Random Forests

Joe Nese

Week 8, Class 1

Agenda

- Random forests
- `{workflows}`
- extract

Random Forests

Trees and Bagging

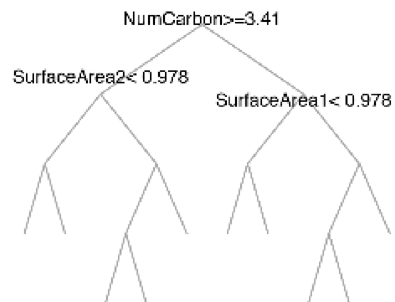
- Single trees do not have great predictive accuracy
  - deep trees: high variance, low bias
  - shallow trees: high bias, low variance
- Bagging trees introduces a random component by building many trees on bootstrapped copies of the training data
- Bagged trees help reduce the variance compared to a single, deep tree
- Bagging aggregates the predictions across all the trees
  - this reduces the variance of the overall procedure and results in improved predictive performance
  - but results in tree correlation that limits the effect of variance reduction

Bagging – tree correlation

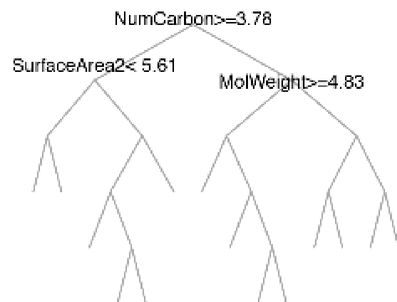
- Trees in a bag are not completely independent since all features are considered at every split of every tree
  - **tree correlation**: trees from different bootstrap samples generally have similar structure to each other (especially at the top of the tree) due to any underlying strong relations
  - prevents bagging from further reducing the variance of the base learner

Limited variance reduction

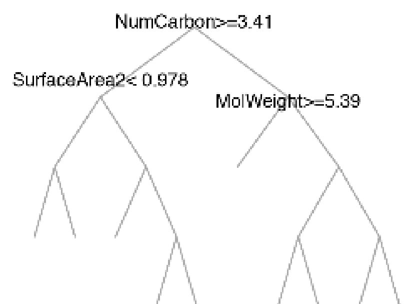
- Suppose there is one very strong predictor in your data, along with other moderately strong predictors
- In a bagged tree, most/all of the trees will use the strong predictor in the top split
    - all trees are quite similar
- Then predictions from the bagged trees will be highly correlated
- Averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities
- This means that bagging will not lead to a substantial reduction in variance over a single tree (in this scenario)



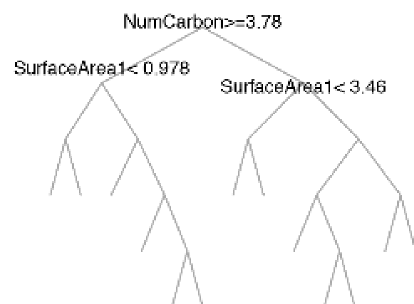
(a) Sample 1



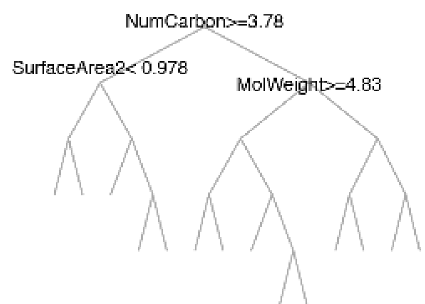
(b) Sample 2



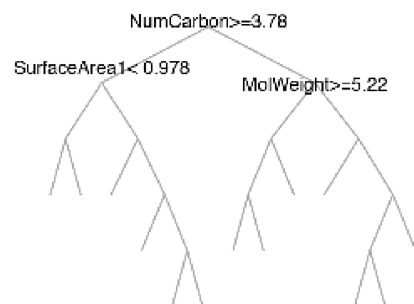
(c) Sample 3



(d) Sample 4



(e) Sample 5



(f) Sample 6

Kuhn & Johnson (2013)

Bagged Tree

- Each tree varies in structure, so predictions vary by tree
- But the first splits are all very similar
- Second-level splits are a bit different, but not much
- So no tree is exactly the same, but they are similar and clearly correlated
- So the solution is to de-correlate the trees

Random Forest

- A random forest can reduce this variance
  - each tree is (more) different, and collectively their decisions will be more accurate
  - reduces tree correlation
- Sum is greater than its parts

Credit to Allison Hill for this analogy

Random Forest

- **Bagging trees** introduces a random component by building many trees on bootstrapped copies of the training data
- **Random forests** introduce another source of randomness that helps reduce tree correlation: **split-variable randomization**
  - Each time a split is to be performed while growing a decision tree, the search for the split variable is limited to a random subset of the original *p* features (**`mtry`**)
  - Everything else about random forests works just as it did with bagging

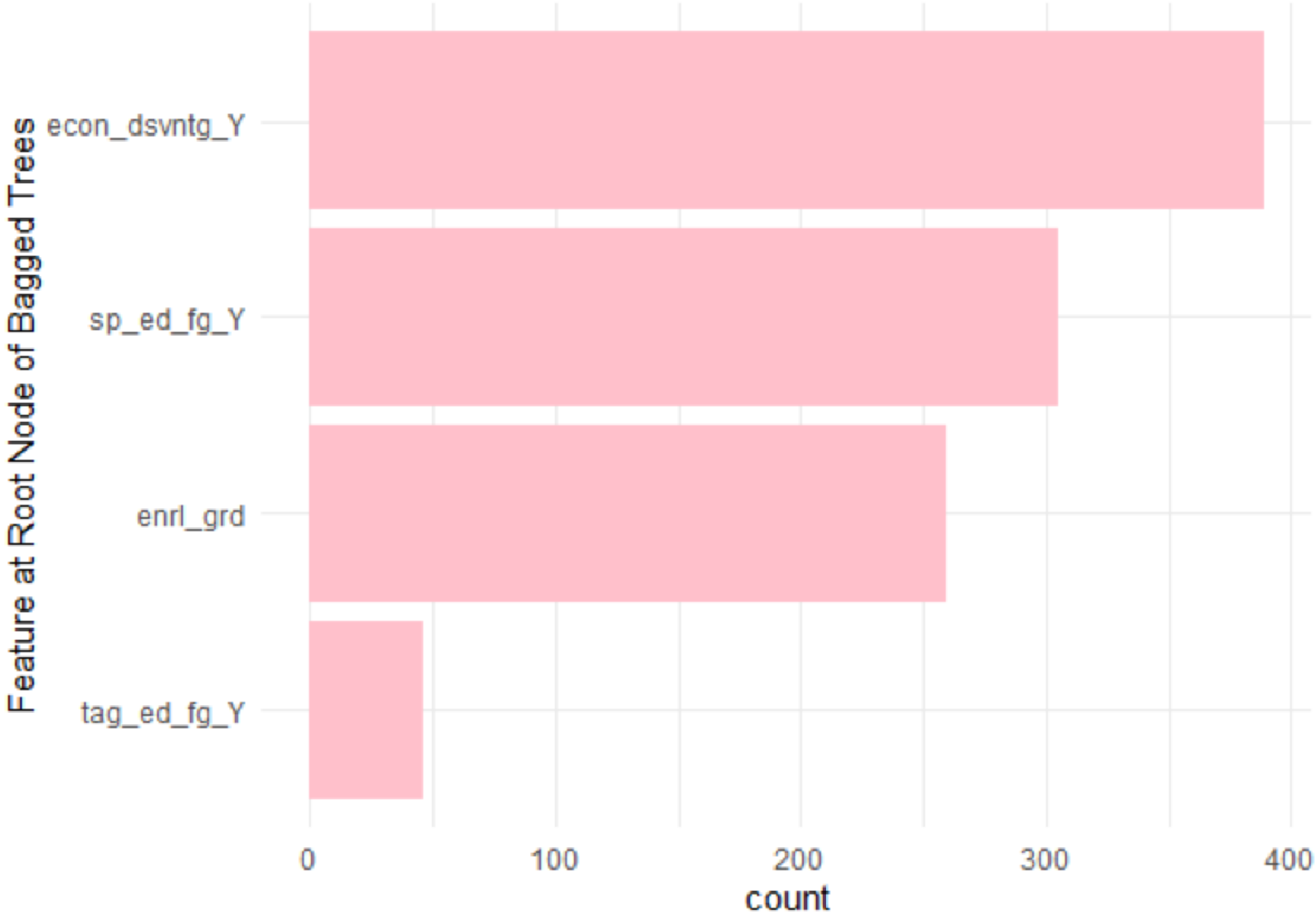
Random Forest

- Random Forest randomly selects a bootstrap sample to train on **and** a random sample of features to use at each split
  - a more diverse set of trees
  - less tree correlation (compared to bagged trees)
  - more predictive power
  - faster than bagging (smaller feature search space at each tree split)
- Each tree in the ensemble is used to generate a prediction for a new sample, and these predictions are aggregated to give the forest’s prediction
- Thus, Random Forests introduce two types of random variation
  - the bootstrapped sample
  - the `mtry` randomly selected predictors

