Boosting

D2K Course Staff

4/2/2019

Note:

This module takes a little time to run due to the training of the boosting models. Please be patient if you are trying to knit the entire markdown file yourself.

Introduction

In the previous module, we covered the theory behind ensembling. If you have not looked at the `bagging_and_random_forests.Rmd` module, please take a look at that module before continuing with this one. The basic concept behind bagging is that we generate lots of samples from our data, and then average predictions across a lot of weak learners. This works pretty well when our weak learners are not highly correlated, but we still treat all predictions across all the models equally for each data point. Surely, not all models perform equally well predicting each observation, there must be some variation between one model's ability to predict one point and a different model's ability to predict another. This concept is the basis of *boosting*.

Instead of aggregating lots of models and weighting them equally, boosting takes advantage of a sequential learning algorithm that iteratively adjusts the weights of each observation. Just like bagging, boosting works by fitting lots of weak learners, but instead of fitting them independently, we fit them sequentially and use the predictions from the previous learner to adjust the weighting for the next model. After fitting the first weak learner, we keep track of which data points were misclassified and increase their relative weight for the next model. We repeat the process, re-adjusting the weights at each iteration and then combine our predictions at the end in our ensembled model.

As we mentioned in the previous module, bagging is a variance-reduction technique, so it works particularly well when combining weak learners that have high variance but low bias. In contrast, *boosting is a bias-reduction technique*, and therefore works well when we have many weak learners that have low variance but high bias. A particularly good method to use for weak learners for boosting is to iteratively fit *decision stumps*, or decision trees with a depth of 1. These base models have very small variance, but are of course highly biased. There are many different boosting algorithms that apply this logic in different ways, but in this module we will cover two of the most popular methods: AdaBoost and Gradient Boosting.

AdaBoost

We first will go over the AdaBoost algorithm, which was one of the first boosting algorithms developed and is still quite popular to this day. We state the algorithm below.

AdaBoost Algorithm

1. Initialize weights of the observations to be $w_i = \frac{1}{N},\ i = 1, 2, ..., N$
2. For $m = 1$ to M
    a. Fit a classifier $G_m(x)$ to the training data using weights w_i
    b. Compute $err_m = \frac{\Sigma_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\Sigma_{i=1}^{N} w_i}$

c. Compute $\alpha_m = log((1 - err_m)/err_m)$

d. Set $w_i \leftarrow w_i \cdot exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, ..., N$

3. Output $G(x) = sign[\Sigma_{m=1}^{M} \alpha_m G_m(x)]$

We can see that the basic idea is to iteratively re-weight our data points as we sequentially fit more and more weak learners. Even though each individual model predicts pretty poorly, when aggregated with this logic we are able to produce much more accurate predictions.

Example

We will show an implementation of Adaboost in R using the `caret` package as a framework to tune the model from the `ada` package, and a data set on Breast Cancer patients in Wisconsin taken from the UCI Machine Learning Repository (archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/). We must first import the data and clean it before we begin our analysis. This data set contains 699 observations and 10 features. We list the features in Table 1.

```
# Load libraries
library(caret)
library(ada)
library(gbm)
library(xgboost)

# Read in breast cancer data
bc_data <- read.table("data/breast-cancer-wisconsin.data.txt",
    sep = ",")

# Name columns
bc_names <- c("code", "clump_thickness", "size_uniformity", "shape_uniformity",
    "marginal_adhesion", "single_epithelial_size", "bare_nuclei",
    "bland_chromatin", "normal_nucleoli", "mitoses", "class")
colnames(bc_data) <- bc_names

# Drop ID column
bc_data <- bc_data %>% select(-c(code))

# Convert class variable to binary factor
bc_data$class <- ifelse(bc_data$class == 2, -1, 1)
bc_data$class <- as.factor(bc_data$class)

# Replace missing values with random values in range
bc_data$bare_nuclei[bc_data$bare_nuclei == "?"] <- sample(1:10,
    replace = TRUE)

# Convert bare nuclei to numeric
bc_data$bare_nuclei <- as.numeric(bc_data$bare_nuclei) - 1

# Summarize data
summary(bc_data)
```

```
 clump_thickness  size_uniformity  shape_uniformity marginal_adhesion
 Min.   : 1.000   Min.   : 1.000   Min.   : 1.000   Min.   : 1.000
 1st Qu.: 2.000   1st Qu.: 1.000   1st Qu.: 1.000   1st Qu.: 1.000
 Median : 4.000   Median : 1.000   Median : 1.000   Median : 1.000
```

