Applications

Of Data Science

Modeling in the Tidyverse

Applications of Data Science - Class 5

Giora Simchoni

gsimchoni@gmail.com and add #dsapps in subject

Stat. and OR Department, TAU

2023-02-24

The Problem

Inconsistency, Inextensibility

```
n <- 10000
x1 <- runif(n)
x2 <- runif(n)
t <- 1 + 2 * x1 + 3 * x2
y <- rbinom(n, 1, 1 / (1 + exp(-t)))
```

```
glm(y ~ x1 + x2, family = "binomial")
```

```
glmnet(as.matrix(cbind(x1, x2)), as.factor(y), family = "binomial'
```

```
randomForest(as.factor(y) ~ x1 + x2)
```

```
gbm(y ~ x1 + x2, data = data.frame(x1 = x1, x2 = x2, y = y))
```

Compare this with `sklearn`

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
  GradientBoostingClassifier

LogisticRegression(penalty='none').fit(X, y)

LogisticRegression(penalty='l2', C=0.001).fit(X, y)

RandomForestClassifier(n_estimators=100).fit(X, y)

GradientBoostingClassifier(n_estimators=100).fit(X, y)
```

Detour: A Regression Problem

Hungarian Blogs: Predicting Feedback

- Dataset was published as part of the UCI ML Repository initiative
- Comes from Buza 2014
- 280 numeric heavily engineered features on blogs and posts published in the last 72 hours
- Can we predict no. of comments in the next 24 hours?

The raw data has over 50K rows: for each blog features like total comments until base time, weekday, words, etc.

We will be predicting `log(fb)` based on all features, no missing values:

```
blogs_fb <- read_csv("~/BlogFeedback/blogData_train.csv", col_name

blogs_fb <- blogs_fb %>%
  rename(fb = X281, blog_len = X62, sunday = X276) %>%
  mutate(fb = log(fb + 1))

dim(blogs_fb)
```

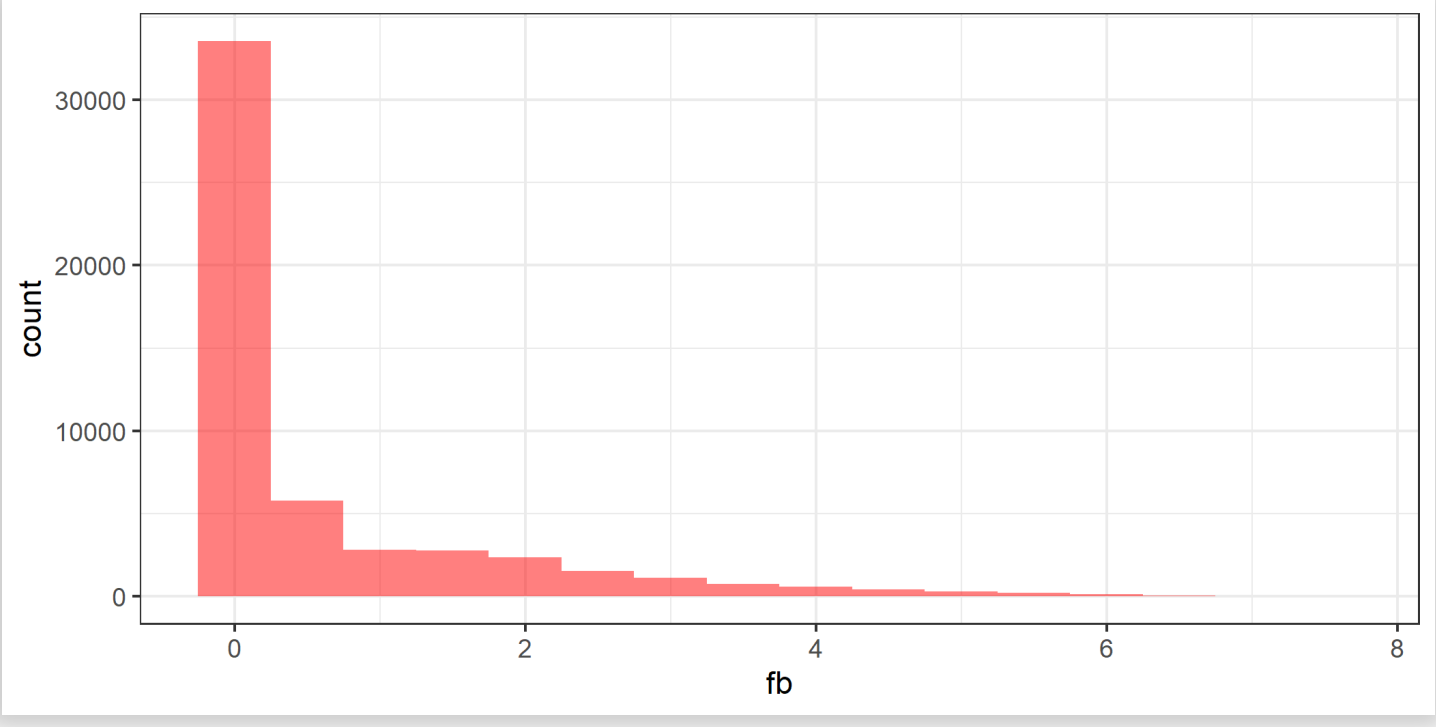
```
## [1] 52397   281
```

```
glimpse(blogs_fb)
```

```
## Rows: 52,397
## Columns: 281
## $ X1       <dbl> 40.30467, 40.30467, 40.30467, 40.30467, 40.30467, 40.30
## $ X2       <dbl> 53.84566, 53.84566, 53.84566, 53.84566, 53.84566, 53.84
## $ X3       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
## $ X4       <dbl> 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401,
## $ X5       <dbl> 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
## $ X6       <dbl> 15.52416, 15.52416, 15.52416, 15.52416, 15.52416, 15.52
## $ X7       <dbl> 32.44188, 32.44188, 32.44188, 32.44188, 32.44188, 32.44
## $ X8       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
## $ X9       <dbl> 377, 377, 377, 377, 377, 377, 377, 377, 377, 377, 377,
## $ X10      <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
## $ X11      <dbl> 14.04423, 14.04423, 14.04423, 14.04423, 14.04423, 14.04
## $ X12      <dbl> 32.61542, 32.61542, 32.61542, 32.61542, 32.61542, 32.61
## $ X13      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
## $ X14      <dbl> 377, 377, 377, 377, 377, 377, 377, 377, 377, 377, 377,
## $ X15      <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
## $ X16      <dbl> 34.56757, 34.56757, 34.56757, 34.56757, 34.56757, 34.56
## $ X17      <dbl> 48.47518, 48.47518, 48.47518, 48.47518, 48.47518, 48.47
## $ X18      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
## $ X19      <dbl> 378, 378, 378, 378, 378, 378, 378, 378, 378, 378, 378,
## $ X20      <dbl> 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
## $ X21      <dbl> 1.479934, 1.479934, 1.479934, 1.479934, 1.479934, 1.479
## $ X22      <dbl> 46.18691, 46.18691, 46.18691, 46.18691, 46.18691, 46.18
## $ X23      <dbl> -356, -356, -356, -356, -356, -356, -356, -356, -356, -
## $ X24      <dbl> 377, 377, 377, 377, 377, 377, 377, 377, 377, 377, 377,
## $ X25      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

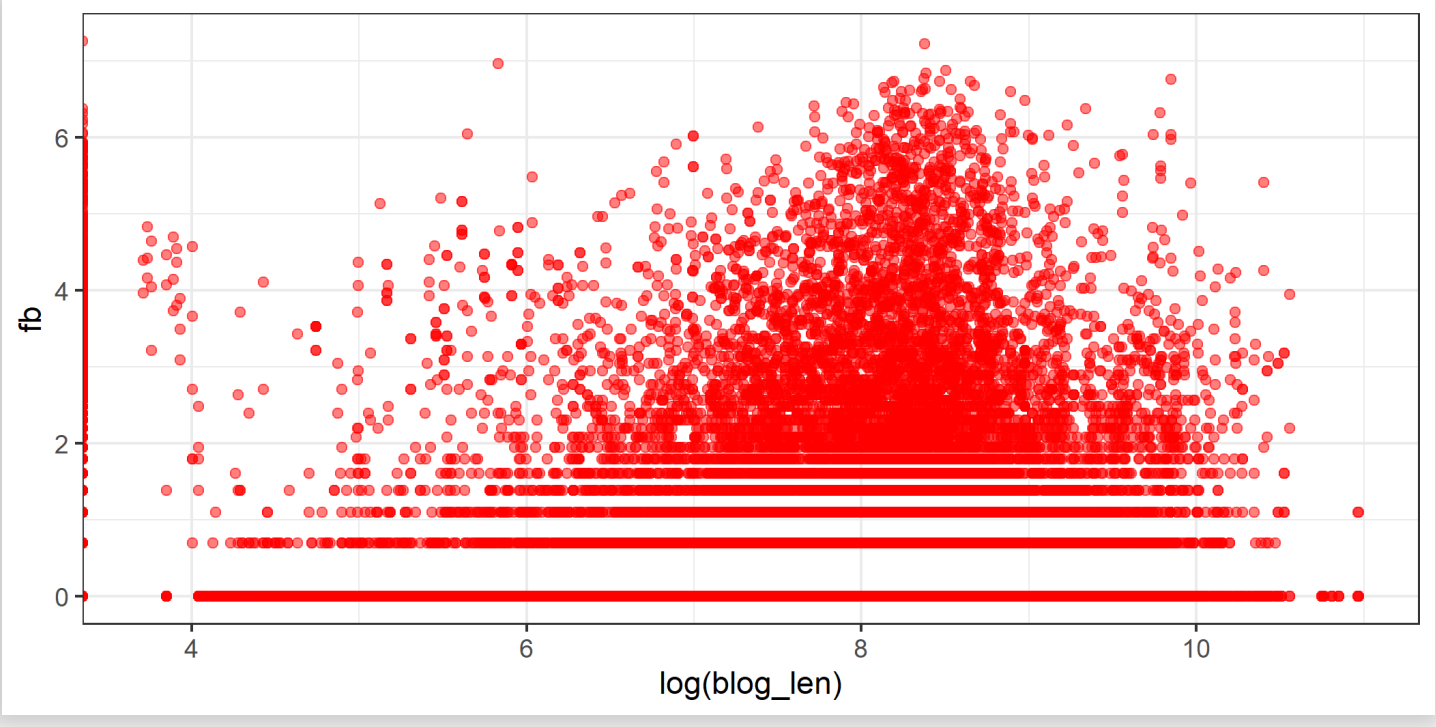
See the dependent variable distribution:

```
ggplot(blogs_fb, aes(fb)) +
  geom_histogram(fill = "red", alpha = 0.5, binwidth = 0.5) +
  theme_bw()
```