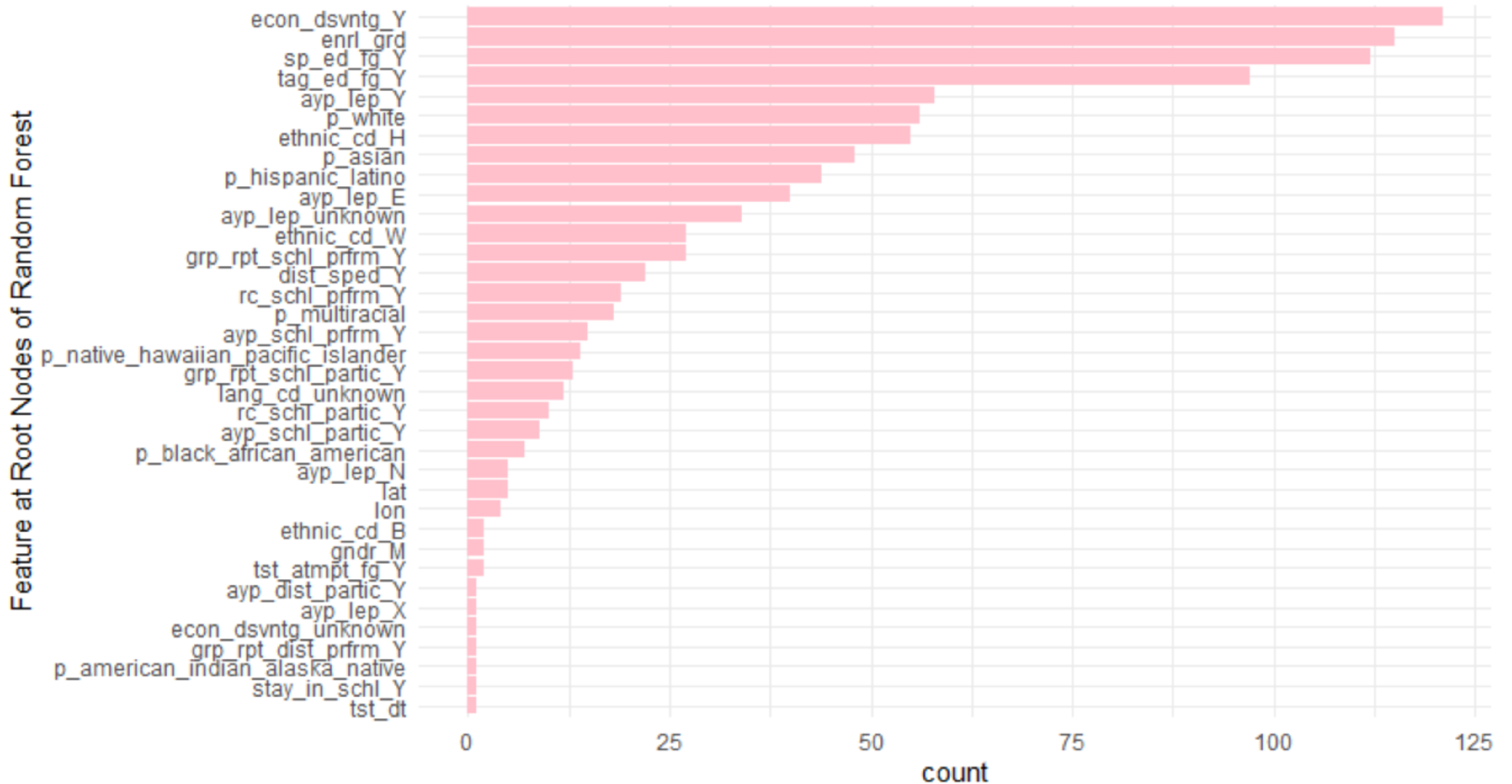
Random Forest

- A subset of `mtry` of the *p* features in the data is selected at random
- Only these `mtry` features are considered for the partition at the node
- Random selection of features reduces the similarity of trees grown from different bootstrap samples—even two trees grown from the same bootstrap sample will likely differ
- At each split in the tree, you're not even using a majority of the available predictors. Why?
  - Suppose there is one very strong predictor in your data, along with other moderately strong predictors
  - On average $(p - mtry)/p$ of the splits won't even consider the strong predictor, so other predictors will have more of a chance to appear at the root
  - This is "decorrelating" the trees
  - Which makes the average of the resulting trees more reliable

The root notes from a **bagged tree** with 1,000 trees (10 folds x 10 hyperparameters x 10 bootstrapped resamples)



The root notes from a **random forest** with 1,0000 trees total (no resampling outside the forest, no tuning)



Random Forest algorithm

1. Given a training data set
2. Select number of trees to build (`trees`)
3. for i = 1 to `trees` do
4. | Generate a bootstrap sample of the original data
5. | Grow a regression/classification tree to the bootstrapped data
6. | for each split do
7. | | Select `mtry` variables at random from all *p* features
8. | | Pick the best variable/split-point among the `mtry`
9. | | Split the node into two child nodes
10. | end
11. | Use typical tree model stopping criteria `min_n` to determine when a tree is complete (but do not prune)
12. end
13. Output ensemble of trees

HML (Boehmke); Max & Johnson (2013)

`rand_forest()`

- `set_engine()`
  - `{ranger}` – default
  - `{randomForest}`
- `set_mode()`
  - “regression”
  - “classification”
- default split method
  - regression = SSE
  - classification = Gini index

Random Forest

- Tend to provide very good performance out-of-the-box
  - default values of tuning parameters tend to produce good result
  - when tuning, have the least variability in prediction accuracy among machine learning algorithms (Probst, Bischl, & Boulesteix, 2018)

tuning parameters

```
rand_forest(mtry = NULL, trees = NULL, min_n = NULL)
```

- `mtry`: number of predictors that will be randomly sampled at each split when creating the tree models
- `trees`: number of trees contained in the ensemble
- `min_n`: minimum number of data points in a node that are required for the node to be split further

`mtry`

- for data with fewer relevant predictors (e.g., noisy data) a higher `mtry` value tends to perform better because it is more likely to select those relevant features
- for data with more relevant predictors, a lower `mtry` may be better
  - fora large number of correlated predictors, a lower `mtry` may be better
- defaults
  - `mtry` = $p/3$ (regression) (Breiman, 2001)
  - `mtry` = $\sqrt{p}$ (classification)
  - `mtry` = p = bagged decision trees (this is what Daniel did last class)
- **Suggestion:** start with five evenly spaced values of `mtry` across the range 2–p centered at the recommended default (Boehmke, 2020; max & Johnson, 2013)

`trees`

- Random forests are protected from overfitting so are not negatively affected by a large number of trees (Brieman, 2001)
- `trees` needs to be sufficiently large to stabilize the error rate
  - more trees provide more robust and stable error estimates and variable importance measures
  - but increases computation time (linearly)
- default
  - `trees` = 500
- **Suggestions**
  - use at least 1,000 trees; if CV performance measures are still improving at 1,000 trees then add trees until performance levels (Max & Johnson, 2013)
  - start with $p \times 10$ trees and adjust as necessary (Boehmke, 2020)

`min_n`

- defaults (Dı'az-Uriarte and De Andres 2006; Goldstein, Polley, and Briggs 2011)
  - `min_n` = 1 (classification)
  - `min_n` = 5 (regression)
- for fewer relevant predictors (e.g., noisy data) and higher `mtry` values, try increasing node size (i.e., decreasing tree depth and complexity)
  - increasing node size will also decrease run time (and perhaps only modestly increase error estimate)
- **Suggestion**: start with three values between 1 to 10 and adjust depending on impact to accuracy and run time (Boehmke, 2020)

```r
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  select(-score) %>%
  sample_frac(.04)

sheets <- readxl::excel_sheets(here::here("data",
                                          "fallmembershipreport_20192020.xlsx"))

ode_schools <- readxl::read_xlsx(here::here("data",
                  "fallmembershipreport_20192020.xlsx"), sheet = sheets[4])

ethnicities <- ode_schools %>%
  select(attnd_schl_inst_id = `Attending School ID`,
         sch_name = `School Name`,
         contains("%")) %>%
  janitor::clean_names()
names(ethnicities) <- gsub("x2019_20_percent", "p", names(ethnicities))

math <- left_join(math, ethnicities)
```

```r
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  select(-score) %>%
  sample_frac(.04)

sheets <- readxl::excel_sheets(here::here("data",
                                    "fallmembershipreport_20192020.xlsx"))

ode_schools <- readxl::read_xlsx(here::here("data",
                  "fallmembershipreport_20192020.xlsx"), sheet = sheets[4])

ethnicities <- ode_schools %>%
  select(attnd_schl_inst_id = `Attending School ID`,
         sch_name = `School Name`,
         contains("%")) %>%
  janitor::clean_names()
names(ethnicities) <- gsub("x2019_20_percent", "p", names(ethnicities))

math <- left_join(math, ethnicities)
```

```r
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  select(-score) %>%
  sample_frac(.04)

sheets <- readxl::excel_sheets(here::here("data",
                                           "fallmembershipreport_20192020.xlsx"))

ode_schools <- readxl::read_xlsx(here::here("data",
                   "fallmembershipreport_20192020.xlsx"), sheet = sheets[4])

ethnicities <- ode_schools %>%
  select(attnd_schl_inst_id = `Attending School ID`,
         sch_name = `School Name`,
         contains("%")) %>%
  janitor::clean_names()
names(ethnicities) <- gsub("x2019_20_percent", "p", names(ethnicities))

math <- left_join(math, ethnicities)
```

```r
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  select(-score) %>%
  sample_frac(.04)

sheets <- readxl::excel_sheets(here::here("data",
                                          "fallmembershipreport_20192020.xlsx"))

ode_schools <- readxl::read_xlsx(here::here("data",
                  "fallmembershipreport_20192020.xlsx"), sheet = sheets[4])

ethnicities <- ode_schools %>%
  select(attnd_schl_inst_id = `Attending School ID`,
         sch_name = `School Name`,
         contains("%")) %>%
  janitor::clean_names()
names(ethnicities) <- gsub("x2019_20_percent", "p", names(ethnicities))

math <- left_join(math, ethnicities)
```

```
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  select(-score) %>%
  sample_frac(.04)

sheets <- readxl::excel_sheets(here::here("data",
                                     "fallmembershipreport_20192020.xlsx"))

ode_schools <- readxl::read_xlsx(here::here("data",
                  "fallmembershipreport_20192020.xlsx"), sheet = sheets[4])

ethnicities <- ode_schools %>%
  select(attnd_schl_inst_id = `Attending School ID`,
         sch_name = `School Name`,
         contains("%")) %>%
  janitor::clean_names()
names(ethnicities) <- gsub("x2019_20_percent", "p", names(ethnicities))

math <- left_join(math, ethnicities)
```

```r
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  select(-score) %>%
  sample_frac(.04)

sheets <- readxl::excel_sheets(here::here("data",
                                     "fallmembershipreport_20192020.xlsx"))

ode_schools <- readxl::read_xlsx(here::here("data",
                  "fallmembershipreport_20192020.xlsx"), sheet = sheets[4])

ethnicities <- ode_schools %>%
  select(attnd_schl_inst_id = `Attending School ID`,
         sch_name = `School Name`,
         contains("%")) %>%
  janitor::clean_names()
names(ethnicities) <- gsub("x2019_20_percent", "p", names(ethnicities))

math <- left_join(math, ethnicities)
```

```r
# Split and Resample

set.seed(3000)
math_split <- initial_split(math, strata = "classification")

set.seed(3000)
math_train <- training(math_split)
math_test <- testing(math_split)

set.seed(3000)
math_cv <- vfold_cv(math_train, strata = "classification")
```

```r
# Split and Resample

set.seed(3000)
math_split <- initial_split(math, strata = "classification")

set.seed(3000)
math_train <- training(math_split)
math_test <- testing(math_split)

set.seed(3000)
math_cv <- vfold_cv(math_train, strata = "classification")
```

```r
# Split and Resample

set.seed(3000)
math_split <- initial_split(math, strata = "classification")

set.seed(3000)
math_train <- training(math_split)
math_test <- testing(math_split)

set.seed(3000)
math_cv <- vfold_cv(math_train, strata = "classification")
```

```
# Recipe

rf_rec <- recipe(classification ~ ., math_train) %>%
  step_mutate(tst_dt = lubridate::mdy_hms(tst_dt)) %>%
  step_mutate(classification = factor(recode(classification,
                                       `1` = "wellbelow",
                                       `2` = "below",
                                       `3` = "above",
                                       `4` = "wellabove"))) %>%
  step_rm(contains("bnch")) %>%
  update_role(contains("id"), ncessch, sch_name, new_role = "id") %>%
  step_novel(all_nominal(), -all_outcomes())  %>%
  step_unknown(all_nominal(), -all_outcomes()) %>%
  step_medianimpute(all_numeric()) %>%
  step_nzv(all_predictors(), freq_cut = 0, unique_cut = 0) %>%
  step_dummy(all_nominal(), -has_role(match = "id"), -all_outcomes()) %>%
  step_nzv(all_predictors())
```

```
prep(rf_rec)

Data Recipe

Inputs:

      role #variables
        id          7
   outcome          1
 predictor         39

Training data contained 5684 data points and 5684 incomplete rows.

Operations:

Variable mutation for tst_dt [trained]
Variable mutation for classification [trained]
Variables removed tst_bnch [trained]
Novel factor level assignment for gndr, ethnic_cd, migrant_ed_fg, ... [trained]
Unknown factor level assignment for gndr, ethnic_cd, migrant_ed_fg, ... [trained]
Median Imputation for id, attnd_dist_inst_id, ... [trained]
Sparse, unbalanced variable filter removed calc_admn_cd [trained]
Dummy variables from gndr, ethnic_cd, migrant_ed_fg, ind_ed_fg, ... [trained]
Sparse, unbalanced variable filter removed 83 items [trained]
```

