2201-353-353m-453-PCA-PCR-PLS-FA-TidyModeling Class 07a-p

Profs: R. H. French, L. S. Bruckman, P. Leu, K. Davis, S. Cirlos

TAs: W. Oltjen, K. Hernandez, M. Li, M. Li, D. Colvin

07 March, 2023

Contents

7.1.2.1 Class Readings, Assignments, Textbooks Syllabus Topics

7.1.2.1.1 Reading, Lab Exercises, SemProjects

- Readings:
  - For today: ISLR8, DL08,09
  - For next class: ISLR9, (R4DS26-30)
- Laboratory Exercises:
  - LE4 given out
  - LE4 due March 9th
- Office Hours: (Class Canvas Calendar for Zoom Link)
  - Wednesdays @ 4:00 PM to 5:00 PM
  - Saturdays @ 3:00 PM to 4:00 PM
  - **Office Hours are on Zoom, and recorded**
- Semester Projects
  - Office Hours for SemProjs: Mondays at 4pm on Zoom

- DSCI 453 Students Biweekly Updates Due
  - Update #4 is Due **Friday March 10th**
- DSCI 453 Students
  - Next Report Out # is Due ** **
- All DSCI 353/353M/453, E1453/2453 Students:
  - **Peer Grading of Report Out #1 is Due Thursday March 2nd**
- Exams
  - MidTerm: **Thursday March 9th**, in class or remote, 11:30 - 12:45 PM
    - **CWRU Spring Break is March 13th to March 17, so NO CLASS**
  - Final: **Thursday May 4th**, 2023, 12:00PM - 3:00PM, Nord 356 or remote

7.1.2.1.2 Textbooks

- Text Books for DSCI353/353M/453
  - R4DS: Wickham: R for Data Science
  - ISLR: Intro to Statistical Learning with R, 2nd Ed.
  - DLwR: Deep Learning with R 2nd. Ed., Chollet, Kalinoxski, Allaire,
  - DLGB: Deep Learning, Goodfellow, Bengio, Courville
- Magazine Articles about Deep Learning
  - DL1 to DL12 are "Deep Learning" articles in 3-readings/2-articles/
- Books from DSCI351/351M/451
  - Peng: R Programming for Data Science
  - Peng: Exploratory Data Analysis with R
  - Open Intro Stats, v4
  - R4DS: Wickham: R for Data Science

7.1.2.1.3 Tidyverse Cheatsheets, Functions and Reading Your Code

- Look at the Tidyverse Cheatsheet
  - **Tidyverse For Beginners Cheatsheet**
    - In the Git/20s-dsci353-353m-453-prof/3-readings/3-CheatSheets/ folder
  - **Data Wrangling with dplyr and tidyr Cheatsheet**

  Tidyverse Functions & Conventions

  - The pipe operator `%>%`
  - Use `dplyr::filter()` to subset data row-wise.
  - Use `dplyr::arrange()` to sort the observations in a data frame
  - Use `dplyr::mutate()` to update or create new columns of a data frame
  - Use `dplyr::summarize()` to turn many observations into a single data point
  - Use `dplyr::arrange()` to change the ordering of the rows of a data frame
  - Use `dplyr::select()` to choose variables from a tibble,
    - keeps only variables you mention
  - Use `dplyr::rename()` keeps all the variables and renames variables
    - rename(iris, petal_length = Petal.Length)
  - These can be combined using `dplyr::group_by()`
    - which lets you perform operations "by group".
  - The `%in%` matches conditions provided by a vector using the c() function
  - The **forcats** package has tidyverse functions
    - for factors (categorical variables)
  - The **readr** package has tidyverse functions
    - to read_...., melt_... col_..., parse_... data and objects

Reading Your Code: Whenever you see

- The assignment operator `<-`, think **"gets"**
- The pipe operator, `%>%`, think **"then"**

7.1.2.1.4 Syllabus

7.1.2.2 ISLR6 is Model and Variable Selection

- So Regularization is reducing the dimensions / degrees of freedom in a model
  - By choosing fewer variables (e.g. variable selection)
  - By forcing β coefficients towards or to zero (e.g. Lasso)
  - Using constrained functions (e.g. natural splines)
  - By combining variables (e.g. PCA)

So its useful to recognize this need

- And learn all the ways we try to accomplish it

Principal component analysis

Principal component regression

Partial least squares regression

- PCA, PCR, PLS are covered in
  - ISLR 6.3
  - ISLR 10.2

Factor analysis

- Factor Analysis is covered in ESL Chapter 14.7

The similarities and differences among these approaches

- Can be subtle, or fine
- They are very powerful approaches.

7.1.2.3 A systematic/programmatic approach to data-driven modeling

- Its not just building one model
- Its building a series of models
- Using training and testing splits for Validation

So in Lab Exercises

- It was written so that you would do a large number of models
- And be discussing all the modeling choices
- And be able to demonstrate why you chose each choice
- And how, by quantitative statistical metrics,
  - The one you choose is the best one

