L07 Model Tuning

Data Science II (STAT 301-2)

Austin Shinn

Overview

This lab covers material up to and including 13. Grid search from Tidy Modeling with R. In this lab, we start with a new data set and go through the entire modeling process – splitting the data and using repeated V-fold cross-validation to choose and tune a model.

This lab can serve as an example of the overall statistical learning process (that you will use for your final project). Your project should generally follow similar steps (although it may include more exploration of the training set, and comparing more types of models).

Load Packages & Set a Seed

```
# Load packages here!
library(tidymodels)
library(ranger)
library(kknn)
library(xgboost)
library(skimr)

# Set seed here!
set.seed(777)
```

Tasks

Task 1

For this lab, we will be working with a simulated data set, designed to accompany the book An Introduction to Statistical Learning with Applications in R. The data set consists of 400 observations about the sales of child car seats at different stores.

Our goal with this data is regression; specifically, to build **and tune** a model that can predict car seat sales as accurately as possible.

Load the data from `data/carseats.csv` into *R* and familiarize yourself with the variables it contains using the codebook (`data/carseats_codebook.txt`).

```
carseats <- read.csv("data/carseats.csv") %>%
  mutate(across(where(is.character), factor))
```

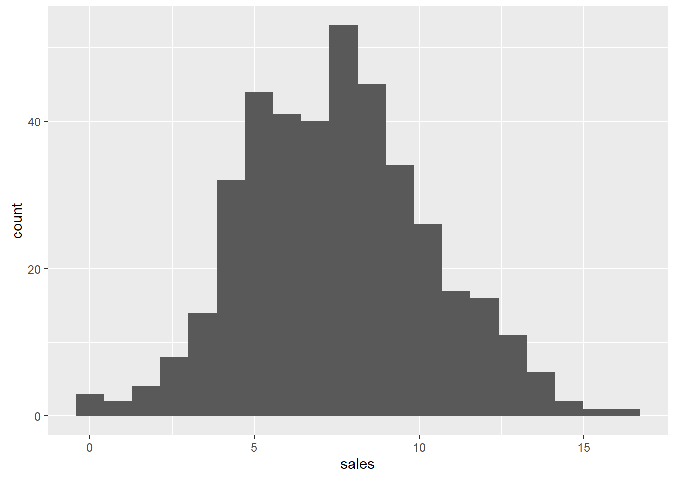
Task 2

Using the full data set, explore/describe the distribution of the outcome variable `sales`. Perform a quick `skim` of the data and note any potential problems (like missingness).

```
summary(carseats$sales)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   5.390   7.490   7.496   9.320  16.270
```

```
ggplot(carseats, aes(sales)) +
  geom_histogram(bins = 20)
```




```
skim(carseats)
```

Data summary

Name	carseats
Number of rows	400
Number of columns	11
_______________________	
Column type frequency:	
factor	3
numeric	8
________________________	
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
shelve_loc	0	1	FALSE	3	Med: 219, Bad: 96, Goo: 85
urban	0	1	FALSE	2	Yes: 282, No: 118
us	0	1	FALSE	2	Yes: 258, No: 142

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
sales	0	1	7.50	2.82	0	5.39	7.49	9.32	16.27	▁▅▇▂▁
comp_price	0	1	124.97	15.33	77	115.00	125.00	135.00	175.00	▁▅▇▂▁
income	0	1	68.66	27.99	21	42.75	69.00	91.00	120.00	▇▆▇▇▅
advertising	0	1	6.64	6.65	0	0.00	5.00	12.00	29.00	▇▃▃▁▁
population	0	1	264.84	147.38	10	139.00	272.00	398.50	509.00	▇▇▇▇▇
price	0	1	115.80	23.68	24	100.00	117.00	131.00	191.00	▁▂▇▅▁
age	0	1	53.32	16.20	25	39.75	54.50	66.00	80.00	▇▆▇▇▇
education	0	1	13.90	2.62	10	12.00	14.00	16.00	18.00	▇▇▁▇▇

No missing values

Task 3

Split the data! **Make sure to set a seed.** Use stratified sampling.

You should choose the proportions to split the data into. Verify that the training and testing data sets have the appropriate number of observations.

Then use V-fold cross-validation with 10 folds, repeated 5 times, to fold the **training** data.

```
initial <- initial_split(carseats, prop = .75, strata = sales)

carseats_train <- training(initial)
carseats_test <- testing(initial)

carseats_fold <- vfold_cv(carseats_train, v = 10, repeats = 5, strata = sales)
```

Task 4

Set up a recipe. The recipe should predict `sales` (the outcome) using all other variables in the data set. Add steps to your recipe to:

- one-hot encode all categorical predictors, &
- center and scale all predictors.

