# Intro to `tidymodels` with `nflfastR`

Tom Mock: `@thomas_mock`

2021-09-07

# What have I done?

- Bachelor's in Kinesiology (2011) - Effect of Sugar vs Sugar-Free Mouth Rinse on Performance
- Master's in Exercise Physiology (2014) - Effect of Exercise on Circulating Brain Biomarkers
- PhD in Neurobiology (2018) - Effect of Aging + Glutathione Deficiency on Motor and Cognitive Function

# What do I do?

- RStudio, Customer Enablement Lead - Help RStudio's customers utilize Open Source data science and our Professional Products

- `#TidyTuesday` - Weekly data analysis community project (~4,000 participants in past 3 years)

- TheMockUp.blog - Explanatory blogging about `How to do Stuff` with data + `#rstats`, *mostly* with NFL data

- `espnscrapeR` - collect data from ESPN's public APIs, mostly for QBR and team standings

- `gtExtras` - User-focused wrappers extensions to the `gt` package

# Focus for Today

90 Minutes (with breaks)

Binary classification:

- Logistic Regression
- Random Forest

Slides at: cmsac-tidymodels.netlify.app/
Source code at: github.com/jthomasmock/nfl-workshop

To follow along, you can read in the subsetted data with the code below:

```
raw_url <- "https://github.com/jthomasmock/nfl-workshop/blob/master/raw_plays.rds?raw=true"
raw_plays <- readRDS(url(raw_url, method = "libcurl"))
```

# Level-Setting

As much as I'd love to learn and teach *all* of Machine Learning/Statistics in 90 min...

It's just not possible!

# Goals for today

- Make you comfortable with the syntax and packages via `tidymodels` unified interface
- So when you're learning or modeling on your own, you get to focus on the stats rather than re-learning different APIs over and over...

Along the way, we'll cover minimal examples and then some more quick best practices where `tidymodels` makes it easier to do more things!

# tidymodels

tidymodels is a collection of packages for modeling and machine learning using tidyverse principles.

## Packages

- rsample: efficient data splitting and resampling
- parsnip: tidy, unified interface to models
- recipes: tidy interface to data pre-processing tools for feature engineering
- workflows: bundles your pre-processing, modeling, and post-processing
- tune: helps optimize the hyperparameters and pre-processing steps
- yardstick: measures the performance metrics
- dials: creates and manages tuning parameters/grids
- broom: converts common R statistical objects into predictable formats
  - broom available methods

# `tidymodels` vs `broom` *alone*

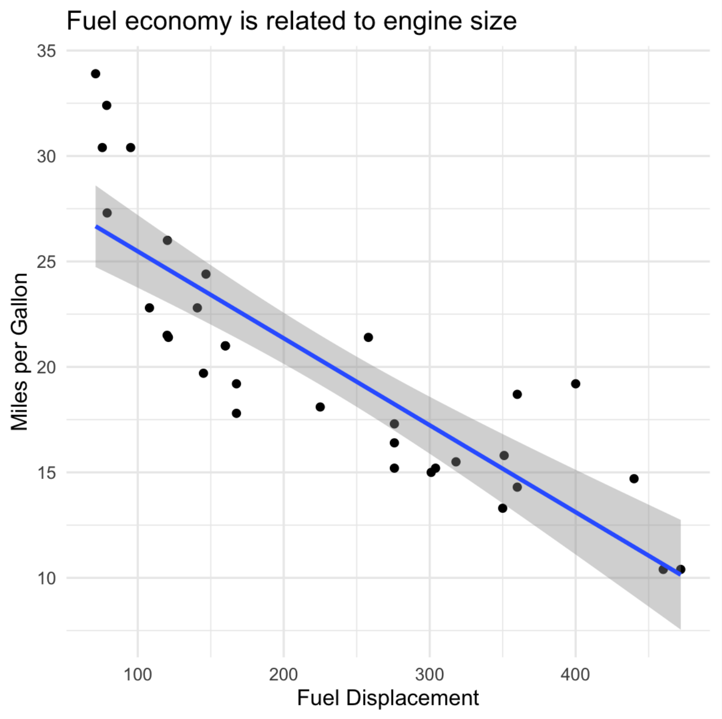
# `broom`

> `broom` summarizes key information about models in tidy `tibble()`s.
>
> `broom` tidies 100+ models from popular modelling packages and almost all of the model objects in the `stats` package that comes with base R. `vignette("available-methods")` lists method availability.
>
> While broom is useful for summarizing the result of a single analysis in a consistent format, it is really designed for high-throughput applications, where you must combine results from multiple analyses.

I *personally* use `broom` for more classical statistics and `tidymodels` for machine learning. A more detailed summary of what `broom` is about can be found in the `broom` docs.

# `lm()` example

```r
library(tidyverse)

basic_plot <- mtcars %>%
  ggplot(
    aes(x = disp, y = mpg)
    ) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_minimal() +
  labs(
    x = "Fuel Displacement", y = "Miles per Ga
    title = "Fuel economy is related to engine
    )
```



# base example

```r
# fit a basic linear model
basic_lm <- lm(mpg ~ disp, data = mtcars)
```

```r
basic_lm
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Coefficients:
## (Intercept)         disp
##    29.59985     -0.04122
```

```r
summary(basic_lm)
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.8922 -2.2022 -0.9631  1.6272  7.2305
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
## disp        -0.041215   0.004712  -8.747 9.38e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.251 on 30 degrees of freedom
## Multiple R-squared:  0.7183,    Adjusted R-squared:  0.709
## F-statistic: 76.51 on 1 and 30 DF,  p-value: 9.38e-10
```

# broom example

```
broom::tidy(basic_lm)
```

```
## # A tibble: 2 × 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)  29.6      1.23        24.1  3.58e-21
## 2 disp         -0.0412   0.00471     -8.75 9.38e-10
```

```
broom::glance(basic_lm)
```

```
## # A tibble: 1 × 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1     0.718         0.709  3.25      76.5 9.38e-10     1  -82.1  170.  175.
## # … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

## Want more `broom`?

There's a lot more to `broom` for *tidy-ier* modeling - out of scope for today, but details at broom.tidymodels.org and R4DS Many Models Chapter.

# Tidy Machine Learning w/ `tidymodels`

# Core ideas for Today

A workflow for `tidy` machine learning

- Split the data
- Pre-Process and Choose a Model
- Combine into a Workflow
- Generate Predictions and Assess Model Metrics

# Goal of Machine Learning

🔨 construct models that

🎯 generate accurate predictions

🆕 for future, yet-to-be-seen data

Feature Engineering - Max Kuhn and Kjell Johnston and Alison Hill

# Classification

Showing two examples today, comparing their outcomes, and then giving you the chance to explore on your own!

# The Dataset

Filtered down from the `nflfastR` datasets (~2.25 GB) to only non-penalty run and pass plays for the 2017-2019 regular seasons, and on downs 1st, 2nd or 3rd. This is about 92,000 plays.

### The goal: Predict if an upcoming play will be a `run` or a `pass`

```
glimpse(raw_plays)
```

```
## Rows: 91,976
## Columns: 20
## $ game_id                   <dbl> 2017090700, 2017090700, 2017090700, 2017090…
## $ posteam                   <chr> "NE", "NE", "NE", "NE", "NE", "NE", "NE", "…
## $ play_type                 <chr> "pass", "pass", "run", "run", "pass", "run"…
## $ yards_gained              <dbl> 0, 8, 8, 3, 19, 5, 16, 0, 2, 7, 0, 3, 10, 0…
## $ ydstogo                   <dbl> 10, 10, 2, 10, 7, 10, 5, 2, 2, 10, 10, 10, …
## $ down                      <dbl> 1, 2, 3, 1, 2, 1, 2, 1, 2, 1, 1, 2, 3, 1, 2…
## $ game_seconds_remaining    <dbl> 3595, 3589, 3554, 3532, 3506, 3482, 3455, 3…
## $ yardline_100              <dbl> 73, 73, 65, 57, 54, 35, 30, 2, 2, 75, 32, 3…
## $ qtr                       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
## $ posteam_score             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7, 7…
## $ defteam                   <chr> "KC", "KC", "KC", "KC", "KC", "KC", "KC", "…
## $ defteam_score             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0…
## $ score_differential        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, -7, 7, 7, 7, 7, …
## $ shotgun                   <dbl> 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0…
## $ no_huddle                 <dbl> 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0…
## $ posteam_timeouts_remaining <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3…
## $ defteam_timeouts_remaining <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3…
## $ wp                        <dbl> 0.5060180, 0.4840546, 0.5100098, 0.5529816,…
## $ goal_to_go                <dbl> 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0…
## $ half_seconds_remaining    <dbl> 1795, 1789, 1754, 1732, 1706, 1682, 1655, 1…
```

# Data Prep

We can read in the data from a RDS file.

