compboost

Fast and Flexible Way of bla

Daniel Schalk

July 7, 2019

LMU Munich
Working Group Computational Statistics

Use-Case

Use-Case

- We own a small booth at the city center that sells beer.
- As we are very interested in our customers' health, we only sell to customers who we expect to drink less than 110 liters per year.
- To estimate how much a customer drinks, we have collected data from 200 customers in recent years.
- These data include the beer consumption (in liter), age, sex, country of origin, weight, body size, and 200 characteristics gained from app usage (that have absolutely no influence).

Use-Case

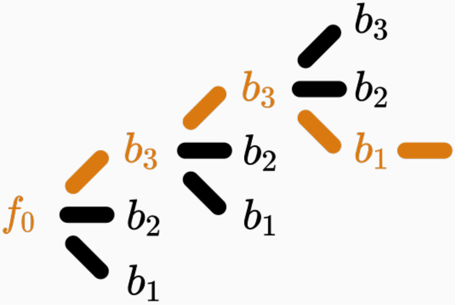
beer_consumption	gender	country	age	weight	height	app_usage1	app_usage2
106.5	m	Seychelles	33	87.17	172.9	0.1680	0.6063
85.5	f	Seychelles	52	89.38	200.4	0.8075	0.9376
116.5	f	Czechia	54	92.03	178.7	0.3849	0.2644
67.0	m	Australia	32	63.53	186.3	0.3277	0.3801
43.0	f	Australia	51	64.73	175.0	0.6021	0.8075
85.0	m	Austria	43	95.74	173.2	0.6044	0.9781
79.0	f	Austria	55	87.65	156.3	0.1246	0.9579
107.0	f	Austria	24	93.17	161.4	0.2946	0.7627
57.0	m	USA	55	76.27	182.5	0.5776	0.5096
89.0	m	USA	16	72.21	203.3	0.6310	0.0645

Use-Case

With these data we want to answer the following questions:

- Which of the customers' characteristics are important to be able to determine the consumption?
- How does the effect of important features look like?
- How does the model behave on unseen data?

What is Component-Wise Boosting?



$$f_3 = f_0 + \beta \sum_{(3,3,1)} b_i$$

- Inherent (unbiased) feature selection.
- Resulting model is sparse since important effects are selected first and therefore it is able to learn in high-dimensional feature spaces ($p \gg n$).
- Parameters are updated iteratively. Therefore, the whole trace of how the model evolves is available.

The Idea Behind Compboost

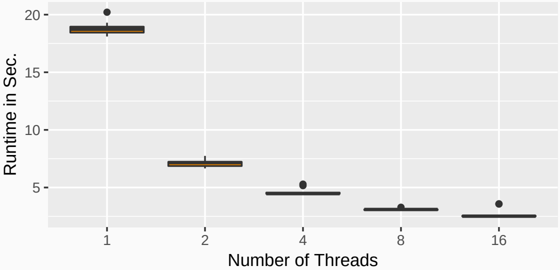
About Compboost

The `compboost` package is a fast and flexible framework for model-based boosting:

- With `mboost` as standard, we want to keep the modular principle of defining custom base-learner and losses.
- Completely written in `C++` and exposed by `Rcpp` to obtain high performance and full memory control.
- `R` API is written in `R6` to provide convenient wrapper.
- Major parts of the `compboost` functionality are unit tested against `mboost` to ensure correctness.

Runtime and Memory Considerations

- Matrices are stored (if possible) as sparse matrix.
- Take advantage of the matrix structure to speed up the algorithm by reducing the number of repetitive or too expensive calculations.
- Optimizer are parallelized via openmp:



A Short Demonstration

Using Convenience Wrapper

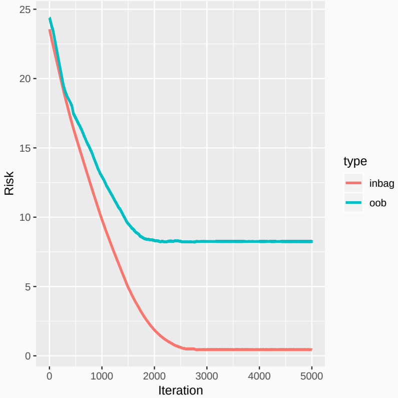
```r
set.seed(618)
cboost = boostSplines(data = beer_data, target = "beer_consumption",
  loss = LossAbsolute$new(), learning_rate = 0.1, iterations = 5000L,
  penalty = 10, oob_fraction = 0.3, trace = 2500L)
```