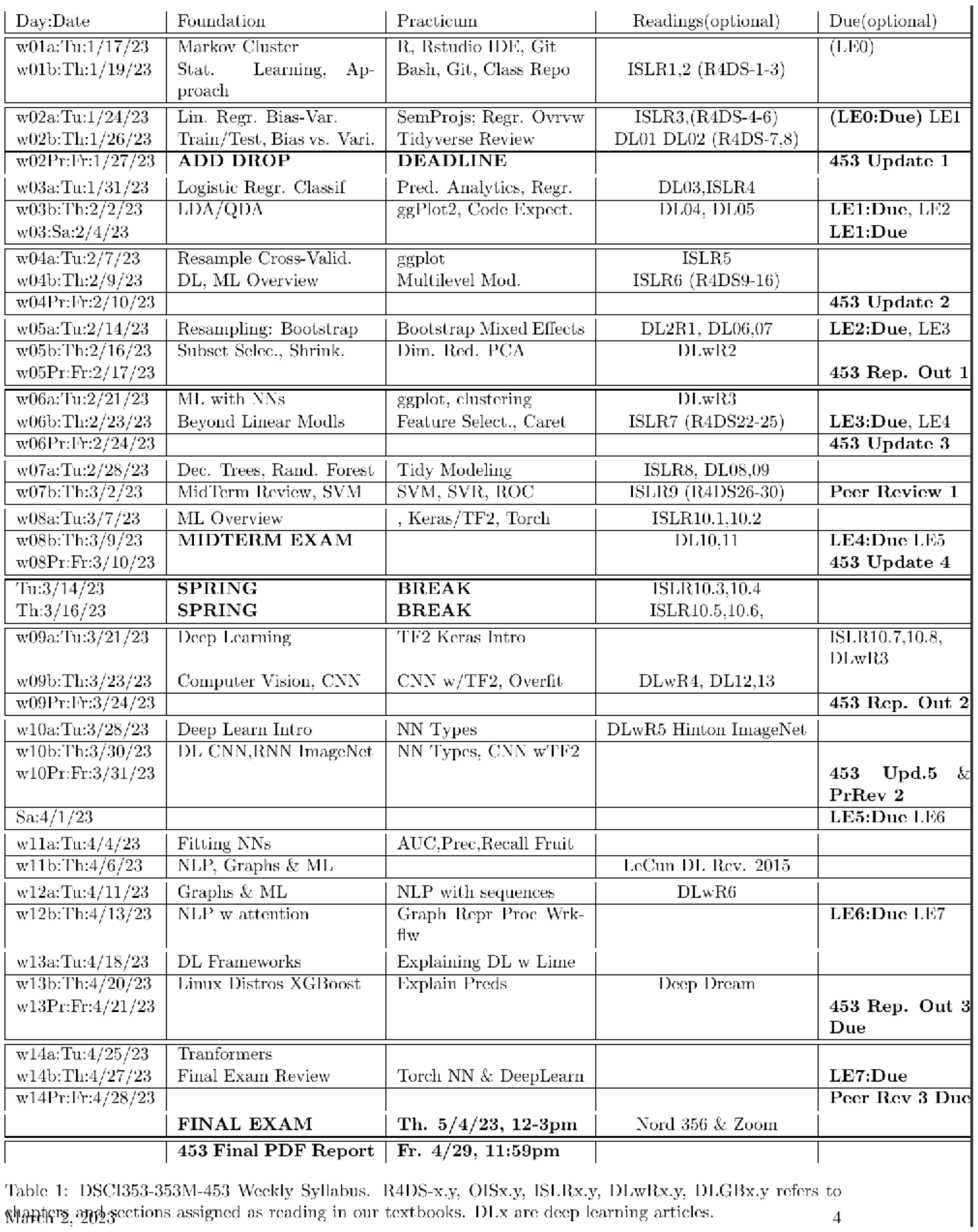
7.1.2.4 Tidy Modeling

- Using Tidy approaches to enable efficient meta-modeling and model selection

7.1.2.4.1 Introduction

- Lets look at the `tidymodels` R package

Let frame where `tidymodels` fit



Day:Date	Foundation	Practicum	Readings(optional)	Due(optional)
w01a:Tu:1/17/23	Markov Cluster	R, Rstudio IDE, Git		(LE0)
w01b:Th:1/19/23	Stat. Learning, Approach	Bash, Git, Class Repo	ISLR1,2 (R4DS-1-3)	
w02a:Tu:1/24/23	Lin. Regr. Bias-Var.	SemProjs; Regr. Ovrvw	ISLR3,(R4DS-4-6)	**(LE0:Due)** LE1
w02b:Th:1/26/23	Train/Test, Bias vs. Vari.	Tidyverse Review	DL01 DL02 (R4DS-7,8)	
w02Pr:Fr:1/27/23	**ADD DROP**	**DEADLINE**		**453 Update 1**
w03a:Tu:1/31/23	Logistic Regr. Classif	Pred. Analytics, Regr.	DL03,ISLR4	
w03b:Th:2/2/23	LDA/QDA	ggPlot2, Code Expect.	DL04, DL05	**LE1:Due**, LE2
w03:Sa:2/4/23				**LE1:Due**
w04a:Tu:2/7/23	Resample Cross-Valid.	ggplot	ISLR5	
w04b:Th:2/9/23	DL, ML Overview	Multilevel Mod.	ISLR6 (R4DS9-16)	
w04Pr:Fr:2/10/23				**453 Update 2**
w05a:Tu:2/14/23	Resampling: Bootstrap	Bootstrap Mixed Effects	DL2R1, DL06,07	**LE2:Due**, LE3
w05b:Th:2/16/23	Subset Selec., Shrink.	Dim. Red. PCA	DLwR2	
w05Pr:Fr:2/17/23				**453 Rep. Out 1**
w06a:Tu:2/21/23	ML with NNs	ggplot, clustering	DLwR3	
w06b:Th:2/23/23	Beyond Linear Modls	Feature Select., Caret	ISLR7 (R4DS22-25)	**LE3:Due**, LE4
w06Pr:Fr:2/24/23				**453 Update 3**
w07a:Tu:2/28/23	Dec. Trees, Rand. Forest	Tidy Modeling	ISLR8, DL08,09	
w07b:Th:3/2/23	MidTerm Review, SVM	SVM, SVR, ROC	ISLR9 (R4DS26-30)	**Peer Review 1**
w08a:Tu:3/7/23	ML Overview	, Keras/TF2, Torch	ISLR10.1,10.2	
w08b:Th:3/9/23	**MIDTERM EXAM**		DL10,11	**LE4:Due** LE5
w08Pr:Fr:3/10/23				**453 Update 4**
Tu:3/14/23	**SPRING**	**BREAK**	ISLR10.3,10.4	
Th:3/16/23	**SPRING**	**BREAK**	ISLR10.5,10.6,	
w09a:Tu:3/21/23	Deep Learning	TF2 Keras Intro		ISLR10.7,10.8, DLwR3
w09b:Th:3/23/23	Computer Vision, CNN	CNN w/TF2, Overfit	DLwR4, DL12,13	
w09Pr:Fr:3/24/23				**453 Rep. Out 2**
w10a:Tu:3/28/23	Deep Learn Intro	NN Types	DLwR5 Hinton ImageNet	
w10b:Th:3/30/23	DL CNN,RNN ImageNet	NN Types, CNN wTF2		
w10Pr:Fr:3/31/23				**453 Upd.5 & PrRev 2**
Sa:4/1/23				**LE5:Due** LE6
w11a:Tu:4/4/23	Fitting NNs	AUC,Prec,Recall Fruit		
w11b:Th:4/6/23	NLP, Graphs & ML		LeCun DL Rev. 2015	
w12a:Tu:4/11/23	Graphs & ML	NLP with sequences	DLwR6	
w12b:Th:4/13/23	NLP w attention	Graph Repr Proc Wrk-flw		**LE6:Due** LE7
w13a:Tu:4/18/23	DL Frameworks	Explaining DL w Lime		
w13b:Th:4/20/23	Linux Distros XGBoost	Explain Preds	Deep Dream	
w13Pr:Fr:4/21/23				**453 Rep. Out 3 Due**
w14a:Tu:4/25/23	Tranformers			
w14b:Th:4/27/23	Final Exam Review	Torch NN & DeepLearn		**LE7:Due**
w14Pr:Fr:4/28/23				**Peer Rev 3 Due**
	FINAL EXAM	**Th. 5/4/23, 12-3pm**	Nord 356 & Zoom	
	453 Final PDF Report	**Fr. 4/29, 11:59pm**		

