Introduction to Machine Learning: Exercises

Florian Huber

June 24, 2025

Exercise 1: Our first machine learning workflow

The WDBC dataset contains measurements from fine needle aspirates of breast masses taken during clinical examinations. Each sample is described by 30 numeric features capturing cell characteristics such as radius, texture, and smoothness. The goal is to classify tumors as malignant or benign based on these features.

Step 1: Load the data set:

```
library(tidyverse)
library(mclust)
data(wdbc)
wdbc <- as_tibble(wdbc)
wdbc$ID <- factor(wdbc$ID)
wdbc

?wdbc
```

Before doing any further investigations on this dataset, we will randomly split this table into a “training set” and a “test set”. The training set will contain 80%, the test set 20% of the observations. **Put the test set “aside” and don’t use it for any data exploration or model training**. It will be used later to evaluate the performance of your machine learning model.

Step 2: Split into train and test set:

```
train_inds <- sample(x = 1:nrow(wdbc), size = round(0.8*nrow(wdbc)))

wdbc_train <- wdbc[train_inds, ]
wdbc_test <- wdbc[-train_inds, ]
```

Step 3: Make your pick:

From the help file (command: `?wdbc`) look at what the different features mean and choose 3 that you think may be helpful in predicting whether the tumour sample will be classified as benign (B) or malign (M). Remove the ID column. Save them in the variables `wdbc_train_subset` and `wdbc_test_subset`. Feel free to do some exploratory data analysis

Step 4a: Train your KNN model

Using k = 1 and two other values for k of your choice run the `knn3()` function from the caret package to train your KNN classifier.

```
library(caret)
```

```
# example with k = 1:
knn_fit_k1 <- knn3(Diagnosis ~ ., data = wdbc_train_subset, k = 1)
```

Step 4b: Evaluate the performance of your KNN model:

Using the models you just fitted, make your predictions! Make the predictions on both your training and your test set.

```
wdbc_train_subset$pred_k1 <- predict(knn_fit_k1, wdbc_train_subset, type = "class")

wdbc_test_subset$pred_k1 <- predict(knn_fit_k1, wdbc_test_subset, type = "class")
```

Which ways can you think of to assess the performance of your classifier, i.e. how well your model predicts the correct outcome?

Task: after deciding on a performance metric, compare the performance on the training and the test set for the different values of k. What can you observe?

Bonus exercise: make a plot showing two of your predictor variables on the x- and y-axes and display Diagnosis in different colours. Do this once for the train and once for the test set.

Steps 5a and 5b: Train a logistic regression model

Repeat the steps from 4a and 4b using logistic regression!

To run the logistic regression you will need:

```
# for training:
log_fit <- glm(Diagnosis ~ feature1 + feature2 + feature3,
               data = wdbc_train_subset, family = "binomial")

# for prediction:
wdbc_train_subset$pred_logreg <- predict(log_fit, type = "response", newdata = wdbc_train_subset)

# to get classes, need to define some probability threshold, e.g.:
prob_thresh <- 0.5
wdbc_train_subset$pred_logreg_class <- ifelse(wdbc_train_subset$pred_logreg > prob_thresh, "M", "B")
mean(wdbc_train_subset$pred_logreg_class == wdbc_train_subset$Diagnosis)
```

Exercise 2: classifier performance: constructing a ROC and a precision-recall curve

- In this exercise we will construct ROC and precision-recall curves from our logistic regression results from exercise 1.
- Before continuing, try answering the following two questions:
  - Do you think that the ROC curve should be constructed from the training or from the test set?
  - Why can't we construct a ROC curve from our KNN classifier?

Step 1: Inspecting the outcome of our logistic regression

Make a scatter plot that has one of the features you selected on the x-axis and the predicted probability on the y-axis. Label the points by the label ("Diagnosis" variable). Indicate your threshold using a horizontal line (e.g. in ggplot using `geom_hline(yintercept = 0.5, linetype = "dotted")`). Where are the true and false positives and negatives in this plot?

