Otto-Group Product-Classification Challenge

Leïla JAMSHIDIAN SALES

Contents

1 Problem Understanding

Otto Group is a big company, selling many products. A dataset is provided for Multi-class classification task. Due to various processes used for information gathering, many identical products get classified differently. Therefore, the quality of product analysis depends heavily on the ability to accurately cluster similar products. The better the classification, the more insights we can be generated the product range. There are nine categories for all products. Each target category represents one of our most important product categories (like fashion, electronics, etc.). The products for the training and testing sets are selected randomly.

First we will prepare the **Otto** dataset and train a model, then we will generate vizualisations to get a clue of what is important to the model, finally, we will see how we can leverage these information.

2 Data Understanding and Preparation

This part is based on the **R** tutorial example by Tong He and Michaël Benesty.

First, let's load the packages and the dataset.

```
require(methods)
require(data.table)
require(magrittr)
train <- fread('./otto_data/train.csv', header = T, stringsAsFactors = F)
test <- fread('./otto_data/test.csv', header=TRUE, stringsAsFactors = F)
```

`magrittr` and `data.table` are here to make the code cleaner and much more rapid.

Let's explore the dataset.

```
# Train dataset dimensions
dim(train)
```

```
## [1] 61878    95
```

```
# Training content
train[1:6,1:5, with =F]
```

```
##    id feat_1 feat_2 feat_3 feat_4
## 1:  1      1      0      0      0
## 2:  2      0      0      0      0
## 3:  3      0      0      0      0
## 4:  4      1      0      0      1
## 5:  5      0      0      0      0
## 6:  6      2      1      0      0
```

```
# Test dataset dimensions
dim(train)
```

```
## [1] 61878    95
```

```
# Test content
test[1:6,1:5, with =F]
```

```
##    id feat_1 feat_2 feat_3 feat_4
## 1:  1      0      0      0      0
## 2:  2      2      2     14     16
## 3:  3      0      1     12      1
## 4:  4      0      0      0      1
## 5:  5      1      0      0      1
## 6:  6      0      0      0      0
```

We only display the 6 first rows and 5 first columns for convenience

Each *column* represents a feature measured by an integer. Each *row* is an **Otto** product.

Obviously the first column (`ID`) doesn't contain any useful information.

To let the algorithm focus on real stuff, we will delete it.

```
# Delete ID column in training dataset
train[, id := NULL]

# Delete ID column in testing dataset
test[, id := NULL]
```

Since it is a multi class classification challenge. We need to extract the labels (here the name of the different classes) from the training data. Usually the labels is in the first or the last column. We already know what is in the first column, let's check the content of the last one.

```
# Check the content of the last column
train[1:6, ncol(train), with  = F]
```

```
##     target
## 1: Class_1
## 2: Class_1
## 3: Class_1
## 4: Class_1
## 5: Class_1
## 6: Class_1
```

```
# Save the name of the last column
nameLastCol <- names(train)[ncol(train)]
```

The classes are provided as character string in the **94**th column called **target** and they are in string format.

So we will convert classes to integers for further processing. Moreover, we will start our indexing from 0 (e.g. class_1 maps to 0 and so on).

Hence, we will:

- extract the target column
- remove "Class_" from each class name
- convert to integers
- remove 1 to the new value

```
# Convert from classes to numbers
y <- train[, nameLastCol, with = F][[1]] %>% gsub('Class_','',.) %>% {as.integer(.) -1}
# Display the first 5 levels
y[1:5]
```

```
## [1] 0 0 0 0 0
```

Doing One-Hot encoding of the label which we will use for feature analysis below.

```
yMat <- model.matrix(~ target - 1, train)
colnames(yMat) <- 1:9
```

Finally, we remove **target** column from training dataset, otherwise our models will learn to just use **target** itself!

```
train[, (nameLastCol):=NULL, with = F]
```

```
## Warning in `[.data.table`(train, , `:=`((nameLastCol), NULL), with = F):
## with=FALSE ignored, it isn't needed when using :=. See ?':=' for examples.
```