Table 1: DSCI353-353M-453 Weekly Syllabus. R4DS-x.y, OISx.y, ISLRx.y, DLwRx.y, DLGBx.y refers to chapters and sections assigned as reading in our textbooks. DLx are deep learning articles.

March 2, 2023 4

Figure 1: DSCI351-351M-451 Syllabus

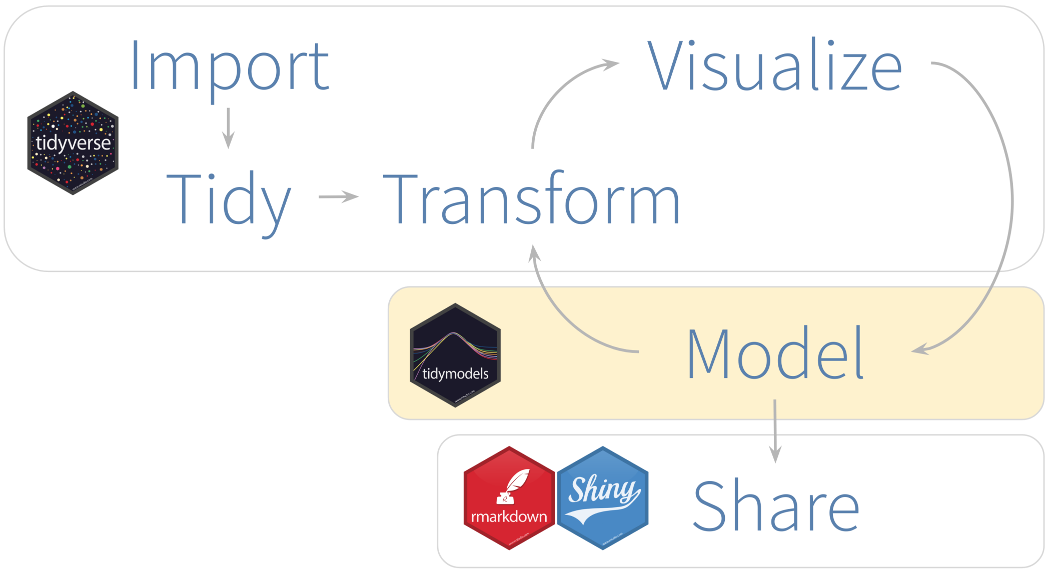


Figure 2: Figure 1. Tidy Model Work Flow, from R4DS

- in a data-driven analysis

Lets illustrates what step each package covers.

Developing models is a bit of a more complex

- developing models can benefit from having a `tidyverse`-friendly interface
- that is where `tidymodels` package can come in

It is important to clarify that

- the group of packages that make up `tidymodels`
  - do not implement statistical models themselves.
- Instead, they focus on making all the tasks
  - around fitting the model much easier.
- Those tasks are
  - data pre-processing and
  - results validation.

So `tidymodels` is an R metapackage

- Similar in this sense to the `caret` package
- `caret` Classification and Regression Training

In a way, the Model step itself has sub-steps.

- For these sub-steps, `tidymodels` provides one or several packages.
- Here we'll look at four `tidymodels` packages:

The Tidymodel packages

- `rsample` - Different types of re-samples
- `recipes` - Transformations for model data pre-processing
- `parsnip` - A common interface for model creation
- `yardstick` - Measure model performance

The following diagram

- illustrates each modeling step, and
- lines up the `tidymodels` packages we'll discuss

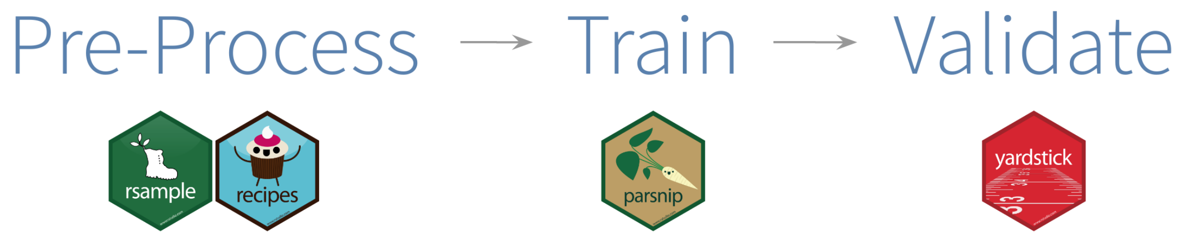


Figure 3: The modeling steps

In a given analysis, a `tidyverse` package may or may not be used.

