Advanced Analytics with Tidymodels in R

Prof. Dr. Jan Kirenz

2020-11-30

Contents

Welcome

This book provides an introduction to advanced analytics with R using the tidymodels framework, a collection of packages for modeling and machine learning using tidyverse principles. You will learn how to build, evaluate, compare, and tune predictive models.

We'll cover key concepts in statistical learning and machine learning including overfitting, the holdout method, the bias-variance trade-off, ensembling, cross-validation, and feature engineering.

Part I

BUILD A MODEL

Chapter 1

Build a Model

The following content is adapted from the excellent book "Hands-on machine learning with scikit-learn, keras and tensorflow" from Géron (2019).

In this chapter you will learn how to specify a regression model with the tidymodels package. To use the code in this article, you will need to install the following packages:

- tidyverse
- tidymodels
- skimr
- GGally
- ggmap

```
library(tidyverse)
library(tidymodels)
library(skimr)
library(GGally)
library(ggmap)
```

In this example, our goal is to build a model of housing prices in California. In particular, the model should learn from California census data and be able to predict the median house price in any district (population of 600 to 3000 people), given some predictor variables. We use the root mean square error (RMSE) as a performance measure for our regression problem.

1.1 Data understanding

In Data Understanding, you will learn how to:

- Import data
- Get an overview about the data structure
- Split data into training and test set using stratified sampling
- Discover and visualize the data to gain insights

1.1.1 Imort Data

First of all, let's import the data:

```
LINK <- "https://raw.githubusercontent.com/kirenz/datasets/master/housing.csv"
housing <- read_csv(LINK)
```

1.1.2 Data overview

Next, we take a look at the data structure:

California census top 4 rows of the DataFrame:

```
head(housing, 4)
```

Data info:

```
glimpse(housing)
```

Data summary of numerical and categorical attributes using a function from the package `skimr`:

```
skim(housing)
```

Count levels of our categorical variable:

```
housing %>%
  count(ocean_proximity,
    sort = TRUE
  )
```

The function `ggscatmat` from the package `GGally` creates a matrix with scatterplots, densities and correlations for numeric columns. In our code, we enter the dataset `housing`, an color column for our categorical variable `ocean_proximity`, and an alpha level of 0.8 (for transparency).

```
ggscatmat(housing, color = "ocean_proximity", alpha = 0.8)
```

To obtain an overview of even more visualizations, we can use the function ggpairs:

```
ggpairs(housing)
```

1.1.3 Data splitting

Let's assume we would know that the median income is a very important attribute to predict median housing prices. Therefore, we would want to create a training and test set using stratified sampling. A *stratum* (plural strata) refers to a subset (part) of the population (entire collection of items under consideration) which is being sampled. Take a look at **??**

```
housing %>%
  ggplot(aes(median_income)) +
  geom_histogram(bins = 30)
```

We want to ensure that the test set is representative of the the various categories of incomes in the whole dataset. In other words, we would like to have instances for each *stratum*, or else the stimate of a strtum's importance may be biased. This means that you should not have too many starta, and each stratum should be large enough. We use 5 strata in our example.

```
set.seed(42)

new_split <- initial_split(housing,
  prop = 3 / 4,
  strata = median_income,
  breaks = 5
)

new_train <- training(new_split)
new_test <- testing(new_split)
```

1.1.4 Data exploration

A Geographical scatterplot of the data:

```
housing %>%
  ggplot(aes(x = longitude, y = latitude)) +
  geom_point(color = "cornflowerblue")
```

A better visualization that highlights high-density areas:

```
housing %>%
  ggplot(aes(x = longitude, y = latitude)) +
  geom_point(color = "cornflowerblue", alpha = 0.1)
```

California housing prices: red is expensive, purple is cheap and larger circles indicate areas with a larger population.

```
housing %>%
  ggplot(aes(x = longitude, y = latitude)) +
  geom_point(aes(size = population, color = median_house_value),
    alpha = 0.4
  ) +
  scale_colour_gradientn(colours = rev(rainbow(4)))
```

```
library(ggmap)

qmplot(
  x = longitude,
  y = latitude,
  data = housing,
  geom = "point",
  color = median_house_value,
  size = population,
  alpha = 0.4
) +
  scale_colour_gradientn(colours = rev(rainbow(4)))
```