See it vs. say "length of post":

```
ggplot(blogs_fb, aes(log(blog_len), fb)) +
  geom_point(color = "red", alpha = 0.5) +
  theme_bw()
```



End of Detour

WARNING

💡 What you're about to see is not a good modeling/prediction flow.

This is just an intro to tidy modeling.

Some of the issues with how things are done here will be raised, some will have to wait till later in the course.

The ~~Present~~ Past Solution: `caret`

Split Data

```
library(caret)

train_idx <- createDataPartition(blogs_fb$fb,
                                 p = 0.6, list = FALSE)

blogs_tr <- blogs_fb[train_idx, ]
blogs_te <- blogs_fb[-train_idx, ]

library(glue)
glue("train no. of rows: {nrow(blogs_tr)}
     test no. of rows: {nrow(blogs_te)}")
```

```
## train no. of rows: 31439
## test no. of rows: 20958
```

Here you might consider some preprocessing.

`caret` has some nice documentation here.

Tuning and Modeling

Define general methodology, e.g. 5-fold Cross-Validation:

```
fit_control <- trainControl(method = "cv", number = 5)

ridge_grid <- expand.grid(alpha=0, lambda = 10^seq(-3, 1, length =
lasso_grid <- expand.grid(alpha=1, lambda = 10^seq(-3, 1, length =
rf_grid <- expand.grid(splitrule = "variance",
                       min.node.size = seq(10, 30, 10),
                       mtry = seq(10, 50, 20))

mod_ridge <- train(fb ~ ., data = blogs_tr, method = "glmnet",
                   trControl = fit_control, tuneGrid = ridge_grid,
                   metric = "RMSE")

mod_lasso <- train(fb ~ ., data = blogs_tr, method = "glmnet",
                   trControl = fit_control, tuneGrid = lasso_grid,
                   metric = "RMSE")

mod_rf <- train(fb ~ ., data = blogs_tr, method = "ranger",
                   trControl = fit_control, tuneGrid = rf_grid,
                   num.trees = 50, metric = "RMSE")
```

Evaluating Models

```
mod_ridge
```

```
## glmnet
##
## 31439 samples
##   280 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 25151, 25152, 25151, 25151, 25151
## Resampling results across tuning parameters:
##
##   lambda        RMSE       Rsquared   MAE
##    0.001000000  0.8462670  0.4467763  0.5891955
##    0.001206793  0.8462670  0.4467763  0.5891955
##    0.001456348  0.8462670  0.4467763  0.5891955
##    0.001757511  0.8462670  0.4467763  0.5891955
##    0.002120951  0.8462670  0.4467763  0.5891955
##    0.002559548  0.8462670  0.4467763  0.5891955
##    0.003088844  0.8462670  0.4467763  0.5891955
##    0.003727594  0.8462670  0.4467763  0.5891955
##    0.004498433  0.8462670  0.4467763  0.5891955
##    0.005428675  0.8462670  0.4467763  0.5891955
##    0.006551286  0.8462670  0.4467763  0.5891955
##    0.007906043  0.8462670  0.4467763  0.5891955
```

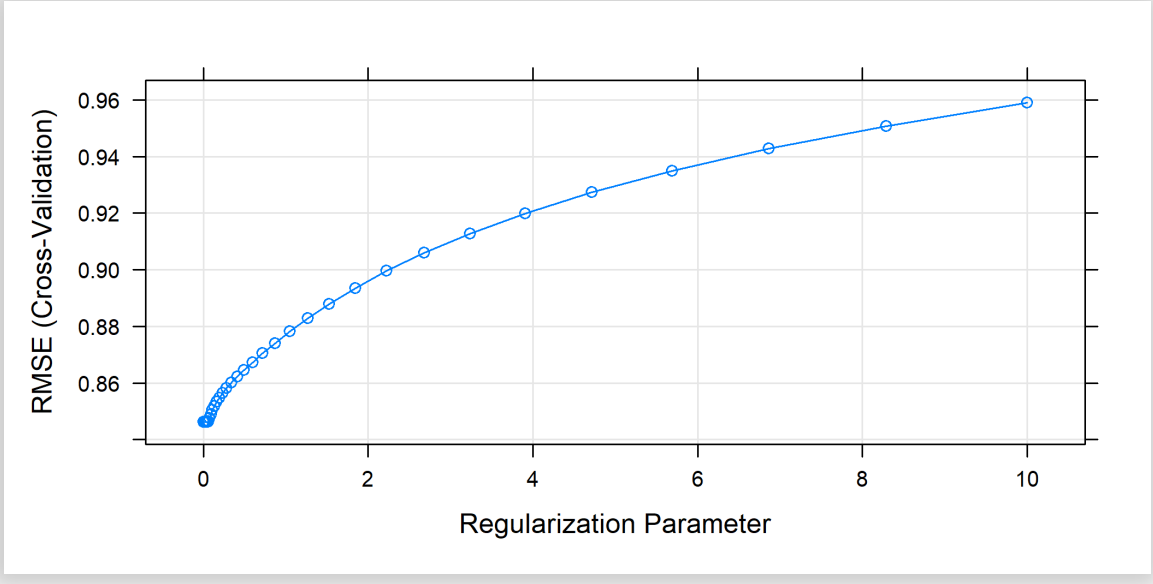
```
mod_lasso
```

```
## glmnet
##
## 31439 samples
##   280 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 25152, 25150, 25151, 25152, 25151
## Resampling results across tuning parameters:
##
##   lambda        RMSE       Rsquared   MAE
##    0.001000000  0.8333364  0.4630729  0.5774531
##    0.001206793  0.8338379  0.4624286  0.5778514
##    0.001456348  0.8344579  0.4616355  0.5784054
##    0.001757511  0.8354024  0.4604269  0.5792853
##    0.002120951  0.8367088  0.4587510  0.5804684
##    0.002559548  0.8379595  0.4571494  0.5815771
##    0.003088844  0.8395353  0.4551339  0.5828745
##    0.003727594  0.8416496  0.4524126  0.5845320
##    0.004498433  0.8434856  0.4500571  0.5857410
##    0.005428675  0.8458443  0.4470098  0.5873684
##    0.006551286  0.8481986  0.4439628  0.5890618
##    0.007906043  0.8502710  0.4412905  0.5905821
##    0.009540955  0.8530251  0.4376999  0.5925616
##    0.011513954  0.8553405  0.4346936  0.5942513
##    0.013894955  0.8575749  0.4318132  0.5959307
##    0.016768329  0.8588602  0.4302636  0.5971247
```