No.	Feature	Domain
1	Sample code number	id number
2	Clump Thickness	1 - 10
3	Uniformity of Cell Size	1 - 10
4	Uniformity of Cell Shape	1 - 10
5	Marginal Adhesion	1 - 10
6	Single Epithelial Cell Size	1 - 10
7	Bare Nuclei	1 - 10
8	Bland Chromatin	1 - 10
9	Normal Nucleoli	1 - 10
10	Mitoses	1 - 10
11	Class	(2 for benign, 4 for malignant)

Table 1: Features for Wisconsin Breast Cancer Data Set

```
 Mean   : 4.418   Mean   : 3.134   Mean   : 3.207   Mean   : 2.807
 3rd Qu.: 6.000   3rd Qu.: 5.000   3rd Qu.: 5.000   3rd Qu.: 4.000
 Max.   :10.000   Max.   :10.000   Max.   :10.000   Max.   :10.000
 single_epithelial_size  bare_nuclei     bland_chromatin  normal_nucleoli
 Min.   : 1.000         Min.   : 1.000   Min.   : 1.000   Min.   : 1.000
 1st Qu.: 2.000         1st Qu.: 1.000   1st Qu.: 2.000   1st Qu.: 1.000
 Median : 2.000         Median : 1.000   Median : 3.000   Median : 1.000
 Mean   : 3.216         Mean   : 2.245   Mean   : 3.438   Mean   : 2.867
 3rd Qu.: 4.000         3rd Qu.: 2.000   3rd Qu.: 5.000   3rd Qu.: 4.000
 Max.   :10.000         Max.   :10.000   Max.   :10.000   Max.   :10.000
    mitoses       class
 Min.   : 1.000   -1:458
 1st Qu.: 1.000   1 :241
 Median : 1.000
 Mean   : 1.589
 3rd Qu.: 1.000
 Max.   :10.000
```

Notice that in the code above we converted the binary outcome variable from labels "2" and "4" to "-1" and "1" to be more explicit about what the outcome labels represent.

In order to train and assess our model we need to divide our data set up into training/validation, query, and test sets. Once we chunk our data, we can perform cross-validation in order to find the optimal values of our hyper-parameters for our model.

```
# Create training/test index
tr_idx <- sample(1:nrow(bc_data), nrow(bc_data) * 0.6, replace = FALSE)

# Create training/validation set
bc_data_train <- bc_data[tr_idx, ]

bc_data_test <- bc_data[-tr_idx, ]

# Create query set from test set
qr_idx <- sample(1:nrow(bc_data_test), nrow(bc_data_test) * 0.5,
    replace = FALSE)

bc_data_query <- bc_data_test[qr_idx, ]

bc_data_test <- bc_data_test[-qr_idx, ]
```

For AdaBoost, we need to tune the number of trees (`iter`), max tree depth (`maxdepth`), and the learning rate (`nu`). First, we will tune the number of trees at the default settings for the other hyper-parameters, and then once we have determined our optimal number of trees we can perform a grid search on our other hyper-parameters. This will allow us to decrease the amount of time it takes to tune our model using cross-validation since we are reducing the total number of combinations we have to test

```
# Create tuning controls for 5-fold CV
tune_control <- trainControl(method = "cv", number = 5)

# Create grid for number of trees
tree_grid <- expand.grid(iter = seq(25, 150, 25), maxdepth = 1,
    nu = 1)

# Fit AdaBoost model with 5-fold CV
adaboost_tree_cv <- train(class ~ ., data = bc_data_train, method = "ada",
    trControl = tune_control, verbose = FALSE, tuneGrid = tree_grid)

kable(adaboost_tree_cv$bestTune, format = "latex")
```