1.2 Model type

1. Pick an `model type`: choose from this list
2. Set the `engine`: choose from this list
3. Set the `mode`: regression or classification

```
library(tidymodels)

lm_spec <- # your model specification
```

```
  linear_reg() %>% # model type
  set_engine(engine = "lm") %>% # model engine
  set_mode("regression") # model mode

# Show your model specification
lm_spec
```

1.3 Model training

In the training process, you run an algorithm on data and thereby produce a model. This process is also called model fitting.

```
lm_fit <- # your fitted model
  fit(
    lm_spec, # your model specification
    Sale_Price ~ Gr_Liv_Area, # a Linear Regression formula
    data = ames # your data
  )

# Show your fitted model
lm_fit
```

1.4 Model predictions

We use our fitted model to make predictions.

```
price_pred <-
  lm_fit %>%
  predict(new_data = ames) %>%
  mutate(price_truth = ames$Sale_Price)

head(price_pred)
```

If we would want to make predictions for new houses, we could proceed as follows:

```
# New values (our x variable)
new_homes <-
  tibble(Gr_Liv_Area = c(334, 1126, 1442, 1500, 1743, 5642))

# Prediction for new houses (predict y)
```

```
lm_fit %>%
  predict(new_data = new_homes)
```

1.5 Model evaluation

We use the Root Mean Squared Error (RMSE) to evaluate our regression model. Therefore, we use the function $rmse(data, truth, estimate)$.

```
rmse(
  data = price_pred,
  truth = price_truth,
  estimate = .pred
)
```

Chapter 2

Process with data splitting

The best way to measure a model's performance at predicting new data is to actually predict new data.

This function "splits" data randomly into a single testing and a single training set: `initial_split(data, prop = 3/4, strata, breaks)`. We also use stratified sampling in this example.

2.1 Data splitting

```
set.seed(100)

ames_split <- initial_split(ames,
  strata = Sale_Price,
  breaks = 4
)

ames_train <- training(ames_split)
ames_test <- testing(ames_split)
```

2.2 Model type

```
lm_spec_2 <-
  linear_reg() %>%
  set_engine(engine = "lm") %>%
  set_mode("regression")
```

2.3 Model training

```
lm_fit_2 <-
  lm_spec_2 %>%
  fit(Sale_Price ~ Gr_Liv_Area,
    data = ames_train
  ) # only use training data
```

2.4 Model predictions

```
price_pred_2 <-
  lm_fit %>%
  predict(new_data = ames_test) %>%
  mutate(price_truth = ames_test$Sale_Price)
```

2.5 Model evaluation

```
rmse(price_pred_2,
  truth = price_truth,
  estimate = .pred
)
```

Part II

K-FOLD CROSS VALIDATION

This section is based on Alisson Hill's excellent tidymodels workshop.

Chapter 3

Import data

```
library(tidyverse)

ames <- read_csv("https://raw.githubusercontent.com/kirenz/datasets/master/ames.csv")

ames <-
  ames %>%
  select(Sale_Price, Gr_Liv_Area)

glimpse(ames)
```

Chapter 4

Build a model

4.1 Model specification

Build the model:

```
library(tidymodels)

lm_spec <-
  linear_reg() %>%
  set_engine("lm") %>%
  set_mode(mode = "regression")
```

4.2 Data split

```
set.seed(100)

new_split <- initial_split(ames)
new_train <- training(new_split)
new_test <- testing(new_split)
```

Chapter 5

Fit the model

```
lm_fit <-
  lm_spec %>%
  fit(Sale_Price ~ Gr_Liv_Area, data = new_train)
```

Make predictions and calculate the RMSE for our training data.

```
lm_fit %>%
  predict(new_train) %>%
  mutate(truth = new_train$Sale_Price) %>%
  rmse(truth, .pred)
```

Chapter 6

Evaluate model

To further evaluate our simple Linear Regression model, let's build a validation set. Note that this process is usually used to compare the performance of multiple models and to tune hyperparameters. In this tutorial however, we only want to demonstrate the process and therefore only use one model.

We can build a set of k-Fold cross validation sets and use this set of resamples to estimate the performance of our model using the `fit_resamples()` function. Note that this function does not do any tuning of the model parameters. The function is only used for computing performance metrics across some set of resamples. It will fit our model `lm_spec` to each resample and evaluate on the heldout set from each resample. It is important to note, that the function does not even keep the models it trains.