Default Model

```
# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)
```

```r
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8
```

Default Model

```
# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

```r
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

```
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

```
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

```
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

```
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

```
# Default Model

# mtry = floor(sqrt(p))= floor(sqrt(39))= 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

Default Model

```
rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")

translate(rf_def_mod)
```

```
Random Forest Model Specification (classification)

Engine-Specific Arguments:
  num.threads = cores
  importance = permutation
  verbose = TRUE

Computational engine: ranger

Model fit template:
ranger::ranger(x = missing_arg(), y = missing_arg(), case.weights = missing_arg(),
    num.threads = cores, importance = "permutation", verbose = TRUE,
    seed = sample.int(10^5, 1), probability = TRUE)
```

```r
# Tuned Model

# mtry = tune()
# trees = 1000
# min_n = tune()


rf_tune_mod <- rf_def_mod %>%
  set_args(
    mtry = tune(),
    trees = 1000,
    min_n = tune()
  )
```

```
# Tuned Model

# mtry = tune()
# trees = 1000
# min_n = tune()



rf_tune_mod <- rf_def_mod %>%
  set_args(
    mtry = tune(),
    trees = 1000,
    min_n = tune()
  )
```

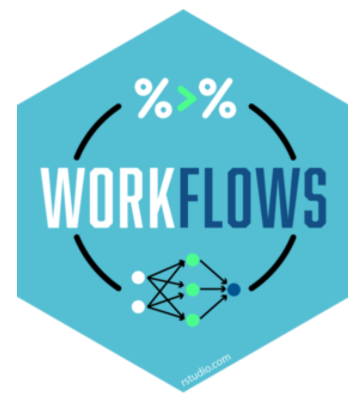
```r
# Tuned Model

rf_tune_mod <- rf_def_mod %>%
  set_args(
    mtry = tune(),
    trees = 1000,
    min_n = tune()
  )

translate(rf_tune_mod)
```

```
Random Forest Model Specification (classification)

Main Arguments:
  mtry = tune()
  trees = 1000
  min_n = tune()

Engine-Specific Arguments:
  num.threads = cores
  importance = permutation
  verbose = TRUE

Computational engine: ranger

Model fit template:
ranger::ranger(x = missing_arg(), y = missing_arg(), case.weights = missing_arg(),
    mtry = min_cols(~tune(), x), num.trees = 1000, min.node.size = min_rows(~tune(), x),
    num.threads = cores, importance = "permutation",
    verbose = TRUE, seed = sample.int(10^5, 1), probability = TRUE)
```

{workflows}

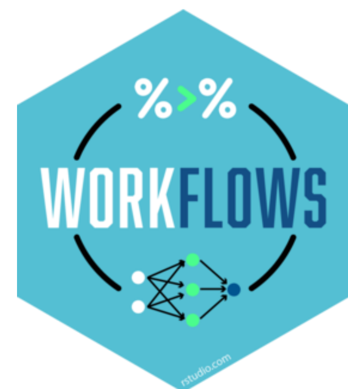
quick detour

{workflows}



- Basically a bundle for your `parsnip` model and `recipe`
- Advantages
  - 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`
  - In the future, workflows will be able to add post-processing operations, such as modifying the probability cutoff for two-class models

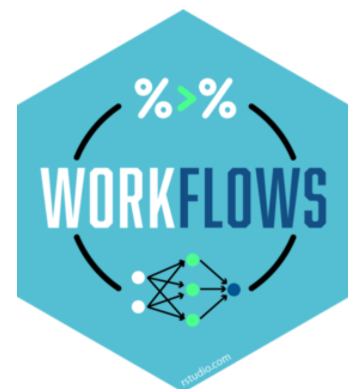
workflows



- Last week we had code like this:

```
rec <- recipe(accuracy_group ~ ., data = train) %>%
  step_mutate(accuracy_group = as.factor(accuracy_group))

mod_random1 <- decision_tree() %>%
  set_mode("classification") %>%
  set_engine("rpart") %>%
  set_args(cost_complexity = 0.01, min_n = 5)

m01 <- fit(mod_random1, accuracy_group ~ ., prep(rec) %>%  juice())
```

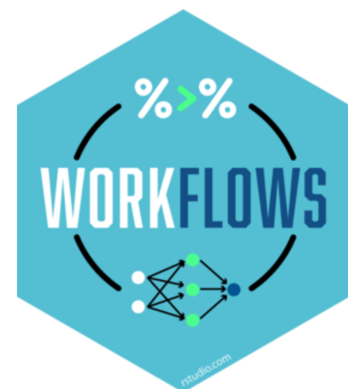
workflows



- Last week we had code like this:

```
rec <- recipe(accuracy_group ~ ., data = train) %>%
  step_mutate(accuracy_group = as.factor(accuracy_group))

mod_random1 <- decision_tree() %>%
  set_mode("classification") %>%
  set_engine("rpart") %>%
  set_args(cost_complexity = 0.01, min_n = 5)

m01 <- fit(mod_random1, accuracy_group ~ ., prep(rec) %>%  juice())
```

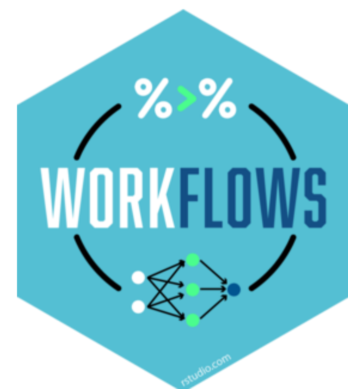
workflows



- Last week we had code like this:

```
rec <- recipe(accuracy_group ~ ., data = train) %>%
  step_mutate(accuracy_group = as.factor(accuracy_group))

mod_random1 <- decision_tree() %>%
  set_mode("classification") %>%
  set_engine("rpart") %>%
  set_args(cost_complexity = 0.01, min_n = 5)

m01 <- fit(mod_random1, accuracy_group ~ ., prep(rec) %>%  juice())
```

workflows



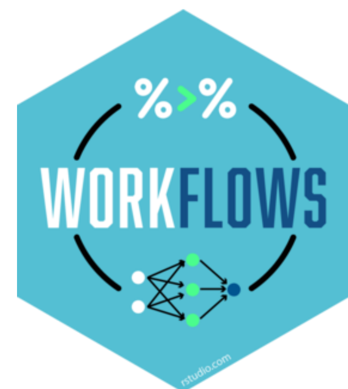
- Last week we had code like this:

```
rec <- recipe(accuracy_group ~ ., data = train) %>%
  step_mutate(accuracy_group = as.factor(accuracy_group))

mod_random1 <- decision_tree() %>%
  set_mode("classification") %>%
  set_engine("rpart") %>%
  set_args(cost_complexity = 0.01, min_n = 5)

m01 <- fit(mod_random1, accuracy_group ~ ., prep(rec) %>%  juice())
```

- Alternate code is:

