Problem Set 1: Linear Methods

author 1 (matriculation number) author 2 (matriculation number)
author 3 (matriculation number)

2021-10-25 till 2021-11-14

- You may answer this problem set in groups of up to three people. Please state your names and matriculation numbers in the `YAML` header before submission (under the `author` field).
- Please hand in your answers via GitHub until 2021-11-14, 23:59 pm. When you have pushed your final results to GitHub, create a GitHub issue with the title "Hand-in" and mark Marius (GitHub username: *puchalla*) and Simon (GitHub username: *simonschoe*) as assignees or reference us in the issue description. Please ensure that all code chunks can be executed without error before pushing your last commit!
- You are supposed to Git commit your code changes regularly! In particular, each group member is required to contribute commits to the group's final submission to indicate equal participation in the group assignment. In case we observe a highly imbalanced distribution of contributions (indicated by the commits), individual seminar participants may be penalized.
- Make sure to answer **all** questions in this Notebook! You may use plain text, R code and LaTeX equations to do so. You must always provide at least one sentence per answer and include the code chunks you used in order to reach a solution. Please comment your code in a transparent manner.
- For several exercises throughout the assignment you may find hints and code snippets that should support you in solving the exercises. You may rely on these to answer the questions, but you don't have to. Any answer that solves the exercise is acceptable.

Setup

Before starting, leverage the `pacman` package by executing the code chunk below. `pacman` is a convenient helper for installing and loading packages at the start of your project. It checks whether a package has already been installed and loads the package if available. Otherwise it installs the package automatically. Use `pacman::p_load()` to install and load the following packages:

- `tidyverse` (meta-package to load `dplyr`, `ggplot2`, and co.),
- `tidymodels` (meta-package to load `rsample`, `parsnip`, `tune`, and co.),
- `GLMsData` (contains the dataset for *Task 1*),
- `ISLR` (contains the dataset for *Task 3*),
- `boot` (functions for bootstrapping),
- `MASS` (functions for discriminant analysis),
- `class`, `kknn` (functions for k-NN),
- `glmnet` (functions for regularized regression),
- `leaps` (functions for stepwise subset selection),
- `discrim` (helper functions for LDA and Naive Bayes using `tidymodels`),

```r
# check if pacman is installed (install if evaluates to FALSE)
if (!require("pacman")) install.packages("pacman")

# load (or install if pacman cannot find an existing installation) the relevant packages
pacman::p_load(
  tidyverse, tidymodels, GLMsData, ISLR,
  boot, MASS, kknn, class, glmnet, leaps
)
pacman::p_load_gh("tidymodels/discrim")
```

In case you need any further help with the above mentioned packages and included functions, use the `help()` function build into RStudio (e.g., by running `help(rsample)` or by using the *Help* pane in the RStudio IDE interface).

Task 1: Multiple Linear Regression

This task deals with modeling lung capacity. You will use the dataset `lungcap` (as part of the `GLMsData` package) which provides information on body variables and smoking habits for a sample of 654 youths, aged between 3 and 19, in the area of East Boston during the middle to late 1970s. First, use the subsequent code to load the data and convert the `Smoke` variable from `int` to `fct`.

```
data("lungcap", package = "GLMsData")

lungcap <- lungcap %>%
  tibble::as_tibble() %>%
  dplyr::mutate(across(Smoke, as.factor))

lungcap %>%
  glimpse
```

```
> Rows: 654
> Columns: 5
> $ Age    <int> 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ~
> $ FEV    <dbl> 1.072, 0.839, 1.102, 1.389, 1.577, 1.418, 1.569, 1.196, 1.400, ~
> $ Ht     <dbl> 46.0, 48.0, 48.0, 48.0, 49.0, 49.0, 50.0, 46.5, 49.0, 49.0, 50.~
> $ Gender <fct> F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F, ~
> $ Smoke  <fct> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
```

You will predict forced expiratory volume (`FEV`), a measure of lung capacity. For each person in the dataset you have measurements of the following variables:

- `FEV`: forced expiratory volume in liters, a measure of lung capacity (type `dbl`),
- `Age`: the age of the subject in completed years (type `int`),
- `Ht`: the height in inches (type `dbl`),
- `Gender`: the gender of the subjects, a factor (`fct`) with levels `F` (female) and `M` (male),
- `Smoke`: the smoking status of the subject, a factor (`fct`) with levels `0` (non-smoker) and `1` (smoker).

For better interpretability, transform the height from inches to cm (one inch corresponds to 2.54cm). Then fit a multiple linear regression model to the data with `log(FEV)` as response and the other variables as predictors.

```
lungcap <- lungcap %>%
  dplyr::mutate(across(Ht, ~ . * 2.54)) %>%
  dplyr::rename(Htcm = Ht)

fit_lc <- lungcap %>%
  lm(log(FEV) ~ Age + Htcm + Gender + Smoke, data = .)

fit_lc %>%
  summary()
```

```
>
> Call:
> lm(formula = log(FEV) ~ Age + Htcm + Gender + Smoke, data = .)
>
> Residuals:
```

```
>      Min       1Q   Median       3Q      Max
> -0.63278 -0.08657  0.01146  0.09540  0.40701
>
> Coefficients:
>              Estimate Std. Error t value Pr(>|t|)
> (Intercept) -1.943998   0.078639 -24.721  < 2e-16 ***
> Age          0.023387   0.003348   6.984  7.1e-12 ***
> Htcm         0.016849   0.000661  25.489  < 2e-16 ***
> GenderM      0.029319   0.011719   2.502   0.0126 *
> Smoke1      -0.046067   0.020910  -2.203   0.0279 *
> ---
> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
> Residual standard error: 0.1455 on 649 degrees of freedom
> Multiple R-squared:  0.8106,  Adjusted R-squared:  0.8095
> F-statistic: 694.6 on 4 and 649 DF,  p-value: < 2.2e-16
```

