

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

2021-12-30

APPLICATIONS



OF DATA SCIENCE

The Problem

APPLICATIONS



OF DATA SCIENCE

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

APPLICATIONS



OF DATA SCIENCE

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 $\log(f_b)$ based on all features, no missing values:

```
blogs_fb <- read_csv("~/BlogFeedback/blogData_train.csv", col_names = c("fb", "blog_len", "sunday"))

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

dim(blogs_fb)

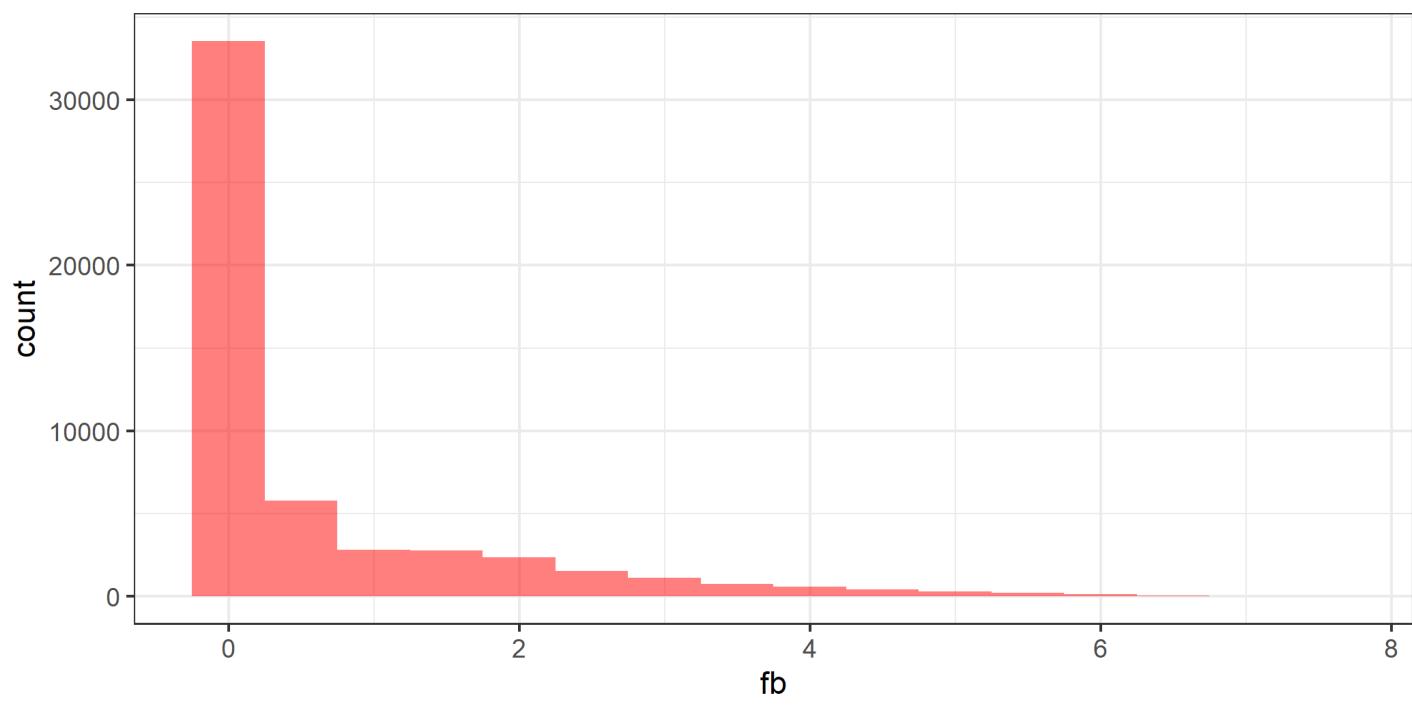
## [1] 52397    281
```

```
glimpse(blogs_fb)
```