First of all, we build a set of 10 validation folds with the function `vfold_cv` (we also use stratified sampling in this example):

```
set.seed(100)

cv_folds <-
  vfold_cv(new_train,
    v = 10,
    strata = Sale_Price,
    breaks = 4
  )

cv_folds
```

Next, let's fit the model on each of the 10 folds with `fit_resamples()` and store the results as `lm_res`:

```
lm_res <-
  lm_spec %>%
  fit_resamples(
    Sale_Price ~ Gr_Liv_Area,
    resamples = cv_folds
  )
```

Now we can collect the performance metrics with `collect_metrics()`. The metrics show the average performance across all folds.

```
lm_res %>%
  collect_metrics()
```

If we would be interested in the results of every split, we could use the option `summarize = FALSE`:

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

Chapter 7

Evaluate final model

Finally, let's use our testing data and see how we can expect this model to perform on new data.

```
lm_fit %>%
  predict(new_test) %>%
  mutate(truth = new_test$Sale_Price) %>%
  rmse(truth, .pred)
```

Part III

RECIPES

The following content is based on the tidymodels documentation and Alisson Hill's tidymodels workshop.

In this tutorial, we'll explore a tidymodels package, `recipes`, which is designed to help you preprocess your data before training your model.

Recipes are built as a series of preprocessing steps, such as:

- converting qualitative predictors to indicator variables (also known as dummy variables),
- transforming data to be on a different scale (e.g., taking the logarithm of a variable),
- transforming whole groups of predictors together,
- extracting key features from raw variables (e.g., getting the day of the week out of a date variable),

and so on. If you are familiar with R's formula interface, a lot of this might sound familiar and like what a formula already does. Recipes can be used to do many of the same things, but they have a much wider range of possibilities. This article shows how to use recipes for modeling.

In summary, the idea of the recipes package is to define a recipe or blueprint that can be used to sequentially define the encodings and preprocessing of the data (i.e. "feature engineering") before we build our models.

Chapter 8

Data preparation

Import data and split the data into training and testing sets using `initial_split()`

```
library(tidyverse)
library(tidymodels)

ames <- read_csv("https://raw.githubusercontent.com/kirenz/datasets/master/ames.csv")

ames <- ames %>%
  select(-matches("Qu"))

set.seed(100)

new_split <- initial_split(ames)
new_train <- training(new_split)
new_test <- testing(new_split)
```

Chapter 9

Recipe

Next, we use a `recipe()` to build a set of steps for data preprocessing and feature engineering.

- First, we must tell the `recipe()` what our model is going to be (using a formula here) and what our training data is.
- `step_novel()` will convert all nominal variables to factors.
- We then convert the factor columns into (one or more) numeric binary (0 and 1) variables for the levels of the training data.
- We remove any numeric variables that have zero variance.
- We normalize (center and scale) the numeric variables.

9.1 recipe()

```
ames_rec <-
  recipe(Sale_Price ~ ., data = new_train) %>%
  step_novel(all_nominal(), -all_outcomes()) %>%
  step_dummy(all_nominal()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_predictors())

# Show the content of our recipe
ames_rec
```

9.2 prep()

Finally, we `prep()` the `recipe()`. This means we actually do something with the steps and our training data.

```
ames_prep <- prep(ames_rec)
```

Print a summary of our prepped recipe:

```
summary(ames_prep)
```

9.3 juice()

To obtain the Dataframe from the prepped recipe, we use the function `juice()`. When we `juice()` the recipe, we squeeze that training data back out, transformed in the ways we specified:

```
juice(ames_prep)
```

The prepped recipe `ames_prep` contains all our transformations for data preprocessing and feature engineering, *as well as* the data these transformations were estimated from.

9.4 bake()

We now can simply apply all of the recipe transformations to the testing data. The function to perform this is called `bake()`:

```
test_trans <- bake(ames_prep, new_data = new_test)
```

Now it's time to **specify** and then **fit** our models.