Alternatively, you may also use the different `tidymodels` packages to fit a linear model:

```
# specify preprocessing recipe (incl. model formula)
rec_lm_lc <- lungcap %>%
  recipes::recipe(formula = FEV ~ Age + Htcm + Gender + Smoke) %>%
  recipes::step_log(FEV)

# specify model
spec_lm_lc <-
  parsnip::linear_reg(mode = "regression", engine = "lm")

# specify modeling workflow
wf_lm_lc <- workflows::workflow() %>%
  workflows::add_model(spec_lm_lc) %>%
  workflows::add_recipe(rec_lm_lc)

# fit model
fit_lc <- wf_lm_lc %>%
  parsnip::fit(lungcap)

fit_lc %>%
  workflows::extract_fit_engine() %>%
  summary
```

```
>
> Call:
> stats::lm(formula = ..y ~ ., data = data)
>
> Residuals:
>      Min       1Q   Median       3Q      Max
> -0.63278 -0.08657  0.01146  0.09540  0.40701
>
> Coefficients:
>              Estimate Std. Error t value Pr(>|t|)
> (Intercept) -1.943998   0.078639 -24.721  < 2e-16 ***
> Age          0.023387   0.003348   6.984  7.1e-12 ***
```

```
> Htcm          0.016849   0.000661  25.489  < 2e-16 ***
> GenderM       0.029319   0.011719   2.502   0.0126 *
> Smoke1       -0.046067   0.020910  -2.203   0.0279 *
> ---
> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
> Residual standard error: 0.1455 on 649 degrees of freedom
> Multiple R-squared:  0.8106,  Adjusted R-squared:  0.8095
> F-statistic: 694.6 on 4 and 649 DF,  p-value: < 2.2e-16
```

Even though, this approach appears slightly more laborious compared to simply using `lm()`, it becomes more efficient when your goal is to compare multiple different models in a streamlined manner. For now, however, we are only interested in understanding the inner workings of the linear regression model.

Task 1.1

Write down the generic linear regression equation as well as the specific equation for the trained `fit_lc` model including the point estimates for the coefficients.

Hint: You may use the `equatiomatic` package to generate valid LaTeX code for your fitted `lung_model`. Refer to Xie/Dervieux/Riederer (2021) as well as the official GitHub page to learn more about its various features. Note that it must be installed first by adding it to the `pacman::p_load()` command above.

Task 1.2

Why is `log(FEV)` used as the response instead of `FEV`? To answer this question, plot `FEV` and `log(FEV)` using the `geom_density()` function as part of your `ggplot` pipeline and compare the mean and median of both, `FEV` and `log(FEV)`. What shape do you observe?

Task 1.3

Explain with your own words and numerical examples what the following statistics in the `summary(fit_lc)` output mean.

i. *`Estimate`*
   Discuss one continuous and one dummy predictor on the log as well as on the original scale of `FEV` (i.e. after reversing the log-transformation).
ii. *`Std. Error`*
   Discuss the statistic based on the `Age` and `Htmc` predictor. Also explain how a 95% confidence interval can be constructed.
iii. *`Residual standard error`*
iv. *`F-statistic`*

Task 1.4

What is the proportion of variability explained by the fitted `fit_lc`?

Task 1.5

The `summary()` output also reports the two statistics `t value` and `Pr(>|t|)` for each coefficient. Briefly explain the hypothesis test that is underlying the t-statistic.

Task 1.6

Consider a 14-year-old male. He is 175cm tall and does not smoke. What is your best guess for his `log(FEV)`? Construct a 95% prediction interval for his forced expiratory volume `FEV` (remember to inverse the logarithm). Please comment on whether you find this prediction interval useful.

Task 1.7

Redo the multiple linear regression, but add an interaction term between `Age` and `Smoke`. What is the meaning of the point estimate (i.e. coefficient) for the interaction term? Is the interaction term statistically significant? What is the effect of the inclusion of the interaction term on the statistical significance of `Smoke` and `Age`?

Hint: If you try solve this task the `tidymodels` way, you may refer to `recipes::step_interact()` in your pre-processing pipeline. Besides, you may have to apply `recipes::step_dummy()` before in order to convert the `Smoke` factor into a numeric dummy variable.

Task 2: Classification (Logit/LDA/k-NN)

In this task, you will work with data from the four major tennis tournaments in 2013 (both men and women), published in the UCI Machine Learning Repository. Note that the `Result` column is formatted as a factor.

```
tennis <- readr::read_csv(file = "./tennisdata.csv") %>%
  dplyr::mutate(across(Result, as.factor))

tennis
```

```
> # A tibble: 943 x 42
>    Player1  Player2 Round Result  FNL1  FNL2 FSP.1 FSW.1 SSP.1 SSW.1 ACE.1 DBF.1
>    <chr>    <chr>   <dbl> <fct>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
>  1 Lukas L~ Novak ~     1 0          0     3    61    35    39    18     5     1
>  2 Leonard~ Albert~     1 1          3     0    61    31    39    13    13     1
>  3 Marcos ~ Denis ~     1 0          0     3    52    53    48    20     8     4
>  4 Dmitry ~ Michae~     1 1          3     0    53    39    47    24     8     6
>  5 Juan Mo~ Ernest~     1 0          1     3    76    63    24    12     0     4
>  6 Santiag~ Sam Qu~     1 0          1     3    65    51    35    22     9     3
>  7 Dudi Se~ Jarkko~     1 0          2     3    68    73    32    24     5     3
>  8 Fabio F~ Alex B~     1 1          2     0    47    18    53    15     3     4
>  9 David G~ Richar~     1 0          0     3    64    26    36    12     3    NA
> 10 Nikolay~ Lukasz~     1 1          3     2    77    76    23    11     6     4
> # ... with 933 more rows, and 30 more variables: WNR.1 <dbl>, UFE.1 <dbl>,
> #   BPC.1 <dbl>, BPW.1 <dbl>, NPA.1 <dbl>, NPW.1 <dbl>, TPW.1 <dbl>,
> #   ST1.1 <dbl>, ST2.1 <dbl>, ST3.1 <dbl>, ST4.1 <dbl>, ST5.1 <dbl>,
> #   FSP.2 <dbl>, FSW.2 <dbl>, SSP.2 <dbl>, SSW.2 <dbl>, ACE.2 <dbl>,
> #   DBF.2 <dbl>, WNR.2 <dbl>, UFE.2 <dbl>, BPC.2 <dbl>, BPW.2 <dbl>,
> #   NPA.2 <dbl>, NPW.2 <dbl>, TPW.2 <dbl>, ST1.2 <dbl>, ST2.2 <dbl>,
> #   ST3.2 <dbl>, ST4.2 <dbl>, ST5.2 <dbl>
```

Our goal is to predict the outcome of a match (success or failure of player 1) using the quality statistics `x1` from player 1 and `x2` from player 2.

