INFO 523 Exercise

Weeks 8-9: Classification and resampling in R

Table of contents

Goal **1**
Submission 2

Overview **2**

Precipitation temporal repacking data **2**
Examine the data 3

Modeling **6**

Estimating Performance **7**
What happened here? 9

Resampling to the Rescue **9**
Fit a Model with Resampling 11

[ADVANCED] **14**

Session Information **15**

Goal

Practice basic R commands/methods for estimating prediction performance of machine learning algorithms.

Note: copying and pasting early in learning will not produce the results you are looking for, and will catch up to you eventually.

Submission

Please submit `.r`, `.rmd`, or `.qmd` files ONLY.

Overview

We will be using the `tidymodels` framework for this exercise instead of what is in the textbook. The purpose of `tidymodels` is to give `tidyverse` users a familiar framework, and to standardize similar model outputs.

For more on `tidymodels`, visit this page: https://www.tidymodels.org/

```
library(pacman)

p_load(dlookr, # for imputing NAs
       randomForest,
       ranger,
       RCurl,
       tidymodels,
       tidyverse)
```

Precipitation temporal repacking data

Let's use the data from [@zhang2021] which is from a study that investigates the effects of larger, fewer precipitation events (due to climate change) on a semi-arid bunchgrass ecosystem. The data examine water flux in bunchgrass across four field precipitation treatments `Treatment` (`S1-S4`) and five community blocks `block` (`H1-H5`).

`Treatment`:

- `S1`: 3.5 days dry interval
- `S2`: 7 days dry interval
- `S3`: 14 days dry interval
- `S4`: 21 days dry interval

Note that the data used are a combination of the `Plot_flux` and `reproductive` sheets found in the original `FE-2021-00845-sup-Data.xlsx`. This was done by joining the two frames and by using MICE missing value imputation from the `dlookr` package to fill in the one missing value; classification algorithms cannot classify with `NA`s. The final data can be reproduced as below:

```
set.seed(123)

FE_2021_Classification_mod <-
  Plot_flux |>
  left_join(Reproductive) |>
  rename(NEE = `NEE( mol m-2 s-1)`,
         ER = `ER( mol m-2 s-1)`,
         Repro_culm = `Reproductive culm(plant-1)`) |>
  mutate(Treatment = factor(Treatment),
         block = factor(block),
         Repro_culm = as.numeric(Repro_culm))

FE_2021_Classification_mod <-
  FE_2021_Classification_mod |>
  mutate(Repro_culm = imputate_na(FE_2021_Classification_mod, Repro_culm, method = "mice")
```

```
# Read in csv from the web
data <- getURL("https://raw.githubusercontent.com/ua-data7/classical-machine-learning-work

data <- read.csv(text = data) |>
    mutate(Treatment = factor(Treatment),
         block = factor(block),
         Repro_culm = as.numeric(Repro_culm))
```

Examine the data

```
data |>
  head()
```

```
  block Treatment       Date      NEE       ER Repro_culm
1    H1        S2 2020-07-15 1.746272 2.529016   1.750000
2    H1        S1 2020-07-15 2.291839 3.136368   1.562500
3    H1        S3 2020-07-15 2.128131 2.705697   6.259259
4    H1        S4 2020-07-15 1.914331 3.095150   7.160000
```

```
5    H2        S2 2020-07-15 2.452358 2.695888   2.600000
6    H2        S3 2020-07-15 2.724574 4.090446   2.000000
```

For each of the precipitation treatments (`Treatment`) and community block (`block`), we know their:

- `NEE`: Net ecosystem exchange, measured as whole-plot CO_2 flux(F_c, mol m^-2 s^-1).
- `ER`: Ecosystem respiration, measured as CO_2 exchange (mol m^-2 s^-1)
- `Repro_culm`: Number of reproductive inflorescence (flowers) (plant$^-1$)

In our previous *Preprocess your data with recipes* article, we started by splitting our data. It is common when beginning a modeling project to separate the data set into two partitions:

- The *training set* is used to estimate parameters, compare models and feature engineering techniques, tune models, etc.
- The *test set* is held in reserve until the end of the project, at which point there should only be one or two models under serious consideration. It is used as an unbiased source for measuring final model performance.