`prep()` and `bake()` your recipe on the training data. How many columns are there in the data after you've processed it? You'll need to use this number as an upper limit for possible values of `mtry`.

```
recipe <- carseats_train %>%
  recipe(sales ~ .) %>%
  step_dummy(all_nominal(), one_hot = TRUE) %>%
  step_normalize(all_predictors())

recipe %>%
  prep(training = carseats_train) %>%
  bake(new_data = NULL) %>%
  view()

save(carseats_fold, recipe, initial, file = "data/save.rda")
```

Task 5

We will train and tune **three competing model types**:

1. A random forest model (`rand_forest()`) with the `ranger` engine;
2. A boosted tree model (`boost_tree()`) with the `xgboost` engine;
3. A *k*-nearest neighbors model (`nearest_neighbors()`) with the `kknn` engine.

Hint: Ensure engine packages are installed.

Set up and store each of these three models with the appropriate function and engine.

For the random forest model, we will tune the hyper-parameters `mtry` and `min_n`. For the boosted tree model, we will tune `mtry`, `min_n`, and `learn_rate`. For the *k*-nearest neighbors model, we will tune `neighbors`. **When you set up these models, you should flag these parameters for tuning with `tune()`.**

```
rf_model <- rand_forest(mode = "regression",
                        min_n = tune(),
                        mtry = tune()) %>%
  set_engine("ranger")

bt_model <- boost_tree(mode = "regression",
                       mtry = tune(),
                    min_n = tune(),
                    learn_rate = tune()) %>%
  set_engine("xgboost")

knn_model <- nearest_neighbor(mode = "regression",
                              neighbors = tune()) %>%
  set_engine("kknn")
```

Task 6

Now we will set up and store **regular grids** with 5 levels of possible values for tuning hyper-parameters for each of the three models.

```
rf_params <- parameters(rf_model) %>%
  update(mtry = mtry(c(2,10)))

rf_grid <- grid_regular(rf_params, levels = 5)

bt_params <- parameters(bt_model) %>%
  update(mtry = mtry(c(2,10)),
         learn_rate = learn_rate(c(-5, -.2)))

bt_grid <- grid_regular(bt_params, levels = 5)

knn_params <- parameters(knn_model)

knn_grid <- grid_regular(knn_params, levels = 5)
```

The parameters `min_n` and `neighbors` have default tuning values should work reasonably well, so we won't need to update their defaults manually. For `mtry`, we will need to use `update()` (as shown above) to change the upper limit value to the number of predictor columns. For `learn_rate`, we will also use `update()`, this time to set `range = c(-5, -0.2)`.

Task 7

Print one of the grid tibbles that you created in Task 6 and explain what it is in your own words. Why are we creating them?

```
rf_grid
```

```
## # A tibble: 25 x 2
##     mtry min_n
##    <int> <int>
##  1     2     2
##  2     4     2
##  3     6     2
##  4     8     2
##  5    10     2
##  6     2    11
##  7     4    11
##  8     6    11
##  9     8    11
## 10    10    11
## # ... with 15 more rows
```

This grid tibble is a grid of potential values of mtry and min_n to try.

Task 8

For each of our 3 competing models (random forest, boosted tree, and knn), set up a workflow, add the appropriate model from Task 5, and add the recipe we created in Task 4.

```
rf_workflow <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(recipe)

bt_workflow <- workflow() %>%
  add_model(bt_model) %>%
  add_recipe(recipe)

knn_workflow <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(recipe)
```

Task 9

Here's the fun part, where we get to tune the parameters for these models and find the values that optimize model performance across folds!

Take each of your three workflows from Task 8. Pipe each one into `tune_grid()`. Supply your folded data and the appropriate grid of parameter values as arguments to `tune_grid()`.

```
rf_tune <- rf_workflow %>%
  tune_grid(resamples = carseats_fold,
            grid = rf_grid)

bt_tune <- bt_workflow %>%
  tune_grid(resamples = carseats_fold,
            grid = bt_grid)

knn_tune <- knn_workflow %>%
  tune_grid(resamples = carseats_fold,
            grid = knn_grid)

save(rf_tune, bt_tune, knn_tune, file = "data/carseats_tune_grids.rda")
```

WARNING: STORE THE RESULTS OF THIS CODE. You will NOT want to re-run this code each time you knit your .Rmd. We strongly recommend running it in an .R script and storing the results for each model with `write_rds()` or similar. You may also want to use RStudio's jobs functionality (but are not required to).