Step 2: Making a ROC and a precision-recall curve using the ROCR package

- Load `library(ROCR)`.
- We will start by making a ROC curve.
- First, create a "prediction object". You will need to pass the predicted probabilities as well as the true labels as arguments. Consult the help file of the `prediction()` function: `?prediction` to find out how to use it.
- Next, create a "performance object" from the previously created prediction object. You will need to pass the performance values you are interested in using the "measure" and "x.measure" arguments. Consult the help page `performance` for further information.
- Passing the performance object to `plot()` will give you the desired plot.
- Now repeat and adjust the previous steps in order to obtain a precision-recall plot.
- Bonus exercise: Can you find out how to extract the exact cutoff for a given TPR? Hint: after creating the correct performance object use `str()` to investigate the object structure and extract the right numbers using, for example, `my_perf_object@x.values`.

Exercise 3: tuning a model

Remember exercise 1 where at some point you were supposed to pick a random value for K and then hoped for the best.

We will now perform some hyperparameter tuning and find the best value for K using a cross-validation approach.

To get started, run:

```r
# let's recreate our train and test sets with the hand-picked features so that we
# don't have column names there that we don't need
wdbc_train_subset <- wdbc_train[, my_cols]
wdbc_test_subset <- wdbc_test[, my_cols]

wdbc_train_subset$ID <- NULL
wdbc_test_subset$ID <- NULL

wdbc_train_subset

grid <- expand.grid(k = seq(1, 80, by = 1))
grid
```

Define a `ctrl` object using the `trainControl()` function. Read the documentation - which two arguments do you need to set for a five-fold cross-validation?

```r
set.seed(123)
ctrl <- trainControl(method = "cv", number = 5, savePredictions = TRUE)
ctrl
```

Now we have all ingredients to run caret's train function using five-fold cross-validation. Take some time to understand what's going on and what the different arguments mean:

```r
knn_fit <- caret::train(Diagnosis ~ ., data = wdbc_train_subset,
                        method = "knn", tuneGrid = grid,
                        trControl = ctrl, preProcess = c("center", "scale"))
knn_fit
```

Inspect your `knn_fit` outcome by evaluating it and then passing it to `plot()`, and `confusionMatrix()`.

```r
print(knn_fit)
plot(knn_fit)
```

```
caret::confusionMatrix(knn_fit$pred$pred, knn_fit$pred$obs)
```

In a proper machine learning workflow, we are not done yet. We still have to steps ahead of us:

- We still need to estimate the generalisation error: train the KNN classifier on the complete training data using the optimal K found and determine the test set accuracy.
- The final classifier that should be used in production should be trained on the complete data set (train + test): the test set has served its purpose. In case further model tuning is desired later on, a new test set will be needed.

Exercise 4

- Run the following code:

```
chemgen <- readRDS("./datasets/m_all.rds")
chemgen <- dplyr::rename(chemgen, "drug" = "drugname_typaslab", "moa" = "process_broad")

chemgen_mlinput <- chemgen
chemgen_mlinput$conc <- NULL
chemgen_mlinput$drug <- NULL
```

- `chemgen` is a chemical genetics data set where *E. coli* mutants where subjected to different drug treatments and their fitness was measured. The drug concentration and mode of action (moa) has been recorded for each observation.
- In the following code, we predict the mode of action using the fitness values for the different genes. Notably, we introduce **blocking** (also known as **grouping**) to make sure that observations with the same drug but different concentrations always 'travel together', i.e. that they *all* end up in **either the training or the test set but not both**).
- What performance do you get if you do not use blocking (grouping)? Comment out the `index` argument in the `trainControl()` function to do so. Why is it problematic?

```
library(ranger)

folds <- groupKFold(group = chemgen$drug, k = 5)

ctrl <- trainControl(method = "cv",
                     number = 5,
                     index = folds,
                     classProbs  = TRUE,
                     savePredictions = "final")

grid <- expand.grid(
  mtry = c(round(sqrt(ncol(chemgen_mlinput))),
           round(0.5*sqrt(ncol(chemgen_mlinput)))),
  splitrule = "gini",
  min.node.size = c(1, 3, 7)
)

rf_fit <- caret::train(
  moa ~ .,
  data = chemgen_mlinput,
  num.trees = 1000,
  method = "ranger",
  tuneGrid = grid,
  trControl = ctrl
)

plot(rf_fit)
```