```
raw_plays <- readRDS(
  url("https://github.com/jthomasmock/nfl-workshop/blob/master/raw_plays.rds?raw=true", method = "l
  )
```

# What is play-by-play data?

Data from ESPN

# Feature Engineering

I added a few features, namely a running total of number of runs/passes pre-snap and what the previous play was.

```r
all_plays <- raw_plays %>%
  group_by(game_id, posteam) %>%
  mutate(
    run = if_else(play_type == "run", 1, 0),
    pass = if_else(play_type == "pass", 1, 0),
    total_runs = if_else(play_type == "run", cumsum(run) - 1, cumsum(run)),
    total_pass = if_else(play_type == "pass", cumsum(pass) - 1, cumsum(pass)),
    previous_play = if_else(posteam == lag(posteam),
                            lag(play_type), "First play of Drive"
    ),
    previous_play = if_else(is.na(previous_play),
                            replace_na("First play of Drive"), previous_play
    )
  ) %>%
  ungroup() %>%
  mutate_at(vars(
    play_type, shotgun, no_huddle,
    posteam_timeouts_remaining, defteam_timeouts_remaining,
    previous_play, goal_to_go
  ), as.factor) %>%
  mutate(
    down = factor(down, levels = c(1, 2, 3), ordered = TRUE),
    qtr = factor(qtr, levels = c(1, 2, 3, 4), ordered = TRUE),
    in_red_zone = if_else(yardline_100 <= 20, 1, 0),
    in_fg_range = if_else(yardline_100 <= 35, 1, 0),
    two_min_drill = if_else(half_seconds_remaining <= 120, 1, 0)
  ) %>%
  mutate(
    in_red_zone = factor(if_else(yardline_100 <= 20, 1, 0)),
    in_fg_range = factor(if_else(yardline_100 <= 35, 1, 0)),
    two_min_drill = factor(if_else(half_seconds_remaining <= 120, 1, 0))
  ) %>%
  select(-run, -pass)
```

# Core ideas for Today

A workflow for `tidy` machine learning

- Split the data
- Pre-Process and Choose a Model
- Combine into a Workflow
- Generate Predictions and Assess Model Metrics

# Split

```
split_data <- initial_split(data, 0.75)

train_data <- training(split_data)
test_data <- testing(split_data)
```

# Pre-Process & choose a model

```r
model_recipe <- recipe(pred ~ predictors, data = train_data)
```

```r
# Choose a model and an engine
lr_mod <- logistic_reg(mode = "classification") %>%
  set_engine("glm")
```

# Combine into a workflow

```
# Combine the model and recipe to the workflow
lr_wflow <- workflow() %>%
  add_recipe(model_recipe) %>%
  add_model(lr_mod)

# Fit/train the model
model_fit <- lr_wflow %>%
  fit(data = train_data)
```

# Predict and get metrics

```
# Get predictions
pred_lr <- predict(pbp_fit_lr, test_data)

# Check metrics
pred_lr %>%
  metrics(truth = pred, .pred_class) %>%
  bind_cols(select(test_data, pred)) %>%
  bind_cols(predict(fit_lr, test_data, type = "prob"))
```

### Split

```r
# Split
split_pbp <- initial_split(all_plays, 0.75, strata = play_type)

# Split into test/train
train_data <- training(split_pbp)
test_data <- testing(split_pbp)
```

### Pre-Process & Choose a model

```r
pbp_rec <- recipe(play_type ~ ., data = train_data)  %>%
  step_rm(half_seconds_remaining) %>% # remove
  step_string2factor(posteam, defteam) %>%  # convert to factors
  update_role(yards_gained, game_id, new_role = "ID") %>%  # add as ID
  step_corr(all_numeric(), threshold = 0.7) %>% # remove auto-correlated
  step_center(all_numeric()) %>%  # substract mean from numeric
  step_zv(all_predictors())  # remove zero-variance predictors

# Choose a model and an engine
lr_mod <- logistic_reg(mode = "classification") %>%
  set_engine("glm")
```

### Combine into a workflow

```r
# Combine the model and recipe to the workflow
lr_wflow <- workflow() %>%
  add_recipe(pbp_rec) %>%
  add_model(lr_mod)

# Fit/train the model
pbp_fit_lr <- lr_wflow %>%
  fit(data = train_data)
```

### Predict and get metrics

```r
# Get predictions
pbp_pred_lr <- predict(pbp_fit_lr, test_data) %>%
  bind_cols(test_data %>% select(play_type)) %>%
  bind_cols(predict(pbp_fit_lr, test_data, type = "prob"))

# Check metrics
pbp_pred_lr %>%
  metrics(truth = play_type, .pred_class)
```

# rsample

# `rsample`

Now that we've created the dataset to use, I'll start with `tidymodels` proper.

`rsample` at a minimum does your train/test split, but also takes care of things like boostrapping, stratification, v-fold cross validation, validation splits, rolling origin, etc.

# Data Splitting w/ `rsample`

Do the initial split and stratify by play type to make sure there are equal ratios of run vs pass in `test` and `train`

```r
split_pbp <- initial_split(all_plays, 0.75, strata = play_type)

split_pbp
```

```
## <Analysis/Assess/Total>
## <68981/22995/91976>
```

```r
# separate the training data
train_data <- training(split_pbp)

# separate the testing data
test_data <- testing(split_pbp)
```

# Test vs Train

Split into `train_data` and `test_data` and then confirm the ratios.

```
train_data %>%
  count(play_type) %>%
  mutate(ratio = n/sum(n))
```

```
## # A tibble: 2 × 3
##   play_type     n ratio
##   <fct>     <int> <dbl>
## 1 pass      40751 0.591
## 2 run       28230 0.409
```

```
test_data %>%
  count(play_type) %>%
  mutate(ratio = n/sum(n))
```

```
## # A tibble: 2 × 3
##   play_type     n ratio
##   <fct>     <int> <dbl>
## 1 pass      13584 0.591
## 2 run        9411 0.409
```

# Model `recipes`

# Add recipe steps with `recipes`

`recipe` steps are changes we make to the dataset, including things like centering, dummy encoding, update columns as ID only, or even custom feature engineering.

```
pbp_rec <- recipe(play_type ~ ., data = train_data)  %>%
  step_rm(half_seconds_remaining) %>% # remove
  step_string2factor(posteam, defteam) %>%  # convert to factors
  # ignore these vars for train/test, but include in data as ID
  update_role(yards_gained, game_id, new_role = "ID") %>%
  # removes vars that have large absolute correlations w/ other vars
  step_corr(all_numeric(), threshold = 0.7) %>%
  step_center(all_numeric()) %>%  # substract mean from numeric
  step_zv(all_predictors())  # remove zero-variance predictors
```

# In recipes vs dplyr/tidyr

Generally:

- In tidyverse, do reshaping or basic cleaning
- In recipes do statistical transformations or other things that are intended for modeling
  - Possible step_??? for many many things!

# usemodels

Relatively early in package life cycle, but helps with boilerplate

```
usemodels::use_ranger(play_type ~ ., train_data)
```

```
## ranger_recipe <-
##   recipe(formula = play_type ~ ., data = train_data) %>%
##   step_string2factor(one_of(posteam, defteam))
##
## ranger_spec <-
##   rand_forest(mtry = tune(), min_n = tune(), trees = 1000) %>%
##   set_mode("classification") %>%
##   set_engine("ranger")
##
## ranger_workflow <-
##   workflow() %>%
##   add_recipe(ranger_recipe) %>%
##   add_model(ranger_spec)
##
## set.seed(66699)
## ranger_tune <-
##   tune_grid(ranger_workflow, resamples = stop("add your rsample object"), grid = stop("add number of candidate points"))
```

# parsnip

# Choose a model and start your engines!

`parsnip` supplies a general modeling interface to the wide world of R models!