- For example, not all projects need to work with time variables,
  - so there is no need to use functions from the `hms` package.
- The same idea applies to `tidymodels`.
- Depending on what type of modeling is going to be done,
  - only functions from some its packages will be used.

7.1.2.5 An Example

- Lets use the `iris` data set for an example.
  - Its data is already imported
  - and sufficiently tidy to move directly to modeling.

7.1.2.5.1 Load only the `tidymodels` library

- We can just load the tidymodel metapackage
  - Apart from loading its core modeling packages,
  - `tidymodels` also conveniently loads some `tidyverse` packages,
    - including `dplyr` and `ggplot2`.
  - Throughout this exercise, we will use some functions out of those packages,
    - but we don't have to explicitly load them into our R session.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.4.1     v purrr   1.0.1
## v tibble  3.1.8     v dplyr   1.1.0
## v tidyr   1.3.0     v stringr 1.5.0
## v readr   2.1.3     v forcats 1.0.0

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(tidymodels)
```

```
## -- Attaching packages -------------------------------------- tidymodels 1.0.0 --

## v broom        1.0.3     v rsample      1.1.1
## v dials        1.1.0     v tune         1.0.1
## v infer        1.0.4     v workflows    1.1.3
## v modeldata    1.1.0     v workflowsets 1.0.0
## v parsnip      1.0.4     v yardstick    1.1.0
## v recipes      1.0.4
```

```
## -- Conflicts ----------------------------------------- tidymodels_conflicts() --
## x scales::discard() masks purrr::discard()
## x dplyr::filter()   masks stats::filter()
## x recipes::fixed()  masks stringr::fixed()
## x dplyr::lag()      masks stats::lag()
## x yardstick::spec() masks readr::spec()
## x recipes::step()   masks stats::step()
## * Dig deeper into tidy modeling with R at https://www.tmwr.org
```

7.1.2.5.2 Pre-process

- This step focuses on making data
  - suitable for modeling by using data transformations.

All transformations can be accomplished

- with `dplyr`, or other `tidyverse` packages

Consider using `tidymodels` packages

- when model development is more heavy and complex.

7.1.2.5.3 Data Sampling

- The `rsample::initial_split()` function is specially built
  - to separate the data set into a training and testing set.
  - By default,
    - it holds 3/4 of the data for training
    - and the rest for testing.
  - That can be changed by passing the `prop` argument.
  - This function generates an `rsplit` object,
    - not a data frame.
  - The printed output shows the row count
    - for testing, training, and total.

```
iris_split <- initial_split(iris, prop = 0.6)
iris_split
```

```
## <Training/Testing/Total>
## <90/60/150>
```

```
## <90/60/150>
```

To access the observations reserved for training,

- use the `training()` function.

Similarly, use `testing()`

- to access the testing data.

```
iris_split %>%
  training() %>%
  glimpse()
```

```
## Rows: 90
## Columns: 5
## $ Sepal.Length <dbl> 5.4, 6.0, 6.1, 6.0, 6.6, 5.8, 6.3, 5.0, 6.2, 6.7, 5.0, 4.~
## $ Sepal.Width  <dbl> 3.9, 2.2, 3.0, 3.0, 3.0, 2.6, 2.5, 3.4, 3.4, 3.1, 3.4, 3.~
## $ Petal.Length <dbl> 1.3, 4.0, 4.9, 4.8, 4.4, 4.0, 5.0, 1.6, 5.4, 5.6, 1.5, 1.~
## $ Petal.Width  <dbl> 0.4, 1.0, 1.8, 1.8, 1.4, 1.2, 1.9, 0.4, 2.3, 2.4, 0.2, 0.~
## $ Species      <fct> setosa, versicolor, virginica, virginica, versicolor, ver~
```

```
## Observations: 90
## Variables: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.9, 5.4, 4…
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 3.1, 3.7, 3…
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.5, 1.5, 1…
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.1, 0.2, 0…
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, set…
```

These sampling functions

- are courtesy of the `rsample` package,
- which is part of `tidymodels`.

7.1.2.5.4 Pre-process interface

- In `tidymodels`, the `recipes` package
  - provides an interface that specializes in data pre-processing.

Within the package,

- the functions that start, or execute,
  - the data transformations are named after cooking actions.
- That makes the interface more user-friendly.

For example:

- `recipe()` - Starts a new set of transformations to be applied,
  - similar to the `ggplot()` command.
  - Its main argument is the model's formula.
- `prep()` - Executes the transformations
  - on top of the data that is supplied
  - (typically, the training data).

Each data transformation is a step.

- Functions correspond to specific types of steps,
  - each of which has a prefix of `step_`

There are several `step_` functions;

- in this example, we will use three of them:
- `step_corr()`
  - Removes variables
  - that have large absolute correlations with other variables
- `step_center()`
  - Normalizes numeric data to have a mean of zero
- `step_scale()`
  - Normalizes numeric data to have a standard deviation of one

Another nice feature is that the step can be applied

- to a specific variable,

- groups of variables,
- or all variables.

The `all_outcomes()` and `all_predictors()` functions

- provide a very convenient way to specify groups of variables.

For example, if we want the `step_corr()`

- to only analyze the predictor variables,
- we use `step_corr(all_predictors())`.
  - This capability saves us from having to enumerate each variable.

In the following example,

- we will put together
  - the `recipe()`, `prep()`, and `step` functions
  - to create a recipe object.
- The `training()` function
  - is used to extract that data set
  - from the previously created split sample data set.

```
iris_recipe <- training(iris_split) %>%
  recipe(Species ~.) %>%
  step_corr(all_predictors()) %>%
  step_center(all_predictors(), -all_outcomes()) %>%
  step_scale(all_predictors(), -all_outcomes()) %>%
  prep()
```

If we call the `iris_recipe` object,

- it will print details about the recipe.
- The Operations section describes what was done to the data.
- One of the operations entries in the example
  - explains that the correlation step
  - removed the `Petal.Length` variable.