Exercise 5

- In this exercise we will apply PCA to a gene expression data set.
- First, load, prepare, and inspect the data set:

```
library(tidyverse)

# read the data
trans_cts <- read_csv("./datasets/counts_transformed.csv")
sample_info <- read_csv("./datasets/sample_info.csv")

# Create a matrix from our table of counts
pca_matrix <- trans_cts %>%
  column_to_rownames("gene") %>%
  as.matrix() %>%
  # transpose the matrix so that rows = samples and columns = variables
  t()

dim(pca_matrix)
pca_matrix[1:5, 1:5]
```

- Now perform a PCA on the data set using the `prcomp()` function and save the result in a variable `pca`. In this case, you can leave the scale argument at FALSE `scale. = FALSE` because the features are already on similar scales.

- Inspect the PCA output using `str(pca)`.

- Can you guess what the different components of `pca` are? Use the documentation (`?prcomp`) for help.

- Extract the samples after rotation (`pca$x`) and convert it to a tibble: `pca_x <- as_tibble(pca$x, rownames = "sample")`.

- Use the `separate()` function to put the sample information into different columns: `separate(pca_x, col = "sample", sep = "_", into = c("genotype", "tpt", "replicate"))`

- Now plot the first principal components against each other and visualise the time point and the genotype on the plot.

- Bonus exercise: each principal component explains a certain fraction of the variance in the original data. Can you plot the proportion of the variance explained (y-axis) vs. the number of the principal component (x-axis)?

- Note: This exercise has been adapted from here, which features an even longer discussion. Credit to Hugo Tavares!

Exercise 6

- Load the iris data set (`data(iris)`) and plot some of the variables against each other, labelling the points by species. This should allow you to already spot some of the clusters.
- Perform k-means clustering using the `kmeans()` function (already loaded when you start R).
- Add the cluster information back to the original data. Can you plot an overlay of the species label and the predicted cluster (tip: combine `geom_colour` and `geom_text`?
- Can you spot which cluster corresponds to which species? Can you think of a way to quantify the correlation between the species and the cluster number?

Exercise 7

- Run the following code:

```r
chemgen <- readRDS("./datasets/m_all.rds")
chemgen <- dplyr::rename(chemgen, "drug" = "drugname_typaslab", "moa" = "process_broad")

my_rows <- paste(chemgen$drug, chemgen$conc, chemgen$moa, sep = "_")
chemgen_m <-
  dplyr::select(chemgen, -one_of("drug", "conc", "moa")) %>%
  as.matrix()
rownames(chemgen_m) <- my_rows
```

- `chemgen` is a chemical genetics data set where *E. coli* mutants where subjected to different drug treatments and their fitness was measured. The drug concentration and mode of action (moa) has been recorded for each observation.
- Run a hierarchical clustering on the observations (rows) using the functions `dist()` and `hclust()`. You can plot the result by passing the hclust object to `plot()` (set the cex argument in plot to `cex = 0.15`).
- Extract the clusters by using the `cutree()` function.
- Bonus exercise: how are the 4 modes of action distributed between the clusters?
- Bonus exercise: perform the clustering by using correlation-based distance (1 = lowest distance, -1 = largest distance). Check the help file for dist, `?dist()` to get a hint.
- Tip: The **ComplexHeatmap** package supports an extensive option of visualising clustering.

License