Excursus: These quality statistics are calculated for each match as follows. Each row in the original dataset contains information about one match. The variables that end with `.1` relate to player 1, while `.2` concerns player 2. In tennis, you have two attempts at the serve. You lose the point if you fail both. The number of these double faults committed is given in the variable `DBF`, while the variable `ACE` is the number of times your opponent fails to return your serve. Similarly, unforced errors (mistakes that are supposedly not forced by good shots of your opponent) are called `UFE`. A skilled player will score many aces while committing few double faults and unforced errors.

Each match involves two players, and there are no draws in tennis. The result of a match is either that

- player 1 wins, coded as 1 (so, success of player 1), or that
- player 2 wins, coded as 0 (so, failure of player 1).

For the two players, player 1 and player 2, the quality of player $c = 1, 2$ can be summarized as $x_c = ACE_c - UFE_c - DBF_c$.

The following code chunk computes the quality scores and stores them in a `tibble` together with the result (y) of each match (explicit missing values are removed via `tidyr::drop_na()`).

```
tennis <- tennis %>%
  dplyr::mutate(
    x1 = ACE.1 - UFE.1 - DBF.1,
    x2 = ACE.2 - UFE.2 - DBF.2
  ) %>%
  dplyr::select(Result, x1, x2) %>%
  dplyr::rename(y = Result) %>%
  tidyr::drop_na()

tennis %>%
  glimpse
```

```
> Rows: 787
> Columns: 3
> $ y  <fct> 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1~
> $ x1 <dbl> -25, 11, -46, -4, -39, -35, -48, -32, -2, -58, -25, -13, -54, 0, -8~
> $ x2 <dbl> -20, -7, -33, -15, -73, -25, -76, -57, -16, -34, -97, -15, -55, -4,~
```

Note: A function named ***`select()`*** *is part of the* ***`MASS`*** *as well as the* ***`dplyr`*** *package. Since you load the* ***`tidyverse`*** *prior to the* ***`MASS`*** *package, the* ***`select()`*** *function in* ***`tidyverse`*** *is overridden (masked) by the* ***`MASS`*** *package. In those cases, you may specify the namespace of the function prior to the function call to resolve ambiguity (i.e. write* ***`dplyr::select()`****).*

Next, perform a random train-test split using the `rsample` package. First, split the dataset into two equal parts using `rsample::initial_split()`. Second, extract the training and test set from the split object using `rsample::training()` and `rsample::testing()`.

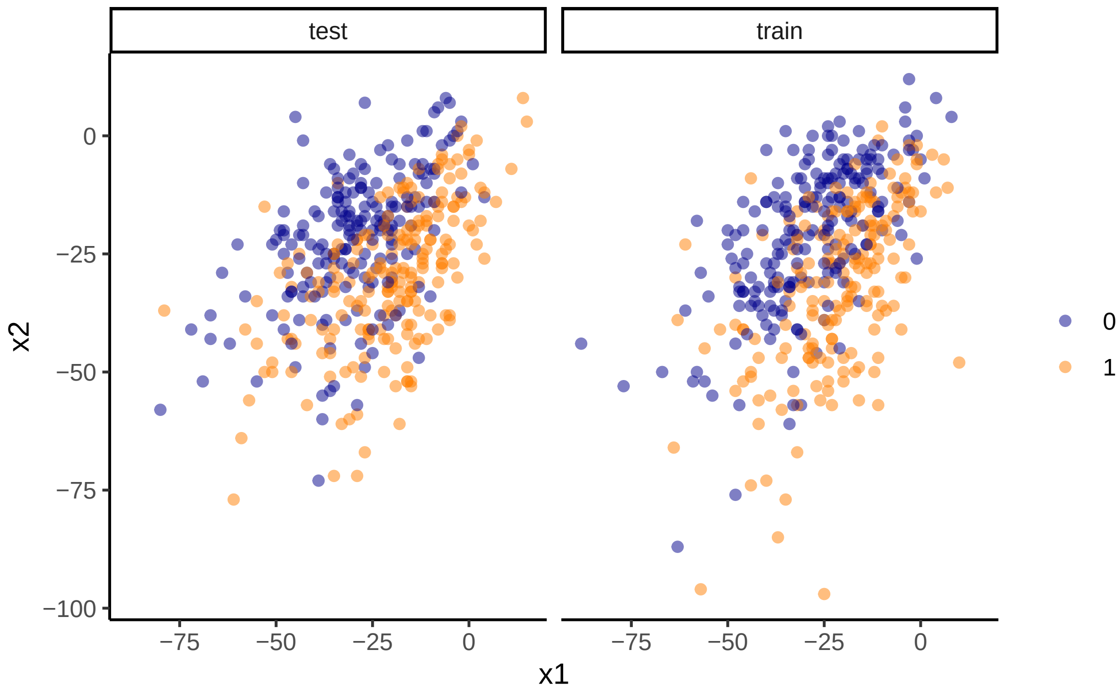
```
set.seed(2021)

tennis_split <- tennis %>%
  rsample::initial_split(prop = 0.5)

train_set_tn <- rsample::training(tennis_split)
test_set_tn <- rsample::testing(tennis_split)
```

In the scatter plot below, matches won by player one (two) are displayed in dark orange (blue). Gladly, we do not observe any systematic differences between training and test set at first glance. This important for training a model that generalizes well to unseen test data.

```
dplyr::bind_rows(
  training(tennis_split) %>% mutate(id = "train"),
  testing(tennis_split) %>% mutate(id = "test")
) %>%
  ggplot2::ggplot() +
  geom_point(aes(x1, x2, color = y), alpha = 0.5) +
  scale_color_manual(values = c("blue4", "darkorange1")) +
  facet_wrap(~ id) +
  theme_classic() +
  theme(legend.title = element_blank(), legend.position = "right")
```



You will now apply different classification methods to predict the match outcome and validate some of your results via k-fold cross-validation (CV).