```r
# Note that mode = "classification" is the default here anyway!
lr_mod <- logistic_reg(mode = "classification") %>%
  set_engine("glm")
```

# workflows

# Combine into a `workflow`

We can now combine the model and recipe into a `workflow` - this allows us to define exactly what model and data are going into our `fit`/train call.

```
lr_wflow <- workflow() %>%
  add_model(lr_mod) %>% # parsnip model
  add_recipe(pbp_rec)   # recipe from recipes
```

## What is a `workflow`?

A workflow is an object that can bundle together your pre-processing, modeling, and post-processing requests. For example, if you have a `recipe` and `parsnip` model, these can be combined into a workflow. The advantages are:

- You don't have to keep track of separate objects in your workspace.
- The recipe prepping and model fitting can be executed using a single call to `fit()`.
- If you have custom tuning parameter settings, these can be defined using a simpler interface when combined with `tune`.

# Steps so far

- Build a recipe for any pre-processing
- Choose and build a model
- Combine them into a `workflow`

# Fit/train the model with `parsnip`

Now we can move forward with fitting/training the model - this is really a one-liner.

```
pbp_fit_lr <- lr_wflow %>%
  fit(data = train_data) # fit the model against the training data
```

# Predict outcomes with parsnip

After the model has been trained we can compare the training data against the holdout testing data.

```r
pbp_pred_lr <- predict(pbp_fit_lr, test_data) %>%
  # Add back a "truth" column for what the actual play_type was
  bind_cols(test_data %>% select(play_type)) %>%
  # Get probabilities for the class for each observation
  bind_cols(predict(pbp_fit_lr, test_data, type = "prob"))
```

```
## # A tibble: 22,995 × 4
##    .pred_class play_type .pred_pass .pred_run
##    <fct>       <fct>          <dbl>     <dbl>
##  1 run         pass           0.168     0.832
##  2 run         run            0.237     0.763
##  3 pass        run            0.665     0.335
##  4 pass        run            0.801     0.199
##  5 pass        pass           0.701     0.299
##  6 run         run            0.370     0.630
##  7 pass        run            0.743     0.257
##  8 pass        pass           0.861     0.139
##  9 pass        run            0.694     0.306
## 10 run         run            0.400     0.600
## # … with 22,985 more rows
```

# Predict outcomes with `parsnip`

Note that out previous code of `predict() %>% bind_cols() %>% bind_cols()` is really the equivalent to the below:

```
pbp_last_lr <- last_fit(lr_mod, pbp_rec, split = split_pbp)

pbp_last_lr %>%
  pluck(".predictions", 1)
```

```
## # A tibble: 22,995 × 6
##    .pred_pass .pred_run  .row .pred_class play_type .config
##         <dbl>     <dbl> <int> <fct>       <fct>     <chr>
##  1      0.168     0.832     8 run         pass      Preprocessor1_Model1
##  2      0.237     0.763     9 run         run       Preprocessor1_Model1
##  3      0.665     0.335    10 pass        run       Preprocessor1_Model1
##  4      0.801     0.199    13 pass        run       Preprocessor1_Model1
##  5      0.701     0.299    19 pass        pass      Preprocessor1_Model1
##  6      0.370     0.630    27 run         run       Preprocessor1_Model1
##  7      0.743     0.257    37 pass        run       Preprocessor1_Model1
##  8      0.861     0.139    38 pass        pass      Preprocessor1_Model1
##  9      0.694     0.306    50 pass        run       Preprocessor1_Model1
## 10      0.400     0.600    52 run         run       Preprocessor1_Model1
## # … with 22,985 more rows
```

# Assessing Accuracy with `yardstick`

# Check outcomes with `yardstick`

For confirming how well the model predicts, we can use `yardstick` to plot ROC curves, get AUC and collect general metrics.

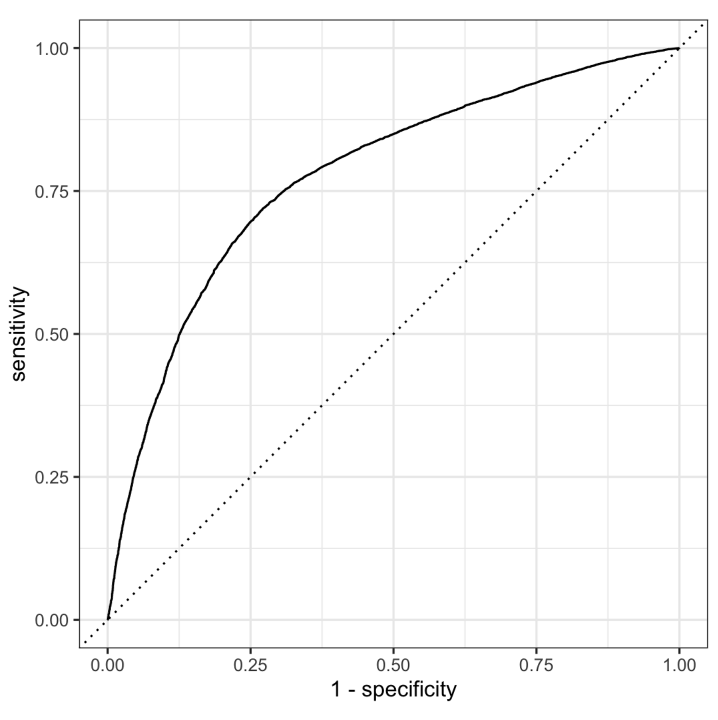
```
pbp_pred_lr %>%
  # get Area under Curve
  roc_auc(truth = play_type,
          .pred_pass)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.776
```

```
pbp_pred_lr %>%
  # collect and report metrics
  metrics(truth = play_type,
          .pred_class)
```

```
## # A tibble: 2 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.722
```

```
pbp_pred_lr %>%
  # calculate ROC curve
  roc_curve(truth = play_type, .pred_pass) %>%
  autoplot()
```



# Note on Checking Outcomes

You *could* use `last_fit()`:

> This functions is intended to be used after fitting a variety of models and the final tuning parameters (if any) have been finalized. The next step would be to fit using the entire training set and verify performance using the test data.

```
lr_last_fit <- last_fit(lr_mod, pbp_rec, split = split_pbp)

collect_metrics(lr_last_fit)
```

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.722 Preprocessor1_Model1
## 2 roc_auc  binary         0.776 Preprocessor1_Model1
```

### Split

```r
# Split
split_pbp <- initial_split(all_plays, 0.75, strata = play_type)

# Split into test/train
train_data <- training(split_pbp)
test_data <- testing(split_pbp)
```

### Pre-Process & Choose a model

```r
pbp_rec <- recipe(play_type ~ ., data = train_data)  %>%
  step_rm(half_seconds_remaining) %>% # remove
  step_string2factor(posteam, defteam) %>%  # convert to factors
  update_role(yards_gained, game_id, new_role = "ID") %>%  # add as ID
  step_corr(all_numeric(), threshold = 0.7) %>% # remove auto-correlated
  step_center(all_numeric()) %>%  # substract mean from numeric
  step_zv(all_predictors())  # remove zero-variance predictors

# Choose a model and an engine
lr_mod <- logistic_reg(mode = "classification") %>%
  set_engine("glm")
```

### Combine into a workflow

```r
# Combine the model and recipe to the workflow
lr_wflow <- workflow() %>%
  add_recipe(pbp_rec) %>%
  add_model(lr_mod)

# Fit/train the model
pbp_fit_lr <- lr_wflow %>%
  fit(data = train_data)
```

### Predict and get metrics

```r
# Get predictions
pbp_pred_lr <- predict(pbp_fit_lr, test_data) %>%
  bind_cols(test_data %>% select(play_type)) %>%
  bind_cols(predict(pbp_fit_lr, test_data, type = "prob"))

# Check metrics
pbp_pred_lr %>%
  metrics(truth = play_type, .pred_class)
```

