GRADIENT BOOSTING

The most widely used and recommended library for gradient boosting is called Xgboost. It is available in both Python and R. The idea behind gradient boosting is to build an ensemble of trees (like a random forest) by iteratively choosing a subset of *data* (rather than variables as done in the random forest). However, the subset of data is not chosen randomly - observations that the current model has a difficult time classifying are more heavily weighted in the next iteration. Some loss function (like average squared error) is chosen, and those observations for which the model's prediction has the largest average squared error are weighted more heavily in the training of the next tree.

Preparing the data

For this tutorial, we'll return to the optical character recognition problem and the handwritten digits (PenDigits) dataset. Let's first discuss some of the inputs to this model. To begin with, we can't use a data frame, we have to use the Matrix package to create a design matrix. That input matrix should be of the class dgCMatrix. For data sets with many categorical variables, this first step creates all the dummy variables going into the model. The design matrix will be a sparse data matrix, meaning that zero entries are not stored. For the PenDigits data, this will not look much different than a regular matrix.

The **label** or target variable should be numeric containing class numbers as 0, ..., num_classes. The digit variable has 10 levels (the digits 0,...,9). However, if you type as.numeric(train$digit) the resulting vector will number the levels 1,..,10. In order to use the true numbers 0,..,9 we'll have to use the following command: as.numeric(levels(train$digit))[train$digit].

```
> # first prepare the data
> load("PenDigits.Rdata")
> library('Matrix')
> sparse_train = sparse.model.matrix(digit ~ . -digit, data=train)
> sparse_valid = sparse.model.matrix(digit ~ . -digit, data=test)
> train_label = as.numeric(levels(train$digit))[train$digit]
```

To see what a sparse model matrix might look like on a dataset with categorical variables, try the above command on the bank data. You'll see the number of columns in the data expand greatly because of the number of dummy variables necessary to represent the categorical variables.

Other model parameters

Now let's discuss the input parameters and run the model. First of all, we have to tell the model what type of classification problem we're doing - is it binary or multiclass or regression? Depending on which type of problem, we'll choose the objective function (loss function).

- For binary problems, use objective = "binary:logistic".
- For multiclass problems, use objective = "multi:softprob" if you'd like to output the probabilities for *every* class or (more likely) use objective = "multi:softmax" to output the decision class based on the max probability.
- For regression problems, use the default objective = reg:linear.

Several other parameters need to be specified and can be tuned to optimize your model. Below the default values of the parameters are given, along with a brief description of their purpose

- eta = 0.03. The step-size shrinkage used to prevent over-fitting, it ranges from 0 to 1 with smaller values creating models more robust to overfitting (although slower to compute). The gradient boosting model is an additive one where (essentially) we try to model the residual values from the previous round in every iteration. To prevent overfitting, we do not use the residual prediction outright but we shrink it by some penalty. This prevents us from fitting the training set perfectly and just inches us closer to the correct decision boundary at every step.
- gamma = 0. The minimum loss reduction required to make a further partition on a leaf node of the tree. The larger the value of gamma, the more conservative the model will be.
- max_depth = 6. The minimum depth of a tree.
- subsample = 1. The proportion of training observations to use at each iteration. For example, setting to 0.5 means that xgboost will randomly collect half of the data to grow trees. Makes computation time shorter.
- colsample_bytree = 1. The proportion of variables to sample and use when constructing each tree (like random forest).
- nrounds. The max number of iterations taken.
- num_class. Specifies the number of levels of the target variable.
- verbose = 1. If 0, method will stay silent and output answer. If 1 will print information of performance (eval_metric) for each iteration.
- eval_metric = . Set automatically by objective but can be over-ridden for another metric.
  - rmse. Root mean square error.
  - logloss. Negative log-likelihood.
  - error. Binary misclassification rate using cutoff probability of 0.5.
  - merror. Multiclass misclassification rate using maximum class probability.

Creating the model

Now that we understand the inputs, let's go ahead and run the model on the PenDigits dataset and check the misclassification rates on the training and validation data sets.

```
> library(xgboost)
> library(readr)
> library(stringr)
> library(caret)
> library(car)
> # tune and run the model
> xgb <- xgboost(data = sparse_train,
+                label = train_label,
+                eta = 0.05,
+                max_depth = 15,
+                gamma = 0,
+                nround=100,
+                subsample = 0.75,
+                colsample_bytree = 0.75,
+                num_class= 10,
+                objective = "multi:softmax",
+                nthread = 3,
+                eval_metric = 'merror',
+                verbose =0)
> ptrain = predict(xgb, sparse_train)
> pvalid = predict(xgb, sparse_valid)
> table(ptrain,train$digit)
```

```
ptrain   0   1   2   3   4   5   6   7   8   9
     0 779   0   0   0   0   0   0   0   0   0
     1   0 779   0   0   0   0   0   0   0   0
     2   0   0 780   1   0   0   0   0   0   0
     3   0   0   0 718   0   0   0   0   0   0
     4   0   0   0   0 780   0   0   0   0   0
     5   0   0   0   0   0 720   1   0   0   0
     6   0   0   0   0   0   0 719   0   0   0
     7   0   0   0   0   0   0   0 778   0   0
     8   0   0   0   0   0   0   0   0 719   0
     9   1   0   0   0   0   0   0   0   0 719
```

```
> XGBmrt = sum(ptrain!=train$digit)/length(train$digit)
> XGBmrv = sum(pvalid!=test$digit)/length(test$digit)
> cat('XGB Training Misclassification Rate:', XGBmrt)
```

```
XGB Training Misclassification Rate: 0.0004003203
```

```
> cat('XGB Validation Misclassification Rate:', XGBmrv)
```

```
XGB Validation Misclassification Rate: 0.0383076
```