Understand your dataset with XGBoost

Tianqi Chen, Tong He, Michaël Benesty, Yuan Tang

1 Understand your dataset with XGBoost

1.1 Introduction

The purpose of this vignette is to show you how to use **XGBoost** to discover and understand your own dataset better.

This vignette is not about predicting anything (see XGBoost presentation). We will explain how to use **XGBoost** to highlight the *link* between the *features* of your data and the *outcome*.

Package loading:

```
require(xgboost)
require(Matrix)
require(data.table)
if (!require('vcd')) install.packages('vcd')
```

> **VCD** package is used for one of its embedded dataset only.

1.2 Preparation of the dataset

1.2.1 Numeric v.s. categorical variables

XGBoost manages only `numeric` vectors.

What to do when you have *categorical* data?

A *categorical* variable has a fixed number of different values. For instance, if a variable called *Colour* can have only one of these three values, *red*, *blue* or *green*, then *Colour* is a *categorical* variable.

> In **R**, a *categorical* variable is called `factor`.
>
> Type `?factor` in the console for more information.

To answer the question above we will convert *categorical* variables to `numeric` one.

1.2.2 Conversion from categorical to numeric variables

1.2.2.1 Looking at the raw data

In this Vignette we will see how to transform a *dense* `data.frame` (*dense* = few zeroes in the matrix) with *categorical* variables to a very *sparse* matrix (*sparse* = lots of zero in the matrix) of `numeric` features.

The method we are going to see is usually called one-hot encoding.

The first step is to load `Arthritis` dataset in memory and wrap it with `data.table` package.

```
data(Arthritis)
df <- data.table(Arthritis, keep.rownames = FALSE)
```

> `data.table` is 100% compliant with **R** `data.frame` but its syntax is more consistent and its performance for large dataset is best in class (`dplyr` from **R** and `Pandas` from **Python** included). Some parts of **XGBoost R** package use `data.table`.

The first thing we want to do is to have a look to the first few lines of the `data.table`:

```
head(df)
```

```
##    ID Treatment  Sex Age Improved
## 1: 57   Treated Male  27     Some
## 2: 46   Treated Male  29     None
## 3: 77   Treated Male  30     None
## 4: 17   Treated Male  32   Marked
## 5: 36   Treated Male  46   Marked
## 6: 23   Treated Male  58   Marked
```

Now we will check the format of each column.

```
str(df)
```

```
## Classes 'data.table' and 'data.frame':   84 obs. of  5 variables:
##  $ ID       : int  57 46 77 17 36 23 75 39 33 55 ...
##  $ Treatment: Factor w/ 2 levels "Placebo","Treated": 2 2 2 2 2 2 2 2 2 2 ...
##  $ Sex      : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
##  $ Age      : int  27 29 30 32 46 58 59 59 63 63 ...
##  $ Improved : Ord.factor w/ 3 levels "None"<"Some"<..: 2 1 1 3 3 3 1 3 1 1 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

2 columns have `factor` type, one has `ordinal` type.

> `ordinal` variable :
>
> - can take a limited number of values (like `factor`) ;
> - these values are ordered (unlike `factor`). Here these ordered values are: `Marked > Some > None`

1.2.2.2 Creation of new features based on old ones

We will add some new *categorical* features to see if it helps.

1.2.2.2.1 Grouping per 10 years

For the first feature we create groups of age by rounding the real age.

Note that we transform it to `factor` so the algorithm treat these age groups as independent values.

Therefore, 20 is not closer to 30 than 60. To make it short, the distance between ages is lost in this transformation.

```
head(df[,AgeDiscret := as.factor(round(Age/10,0))])
```

```
##    ID Treatment  Sex Age Improved AgeDiscret
## 1: 57   Treated Male  27     Some          3
## 2: 46   Treated Male  29     None          3
## 3: 77   Treated Male  30     None          3
## 4: 17   Treated Male  32   Marked          3
## 5: 36   Treated Male  46   Marked          5
## 6: 23   Treated Male  58   Marked          6
```

1.2.2.2.2 Random split into two groups

Following is an even stronger simplification of the real age with an arbitrary split at 30 years old. We choose this value **based on nothing**. We will see later if simplifying the information based on arbitrary values is a good strategy (you may already have an idea of how well it will work…).

