Lessons from Kaggle Competitions

Based on interviews posted in Kaggle's blog

Brooke Anderson

May 3, 2016

Airbnb challenge

Outcome: Which country a new user will book in first (out of 10 possible countries)

Predictors: Various information for each user, including:

- demographics (gender, age, language)
- web session records (actions, timing of actions, details of actions)
- summary statistics (when account was created, timestamp of the first activity, date of first booking, method of signing up)

Prize: This was a recruitment challenge.

Evaluation

The competition was evaluated using Normalized discounted cumulative gain (NDCG).

From Kaggle:

> *“Normalized discounted cumulative gain (NDCG) measures the performance of a recommendation system based on the graded relevance of the recommended entities. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the entities. This metric is commonly used in information retrieval and to evaluate the performance of web search engines.”"*

Evaluation

From Kaggle:

> *"For each new user, you are to make a maximum of 5 predictions on the country of the first booking. The ground truth country is marked with relevance = 1, while the rest have relevance = 0."*

Challenges

Some of the challenging parts of this competition were:

- 12-class classification
- highly unbalanced
- hard to cross-validate (test data only had accounts created in the last three months of the data)

Airbnb: Second place

Second place winner: Keiichi Kuoraya

An interview on how he did it is available here.

Airbnb: Pre-processing

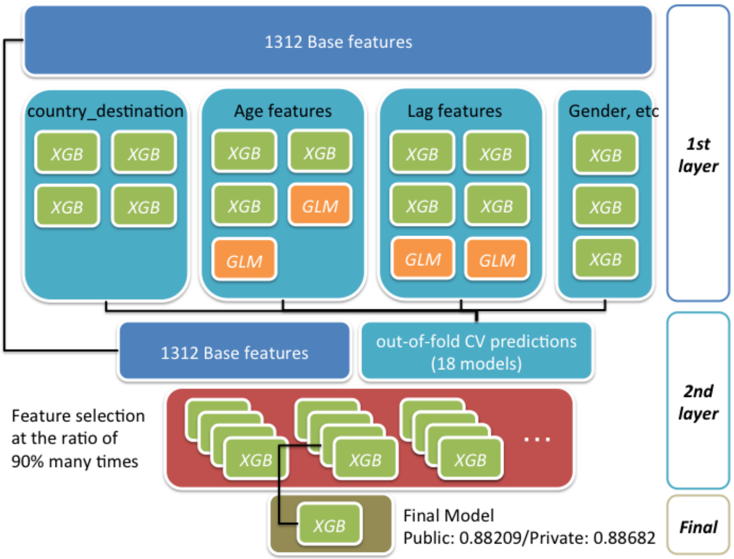
- For numeric variables, kept everything as-is except age. This needed some cleaning up, because it had some implausible values.
- Created a new variable that gave a combined measure of age and gender.
- For categorical variables, used “one-hot encoding”. This describes the process of transforming categorical predictors into dummy variables.
- Created a new variable with the lag between the first booking and the date the user signed up.
- “Summarized”secs_elapsed" and counted the numbers of rows of “sessions” dataset by user_id, action (similarly, action_type, action_detail and device_type)."

In total, created $>$1,300 features for the dataset.

Airbnb: Modeling

Stacked generalization: Stacked together 18 models and calculated out-of-bag cross-validation values. "The basic idea behind stacked generalization is to use a pool of base classifiers, then using another classifier to combine their predictions, with the aim of reducing the generalization error."
XGBoost model: Created with the base predictors as well as the out-of-bag predictions from the cross-validation (like SuperLearner)