```
random1_wflow <- workflow() %>%
  add_recipe(rec) %>%
  add_model(mod_random1)
```

workflows



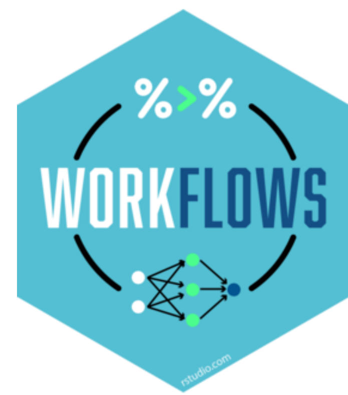
- Last week we had code like this:

```
rec <- recipe(accuracy_group ~ ., data = train) %>%
  step_mutate(accuracy_group = as.factor(accuracy_group))

mod_random1 <- decision_tree() %>%
  set_mode("classification") %>%
  set_engine("rpart") %>%
  set_args(cost_complexity = 0.01, min_n = 5)

m01 <- fit(mod_random1, accuracy_group ~ ., prep(rec) %>%  juice())
```

- Alternate code is:

```
random1_wflow <- workflow() %>%
  add_recipe(rec) %>%
  add_model(mod_random1 )

m01 <- fit(random1_wflow, data = train)
```

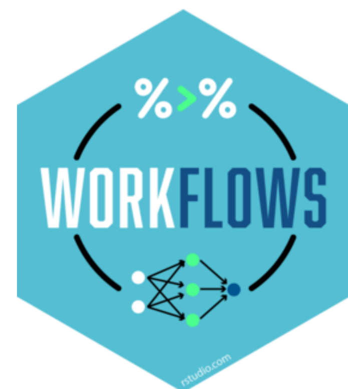
`workflows`



You can alter existing workflows using

- `update_recipe()` and/or `update_model()`
- `remove_recipe()` and/or `remove_model()`

workflows



Other workflows functions

- `pull_workflow_preprocessor()`
  - returns either the formula or recipe used for preprocessing
- `pull_workflow_prepped_recipe()`
  - returns the prepped recipe
- `pull_workflow_spec()`
  - returns the `{parsnip}` model specification
- `pull_workflow_fit()`
  - returns the parsnip model fit

Back to our random forests

```r
# Workflows

rf_def_workflow <-
  workflow() %>%
  add_model(rf_def_mod) %>%
  add_recipe(rf_rec)

rf_tune_workflow <-
  workflow() %>%
  add_model(rf_tune_mod) %>%
  add_recipe(rf_rec)
```

```
# Fit Models (without workflows)

set.seed(3000)
rf_def_res <- fit_resamples(
  rf_def_mod,
  rf_rec,
  math_cv,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) x)
)
```

```
# Fit Models (with workflows)

set.seed(3000)
rf_def_res <- fit_resamples(
  rf_def_workflow,
  math_cv,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) x)
)
```

Fit Default Model

```
tictoc::tic()
set.seed(210)
rf_def_res <- fit_resamples(
  rf_def_workflow,
  math_cv,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) x)
)
tictoc::toc()
66.73 sec elapsed
```

`extract()`

- results in an additional column to be returned called `.extracts`
- `.extracts` is a list column that has tibbles containing the results of the user's function for each tuning parameter combination
  - `extract_model(x)` returns the model created during resampling
  - `extract_recipe(x)` returns the recipe created during resampling
  - `x` returns the workflow created during resampling

rf_def_res

```
#  10-fold cross-validation using stratification
# A tibble: 10 x 6
   splits             id     .metrics          .notes           .extracts        .predictions
   <list>             <chr>  <list>            <list>           <list>           <list>
 1 <split [5.1K/5~ Fold01 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 2 <split [5.1K/5~ Fold02 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 3 <split [5.1K/5~ Fold03 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 4 <split [5.1K/5~ Fold04 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 5 <split [5.1K/5~ Fold05 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 6 <split [5.1K/5~ Fold06 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 7 <split [5.1K/5~ Fold07 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [569 x ~
 8 <split [5.1K/5~ Fold08 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [568 x ~
 9 <split [5.1K/5~ Fold09 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [568 x ~
10 <split [5.1K/5~ Fold10 <tibble [2 x ~ <tibble [1 x~ <tibble [1 x ~ <tibble [566 x ~
```

```
rf_def_res$.extracts[[1]]
```

```
# A tibble: 1 x 1
  .extracts
  <list>
1 <workflow>
```

```
pluck(rf_def_res$.extracts[[1]], 1)
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: rand_forest()

-- Preprocessor ----------------------------------------------------------------
9 Recipe Steps

* step_mutate()
* step_mutate()
* step_rm()
* step_novel()
* step_unknown()
* step_medianimpute()
* step_nzv()
* step_dummy()
* step_nzv()

-- Model -----------------------------------------------------------------------
Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, num.threads = ~cores,      importance = ~"permutation", verbose = ~TRUE,
seed = sample.int(10^5,          1), probability = TRUE)

Type:                             Probability estimation
Number of trees:                  500
Sample size:                      5114
Number of independent variables:  19
Mtry:                             4
Target node size:                 10
Variable importance mode:         permutation
Splitrule:                        gini
OOB prediction error (Brier s.):  0.4523303
```

```
pluck(rf_def_res$.extracts[[1]], 1)
```

```
== Workflow ==========================================================================
Preprocessor: Recipe
Model: rand_forest()

-- Preprocessor ----------------------------------------------------------------------
9 Recipe Steps

* step_mutate()
* step_mutate()
* step_rm()
* step_novel()
* step_unknown()
* step_medianimpute()
* step_nzv()
* step_dummy()
* step_nzv()

-- Model -----------------------------------------------------------------------------
Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, num.threads = ~cores,      importance = ~"permutation", verbose = ~TRUE,
seed = sample.int(10^5,          1), probability = TRUE)

Type:                             Probability estimation
Number of trees:                  500
Sample size:                      5114
Number of independent variables:  19
Mtry:                             4
Target node size:                 10
Variable importance mode:         permutation
Splitrule:                        gini
OOB prediction error (Brier s.):  0.4523303
```

```
pluck(rf_def_res$.extracts[[1]], 1)
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: rand_forest()

-- Preprocessor ----------------------------------------------------------------
9 Recipe Steps

* step_mutate()
* step_mutate()
* step_rm()
* step_novel()
* step_unknown()
* step_medianimpute()
* step_nzv()
* step_dummy()
* step_nzv()

-- Model -----------------------------------------------------------------------
Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, num.threads = ~cores,      importance = ~"permutation", verbose = ~TRUE,
seed = sample.int(10^5,           1), probability = TRUE)

Type:                             Probability estimation
Number of trees:                  500
Sample size:                      5114
Number of independent variables:  19
Mtry:                             4
Target node size:                 10
Variable importance mode:         permutation
Splitrule:                        gini
OOB prediction error (Brier s.):  0.4523303
```

```r
rf_def_res %>%
  mutate(oob = map_dbl(.extracts,
                       ~pluck(.x$.extracts, 1)$fit$fit$fit$prediction.error)) %>%
  select(id, oob)
```

```
rf_def_res %>%
  mutate(oob = map_dbl(.extracts,
                       ~pluck(.x$.extracts, 1)$fit$fit$fit$prediction.error)) %>%
  select(id, oob)
```

```
# A tibble: 10 x 2
   id       oob
   <chr>  <dbl>
 1 Fold01 0.452
 2 Fold02 0.450
 3 Fold03 0.453
 4 Fold04 0.449
 5 Fold05 0.453
 6 Fold06 0.450
 7 Fold07 0.454
 8 Fold08 0.448
 9 Fold09 0.454
10 Fold10 0.449
```

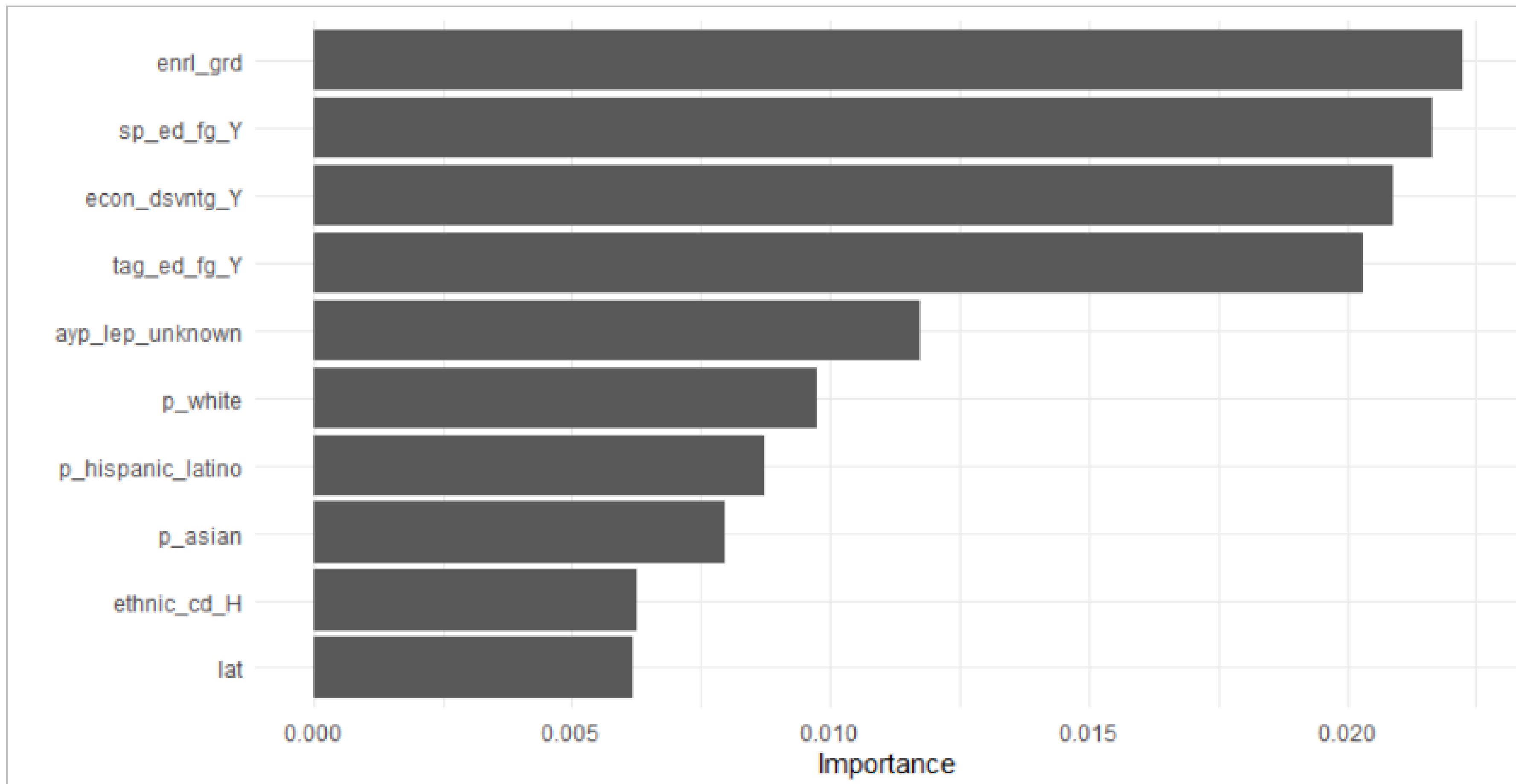
- Brier score: measures the accuracy of probabilistic predictions; the mean squared difference between the predicted **probability** assigned to the possible outcomes, and the actual outcome