Chapter 10

Model building

10.1 Specify model

```
lm_spec <-
  linear_reg() %>%
  set_engine("lm") %>%
  set_mode(mode = "regression")
```

10.2 Fit model

```
lm_fit <-
  lm_spec %>%
  fit(Sale_Price ~ ., data = juice(ames_prep))
```

10.3 Evaluate model

```
set.seed(100)

cv_folds <-
  vfold_cv(juice(ames_prep),
    v = 10,
    strata = Sale_Price,
    breaks = 4
```

```
  )

lm_res <-
  lm_spec %>%
  fit_resamples(
    Sale_Price ~ .,
    resamples = cv_folds
  )

lm_res %>%
  collect_metrics()
```

10.4 Evaluate final model

Finally, let's use our testing data and see how we can expect this model to perform on new data.

```
lm_fit %>%
  predict(test_trans) %>%
  mutate(truth = test_trans$Sale_Price) %>%
  rmse(truth, .pred)
```

Chapter 11

Components of our recipe

Let's have a closer look at the different components of the recipe.

11.1 recipe()

First of all, we created a simple recipe (we call it `rec`) containing only an outcome (`Sale_Price`) and predictors (all other variables in the dataset: `.`). To demonstrate the use of recipes step by step, we create a new object with the name `rec`:

```
rec <- recipe(Sale_Price ~ ., data = ames)
```

The formula `Sale_Price ~ .` indicates outcomes vs predictors.

11.2 Helper functions

Here some helper functions for selecting sets of variables:

- `all_predictors()`: Each x variable (right side of ~)
- `all_outcomes()`: Each y variable (left side of ~)
- `all_numeric()`: Each numeric variable
- `all_nominal()`: Each categorical variable (e.g. factor, string)
- `dplyr::select()` helpers starts_with('Lot_'), etc.

11.3 step_novel()

`step_novel()` will convert all nominal variables to factors. It adds a catch-all level to a factor for any new values, which lets R intelligently predict new levels in the test set. Missing values will remain missing.

```
rec %>%
  step_novel(all_nominal(), -all_outcomes())
```

11.4 step_dummy()

Converts nominal data into dummy variables.

```
rec %>%
  step_dummy(all_nominal())
```

11.5 step_zv()

`step_zv()` removes zero variance variables (variables that contain only a single value).

```
rec %>%
  step_zv(all_predictors())
```

When the recipe is applied to the data set, a column could contain only zeros. This is a "*zero-variance predictor*" that has no information within the column. While some R functions will not produce an error for such predictors, it usually causes warnings and other issues. `step_zv()` will remove columns from the data when the training set data have a single value- This step should be added to the recipe after `step_dummy()`.

11.6 step_normalize()

Centers then scales numeric variable (mean = 0, sd = 1)

```
rec %>%
  step_normalize(all_numeric())
```

Chapter 12

Workflows

To combine the data preparation with the model building, we alternatively could use the package workflows.

A workflow is an object that can bundle together your pre-processing, modeling, and post-processing requests. Note that you don't have to use the `prep()`, `bake()` or `juice()` if you use workflows

12.1 workflow()

```
new_wf <-
  workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(lm_spec)
```

12.2 fit()

```
new_wf_fit <-
  new_wf %>%
  fit(data = new_train)
```

12.3 predict() & rmse()

Make predictions and calculate the RMSE for our training data.

```
new_wf_fit %>%
  predict(new_train) %>%
  mutate(truth = new_train$Sale_Price) %>%
  rmse(truth, .pred)
```

Part IV

WORKFLOWS & MODELS

The following content is based on the excellent tidymodels documentation.

In this tutorial, we'll explore the tidymodels packages `recipes` and `workflows` which are designed to help you preprocess your data before training your model.

`Recipes` are built as a series of preprocessing steps, such as:

- converting qualitative predictors to indicator variables (also known as dummy variables),
- transforming data to be on a different scale (e.g., taking the logarithm of a variable),
- transforming whole groups of predictors together,
- extracting key features from raw variables (e.g., getting the day of the week out of a date variable),

and so on. If you are familiar with R's formula interface, a lot of this might sound familiar and like what a formula already does. Recipes can be used to do many of the same things, but they have a much wider range of possibilities. This article shows how to use recipes for modeling.

To combine all of the steps discussed above with model building, we can use the package workflows. A workflow is an object that can bundle together your pre-processing, modeling, and post-processing requests.

To use the code in this tutorial, you will need to install the following packages:

- `nycflights13`,
- `skimr`, and
- `tidymodels`.

```
library(tidymodels)
# for flight data:
library(nycflights13)
# for variable summaries:
library(skimr)
```

Chapter 13

Data preparation

Let's use the `nycflights13` data to predict whether a plane arrives more than 30 minutes late. This data set contains information on 325,819 flights departing near New York City in 2013. Furthermore, it contains weather data (hourly meterological data for LGA, JFK and EWR).