```
iris_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          4
##
## Training data contained 90 data points and no missing data.
##
## Operations:
##
## Correlation filter on Petal.Length [trained]
## Centering for Sepal.Length, Sepal.Width, Petal.Width [trained]
## Scaling for Sepal.Length, Sepal.Width, Petal.Width [trained]
```

```
## Data Recipe
##
## Inputs:
##
```

```
##       role #variables
##    outcome          1
##  predictor          4
##
## Training data contained 90 data points and no missing data.
##
## Operations:
##
## Correlation filter removed Petal.Length [trained]
## Centering for Sepal.Length, Sepal.Width, Petal.Width [trained]
## Scaling for Sepal.Length, Sepal.Width, Petal.Width [trained]
```

7.1.2.5.5 Execute the pre-processing

- The testing data can now be transformed
    - using the exact same
        - steps,
        - weights,
        - and categorization
    - used to pre-process the training data.

To do this, another function

- with a cooking term is used: `bake()`.
- Notice that the `testing()` function is used
    - in order to extract the appropriate data set.

```
iris_testing <- iris_recipe %>%
  bake(testing(iris_split))

glimpse(iris_testing)
```

```
## Rows: 60
## Columns: 4
## $ Sepal.Length <dbl> -1.37182722, -1.49717184, -1.74786107, -1.12113798, -1.24~
## $ Sepal.Width  <dbl> 0.21695305, 0.66582143, -0.45634952, -0.00748114, -0.2319~
## $ Petal.Width  <dbl> -1.2513777, -1.1236861, -1.2513777, -1.3790693, -1.379069~
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s~
```

```
## Observations: 60
## Variables: 4
## $ Sepal.Length <dbl> -1.597601746, -1.138960096, 0.007644027, -0.7949788…
## $ Sepal.Width  <dbl> -0.41010139, 0.71517681, 2.06551064, 1.61539936, 0.…
## $ Petal.Width  <dbl> -1.2085003, -1.2085003, -1.2085003, -1.0796318, -1.…
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, set…
```

Performing the same operation

- over the training data is redundant,
    - because that data has already been prepped.

To load the prepared training data

- into a variable,
    - we use juice().
- It will extract the data
    - from the iris_recipe object.

```r
iris_training <- juice(iris_recipe)

glimpse(iris_training)
```

```
## Rows: 90
## Columns: 4
## $ Sepal.Length <dbl> -0.49441489, 0.25765283, 0.38299745, 0.25765283, 1.009720~
## $ Sepal.Width  <dbl> 1.78799238, -2.02738885, -0.23191533, -0.23191533, -0.231~
## $ Petal.Width  <dbl> -0.99599447, -0.22984488, 0.79168791, 0.79168791, 0.28092~
## $ Species      <fct> setosa, versicolor, virginica, virginica, versicolor, ver~
```

```r
## Observations: 90
## Variables: 4
## $ Sepal.Length <dbl> -0.7949789, -1.0242997, -1.2536205, -1.3682809, -0.…
## $ Sepal.Width  <dbl> 0.94023245, -0.18504575, 0.26506553, 0.04000989, 1.…
## $ Petal.Width  <dbl> -1.2085003, -1.2085003, -1.2085003, -1.2085003, -1.…
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, set…
```

7.1.2.5.6 Model training

- In R, there are multiple packages that fit the same type of model.
    - It is common for each package to provide a unique interface.
    - In other words,
        * things such as an argument for the same model attribute
        * is defined differently for each package.

For example, the `ranger` and `randomForest` packages

- fit Random Forest models.
- In the `ranger()` function,
    - to define the number of trees we use `num.trees`.
- In `randomForest` package,
    - that argument is named `ntree`.
- It is not easy to switch between packages
    - to run the same model.

Instead of replacing the modeling package,

- `tidymodels` replaces the interface.

Better said, `tidymodels`

- provides a single set of functions and arguments
    - to define a model.
- It then fits the model
    - against the requested modeling package.

In the example below,

- the `rand_forest()` function is used
    - to initialize a Random Forest model.
    - To define the number of trees,
    - the trees argument is used.
- To use the ranger version of Random Forest,
    - the `set_engine()` function is used.
- Finally, to execute the model,
    - the `fit()` function is used.
- The expected arguments are the formula and data.

- Notice that the model runs
    - on top of the juiced trained data.

```
iris_ranger <- rand_forest(trees = 100, mode = "classification") %>%
  set_engine("ranger") %>%
  fit(Species ~ ., data = iris_training)
```

The payoff is that

- if we now want to run the same model
    - against randomForest,
- we simply change the value in set_engine()
    - to `randomForest`.

```
iris_rf <-  rand_forest(trees = 100, mode = "classification") %>%
  set_engine("randomForest") %>%
  fit(Species ~ ., data = iris_training)
```

It is also worth mentioning that

- the model is not defined
    - in a single, large function with a lot of arguments.
- The model definition is separated
    - into smaller functions
    - such as `fit()` and `set_engine()`.
- This allows for a more flexible, and easier to learn, interface.

7.1.2.5.7 Predictions

- Instead of a vector,
    - the `predict()` function
        - ran against a parsnip model
        - returns a tibble.
    - By default, the prediction variable
        - is called `.pred_class`.
    - In the example, notice that
        - the baked testing data is used.

```
predict(iris_ranger, iris_testing)
```

```
## # A tibble: 60 x 1
##    .pred_class
##    <fct>
##  1 setosa
##  2 setosa
##  3 setosa
##  4 setosa
##  5 setosa
##  6 setosa
##  7 setosa
##  8 setosa
##  9 setosa
## 10 setosa
## # ... with 50 more rows
```

```
## # A tibble: 60 x 1
##    .pred_class
##    <fct>
##  1 setosa
##  2 setosa
##  3 setosa
##  4 setosa
##  5 setosa
##  6 setosa
##  7 setosa
##  8 setosa
##  9 setosa
## 10 setosa
## # … with 50 more rows
```

It is very easy to add the predictions

- to the baked testing data
- by using `dplyr`'s `bind_cols()` function.