```
rf_def_res %>%
  collect_metrics(summarize = FALSE)
```

```
# A tibble: 20 x 4
   id     .metric  .estimator .estimate
   <chr>  <chr>    <chr>          <dbl>
 1 Fold01 accuracy multiclass     0.432
 2 Fold01 roc_auc  hand_till      0.694
 3 Fold02 accuracy multiclass     0.423
 4 Fold02 roc_auc  hand_till      0.677
 5 Fold03 accuracy multiclass     0.456
 6 Fold03 roc_auc  hand_till      0.696
 7 Fold04 accuracy multiclass     0.430
 8 Fold04 roc_auc  hand_till      0.667
 9 Fold05 accuracy multiclass     0.460
10 Fold05 roc_auc  hand_till      0.703
11 Fold06 accuracy multiclass     0.423
12 Fold06 roc_auc  hand_till      0.682
13 Fold07 accuracy multiclass     0.467
14 Fold07 roc_auc  hand_till      0.712
15 Fold08 accuracy multiclass     0.411
16 Fold08 roc_auc  hand_till      0.658
17 Fold09 accuracy multiclass     0.475
18 Fold09 roc_auc  hand_till      0.704
19 Fold10 accuracy multiclass     0.431
20 Fold10 roc_auc  hand_till      0.664
```

```
pluck(rf_def_res$.extracts[[1]]$.extracts, 1) %>%
  pull_workflow_fit() %>%
  vip()
```

```
pluck(rf_def_res$.extracts[[1]]$.extracts, 1) %>%
  pull_workflow_fit() %>%
  vip()
```



```r
rf_def_res %>%
  mutate(vip = map(.extracts,
                   ~pluck(.x$.extracts, 1) %>%
                     pull_workflow_fit() %>%
                     vip())) %>%
  select(id, vip)
```

```
# A tibble: 10 x 2
   id     vip
   <chr>  <list>
 1 Fold01 <gg>
 2 Fold02 <gg>
 3 Fold03 <gg>
 4 Fold04 <gg>
 5 Fold05 <gg>
 6 Fold06 <gg>
 7 Fold07 <gg>
 8 Fold08 <gg>
 9 Fold09 <gg>
10 Fold10 <gg>
```

```
pluck(rf_def_res$.extracts[[1]]$.extracts, 1) %>%
  pull_workflow_preprocessor()
```

```
Data Recipe

Inputs:

      role #variables
        id          7
   outcome          1
 predictor         39

Operations:

Variable mutation for tst_dt
Variable mutation for classification
Delete terms contains("bnch")
Novel factor level assignment for all_nominal(), -all_outcomes()
Unknown factor level assignment for all_nominal(), -all_outcomes()
Median Imputation for all_numeric()
Sparse, unbalanced variable filter on all_predictors()
Dummy variables from all_nominal(), -has_role(match = "id"), -all_outcomes()
Sparse, unbalanced variable filter on all_predictors()
```

Fit Tuned Model (without workflows)

```
set.seed(3000)
rf_tune_res <- tune_grid(
  rf_tune_mod,
  rf_rec,
  math_cv,
  tune = 20,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) extract_model(x))
)
```

Fit Tuned Model (with workflows)

```
set.seed(3000)
rf_tune_res <- tune_grid(
  rf_tune_worfklow,
  math_cv,
  tune = 20,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) extract_model(x))
)
```

Fit Tuned Model (with workflows)

```
set.seed(3000)
rf_tune_res <- tune_grid(
  rf_tune_workfklow,
  math_cv,
  tune = 20,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) extract_model(x))
)
```

```
# Fit Tuned Model (with workflows)

set.seed(3000)
rf_tune_res <- tune_grid(
  rf_tune_worfklow,
  math_cv,
  tune = 20,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) extract_model(x))
)
```

```
# Fit Tuned Model (with workflows)

tictoc::tic()
set.seed(3000)
rf_tune_res <- tune_grid(
  rf_tune_worfklow,
  math_cv,
  tune = 20,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE,
                              extract = function(x) extract_model(x))
)
tictoc::toc()
892.26 sec elapsed
```

(about 15 mins)

compare to `66.73 sec elapsed` for the default settings (no tuning)

rf_tune_res

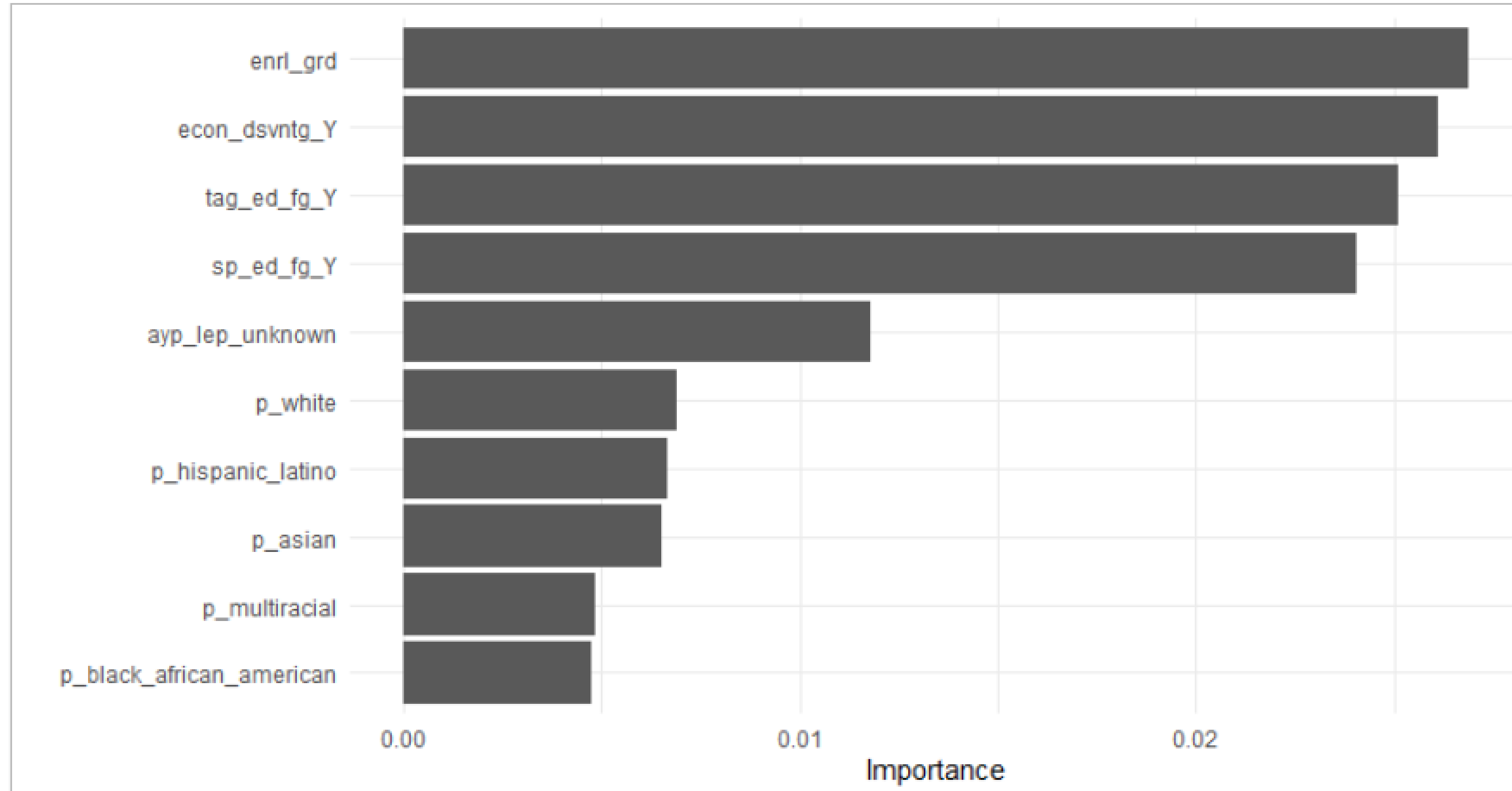
```
# Tuning results
# 10-fold cross-validation using stratification
# A tibble: 10 x 6
   splits             id     .metrics          .notes           .extracts         .predictions
   <list>             <chr>  <list>            <list>           <list>            <list>
 1 <split [5.1K/570]> Fold01 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,700 x 10]>
 2 <split [5.1K/570]> Fold02 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,700 x 10]>
 3 <split [5.1K/570]> Fold03 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,700 x 10]>
 4 <split [5.1K/570]> Fold04 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,700 x 10]>
 5 <split [5.1K/570]> Fold05 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,700 x 10]>
 6 <split [5.1K/568]> Fold06 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,680 x 10]>
 7 <split [5.1K/567]> Fold07 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,670 x 10]>
 8 <split [5.1K/567]> Fold08 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,670 x 10]>
 9 <split [5.1K/566]> Fold09 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,660 x 10]>
10 <split [5.1K/566]> Fold10 <tibble [20 x 6]> <tibble [0 x 1]> <tibble [10 x 4]> <tibble [5,660 x 10]>
```