There are different ways to create these partitions of the data. The most common approach is to use a random sample. Suppose that one quarter of the data were reserved for the test set. Random sampling would randomly select 25% for the test set and use the remainder for the training set. We can use the rsample package for this purpose.

Since random sampling uses random numbers, it is important to set the random number seed. This ensures that the random numbers can be reproduced at a later time (if needed).

The function `rsample::initial_split()` takes the original data and saves the information on how to make the partitions.

```
set.seed(333)

FE_split <- initial_split(data, strata = Treatment)
```

Here we used the `strata` argument, which conducts a stratified split. This ensures that, despite the imbalance we noticed in our `class` variable, our training and test data sets will keep roughly the same proportions of poorly and well-segmented cells as in the original data. After the `initial_split`, the `training()` and `testing()` functions return the actual data sets.

```r
FE_train <- training(FE_split)
FE_test  <- testing(FE_split)

# number of rows in training dataset
nrow(FE_train)
```

```
[1] 180
```

```r
# proportion of rows in training dataset over fill dataset
nrow(FE_train)/nrow(data)
```

```
[1] 0.75
```

```r
# training set proportions by class
FE_train |>
  count(Treatment) |>
  mutate(prop = n/sum(n))
```

```
  Treatment  n prop
1        S1 45 0.25
2        S2 45 0.25
3        S3 45 0.25
4        S4 45 0.25
```

```r
# test set proportions by class
FE_test |>
  count(Treatment) |>
  mutate(prop = n/sum(n))
```

```
  Treatment  n prop
1        S1 15 0.25
2        S2 15 0.25
3        S3 15 0.25
4        S4 15 0.25
```

The majority of the modeling work is then conducted on the training set data.

Modeling

Random forest models are ensembles of decision trees. A large number of decision tree models are created for the ensemble based on slightly different versions of the training set. When creating the individual decision trees, the fitting process encourages them to be as diverse as possible. The collection of trees are combined into the random forest model and, when a new sample is predicted, the votes from each tree are used to calculate the final predicted value for the new sample. For categorical outcome variables like `Treatment` in our `data` data example, the majority vote across all the trees in the random forest determines the predicted treatment for the new sample.

One of the benefits of a random forest model is that it is very low maintenance; it requires very little preprocessing of the data and the default parameters tend to give reasonable results. For that reason, we won't create a recipe for the `data` data.

At the same time, the number of trees in the ensemble should be large (in the thousands) and this makes the model moderately expensive to compute.

To fit a random forest model on the training set, let's use the parsnip package with the ranger engine. We first define the model that we want to create:

```
rf_mod <-
  rand_forest(trees = 1000) |>
  set_engine("ranger") |>
  set_mode("classification")
```

Starting with this parsnip model object, the `fit()` function can be used with a model formula. Since random forest models use random numbers, we again set the seed prior to computing:

```
set.seed(234)
rf_fit <-
  rf_mod |>
  fit(Treatment ~ ., data = FE_train)
rf_fit
```

```
parsnip model object

Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, num.trees = ~1000,      num.threads = 1, verb

Type:                             Probability estimation
```

```
Number of trees:                  1000
Sample size:                      180
Number of independent variables:  5
Mtry:                             2
Target node size:                 10
Variable importance mode:         none
Splitrule:                        gini
OOB prediction error (Brier s.):  0.2267824
```

This new `rf_fit` object is our fitted model, trained on our training data set.

Estimating Performance

During a modeling project, we might create a variety of different models. To choose between them, we need to consider how well these models do, as measured by some performance statistics. In our example in this article, some options we could use are:

- the area under the Receiver Operating Characteristic (ROC) curve, and
- overall classification accuracy.

