Navigate to a section:

Introduction to Gradient Boosting

The general idea behind gradient boosting is combining weak learners to produce a more accurate model. These “weak learners” are essentially decision trees, and gradient boosting aims to combine multiple decision trees to lower the model error somehow.

The term “boosting” was introduced the first time successfully in **AdaBoost** (Adaptive Boosting). This algorithm combines multiple single split decision trees. AdaBoost puts more emphasis on observations that are more difficult to classify by adding new weak learners where needed.

In a nutshell, gradient boosting is comprised of only three elements:

- **Weak Learners** – simple decision trees that are constructed based on purity scores (e.g., *Gini*).
- **Loss Function** – a differentiable function you want to minimize. In regression, this could be a *mean squared error*, and in classification, it could be *log loss*.
- **Additive Models** – additional trees are added where needed, and a functional gradient descent procedure is used to minimize the loss when adding trees.

You now know the basics of gradient boosting. The following section will introduce the most popular gradient boosting algorithm – XGBoost.

Introduction to XGBoost

XGBoost stands for *eXtreme Gradient Boosting* and represents the algorithm that wins most of the Kaggle competitions. It is an algorithm specifically designed to implement state-of-the-art results fast.

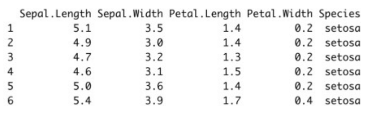
XGBoost is used both in regression and classification as a go-to algorithm. As the name suggests, it utilizes the *gradient boosting* technique to accomplish enviable results – by adding more and more weak learners until no further improvement can be made.

Today you’ll learn how to use the XGBoost algorithm with R by modeling one of the most trivial datasets – the Iris dataset – starting from the next section.

Dataset Loading and Preparation

As mentioned earlier, the Iris dataset will be used to demonstrate how the XGBoost algorithm works. Let’s start simple with a necessary first step – library and dataset imports. You’ll need only a few, and the dataset is built into R:

Here’s how the first couple of rows look like:



```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Image 1 – The first six rows of the Iris dataset

There’s no point in further exploration of the dataset, as anyone in the world of data already knows everything about it.

The next step is dataset splitting into training and testing subsets. The following code snippet splits the dataset in a 70:30 ratio and then further splits the dataset in features (X) and target (y) for both subsets. This step is necessary for the training process:

You now have everything needed to start with the training process. Let’s do that in the next section.

Modeling

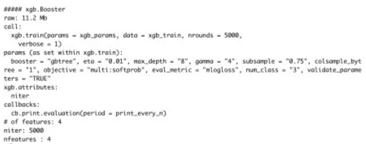
XGBoost uses something knows as a DMatrix to store data. DMatrix is nothing but a specific data structure used to store data in a way optimized for both memory efficiency and training speed.

Besides the DMatrix, you’ll also have to specify the parameters for the XGBoost model. You can learn more about all the available parameters here, but we’ll stick to a subset of the most basic ones.

The following snippet shows you how to construct DMatrix data structures for both training and testing subsets and how to build a list of parameters:

Now you have everything needed to build a model. Here’s how:

The results of calling `xgb_model` are displayed below:



```
##### xgb.Booster
raw: 11.2 Mb
call:
  xgb.train(params = xgb_params, data = xgb_train, nrounds = 5000,
    verbose = 1)
params (as set within xgb.train):
  booster = "gbtree", eta = "0.01", max_depth = "8", gamma = "4", subsample = "0.75", colsample_byt
ree = "1", objective = "multi:softprob", eval_metric = "mlogloss", num_class = "3", validate_parame
ters = "TRUE"
xgb.attributes:
  niter
callbacks:
  cb.print.evaluation(period = print_every_n)
# of features: 4
niter: 5000
nfeatures : 4
```

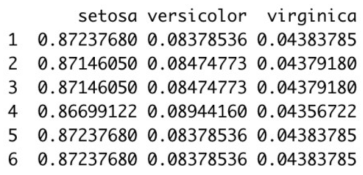
Image 2 – XGBoost model after training

And that’s all you have to do to train your first gradient boosting model! You’ll learn how to evaluate it in the next section.

Predictions and Evaluations

You can use the `predict()` function to make predictions with the XGBoost model, just as with any other model. The next step is to covert the predictions to a data frame and assign column names, as the predictions are returned in the form of probabilities:

Here’s what the above code snippet produces:



```
      setosa versicolor  virginica
1 0.87237680 0.08378536 0.04383785
2 0.87146050 0.08474773 0.04379180
3 0.87146050 0.08474773 0.04379180
4 0.86699122 0.08944160 0.04356722
5 0.87237680 0.08378536 0.04383785
6 0.87237680 0.08378536 0.04383785
```

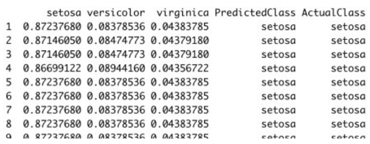
Image 3 – Prediction probabilities for every flower species

As you would imagine, these probabilities add up to 1 for a single row. The column with the highest probability is the flower species predicted by the model.

Still, it would be nice to have two additional columns. The first one represents the predicted class (max of predicted probabilities). The other represents the actual class, so we can estimate how good the model performs on previously unseen data.

The following snippet does just that:

The results are displayed in the following figure:



```
      setosa versicolor  virginica PredictedClass ActualClass
1 0.87237680 0.08378536 0.04383785         setosa      setosa
2 0.87146050 0.08474773 0.04379180         setosa      setosa
3 0.87146050 0.08474773 0.04379180         setosa      setosa
4 0.86699122 0.08944160 0.04356722         setosa      setosa
5 0.87237680 0.08378536 0.04383785         setosa      setosa
6 0.87237680 0.08378536 0.04383785         setosa      setosa
7 0.87237680 0.08378536 0.04383785         setosa      setosa
8 0.87237680 0.08378536 0.04383785         setosa      setosa
9 0.87237680 0.08378536 0.04383785         setosa      setosa
```

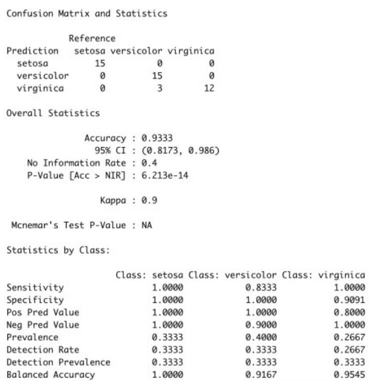
Image 4 – Predicted class vs. actual class on the test set

Things look promising, to say at least, but that’s no reason to jump to conclusions. Next, we can calculate the overall accuracy score as a sum of instances where predicted and actual classes match divided by the total number of rows:

Executing the above code prints out 0.9333 to the console, indicating we have a 93% accurate model on previously unseen data.

While we’re here, we can also print the confusion matrix to see what exactly did the model misclassify:

The results are shown below:



```
Confusion Matrix and Statistics

            Reference
Prediction   setosa versicolor virginica
  setosa         15          0         0
  versicolor      0         15         0
  virginica       0          3        12

Overall Statistics

               Accuracy : 0.9333
                 95% CI : (0.8173, 0.986)
    No Information Rate : 0.4
    P-Value [Acc > NIR] : 6.213e-14

                  Kappa : 0.9

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: setosa Class: versicolor Class: virginica
Sensitivity                 1.0000            0.8333           1.0000
Specificity                 1.0000            1.0000           0.9091
Pos Pred Value              1.0000            1.0000           0.8000
Neg Pred Value              1.0000            0.9000           1.0000
Prevalence                  0.3333            0.4000           0.2667
Detection Rate              0.3333            0.3333           0.2667
Detection Prevalence        0.3333            0.3333           0.3333
Balanced Accuracy           1.0000            0.9167           0.9545
```

Image 5 – Confusion matrix for XGBoost model on the test set

As you can see, only three *virginica* species were classified as *versicolor*. There were no misclassifications in the *setosa* species.

And that’s how you can train and evaluate XGBoost models with R. Let’s wrap things up in the next section.

Conclusion

XGBoost is a complex state-of-the-art algorithm for both classification and regression – thankfully, with a simple R API. Entire books are written on this single algorithm alone, so cramming everything in a single article isn’t possible.