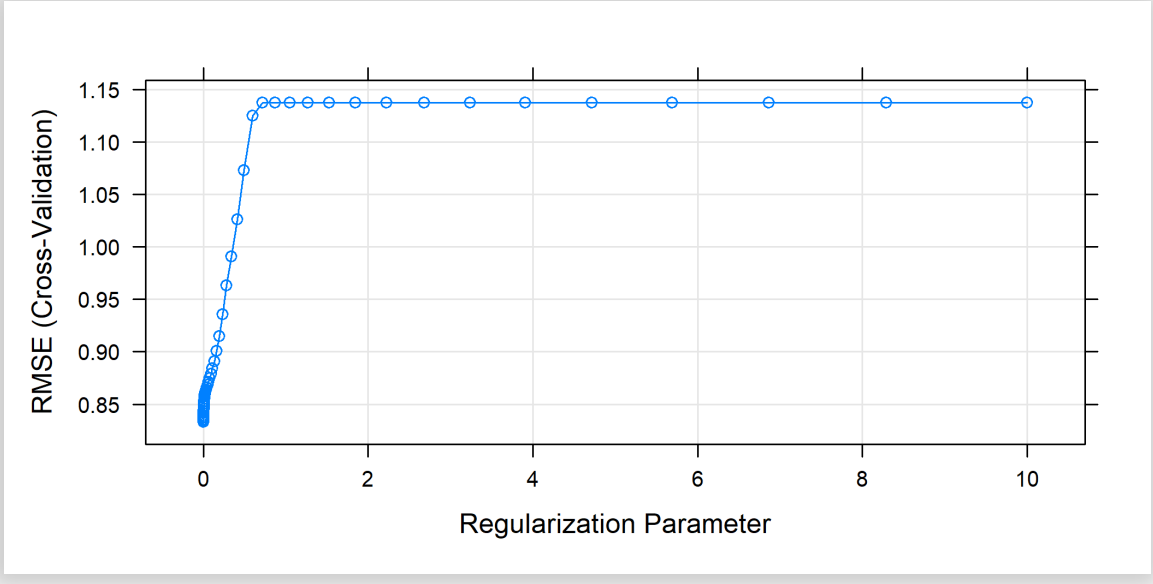
```
mod_rf
```

```
## Random Forest
##
## 31439 samples
##   280 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 25151, 25152, 25152, 25151, 25150
## Resampling results across tuning parameters:
##
##   min.node.size  mtry  RMSE       Rsquared   MAE
##   10             10    0.6982353  0.6300362  0.4477895
##   10             30    0.6565384  0.6679521  0.4090834
##   10             50    0.6493738  0.6744561  0.4023975
##   20             10    0.7057277  0.6227657  0.4538613
##   20             30    0.6616067  0.6633432  0.4160850
##   20             50    0.6511654  0.6730767  0.4062798
##   30             10    0.7110096  0.6177993  0.4593662
##   30             30    0.6652902  0.6597730  0.4189310
##   30             50    0.6539612  0.6703397  0.4090870
##
## Tuning parameter 'splitrule' was held constant at a value of variance
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were mtry = 50, splitrule = variance
##  and min.node.size = 10.
```

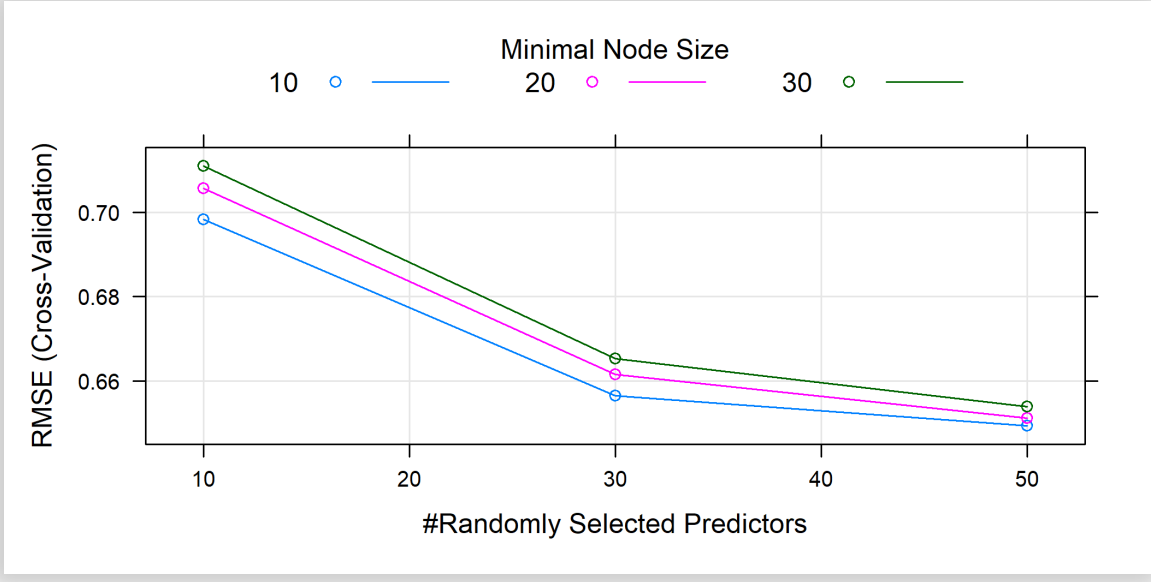
```
plot(mod_ridge)
```



```
plot(mod_lasso)
```



```
plot(mod_rf)
```

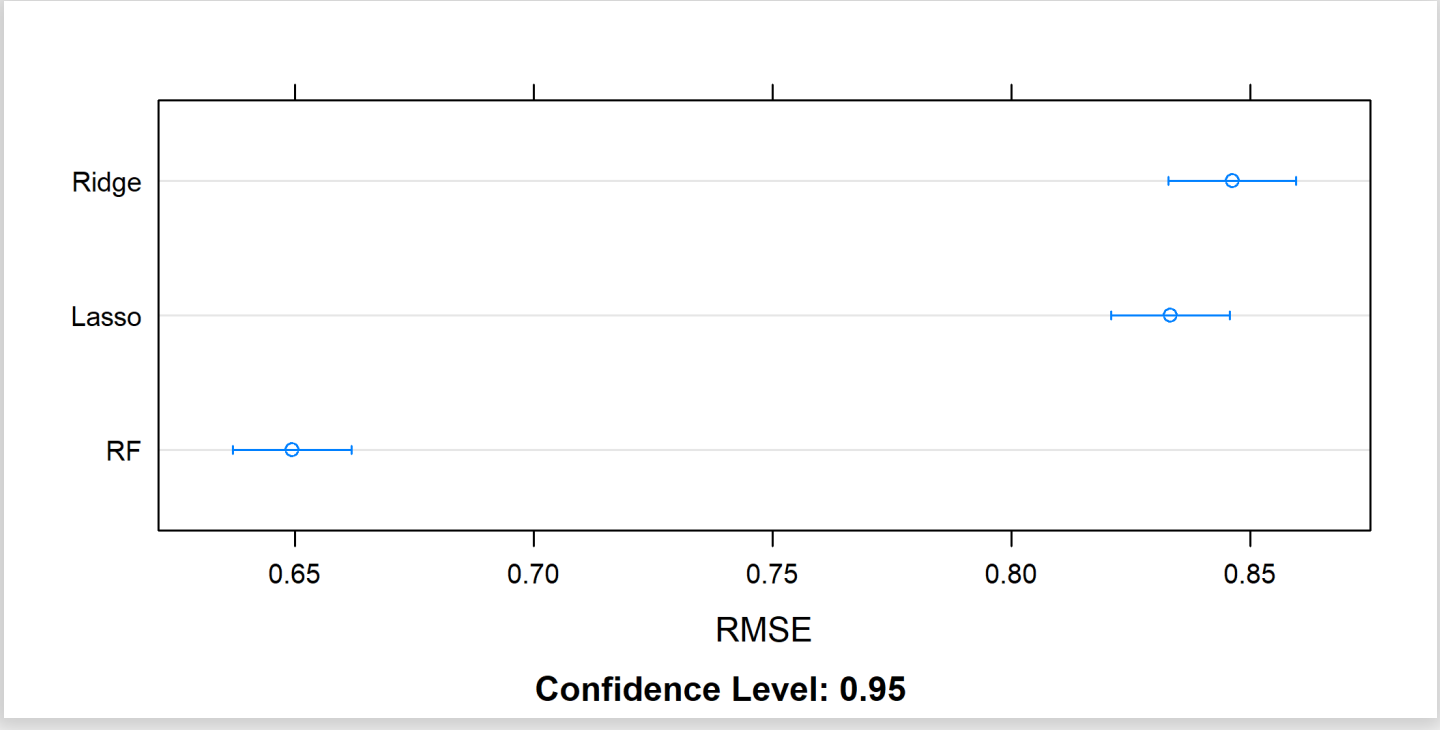

Minimal Node Size
10
20
30
RMSE (Cross-Validation)
0.70
0.68
0.66
10
20
30
40
50
#Randomly Selected Predictors

Comparing Models

```
resamps <- resamples(list(Ridge = mod_ridge, Lasso = mod_lasso,
                          RF = mod_rf))
summary(resamps)
```

```
##
## Call:
## summary.resamples(object = resamps)
##
## Models: Ridge, Lasso, RF
## Number of resamples: 5
##
## MAE
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## Ridge 0.5836974 0.5897172 0.5901640 0.5891955 0.5910708 0.5913280    0
## Lasso 0.5721159 0.5744051 0.5752703 0.5774531 0.5815676 0.5839066    0
## RF    0.3935699 0.4009682 0.4022208 0.4023975 0.4034971 0.4117315    0
##
## RMSE
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## Ridge 0.8325399 0.8412139 0.8439569 0.8462670 0.8534569 0.8601674    0
## Lasso 0.8184963 0.8311307 0.8320203 0.8333364 0.8413560 0.8436785    0
## RF    0.6399914 0.6433239 0.6449060 0.6493738 0.6538199 0.6648279    0
##
## Rsquared
```

```
dotplot(resamps, metric = "RMSE")
```