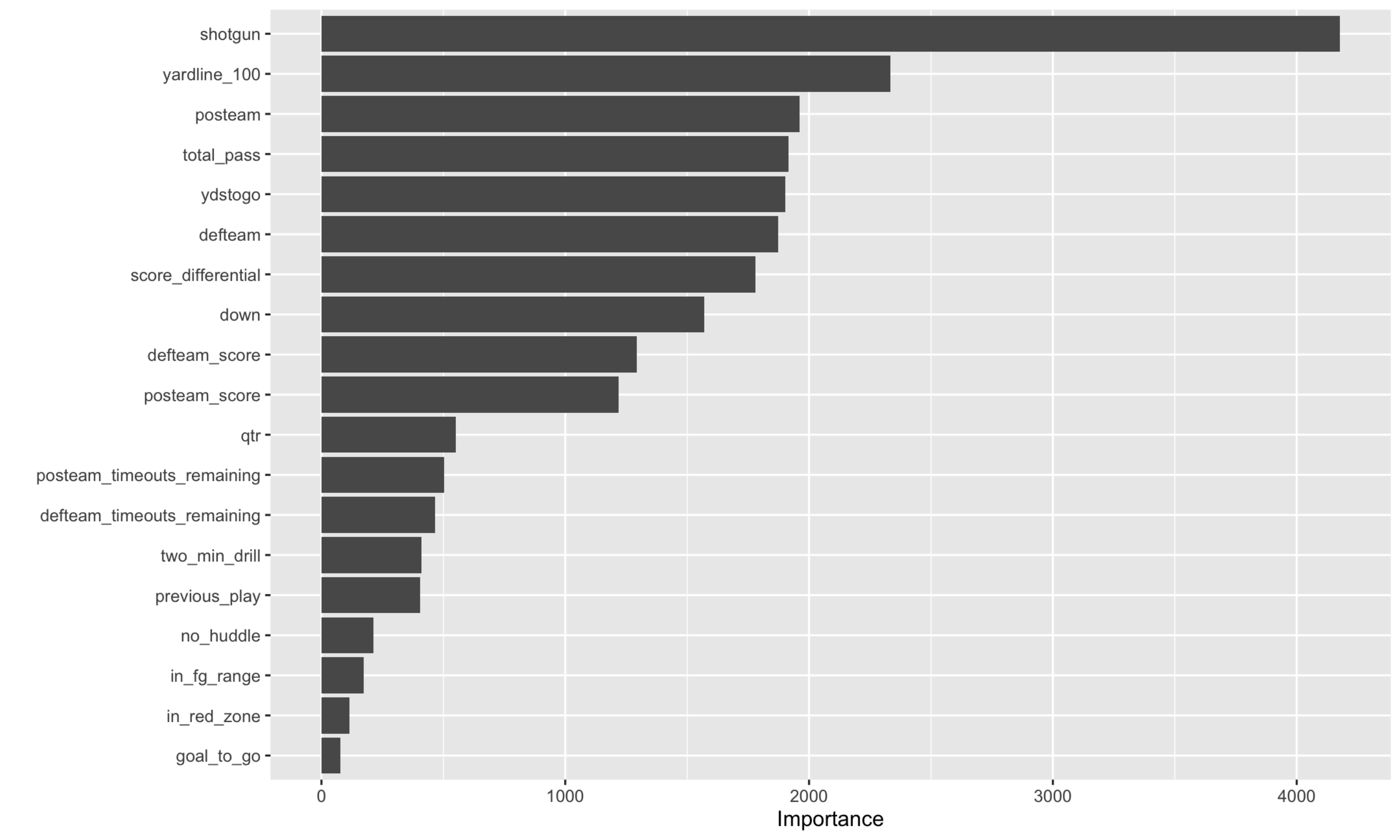
# Change the model

How about a Random Forest model? Just change the model and re-run!

```
rf_mod <- rand_forest(trees = 100) %>%
  set_engine("ranger",
             importance = "impurity", # variable importance
             num.threads = 4) %>%     # Parallelize
  set_mode("classification")

rf_wflow <- workflow() %>%
  add_recipe(pbp_rec) %>% # Same recipe
  add_model(rf_mod)     # New model


pbp_fit_rf <- rf_wflow %>% # New workflow
  fit(data = train_data)   # Fit the Random Forest

# Get predictions and check metrics
pbp_pred_rf <- predict(pbp_fit_rf, test_data) %>%
  bind_cols(test_data %>% select(play_type)) %>%
  bind_cols(predict(pbp_fit_rf, test_data, type = "prob"))
```

# Feature Importance

```
pbp_fit_rf %>%
  pull_workflow_fit() %>%
  vip(num_features = 20)
```



# Quick Model Comparison

The random forest model slightly outperforms the logistic regression, although both are not perfect

```
pbp_pred_lr %>% # Logistic Regression predictions
  metrics(truth = play_type, .pred_class)
```

```
## # A tibble: 2 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.722
## 2 kap      binary         0.416
```

```
pbp_pred_rf %>% # Random Forest predictions
  metrics(truth = play_type, .pred_class)
```

```
## # A tibble: 2 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.729
## 2 kap      binary         0.438
```

# Quick Model Comparison

```
pbp_pred_lr %>% # Logistic Regression predictions
  conf_mat(truth = play_type, .pred_class)
```

```
##            Truth
## Prediction  pass   run
##       pass 10837  3649
##       run   2747  5762
```

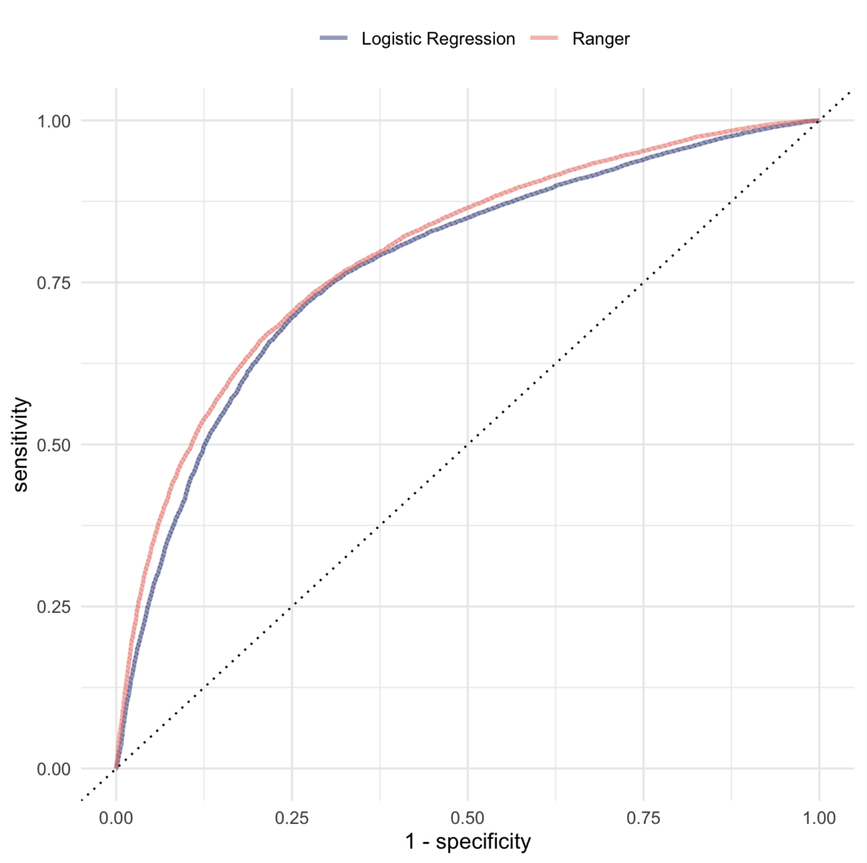
```
pbp_pred_rf %>% # Random Forest predictions
  conf_mat(truth = play_type, .pred_class)
```

```
##            Truth
## Prediction  pass   run
##       pass 10558  3209
##       run   3026  6202
```