```
iris_ranger %>%
  predict(iris_testing) %>%
  dplyr::bind_cols(iris_testing) %>%
  glimpse()
```

```
## Rows: 60
## Columns: 5
## $ .pred_class  <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s~
## $ Sepal.Length <dbl> -1.37182722, -1.49717184, -1.74786107, -1.12113798, -1.24~
## $ Sepal.Width  <dbl> 0.21695305, 0.66582143, -0.45634952, -0.00748114, -0.2319~
## $ Petal.Width  <dbl> -1.2513777, -1.1236861, -1.2513777, -1.3790693, -1.379069~
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s~
```

```
## Observations: 60
## Variables: 5
## $ .pred_class  <fct> setosa, setosa, setosa, setosa, setosa, setosa, set…
## $ Sepal.Length <dbl> -1.597601746, -1.138960096, 0.007644027, -0.7949788…
## $ Sepal.Width  <dbl> -0.41010139, 0.71517681, 2.06551064, 1.61539936, 0.…
## $ Petal.Width  <dbl> -1.2085003, -1.2085003, -1.2085003, -1.0796318, -1.…
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, set…
```

7.1.2.5.8 Model Validation

- Use the `metrics()` function
  - to measure the performance of the model.

It will automatically choose metrics

- appropriate for a given type of model.

The function expects a tibble

- that contains the actual results (truth)
- and what the model predicted (estimate).

```
iris_ranger %>%
  predict(iris_testing) %>%
  bind_cols(iris_testing) %>%
  metrics(truth = Species, estimate = .pred_class)
```

```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.917
## 2 kap      multiclass     0.874
```

```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.917
## 2 kap      multiclass     0.874
```

Because of the consistency of the new interface,

- measuring the same metrics
  - against the `randomForest` model
- is as easy as replacing the model variable
  - at the top of the code.

```
iris_rf %>%
  predict(iris_testing) %>%
  dplyr::bind_cols(iris_testing) %>%
  metrics(truth = Species, estimate = .pred_class)
```

```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.917
## 2 kap      multiclass     0.874
```

```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.883
## 2 kap      multiclass     0.824
```

7.1.2.5.9 Per classifier metrics

- It is easy to obtain the probability
  - for each possible predicted value
    - by setting the type argument to prob.
  - That will return a tibble
    - with as many variables
    - as there are possible predicted values.
  - Their name will default
    - to the original value name,
    - prefixed with .pred_.

```
iris_ranger %>%
  predict(iris_testing, type = "prob") %>%
  glimpse()
```

```
## Rows: 60
## Columns: 3
## $ .pred_setosa     <dbl> 0.95980556, 0.98407143, 0.93060317, 0.92616270, 0.950~
## $ .pred_versicolor <dbl> 0.02852778, 0.00900000, 0.06939683, 0.06169444, 0.043~
## $ .pred_virginica  <dbl> 0.011666667, 0.006928571, 0.000000000, 0.012142857, 0~
## Observations: 60
## Variables: 3
## $ .pred_setosa     <dbl> 0.677480159, 0.978293651, 0.783250000, 0.983972…
## $ .pred_versicolor <dbl> 0.295507937, 0.011706349, 0.150833333, 0.001111…
## $ .pred_virginica  <dbl> 0.02701190, 0.01000000, 0.06591667, 0.01491667,…
```

Again, use `dplyr::bind_cols()`

- to append the predictions
- to the baked testing data set.

```
iris_probs <- iris_ranger %>%
  predict(iris_testing, type = "prob") %>%
  dplyr::bind_cols(iris_testing)

glimpse(iris_probs)
```

```
## Rows: 60
## Columns: 7
## $ .pred_setosa     <dbl> 0.95980556, 0.98407143, 0.93060317, 0.92616270, 0.950~
## $ .pred_versicolor <dbl> 0.02852778, 0.00900000, 0.06939683, 0.06169444, 0.043~
## $ .pred_virginica  <dbl> 0.011666667, 0.006928571, 0.000000000, 0.012142857, 0~
## $ Sepal.Length     <dbl> -1.37182722, -1.49717184, -1.74786107, -1.12113798, -~
## $ Sepal.Width      <dbl> 0.21695305, 0.66582143, -0.45634952, -0.00748114, -0.~
## $ Petal.Width      <dbl> -1.2513777, -1.1236861, -1.2513777, -1.3790693, -1.37~
## $ Species          <fct> setosa, setosa, setosa, setosa, setosa, setosa, setos~
```

```
## Observations: 60
## Variables: 7
## $ .pred_setosa     <dbl> 0.677480159, 0.978293651, 0.783250000, 0.983972…
## $ .pred_versicolor <dbl> 0.295507937, 0.011706349, 0.150833333, 0.001111…
## $ .pred_virginica  <dbl> 0.02701190, 0.01000000, 0.06591667, 0.01491667,…
## $ Sepal.Length     <dbl> -1.597601746, -1.138960096, 0.007644027, -0.794…
## $ Sepal.Width      <dbl> -0.41010139, 0.71517681, 2.06551064, 1.61539936…
## $ Petal.Width      <dbl> -1.2085003, -1.2085003, -1.2085003, -1.0796318,…
## $ Species          <fct> setosa, setosa, setosa, setosa, setosa, setosa,…
```

Now that everything is in one tibble,

- it is easy to calculate curve methods.
- In this case we are using `gain_curve()`.