Task 2.1

Get a general overview of the data frame `tennis`. How many observations are there? What are the median values of `x1` and `x2`? Display the matrix of pairwise scatterplots and briefly comment on the relationship between `y` and `x1` respectively `y` and `x2`.

Hint 1: If you compute the median the `tidyverse`-way, remember that `summarise(across(...))` can operate on multiple variables by feeding `across` a vector of column names (e.g., `across(c(var1, var2), ...)`.

Hint 2: Depending on the employed plot matrix function, `R` is eventually encoding your categorical variable `y` as a numerical variable. Hence, your `y` variable is mapped onto the $[1;2]$-interval, since `R` assigns the first factor level (here 0) a value of 1 and the second factor level (here 1) a value of 2.

Task 2.2

Train a logistic regression on the entire dataset and print the results using `summary()`. In addition, address the following questions:

i. Are the estimated coefficients for `x1` and `x2` statistically significant?
ii. Just by looking at the two coefficients: What is the effect on `y` if both `x1` and `x2` increase by one?
iii. What is the effect on the odds of success for player 1 if `x1` increases by one?
iv. In the first match, player 1 has a quality of -25 and player 2's quality is -20. What is the value for the odds-ratio for the given prediction and how can it be interpreted? What is the predicted probability that player 1 wins this match? Did player 1 actually win the the match?

Hint: You may implement the logistic model using base `R` `stats::glm()` function or via the `tidymodels` wrapper `parsnip::logistic_reg()`.

Task 2.3

The *receiver operating characteristic* (*ROC*)-curve is a visual tool that enables you to compare the performance of different classifiers. The trajectory of the ROC-curve is derived by systematically varying the probability threshold which determines if a predicted probability is assigned to class 0 (loss player 1) or class 1 (success player 1). Use the predictions of the logistic regression model fitted in the previous task to construct a ROC-curve. In addition, answer the following questions:

i. What is the model's predictive accuracy (i.e. proportion of correctly predicted data points)?
ii. What is its *area under the receiver operator curve* (*ROC-AUC*)?
iii. What is its *sensitivity* and *specificity* assuming a probability cutoff of 0.5? How can these two measures be interpreted?
iv. Taking all of the above into account, how would you interpret the performance of the model in your own words?

Hint: The `yardstick` package provides some convenient functions that may help you solve this task. In case you decide to use the `yardstick::roc_curve()` function, consider carefully the `event_level` argument of the function to receive the desired output.

Task 2.4

Train a logistic regression on the training data and compute the confusion matrix for the test set. What is the *accuracy*, *sensitivity* and *specificity*? How does the model performance compare to the previous task where the model is fitted on the entire dataset?

Task 2.6

Train a linear discriminant analysis (LDA) using `MASS::lda()` or `tidymodel`'s `discrim::discrim_linear()` on the training data and compute the confusion matrix for the test set. What is the *accuracy*, *sensitivity* and *specificity*? How does the model performance compare to the logit model in the previous task?

Note: In the lecture you have learned that LDA is preferable if the classes are well-separated, the predictors follow a normal distribution and n is relatively small. Even though these requirements may not be fulfilled here, you may still use the method to compare its performance with other methods.

Task 2.7

Suppose you know about both players' quality in a specific match in the test set but you do not know the outcome `y`. According to LDA, how many match results (from the test data) can you predict with a probability larger than 80%? Put differently: In how many cases is the LDA more than 80% sure about the match outcome?

Task 2.8

Use the following code snippet+ to compute the misclassification error on the train and test set using k-Nearest-Neighbor (k-NN) for all $k = 1, 2, ..., 30$. The code leverages the `purrr::map_dfr()` function to apply the custom `knn_predict()` function to each element in `seq(1L, 30L, 1L)` (i.e. the parameters 1 to 30) which yields a row-wise `tibble`.

Note: Try to run the code chunk piece by piece in order to grasp the logic underlying the computations.

1. Define custom predict function:

```r
knn_predict <- function(k, new_data) {

  # specify preprocessing recipe (incl. model formula)
  rec_knn <-
    recipes::recipe(formula = y ~ x1 + x2, data = tennis)

  # specify model
  spec_knn <-
    parsnip::nearest_neighbor(mode = "classification", engine = "kknn", neighbors = k)

  # specify modeling workflow
  wf_knn <- workflows::workflow() %>%
    workflows::add_model(spec_knn) %>%
    workflows::add_recipe(rec_knn)

  # fit model on the training set
  knn_fit <- wf_knn %>%
    parsnip::fit(train_set_tn)

  # generate prediction for `new_data`
  preds <- predict(knn_fit, new_data = new_data) %>%
    dplyr::mutate(neighbors = k)

  return(preds)

}
```

2. Test custom predict function for $k = 5$:

```r
knn_predict(k = 5, new_data = train_set_tn) %>% glimpse
```

```
> Rows: 393
> Columns: 2
> $ .pred_class <fct> 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0~
> $ neighbors   <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5~
```

3. Compute misclassification error on the training set:

```r
train_set_errors <-
  # iterate over .x and repeatedly apply `knn_predict()` to train set
  map_dfr(
    .x = seq(1L, 30L, 1L),
    .f = ~ knn_predict(.x, train_set_tn)
  ) %>%
  # check for each k if the predictions are unequal to the true class
  dplyr::group_by(neighbors) %>%
  dplyr::mutate(pred_error = (.pred_class != train_set_tn$y)) %>%
  # compute the misclassification error for each k
  dplyr::summarize(across(pred_error, mean), .groups = "drop")

train_set_errors %>% glimpse
```

```
> Rows: 30
> Columns: 2
> $ neighbors  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ~
> $ pred_error <dbl> 0.01272265, 0.01272265, 0.01272265, 0.14249364, 0.14503817,~
```