# Comparing Models Together

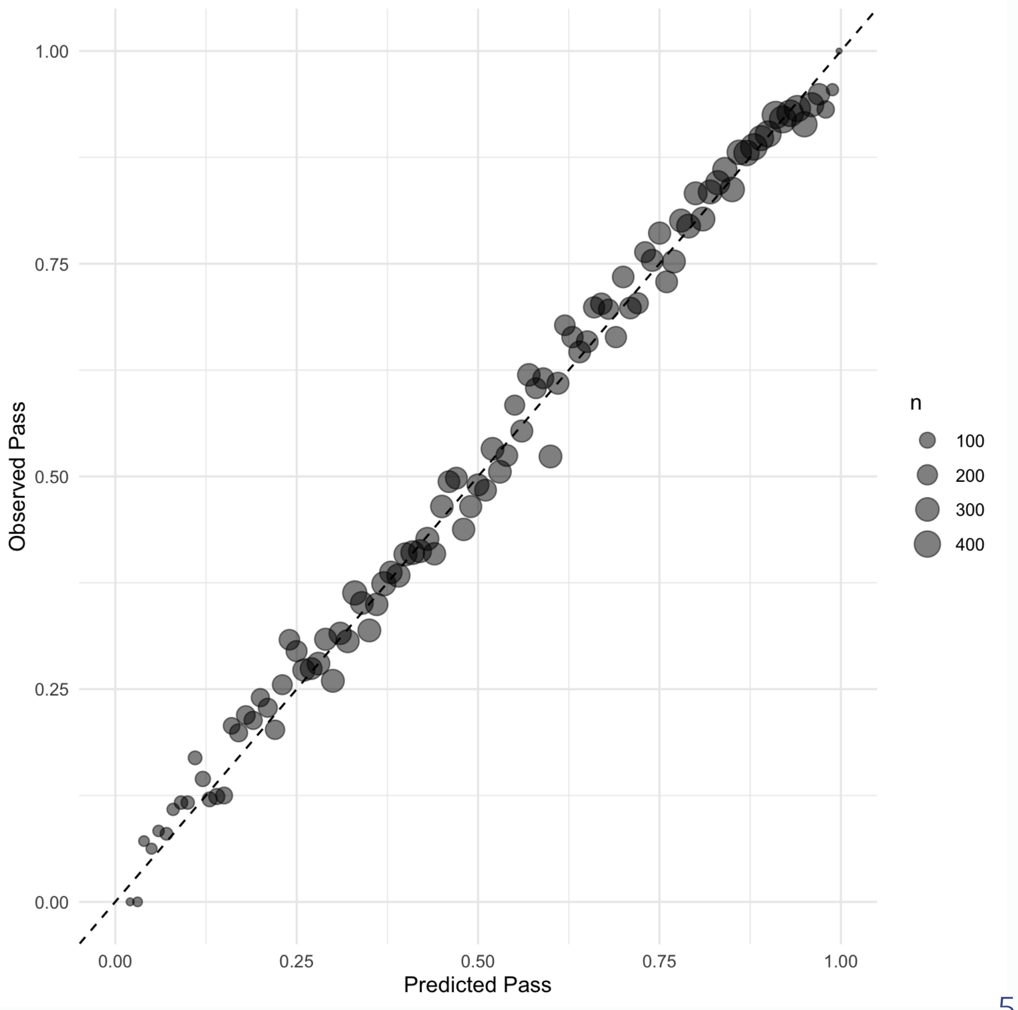
```r
roc_rf <- pbp_pred_rf %>%
  roc_curve(truth = play_type, .pred_pass) %>%
  mutate(model = "Ranger")

roc_lr <- pbp_pred_lr %>%
  roc_curve(truth = play_type, .pred_pass) %>%
  mutate(model = "Logistic Regression")

full_plot <- bind_rows(roc_rf, roc_lr) %>%
  # Note that autoplot() works here!
  ggplot(aes(x = 1 - specificity,
             y = sensitivity,
             color = model)) +
  geom_path(lwd = 1, alpha = 0.5) +
  geom_abline(lty = 3) +
  scale_color_manual(
    values = c("#374785", "#E98074")
    ) +
  theme_minimal() +
  theme(legend.position = "top",
        legend.title = element_blank())
```

```r
full_plot
```



# Calibration Plot

```
calibration_plot <- pbp_pred_rf %>%
  mutate(
    pass = if_else(play_type == "pass", 1, 0),
    pred_rnd = round(.pred_pass, 2)
    ) %>%
  group_by(pred_rnd) %>%
  summarize(
    mean_pred = mean(.pred_pass),
    mean_obs = mean(pass),
    n = n()
    ) %>%
  ggplot(aes(x = mean_pred, y = mean_obs)) +
  geom_abline(linetype = "dashed") +
  geom_point(aes(size = n), alpha = 0.5) +
  theme_minimal() +
  labs(
    x = "Predicted Pass",
    y = "Observed Pass"
    ) +
  coord_cartesian(
    xlim = c(0,1), ylim = c(0, 1)
    )
```



# Quick Re-Cap

A workflow for `tidy` modeling

- Split the data
- Pre-Process and Choose a Model
- Combine into a Workflow
- Generate Predictions and Assess Model Metrics

So the unified interface hopefully makes the idea of learning and applying many algorithms easier.

`tidymodels` *really* shines when you start to go further or apply best practices like:

- Resampling, Cross Validation, Bootstrapping
- Model Tuning and Model Optimization
- Grid Search, Iterative Search

# A Deeper Dive on Best Practices

# Comparing Models

Previously we've just compared two models by seeing how accurate they were on our `testing` data, but....

> The test set as the data that *should* be used to conduct a proper evaluation of model performance on the final model(s). This begs the question of, “How can we tell what is best if we don’t measure performance until the test set?”
>
> However, we usually need to understand the effectiveness of the model *before using the test set.*

- *Tidy Modeling with R*

# Resampling and Cross Validation

> Resampling methods are empirical simulation systems that emulate the process of using some data for modeling and different data for evaluation. Most resampling methods are iterative, meaning that this process is repeated multiple times.
>
> Cross-validation is a well established resampling method

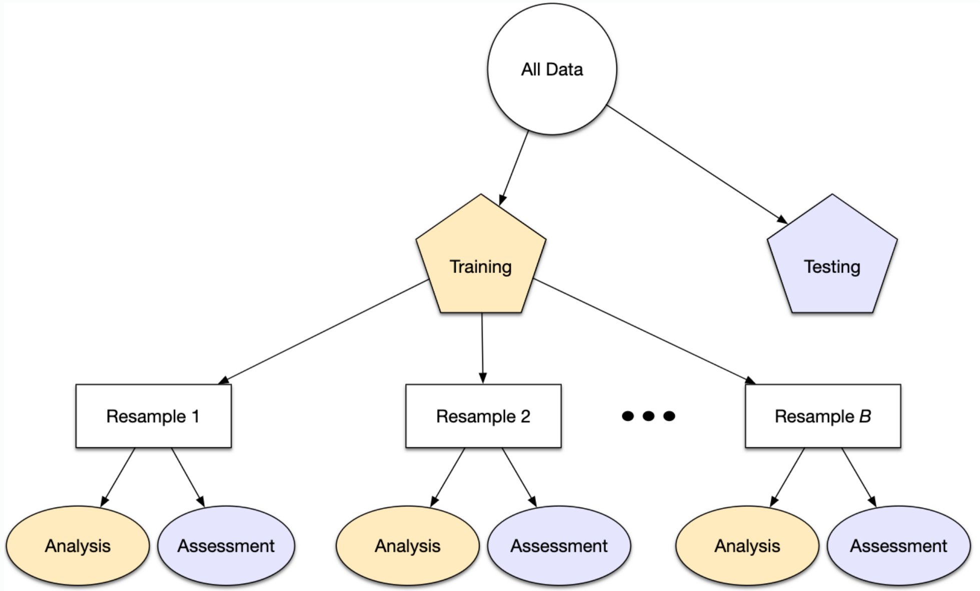
- *Tidy Modeling with R*

Get Started w/ Resampling and test drive on RStudio Cloud.

# Resampling and Cross Validation

> **Resampling is only conducted on the training set**. The test set is not involved. For each iteration of resampling, the data are partitioned into two subsamples:
>
> - The model is fit with the **analysis set**.
> - The model is evaluated with the **assessment set**.