Predicting

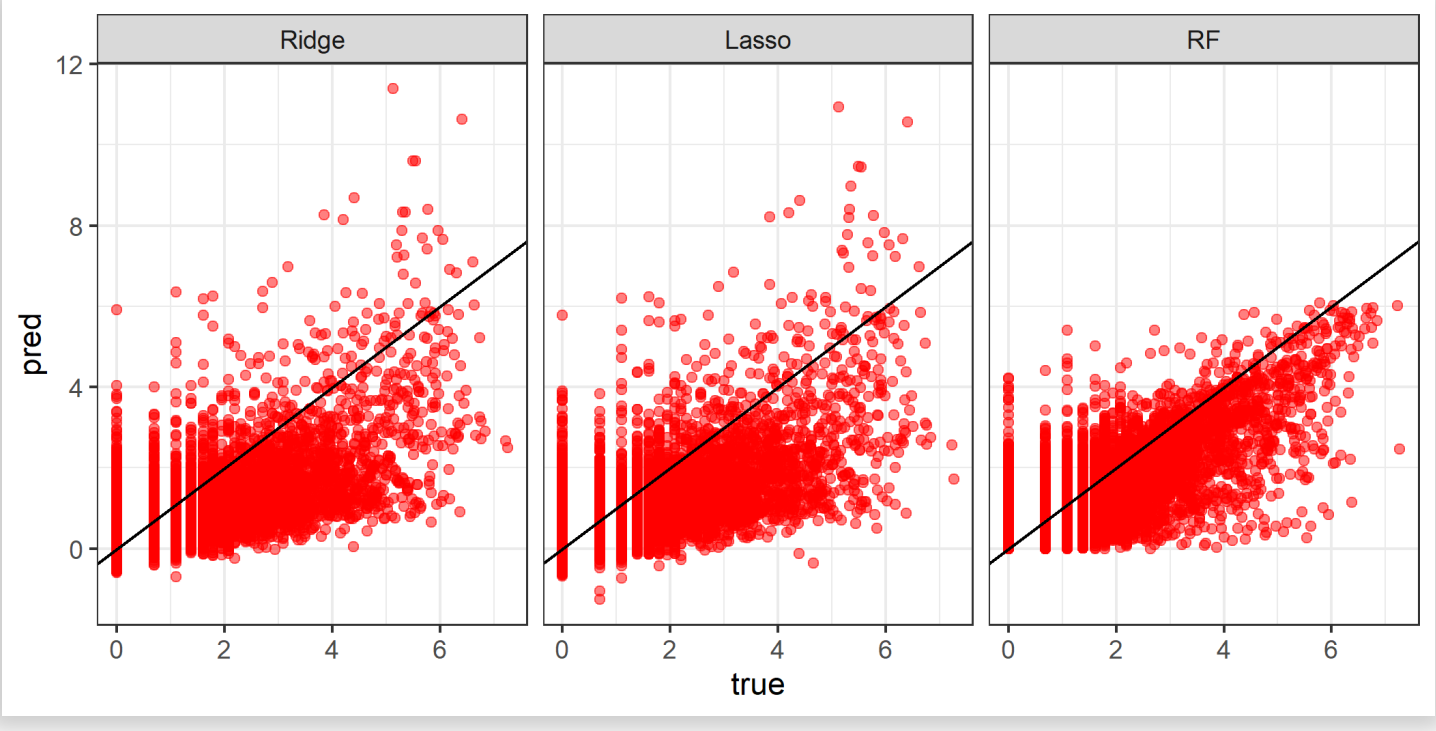
```
pred_ridge <- predict(mod_ridge, newdata = blogs_te)
pred_lasso <- predict(mod_lasso, newdata = blogs_te)
pred_rf <- predict(mod_rf, newdata = blogs_te)

rmse_ridge <- RMSE(pred_ridge, blogs_te$fb)
rmse_lasso <- RMSE(pred_lasso, blogs_te$fb)
rmse_rf <- RMSE(pred_rf, blogs_te$fb)

glue("Test RMSE Ridge: {format(rmse_ridge, digits = 3)}
     Test RMSE Lassoe: {format(rmse_lasso, digits = 3)}
     Test RMSE RF: {format(rmse_rf, digits = 3)}")
```

```
## Test RMSE Ridge: 0.848
## Test RMSE Lassoe: 0.834
## Test RMSE RF: 0.648
```

```
bind_rows(
  tibble(method = "Ridge", pred = pred_ridge, true = blogs_te$fb),
  tibble(method = "Lasso", pred = pred_lasso, true = blogs_te$fb),
  tibble(method = "RF", pred = pred_rf, true = blogs_te$fb)) %>%
  ggplot(aes(true, pred)) +
  geom_point(color = "red", alpha = 0.5) +
  geom_abline(slope = 1, intercept = 0) +
  facet_wrap(~ factor(method, levels = c("Ridge", "Lasso", "RF"))) +
  theme_bw()
```



The ~~Future~~ Present Solution: `tidymodels`

Inspired by Julia Silge

Packages under tidymodels

- `parsnip`: tidy `caret`
- `dials` and `tune`: specifying and tuning model parameters
- `rsample`: sampling, data partitioning
- `recipes`, `embed`, `themis`: preprocessing and creating model matrices
- `infer`: tidy statistics
- `yardstick`: measuring models performance
- `broom`: convert models output into tidy tibbles

And more.

> ⚠ All `tidymodels` packages are under development!

Split Data

The `initial_split()` function is from the `rsample` package:

```
library(tidymodels)

blogs_split_obj <- blogs_fb %>%
  initial_split(prop = 0.6)

print(blogs_split_obj)
```

```
## <Training/Testing/Total>
## <31438/20959/52397>
```

```
blogs_tr <- training(blogs_split_obj)
blogs_te <- testing(blogs_split_obj)

glue("train no. of rows: {nrow(blogs_tr)}
     test no. of rows: {nrow(blogs_te)}")
```

```
## train no. of rows: 31438
## test no. of rows: 20959
```

Preprocess (but we're not gonna use it)

The `recipe()` function is from the `recipes` package. It allows you to specify a sklearn-like pipe you can later apply to any dataset, including all preprocessing steps:

```
blogs_rec <- recipe(fb ~ ., data = blogs_tr)
blogs_rec
```

```
##
## ── Recipe ──────────────────────────────────────────────────────────────────────
##
## ── Inputs
## Number of variables by role
## outcome:     1
## predictor: 280
```

The `recipes` package contains more preprocessing `step_s` than you imagine:

```
blogs_rec <-  blogs_rec %>%
  step_zv(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors())
```

After you have your `recipe` you need to `prep()` materials...

```
blogs_rec <- blogs_rec %>% prep(blogs_tr)

blogs_rec
```

```
##
## ── Recipe ──────────────────────────────────────────────────────────────────
##
## ── Inputs
## Number of variables by role
## outcome:     1
## predictor: 280
##
## ── Training information
## Training data contained 31438 data points and no incomplete rows.
##
```

At this point our `recipe` has all necessary `sd` and `means` for numeric variables.

```
blogs_rec$var_info
```

```
## # A tibble: 281 × 4
##    variable type      role      source
##    <chr>    <list>    <chr>     <chr>
##  1 X1       <chr [2]> predictor original
##  2 X2       <chr [2]> predictor original
##  3 X3       <chr [2]> predictor original
##  4 X4       <chr [2]> predictor original
##  5 X5       <chr [2]> predictor original
##  6 X6       <chr [2]> predictor original
##  7 X7       <chr [2]> predictor original
##  8 X8       <chr [2]> predictor original
##  9 X9       <chr [2]> predictor original
## 10 X10      <chr [2]> predictor original
## # … with 271 more rows
```

```
blogs_rec$steps[[2]]$means |> head()
```

```
##           X1           X2           X3           X4           X5           X6
##  39.0127626  46.5005203   0.3999936 337.9553407  24.2832559  15.0273473
```

```
blogs_rec$steps[[2]]$sds |> head()
```

```
##         X1         X2         X3         X4         X5         X6
##  78.02770  61.73541   8.27524 437.68953  68.58153  31.73173
```

And then we `bake()` (or `juice()`):

```
blogs_tr2 <- blogs_rec %>% bake(blogs_tr)
blogs_te2 <- blogs_rec %>% bake(blogs_te)

glue("mean of comments in orig training: {format(mean(blogs_tr$X51
     mean of comments in baked training: {format(mean(blogs_tr2$X5
```

```
## mean of comments in orig training: 39, sd: 110
## mean of comments in baked training: 3e-17, sd: 1
```

```
glue("mean of comments in orig testing: {format(mean(blogs_te$X51)
     mean of comments in baked testing: {format(mean(blogs_te2$X51
```

```
## mean of comments in orig testing: 40.2, sd: 112
## mean of comments in baked testing: 0.01, sd: 1.02
```

Or you can do it all in a single pipe:

```
blogs_rec <- recipe(fb ~ ., data = blogs_tr) %>%
  step_zv(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  prep(blogs_tr)

blogs_tr2 <- blogs_rec %>% bake(blogs_tr)
blogs_te2 <- blogs_rec %>% bake(blogs_te)

glue("mean of comments in orig training: {format(mean(blogs_tr$X51
     mean of comments in baked training: {format(mean(blogs_tr2$X5
```

```
## mean of comments in orig training: 39, sd: 110
## mean of comments in baked training: 3e-17, sd: 1
```

```
glue("mean of comments in orig testing: {format(mean(blogs_te$X51)
     mean of comments in baked testing: {format(mean(blogs_te2$X51
```

```
## mean of comments in orig testing: 40.2, sd: 112
## mean of comments in baked testing: 0.01, sd: 1.02
```

Can also `tidy()` a recipe:

```
tidy(blogs_rec)
```

```
## # A tibble: 2 × 6
##   number operation type      trained skip  id
##    <int> <chr>     <chr>     <lgl>   <lgl> <chr>
## 1      1 step      zv        TRUE    FALSE zv_Dc9HG
## 2      2 step      normalize TRUE    FALSE normalize_79eTz
```

Fast Forward 10 weeks from now...

```
rec_int_topints <- recipe(pets ~ ., data = okcupid_tr) %>%
  step_textfeature(essays, prefix = "t",
                   extract_functions = my_text_funs) %>%
  update_role(essays, new_role = "discarded") %>%
  step_mutate_at(starts_with("t_"), fn = ~ifelse(is.na(.x), 0, .x)
  step_log(income, starts_with("len_"), starts_with("t_"),
           -t_essays_sent_bing, offset = 1) %>%
  step_meanimpute(income) %>%
  step_other(
    all_nominal(), -has_role("discarded"), -all_outcomes(),
    other = "all_else", threshold = 0.1) %>%
  step_novel(
    all_nominal(), -has_role("discarded"), -all_outcomes()) %>%
  step_modeimpute(all_nominal(), -has_role("discarded"), -all_outc
  step_dummy(all_nominal(), -all_outcomes(),
             -has_role("discarded"), one_hot = FALSE) %>%
  step_interact(topint_ints) %>%
  step_nzv(all_numeric(), freq_cut = 99/1) %>%
  step_upsample(pets, over_ratio = 1, seed = 42)
```

Modeling

For now let us use the original `blogs_tr` data.