Let's start by loading the data and making a few changes to the variables:

```
flight_data_all <-
  flights %>%
  mutate(
    # Convert the arrival delay to a factor
    arr_delay = ifelse(arr_delay >= 30,
      "late",
      "on_time"
    ),
    arr_delay = factor(arr_delay),
    # We will use the date (not date-time)
    # in the recipe below
    date = as.Date(time_hour)
  ) %>%
  # Include  weather data
  inner_join(weather, by = c("origin", "time_hour")) %>%
  # Only retain the specific columns we will use
  select(
    dep_time, flight, origin,
    dest, air_time, distance,
    carrier, date, arr_delay, time_hour
  ) %>%
  # Exclude missing data
  na.omit() %>%
```

```
  # For creating models, it is
  # better to have qualitative columns
  # encoded as factors (instead of character strings)
  mutate(across(where(is.character), as.factor))
```

To speed up later calculations we only use a sample of the data:

```
set.seed(123)

flight_data <- sample_n(
  flight_data_all,
  10000
)
```

We can see that around 16% of the flights in this data set arrived more than 30 minutes late:

```
flight_data %>%
  count(arr_delay) %>%
  mutate(prop = n / sum(n))
```

13.1 Data overview

Before we start building up our recipe, let's take a quick look at a few specific variables that will be important for both preprocessing and modeling.

First, notice that the variable we created called `arr_delay` is a factor variable; it is important that our outcome variable for training a classification model (at least a logistic regression model) is numeric.

```
glimpse(flight_data)
```

Second, there are two variables that we don't want to use as predictors in our model, but that we would like to retain as identification variables that can be used to troubleshoot poorly predicted data points. These are flight, a numeric value, and time_hour, a date-time value.

Third, there are 79 flight destinations contained in dest and 14 distinct carriers.

```
flight_data %>%
  skimr::skim(dest, carrier)
```

Because we'll be using a logistic regression model in this tutorial, the variables `dest` and `carrier` will be converted to dummy variables.

However, some of these values do not occur very frequently and this could complicate our analysis. We'll discuss specific steps later in this tutorial that we can add to our recipe to address this issue before modeling.

Chapter 14

Data splitting

To get started, let's split this single dataset into two: a training set and a testing set. We'll keep most of the rows in the original dataset (subset chosen randomly) in the training set. The training data will be used to fit the model, and the testing set will be used to measure model performance.

To do this, we can use the `rsample` package (included in `tidymodels`) to create an object that contains the information on how to split the data, and then two more rsample functions to create data frames for the training and testing sets:

```
# Fix the random numbers by setting the seed
# This enables the analysis to be reproducible
# when random numbers are used
set.seed(555)

# Put 3/4 of the data into the training set
data_split <- initial_split(flight_data,
  prop = 3 / 4
)

# Create data frames for the two sets:
train_data <- training(data_split)
test_data <- testing(data_split)
```

Chapter 15

Create recipe and roles

To get started, let's create a recipe for a classification model. Before training the models, we can use a recipe to create a few new predictors and conduct some preprocessing required by the model.

The `recipe()` function has two arguments:

- *A formula.* Any variable on the left-hand side of the tilde (~) is considered the model outcome (here, `arr_delay`). On the right-hand side of the tilde are the predictors. Variables may be listed by name, or you can use the dot (.) to indicate all other variables as predictors.
- *The data.* A recipe is associated with the data set used to create the model. This will typically be the training set, so `data = train_data` here. Naming a data set doesn't actually change the data itself; it is only used to catalog the names of the variables and their types, like factors, integers, dates, etc.

We can also add roles to this recipe. We can use the `update_role()` function to let recipes know that `flight` and `time_hour` are variables with a custom role that we call "ID" (a role can have any character value). Whereas our formula included all variables in the training set other than `arr_delay` as predictors, this tells the recipe to keep these two variables but not use them as either outcomes or predictors.

```
flights_rec <-
  recipe(arr_delay ~ ., data = train_data) %>%
  update_role(flight,
    time_hour,
    new_role = "ID"
  )
```

This step of adding roles to a recipe is optional; the purpose of using it here is that those two variables can be retained in the data but not included in the model. This can be convenient when, after the model is fit, we want to investigate some poorly predicted value. These ID columns will be available and can be used to try to understand what went wrong.