Task 10

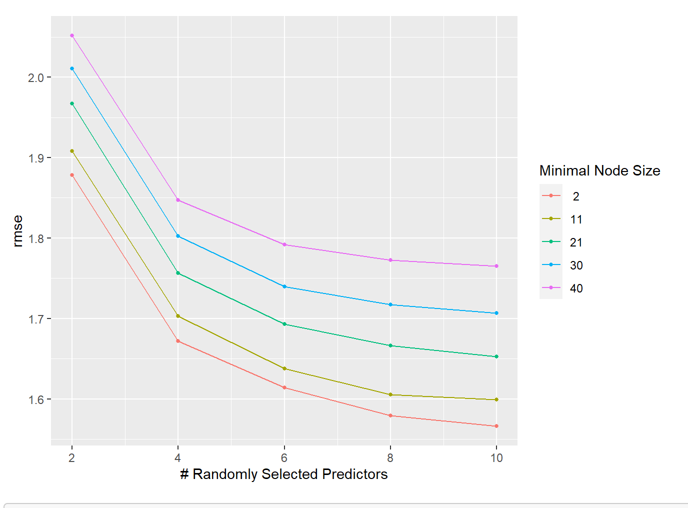
Let's check out the results!

Use `autoplot()` on each of the objects you stored in Task 9. Set the `metric` argument of `autoplot()` to `"rmse"` for each. (Otherwise it will produce plots for R^2 as well – doesn't hurt, but we're interested in RMSE.)

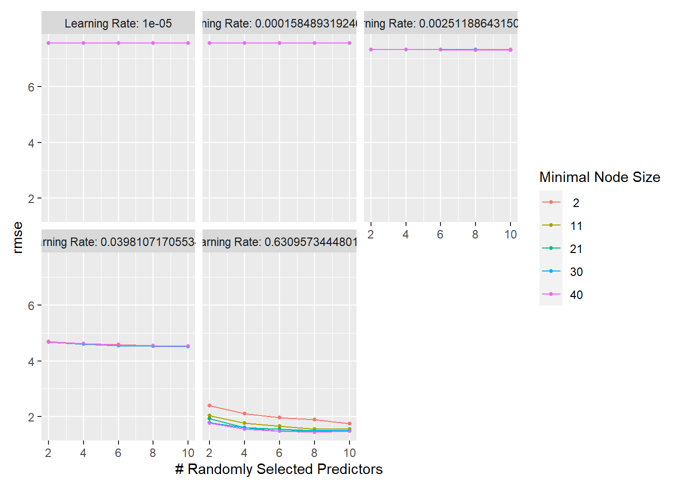
Pick one of the three `autoplot()`s you've produced and describe it in your own words. What happens to the RMSE as the values of the tuning parameters change?

Example for random forest:

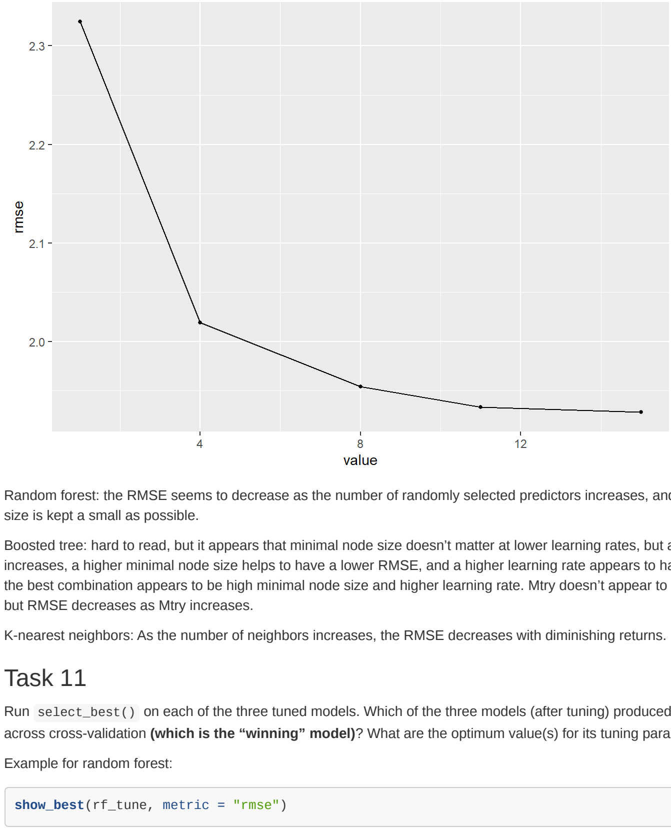
```
load("data/carseats_tune_grids.rda")

autoplot(rf_tune, metric = "rmse")
```




```
autoplot(bt_tune, metric = "rmse")
```




```
autoplot(knn_tune, metric = "rmse")
```



Random forest: the RMSE seems to decrease as the number of randomly selected predictors increases, and the minimal node size is kept a small as possible.