```
rf_tune_res$.extracts[[1]]

# A tibble: 10 x 4
    mtry min_n .extracts .config
   <int> <int> <list>    <chr>
 1    10    30 <ranger>  Model01
 2    14    33 <ranger>  Model02
 3    17    24 <ranger>  Model03
 4     8     3 <ranger>  Model04
 5     4    40 <ranger>  Model05
 6     8    13 <ranger>  Model06
 7     1     6 <ranger>  Model07
 8    16    25 <ranger>  Model08
 9    13    17 <ranger>  Model09
10     5    20 <ranger>  Model10
```

rf_tune_res$.extracts[[1]]$.extracts[[1]]

```
Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~10L,      x), 
num.trees = ~1000, min.node.size = min_rows(~30L, x),      num.threads = ~cores, 
importance = ~"permutation", verbose = ~TRUE,      seed = sample.int(10^5, 1), 
probability = TRUE) 

Type:                             Probability estimation 
Number of trees:                  1000 
Sample size:                      5114 
Number of independent variables:  19 
Mtry:                             10 
Target node size:                 30 
Variable importance mode:         permutation 
Splitrule:                        gini 
OOB prediction error (Brier s.):  0.4436529 
```

```
rf_tune_res$.extracts[[1]]$.extracts[[1]] %>%
  vip()
```



```r
rf_tune_res$.extracts[[1]]$.extracts[[1]] %>%
  pull_workflow_preprocessor()

Error: `x` must be a workflow, not a ranger.
Run `rlang::last_error()` to see where the error occurred.
```

```r
rf_def_res %>%
  collect_metrics()
```

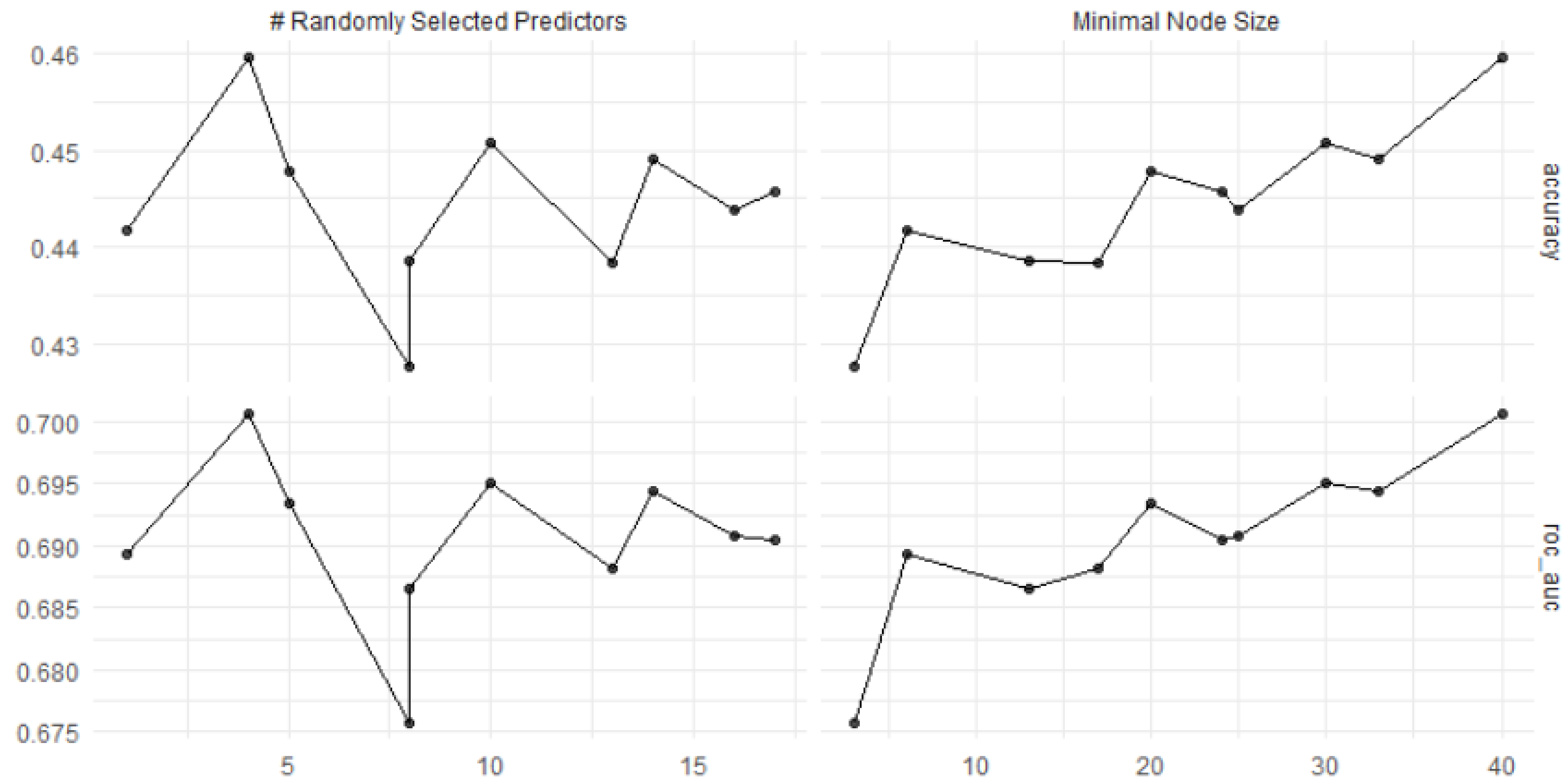
```
# A tibble: 2 x 5
  .metric  .estimator  mean     n std_err
  <chr>    <chr>      <dbl> <int>   <dbl>
1 accuracy multiclass 0.441    10 0.00694
2 roc_auc  hand_till  0.686    10 0.00596
```

```r
rf_tune_res %>%
  collect_metrics() %>%
  arrange(.metric, desc(mean)) %>%
  group_by(.metric) %>%
  slice(1:5)
```