- *Tidy Modeling with R*



# Resampling and Cross-validation

```r
vfold_cv(train_data, v = 10)
```

```
## #  10-fold cross-validation
## # A tibble: 10 × 2
##    splits               id
##    <list>               <chr>
##  1 <split [62082/6899]> Fold01
##  2 <split [62083/6898]> Fold02
##  3 <split [62083/6898]> Fold03
##  4 <split [62083/6898]> Fold04
##  5 <split [62083/6898]> Fold05
##  6 <split [62083/6898]> Fold06
##  7 <split [62083/6898]> Fold07
##  8 <split [62083/6898]> Fold08
##  9 <split [62083/6898]> Fold09
## 10 <split [62083/6898]> Fold10
```

```r
vfold_cv(train_data, v = 10, repeats = 5)
```

```
## #  10-fold cross-validation repeated 5 times
## # A tibble: 50 × 3
##    splits               id      id2
##    <list>               <chr>   <chr>
##  1 <split [62082/6899]> Repeat1 Fold01
##  2 <split [62083/6898]> Repeat1 Fold02
##  3 <split [62083/6898]> Repeat1 Fold03
##  4 <split [62083/6898]> Repeat1 Fold04
##  5 <split [62083/6898]> Repeat1 Fold05
##  6 <split [62083/6898]> Repeat1 Fold06
##  7 <split [62083/6898]> Repeat1 Fold07
##  8 <split [62083/6898]> Repeat1 Fold08
##  9 <split [62083/6898]> Repeat1 Fold09
## 10 <split [62083/6898]> Repeat1 Fold10
## # … with 40 more rows
```

# Estimate Performance w/ Cross Validation

NOTE: Fitting the model multiple times can take a while with larger models or more folds/repeats! I recommend running this as a background job in RStudio, so you don't lock up your session for the duration.

```r
set.seed(20201024)
# Create 10 folds and 5 repeats
pbp_folds <- vfold_cv(train_data, v = 10, repeats = 5)

pbp_folds
```

```
## #  10-fold cross-validation repeated 5 times
## # A tibble: 50 × 3
##    splits               id      id2
##    <list>               <chr>   <chr>
##  1 <split [62084/6899]> Repeat1 Fold01
##  2 <split [62084/6899]> Repeat1 Fold02
##  3 <split [62084/6899]> Repeat1 Fold03
##  4 <split [62085/6898]> Repeat1 Fold04
##  5 <split [62085/6898]> Repeat1 Fold05
##  6 <split [62085/6898]> Repeat1 Fold06
##  7 <split [62085/6898]> Repeat1 Fold07
##  8 <split [62085/6898]> Repeat1 Fold08
##  9 <split [62085/6898]> Repeat1 Fold09
## 10 <split [62085/6898]> Repeat1 Fold10
## # … with 40 more rows
```

# Estimate Performance w/ Cross Validation

```r
keep_pred <- control_resamples(save_pred = TRUE, verbose = TRUE)
set.seed(20201024)
# Fit resamples
rf_res <- fit_resamples(rf_wflow, resamples = pbp_folds, control = keep_pred)

rf_res
```

```
## # Resampling results
## # 10-fold cross-validation repeated 5 times
## # A tibble: 50 × 6
##    splits               id      id2    .metrics         .notes           .predictions
##    <list>               <chr>   <chr>  <list>           <list>           <list>
##  1 <split [62084/6899]> Repeat1 Fold01 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  2 <split [62084/6899]> Repeat1 Fold02 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  3 <split [62084/6899]> Repeat1 Fold03 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  4 <split [62085/6898]> Repeat1 Fold04 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  5 <split [62085/6898]> Repeat1 Fold05 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  6 <split [62085/6898]> Repeat1 Fold06 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  7 <split [62085/6898]> Repeat1 Fold07 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  8 <split [62085/6898]> Repeat1 Fold08 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
##  9 <split [62085/6898]> Repeat1 Fold09 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
## 10 <split [62085/6898]> Repeat1 Fold10 <tibble [2 × 3]> <tibble [0 × 1]> <tibble [6,…
## # … with 40 more rows
```

# What just happened???

We just fit a model for each resample, evaluated it against a within resample assessment set, and stored it into a single `tibble`!

```r
rf_res %>%
  # grab specific columns and resamples
  pluck(".metrics", 1)
```

```
## # A tibble: 2 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.730
## 2 roc_auc  binary         0.794
```

```r
rf_res %>%
  # grab specific columns and resamples
  pluck(".predictions", 10)
```

```
## # A tibble: 6,898 × 5
##    .pred_pass .pred_run  .row .pred_class play_typ
##         <dbl>     <dbl> <int> <fct>       <fct>
##  1      0.552    0.448      8 pass        pass
##  2      0.711    0.289     18 pass        pass
##  3      0.697    0.303     26 pass        run
##  4      0.977    0.0227    41 pass        pass
##  5      0.947    0.0530    48 pass        pass
##  6      0.295    0.705     51 run         run
##  7      0.437    0.563     75 run         run
##  8      0.701    0.299     96 pass        pass
##  9      0.610    0.390    111 pass        run
## 10      0.810    0.190    117 pass        pass
## # … with 6,888 more rows
```

# What else can you do?

```r
# Summarize all metrics
rf_res %>%
  collect_metrics(summarize = TRUE)
```

```
## # A tibble: 2 × 5
##   .metric  .estimator  mean     n  std_err
##   <chr>    <chr>      <dbl> <int>    <dbl>
## 1 accuracy binary     0.730    50 0.000651
## 2 roc_auc  binary     0.797    50 0.000631
```

```r
rf_res %>%
  # combine ALL predictions
  collect_predictions()
```

```
## # A tibble: 344,915 × 7
##    id      id2    .pred_pass .pred_run  .row .pred_class play_type
##    <chr>   <chr>       <dbl>     <dbl> <int> <fct>       <fct>
##  1 Repeat1 Fold01      0.663     0.337     3 pass        run
##  2 Repeat1 Fold01      0.730     0.270    22 pass        pass
##  3 Repeat1 Fold01      0.424     0.576    23 run         pass
##  4 Repeat1 Fold01      0.348     0.652    27 run         run
##  5 Repeat1 Fold01      0.482     0.518    28 run         run
##  6 Repeat1 Fold01      0.695     0.305    35 pass        run
##  7 Repeat1 Fold01      0.381     0.619    61 run         pass
##  8 Repeat1 Fold01      0.166     0.834    69 run         run
##  9 Repeat1 Fold01      0.781     0.219    80 pass        pass
## 10 Repeat1 Fold01      0.333     0.667    84 run         run
## # … with 344,905 more rows
```

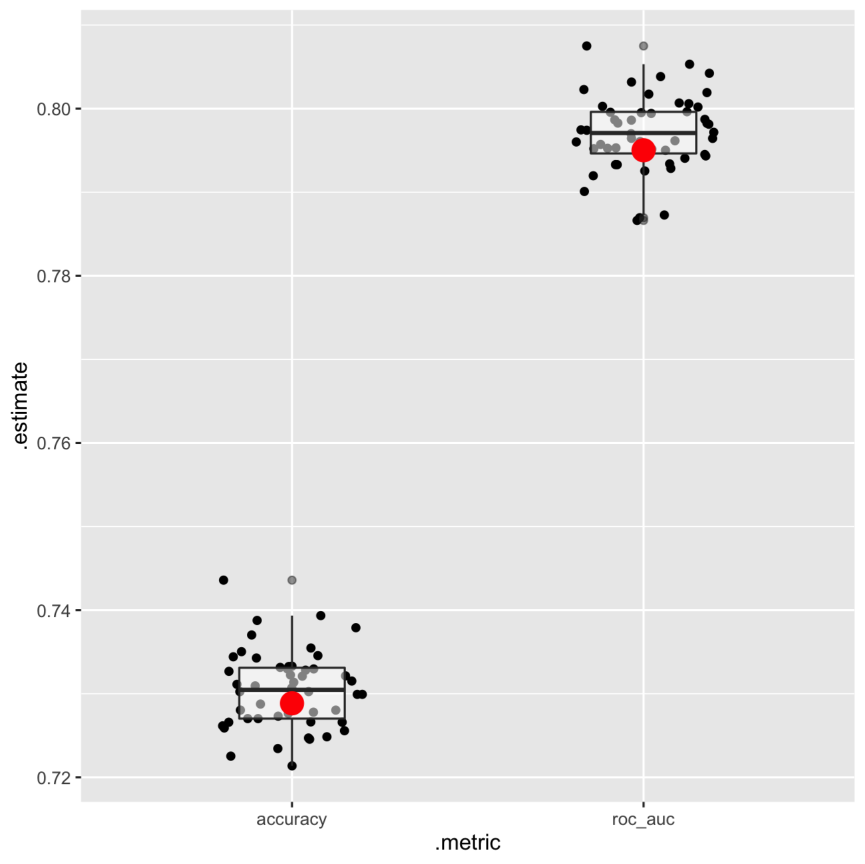
# Collect metrics

First show our predicted model with compared against our test data.

```
# Naive Model on Testing Data
rf_compare_df <- bind_rows(
  accuracy(pbp_pred_rf, truth = play_type, .pred_class),
  roc_auc(pbp_pred_rf, truth = play_type, .pred_pass)
  )
```