```
head(df[,AgeCat:= as.factor(ifelse(Age > 30, "Old", "Young"))])
```

```
##    ID Treatment  Sex Age Improved AgeDiscret AgeCat
## 1: 57   Treated Male  27     Some          3  Young
## 2: 46   Treated Male  29     None          3  Young
## 3: 77   Treated Male  30     None          3  Young
## 4: 17   Treated Male  32   Marked          3    Old
## 5: 36   Treated Male  46   Marked          5    Old
## 6: 23   Treated Male  58   Marked          6    Old
```

1.2.2.2.3 Risks in adding correlated features

These new features are highly correlated to the `Age` feature because they are simple transformations of this feature.

For many machine learning algorithms, using correlated features is not a good idea. It may sometimes make prediction less accurate, and most of the time make interpretation of the model almost impossible. GLM, for instance, assumes that the features are uncorrelated.

Fortunately, decision tree algorithms (including boosted trees) are very robust to these features. Therefore we have nothing to do to manage this situation.

1.2.2.2.4 Cleaning data

We remove ID as there is nothing to learn from this feature (it would just add some noise).

```
df[,ID:=NULL]
```

We will list the different values for the column `Treatment` :

```
levels(df[,Treatment])
```

```
## [1] "Placebo" "Treated"
```

1.2.2.3 Encoding categorical features

Next step, we will transform the categorical data to dummy variables. Several encoding methods exist, e.g., one-hot encoding is a common approach. We will use the dummy contrast coding which is popular because it produces “full rank” encoding (also see this blog post by Max Kuhn).

The purpose is to transform each value of each *categorical* feature into a *binary* feature `{0, 1}` .

For example, the column `Treatment` will be replaced by two columns, `TreatmentPlacebo` , and `TreatmentTreated` . Each of them will be *binary*. Therefore, an observation which has the value `Placebo` in column `Treatment` before the transformation will have after the transformation the value `1` in the new column `TreatmentPlacebo` and the value `0` in the new column

TreatmentTreated . The column TreatmentPlacebo will disappear during the contrast encoding, as it would be absorbed into a common constant intercept column.

Column Improved is excluded because it will be our label column, the one we want to predict.

```
sparse_matrix <- sparse.model.matrix(Improved ~ ., data = df)[,-1]
head(sparse_matrix)
```

```
## 6 x 9 sparse Matrix of class "dgCMatrix"
##   TreatmentTreated SexMale Age AgeDiscret3 AgeDiscret4 AgeDiscret5 AgeDiscret6
## 1                1       1  27           1           .           .           .
## 2                1       1  29           1           .           .           .
## 3                1       1  30           1           .           .           .
## 4                1       1  32           1           .           .           .
## 5                1       1  46           .           .           1           .
## 6                1       1  58           .           .           .           1
##   AgeDiscret7 AgeCatYoung
## 1           .           1
## 2           .           1
## 3           .           1
## 4           .           .
## 5           .           .
## 6           .           .
```

> Formula Improved ~ . used above means transform all *categorical* features but column Improved to binary values. The -1 column selection removes the intercept column which is full of 1 (this column is generated by the conversion). For more information, you can type ?sparse.model.matrix in the console.

Create the output numeric vector (not as a sparse Matrix):

```
output_vector = df[,Improved] == "Marked"
```

1. set Y vector to 0 ;
2. set Y to 1 for rows where Improved == Marked is TRUE ;
3. return Y vector.

1.3 Build the model

The code below is very usual. For more information, you can look at the documentation of xgboost function (or at the vignette XGBoost presentation).