4. Compute misclassification error on the test set:

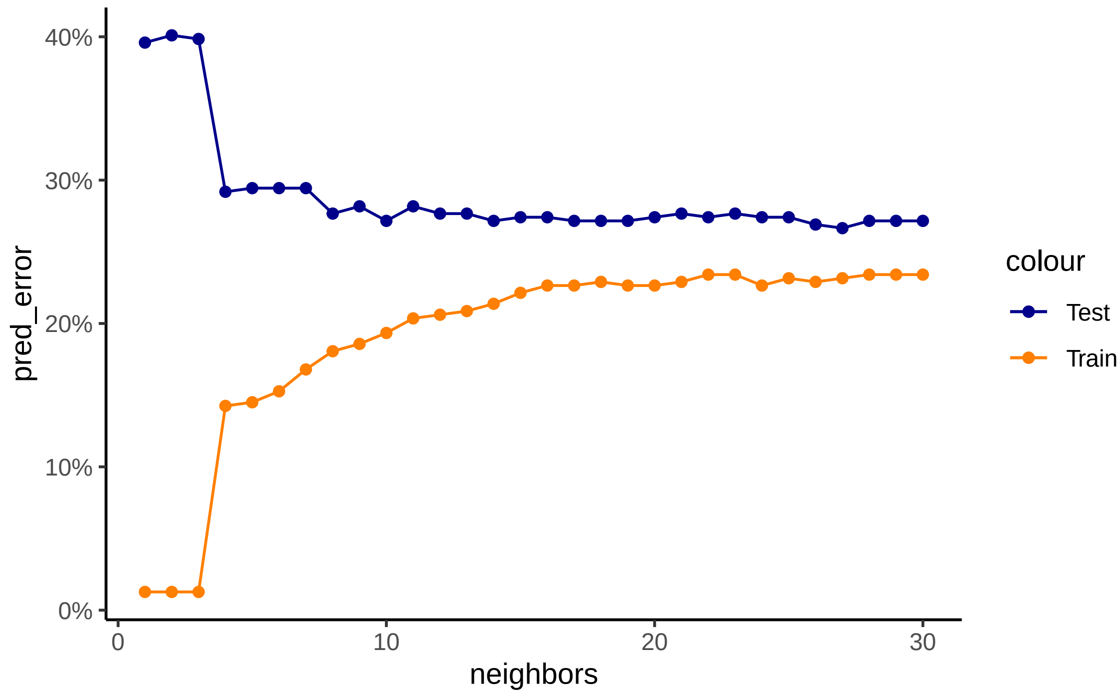
```
test_set_errors <-
  # iterate over .x and repeatedly apply `knn_predict()` to test set
  map_dfr(
    .x = seq(1L, 30L, 1L),
    .f = ~ knn_predict(.x, test_set_tn) %>% mutate(k = .x)
  ) %>%
  # check for each k if the predictions are unequal to the true class
  dplyr::group_by(neighbors) %>%
  dplyr::mutate(pred_error = (.pred_class != test_set_tn$y)) %>%
  # compute the misclassification error for each k
  dplyr::summarize(across(pred_error, mean), .groups = "drop")

test_set_errors %>% glimpse
```

```
> Rows: 30
> Columns: 2
> $ neighbors  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ~
> $ pred_error <dbl> 0.3959391, 0.4010152, 0.3984772, 0.2918782, 0.2944162, 0.29~
```

5. Plot misclassification error against the number of neighbors k:

```
ggplot() +
  geom_point(aes(x = neighbors, y = pred_error, color = "Train"), train_set_errors) +
  geom_line(aes(x = neighbors, y = pred_error, color = "Train"), train_set_errors) +
  geom_point(aes(x = neighbors, y = pred_error, color = "Test"), test_set_errors,) +
  geom_line(aes(x = neighbors, y = pred_error, color = "Test"), test_set_errors) +
  scale_color_manual(values = c("blue4", "darkorange1")) +
  scale_y_continuous(labels = scales::label_percent(1.)) +
  theme_classic()
```



Briefly describe in your own words what the two curves depict. What can you say about the bias and variance of the model when `k` increases? What is the optimal `k`? Motivate your answer.

Task 2.9

In order to obtain a truly unbiased estimate for a model's performance, the optimal `k` is regularly identified via CV. This approach is also referred to as *hyperparameter tuning* with `k` being the hyperparameter of interest. Hyperparameters are subjectively chosen by the user of a machine learning model and, hence, to be distinguished from those model parameters that are derived from the data itself (e.g., coefficients). When conducting hyperparameter tuning, the data is generally split into three distinct sets:[1]

- the *training set*, used for fitting the model (i.e. estimating the model coefficients),
- the *validation set*, used for finding the optimal hyperparameter (aka model configuration), and
- the *test set*, used for computing a robust estimate of the misclassification error on unseen data.

Consider the code below where the original training data is sub-divided into 5 disjunct folds using `vfold_cv()` from the `rsample` package.

```
set.seed(2021)

cv_tn <- train_set_tn %>%
  rsample::vfold_cv(v = 5, repeats = 1)

cv_tn
```

```
> #  5-fold cross-validation
> # A tibble: 5 x 2
```

[1] See also Kuhn/Silge (2021), ch. 10.2 for a visual illustration of this three-way data split.

```
>   splits           id
>   <list>           <chr>
> 1 <split [314/79]> Fold1
> 2 <split [314/79]> Fold2
> 3 <split [314/79]> Fold3
> 4 <split [315/78]> Fold4
> 5 <split [315/78]> Fold5
```

The output of `rsample::vfold_cv()` is a `tibble` with an `id` column as well as a list column which contains the cross-validation `splits`. Each split contains 80% of the original training set observations as resampled training data (which can be accessed via `rsample::training()`) and 20% as validation data (which can be accessed via `rsample::testing()`). Have a look at `Fold1`:

```
cv_tn %>%
  purrr::pluck("splits", 1)
```

```
> <Analysis/Assess/Total>
> <314/79/393>
```

Next, we recycle some of the code from task 2.8 and define the modeling workflow using the respective `tidymodels` packages. Note that the `neighbors` argument of the `nearest_neighbor` model is now set to `tune()` to indicate that this hyperparameter should be optimized for by trying out different values for `k`.

```
# specify model
spec_knn <-
  parsnip::nearest_neighbor(mode = "classification", engine = "kknn", neighbors = tune())

# specify preprocessing recipe (incl. model formula)
rec_knn <-
  recipes::recipe(formula = y ~ x1 + x2, data = tennis)

# specify modeling workflow
wf_knn <- workflows::workflow() %>%
  workflows::add_model(spec_knn) %>%
  workflows::add_recipe(rec_knn)

wf_knn
```

```
> == Workflow ====================================================================
> Preprocessor: Recipe
> Model: nearest_neighbor()
>
> -- Preprocessor ----------------------------------------------------------------
> 0 Recipe Steps
>
> -- Model -----------------------------------------------------------------------
> K-Nearest Neighbor Model Specification (classification)
>
> Main Arguments:
>   neighbors = tune()
>
> Computational engine: kknn
```