Functions `linear_reg()` and `set_engine()` are from the `parsnip` package:

```
mod_ridge_spec <- linear_reg(mixture = 0, penalty = 0.001) %>%
  set_engine(engine = "glmnet")

mod_ridge_spec
```

```
## Linear Regression Model Specification (regression)
##
## Main Arguments:
##   penalty = 0.001
##   mixture = 0
##
## Computational engine: glmnet
```

```
mod_ridge <- mod_ridge_spec %>%
  fit(fb ~ ., data = blogs_tr)

mod_ridge
```

```
## parsnip model object
##
##
## Call:  glmnet::glmnet(x = maybe_matrix(x), y = y, family = "gaussian",
##
##       Df  %Dev Lambda
## 1    276  0.00 600.50
## 2    276  2.57 547.10
## 3    276  2.80 498.50
## 4    276  3.05 454.30
## 5    276  3.32 413.90
## 6    276  3.61 377.10
## 7    276  3.92 343.60
## 8    276  4.25 313.10
## 9    276  4.61 285.30
## 10   276  4.99 259.90
## 11   276  5.40 236.80
## 12   276  5.84 215.80
## 13   276  6.30 196.60
## 14   276  6.79 179.20
## 15   276  7.30 163.20
## 16   276  7.85 148.70
## 17   276  8.42 135.50
## 18   276  9.02 123.50
```

In a single pipe:

```
mod_lasso <- linear_reg(mixture = 1, penalty = 0.001) %>%
  set_engine(engine = "glmnet") %>%
  fit(fb ~ ., data = blogs_tr)

mod_lasso
```

```
## parsnip model object
##
##
## Call:  glmnet::glmnet(x = maybe_matrix(x), y = y, family = "gaussian",
##
##      Df  %Dev  Lambda
## 1     0  0.00 0.60050
## 2     1  4.79 0.54710
## 3     1  8.76 0.49850
## 4     3 12.97 0.45430
## 5     3 16.73 0.41390
## 6     3 19.85 0.37710
## 7     3 22.45 0.34360
## 8     3 24.60 0.31310
## 9     4 26.54 0.28530
## 10    4 28.94 0.25990
## 11    4 30.94 0.23680
## 12    4 32.59 0.21580
## 13    4 33.97 0.19660
## 14    4 35.11 0.17920
```

Can also use `fit_xy()` a-la `sklearn`:

```
mod_rf <- rand_forest(mode = "regression", mtry = 50, trees = 50,
  set_engine("ranger") %>%
  fit_xy(x = blogs_tr[, -281],
         y = blogs_tr$fb)

mod_rf
```

```
## parsnip model object
##
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~50,
##
## Type:                             Regression
## Number of trees:                  50
## Sample size:                      31438
## Number of independent variables:  280
## Mtry:                             50
## Target node size:                 10
## Variable importance mode:         none
## Splitrule:                        variance
## OOB prediction error (MSE):       0.4304248
## R squared (OOB):                  0.6633549
```

Notice how easy it is to get the model's results in a tidy way using the `tidy()` function:

```
tidy(mod_ridge)
```

```
## # A tibble: 281 × 3
##    term         estimate penalty
##    <chr>           <dbl>   <dbl>
##  1 (Intercept)  0.684      0.001
##  2 X1           0.00129    0.001
##  3 X2           0.00131    0.001
##  4 X3          -0.00149    0.001
##  5 X4           0.000321   0.001
##  6 X5           0.000212   0.001
##  7 X6           0.00217    0.001
##  8 X7          -0.00128    0.001
##  9 X8           0.115      0.001
## 10 X9          -0.000138   0.001
## # … with 271 more rows
```

Predicting

```
results_test <- mod_ridge %>%
  predict(new_data = blogs_te, penalty = 0.001) %>%
  mutate(
    truth = blogs_te$fb,
    method = "Ridge"
  ) %>%
  bind_rows(mod_lasso %>%
    predict(new_data = blogs_te) %>%
    mutate(
      truth = blogs_te$fb,
      method = "Lasso"
  )) %>%
  bind_rows(mod_rf %>%
    predict(new_data = blogs_te) %>%
    mutate(
      truth = blogs_te$fb,
      method = "RF"
  ))

dim(results_test)
```

```
## [1] 62877     3
```

```
head(results_test)
```

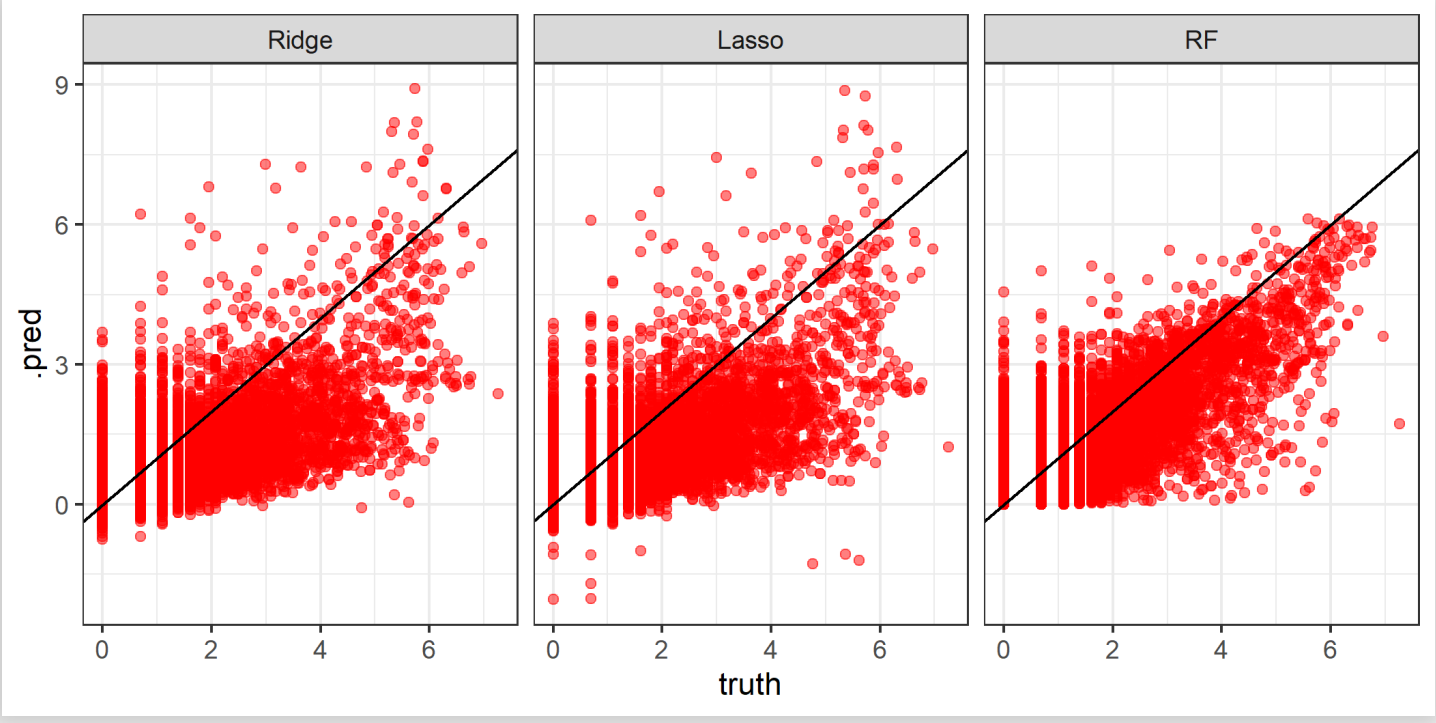
Comparing Models

The package `yardstick` has tons of performance metrics:

```
results_test %>%
  group_by(method) %>%
  rmse(truth = truth, estimate = .pred)
```

```
## # A tibble: 3 × 4
##   method .metric .estimator .estimate
##   <chr>  <chr>   <chr>          <dbl>
## 1 Lasso  rmse    standard       0.834
## 2 RF     rmse    standard       0.643
## 3 Ridge  rmse    standard       0.845
```

```
results_test %>%
  ggplot(aes(truth, .pred)) +
  geom_point(color = "red", alpha = 0.5) +
  geom_abline(slope = 1, intercept = 0) +
  facet_wrap(~ factor(method, levels = c("Ridge", "Lasso", "RF"))) +
  theme_bw()
```