```
# A tibble: 10 x 8
# Groups:   .metric [2]
    mtry min_n .metric  .estimator  mean     n std_err .config
   <int> <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
 1     4    40 accuracy multiclass 0.460    10 0.00669 Model05
 2    10    30 accuracy multiclass 0.451    10 0.00701 Model01
 3    14    33 accuracy multiclass 0.449    10 0.00739 Model02
 4     5    20 accuracy multiclass 0.448    10 0.00673 Model10
 5    17    24 accuracy multiclass 0.446    10 0.00565 Model03
 6     4    40 roc_auc  hand_till  0.701    10 0.00636 Model05
 7    10    30 roc_auc  hand_till  0.695    10 0.00618 Model01
 8    14    33 roc_auc  hand_till  0.694    10 0.00609 Model02
 9     5    20 roc_auc  hand_till  0.693    10 0.00629 Model10
10    16    25 roc_auc  hand_till  0.691    10 0.00606 Model08
```

- We improved our predictions (based on two metrics)
- But it took about 15 times as long
- Worth it?

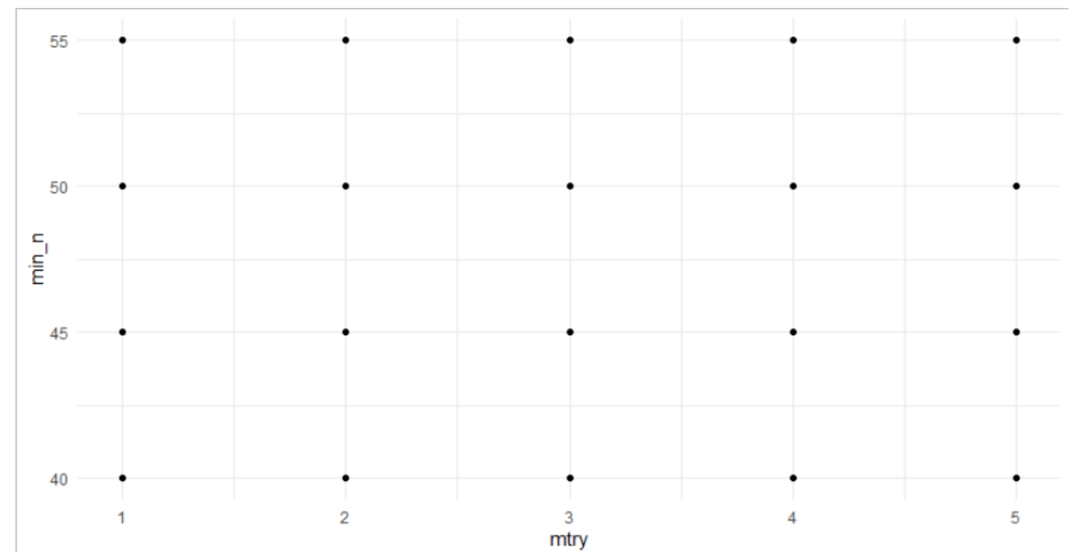
```
rf_tune_res %>%
  autoplot() +
  geom_line()
```



- Could probably increase `min_n` beyond 35
- Maybe lower values of `mtry` increased performance (between 0 and 5)?

Create grid and tune model

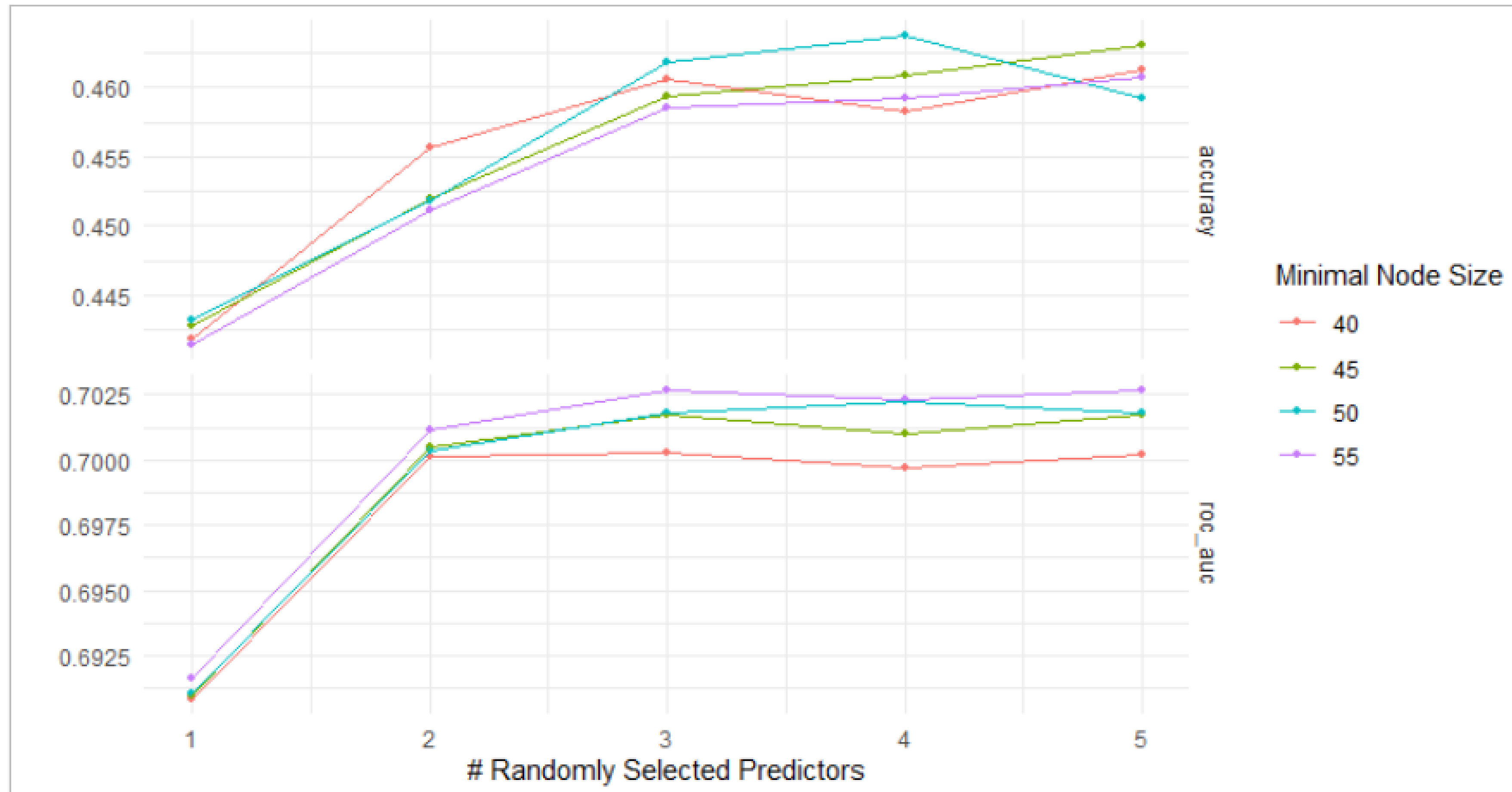
```
rf_grid_reg <- grid_regular(
  mtry(range = c(1, 15)),
  min_n(range = c(30, 50)),
  levels = c(5, 5)
)
```




```
tic()
set.seed(3000)
rf_grid_res <- tune_grid(
  rf_tune_workflow,
  math_cv,
  grid = rf_grid_reg,
  control = control_resamples(verbose = TRUE,
                              save_pred = TRUE)
)
toc()
1607.22 sec elapsed
```

(about 27 min)

```
rf_grid_res %>%
  autoplot() +
  geom_line()
```



```r
show_best(rf_tune_res, metric = "accuracy", n = 10) %>%
  bind_rows(show_best(rf_tune_res, metric = "roc_auc", n = 10))
%>%
  group_by(.metric) %>%
  slice(1:5)
```

```
# A tibble: 10 x 8
# Groups:   .metric [2]
    mtry min_n .metric  .estimator  mean     n std_err .config
   <int> <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>  
 1     4    40 accuracy multiclass 0.460    10 0.00669 Model05
 2    10    30 accuracy multiclass 0.451    10 0.00701 Model01
 3    14    33 accuracy multiclass 0.449    10 0.00739 Model02
 4     5    20 accuracy multiclass 0.448    10 0.00673 Model10
 5    17    24 accuracy multiclass 0.446    10 0.00565 Model03
 6     4    40 roc_auc  hand_till  0.701    10 0.00636 Model05
 7    10    30 roc_auc  hand_till  0.695    10 0.00618 Model01
 8    14    33 roc_auc  hand_till  0.694    10 0.00609 Model02
 9     5    20 roc_auc  hand_till  0.693    10 0.00629 Model10
10    16    25 roc_auc  hand_till  0.691    10 0.00606 Model08
```

```r
show_best(rf_grid_res, metric = "accuracy", n = 10) %>%
  bind_rows(show_best(rf_grid_res, metric = "roc_auc", n = 10))
%>%
  group_by(.metric) %>%
  slice(1:5)
```

```
# A tibble: 10 x 8
# Groups:   .metric [2]
    mtry min_n .metric  .estimator  mean     n std_err .config
   <int> <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>  
 1     4    50 accuracy multiclass 0.464    10 0.00663 Model14
 2     5    45 accuracy multiclass 0.463    10 0.00652 Model10
 3     3    50 accuracy multiclass 0.462    10 0.00668 Model13
 4     5    40 accuracy multiclass 0.461    10 0.00738 Model05
 5     4    45 accuracy multiclass 0.461    10 0.00663 Model09
 6     3    55 roc_auc  hand_till  0.703    10 0.00615 Model18
 7     5    55 roc_auc  hand_till  0.703    10 0.00641 Model20
 8     4    55 roc_auc  hand_till  0.702    10 0.00637 Model19
 9     4    50 roc_auc  hand_till  0.702    10 0.00626 Model14
10     3    50 roc_auc  hand_till  0.702    10 0.00634 Model13
```

- Worth it?

```
select_best(rf_tune_res, metric = "roc_auc")

# A tibble: 1 x 3
   mtry min_n .config
  <int> <int> <chr>
1     4    40 Model05
```

```
select_best(rf_grid_res, metric = "roc_auc")

# A tibble: 1 x 3
   mtry min_n .config
  <int> <int> <chr>
1     3    55 Model18
```

```
rf_best <- select_best(rf_grid_res, metric = "roc_auc")
```

```r
rf_best <- select_best(rf_grid_res, metric = "roc_auc")

rf_wf_final <- finalize_workflow(
  rf_tune_workflow,
  rf_best
)
```

```r
rf_best <- select_best(rf_grid_res, metric = "roc_auc")

rf_wf_final <- finalize_workflow(
  rf_tune_workflow,
  rf_best
)

rf_wf_final
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: rand_forest()

-- Preprocessor ----------------------------------------------------------------
9 Recipe Steps

* step_mutate()
* step_mutate()
* step_rm()
* step_novel()
* step_unknown()
* step_medianimpute()
* step_nzv()
* step_dummy()
* step_nzv()

-- Model -----------------------------------------------------------------------
Random Forest Model Specification (classification)

Main Arguments:
  mtry = 3
  trees = 1000
  min_n = 55

Engine-Specific Arguments:
  num.threads = cores
  importance = permutation
  verbose = TRUE

Computational engine: ranger
```

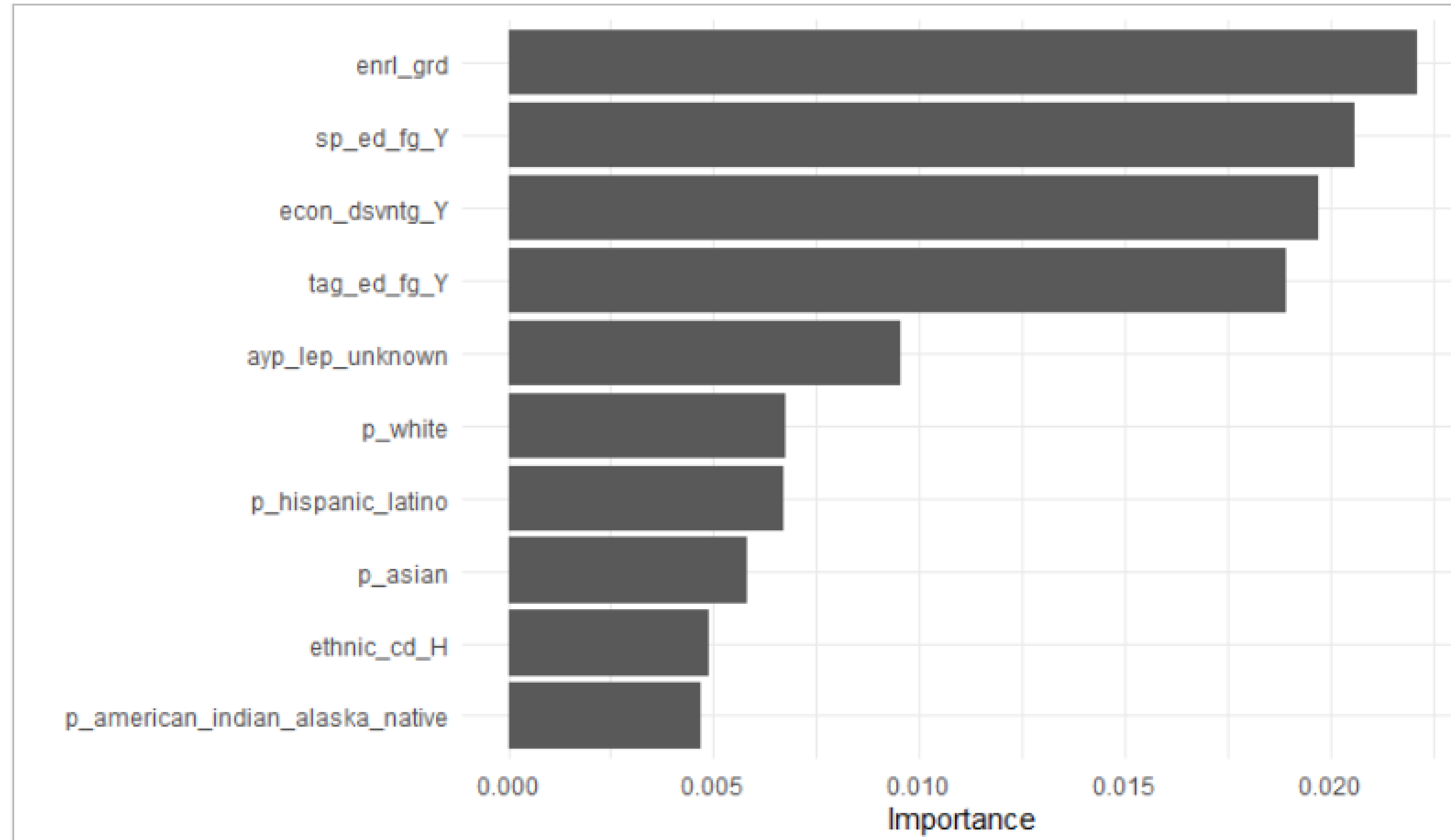
```
tictoc::tic()
set.seed(3000)
rf_res_final <- last_fit(rf_wf_final,
                         split = math_split)
tictoc::toc()
```