In order to optimize for `k`, we define a grid of values over which to iterate using `dials::grid_regular()`. Since `k` is the only relevant hyperparameter in this exercise, the grid is simply a sequence of values for `k`.

```
grid_knn <-
  dials::grid_regular(dials::neighbors(c(1, 30)), levels = 30)

grid_knn %>% glimpse
```

```
> Rows: 30
> Columns: 1
> $ neighbors <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1~
```

Finally, we have gathered all the components required to find the optimal value for `k` (i.e. a model specification, a pre-processing recipe, a set of cross-validated resamples and a parameter grid over which to optimize). Hyperparameter tuning in the `tidymodels` ecosystem is performed by the `tune::tune_grid()` function. It automatically computes the relevant error statistics (i.e. accuracy and ROC-AUC) for each `k` and over all resamples, amounting to 150 models in total.

```
fit_knn <-
  tune::tune_grid(
    wf_knn, cv_tn, grid = grid_knn,
    control = tune::control_grid(verbose = T)
  )

fit_knn
```

```
> # Tuning results
> # 5-fold cross-validation
> # A tibble: 5 x 4
>   splits           id    .metrics          .notes
>   <list>           <chr> <list>            <list>
> 1 <split [314/79]> Fold1 <tibble [60 x 5]> <tibble [0 x 1]>
> 2 <split [314/79]> Fold2 <tibble [60 x 5]> <tibble [0 x 1]>
> 3 <split [314/79]> Fold3 <tibble [60 x 5]> <tibble [0 x 1]>
> 4 <split [315/78]> Fold4 <tibble [60 x 5]> <tibble [0 x 1]>
> 5 <split [315/78]> Fold5 <tibble [60 x 5]> <tibble [0 x 1]>
```

The error statistics are contained in the `.metrics` columns of the `fit_knn` object and can be extracted using `purrr::pluck(fit_knn, ".metrics", 1)` (for the first fold), or by relying on one of the helper functions that the `tune` package provides.

```
tune::collect_metrics(fit_knn, summarize = FALSE)
```

```
> # A tibble: 300 x 6
>    id    neighbors .metric  .estimator .estimate .config
>    <chr>     <int> <chr>    <chr>          <dbl> <chr>
>  1 Fold1         1 accuracy binary         0.620 Preprocessor1_Model01
>  2 Fold1         1 roc_auc  binary         0.633 Preprocessor1_Model01
>  3 Fold2         1 accuracy binary         0.696 Preprocessor1_Model01
>  4 Fold2         1 roc_auc  binary         0.688 Preprocessor1_Model01
>  5 Fold3         1 accuracy binary         0.722 Preprocessor1_Model01
>  6 Fold3         1 roc_auc  binary         0.720 Preprocessor1_Model01
```

```
>  7 Fold4         1 accuracy binary        0.654 Preprocessor1_Model01
>  8 Fold4         1 roc_auc  binary        0.655 Preprocessor1_Model01
>  9 Fold5         1 accuracy binary        0.705 Preprocessor1_Model01
> 10 Fold5         1 roc_auc  binary        0.709 Preprocessor1_Model01
> # ... with 290 more rows
```

Look at the output of `tune::collect_metrics()` in the previous code chunk. What is the meaning of the `.estimate` column? For each `k`, compute the average CV statistic as well as its standard error using `dplyr::group_by()` and `dplyr::summarise()` for both performance metrics (i.e. ROC-AUC and accuracy). Save your results in a variable called `cv_errors`. Which `k` corresponds to the smallest CV accuracy? What is the highest `k` that still satisfies the one-standard error-rule and, hence, results in a more parsimonious model (based on accuracy)?

Hint: The resulting `cv_errors` object should contain at least the four following columns: `neighbors`, `.metric`, `mean`, `std_err`. Otherwise, you may have to adjust the variable names employed in the plot in task 2.10.

Task 2.10

Plot the misclassification errors using the code below. Why does the CV misclassification error curve differ from the test error curve? Does the CV approach indeed produce an unbiased estimate of a model's performance? Please explain your judgement.

Note: The misclassification error of a model can be computed as one minus the accuracy of the model.

```
ggplot2::ggplot() +
  # plot train errors
  geom_point(aes(x = neighbors, y = pred_error, color = "Train"), train_set_errors) +
  geom_line(aes(x = neighbors, y = pred_error, color = "Train"), train_set_errors) +
  # plot test errors
  geom_point(aes(x = neighbors, y = pred_error, color = "Test"), test_set_errors) +
  geom_line(aes(x = neighbors, y = pred_error, color = "Test"), test_set_errors) +
  # plot cv errors
  geom_point(aes(x = neighbors, y = (1-mean), color = "CV"),
             cv_errors %>% filter(.metric == "accuracy")) +
  geom_line(aes(x = neighbors, y = (1-mean), color = "CV"),
            cv_errors %>% filter(.metric == "accuracy")) +
  # plot cv error uncertainty
  geom_errorbar(aes(x = neighbors, y = (1-mean),
                    ymin = (1-mean) - std_err, ymax = (1-mean) + std_err),
                cv_errors %>% filter(.metric == "accuracy"), width = .5, alpha = 0.4) +
  labs(x = "Number of neighbors", y = "Misclassification error", color = "Legend") +
  theme_classic()
```

Task 2.11

Run the code below step-by-step and briefly explain the graph that it creates.