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

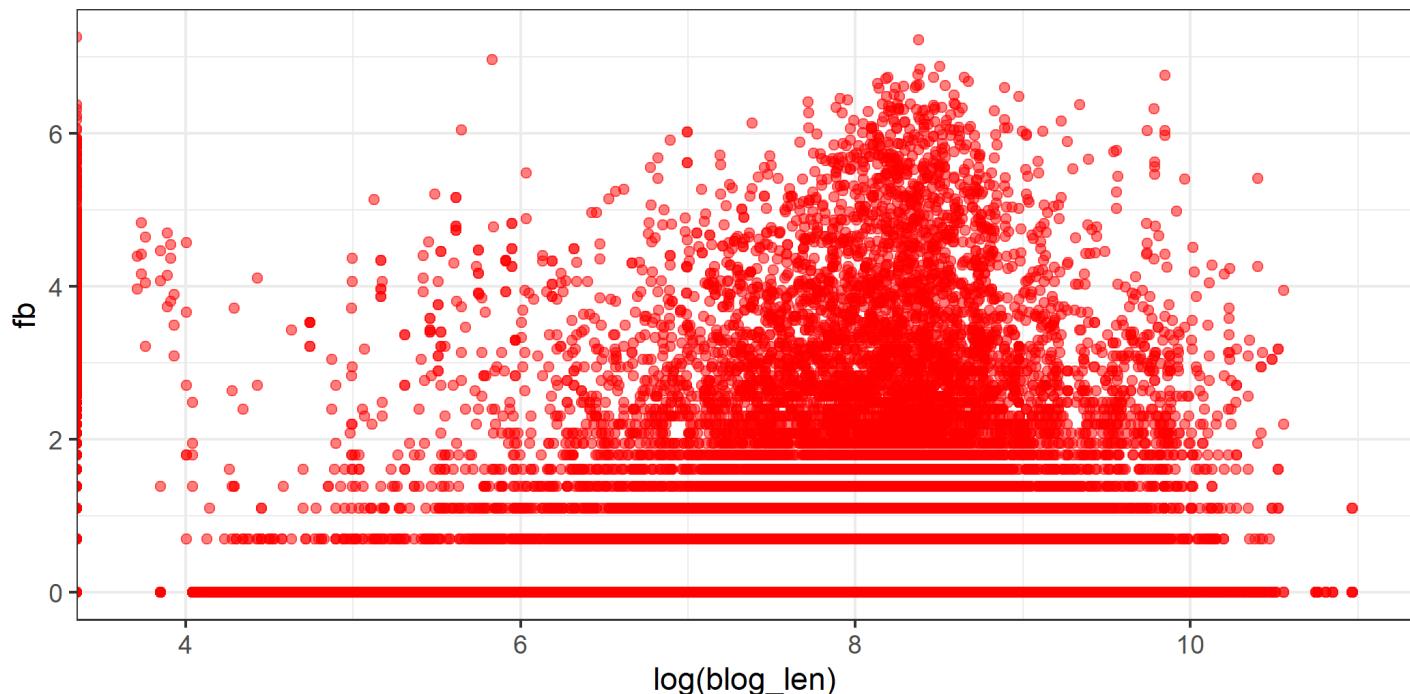
See the dependent variable distribution:

```
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

APPLICATIONS



OF DATA SCIENCE

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

APPLICATIONS



OF DATA SCIENCE

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)}\n"
     "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, 25151, 25151, 25152, 25151
## Resampling results across tuning parameters:
##
##     lambda      RMSE      Rsquared      MAE
##     0.001000000  0.8467994  0.4387687  0.5919345
##     0.001206793  0.8467994  0.4387687  0.5919345
##     0.001456348  0.8467994  0.4387687  0.5919345
##     0.001757511  0.8467994  0.4387687  0.5919345
##     0.002120951  0.8467994  0.4387687  0.5919345
##     0.002559548  0.8467994  0.4387687  0.5919345
##     0.003088844  0.8467994  0.4387687  0.5919345
##     0.003727594  0.8467994  0.4387687  0.5919345
##     0.004498433  0.8467994  0.4387687  0.5919345
##     0.005428675  0.8467994  0.4387687  0.5919345
##     0.006551286  0.8467994  0.4387687  0.5919345
##     0.007000000  0.8467994  0.4387687  0.5919345
```

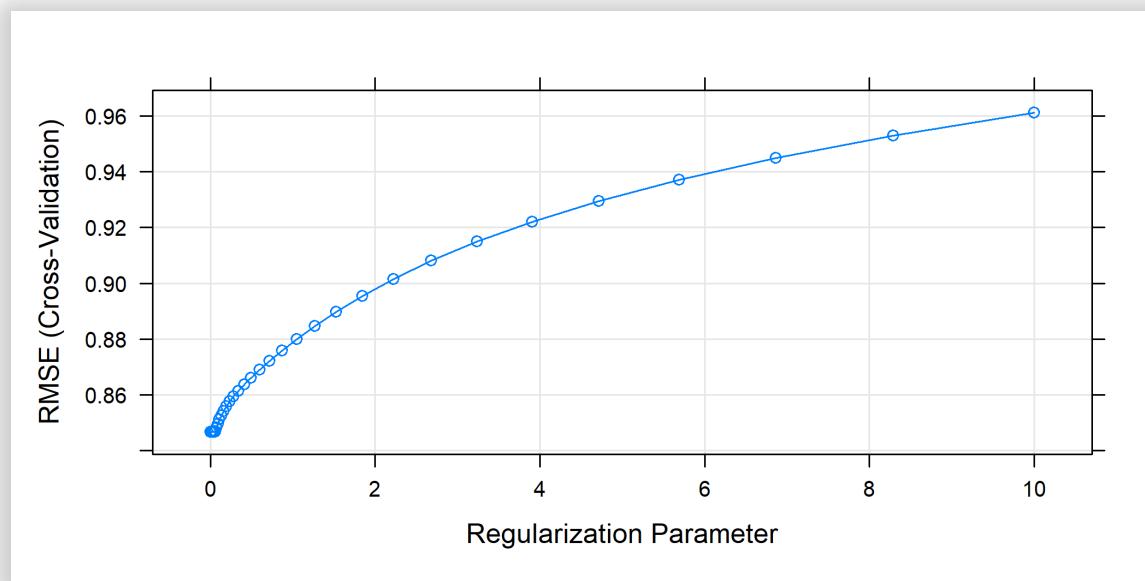
```
mod_lasso
```

```
## glmnet
##
## 31439 samples
##    280 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 25151, 25151, 25152, 25151, 25151
## Resampling results across tuning parameters:
##
##     lambda      RMSE      Rsquared      MAE
## 0.001000000  0.8337688  0.4554690  0.5798744
## 0.001206793  0.8341500  0.4549660  0.5802843
## 0.001456348  0.8347039  0.4542405  0.5808603
## 0.001757511  0.8355192  0.4531749  0.5817068
## 0.002120951  0.8366514  0.4517014  0.5828165
## 0.002559548  0.8377229  0.4503159  0.5837793
## 0.003088844  0.8391754  0.4484291  0.5849757
## 0.003727594  0.8412298  0.4457342  0.5865769
## 0.004498433  0.8431266  0.4432589  0.5880512
## 0.005428675  0.8451364  0.4406221  0.5893364
## 0.006551286  0.8475508  0.4374386  0.5908574
## 0.007906043  0.8503312  0.4337628  0.5929027
## 0.009540955  0.8530921  0.4301238  0.5950345
## 0.011513954  0.8557984  0.4265541  0.5970637
## 0.013894955  0.8581235  0.4235255  0.5987878
## 0.016768329  0.8604976  0.4204546  0.6007478
```