Airbnb: Modeling

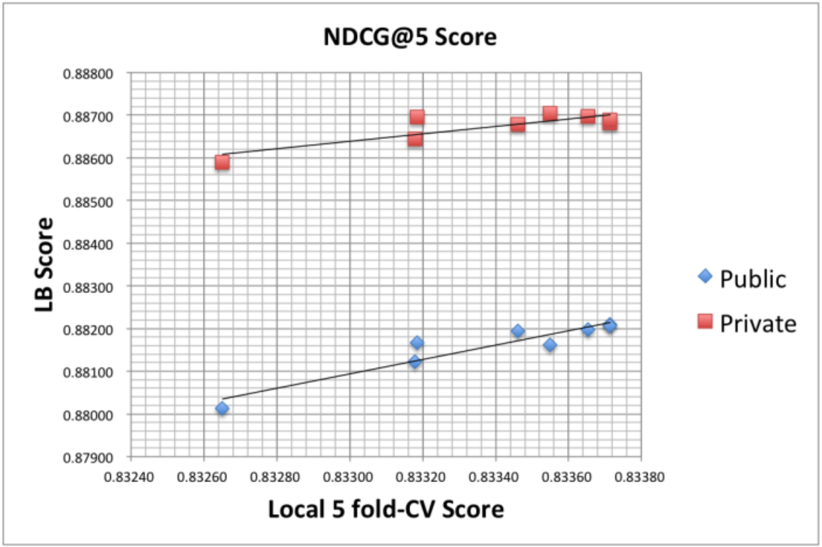


Airbnb: Modeling

Final stage modeling and final model selection:

> *"When I made several attempts to build it, I found that some features decreased the* `NDCG@5` *score, so I selected randomly features at the ratio of 90% and built repeatedly a single XGBoost many times. Finally, I selected the best XGBoost model (5 fold-CV: 0.833714) from the built models."*

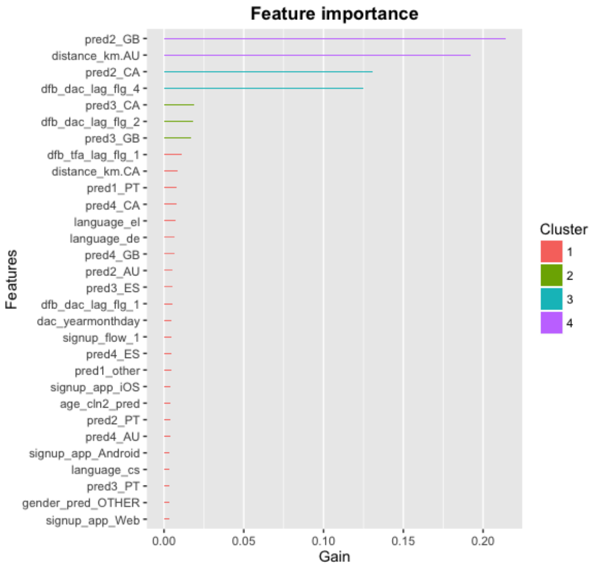
Airbnb: Modeling



Airbnb: Importance of engineering features

"I found that the out-of-fold CV predictions of categorized lag features were very important. As far as I saw in the forum, many of the participants may have not created these features."

Airbnb: Importance of engineering features



Airbnb: R packages

This person used used:

- `DescTools` for exploratory data analysis.
- `XGBoost`, `glmnet` for statistical modeling.

Airbnb: Third place

Third place winner: Sandro Vega Pons

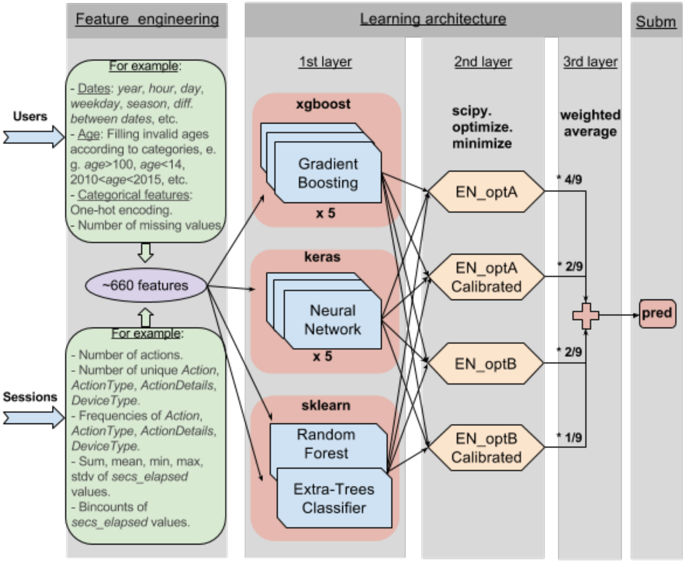
An interview on how he did it is available here.