Tuning

Define your model spec, using `tune()` from the `tune` package for a parameter you wish to tune:

```
mod_rf_spec <- rand_forest(mode = "regression",
                           mtry = tune(),
                           min_n = tune(),
                           trees = 100) %>%
  set_engine("ranger")
```

Define the `grid` on which you train your params, with the `dials` package:

```
rf_grid <- grid_regular(mtry(range(10, 70)), min_n(range(10, 30)),
                        levels = c(4, 3))

rf_grid
```

```
## # A tibble: 12 × 2
##     mtry min_n
##    <int> <int>
##  1    10    10
##  2    30    10
##  3    50    10
##  4    70    10
##  5    10    20
##  6    30    20
##  7    50    20
##  8    70    20
##  9    10    30
## 10    30    30
## 11    50    30
## 12    70    30
```

Split your data into a few folds for Cross Validation with `vfold_cv()` from the `rsample` package:

```
cv_splits <- vfold_cv(blogs_tr, v = 5)

cv_splits
```

```
## #  5-fold cross-validation
## # A tibble: 5 × 2
##   splits               id
##   <list>               <chr>
## 1 <split [25150/6288]> Fold1
## 2 <split [25150/6288]> Fold2
## 3 <split [25150/6288]> Fold3
## 4 <split [25151/6287]> Fold4
## 5 <split [25151/6287]> Fold5
```

Now perform cross validation with `tune_grid()` from the `tune` package:

```
tune_res <- tune_grid(mod_rf_spec,
                      recipe(fb ~ ., data = blogs_tr),
                      resamples = cv_splits,
                      grid = rf_grid,
                      metrics = metric_set(rmse))
tune_res
```

```
## # Tuning results
## # 5-fold cross-validation
## # A tibble: 5 × 4
##   splits               id    .metrics          .notes
##   <list>               <chr> <list>            <list>
## 1 <split [25150/6288]> Fold1 <tibble [12 × 6]> <tibble [0 × 1]>
## 2 <split [25150/6288]> Fold2 <tibble [12 × 6]> <tibble [0 × 1]>
## 3 <split [25150/6288]> Fold3 <tibble [12 × 6]> <tibble [0 × 1]>
## 4 <split [25151/6287]> Fold4 <tibble [12 × 6]> <tibble [0 × 1]>
## 5 <split [25151/6287]> Fold5 <tibble [12 × 6]> <tibble [0 × 1]>
```

```
tune_res$.metrics[[1]]
```

```
## # A tibble: 12 × 6
##     mtry min_n .metric .estimator .estimate .config
##    <int> <int> <chr>   <chr>          <dbl> <chr>
##  1    10    10 rmse    standard       0.687 Preprocessor1_Model01
##  2    30    10 rmse    standard       0.649 Preprocessor1_Model02
##  3    50    10 rmse    standard       0.644 Preprocessor1_Model03
##  4    70    10 rmse    standard       0.642 Preprocessor1_Model04
##  5    10    20 rmse    standard       0.697 Preprocessor1_Model05
##  6    30    20 rmse    standard       0.655 Preprocessor1_Model06
##  7    50    20 rmse    standard       0.646 Preprocessor1_Model07
##  8    70    20 rmse    standard       0.643 Preprocessor1_Model08
##  9    10    30 rmse    standard       0.701 Preprocessor1_Model09
## 10    30    30 rmse    standard       0.657 Preprocessor1_Model10
## 11    50    30 rmse    standard       0.651 Preprocessor1_Model11
## 12    70    30 rmse    standard       0.643 Preprocessor1_Model12
```

Collect the mean metric across folds:

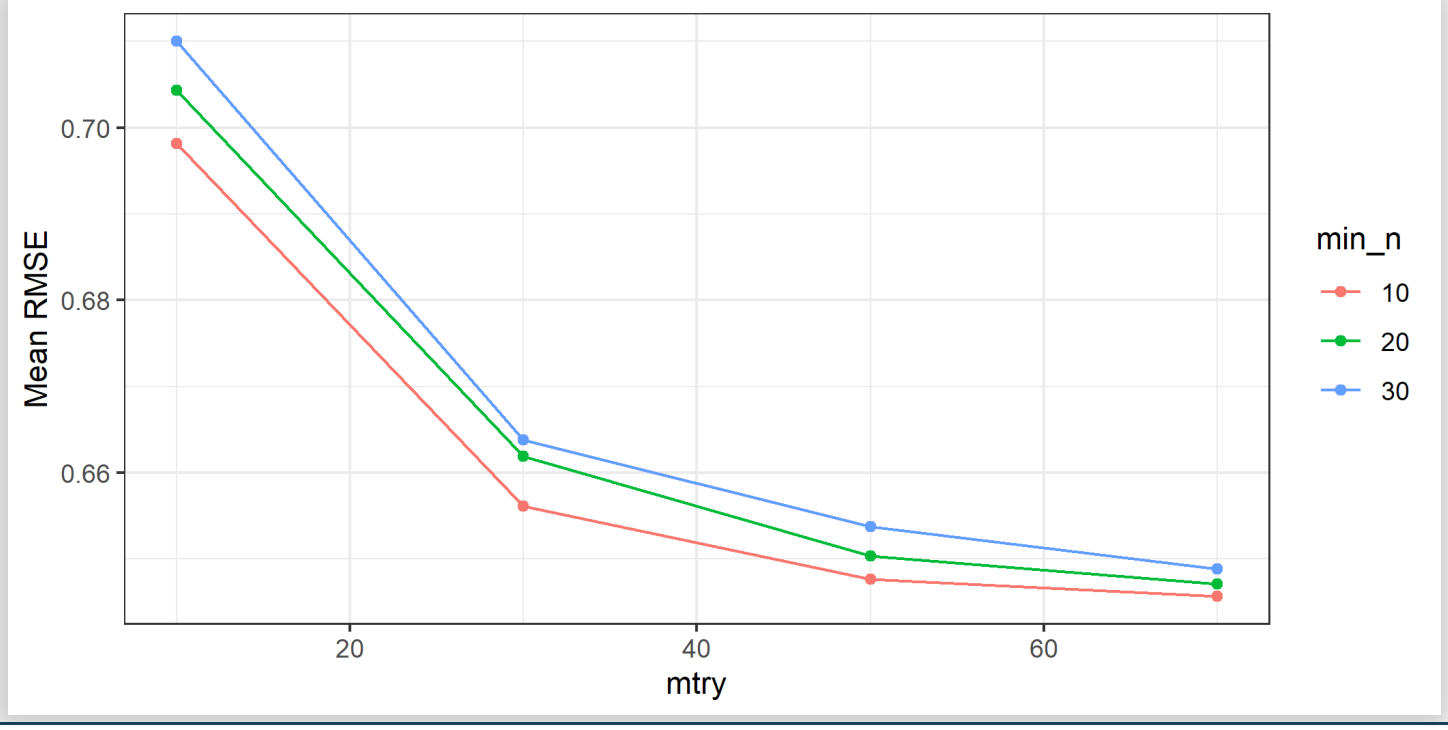
```
estimates <- collect_metrics(tune_res)

estimates
```

```
## # A tibble: 12 × 8
##     mtry min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1    10    10 rmse    standard   0.698     5 0.00647 Preprocessor1_Mode
##  2    30    10 rmse    standard   0.656     5 0.00673 Preprocessor1_Mode
##  3    50    10 rmse    standard   0.648     5 0.00569 Preprocessor1_Mode
##  4    70    10 rmse    standard   0.646     5 0.00572 Preprocessor1_Mode
##  5    10    20 rmse    standard   0.704     5 0.00668 Preprocessor1_Mode
##  6    30    20 rmse    standard   0.662     5 0.00638 Preprocessor1_Mode
##  7    50    20 rmse    standard   0.650     5 0.00527 Preprocessor1_Mode
##  8    70    20 rmse    standard   0.647     5 0.00576 Preprocessor1_Mode
##  9    10    30 rmse    standard   0.710     5 0.00740 Preprocessor1_Mode
## 10    30    30 rmse    standard   0.664     5 0.00590 Preprocessor1_Mode
## 11    50    30 rmse    standard   0.654     5 0.00526 Preprocessor1_Mode
## 12    70    30 rmse    standard   0.649     5 0.00567 Preprocessor1_Mode
```

Choose best parameter:

```
estimates %>%
  mutate(min_n = factor(min_n)) %>%
  ggplot(aes(x = mtry, y = mean, color = min_n)) +
  geom_point() +
  geom_line() +
  labs(y = "Mean RMSE") +
  theme_bw()
```



There are of course also methods for helping us choose best params and final model.

```
best_rmse <- tune_res %>% select_best(metric = "rmse")
best_rmse
```

```
## # A tibble: 1 × 3
##    mtry min_n .config
##   <int> <int> <chr>
## 1    70    10 Preprocessor1_Model04
```