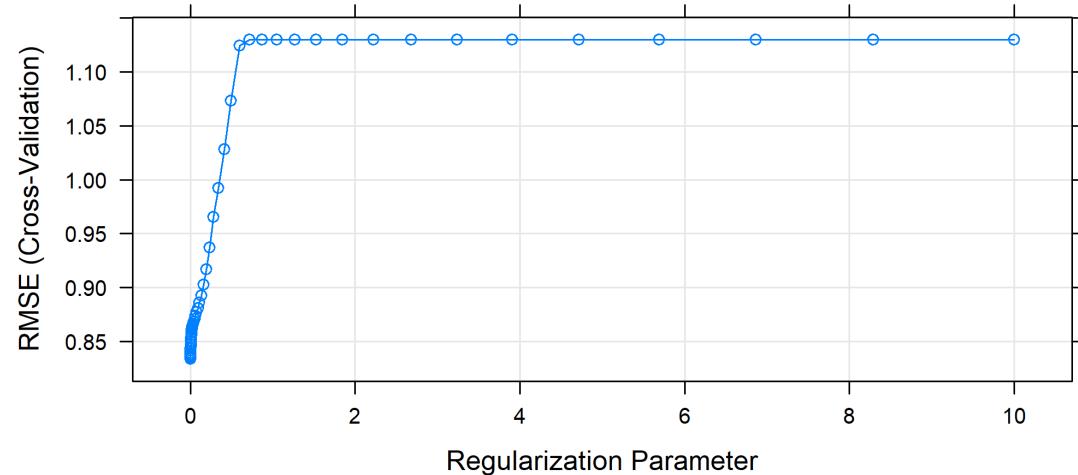
```
mod_rf
```

```
## Random Forest
##
## 31439 samples
##    280 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 25151, 25151, 25151, 25152, 25151
## Resampling results across tuning parameters:
##
##   min.node.size  mtry   RMSE      Rsquared     MAE
##   10            10    0.6994401  0.6235614  0.4490280
##   10            30    0.6608756  0.6590346  0.4122673
##   10            50    0.6550989  0.6642348  0.4053400
##   20            10    0.7067291  0.6160797  0.4553954
##   20            30    0.6648241  0.6554564  0.4179512
##   20            50    0.6556846  0.6640163  0.4091380
##   30            10    0.7125389  0.6106795  0.4608372
##   30            30    0.6662476  0.6544456  0.4200763
##   30            50    0.6575038  0.6623432  0.4109595
##
## 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.
```

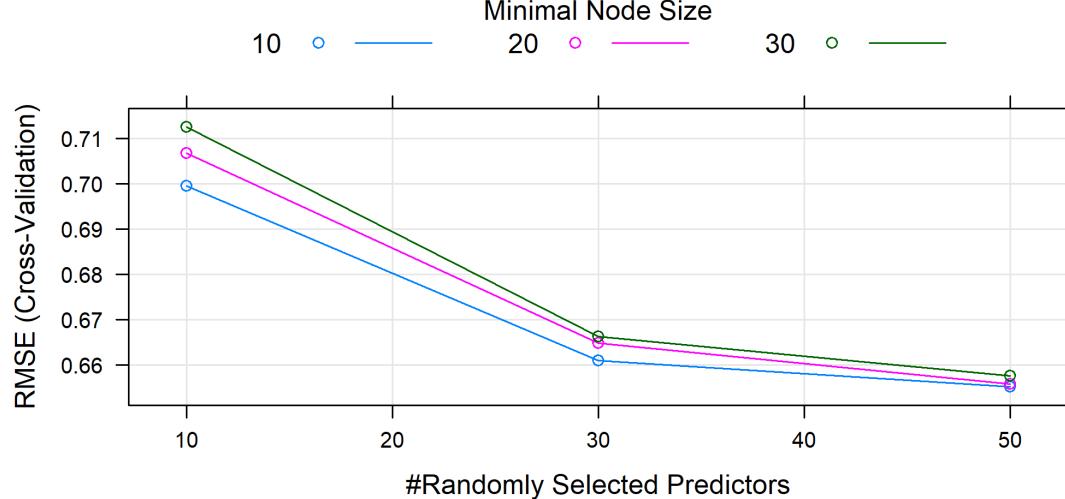
```
plot(mod_ridge)
```



```
plot(mod_lasso)
```



```
plot(mod_rf)
```