Plotting the linear correlation between classes and features

```
#Calculation linear correlation between classes and features
linearCor <- abs(cor(train, yMat))

library("ggplot2")
heatmapMatrix <- function(mat, xlab = "X", ylab = "Y", zlab = "Z", low = "white", high = "black",
                          grid = "grey", limits = NULL, legend.position = "top", colours = NULL){
  nr <- nrow(mat)
  nc <- ncol(mat)
  rnames <- rownames(mat)
  cnames <- colnames(mat)
  if(is.null(rnames)) rnames <- paste(xlab, 1:nr, sep = "_")
  if(is.null(cnames)) cnames <- paste(ylab, 1:nc, sep = "_")
  x <- rep(rnames, nc)
  y <- rep(cnames, each = nr)
  df <- data.frame(factor(x, levels = unique(x)),
                   factor(y, levels = unique(y)),
                   as.vector(mat))
  colnames(df) <- c(xlab, ylab, zlab)
  p <- ggplot(df, aes_string(ylab, xlab)) +
    geom_tile(aes_string(fill = zlab), colour = grid) +
      theme(legend.position=legend.position)
  if(is.null(colours)){
    p + scale_fill_gradient(low = low, high = high, limits = limits)
  }else{
    p + scale_fill_gradientn(colours = colours)
  }
}
```

```
p1 <- heatmapMatrix(linearCor, xlab = "X", ylab = "Class", zlab = "Corr", high = "blue")
plot(p1,height=5)
```

As can be observed above, the feature-34 has high correlation with class-5, and class-6 also has high linear correlation with a bunch of features. Note that class-1 has low correlation with any

feature, this means that class-1 is either non-linearly dependent on the features or there is no dependence at all.

3 Modelling

We shall now try and use various models train our classifier.

3.1 XGBoost

XGBoost is a popular implementation of Gradient Boosting methods. It builds a boosted boosted classifier/regressor with decision tree as its base-learner. The incremental weights added to each base-learner is based upon the gradient of some loss define over the space of classifer/regressor functions.

To begin with, note that `data.table` is not an implementation of data.frame which is natively supported by **XGBoost**. So, we need to convert both datasets (training and test) in numeric Matrix format.

```
trainMatrix <- train[,lapply(.SD,as.numeric)] %>% as.matrix
testMatrix <- test[,lapply(.SD,as.numeric)] %>% as.matrix
```

Before the learning we will use the cross validation to evaluate our error rate.

Basically **XGBoost** will divide the training data in `nfold` parts, then **XGBoost** will retain the first part and use it as the test data. Then it will reintegrate the first part to the training dataset and retain the second part, do a training and so on...

```
library("xgboost")
numberOfClasses <- max(y) + 1

param <- list("objective" = "multi:softprob",
              "eval_metric" = "mlogloss",
              "num_class" = numberOfClasses)

cv.nround <- 5
cv.nfold <- 3

bst.cv = xgb.cv(param=param, data = trainMatrix, label = y,
                nfold = cv.nfold, nrounds = cv.nround)
```

```
## [1]  train-mlogloss:1.540819+0.001938    test-mlogloss:1.555003+0.002474
## [2]  train-mlogloss:1.283619+0.002888    test-mlogloss:1.306291+0.001734
## [3]  train-mlogloss:1.114519+0.002699    test-mlogloss:1.143518+0.002434
## [4]  train-mlogloss:0.993382+0.002443    test-mlogloss:1.028087+0.003100
## [5]  train-mlogloss:0.902310+0.003792    test-mlogloss:0.942366+0.002798
```

As we can see the error rate is low on the test dataset (for a 5mn trained model).

Finally, we train a real model:

```
nround = 50
bst = xgboost(param=param, data = trainMatrix, label = y, nrounds=nround)
```

```
## [1]  train-mlogloss:1.541729
## [2]  train-mlogloss:1.286051
## [3]  train-mlogloss:1.119220
## [4]  train-mlogloss:1.000465
## [5]  train-mlogloss:0.909708
```

```
## [6]  train-mlogloss:0.840088
## [7]  train-mlogloss:0.783480
## [8]  train-mlogloss:0.740530
## [9]  train-mlogloss:0.703314
## [10] train-mlogloss:0.672674
## [11] train-mlogloss:0.646605
## [12] train-mlogloss:0.624773
## [13] train-mlogloss:0.606139
## [14] train-mlogloss:0.588466
## [15] train-mlogloss:0.574412
## [16] train-mlogloss:0.561439
## [17] train-mlogloss:0.549918
## [18] train-mlogloss:0.539848
## [19] train-mlogloss:0.530086
## [20] train-mlogloss:0.522048
## [21] train-mlogloss:0.514230
## [22] train-mlogloss:0.507228
## [23] train-mlogloss:0.500830
## [24] train-mlogloss:0.494447
## [25] train-mlogloss:0.488582
## [26] train-mlogloss:0.482355
## [27] train-mlogloss:0.475927
## [28] train-mlogloss:0.470982
## [29] train-mlogloss:0.466095
## [30] train-mlogloss:0.462315
## [31] train-mlogloss:0.457687
## [32] train-mlogloss:0.452818
## [33] train-mlogloss:0.448694
## [34] train-mlogloss:0.444524
## [35] train-mlogloss:0.440993
## [36] train-mlogloss:0.437048
## [37] train-mlogloss:0.432918
## [38] train-mlogloss:0.430102
## [39] train-mlogloss:0.427420
## [40] train-mlogloss:0.423794
## [41] train-mlogloss:0.419426
## [42] train-mlogloss:0.416207
## [43] train-mlogloss:0.413257
## [44] train-mlogloss:0.409784
## [45] train-mlogloss:0.407097
## [46] train-mlogloss:0.405115
## [47] train-mlogloss:0.402373
## [48] train-mlogloss:0.399465
## [49] train-mlogloss:0.397113
## [50] train-mlogloss:0.394394
```