See also `?select_by_one_std_err`.

```
mod_rf_final <- finalize_model(mod_rf_spec, best_rmse)
mod_rf_final
```

```
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = 70
##   trees = 100
##   min_n = 10
##
## Computational engine: ranger
```

```
mod_rf_final %>%
  fit(fb ~ ., data = blogs_tr) %>%
  predict(new_data = blogs_te) %>%
  mutate(truth = blogs_te$fb) %>%
  head(10)
```

```
## # A tibble: 10 × 2
##    .pred truth
##    <dbl> <dbl>
##  1 0.584 0.693
##  2 0.584 0.693
##  3 0.238 0
##  4 1.79  2.30
##  5 1.79  2.30
##  6 0.145 0
##  7 3.33  1.10
##  8 0.719 1.39
##  9 2.40  3.09
## 10 1.14  0.693
```

Workflow

As we shall see, this manual approach won't scale, is prone to bugs and will not play nicely with other modeling components:

```
results_test <- mod_ridge %>%
  predict(new_data = blogs_te, penalty = 0.001) %>%
  mutate(
    truth = blogs_te$fb,
    method = "Ridge"
  ) %>%
  bind_rows(mod_lasso %>%
    predict(new_data = blogs_te) %>%
    mutate(
      truth = blogs_te$fb,
      method = "Lasso"
  )) %>%
  bind_rows(mod_rf %>%
    predict(new_data = blogs_te) %>%
    mutate(
      truth = blogs_te$fb,
      method = "RF"
  ))
```

Similar to sklearn's `Pipeline()` class, we need a `workflow()`: to bundle together your pre-processing, modeling, and post-processing requests.

```
mod_rf <- rand_forest(mode = "regression", mtry = 70, trees = 100,
  set_engine("ranger")

blogs_rec <- recipe(fb ~ ., data = blogs_tr) %>%
  step_zv(all_numeric_predictors()) %>%
  step_normalize(all_numeric_predictors())

wflow_rf <-
  workflow() %>%
  add_recipe(blogs_rec) %>%
  add_model(mod_rf)
```

```
wflow_rf
```

```
## ══ Workflow ════════════════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: rand_forest()
##
## ── Preprocessor ────────────────────────────────────────────────────────────
## 2 Recipe Steps
##
## • step_zv()
## • step_normalize()
##
## ── Model ───────────────────────────────────────────────────────────────────
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = 70
##   trees = 100
##   min_n = 10
##
## Computational engine: ranger
```

Calling `fit()` will `prep()` the recipe **and** `fit()` the model:

```
fit_rf <- fit(wflow_rf, blogs_tr)
```

It is still a `workflow()` object:

```
fit_rf
```

```
## ══ Workflow [trained] ══════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: rand_forest()
##
## ── Preprocessor ────────────────────────────────────────────────────────────
## 2 Recipe Steps
##
## • step_zv()
## • step_normalize()
##
## ── Model ───────────────────────────────────────────────────────────────────
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~70,
##
## Type:                             Regression
## Number of trees:                  100
```

Can extract the prepped recipe:

```
fit_rf %>%
  extract_recipe()
```

```
##
## ── Recipe ──────────────────────────────────────────────────────────────────────
##
## ── Inputs
## Number of variables by role
## outcome:     1
## predictor: 280
##
## ── Training information
## Training data contained 31438 data points and no incomplete rows.
##
```

Can extract the actual parsnip model:

```
fit_rf %>%
  extract_fit_parsnip()
```

```
## parsnip model object
##
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~70,
##
## Type:                             Regression
## Number of trees:                  100
## Sample size:                      31438
## Number of independent variables:  276
## Mtry:                             70
## Target node size:                 10
## Variable importance mode:         none
## Splitrule:                        variance
## OOB prediction error (MSE):       0.4176772
## R squared (OOB):                  0.677417
```

Calling `predict()` with a new dataset will `bake()` the new dataset using the prepped recipe, and `predict()` the trained model:

```
res_rf <- predict(fit_rf, blogs_te)

res_rf %>% head(10)
```

```
## # A tibble: 10 × 1
##      .pred
##      <dbl>
##  1 0.477
##  2 0.0928
##  3 0.0928
##  4 2.16
##  5 0.150
##  6 0.163
##  7 2.48
##  8 2.48
##  9 2.68
## 10 2.68
```

Calling `fit_resamples()` or `tune_grid()` works with a `vfold_cv()` splits object:

```
## # Resampling results
## # 5-fold cross-validation
## # A tibble: 5 × 4
##   splits               id    .metrics          .notes
##   <list>               <chr> <list>            <list>
## 1 <split [25150/6288]> Fold1 <tibble [1 × 4]> <tibble [0 × 3]>
## 2 <split [25150/6288]> Fold2 <tibble [1 × 4]> <tibble [0 × 3]>
## 3 <split [25150/6288]> Fold3 <tibble [1 × 4]> <tibble [0 × 3]>
## 4 <split [25151/6287]> Fold4 <tibble [1 × 4]> <tibble [0 × 3]>
## 5 <split [25151/6287]> Fold5 <tibble [1 × 4]> <tibble [0 × 3]>
```

```
cv_splits <- vfold_cv(blogs_tr, v = 5)

fit_rf_cv <- fit_resamples(wflow_rf, cv_splits, metrics = metric_s

fit_rf_cv
```

Can use `collect_metrics()` etc.

But the real advantage of working with `workflow()`s is when comparing different recipes $\times$ different models, via `workflow_set()`:

```
mod_ridge <- linear_reg(mixture = 0, penalty = 0.001) %>%
  set_engine(engine = "glmnet")

mod_lasso <- linear_reg(mixture = 1, penalty = 0.001) %>%
  set_engine(engine = "glmnet")

mod_list <- list(RF = mod_rf, Ridge = mod_ridge, Lasso = mod_lasso

rec_list <- list(basic = blogs_rec) # can add more..

wset <- workflow_set(rec_list, mod_list) # checkout the cross arg

wset
```

```
## # A workflow set/tibble: 3 × 4
##   wflow_id    info             option    result
##   <chr>       <list>           <list>    <list>
## 1 basic_RF    <tibble [1 × 4]> <opts[0]> <list [0]>
## 2 basic_Ridge <tibble [1 × 4]> <opts[0]> <list [0]>
## 3 basic_Lasso <tibble [1 × 4]> <opts[0]> <list [0]>
```

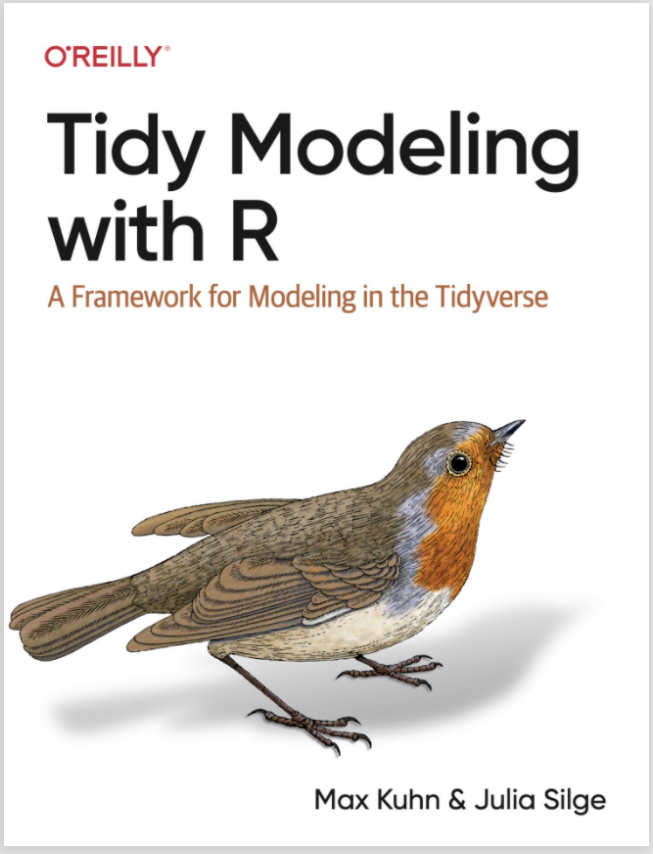
Now use a `purrr`-like mapping function to fit all on all resamples:

```
wset <-
  wset %>%
  workflow_map("fit_resamples", resamples = cv_splits)
```

```
## # A workflow set/tibble: 3 × 4
##   wflow_id    info             option    result
##   <chr>       <list>           <list>    <list>
## 1 basic_RF    <tibble [1 × 4]> <opts[1]> <rsmp[+]>
## 2 basic_Ridge <tibble [1 × 4]> <opts[1]> <rsmp[+]>
## 3 basic_Lasso <tibble [1 × 4]> <opts[1]> <rsmp[+]>
```

And use e.g. `collect_metrics()` to select the best recipe/model combination.

Book (WIP?)



https://www.tmwr.org/

`infer`: Tidy Statistics

Statistical Q1

Is there a relation between posts published on Sundays and blogger hand dominance, where hand dominance is totally made up? 😬

```
pub_vs_hand <- blogs_fb %>%
  select(sunday, hand) %>%
  table()

pub_vs_hand
```

```
##        hand
## sunday  left right
##     NS  4788 42950
##     S    436  4223
```

```
prop.table(pub_vs_hand, margin = 1)
```

```
##        hand
## sunday       left      right
##     NS 0.10029746 0.89970254
##     S  0.09358231 0.90641769
```

Statistical Q2

Is there a difference in feedback between posts published on Sundays and posts published on another day?

```
blogs_fb %>%
  group_by(sunday) %>% summarise(avg = mean(fb), sd = sd(fb), n =
```

```
## # A tibble: 2 × 4
##   sunday   avg    sd     n
##   <fct>  <dbl> <dbl> <int>
## 1 NS     0.645  1.13 47738
## 2 S      0.605  1.12  4659
```

Same Problem!