To get the current set of variables and roles, use the summary() function:

```
summary(flights_rec)
```

15.1 Create features

Now we can start adding steps onto our recipe using the pipe operator.

15.1.1 Date

Perhaps it is reasonable for the `date` of the flight to have an effect on the likelihood of a late arrival. A little bit of feature engineering might go a long way to improving our model. How should the date be encoded into the model? The date column has an R date object so including that column "as is" will mean that the model will convert it to a numeric format equal to the number of days after a reference date:

```
flight_data %>%
  distinct(date) %>%
  mutate(numeric_date = as.numeric(date))
```

It's possible that the numeric date variable is a good option for modeling. However, it might be better to add model terms derived from the date that have a better potential to be important to the model. For example, we could derive the following meaningful features from the single date variable:

- the day of the week,
- the month, and
- whether or not the date corresponds to a holiday.

Let's do all three of these by adding steps to our recipe:

```
flights_rec <-
  recipe(arr_delay ~ .,
    data = train_data
  ) %>%
```

```
update_role(flight,
  time_hour,
  new_role = "ID"
) %>%
step_date(date,
  features = c("dow", "month")
) %>%
step_holiday(date,
  holidays = timeDate::listHolidays("US")
) %>%
step_rm(date)
```

What do each of these steps do?

- With `step_date()`, we created two new factor columns with the appropriate day of the week (dow) and the month.
- With `step_holiday()`, we created a binary variable indicating whether the current date is a holiday or not. The argument value of `timeDate::listHolidays("US")` uses the timeDate package to list the 17 standard US holidays.
- With `step_rm()`, we remove the original date variable since we no longer want it in the model.

Next, we'll turn our attention to the variable types of our predictors. Because we plan to train a classifiaction model, we know that predictors will ultimately need to be numeric, as opposed to factor variables. In other words, there may be a difference in how we store our data (in factors inside a data frame), and how the underlying equations require them (a purely numeric matrix).

15.1.2 Dummy variables

For factors like `dest` and `origin`, standard practice is to convert them into *dummy* or indicator variables to make them numeric. These are binary values for each level of the factor. For example, our origin variable has values of "EWR", "JFK", and "LGA". The standard dummy variable encoding, shown below, will create two numeric columns of the data that are 1 when the originating airport is "JFK" or "LGA" and zero otherwise, respectively.

ORIGIN	ORIGIN_JFK	ORIGIN_LGA
EWR	0	0
JFK	1	0
LGA	0	1

But, unlike the standard model formula methods in R, a recipe does not automatically create these dummy variables for you; you'll need to tell your recipe to add this step. This is for two reasons. First, many models do not require numeric predictors, so dummy variables may not always be preferred. Second, recipes can also be used for purposes outside of modeling, where non-dummy versions of the variables may work better. For example, you may want to make a table or a plot with a variable as a single factor. For those reasons, you need to explicitly tell recipes to create dummy variables using step_dummy():

```
flights_rec <-
  recipe(arr_delay ~ .,
    data = train_data
  ) %>%
  update_role(flight,
    time_hour,
    new_role = "ID"
  ) %>%
  step_date(date,
    features = c("dow", "month")
  ) %>%
  step_holiday(date,
    holidays = timeDate::listHolidays("US")
  ) %>%
  step_rm(date) %>%
  step_dummy(all_nominal(), -all_outcomes())
```

Here, we did something different than before: instead of applying a step to an individual variable, we used selectors to apply this recipe step to several variables at once.

- The first selector, `all_nominal()`, selects all variables that are either factors or characters.
- The second selector, `-all_outcomes()` removes any outcome variables from this recipe step.

With these two selectors together, our recipe step above translates to:

> Create dummy variables for all of the factor or character columns unless they are outcomes.

At this stage in the recipe, this step selects the `origin`, `dest`, and `carrier` variables. It also includes two new variables, `date_dow` and `date_month`, that were created by the earlier `step_date()`.

More generally, the recipe selectors mean that you don't always have to apply steps to individual variables one at a time. Since a recipe knows the variable type and role of each column, they can also be selected (or dropped) using this information.