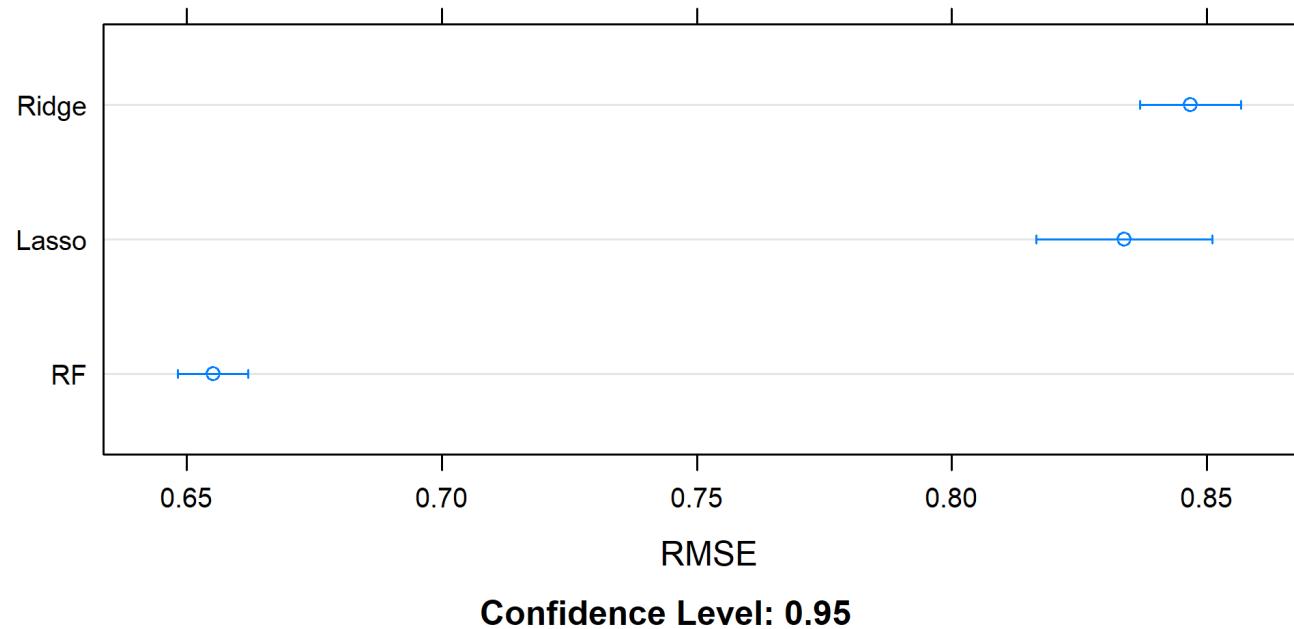


Comparing Models

```
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 0.5865591 0.5865627 0.5926886 0.5919345 0.5968818 0.5969802 0  
## Lasso 0.5656421 0.5818206 0.5835294 0.5798744 0.5839981 0.5843815 0  
## RF    0.4033943 0.4046754 0.4050187 0.4053400 0.4060125 0.4075993 0  
##  
## RMSE  
##          Min.   1st Qu.   Median   Mean   3rd Qu.   Max. NA's  
## Ridge 0.8380841 0.8412998 0.8453311 0.8467994 0.8512373 0.8580448 0  
## Lasso 0.8109915 0.8299759 0.8412628 0.8337688 0.8426856 0.8439281 0  
## RF    0.6454345 0.6559274 0.6567492 0.6550989 0.6577902 0.6595931 0  
##  
## Rsquared  
##
```