```
# Resampling results
# Monte Carlo cross-validation (0.75/0.25) with 1 resamples
# A tibble: 1 x 6
  splits              id               .metrics         .notes           .predictions        .workflow
  <list>              <chr>            <list>           <list>           <list>              <list>
1 <split [5.7K/1.9K]> train/test split <tibble [2 x 3]> <tibble [0 x 1]> <tibble [1,893 x 7]> <workflow>
```

```
rf_res_final %>%
  pluck(".workflow", 1)
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: rand_forest()

-- Preprocessor ------------------------------------------------------------------------
9 Recipe Steps

* step_mutate()
* step_mutate()
* step_rm()
* step_novel()
* step_unknown()
* step_medianimpute()
* step_nzv()
* step_dummy()
* step_nzv()

-- Model -----------------------------------------------------------------------------
Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~3L,      x), num.trees = ~1000,
min.node.size = min_rows(~55L, x),      num.threads = ~cores, importance = ~"permutation", verbose = ~TRUE,
seed = sample.int(10^5, 1), probability = TRUE)

Type:                             Probability estimation
Number of trees:                  1000
Sample size:                      5684
Number of independent variables:  19
Mtry:                             3
Target node size:                 55
Variable importance mode:         permutation
Splitrule:                        gini
OOB prediction error (Brier s.):  0.4487589
```

```r
rf_res_final %>%
  pluck(".workflow", 1) %>%
  pull_workflow_fit() %>%
  vip()
```



```r
# Default Model

# mtry = floor(sqrt(p)) = floor(sqrt(39)) = 6
# trees = 500 (num.trees)
# min_n = 1 (min.node.size)

(cores <- parallel::detectCores())
8

rf_def_mod <-
  rand_forest() %>%
  set_engine("ranger",
             num.threads = cores, #argument from {ranger}
             importance = "permutation", #argument from {ranger}
             verbose = TRUE) %>% #argument from {ranger}
  set_mode("classification")
```

ranger::ranger

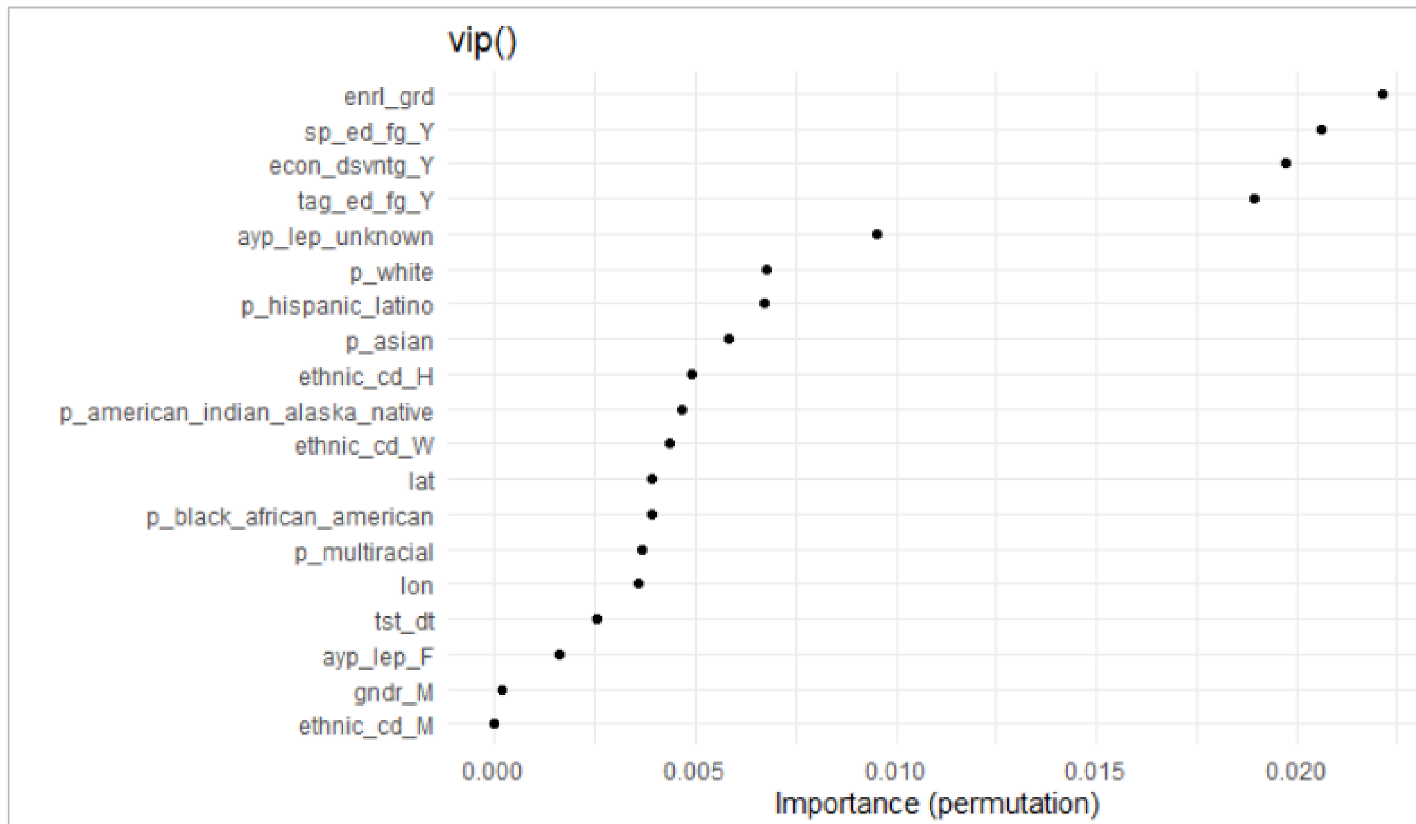
Usage

```
ranger(
  formula = NULL,
  data = NULL,
  num.trees = 500,
  mtry = NULL,
  importance = "none",
  write.forest = TRUE,
  probability = FALSE,
  min.node.size = NULL,
  max.depth = NULL,
  replace = TRUE,
  sample.fraction = ifelse(replace, 1, 0.632),
  case.weights = NULL,
  class.weights = NULL,
  splitrule = NULL,
  num.random.splits = 1,
  alpha = 0.5,
  minprop = 0.1,
  split.select.weights = NULL,
  always.split.variables = NULL,
```

`importance` Variable importance mode, one of 'none', 'impurity', 'impurity_corrected', 'permutation'. The 'impurity' measure is the Gini index for classification, the variance of the responses for regression and the sum of test statistics (see `splitrule`) for survival.

- `impurity`: the probability of a variable being wrongly classified when it is randomly chosen (Gini index)
  - if all elements belong to a single class, then is “pure”; Gini index = 0
  - elements randomly distributed across classes; Gini index = 1
  - lowest Gini is selected for the root
- impurity-based feature importance can inflate the importance of numerical features
  - each time a break point is selected in a variable, every level of the variable is tested to find the best break point
  - continuous variables will have many more split points, which results in a higher probability that by chance that variable happens to predict the outcome well, since variables where more splits are tried will appear more often in the tree
- `permutation`: calculate the increase in the model’s prediction error after permuting the feature
  - a feature is “important” if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction
  - a feature is “unimportant” if shuffling its values leaves the model error unchanged, because in this case the model ignored the feature for the prediction
- permutation-based feature importance is more reliable than impurity, but:
  - more computationally expensive
  - potentially biased toward collinear predictive variables

```
rf_res_final %>%
  pluck(".workflow", 1) %>%
  pull_workflow_fit() %>%
  vip(geom = "point",
      num_features = 20) +
  labs(y = "Importance (permutation)",
      title = "vip()")
```



mtry

- Suggestion (Boehmke): start with five evenly spaced values of `mtry` across the range 2 to *p* centered at the recommended default

```
p <- 20 #length(setdiff(names(data_train), “outcome"))
grid_max_entropy(mtry(range = c(2, p)), size = 5)
```

```
# A tibble: 5 x 1
   mtry
  <int>
1     2
2     6
3    19
4    10
5    15
```