3.2 Feature importance

So far, we have built a model made of **50** trees.

To build a tree, the dataset is divided recursively several times. At the end of the process, you get groups of observations (here, these observations are properties regarding **Otto** products).

Each division operation is called a *split*.

Each group at each division level is called a branch and the deepest level is called a **leaf**.

In the final model, these leafs are supposed to be as pure as possible for each tree, meaning in our case that each leaf should be made of one class of **Otto** product only (of course it is not true, but that's what we try to achieve in a minimum of splits).

Not all splits are equally important. Basically the first split of a tree will have more impact on the purity than, for instance, the deepest split. Intuitively, we understand that the first split makes most of the work, and the following splits focus on smaller parts of the dataset which have been missclassified by the first tree.

In the same way, in Boosting we try to optimize the missclassification at each round (it is called the **loss**). So the first tree will do the big work and the following trees will focus on the remaining, on the parts not correctly learned by the previous trees.

The improvement brought by each split can be measured, it is the **gain**.

Each split is done on one feature only at one value.

Let's see what the model looks like.

```
model <- xgb.dump(bst, with.stats = T)
```

```
## Warning: 'with.stats' is deprecated.
## Use 'with_stats' instead.
## See help("Deprecated") and help("xgboost-deprecated").
```

```
model[1:10]
```

```
##  [1] "booster[0]"
##  [2] "0:[f16<1.5] yes=1,no=2,missing=1,gain=309.719,cover=12222.8"
##  [3] "1:[f29<26.5] yes=3,no=4,missing=3,gain=161.964,cover=11424"
##  [4] "3:[f77<2.5] yes=7,no=8,missing=7,gain=106.092,cover=11416.3"
##  [5] "7:[f52<12.5] yes=13,no=14,missing=13,gain=43.1389,cover=11211.9"
##  [6] "13:[f76<1.5] yes=25,no=26,missing=25,gain=37.407,cover=11143.5"
##  [7] "25:[f16<0.5] yes=49,no=50,missing=49,gain=36.3329,cover=10952.1"
##  [8] "49:leaf=-0.148413,cover=9861.33"
##  [9] "50:leaf=-0.0905567,cover=1090.77"
## [10] "26:[f83<0.5] yes=51,no=52,missing=51,gain=167.766,cover=191.407"
```

For convenience, we are displaying the first 10 lines of the model only.

Clearly, it is not easy to understand what it means.

Basically each line represents a branch, there is the tree ID, the feature ID, the point where it splits, and information regarding the next branches (left, right, when the row for this feature is N/A).

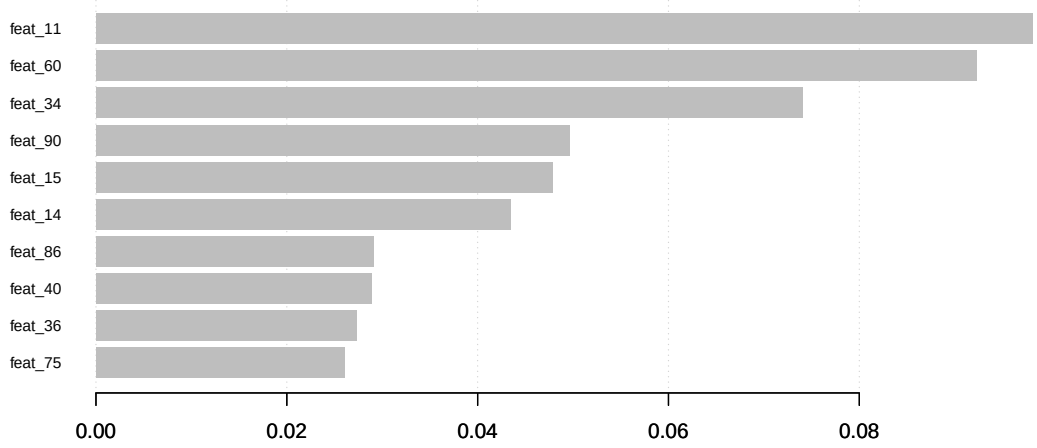
Fortunately, **XGBoost** offers better visual representation for the learnt model.