```
bst <- xgboost(data = sparse_matrix, label = output_vector, max_depth = 4,
               eta = 1, nthread = 2, nrounds = 10,objective = "binary:logistic")
```

```
## [1]  train-logloss:0.485466
## [2]  train-logloss:0.438534
## [3]  train-logloss:0.412250
## [4]  train-logloss:0.395828
## [5]  train-logloss:0.384264
## [6]  train-logloss:0.374028
## [7]  train-logloss:0.365005
## [8]  train-logloss:0.351233
## [9]  train-logloss:0.341678
## [10] train-logloss:0.334465
```

You can see some `train-error: 0.XXXXX` lines followed by a number. It decreases. Each line shows how well the model explains your data. Lower is better.

A small value for training error may be a symptom of overfitting, meaning the model will not accurately predict the future values.

> Here you can see the numbers decrease until line 7 and then increase.
>
> It probably means we are overfitting. To fix that I should reduce the number of rounds to `nrounds = 4`. I will let things like that because I don't really care for the purpose of this example :-)

1.4 Feature importance

1.5 Measure feature importance

1.5.1 Build the feature importance data.table

Remember, each binary column corresponds to a single value of one of *categorical* features.

```
importance <- xgb.importance(feature_names = colnames(sparse_matrix), model = bst)
head(importance)
```

```
##             Feature        Gain      Cover  Frequency
## 1:              Age 0.622031768 0.67251696 0.67241379
## 2: TreatmentTreated 0.285750540 0.11916651 0.10344828
## 3:          SexMale 0.048744026 0.04522028 0.08620690
## 4:      AgeDiscret6 0.016604636 0.04784639 0.05172414
## 5:      AgeDiscret3 0.016373781 0.08028951 0.05172414
## 6:      AgeDiscret4 0.009270558 0.02858801 0.01724138
```

> The column `Gain` provide the information we are looking for.
>
> As you can see, features are classified by `Gain`.

`Gain` is the improvement in accuracy brought by a feature to the branches it is on. The idea is that before adding a new split on a feature X to the branch there was some wrongly classified elements, after adding the split on this feature, there are two new branches, and each of these branch is more accurate (one branch saying if your observation is on this branch then it should be classified as `1`, and the other branch saying the exact opposite).

`Cover` measures the relative quantity of observations concerned by a feature.

`Frequency` is a simpler way to measure the `Gain`. It just counts the number of times a feature is used in all generated trees. You should not use it (unless you know why you want to use it).

1.5.1.1 Improvement in the interpretability of feature importance data.table

We can go deeper in the analysis of the model. In the `data.table` above, we have discovered which features counts to predict if the illness will go or not. But we don't yet know the role of these features. For instance, one of the question we may want to answer would be: does receiving a placebo treatment helps to recover from the illness?

One simple solution is to count the co-occurrences of a feature and a class of the classification.

For that purpose we will execute the same function as above but using two more parameters, `data` and `label`.

```r
importanceRaw <- xgb.importance(feature_names = colnames(sparse_matrix), model = bst,
         data = sparse_matrix, label = output_vector)
```

```
## Warning in xgb.importance(feature_names = colnames(sparse_matrix), model =
## bst, : xgb.importance: parameters 'data', 'label' and 'target' are deprecated
```

```r
# Cleaning for better display
importanceClean <- importanceRaw[,`:=`(Cover=NULL, Frequency=NULL)]