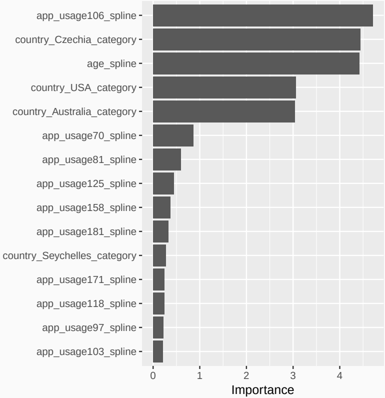
```
##    1/5000   risk = 24  oob_risk = 24
## 2500/5000   risk = 0.6  oob_risk = 8.3
## 5000/5000   risk = 0.44  oob_risk = 8.3
##
##
## Train 5000 iterations in 186 Seconds.
## Final risk based on the train set: 0.44
```

Visualizing Results

```
gg1 = cboost$plotInbagVsOobRisk()
gg2 = cboost$plotFeatureImportance()

gridExtra::grid.arrange(gg1, gg2, ncol = 2L)
```



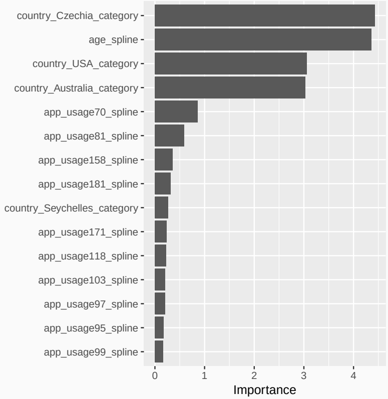


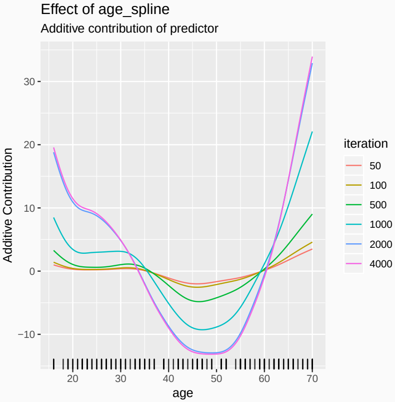
Visualizing Results

```
cboost$train(2000L)

gg1 = cboost$plotFeatureImportance()
gg2 = cboost$plot("age_spline", iters = c(50, 100, 500, 1000, 2000, 4000))

gridExtra::grid.arrange(gg1, gg2, ncol = 2L)
```





Using the R6 Interface

```
cboost = Compboost$new(data = beer_data, target = "beer_consumption",
  loss = LossQuantile$new(0.9), learning_rate = 0.1, oob_fraction = 0.3)

cboost$addBaselearner("age", "spline", BaselearnerPSpline)
cboost$addBaselearner("country", "category", BaselearnerPolynomial)

cboost$addLogger(logger = LoggerTime, use_as_stopper = TRUE, logger_id = "time",
  max_time = 2L, time_unit = "seconds")

cboost$train(10000, trace = 500)


##     1/10000   risk = 11  oob_risk = 10   time = 0
##   500/10000   risk = 7.9  oob_risk = 8.2   time = 0
##  1000/10000   risk = 6.3  oob_risk = 6.6   time = 0
##  1500/10000   risk = 5.1  oob_risk = 5.4   time = 0
##  2000/10000   risk = 4.2  oob_risk = 4.5   time = 1
##  2500/10000   risk = 3.5  oob_risk = 3.8   time = 1
##  3000/10000   risk = 3.2  oob_risk = 3.5   time = 1
##
##
## Train 3157 iterations in 2 Seconds.
## Final risk based on the train set: 3
```

Functionality Overview

- Base-learner
- Loss functions
- Logger/Stopper
- Custom loss function and base-learner via `R` or `C++` without recompiling the package

What's Next?

What's Next?

- Research on computational aspects of the algorithm:
  - More stable base-learner selection process via resampling
  - Base-learner selection for arbitrary performance measures
  - Smarter and faster optimizers
- Greater functionality:
  - Functional data structures and loss functions
  - Unbiased feature selection
  - Effect decomposition into constant, linear, and non-linear
- Reducing the memory load by applying binning to numerical features.
- Python API

Questions?