Airbnb: Modeling

He used an ensemble of:

- GradientBoosting
- Multilayer perceptron (MLP)
- a RandomForest
- an Extra-Trees Classifier

Airbnb: Preprocessing and Modeling



Feature engineering
Learning architecture
Subm
Users
Sessions
For example:
- Dates: year, hour, day, weekday, season, diff. between dates, etc.
- Age: Filling invalid ages according to categories, e. g. age>100, age<14, 2010<age<2015, etc.
- Categorical features: One-hot encoding.
- Number of missing values
~660 features
For example:
- Number of actions.
- Number of unique Action, ActionType, ActionDetails, DeviceType.
- Frequencies of Action, ActionType, ActionDetails, DeviceType.
- Sum, mean, min, max, stdv of secs_elapsed values.
- Bincounts of secs_elapsed values.
1st layer
xgboost
Gradient Boosting
x 5
keras
Neural Network
x 5
sklearn
Random Forest
Extra-Trees Classifier
2nd layer
scipy. optimize. minimize
EN_optA
EN_optA Calibrated
EN_optB
EN_optB Calibrated
3rd layer
weighted average
* 4/9
* 2/9
* 2/9
* 1/9
pred

Airbnb: Python tools

- numpy
- pandas
- scipy
- xgboost
- keras
- scikit-learn

Airbnb: Conclusions

Helpfulness of using deep-learning models with tree-based models:

> *" I am always a bit surprised about how well deep learning models (like deep MLP) and tree-based models (like GradientBoosting) complement each other, and therefore how good they perform when ensembled. The combination of these two types of learning algorithms seems to be a recurrent choice in Kaggle competitions. "*

Airbnb: Conclusions

Advice for future competitions:

> *“Do not move forward to complex ideas before having a deep understanding of the evaluation measure and a reliable validation strategy. Depending on the competition, this can be trivial or very tricky.”*

XGBoost

XGBoost: eXtreme Gradient Boosting

"An efficient and scalable implementation of gradient boosting framework"

Includes solvers for:

- linear model
- tree

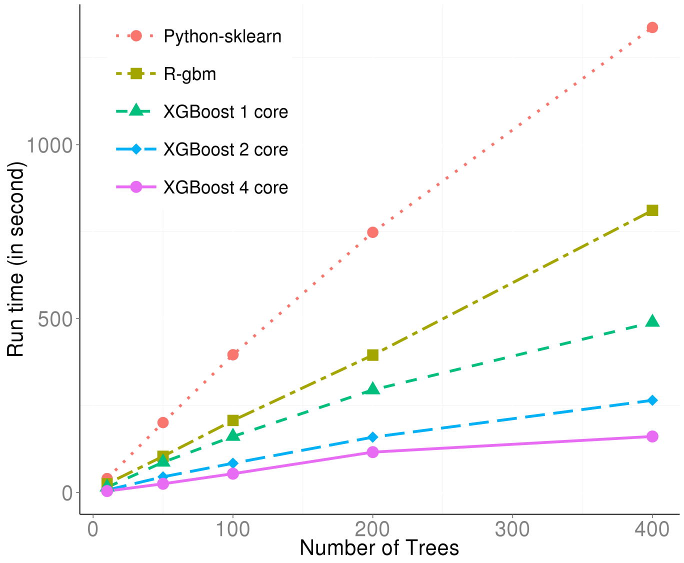
There is documentation in their GitHub repository.