head(importanceClean)
```

```
##              Feature        Gain
## 1:               Age 0.622031768
## 2: TreatmentTreated 0.285750540
## 3:           SexMale 0.048744026
## 4:       AgeDiscret6 0.016604636
## 5:       AgeDiscret3 0.016373781
## 6:       AgeDiscret4 0.009270558
```

> In the table above we have removed two not needed columns and select only the first lines.

First thing you notice is the new column `Split`. It is the split applied to the feature on a branch of one of the tree. Each split is present, therefore a feature can appear several times in this table. Here we can see the feature `Age` is used several times with different splits.

How the split is applied to count the co-occurrences? It is always `<` . For instance, in the second line, we measure the number of persons under 61.5 years with the illness gone after the treatment.

The two other new columns are `RealCover` and `RealCover %` . In the first column it measures the number of observations in the dataset where the split is respected and the label marked as `1` . The second column is the percentage of the whole population that `RealCover` represents.

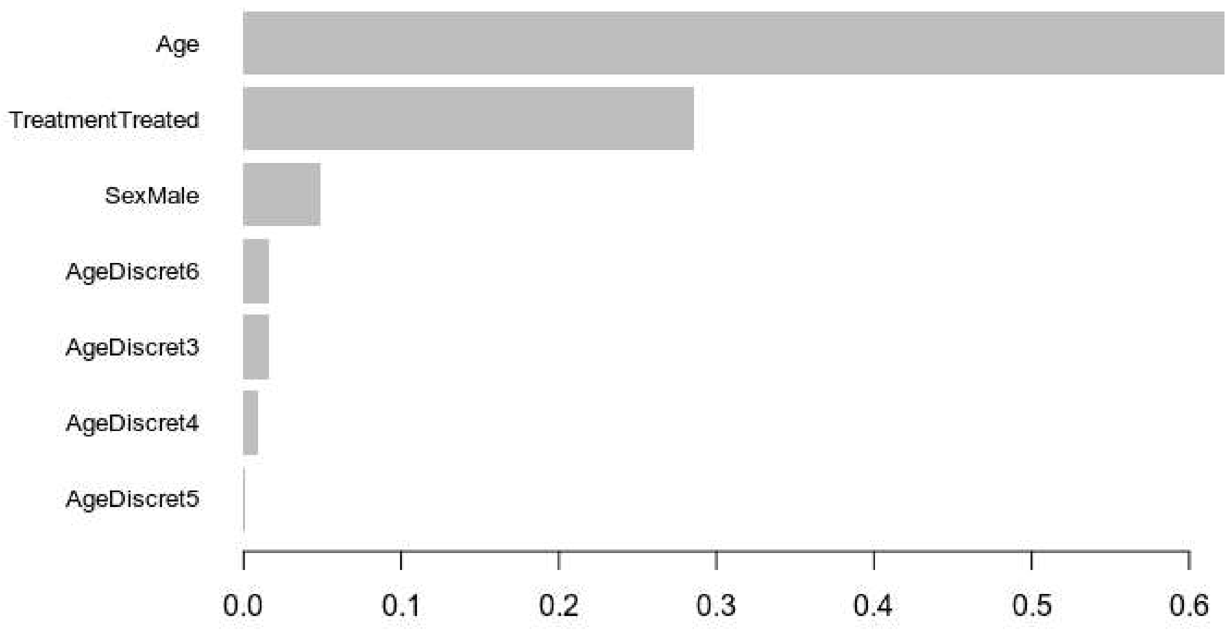
Therefore, according to our findings, getting a placebo doesn't seem to help but being younger than 61 years may help (seems logic).

> You may wonder how to interpret the `< 1.00001` on the first line. Basically, in a sparse `Matrix` , there is no `0` , therefore, looking for one hot-encoded categorical observations validating the rule `< 1.00001` is like just looking for `1` for this feature.

1.5.2 Plotting the feature importance

All these things are nice, but it would be even better to plot the results.

```
xgb.plot.importance(importance_matrix = importance)
```



Feature have automatically been divided in 2 clusters: the interesting features… and the others.

> Depending of the dataset and the learning parameters you may have more than two clusters. Default value is to limit them to `10` ,

> but you can increase this limit. Look at the function documentation for more information.

According to the plot above, the most important features in this dataset to predict if the treatment will work are :

- the Age ;
- having received a placebo or not ;
- the sex is third but already included in the not interesting features group ;
- then we see our generated features (AgeDiscret). We can see that their contribution is very low.

1.5.3 Do these results make sense?

Let's check some **Chi2** between each of these features and the label.

Higher **Chi2** means better correlation.