15.1.3 Zero variance

Note that since `carrier` and `dest` have some infrequently occurring values, it is possible that dummy variables might be created for values that don't exist in the training set. For example, there could be destinations that are only in the test set. The function `anti_join()` returns all rows from x (test_data) where there are not matching values in y (train_data), keeping just columns from x.:

```
test_data %>%
  distinct(dest) %>%
  anti_join(train_data)
```

When the recipe is applied to the training set, a column could contain only zeros. This is a "*zero-variance predictor*" that has no information within the column. While some R functions will not produce an error for such predictors, it usually causes warnings and other issues. `step_zv()` will remove columns from the data when the training set data have a single value, so it is added to the recipe after `step_dummy()`:

```
flights_rec <-
  recipe(arr_delay ~ .,
    data = train_data
  ) %>%
  update_role(flight,
    time_hour,
    new_role = "ID"
  ) %>%
  step_date(date,
    features = c("dow", "month")
  ) %>%
  step_holiday(date,
    holidays = timeDate::listHolidays("US")
  ) %>%
  step_rm(date) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())
```

15.1.4 Correlations

As a final step, we remove predictor variables that have large absolute correlations with other variables

```
flights_rec <-
  recipe(arr_delay ~ .,
    data = train_data
  ) %>%
  update_role(flight,
    time_hour,
    new_role = "ID"
  ) %>%
  step_date(date,
    features = c("dow", "month")
  ) %>%
  step_holiday(date,
    holidays = timeDate::listHolidays("US")
  ) %>%
  step_rm(date) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors()) %>%
  step_corr(all_predictors())
```

Now we've created a specification of what should be done with the data.

Chapter 16

Model building

Let's use some classification algorithms to model the flight data. We start by building some models using the `parsnip` package:

- List of algorithms

16.1 Logistic regression

```
lr_mod <-
  logistic_reg() %>%
  set_engine("glm")
```

16.2 Decision tree

```
dt_mod <-
  decision_tree() %>%
  set_engine("C5.0") %>%
  set_mode("classification")
```

16.3 Random forest

```
rf_mod <-
  rand_forest() %>%
  set_engine("ranger") %>%
  set_mode("classification")
```

16.4 Boosted tree (XGBoost)

```
# install.packages("xgboost")

xgb_mod <-
  boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("classification")
```

Chapter 17

Recipe and Models

We will want to use our recipe across several steps as we train and test our models. We will:

1. Process the recipe using the training set: This involves any estimation or calculations based on the training set. For our recipe, the training set will be used to determine which predictors should be converted to dummy variables and which predictors will have zero-variance in the training set, and should be slated for removal.
2. Apply the recipe to the training set: We create the final predictor set on the training set.
3. Apply the recipe to the test set: We create the final predictor set on the test set. Nothing is recomputed and no information from the test set is used here; the dummy variable and zero-variance results from the training set are applied to the test set.

To simplify this process, we can use a *model workflow*, which pairs a model and recipe together. This is a straightforward approach because different recipes are often needed for different models, so when a model and recipe are bundled, it becomes easier to train and test workflows. We'll use the `workflow` package from tidymodels to bundle our parsnip model (lr_mod etc.) with our recipe (flights_rec).

17.1 Fit models with workflows

To combine the data preparation with the model building, we use the package workflows. A workflow is an object that can bundle together your pre-processing, modeling, and post-processing requests.

17.1.1 Logistic regression

```
flights_wflow_lr_mod <-
  workflow() %>%
  add_model(lr_mod) %>%
  add_recipe(flights_rec)

flights_wflow_lr_mod
```

17.1.2 Decision tree

```
flights_wflow_dt_mod <-
  workflow() %>%
  add_model(dt_mod) %>%
  add_recipe(flights_rec)

flights_wflow_dt_mod
```

17.1.3 Random forest

```
flights_wflow_rf_mod <-
  workflow() %>%
  add_model(rf_mod) %>%
  add_recipe(flights_rec)

flights_wflow_rf_mod
```

17.1.4 XGBoost

```
flights_wflow_xgb_mod <-
  workflow() %>%
  add_model(xgb_mod) %>%
  add_recipe(flights_rec)

flights_wflow_xgb_mod
```

17.2 Train models

Now, there is a single function that can be used to prepare the recipe and train the models from the resulting predictors.