XGBoost

Advantages:

- Speed: "Generally over 10 times faster than `gbm`" (parallel computation)
- Takes sparse matrices as inputs
- Allows customization (objective function, evaluation function)

XGBoost



Python-sklearn
R-gbm
XGBoost 1 core
XGBoost 2 core
XGBoost 4 core
Run time (in second)
0
500
1000
0
100
200
300
400
Number of Trees

XGBoost

The package includes a dataset with example data on classifying mushrooms as poisonous or not.

```r
library(xgboost)
data(agaricus.train)
data(agaricus.test)
train <- agaricus.train
test <- agaricus.test
head(colnames(train$data))
```

```
## [1] "cap-shape=bell"    "cap-shape=conical" "cap-shape=c
## [4] "cap-shape=flat"    "cap-shape=knobbed" "cap-shape=s
```

```r
dim(train$data)
```

```
## [1] 6513  126
```

XGBoost

This training dataframe is a sparse matrix:

```
train$data[1:5, 1:10]
```

```
## 5 x 10 sparse Matrix of class "dgCMatrix"

##     [[ suppressing 10 column names 'cap-shape=bell', 'cap

##
## [1,] . . 1 . . . . . . 1
## [2,] . . 1 . . . . . . 1
## [3,] 1 . . . . . . . . 1
## [4,] . . 1 . . . . . 1 .
## [5,] . . 1 . . . . . . 1
```

(The actual class of this is `dgCMatrix`.)

Sparse matrices

From the tutorial:

> *“In a sparse matrix, cells containing 0 are not stored in memory. Therefore, in a dataset mainly made of 0, memory size is reduced.”*

Training model

Training a decision model using XGBoost:

```
bstSparse <- xgboost(data = train$data,
                     label = train$label,
                     max.depth = 2, eta = 1,
                     nthread = 2, nround = 2,
                     objective = "binary:logistic")
```

```
## [0]  train-error:0.046522
## [1]  train-error:0.022263
```

Training model

Function arguments:

- `nrounds`: Maximum number of iterations
- `eta`: Controls the learning rate (can be used to help prevent overfitting; lower value is more robust to overfitting but takes longer to compute)
- `max.depth`: Maximum depth of a tree
- `nthread`: Number of parallel threads to use
- `nround`: How many passes of the data there will be (the more complex the relationship between your predictors and outcome, the more passes you'll need)

Prediction

To predict from an XGBoost model:

```
pred <- predict(bstSparse, test$data)
head(pred, 3)
```

```
## [1] 0.2858302 0.9239239 0.2858302
```

```
mean(as.numeric(pred > 0.5) == test$label)
```

```
## [1] 0.9782744
```

Measuring learning performance

xgboost can give you the performance of the model after each round of training. This can help you assess if you're doing too many rounds and starting to overfit.

```
watchlist <- list(train = train, test = test)
bst <- xgb.train(data = train, max.depth=2,
                 eta=1, nthread = 2, nround=2,
                 watchlist=watchlist,
                 objective = "binary:logistic")
```

```
## [0]  train-error:0.046522    test-error:0.042831
## [1]  train-error:0.022263    test-error:0.021726
```

Measuring learning performance

You can also change the evaluation metric when you do this:

```
bst <- xgb.train(data = train, max.depth=2,
                 eta=1, nthread = 2, nround=2,
                 watchlist=watchlist,
                 eval.metric = "error",
                 eval.metric = "logloss",
                 objective = "binary:logistic")
```

```
## [0]  train-error:0.046522    train-logloss:0.233357  tes
## [1]  train-error:0.022263    train-logloss:0.136649  tes
```

Early stopping

You can also use the argument `early.stop.round`. This will make the algorithm stop if you have had that many rounds without an improvement in performance.

You can also continue training on an existing model.