Feature importance is about averaging the gain of each feature for all split and all trees. Another important aspect would be to know the distribution of obersevations are various leaves (and at what depth do these leaf nodes occur). we can use the functions `xgb.plot.importance` and `xgb.plot.deepness` for this purpose.

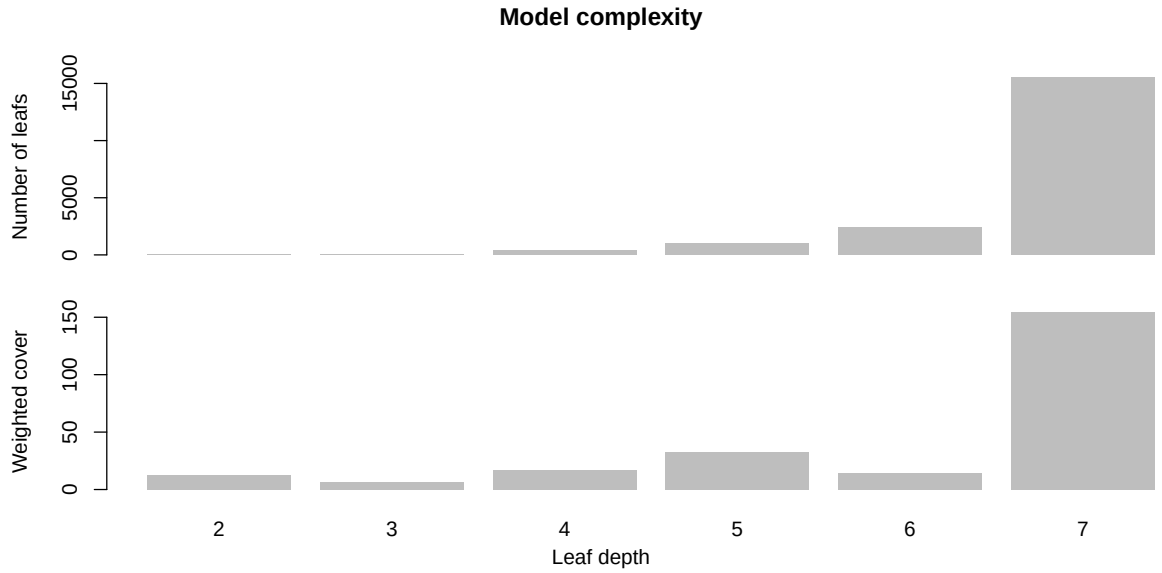
```
# Get the feature real names
names <- dimnames(trainMatrix)[[2]]

# Compute feature importance matrix
importance_matrix <- xgb.importance(names, model = bst)

# Graph of feature importance
xgb.plot.importance(importance_matrix[1:10,])
```




```
# Another important visualization
xgb.plot.deepness(model = bst)
```



3.3 Interpretation

In the feature importance above, we can see the first 10 most important features. This function gives a color to each bar. Basically a K-means clustering is applied to group each feature by importance.

For the depth visualizations, first plot plots a histogram of leaves at varying nodes. The y-axis on the second plot says **cover**, which means that it specifies the weighted observations at each depth. Note that the observations are weighted as the algorithm weghts each observation again, for learning the next base-learner

that is to be added to the ensemble.

Therefore decision-trees in general serve the dual purpose of being a classifier and as a useful pre-processing tool. From here on, one can take several steps. For instance we can remove the less important feature (feature selection process), or go deeper into the interaction between the most important features and labels.

Or we could just reason about why these features are so important (in **Otto** challenge we can't go this way because there is not enough information).

4 Evaluation and Deployment

```
#Build submission
submit <- predict(bst, testMatrix, type="prob")
# shrink the size of submission
submit <- format(submit, digits=2, scientific = FALSE)
submit <- cbind(id=1:nrow(testMatrix), submit)
#Write to csv
write.csv(submit, "submit.csv", row.names=FALSE)
```

The test-score of our model is 0.54, which gives us a position of less than 2000 on the leaderboard.

5 Going deeper

There are 3 documents you may be interested in:

- xgboostPresentation.Rmd: general presentation
- discoverYourData.Rmd: explaining feature analysus
- Feature Importance Analysis with XGBoost in Tax audit: use case
- Fast caliberated KNN: use case of KNN features combined with Generalized Linear Models (GLM).
- Example Kaggle kernel for FKNN: See kaggle kernel for more details.