Varied interface, varied output.

```
prop.test(pub_vs_hand[,1], rowSums(pub_vs_hand))
```

```
##
##     2-sample test for equality of proportions with continuity correction
##
## data:  pub_vs_hand[, 1] out of rowSums(pub_vs_hand)
## X-squared = 2.0583, df = 1, p-value = 0.1514
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##  -0.002189083  0.015619369
## sample estimates:
##     prop 1     prop 2
## 0.10029746 0.09358231
```

```
t.test(fb ~ sunday, data = blogs_fb)
```

```
##
##     Welch Two Sample t-test
##
## data:  fb by sunday
## t = 2.3606, df = 5638.9, p-value = 0.01828
## alternative hypothesis: true difference in means between group NS and gr
## 95 percent confidence interval:
##  0.006863705 0.074103874
## sample estimates:
## mean in group NS  mean in group S
##        0.6454439        0.6049601
```

The `generics::tidy()` Approach

(Also available when you load several other packages, like `broom` and `yardstick`)

```
tidy(prop.test(pub_vs_hand[,1], rowSums(pub_vs_hand)))
```

```
## # A tibble: 1 × 9
##   estimate1 estimate2 statistic p.value parameter conf.low conf.high met
##       <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl> <ch
## 1     0.100    0.0936      2.06   0.151         1 -0.00219    0.0156 2-s
## # … with 1 more variable: alternative <chr>
```

```
tidy(t.test(fb ~ sunday, data = blogs_fb))
```

```
## # A tibble: 1 × 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf
##      <dbl>     <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>
## 1   0.0405     0.645     0.605      2.36  0.0183     5639.  0.00686    0
## # … with 2 more variables: method <chr>, alternative <chr>
```

The `infer` Approach

> infer implements an expressive grammar to perform statistical inference that coheres with the tidyverse design framework

4 main verbs for a typical flow:

- `specify()` - dependent/independent variables, formula
- `hypothesize()` - declare the null hypothesis
- `generate()` - generate data reflecting the null hypothesis (the permutation/bootstrap approach)
- `calculate()` - calculate a distribution of statistics from the generated data, from which you can extract conclusion based on a p-value for example

`infer` Diff in Proportions Test

Get the observed statistic (here manually in order to not confuse you, there *is* a way via `infer`):

```
pub_vs_hand
```

```
##       hand
## sunday  left right
##     NS  4788 42950
##     S    436  4223
```

```
p_NS <- pub_vs_hand[1, 1] / (sum(pub_vs_hand[1, ]))
p_S <- pub_vs_hand[2, 1] / (sum(pub_vs_hand[2, ]))
obs_diff <- p_NS - p_S
obs_diff
```

```
## [1] 0.006715143
```

Get distribution of the difference in proportions under null hypothesis

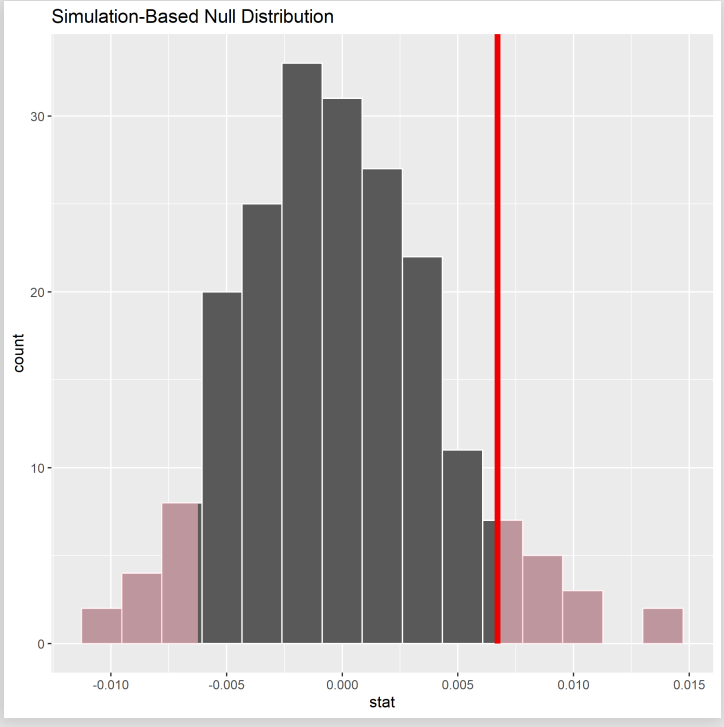
```
diff_null_perm <- blogs_fb %>%
  specify(hand ~ sunday, success = "left") %>%
  hypothesize(null = "independence") %>%
  generate(reps = 200, type = "permute") %>%
  calculate(stat = "diff in props", order = c("NS", "S"))

diff_null_perm
```

```
## Response: hand (factor)
## Explanatory: sunday (factor)
## Null Hypothesis: independence
## # A tibble: 200 × 2
##    replicate      stat
##        <int>     <dbl>
##  1         1 -0.000588
##  2         2 -0.00153
##  3         3  0.00884
##  4         4  0.00365
##  5         5 -0.00129
##  6         6 -0.00177
##  7         7  0.00106
##  8         8  0.00106
##  9         9 -0.00247
## 10        10 -0.000352
## # … with 190 more rows
```

Visualize the permuted difference null distribution and the p-value

```
visualize(diff_null_perm) +
  shade_p_value(obs_stat = obs_diff, direction = "two_sided")
```



Get the actual p-value:

```
diff_null_perm %>%
  get_p_value(obs_stat = obs_diff, direction = "two_sided")
```

```
## # A tibble: 1 × 1
##   p_value
##     <dbl>
## 1    0.14
```

`infer` t Test (independent samples)

Get the observed statistic (here via `infer`):

```
obs_t <- blogs_fb %>%
  specify(fb ~ sunday) %>%
  calculate(stat = "t", order = c("NS", "S"))
obs_t
```

```
## Response: fb (numeric)
## Explanatory: sunday (factor)
## # A tibble: 1 × 1
##    stat
##   <dbl>
## 1  2.36
```

Get distribution of the t statistic under null hypothesis

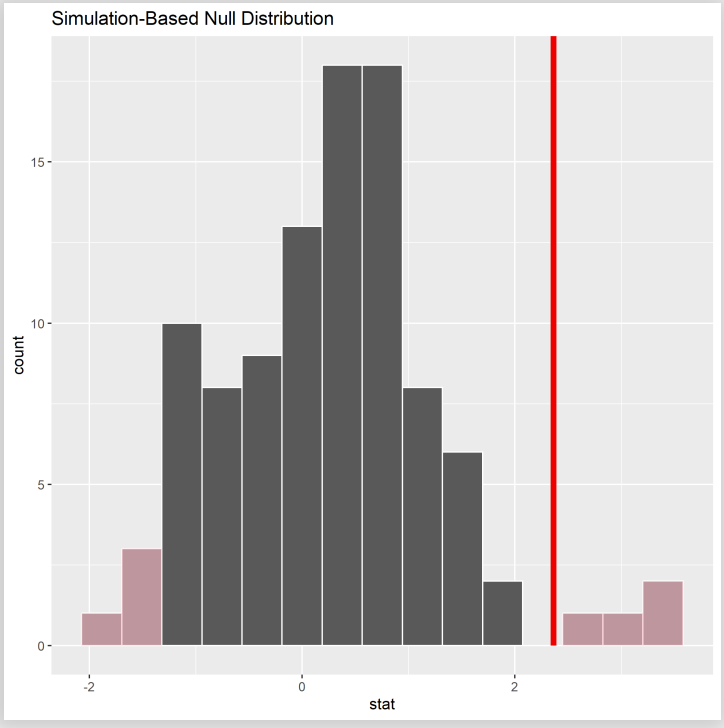
```
t_null_perm <- blogs_fb %>%
  specify(fb ~ sunday) %>%
  hypothesize(null = "independence") %>%
  generate(reps = 100, type = "permute") %>%
  calculate(stat = "t", order = c("NS", "S"))

t_null_perm
```

```
## Response: fb (numeric)
## Explanatory: sunday (factor)
## Null Hypothesis: independence
## # A tibble: 100 × 2
##    replicate     stat
##        <int>    <dbl>
##  1         1 -2.04
##  2         2  0.961
##  3         3  0.782
##  4         4  0.00496
##  5         5 -1.30
##  6         6 -0.635
##  7         7 -0.634
##  8         8  0.913
##  9         9 -1.04
## 10        10  1.16
## # … with 90 more rows
```

Visualize the permuted t statistic null distribution and the two-sided p-value

```
visualize(t_null_perm) +
  shade_p_value(obs_stat = obs_t, direction = "two_sided")
```



Get the actual p-value:

```
t_null_perm %>%
  get_p_value(obs_stat = obs_t, direction = "two_sided")
```

```
## # A tibble: 1 × 1
##   p_value
##     <dbl>
## 1    0.08
```