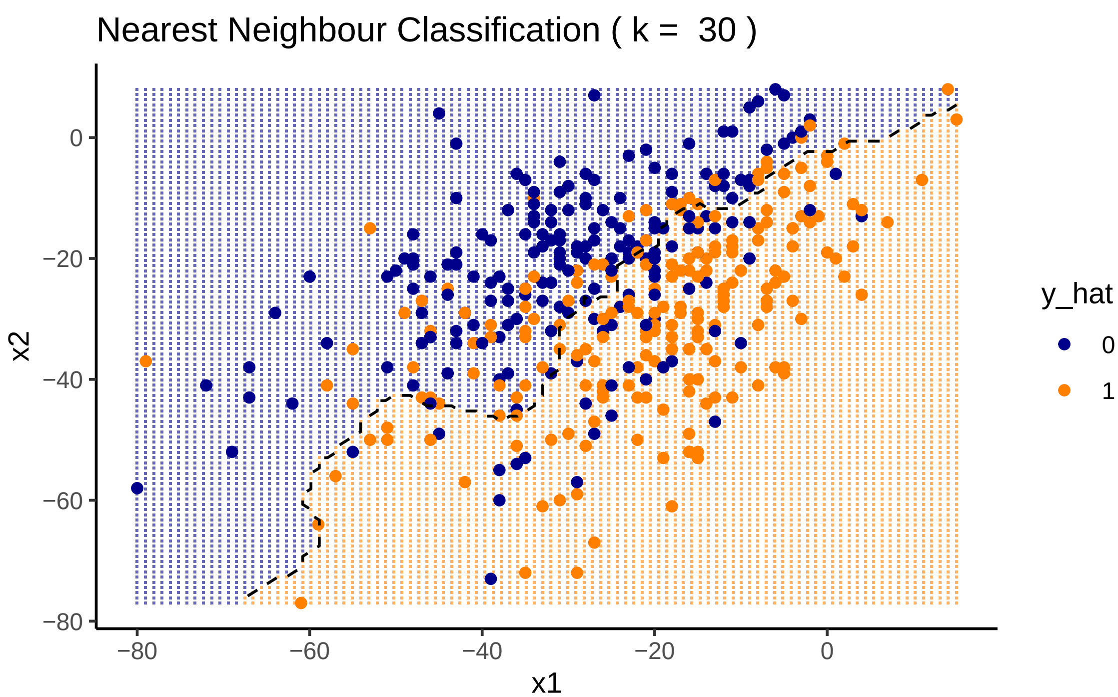
```
k <- 30

data_grid <-
  tidyr::expand_grid(
    x1 = seq(min(test_set_tn$x1), max(test_set_tn$x1), length = 100),
```

```r
    x2 = seq(min(test_set_tn$x2), max(test_set_tn$x2), length = 100)
  )

preds_knn <-
  knn_predict(k = k, new_data = data_grid)$.pred_class

data_grid %>%
  dplyr::mutate(y_hat = preds_knn) %>%
  ggplot2::ggplot(aes(x = x1, y = x2, z = as.integer(y_hat))) +
    geom_point(aes(color = y_hat), shape = ".", alpha = 0.6) +
    geom_point(aes(x = x1, y = x2, z = NULL, color = y), data = test_set_tn) +
    geom_contour(colour = "black", size = .5, bins = 1, lty = "dashed") +
    scale_color_manual(values = c("blue4", "darkorange1")) +
    labs(title = paste("Nearest Neighbour Classification ( k = ", k, ")")) +
    theme_classic()
```



Task 2.12

Run the code again, but choose `k = 1`, `k = 50` and `k = 300` in the first line. Compare the graphs. Which of these three models is the most flexible? Explain in one sentence what happens if you would set `k = 500`.

Task 3: Bootstrapping

In this exercise, you will work with the `Carseats` dataset from the `ISLR` package. First, you will try to predict the unit sales at each location using multiple linear regression, and estimate the test error of this regression model using the train-test-split approach.

```
data(Carseats, package = "ISLR")

Carseats %>%
  tibble::as_tibble()
```

```
> # A tibble: 400 x 11
>    Sales CompPrice Income Advertising Population Price ShelveLoc   Age Education
>    <dbl>     <dbl>  <dbl>       <dbl>      <dbl> <dbl> <fct>     <dbl>     <dbl>
>  1  9.5        138     73          11        276   120 Bad          42        17
>  2 11.2        111     48          16        260    83 Good         65        10
>  3 10.1        113     35          10        269    80 Medium       59        12
>  4  7.4        117    100           4        466    97 Medium       55        14
>  5  4.15       141     64           3        340   128 Bad          38        13
>  6 10.8        124    113          13        501    72 Bad          78        16
>  7  6.63       115    105           0         45   108 Medium       71        15
>  8 11.8        136     81          15        425   120 Good         67        10
>  9  6.54       132    110           0        108   124 Medium       76        10
> 10  4.69       132    113           0        131   124 Medium       76        17
> # ... with 390 more rows, and 2 more variables: Urban <fct>, US <fct>
```

Task 3.1

Fit a multiple linear regression model, called `fit_lm_cs`, that uses `Price`, `Urban`, and `US` to predict `Sales`. Print the results using `summary()`.

Task 3.2

Estimate the test error of this model using the train-test-split approach. In order to do this, perform the following steps:

i. Split the dataset into a training set and a validation set, each encompassing half of the data. Use `set.seed(2021)` in your code to ensure reproducibility. *Hint: You may use the `initial_split()` function from the `rsample` package.*
ii. Fit a multiple linear regression model called `fit_lm_cs_train` using only the training observations. Briefly compare the results with `fit_lm_cs` from task 3.1 (regarding the estimates, standard errors and p-values).
iii. Predict the response for the 200 test set observations and calculate the mean squared error (MSE).
iv. How does your answer to iii change if you use the random seeds 2020 or 2022 instead of 2021 when splitting the dataset?
v. Compute the leave-oneout cross-validation (LOO-CV) estimate for the MSE using the `cv.glm()` function from the `boot` package. *Hint: The mean squared error (MSE) can be extracted from the resulting list via $delta.*

Task 3.3

Use the `regsubsets()` function from the `leaps` package to find the best subset consisting of three predictors to estimate `Sales`, excluding `ShelveLoc`. Apply the function in a way to conduct *stepwise forward selection*. What are the three predictors selected by this approach? Fit a multiple linear regression using these three predictors and compute the LOO-CV estimate for the MSE. Compare with your answer to task 3.2 v. Can you explain the difference?

Task 3.4

Compute the mean of the response (`Sales`). In a next step, you want to understand the empirical distribution of the mean. For this purpose, create 20 bootstrapped replicates of the data, using random seed 2021 (`set.seed(2021)`). Then apply the mean function to each bootstrapped replicate. What is the range of the mean, i.e. the lowest and the largest number?