And then the what our resampled data looks like, which still would leave our test data as unseen.

```
combo_plot <- rf_res %>%
  collect_metrics(summarize = FALSE) %>%
  ggplot(aes(x = .metric, y = .estimate)) +
  geom_jitter(width = 0.2) +
  geom_boxplot(width = 0.3, alpha = 0.5) +
  geom_point(data = rf_compare_df,color = "red", size = 5)
```



# Estimate Performance w/ Cross Validation

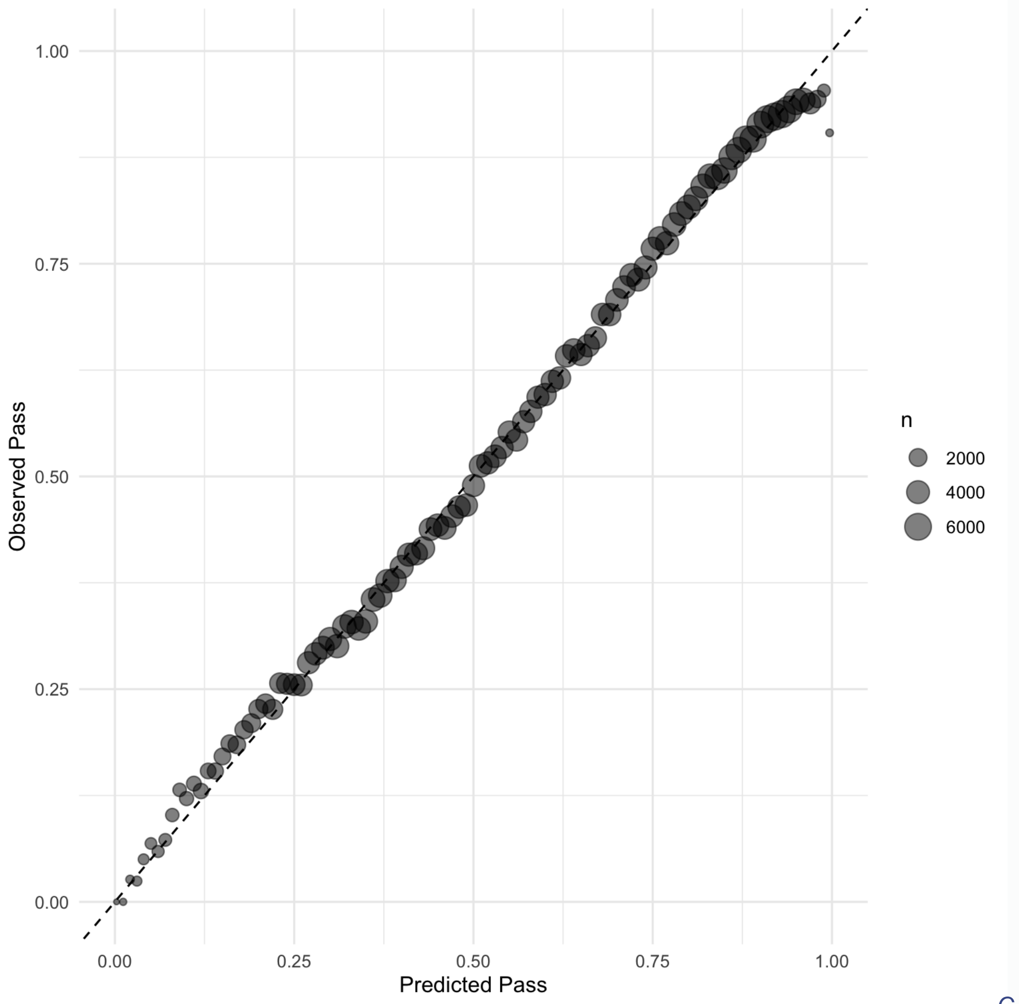
Now, since we aren't supposed to "know" our test results… we can collect our predictions and do another calibration plot. I'm going to round to 3 decimal places and get ~100 data points to plot (instead of our actual ~345,000 points from the combined 50 runs).

```
assess_res <- collect_predictions(rf_res)

assess_res
```

```
## # A tibble: 344,915 × 7
##    id      id2    .pred_pass .pred_run  .row .pred_class play_type
##    <chr>   <chr>       <dbl>     <dbl> <int> <fct>       <fct>
##  1 Repeat1 Fold01      0.663     0.337     3 pass        run
##  2 Repeat1 Fold01      0.730     0.270    22 pass        pass
##  3 Repeat1 Fold01      0.424     0.576    23 run         pass
##  4 Repeat1 Fold01      0.348     0.652    27 run         run
##  5 Repeat1 Fold01      0.482     0.518    28 run         run
##  6 Repeat1 Fold01      0.695     0.305    35 pass        run
##  7 Repeat1 Fold01      0.381     0.619    61 run         pass
##  8 Repeat1 Fold01      0.166     0.834    69 run         run
##  9 Repeat1 Fold01      0.781     0.219    80 pass        pass
## 10 Repeat1 Fold01      0.333     0.667    84 run         run
## # … with 344,905 more rows
```

# Cross Validation Calibration Plot

```
res_calib_plot <- assess_res %>%
  mutate(
    pass = if_else(play_type == "pass", 1, 0),
    pred_rnd = round(.pred_pass, 2)
    ) %>%
  group_by(pred_rnd) %>%
  summarize(
    mean_pred = mean(.pred_pass),
    mean_obs = mean(pass),
    n = n()
    ) %>%
  ggplot(aes(x = mean_pred, y = mean_obs)) +
  geom_abline(linetype = "dashed") +
  geom_point(aes(size = n), alpha = 0.5) +
  theme_minimal() +
  labs(
    x = "Predicted Pass",
    y = "Observed Pass"
    ) +
  coord_cartesian(
    xlim = c(0,1), ylim = c(0, 1)
    )
```



# Model Tuning with `tune`

# tune

We never adjusted our model! We just used naive models and evaluated their performance.

Now, their performance was pretty decent (~68-73% accuracy), but could we get better?

Get Started with Tuning and test drive on RStudio Cloud

# Resample + Tune

We're going to use grid-search for our tuning process, and we also need to specify which hyperparameters of our random forest we want to tune.

> Note: A hyperparameter is a parameter who value is used to control the learning process - Wikipedia)

```
tune_pbp_rf <- rand_forest(
  mtry = tune(), # add placeholder for tune
  trees = 100,
  min_n = tune() # add placeholder for tune
) %>%
  set_mode("classification") %>%
  set_engine("ranger")

tune_rf_wf <- workflow() %>%
  add_recipe(pbp_rec) %>%
  add_model(tune_pbp_rf)
```

```
tune_rf_wf
```

```
## ══ Workflow ════════════════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: rand_forest()
##
## ── Preprocessor ────────────────────────────────────────────────────────────
## 5 Recipe Steps
##
## • step_rm()
## • step_string2factor()
## • step_corr()
## • step_center()
## • step_zv()
##
## ── Model ───────────────────────────────────────────────────────────────────
## Random Forest Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = 100
##   min_n = tune()
##
## Computational engine: ranger
```

# Grid Search

We'll create a grid of possible hyperparameters and then estimate how well they fit with our resamples.

Note that this took about 20 min to run!