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



Predicting

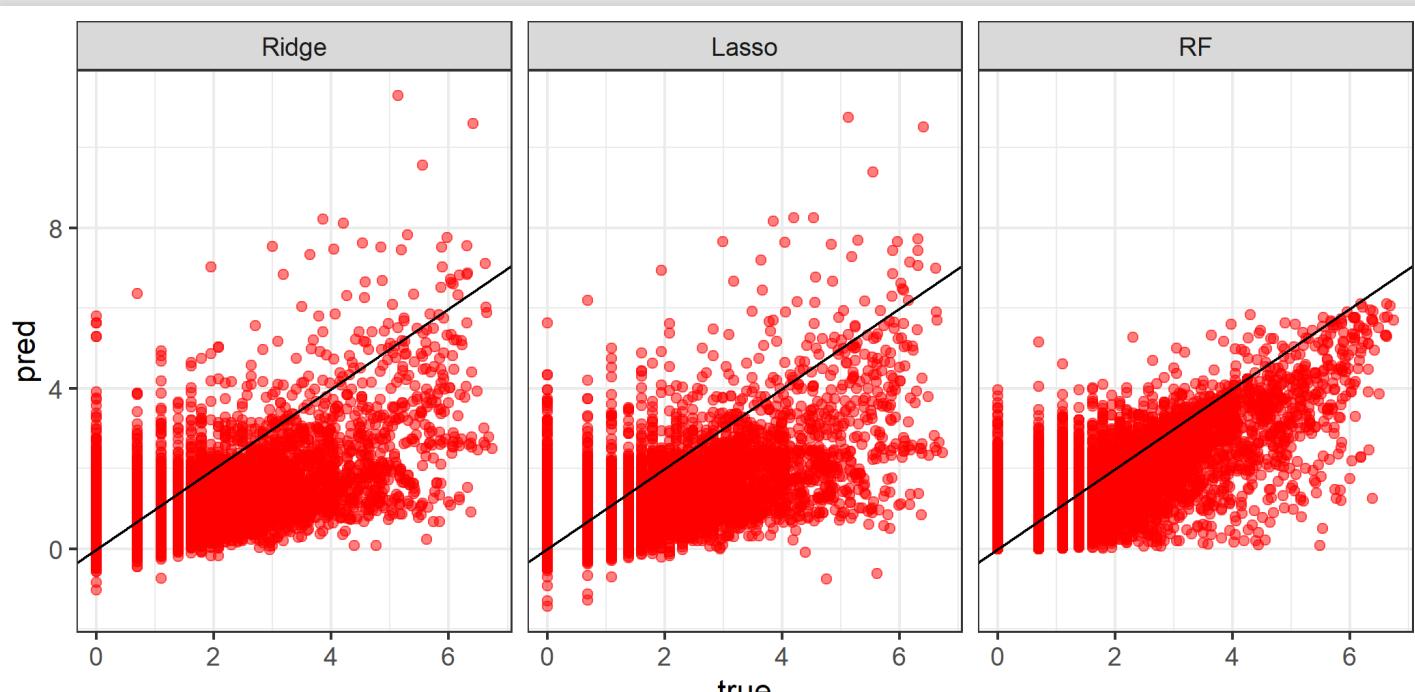
```
pred_ridge <- predict(mod_ridge, newdata = blogs_te)
pred_lasso <- predict(mod_lasso, newdata = blogs_te)
pred_rf <- predict(mod_rf, newdata = blogs_te)

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: 0.85
## Test RMSE Lassoe: 0.835
## Test RMSE RF: 0.641
```

```
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](#)

APPLICATIONS



OF DATA SCIENCE

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)

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

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

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

print(blogs_split_obj)

## <Analysis/Assess/Total>
## <31438/20959/52397>
```

Preprocess (but we're not gonna use it)

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

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

```
## Recipe  
##  
## Inputs:  
##  
##       role #variables  
##       outcome          1  
##   predictor        280
```

`recipes` contains more preprocessing step_s than you imagine:

```
blogs_rec <- blogs_rec %>%  
  step_normalize(all_numeric())
```

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

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

```
## Recipe  
##  
## Inputs:  
##  
##      role #variables  
##      outcome          1  
## predictor        280  
##  
## Training data contained 31438 data points and no missing data.  
##  
## Operations:  
##  
## Centering and scaling for X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, :  
##
```

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

```
blogs_rec$var_info
```

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

```
blogs_rec$steps[[1]]$means |> head()
```

```
##          X1          X2          X3          X4          X5          X6
## 39.5743447 46.8934480 0.4058464 339.6720847 24.7984764 15.2661519
```

```
blogs_rec$steps[[1]]$sds |> head()
```

```
##          X1          X2          X3          X4          X5          X6
## 79.727511 62.429599 8.297409 440.376463 70.466659 32.499697
```

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$X51))
      mean of comments in baked training: {format(mean(blogs_tr2$X51))

## mean of comments in orig training: 39.5, sd: 113
## mean of comments in baked training: 0, 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: 39.4, sd: 109
## mean of comments in baked testing: -0, sd: 0.963
```

Or you can do it all in a single pipe:

```
blogs_rec <- recipe(fb ~ ., data = blogs_tr) %>%
  step_normalize(all_numeric()) %>%
  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$x51))
      mean of comments in baked training: {format(mean(blogs_tr2$x51))

## mean of comments in orig training: 39.5, sd: 113
## mean of comments in baked training: 0, 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: 39.4, sd: 109
## mean of comments in baked testing: -0, sd: 0.963
```

Fast Forward 10 weeks from now...

```
rec_int_topoints <- 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_outcomes())
  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
##
## Fit time: 771ms
##
## Call: glmnet::glmnet(x = maybe_matrix(x), y = y, family = "gaussian",
## 
##          Df  %Dev Lambda
## 1    276  0.00 600.90
## 2    276  2.53 547.50
## 3    276  2.76 498.90
## 4    276  3.00 454.60
## 5    276  3.27 414.20
## 6    276  3.55 377.40
## 7    276  3.85 343.90
## 8    276  4.18 313.30
## 9    276  4.53 285.50
## 10   276  4.91 260.10
## 11   276  5.31 237.00
## 12   276  5.73 216.00
## 13   276  6.18 196.80
## 14   276  6.66 179.30
## 15   276  7.17 163.40
## 16   276  7.70 148.90
## 17   276  8.26 135.60
```

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
##
## Fit time: 1.1s
##
## Call: glmnet::glmnet(x = maybe_matrix(x), y = y, family = "gaussian",
##
##          Df  %Dev  Lambda
## 1      0  0.00 0.60090
## 2      1  4.77 0.54750
## 3      1  8.73 0.49890
## 4      3 12.52 0.45460
## 5      3 16.28 0.41420
## 6      3 19.40 0.37740
## 7      3 21.99 0.34390
## 8      3 24.14 0.31330
## 9      4 26.03 0.28550
## 10     4 28.45 0.26010
## 11     4 30.45 0.23700
## 12     4 32.12 0.21600
## 13     4 33.51 0.19680
```