```
iris_probs %>%
  gain_curve(Species, .pred_setosa:.pred_virginica) %>%
  glimpse()
```

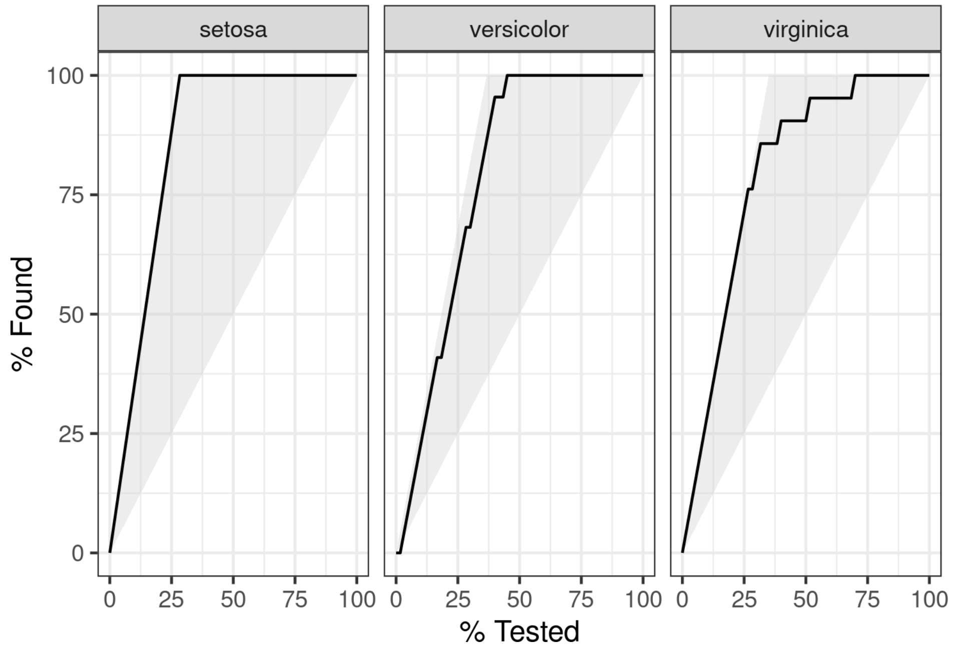
```
## Rows: 137
## Columns: 5
## $ .level          <chr> "setosa", "setosa", "setosa", "setosa", "setosa", "set~
## $ .n              <dbl> 0, 1, 2, 3, 4, 5, 7, 9, 10, 12, 13, 14, 15, 16, 17, 18~
## $ .n_events       <dbl> 0, 1, 2, 3, 4, 5, 7, 9, 10, 12, 13, 14, 15, 16, 17, 17~
## $ .percent_tested <dbl> 0.000000, 1.666667, 3.333333, 5.000000, 6.666667, 8.33~
## $ .percent_found  <dbl> 0.000000, 5.882353, 11.764706, 17.647059, 23.529412, 2~
```

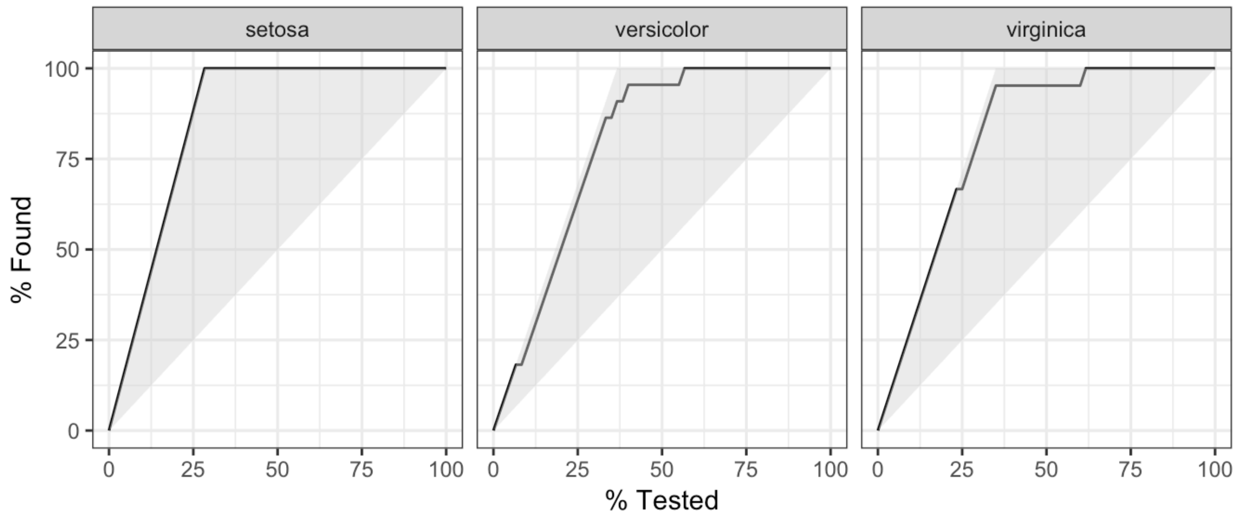
```
## Observations: 141
## Variables: 5
## $ .level          <chr> "setosa", "setosa", "setosa", "setosa", "setosa"…
## $ .n              <dbl> 0, 1, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 16, …
## $ .n_events       <dbl> 0, 1, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 16, …
## $ .percent_tested <dbl> 0.000000, 1.666667, 5.000000, 6.666667, 8.333333…
## $ .percent_found  <dbl> 0.000000, 5.882353, 17.647059, 23.529412, 29.411…
```

The curve methods

- include an `autoplot()` function
- that easily creates a `ggplot2` visualization.