```r
c2 <- chisq.test(df$Age, output_vector)
print(c2)
```

```
##
##  Pearson's Chi-squared test
##
## data:  df$Age and output_vector
## X-squared = 35.475, df = 35, p-value = 0.4458
```

Pearson correlation between Age and illness disappearing is **35.48**.

```r
c2 <- chisq.test(df$AgeDiscret, output_vector)
print(c2)
```

```
##
##  Pearson's Chi-squared test
##
## data:  df$AgeDiscret and output_vector
## X-squared = 8.2554, df = 5, p-value = 0.1427
```

Our first simplification of Age gives a Pearson correlation is **8.26**.

```r
c2 <- chisq.test(df$AgeCat, output_vector)
print(c2)
```

```
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  df$AgeCat and output_vector
## X-squared = 2.3571, df = 1, p-value = 0.1247
```

The perfectly random split I did between young and old at 30 years old have a low correlation of **2.36**. It's a result we may expect as may be in my mind > 30 years is being old (I am 32 and starting feeling old, this may explain that), but for the illness we are studying, the age to be vulnerable is not the same.

Morality: don't let your *gut* lower the quality of your model.

In *data science* expression, there is the word *science* :-)

1.6 Conclusion

As you can see, in general *destroying information by simplifying it won't improve your model*. **Chi2** just demonstrates that.

But in more complex cases, creating a new feature based on existing one which makes link with the outcome more obvious may help the algorithm and improve the model.

The case studied here is not enough complex to show that. Check Kaggle website for some challenging datasets. However it's almost always worse when you add some arbitrary rules.

Moreover, you can notice that even if we have added some not useful new features highly correlated with other features, the boosting tree algorithm have been able to choose the best one, which in this case is the Age.

Linear model may not be that smart in this scenario.

1.7 Special Note: What about Random Forests™?

As you may know, Random Forests algorithm is cousin with boosting and both are part of the ensemble learning family.

Both trains several decision trees for one dataset. The *main* difference is that in Random Forests, trees are independent and in boosting, the tree `N+1` focus its learning on the loss (<=> what has not been well modeled by the tree `N`).

This difference have an impact on a corner case in feature importance analysis: the *correlated features*.

Imagine two features perfectly correlated, feature `A` and feature `B`. For one specific tree, if the algorithm needs one of them, it will choose randomly (true in both boosting and Random Forests).

However, in Random Forests this random choice will be done for each tree, because each tree is independent from the others. Therefore, approximatively, depending of your parameters, 50% of the trees will choose feature `A` and the other 50% will choose feature `B`. So the *importance* of the information contained in `A` and `B` (which is the same, because they are perfectly correlated) is diluted in `A` and `B`. So you won't easily know this information is important to predict what you want to predict! It is even worse when you have 10 correlated features…

In boosting, when a specific link between feature and outcome have been learned by the algorithm, it will try to not refocus on it (in theory it is what happens, reality is not always that simple). Therefore, all the importance will be on feature `A` or on feature `B` (but not both). You will know that one feature have an important role in the link between the observations and the label. It is still up to you to search for the correlated features to the one detected as important if you need to know all of them.

If you want to try Random Forests algorithm, you can tweak XGBoost parameters!

For instance, to compute a model with 1000 trees, with a 0.5 factor on sampling rows and columns:

```
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test

#Random Forest - 1000 trees
bst <- xgboost(data = train$data, label = train$label, max_depth = 4, num_parallel_tree
         = 1000, subsample = 0.5, colsample_bytree =0.5, nrounds = 1, objective =
         "binary:logistic")
```

```
## [1]  train-logloss:0.456380
```

```
#Boosting - 3 rounds
bst <- xgboost(data = train$data, label = train$label, max_depth = 4, nrounds = 3,
         objective = "binary:logistic")
```

```
## [1]  train-logloss:0.444882
## [2]  train-logloss:0.302428
## [3]  train-logloss:0.212847
```

> Note that the parameter `round` is set to `1`.

> **Random Forests** is a trademark of Leo Breiman and Adele Cutler and is licensed exclusively to Salford Systems for the commercial release of the software.