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
##
## Fit time: 43.8s
## 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):    0.4372567
##   ## R squared (OOB):                0.6598233
```

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 x 3
##   term      estimate  penalty
##   <chr>     <dbl>    <dbl>
## 1 (Intercept) 0.681    0.001
## 2 x1         0.00120   0.001
## 3 x2         0.00124   0.001
## 4 x3        -0.00152   0.001
## 5 x4         0.000314  0.001
## 6 x5         0.000230  0.001
## 7 x6         0.00211   0.001
## 8 x7        -0.000994  0.001
## 9 x8         0.148    0.001
## 10 x9        -0.000129 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
```

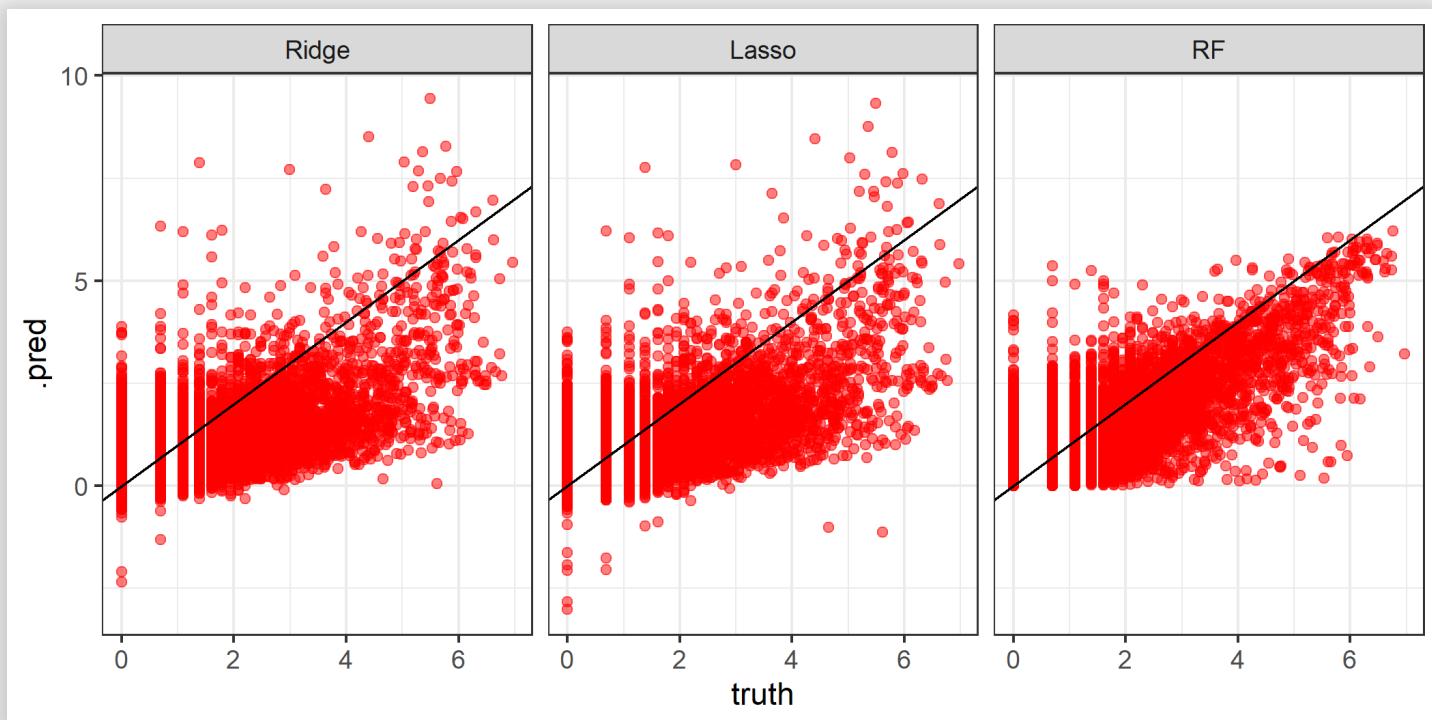
Comparing Models

The package `yardstick` has tons of performance [metrics](#):

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

```
## # A tibble: 3 x 4
##   method .metric .estimator .estimate
##   <chr>   <chr>    <chr>        <dbl>
## 1 Lasso   rmse     standard     0.825
## 2 RF      rmse     standard     0.636
## 3 Ridge   rmse     standard     0.838
```

```
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 x 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 x 4
##   splits              id     .metrics      .notes
##   <list>            <chr>  <list>        <list>
## 1 <split [25150/6288]> Fold1 <tibble [12 x 6]> <tibble [0 x 1]>
## 2 <split [25150/6288]> Fold2 <tibble [12 x 6]> <tibble [0 x 1]>
## 3 <split [25150/6288]> Fold3 <tibble [12 x 6]> <tibble [0 x 1]>
## 4 <split [25151/6287]> Fold4 <tibble [12 x 6]> <tibble [0 x 1]>
## 5 <split [25151/6287]> Fold5 <tibble [12 x 6]> <tibble [0 x 1]>
```