The ROC curve uses the class probability estimates to give us a sense of performance across the entire set of potential probability cutoffs. Overall accuracy uses the hard class predictions to measure performance. The hard class predictions tell us whether our model predicted class (`Treatment`: `S1`-`S4`) for each sample. But, behind those predictions, the model is actually estimating a probability. A simple 50% probability cutoff is used to categorize a sample as poorly classified.

The yardstick package has functions for computing both of these measures called `roc_auc()` and `accuracy()`.

At first glance, it might seem like a good idea to use the training set data to compute these statistics. (This is actually a very bad idea.) Let's see what happens if we try this. To evaluate performance based on the training set, we call the `predict()` method to get both types of predictions (i.e. probabilities and hard class predictions).

```
rf_training_pred <-
  predict(rf_fit, FE_train) |>
  bind_cols(predict(rf_fit, FE_train, type = "prob")) |>
  # Add the true outcome data back in
  bind_cols(FE_train |>
```

```
              select(Treatment))
```

Using the yardstick functions, this model has spectacular results, so spectacular that you might be starting to get suspicious:

```
rf_training_pred |>                # training set predictions
  roc_auc(truth = Treatment, c(.pred_S1, .pred_S2, .pred_S3, .pred_S4))
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc hand_till      0.999
```

```
rf_training_pred |>                # training set predictions
  accuracy(truth = Treatment, .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy multiclass     0.983
```

Now that we have this model with exceptional performance, we proceed to the test set. Unfortunately, we discover that, although our results aren't bad, they are certainly worse than what we initially thought based on predicting the training set:

```
rf_testing_pred <-
  predict(rf_fit, FE_test) |>
  bind_cols(predict(rf_fit, FE_test, type = "prob")) |>
  # Add the true outcome data back in
  bind_cols(FE_test |>
              select(Treatment))
```

```
rf_testing_pred |>                # test set predictions
  roc_auc(truth = Treatment, c(.pred_S1, .pred_S2, .pred_S3, .pred_S4))
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc hand_till      0.976
```

```
rf_testing_pred |>                # test set predictions
  accuracy(truth = Treatment, .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy multiclass     0.883
```

What happened here?

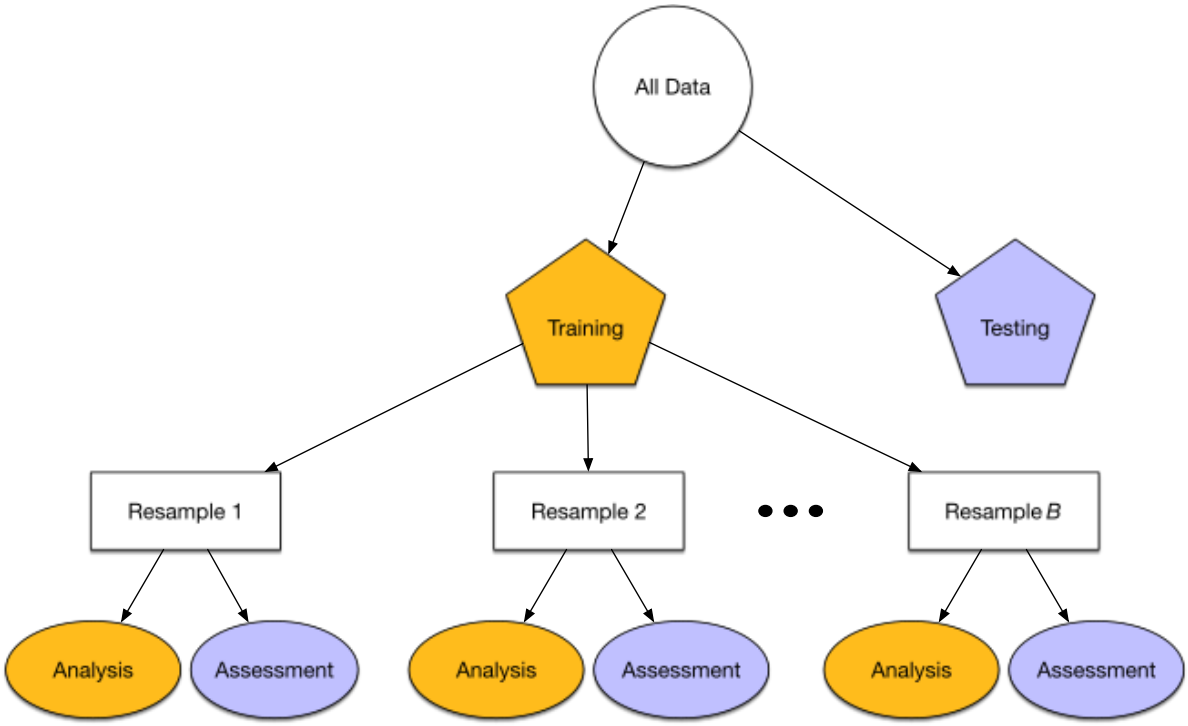
There are several reasons why training set statistics like the ones shown in this section can be unrealistically optimistic:

- Models like random forests, neural networks, and other black-box methods can essentially memorize the training set. Re-predicting that same set should always result in nearly perfect results.
- The training set does not have the capacity to be a good arbiter of performance. It is not an independent piece of information; predicting the training set can only reflect what the model already knows.

To understand that second point better, think about an analogy from teaching. Suppose you give a class a test, then give them the answers, then provide the same test. The student scores on the *second* test do not accurately reflect what they know about the subject; these scores would probably be higher than their results on the first test.

Resampling to the Rescue

Resampling methods, such as cross-validation and the bootstrap, are empirical simulation systems. They create a series of data sets similar to the training/testing split discussed previously; a subset of the data are used for creating the model and a different subset is used to measure performance. Resampling is always used with the *training set*. This schematic from Kuhn and Johnson (2019) illustrates data usage for resampling methods:



In the first level of this diagram, you see what happens when you use `rsample::initial_split()`, which splits the original data into training and test sets. Then, the training set is chosen for resampling, and the test set is held out.

Let's use 10-fold cross-validation (CV) in this example. This method randomly allocates the 1514 cells in the training set to 10 groups of roughly equal size, called "folds". For the first iteration of resampling, the first fold of about 151 cells are held out for the purpose of measuring performance. This is similar to a test set but, to avoid confusion, we call these data the *assessment set* in the tidymodels framework.

The other 90% of the data (about 1362 cells) are used to fit the model. Again, this sounds similar to a training set, so in tidymodels we call this data the *analysis set*. This model, trained on the analysis set, is applied to the assessment set to generate predictions, and performance statistics are computed based on those predictions.

In this example, 10-fold CV moves iteratively through the folds and leaves a different 10% out each time for model assessment. At the end of this process, there are 10 sets of performance statistics that were created on 10 data sets that were not used in the modeling process. For the cell example, this means 10 accuracies and 10 areas under the ROC curve. While 10 models were created, these are not used further; we do not keep the models themselves trained on these folds because their only purpose is calculating performance metrics.

The final resampling estimates for the model are the **averages** of the performance statistics replicates. For example, suppose for our data the results were:

Resample	Accuracy	ROC_AUC	Assessment Size
Fold01	0.889	0.978	18
Fold02	0.722	0.924	18
Fold03	0.889	0.975	18
Fold04	0.778	1	18
Fold05	0.944	0.995	18
Fold06	0.833	0.923	18
Fold07	0.889	0.977	18
Fold08	0.778	0.940	18
Fold09	0.667	0.971	18
Fold10	0.833	0.956	18

From these resampling statistics, the final estimate of performance for this random forest model would be 0.964 for the area under the ROC curve and 0.822 for accuracy.

These resampling statistics are an effective method for measuring model performance *without* predicting the training set directly as a whole.

