eXtreme Gradient Boosted Trees Training (xGBoost)

Dataset specific hyper-parameter tuning for min_n, tree_depth, learn_rate, and loss reduction

Amos Okutse

2022-12-22

```
## data sets for tuning each scenario and use in the treatment effects estimation

rm(list = ls())
## load the saved single data files
load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\df_one.RData")
load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\df_two.RData")
load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\df_three.RData")
load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\df_four.RData")


## load the saved list data files
#load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\dsets1.RData")
#load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\dsets2.RData")
#load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\dsets3.RData")
#load("C:\\Users\\aokutse\\OneDrive - Brown University\\ThesisResults\\data\\dsets4.RData")
```

Introduction

- explain the model, how it works, and the loss function/objective function
- why xgboost model rather than light gbm

XGBoost is an optimized Gradient Boosting Machine Learning library. It is originally written in C++, but has API in several other languages. The core XGBoost algorithm is parallelizable i.e. it does parallelization within a single tree. There are some of the cons of using XGBoost:

It is one of the most powerful algorithms with high speed and performance. It can harness all the processing power of modern multicore computers. It is feasible to train on large datasets. Consistently outperform all single algorithm methods.

- Hyper-parameter tuning for the XGBoost model

We are using the `dails::grid_max_entropy()` function which covers the hyperparameter space such that any portion of the space has an observed combination that is not too far from it.

To tune our model, we perform grid search over our xgboost_grid's grid space to identify the hyperparameter values that have the lowest prediction error.

tune_grid() performed grid search over all our 30 grid parameter combinations defined in xgboost_grid and used 10 fold cross validation along with rmse (Root Mean Squared Error), rsq (R Squared), and mae (Mean

Absolute Error) to measure prediction accuracy. So our tidymodels tuning just fit 60 X 5 = 300 XGBoost models each with 1,000 trees all in search of the optimal hyperparameters.

- Tuning procedure adapted from the blog: https://www.r-bloggers.com/2020/05/using-xgboost-with-tidymodels/ and the blog https://juliasilge.com/blog/xgboost-tune-volleyball/

HYPER-PARAMETER TUNING FOR FULL DATA ANALYSIS

Case 1 [n = 500 and SD = 1]

```
## data
train <- df_one[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()#,  ## ctrls model complexity
    #mtry = tune(),  ## predictors to be randomly sampled at a node
    #sample_size = tune(),  ## randomness; prop sampled for training
    #stop_iter = tune()  ## # of iterations without improvement before stopping
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
24	10	0.0000000	0.0000133
17	7	0.0000001	0.0000000
20	12	0.0000000	0.8151923
11	13	0.0045485	6.5373590
3	3	0.0126220	0.0000084
8	9	0.0205246	0.0000120

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
8	9	0.0205246	1.2e-05	Preprocessor1_Model06

Case 2 [n = 500, SD = 45]

```
## data
train <- df_two[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()  ## ctrls model complexity
```

```
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
```

```
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
23	1	0.0369178	2.9e-06	Preprocessor1_Model27

Case 3 [n = 2000, SD = 1]

```
## data
train <- df_three[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
```

```
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
22	9	0.045406	0.0179404	Preprocessor1_Model24

Case 4 [n = 2000, SD = 45]

```
## data
train <- df_four[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
```

```
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
33	3	0.0197791	0.0499346	Preprocessor1_Model18

HYPER-PARAMETER TUNING FOR OBSERVED DATA ANALYSIS

Case 1 [n = 500 and SD = 1]

```
## data
train1 <- base::subset(df_one, R == 1)
train = train1[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()#,  ## ctrls model complexity
    #mtry = tune(),  ## predictors to be randomly sampled at a node
    #sample_size = tune(),  ## randomness; prop sampled for training
    #stop_iter = tune()  ## # of iterations without improvement before stopping
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
```

```
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
8	8	0.0428784	2.37341	Preprocessor1_Model12

Case 2 [n = 500, SD = 45]

```
## data
train1 <- base::subset(df_two, R == 1)
train = train1[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()  ## ctrls model complexity

  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
```

```
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
8	8	0.0428784	2.37341	Preprocessor1_Model12

Case 3 [n = 2000, SD = 1]

```
## data
train1 <- base::subset(df_three, R == 1)
train = train1[, -7]
# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
```

```
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
8	8	0.0428784	2.37341	Preprocessor1_Model12

Case 4 [n = 2000, SD = 45]

```
## data
train1 <- base::subset(df_four, R == 1)
train = train1[, -7]

# XGBoost model specification
xgboost_model <-
  parsnip::boost_tree(
    mode = "regression",
    trees = 1000,
    min_n = tune(), ## then # of data points at a node before being split further
    tree_depth = tune(),  ## max depth of a tree
    learn_rate = tune(),  ## shrinkage parameter; step size
    loss_reduction = tune()
  ) %>%
    set_engine("xgboost", objective = "reg:squarederror", nthreads = parallel::detectCores()-1)


# grid specification for tuning the hyper-parameters
## first set-up the parameters to be tuned
xgboost_params <-
  dials::parameters(
    min_n(),
    tree_depth(),
    learn_rate(),
    loss_reduction()
  )
## then use the vector of parameters in the grid_max_entropy() call for parameter tuning
xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 30
  )
knitr::kable(head(xgboost_grid)) ## print out some of the grid values
```

min_n	tree_depth	learn_rate	loss_reduction
34	15	0.0000026	0.0000000
19	6	0.0000001	0.0002127
21	4	0.0000632	7.4781640
32	12	0.0018544	21.0148833
39	9	0.0138207	0.0000796
5	3	0.0000204	0.0620218

```
# define the workflow
xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(y ~ A + x1 + x2 + x3 + x4)

# set up the cross-validation folds
cv_folds <- vfold_cv(train, v = 10)

# hyper-parameter tuning via tune_grid()

doParallel::registerDoParallel()
set.seed(123)
xgboost_tuned <- tune::tune_grid(
  object = xgboost_wf,
  resamples = cv_folds,
  grid = xgboost_grid,
  metrics = yardstick::metric_set(rmse), ## can use mae and rsq
  control = tune::control_grid(verbose = TRUE)
)

# isolate the best performing model parameters
xgboost_best_params <- xgboost_tuned %>%
  tune::select_best("rmse")
knitr::kable(xgboost_best_params)
```

min_n	tree_depth	learn_rate	loss_reduction	.config
23	1	0.0369178	2.9e-06	Preprocessor1_Model27