Hint: You may refer to the `bootstraps()` function from the `rsample` or to the `boot()` function from the `boot` package. If you are unsure how to apply the function, you may study the 'Examples' section on the help page of the respective function (e.g., by calling `help(bootstraps)`).

Task 3.5

Repeat the above analysis using `boot()` and `set.seed(2021)`. What is the 99% confidence interval for the mean?

Hint: A quick Google search query with "R boot compute mean" may help you find a good solution for implementing the given task.

Task 4: Linear Model Selection and Regularization

In this final exercise, you will predict diamond prices using the `diamonds` dataset from the `ggplot2` package. The diamonds dataset consists of:

- `price` (in US dollars), which will be the response,

and quality information (9 predictors) for around 54,000 diamonds. There are four C's of diamond quality:

- `carat` (weight),
- `cut` (quality of the cut: Fair/Good/Very Good/Premium/Ideal),
- `colour` (from worst J to best D) and
- `clarity` (from worst to best: I1, SI1, SI2, VS1, VVS1, VVS2, IF).

In addition, there are five physical measurements:

- `depth` (total depth percentage, calculated from x, y and z),
- `table` (width of top of diamond relative to widest point),
- `x` (length in mm),
- `y` (width in mm), and
- `z` (depth in mm).

```
data(diamonds, package = "ggplot2")

diamonds
```

```
> # A tibble: 53,940 x 10
>    carat cut       color clarity depth table price     x     y     z
>    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
>  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
>  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
>  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
>  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
>  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
>  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
>  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
>  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
>  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
> 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
> # ... with 53,930 more rows
```

To make things easier (in terms of runtime), the following code chunk reduces the data to a tenth of its size.

```
set.seed(2021)

diamonds <- diamonds %>%
  dplyr::slice_sample(n = nrow(.) / 10)

diamonds
```

```
> # A tibble: 5,394 x 10
>    carat cut       color clarity depth table price     x     y     z
>    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
>  1  0.71 Ideal     F     SI2      62.1    54  2365  5.73  5.76  3.57
>  2  1.61 Ideal     G     VS2      62.2    56 13553  7.52  7.47  4.66
>  3  0.24 Ideal     D     VVS2     61      57   526  4.03  4.07  2.47
>  4  1.01 Premium   F     VS1      60.8    59  7017  6.45  6.41  3.91
>  5  0.32 Premium   D     VVS1     62      60   973  4.4   4.37  2.72
>  6  1.29 Premium   H     SI1      61.6    57  6588  7.02  6.97  4.31
>  7  0.4  Very Good E     VS2      62.2    58   912  4.67  4.72  2.92
>  8  1.4  Ideal     H     SI2      61.2    56  7084  7.18  7.23  4.41
>  9  0.96 Premium   I     SI2      61.3    60  2669  6.45  6.25  3.9
> 10  1    Ideal     E     SI1      62.9    56  5935  6.33  6.38  4
> # ... with 5,384 more rows
```

Your aim will be to predict `price`, based on some or all of the predictors. Eventually, you want to understand which of the predictors are important in estimating the price and how the predictors are related to the price.

Task 4.1

Get an overview of the `diamonds` data. What are the three highest prices in the dataset? How many carats do those diamonds weigh? What is the mean weight? Which color is the most prevalent? Plot `price` against `carat` as well as their logged forms against each other using `ggplot()`.

Task 4.2

Due to skewed, non-linear relationship between `price` and `carat` observed in task 4.1, it seems reasonable to transform `price` and `carat` to `log_carat` and `log_price` for the following linear regressions.

```
diamonds <- diamonds %>%
  mutate(across(c(price, carat), log, .names = "log_{.col}")) %>%
  dplyr::select(-price, -carat)

diamonds
```

```
> # A tibble: 5,394 x 10
>    cut       color clarity depth table     x     y     z log_price log_carat
>    <ord>     <ord> <ord>   <dbl> <dbl> <dbl> <dbl> <dbl>     <dbl>     <dbl>
>  1 Ideal     F     SI2      62.1    54  5.73  5.76  3.57      7.77  -0.342
>  2 Ideal     G     VS2      62.2    56  7.52  7.47  4.66      9.51   0.476
>  3 Ideal     D     VVS2     61      57  4.03  4.07  2.47      6.27  -1.43
>  4 Premium   F     VS1      60.8    59  6.45  6.41  3.91      8.86   0.00995
>  5 Premium   D     VVS1     62      60  4.4   4.37  2.72      6.88  -1.14
>  6 Premium   H     SI1      61.6    57  7.02  6.97  4.31      8.79   0.255
>  7 Very Good E     VS2      62.2    58  4.67  4.72  2.92      6.82  -0.916
>  8 Ideal     H     SI2      61.2    56  7.18  7.23  4.41      8.87   0.336
>  9 Premium   I     SI2      61.3    60  6.45  6.25  3.9       7.89  -0.0408
> 10 Ideal     E     SI1      62.9    56  6.33  6.38  4         8.69   0
> # ... with 5,384 more rows
```

Now, perform forward and backward stepwise selection to choose the best subset of predictors for `log_price`. Compare the two results qualitatively as well as visually (by plotting the fitted objects using `plot(...,` `scale = "adjr2")`). Using adjusted R^2 as decision criterion, how large is the best subset from the backward stepwise selection?

Hint: Adjusted R^2 values can be extracted from the fitted model using `summary(model)$adjr2`.

Task 4.3

What are the main differences between using adjusted R^2 for model selection and using cross-validation (with mean squared test error, i.e. MSE)?

Task 4.4

Use Lasso and Ridge regression on the data to predict the `log_price`. Therefore, employ the `cv.glmnet()` function from the `glmnet` package and fit the models using the MSE as loss function and `nfolds = 5` as the number of folds. What is the optimal penalty hyperparameter (*lambda*) and the corresponding MSE in both cases? Which predictors are selected by the optimal Lasso model?

Hint: Unfortunately, you cannot feed a data frame object to the first argument `x` of `cv.glmnet()`. Instead, the function requires a predictor matrix as input. To convert the `diamonds` data frame into a matrix object you may use the `model.matrix` function.

Sources

Some of the exercises are based on those from other machine/statistical learning courses, i.e. by the Norwegian University of Science and Technology (NTNU) and the Albert-Ludwigs-Universität Freiburg.