Fit a Model with Resampling

To generate these results, the first step is to create a resampling object using `rsample`. There are several resampling methods implemented in rsample; cross-validation folds can be created using `vfold_cv()`:

```
set.seed(345)

folds <- vfold_cv(FE_train, v = 10)
folds$splits
```

```
[[1]]
<Analysis/Assess/Total>
<162/18/180>

[[2]]
<Analysis/Assess/Total>
<162/18/180>

[[3]]
```

```
<Analysis/Assess/Total>
<162/18/180>

[[4]]
<Analysis/Assess/Total>
<162/18/180>

[[5]]
<Analysis/Assess/Total>
<162/18/180>

[[6]]
<Analysis/Assess/Total>
<162/18/180>

[[7]]
<Analysis/Assess/Total>
<162/18/180>

[[8]]
<Analysis/Assess/Total>
<162/18/180>

[[9]]
<Analysis/Assess/Total>
<162/18/180>

[[10]]
<Analysis/Assess/Total>
<162/18/180>
```

The list column for `splits` contains the information on which rows belong in the analysis and assessment sets. There are functions that can be used to extract the individual resampled data called `analysis()` and `assessment()`.

However, the tune package contains high-level functions that can do the required computations to resample a model for the purpose of measuring performance. You have several options for building an object for resampling:

- Resample a model specification preprocessed with a formula or recipe, or
- Resample a `workflow()` that bundles together a model specification and formula/recipe.

For this example, let's use a `workflow()` that bundles together the random forest model and a formula, since we are not using a recipe. Whichever of these options you use, the syntax to `fit_resamples()` is very similar to `fit()`:

```
rf_wf <-
  workflow() |>
  add_model(rf_mod) |>
  add_formula(Treatment ~ .)

set.seed(456)
rf_fit_rs <-
  rf_wf |>
  fit_resamples(folds)
```

```
rf_fit_rs
```

```
# Resampling results
# 10-fold cross-validation
# A tibble: 10 x 4
   splits           id     .metrics         .notes
   <list>           <chr>  <list>           <list>
 1 <split [162/18]> Fold01 <tibble [2 x 4]> <tibble [0 x 3]>
 2 <split [162/18]> Fold02 <tibble [2 x 4]> <tibble [0 x 3]>
 3 <split [162/18]> Fold03 <tibble [2 x 4]> <tibble [0 x 3]>
 4 <split [162/18]> Fold04 <tibble [2 x 4]> <tibble [0 x 3]>
 5 <split [162/18]> Fold05 <tibble [2 x 4]> <tibble [0 x 3]>
 6 <split [162/18]> Fold06 <tibble [2 x 4]> <tibble [0 x 3]>
 7 <split [162/18]> Fold07 <tibble [2 x 4]> <tibble [0 x 3]>
 8 <split [162/18]> Fold08 <tibble [2 x 4]> <tibble [0 x 3]>
 9 <split [162/18]> Fold09 <tibble [2 x 4]> <tibble [0 x 3]>
10 <split [162/18]> Fold10 <tibble [2 x 4]> <tibble [0 x 3]>
```

The results are similar to the `folds` results with some extra columns. The column `.metrics` contains the performance statistics created from the 10 assessment sets. These can be manually unnested but the tune package contains a number of simple functions that can extract these data:

```
collect_metrics(rf_fit_rs)
```

```
# A tibble: 2 x 6
  .metric  .estimator  mean     n std_err .config
```

```
  <chr>    <chr>      <dbl> <int>   <dbl> <chr>
1 accuracy multiclass 0.833    10  0.0287 Preprocessor1_Model1
2 roc_auc  hand_till  0.943    10  0.0146 Preprocessor1_Model1
```

Think about these values we now have for accuracy and AUC. These performance metrics are now more realistic (i.e. lower) than our ill-advised first attempt at computing performance metrics in the section above. If we wanted to try different model types for this data set, we could more confidently compare performance metrics computed using resampling to choose between models. Also, remember that at the end of our project, we return to our test set to estimate final model performance. We have looked at this once already before we started using resampling, but let's remind ourselves of the results:

```
rf_testing_pred |>                # test set predictions
  roc_auc(truth = Treatment, c(.pred_S1, .pred_S2, .pred_S3, .pred_S4))
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc hand_till      0.976
```

```
rf_testing_pred |>                # test set predictions
  accuracy(truth = Treatment, .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy multiclass     0.883
```

[ADVANCED]

I have found that performanceEstimation does not work on all datasets and with all mining methods. If you could use it on your dataset, that would be great and try it. Otherwise, use other methods we studied to tune your mining methods.