```
iris_probs %>%
  gain_curve(Species, .pred_setosa:.pred_virginica) %>%
  autoplot()
```



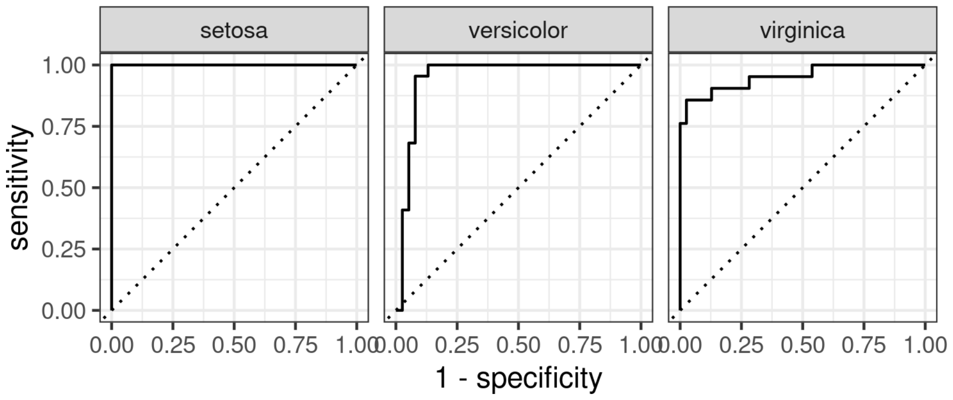


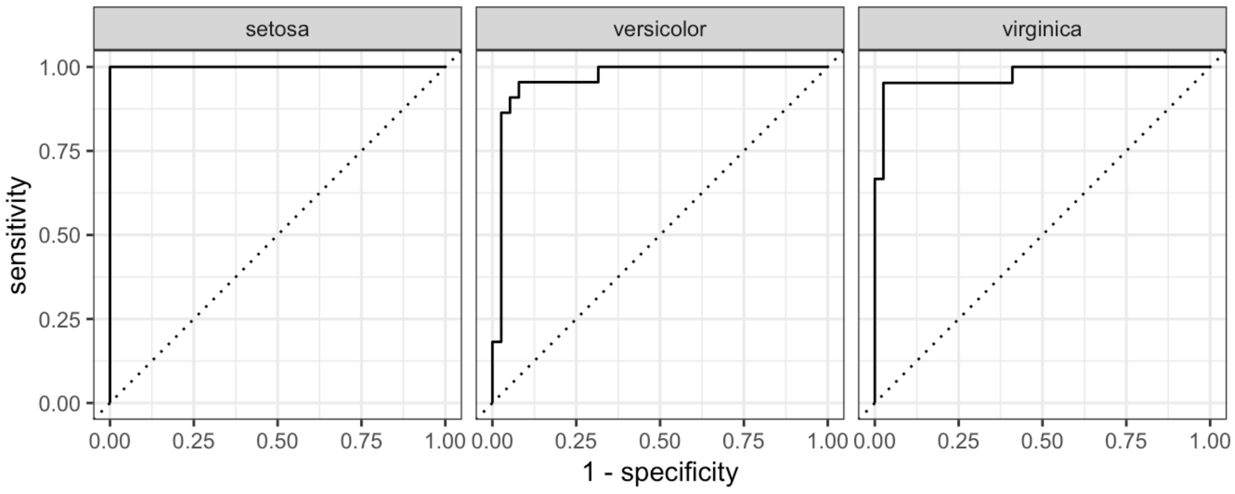
This is an example of a `roc_curve()`.

- roc means “Receiver Operating Characteristic”

- Again, because of the consistency of the interface,
  - only the function name needs to be modified;
  - even the argument values remain the same.

```
iris_probs %>%
  roc_curve(Species, .pred_setosa:.pred_virginica) %>%
  autoplot()
```





To measured the combined

- single predicted value
  - and the probability of each possible value,
- combine the two prediction modes
  - (with and without prob type).

In this example,

- using `dplyr`'s `select()`
- makes the resulting tibble easier to read.

```
predict(iris_ranger, iris_testing, type = "prob") %>%
  bind_cols(predict(iris_ranger, iris_testing)) %>%
  bind_cols(select(iris_testing, Species)) %>%
  glimpse()
```

```
## Rows: 60
## Columns: 5
## $ .pred_setosa     <dbl> 0.95980556, 0.98407143, 0.93060317, 0.92616270, 0.950~
## $ .pred_versicolor <dbl> 0.02852778, 0.00900000, 0.06939683, 0.06169444, 0.043~
## $ .pred_virginica  <dbl> 0.011666667, 0.006928571, 0.000000000, 0.012142857, 0~
## $ .pred_class      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setos~
## $ Species          <fct> setosa, setosa, setosa, setosa, setosa, setosa, setos~
```

```
## Observations: 60
## Variables: 5
## $ .pred_setosa     <dbl> 0.677480159, 0.978293651, 0.783250000, 0.983972…
## $ .pred_versicolor <dbl> 0.295507937, 0.011706349, 0.150833333, 0.001111…
## $ .pred_virginica  <dbl> 0.02701190, 0.01000000, 0.06591667, 0.01491667,…
## $ .pred_class      <fct> setosa, setosa, setosa, setosa, setosa, setosa,…
## $ Species          <fct> setosa, setosa, setosa, setosa, setosa, setosa,…
```

Pipe the resulting table into `metrics()`.

In this case, specify `.pred_class` as the estimate.

```
predict(iris_ranger, iris_testing, type = "prob") %>%
  bind_cols(predict(iris_ranger, iris_testing)) %>%
  bind_cols(select(iris_testing, Species)) %>%
  metrics(Species, .pred_setosa:.pred_virginica, estimate = .pred_class)
```

```
## # A tibble: 4 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 accuracy    multiclass     0.917
## 2 kap         multiclass     0.874
## 3 mn_log_loss multiclass     0.295
## 4 roc_auc     hand_till      0.970
```

```
## # A tibble: 4 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 accuracy    multiclass     0.917
## 2 kap         multiclass     0.874
## 3 mn_log_loss multiclass     0.274
## 4 roc_auc     hand_till      0.980
```

7.1.2.5.10 Summary

- This end-to-end example can serve as a gentle introduction to `tidymodels`.

The number of functions,

- and options of such functions,
- were kept at a minimum for the purposes of this demonstration,
  - but there is much more that can be done with this wonderful metapackages.

Hopefully, this serves as an initial example

- you can build from
- for automating your data-driven modeling

7.1.2.6 Links

- Background on these topics

Principal Components Analysis

Principal Components Regression

Partial Least Squares Regression

Factor Analysis

Edgar Ruiz, A Gentle Introduction to tidymodels