scala.
* Compatible with Linux, Microsoft Windows, and Macintosh Operating System (macOS).
* Based on the project description, it aims to provide a “Scalable, Portable and Distributed Gradient Boosting (GBM, GBRT, GBDT) Library”.
* Runs on a single machine, as well as the distributed processing frameworks like Apache Hadoop, Apache Spark, Apache Flink, and Dask.
* Initially, it began as a terminal application which could be configured using a LIBSVM configuration file, where became well-known in the Machine Learning competition circles after its use in the winning solution of the Higgs Machine Learning Challenge.
* Features:
* Clever penalization of trees
* Proportional shrinking of leaf nodes
* Newton Boosting
* Extra randomization parameter
* Implementation on single, distributed systems and out-of-core computation
* Automatic feature selection
* Theoretically justified, weighted quantile sketching for efficient computation
* Parallel tree structure boosting with sparsity
* Efficient cacheable block structure for decision tree training

Parameter tuning:

XGBoost has a few parameters that can dramatically affect accuracy and training speed. The first parameters were include:

* n\_estimators

n\_estimators specify how many times to go through the modeling cycle. It is equal to the number of models included in the ensemble.

* Lowest value causes ***underfitting***, which leads to inaccurate predictions on both training data and test data.
* Highest value causes ***overfitting***, which causes accurate predictions on training data, but inaccurate predictions on test data.
* early\_stopping\_rounds
* early\_stopping\_rounds offers a way to automatically find the ideal value for n\_estimators. Early stopping causes the model to stop iterating when the validation score stops improving, even if we aren't at the hard stop for n\_estimators. It's smart to set a high value for n\_estimators and then use early\_stopping\_rounds to find the optimal time to stop iterating.
* Since random chance sometimes causes a single round where validation scores do not improve, the user needs to specify a number for how many rounds of straight deterioration to allow before stopping. Setting early\_stopping\_rounds=5 is a reasonable choice. In this case, the operation stopped after 5 straight rounds of deteriorating validation scores.
* learning\_rate

Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number (known as the ***learning rate***) before adding them in. This means each tree we add to the ensemble helps us less. So, we can set a higher value for n\_estimators without overfitting. If we use early stopping, the appropriate number of trees will be determined automatically.

* n\_jobs

On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter n\_jobs equal to the number of cores on your machine. On smaller datasets, this won't help. The resulting model won't be any better, so micro-optimizing for fitting time is typically nothing but a distraction. But, it's useful in large datasets where you would otherwise spend a long time waiting during the fit command.