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

```
## # A tibble: 12 x 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:

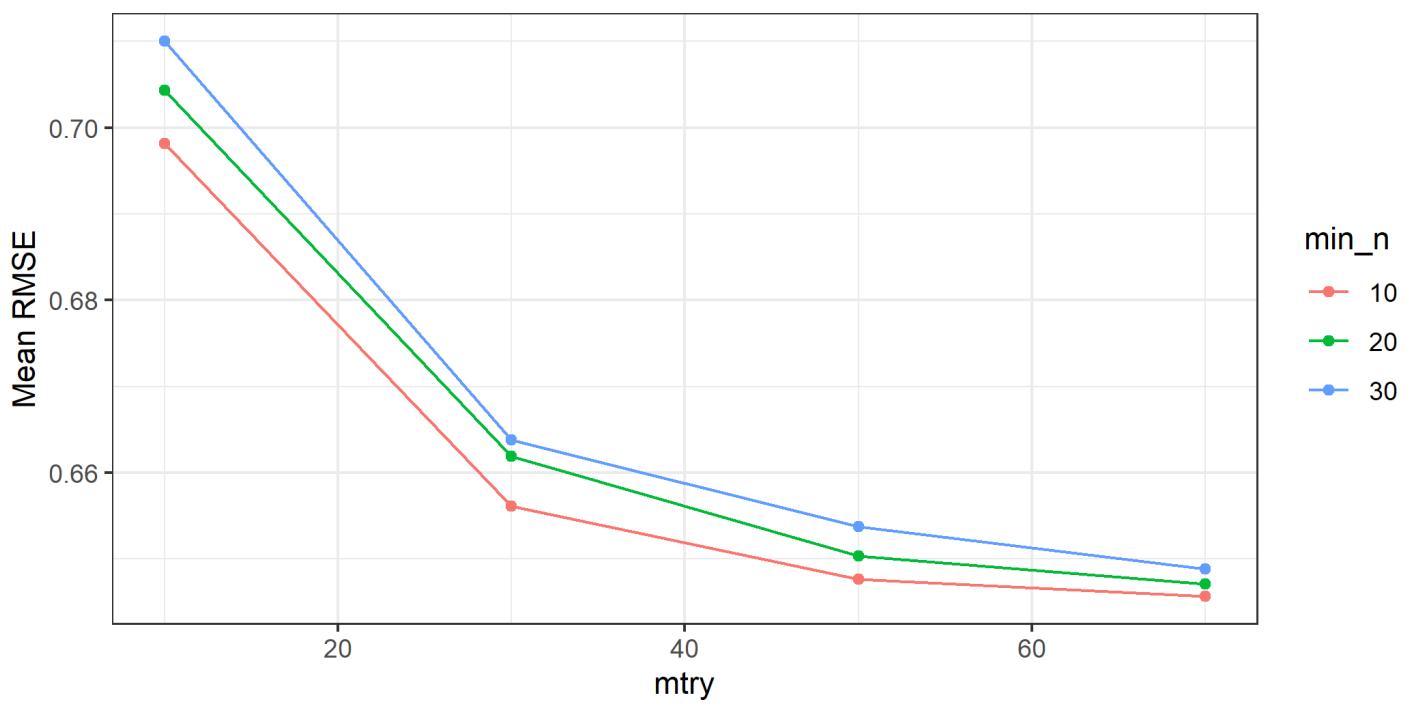
```
estimates <- collect_metrics(tune_res)
```

```
estimates
```

```
## # A tibble: 12 x 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 x 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)
```

```
## # A tibble: 20,959 x 2
##       .pred truth
##   <dbl> <dbl>
## 1 0.858  0.693
## 2 0.337  0
## 3 0.147  0
## 4 2.17   2.30
## 5 3.37   1.10
## 6 2.33   2.64
## 7 1.48   2.77
## 8 1.48   2.77
## 9 2.33   2.64
## 10 1.71   3.09
## # ... with 20,949 more rows
```

infer: Tidy Statistics

APPLICATIONS



OF DATA SCIENCE

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    4842  42896
##       S     456   4203
```

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

```
##          hand
## sunday      left      right
##       NS  0.10142863 0.89857137
##       S   0.09787508 0.90212492
```

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 x 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 = 0.5513, df = 1, p-value = 0.4578  
## alternative hypothesis: two.sided  
## 95 percent confidence interval:  
## -0.005516108 0.012623209  
## sample estimates:  
## prop 1 prop 2  
## 0.10142863 0.09787508
```

```
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 x 9
##   estimate1 estimate2 statistic p.value parameter conf.low conf.high method
##       <dbl>      <dbl>     <dbl>    <dbl>      <dbl>      <dbl>      <dbl> <chr>
## 1     0.101     0.0979    0.551    0.458      1 -0.00552    0.0126 2-s...
```

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

```
## # A tibble: 1 x 10
##   estimate1 estimate2 statistic p.value parameter conf.low conf.high method
##       <dbl>      <dbl>     <dbl>    <dbl>      <dbl>      <dbl>      <dbl> <chr>
## 1     0.0405     0.645     0.605    2.36     0.0183    5639.    0.00686 0...
```

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   4842 42896
##      S    456  4203
```

```
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] 0.003553551
```