17.2.1 Logistic regression

```
flights_fit_lr_mode <-
  flights_wflow_lr_mod %>%
  fit(data = train_data)

flights_fit_lr_mode
```

17.2.2 Decision tree

```
flights_fit_dt_mode <-
  flights_wflow_dt_mod %>%
  fit(data = train_data)

flights_fit_dt_mode
```

17.2.3 Random forest

```
flights_fit_rf_mode <-
  flights_wflow_rf_mod %>%
  fit(data = train_data)

flights_fit_rf_mode
```

17.2.4 XG Boost

```
flights_fit_xgb_mode <-
  flights_wflow_xgb_mod %>%
  fit(data = train_data)

flights_fit_xgb_mode
```

17.3 Model recipe objects

The objects above have the finalized recipe and fitted model objects inside. You may want to extract the model or recipe objects from the workflow. To do this, you can use the helper functions `pull_workflow_fit()` and `pull_workflow_prepped_recipe()`.

For example, here we pull the fitted model object then use the `broom::tidy()` function to get a tidy tibble of the Logisitc Regression model coefficients.

17.3.1 Logistic regression

```
flights_fit_lr_mode %>%
  pull_workflow_fit() %>%
  tidy()
```

17.3.2 Decision tree

```
flights_fit_xgb_mode %>%
  pull_workflow_prepped_recipe()
```

17.3.3 Random forest

```
flights_fit_rf_mode %>%
  pull_workflow_prepped_recipe()
```

17.3.4 XG Boost

```
flights_fit_xgb_mode %>%
  pull_workflow_prepped_recipe()
```

17.4 Summary

Our goal was to predict whether a plane arrives more than 30 minutes late. We have just:

1. Built the model (`lr_mod` etc.),
2. Created a preprocessing recipe (`flights_rec`),
3. Bundled the model and recipe (`flights_wflow`), and
4. Trained our workflow using a single call to `fit()`.

The next step is to use the trained workflow (`flights_fit`) to predict with the unseen test data, which we will do with a single call to predict().

Chapter 18

Prediction

The `predict()` method applies the recipe to the new data, then passes them to the fitted model. Let's use the logistic regression model as an example for the next steps.

18.1 Logistic regression

```
predict(
  flights_fit_lr_mode,
  test_data
)
```

Because our outcome variable here is a factor, the output from `predict()` returns the predicted class: late versus on_time. But, let's say we want the predicted class *probabilities* for each flight instead. To return those, we can specify `type = "prob"` when we use `predict()`. We'll also bind the output with some variables from the test data and save them together:

```
flights_pred_lr_mod <-
  predict(flights_fit_lr_mode,
    test_data,
    type = "prob"
  ) %>%
  bind_cols(test_data %>%
    select(
      arr_delay,
      time_hour,
```

```
      flight
    ))
```

The data look like:

```
head(flights_pred_lr_mod)
```

Now that we have a tibble with our predicted class probabilities, how will we evaluate the performance of our workflow? We would like to calculate a metric that tells how well our model predicted late arrivals, compared to the true status of our outcome variable, `arr_delay`.

18.1.1 ROC curve

Let's use the area under the ROC curve as our metric, computed using `roc_curve()` and `roc_auc()` from the `yardstick` package.

To generate a ROC curve, we need the predicted class probabilities for `late` and `on_time`, which we just calculated in the code chunk above. We can create the ROC curve with these values, using `roc_curve()` and then piping to the `autoplot()` method:

```
flights_pred_lr_mod %>%
  roc_curve(
    truth = arr_delay,
    .pred_late
  ) %>%
  autoplot()
```

18.1.2 AUC

Similarly, `roc_auc()` estimates the area under the curve:

```
flights_pred_lr_mod %>%
  roc_auc(truth = arr_delay, .pred_late)
```

18.1.3 Accuracy

We 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).

```
flights_fit_lr_mode %>%
  predict(test_data) %>%
  bind_cols(test_data) %>%
  metrics(
    truth = arr_delay,
    estimate = .pred_class
  )
```

18.1.4 Recall

```
flights_fit_lr_mode %>%
  predict(test_data) %>%
  bind_cols(test_data) %>%
  recall(
    truth = arr_delay,
    estimate = .pred_class
  )
```

18.1.5 Precision

```
flights_fit_lr_mode %>%
  predict(test_data) %>%
  bind_cols(test_data) %>%
  precision(
    truth = arr_delay,
    estimate = .pred_class
  )
```

Bibliography

Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O'Reilly Media.