Boosted tree: hard to read, but it appears that minimal node size doesn't matter at lower learning rates, but as the learning rate increases, a higher minimal node size helps to have a lower RMSE, and a higher learning rate appears to have a lower RMSE. So the best combination appears to be high minimal node size and higher learning rate. Mtry doesn't appear to have a large effect, but RMSE decreases as Mtry increases.

K-nearest neighbors: As the number of neighbors increases, the RMSE decreases with diminishing returns.

Task 11

Run `select_best()` on each of the three tuned models. Which of the three models (after tuning) produced the smallest RMSE across cross-validation **(which is the "winning" model)**? What are the optimum value(s) for its tuning parameters?

Example for random forest:

```
show_best(rf_tune, metric = "rmse")
```

```
## # A tibble: 5 x 8
##    mtry min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    10     2 rmse    standard    1.57    50  0.0285 Preprocessor1_Model05
## 2     8     2 rmse    standard    1.58    50  0.0297 Preprocessor1_Model04
## 3    10    11 rmse    standard    1.60    50  0.0294 Preprocessor1_Model10
## 4     8    11 rmse    standard    1.61    50  0.0296 Preprocessor1_Model09
## 5     6     2 rmse    standard    1.61    50  0.0294 Preprocessor1_Model03
```

```
show_best(bt_tune, metric = "rmse")
```

```
## # A tibble: 5 x 9
##    mtry min_n learn_rate .metric .estimator  mean     n std_err .config
##   <int> <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     8    40      0.631 rmse    standard    1.44    50  0.0280 Preprocessor1_M~
## 2    10    40      0.631 rmse    standard    1.47    50  0.0281 Preprocessor1_M~
## 3     6    40      0.631 rmse    standard    1.48    50  0.0313 Preprocessor1_M~
## 4    10    21      0.631 rmse    standard    1.49    50  0.0283 Preprocessor1_M~
## 5     6    30      0.631 rmse    standard    1.49    50  0.0321 Preprocessor1_M~
```

```
show_best(knn_tune, metric = "rmse")
```

```
## # A tibble: 5 x 7
##   neighbors .metric .estimator  mean     n std_err .config
##       <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1        15 rmse    standard    1.93    50  0.0348 Preprocessor1_Model5
## 2        11 rmse    standard    1.93    50  0.0341 Preprocessor1_Model4
## 3         8 rmse    standard    1.95    50  0.0338 Preprocessor1_Model3
## 4         4 rmse    standard    2.02    50  0.0328 Preprocessor1_Model2
## 5         1 rmse    standard    2.32    50  0.0361 Preprocessor1_Model1
```

Random forest: 10 mtry, 2 min_n

Boosted tree: 8 mtry, 40 minimum node size, .631 learn rate

K-nearest neighbors: 15 neighbors

Boosted tree appears to have the lowest RMSE, with standard error similar to the other models, so it looks to be the "winner" here.

Task 12

We've now used 10-fold cross-validation (with 5 repeats) to tune three competing models – a random forest, a boosted tree, and a KNN model. You've selected the "winning" model and learned the optimal values of its tuning parameters to produce the lowest RMSE on assessment data across folds.

Now we can **use the winning model and the tuning values to fit the model to the entire training data set**.

```
bt_workflow_tuned <- bt_workflow %>%
  finalize_workflow(select_best(bt_tune, metric = "rmse"))

bt_results <- fit(bt_workflow_tuned, carseats_train)
```

Task 13

Finally, at long last, we can use the **testing data set** that we set aside in Task 3!

Use `predict()`, `bind_cols()`, and `metric_set()` to fit your tuned model to the testing data.

Example, if the random forest performed best:

```
carseat_metric <- metric_set(rmse)

predict(bt_results, new_data = carseats_test) %>%
  bind_cols(carseats_test %>% select(sales)) %>%
  carseat_metric(truth = sales, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        1.62
```

Task 14

How did your model do on the brand-new, untouched testing data set? Is the RMSE it produced on the testing data similar to the RMSE estimate you saw while tuning?

The RMSE is a bit higher, but not much too much off from the RMSE we got from the training data.