Get distribution of the difference in proportions under null hypothesis

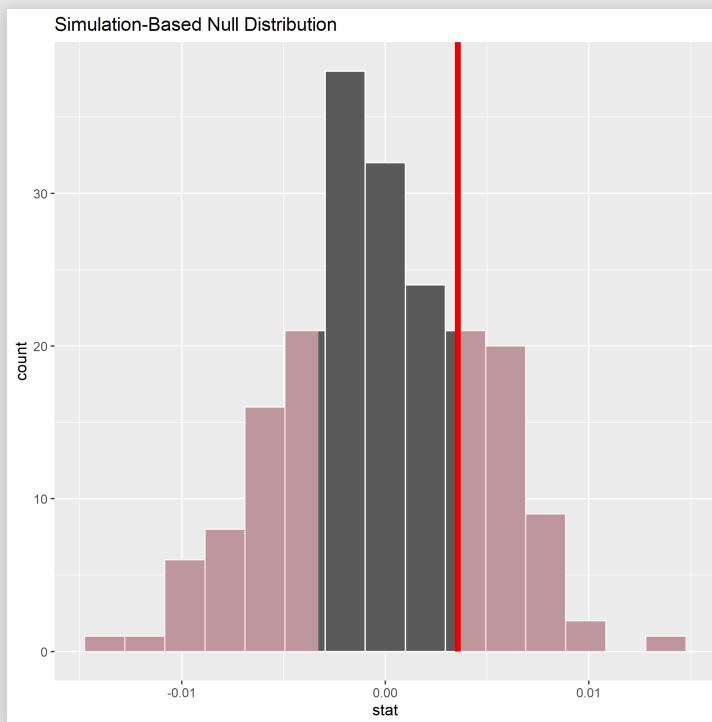
```
diff_null_perm <- blogs_fb %>%
  specify(hand ~ sunday, success = "left") %>%
  hypothesize(null = "independence") %>%
  generate(reps = 200, type = "permute") %>%
  calculate(stat = "diff in props", order = c("NS", "S"))
```

```
diff_null_perm
```

```
## Response: hand (factor)
## Explanatory: sunday (factor)
## Null Hypothesis: independence
## # A tibble: 200 x 2
##       replicate      stat
##          <int>     <dbl>
## 1            1  0.00332
## 2            2  0.00308
## 3            3  0.00143
## 4            4  0.00214
## 5            5 -0.00234
## 6            6  0.00779
## 7            7  0.0000198
## 8            8 -0.00940
## 9            9 -0.00116
## 10           10 -0.000687
## # ... 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 x 1
##   p_value
##   <dbl>
## 1 0.48
```

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 x 1
##       stat
##   <dbl>
## 1  2.36
```

Get distribution of the t statistic under null hypothesis

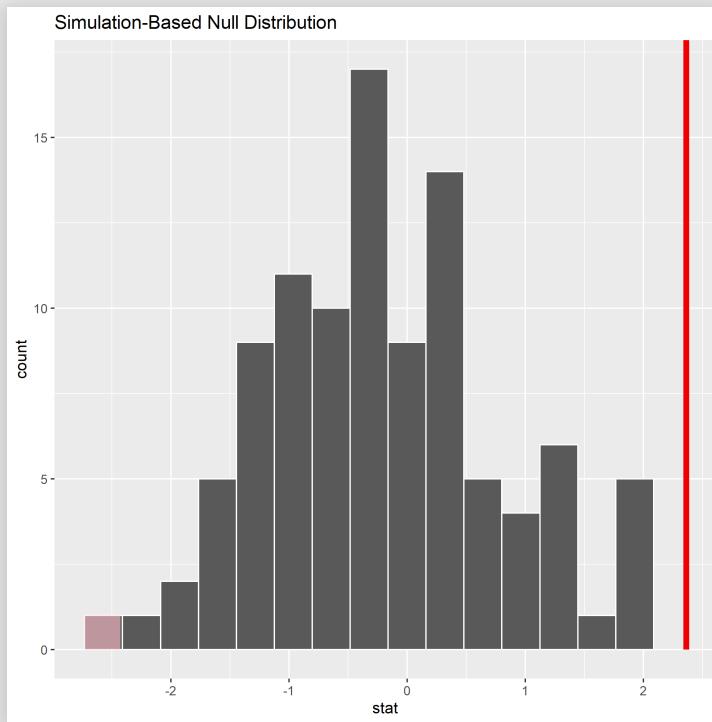
```
t_null_perm <- blogs_fb %>%
  specify(fb ~ sunday) %>%
  hypothesize(null = "independence") %>%
  generate(reps = 100, type = "permute") %>%
  calculate(stat = "t", order = c("NS", "S"))

t_null_perm
```

```
## Response: fb (numeric)
## Explanatory: sunday (factor)
## Null Hypothesis: independence
## # A tibble: 100 x 2
##       replicate   stat
##          <int>   <dbl>
## 1            1 -0.223
## 2            2 -0.392
## 3            3 -0.486
## 4            4 -1.09 
## 5            5  0.0289
## 6            6 -1.99 
## 7            7  0.411 
## 8            8  0.202 
## 9            9  0.161 
## 10           10 -0.453
## # ... 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")

## Warning: Please be cautious in reporting a p-value of 0. This result is a
## approximation based on the number of `reps` chosen in the `generate()` `s
## `?get_p_value()` for more information.

## # A tibble: 1 × 1
##   p_value
##   <dbl>
## 1 0
```