Also note that there are several other resampling methods within `tidymodels`. This example serves to give you the most widely used methods to estimate performance and correct overfitting through resampling. The textbook also gives a different approach, which I have attached as supplementary for the unit.

Session Information

```
- Session info ---------------------------------------------------------------
 setting  value
 version  R version 4.3.1 (2023-06-16)
 os       macOS Ventura 13.3.1
 system   aarch64, darwin20
 ui       X11
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       America/Phoenix
 date     2023-08-03
 pandoc   3.1.1 @ /Applications/RStudio.app/Contents/Resources/app/quarto/bin/tools/ (via rma

- Packages -------------------------------------------------------------------
 package      * version   date (UTC) lib source
 broom        * 1.0.5     2023-06-09 [1] CRAN (R 4.3.0)
 dials        * 1.2.0     2023-04-03 [1] CRAN (R 4.3.0)
 dlookr       * 0.6.2     2023-07-01 [1] CRAN (R 4.3.0)
 dplyr        * 1.1.2     2023-04-20 [1] CRAN (R 4.3.0)
 forcats      * 1.0.0     2023-01-29 [1] CRAN (R 4.3.0)
 ggplot2      * 3.4.2     2023-04-03 [1] CRAN (R 4.3.0)
 infer        * 1.0.4     2022-12-02 [1] CRAN (R 4.3.0)
 lubridate    * 1.9.2     2023-02-10 [1] CRAN (R 4.3.0)
 modeldata    * 1.1.0     2023-01-25 [1] CRAN (R 4.3.0)
 pacman       * 0.5.1     2019-03-11 [1] CRAN (R 4.3.0)
 parsnip      * 1.1.0     2023-04-12 [1] CRAN (R 4.3.0)
 purrr        * 1.0.1     2023-01-10 [1] CRAN (R 4.3.0)
 randomForest * 4.7-1.1   2022-05-23 [1] CRAN (R 4.3.0)
 ranger       * 0.15.1    2023-04-03 [1] CRAN (R 4.3.0)
 RCurl        * 1.98-1.12 2023-03-27 [1] CRAN (R 4.3.0)
 readr        * 2.1.4     2023-02-10 [1] CRAN (R 4.3.0)
 recipes      * 1.0.6     2023-04-25 [1] CRAN (R 4.3.0)
 rsample      * 1.1.1     2022-12-07 [1] CRAN (R 4.3.0)
 scales       * 1.2.1     2022-08-20 [1] CRAN (R 4.3.0)
 stringr      * 1.5.0     2022-12-02 [1] CRAN (R 4.3.0)
 tibble       * 3.2.1     2023-03-20 [1] CRAN (R 4.3.0)
 tidymodels   * 1.1.0     2023-05-01 [1] CRAN (R 4.3.0)
```

```
 tidyr        * 1.3.0     2023-01-24 [1] CRAN (R 4.3.0)
 tidyverse    * 2.0.0     2023-02-22 [1] CRAN (R 4.3.0)
 tune         * 1.1.1     2023-04-11 [1] CRAN (R 4.3.0)
 workflows    * 1.1.3     2023-02-22 [1] CRAN (R 4.3.0)
 workflowsets * 1.0.1     2023-04-06 [1] CRAN (R 4.3.0)
 yardstick    * 1.2.0     2023-04-21 [1] CRAN (R 4.3.0)

 [1] /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/library
```