iter	maxdepth	nu
25	1	1

We can see the above output of our optimal hyper-parameters from our cross-validation process. Now, we can search over a grid of the other two hyper-parameters and find the optimal values for max depth and learning rate.

```
# Create grid for other hyper-parameters
tune_grid <- expand.grid(iter = adaboost_tree_cv$bestTune[["iter"]],
    maxdepth = seq(1, 4), nu = 10^seq(-1, -4, -1))

# Fit AdaBoost model with 5-fold CV
adaboost_tune_cv <- train(class ~ ., data = bc_data_train, method = "ada",
    trControl = tune_control, verbose = FALSE, tuneGrid = tune_grid)

kable(adaboost_tune_cv$bestTune, format = "latex")
```

	iter	maxdepth	nu
16	25	4	0.1

Again, we can see our optimal hyper-parameter values in the output above. We then refit the AdaBoost model on the training data at the hyper-parameters values tuned by our cross-validation process.

```
# Retrain on training data
adaboost_fit <- ada(class ~ ., data = bc_data_train, loss = "exponential",
    iter = adaboost_tune_cv$bestTune[["iter"]], rpart.control(maxdepth = adaboost_tune_cv$bestTune[["ma
    nu = adaboost_tune_cv$bestTune[["nu"]])

# Predict on the query set
ada_y_hat_query <- predict(adaboost_fit, bc_data_query)

# Calculate prediction accuracy
ada_err <- 1 - mean(ada_y_hat_query == bc_data_query$class)
```

We can see that when predicting on the query set we achieve a misclassification error of only:

```
ada_err
```

```
[1] 0.03571429
```

We can use this prediction error to compare the AdaBoost model to other boosting models we will fit.

Gradient Boosting

Gradient Boosting takes the fundamental concepts introduced by the original AdaBoost algorithm, but adapts them so that the algorithm is not as greedy. By computing the sequential base learner that maximizes the reduction in error such as in the AdaBoost algorithm, we actually create dependence between our tree components as they are equivalent to the components of the negative gradient. This would be a preferred approach if we were solely interested in minimizing prediction error on the training data, but of course, we are interested in minimizing prediction error on **unseen data**.

Instead, Gradient Boosting seeks to minimize the difference between the newly induced tree and the components of the negative gradient. This means that instead of fitting the new trees to the data, we are fitting the new tree to the components of the negative gradient. The exact loss function between the newly fitted tree and the negative gradient is based on the prediction task and is subject to some choice. We outline the Gradient Boosting algorithm for regression below. The algorithm for classification trees is quite similar, except we use different functions to output probabilities or predicted classes at the very end (the $f_M(x)$ function).

Gradient Boosting Algorithm for Regression

1. Initialize $f_0(x) = \underset{\gamma}{\text{argmin}} \Sigma_{i=1}^{N} L(y_i, \gamma)$
2. For $m = 1$ to M:
   a. For $i = 1, 2, ..., N$ compute

$$r_{im} = -[\frac{\delta L(y_i, f(x_i))}{\delta f(x_i)}]_{f=f_{m-1}}$$

   b. Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, ..., J_M$
   c. For $j = 1, 2, ..., J_M$ compute

$$\gamma_{jm} = \underset{\gamma}{\text{argmin}} \Sigma_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

   d. Update $f_m(x) = f_{m-1}(x) + \Sigma_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$
3. Output $\hat{f}(x) = f_M(x)$

Example

Just like we did the AdaBoost implementation, we will use the `caret` package as a framework to implement a Gradient Boosting model from the `gbm` package. We will first establish the number of trees to build at the default values of the other hyper-parameters before tuning the rest using a grid search. This alllows us to speed up the computational time of training since we are not searching over **all** possible combination of values.

```
# Adjust class labels for gbm package
bc_data_train$class <- as.factor(ifelse(bc_data_train$class ==
    1, 1, 0))
bc_data_query$class <- as.factor(ifelse(bc_data_query$class ==
    1, 1, 0))
```

```
bc_data_test$class <- as.factor(ifelse(bc_data_test$class ==
    1, 1, 0))

# Create grid for number of trees
tree_grid <- expand.grid(n.trees = seq(25, 150, 25), interaction.depth = 1,
    shrinkage = 0.1, n.minobsinnode = 10)

# Fit gradient boosting model with 5-fold CV
gbm_tree_cv <- train(class ~ ., data = bc_data_train, method = "gbm",
    trControl = tune_control, verbose = FALSE, tuneGrid = tree_grid)

kable(gbm_tree_cv$bestTune, format = "latex")
```