I'm doing 15x models by 5x folds, where we train a model and predict outcomes each time! The beauty here is that you could run this as a background job.

```
set.seed(20201024)

pbp_folds <- vfold_cv(train_data, v = 5)

tic()
tune_res <- tune_grid(
  tune_rf_wf,
  resamples = pbp_folds,
  grid = 15, # 15 combos of model parameters
  control = control_grid(verbose = TRUE)
)
toc()
# 1478.385 sec elapsed
```

# Grid Search

Here are the results!

```
tune_res
```

```
## # Tuning results
## # 5-fold cross-validation
## # A tibble: 5 × 4
##   splits                id    .metrics          .notes
##   <list>                <chr> <list>            <list>
## 1 <split [55186/13797]> Fold1 <tibble [30 × 6]> <tibble [0 × 1]>
## 2 <split [55186/13797]> Fold2 <tibble [30 × 6]> <tibble [0 × 1]>
## 3 <split [55186/13797]> Fold3 <tibble [30 × 6]> <tibble [0 × 1]>
## 4 <split [55187/13796]> Fold4 <tibble [30 × 6]> <tibble [0 × 1]>
## 5 <split [55187/13796]> Fold5 <tibble [30 × 6]> <tibble [0 × 1]>
```

# Check it out

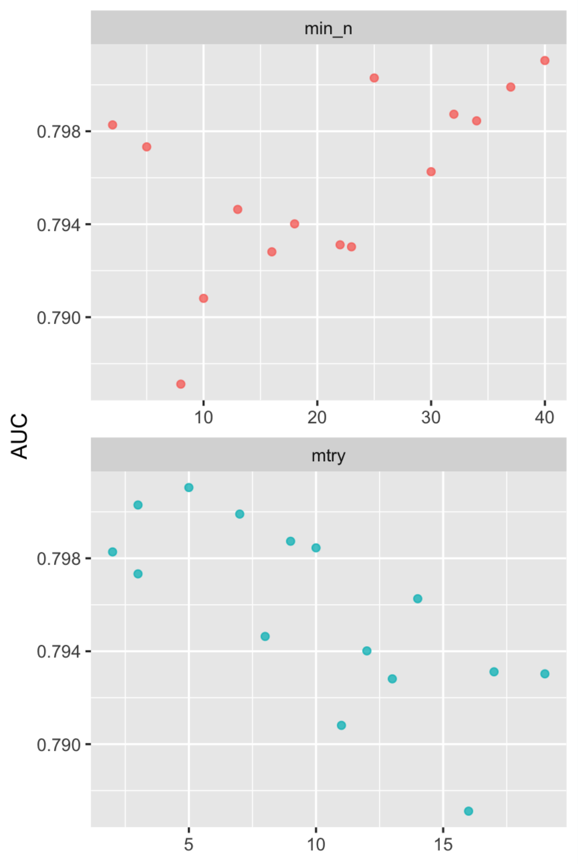
It's nested tibbles for the split data, the fold id, metrics, and any notes.

```
# Essentially the same as tune_res[[".metrics"]][[1]]
tune_res %>%
  pluck(".metrics", 3)
```

```
## # A tibble: 30 × 6
##     mtry min_n .metric  .estimator .estimate .config
##    <int> <int> <chr>    <chr>          <dbl> <chr>
##  1     9    32 accuracy binary         0.735 Model01
##  2     9    32 roc_auc  binary         0.797 Model01
##  3    17    22 accuracy binary         0.729 Model02
##  4    17    22 roc_auc  binary         0.792 Model02
##  5    13    16 accuracy binary         0.727 Model03
##  6    13    16 roc_auc  binary         0.792 Model03
##  7     2     2 accuracy binary         0.735 Model04
##  8     2     2 roc_auc  binary         0.796 Model04
##  9     3    25 accuracy binary         0.737 Model05
## 10     3    25 roc_auc  binary         0.800 Model05
## # … with 20 more rows
```

# Check it out

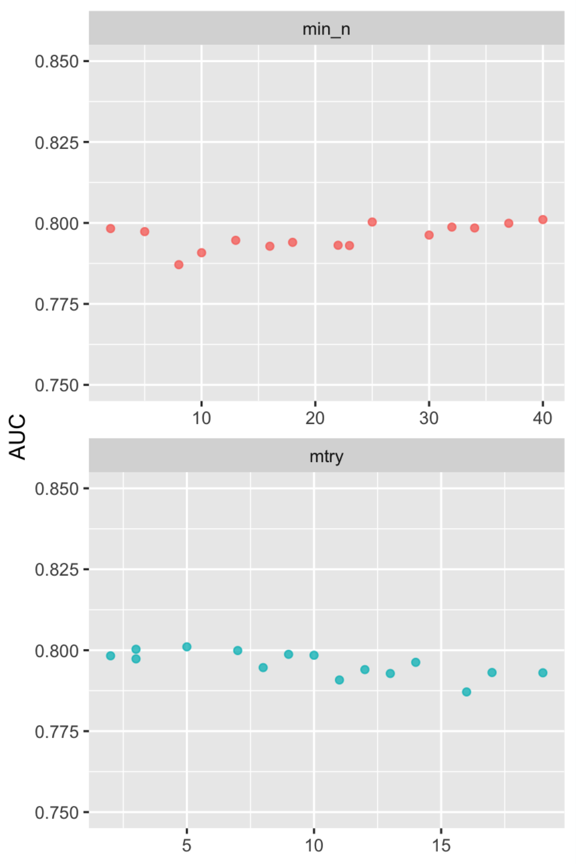
```
plot_tuned <- tune_res %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  dplyr::select(mean, mtry:min_n) %>%
  pivot_longer(mtry:min_n,
               values_to = "value",
               names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE)
  facet_wrap(~parameter, scales = "free_x", nco
  labs(x = NULL, y = "AUC")
```



# Check it out (scaling matters!)

```
plot_tuned <- tune_res %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  dplyr::select(mean, mtry:min_n) %>%
  pivot_longer(mtry:min_n,
               values_to = "value",
               names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE)
  facet_wrap(~parameter, scales = "free_x", nc
  labs(x = NULL, y = "AUC")
```

```
plot_tuned +
  scale_y_continuous(limits = c(0.75, 0.85))
```



# Finalize

Here we are investigating which hyperparameters maximized ROC Area Under the Curve.

```r
# Which 5x were best?
show_best(tune_res, "roc_auc", n = 5)
```

```
## # A tibble: 5 × 8
##    mtry min_n .metric .estimator  mean     n  std_err .config
##   <int> <int> <chr>   <chr>      <dbl> <int>    <dbl> <chr>
## 1     5    40 roc_auc binary     0.801     5 0.000944 Model14
## 2     3    25 roc_auc binary     0.800     5 0.000780 Model05
## 3     7    37 roc_auc binary     0.800     5 0.000926 Model07
## 4     9    32 roc_auc binary     0.799     5 0.00106  Model01
## 5    10    34 roc_auc binary     0.798     5 0.000781 Model06
```

```r
# Select the best
best_fit_auc <- select_best(tune_res, "roc_auc")

# Select wflow for the model with best hyperparams
rf_tuned <- finalize_workflow(
  rf_wflow,
  parameters = best_fit_auc
)
```

# Finalize

Show the outcomes!

```
set.seed(20201024)
rf_tuned_fit <- last_fit(rf_tuned, split_pbp)

rf_tuned_fit %>%  # tuned model metrics
  collect_metrics()
```

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.730 Preprocessor1_Model1
## 2 roc_auc  binary         0.796 Preprocessor1_Model1
```

```
rf_compare_df # naive model metrics
```

```
## # A tibble: 2 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.729
## 2 roc_auc  binary         0.795
```

# Addendums

- Model training/fitting (or simulation) is likely to be the most time-intensive computation you do - as such, it's a good idea to run them as **background jobs** in RStudio
- Also can turn on verbose reporting so you know where you're at in the Cross-validation or tuning steps
  - `control_grid(verbose = TRUE)`

# Going Deeper

Tidy Modeling with R - get started quickly with `tidymodels`

Introduction to Statistical Learning - the 2nd Edition was just released!

Hands on Machine Learning with R - get started quickly with modeling in R (mix of base R, `caret`, and `tidymodels`)

# Thank you

- All y'all for listening in 🤠

# Learn more

- `tidymodels`.org has step by step guides of various complexities
- Julia Silge's (a `tidymodels` maintainer) blog, video series, or free interactive course
- Alison Hill's Workshop from rstudio::conf2020