	n.trees	interaction.depth	shrinkage	n.minobsinnode
3	75	1	0.1	10

We can see that the optimal number of trees in the output above. Now, we will perform a grid search over a combination of the other hyper-parameter values to find the optimal values.

```
# Create grid for number of trees
tune_grid <- expand.grid(n.trees = gbm_tree_cv$bestTune[["n.trees"]],
    interaction.depth = seq(1, 4), shrinkage = 10^seq(-1, -4,
        -1), n.minobsinnode = seq(5, 15, 5))

# Fit gradient boosting model with 5-fold CV
gbm_tune_cv <- train(class ~ ., data = bc_data_train, method = "gbm",
    trControl = tune_control, verbose = FALSE, tuneGrid = tune_grid)

kable(gbm_tune_cv$bestTune, format = "latex")
```

	n.trees	interaction.depth	shrinkage	n.minobsinnode
48	75	4	0.1	15

We see from the results from our cross-validation process the optimal values outputted above. Now, we can re-fit the model at the optimal values of our hyper-parameters and assess its predictive accuracy on the query set to compare it to the AdaBoost model.

```
# Convert to numeric
bc_data_train$class <- as.numeric(bc_data_train$class) - 1
bc_data_query$class <- as.numeric(bc_data_query$class) - 1
bc_data_test$class <- as.numeric(bc_data_test$class) - 1


# Refit model at optimal values
gbm_fit <- gbm(class ~ ., data = bc_data_train, distribution = "bernoulli",
    n.trees = gbm_tune_cv$bestTune[["n.trees"]], interaction.depth = gbm_tune_cv$bestTune[["interaction
    shrinkage = gbm_tune_cv$bestTune[["shrinkage"]], n.minobsinnode = gbm_tune_cv$bestTune[["n.minobsin

# Predict on the query set
gbm_y_hat_query <- predict(gbm_fit, bc_data_query, n.trees = gbm_tune_cv$bestTune[["n.trees"]],
    type = "response")

# Calculate prediction accuracy
gbm_err <- 1 - mean(round(gbm_y_hat_query) == bc_data_query$class)
```

We can see that the Gradient Boosting algorithm performs somewhat better on new data as compared to the AdaBoost algorithm with a classification error of:

```
gbm_err
```

```
[1] 0.03571429
```

Extreme Gradient Boosting

One of the most popular methods in online machine learning competitions is a technique called Extreme Gradient Boosting (xgboost). Xgboost is able to perform gradient boosting in a more computationally efficient manner, as well as adding regularization in order to control for over-fitting. In general, xgboost has been shown to perform better than standard implementations of Gradient Boosting and in general is one of the most powerful predictive models.

Example

We show an implementation on the xgboost algorithm using the `caret` framework for the `xgboost` package. We first tune the number of trees, maximum depth, and using the default values of our other hyper-parameters. The reason we divide the tuning process into two stages is due to the high number of hyper-parameters in the model. Even if we only tune over 3 values of each hyper-parameter, the total number of models we will have to fit will be $3^7 \times 5 = 10935$.

```
# Adjust class labels for xgboost package
bc_data_train$class <- as.factor(ifelse(bc_data_train$class ==
    1, 1, 0))
bc_data_query$class <- as.factor(ifelse(bc_data_query$class ==
    1, 1, 0))
bc_data_test$class <- as.factor(ifelse(bc_data_test$class ==
    1, 1, 0))

# Create grid for number of trees
tune_grid1 <- expand.grid(nrounds = seq(25, 150, 25), max_depth = seq(2,
    8, 2), eta = seq(0.1, 0.9, 0.2), gamma = 0, colsample_bytree = 1,
    min_child_weight = 1, subsample = 1)

# Fit xgboost model with 5-fold CV
xgb_tree_cv <- train(class ~ ., data = bc_data_train, method = "xgbTree",
    trControl = tune_control, verbose = FALSE, tuneGrid = tune_grid1)

kable(xgb_tree_cv$bestTune, format = "latex")
```

	nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
67	25	8	0.5	0	1	1	1

We can see the that the optimal values of the tuned hyper-parameters are displayed above. We now tune the rest of our hyper-parameters in order to discover our best model.

```
# Create grid for number of trees
tune_grid2 <- expand.grid(nrounds = xgb_tree_cv$bestTune[["nrounds"]],
    max_depth = xgb_tree_cv$bestTune[["max_depth"]], eta = xgb_tree_cv$bestTune[["eta"]],
    gamma = c(0, 1, 5, 10), colsample_bytree = seq(0.1, 1, 0.3),
    min_child_weight = c(0, 1, 5, 10), subsample = seq(0.1, 1,
        0.3))

# Fit AdaBoost model with 5-fold CV
```

```
xgb_tune_cv <- train(class ~ ., data = bc_data_train, method = "xgbTree",
    trControl = tune_control, verbose = FALSE, tuneGrid = tune_grid2)

kable(xgb_tune_cv$bestTune, format = "latex")
```

	nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
49	25	8	0.5	0	1	0	0.1

With our optimal values set for `nrounds`, `max_depth`, and `eta`, we found the optimal values of our other hyper-parameters to be `gamma`, `colsample_bytree`, `min_child_weight`, and `subsample`. The optimal values of these being displayed above.

```
# Convert to numeric
bc_data_train$class <- as.numeric(bc_data_train$class) - 1
bc_data_query$class <- as.numeric(bc_data_query$class) - 1
bc_data_test$class <- as.numeric(bc_data_test$class) - 1

# Refit model at optimal values
xgb_fit <- xgboost(data = as.matrix(bc_data_train[, -10]), label = as.numeric(bc_data_train[,
    10]), nrounds = xgb_tune_cv$bestTune[["nrounds"]], objective = "binary:logistic",
    verbose = 0, params = list(max_depth = xgb_tune_cv$bestTune[["max_depth"]],
        eta = xgb_tune_cv$bestTune[["eta"]], gamma = xgb_tune_cv$bestTune[["gamma"]],
        colsample_bytree = xgb_tune_cv$bestTune[["colsample_bytree"]],
        min_child_weight = xgb_tune_cv$bestTune[["min_child_weight"]],
        subsample = xgb_tune_cv$bestTune[["subsample"]]))

# Predict on the query set
xgb_y_hat_query <- predict(xgb_fit, as.matrix(bc_data_query[,
    -10]), type = "response")

# Calculate prediction accuracy
xgb_err <- 1 - mean(round(xgb_y_hat_query) == bc_data_query$class)
```

We can see that our optimally tuned xgboost model had a prediction error on the query set of:

```
xgb_err
```

```
[1] 0.02857143
```

This means that it performed better than both the AdaBoost and gradient boosting models.

Final Test Error

Since xgboost performed the best out of the three models we tried on the query set, we will use it to report our final prediction error on the test set. We re-fit the model at its optimal hyper-parameter vaules on the combined training and query set before predicting on the test.

```
# Combine training and query sets
bc_data_train_query <- data.frame(rbind(bc_data_train, bc_data_query))

# Train on training & query data Refit model at optimal
# values
test_model_fit <- xgboost(data = as.matrix(bc_data_train_query[,
    -10]), label = as.numeric(bc_data_train_query[, 10]), nrounds = xgb_tune_cv$bestTune[["nrounds"]],
    objective = "binary:logistic", verbose = 0, params = list(max_depth = xgb_tune_cv$bestTune[["max_de
```

```
        eta = xgb_tune_cv$bestTune[["eta"]], gamma = xgb_tune_cv$bestTune[["gamma"]],
        colsample_bytree = xgb_tune_cv$bestTune[["colsample_bytree"]],
        min_child_weight = xgb_tune_cv$bestTune[["min_child_weight"]],
        subsample = xgb_tune_cv$bestTune[["subsample"]]))

# Predict on the test set
y_hat_test <- predict(test_model_fit, as.matrix(bc_data_test[,
    -10]), type = "response")

# Calculate prediction accuracy on test set
test_err <- 1 - mean(round(y_hat_test) == bc_data_test$class)
test_err
```

```
[1] 0.04285714
```

We can see that the xgboost algorithm did not predict quite as well on the test set as it did on the query set. Most likely, we slightly overfit to the training data, but this margin of error is not so large that it couldn't just be due to random chance. In general, ensemble tree models do a good job of not overfitting with many trees, but they can easily overfit if the tree depth is too high. It is often a good idea to try boosting and bagging with tree stumps (i.e. trees with a depth of 1) and compare them to the more complex models.