Introduction to Machine Learning

Working Group "Computational Statistics" – Bernd Bischl et al.

Code demo for KNN

Recap the functioning of k-NN

- For each point to be predicted, calculate the distance or dissimilarity to all training data points using any distance measure
- Most common distance measures:
  - Metric feature vectors in $\mathbb{R}^p$:
    - Euclidean distance: $d_2(a,b) = \sqrt{(a-b)^T(a-b)} = \sqrt{(a_1-b_1)^2 + \cdots + (a_p-b_p)^2}$
    - Manhattan distance: $d_1(a,b) = |a_1-b_1| + \cdots + |a_p-b_p|$
    - Mahalanobis distance: $d_{\text{Mahalanobis}}(a,b) = \sqrt{(a-b)^T S^{-1}(a-b)}$, where S is the covariance matrix of the features.
  - Categorical feature vectors:
    - Simple Matching Coefficient: relative proportion of matching attributes
    - Jaccard Coefficient: Similar to above, but discards entries where neither a nor b have a value of 1.
  - Mixed feature vectors:
    - Gower distance
- Select the k nearest neighbors to the point and use the most frequently occurring class in this neighborhood as prediction

Example

We look at the `iris` data set which consists of only numeric features and a categorical output. In the lecture we learnt that for k-NN the *representation* is the training data. Let's say our training data is only 50 percent of our original data.

```
data(iris)

set.seed(7)

train_relative_size <- 0.5
train_indices <- sample(
  x = seq(1, nrow(iris), by = 1), size =
    ceiling(train_relative_size * nrow(iris)), replace = FALSE
)
iris_train <- iris[ train_indices, ]
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1..
```

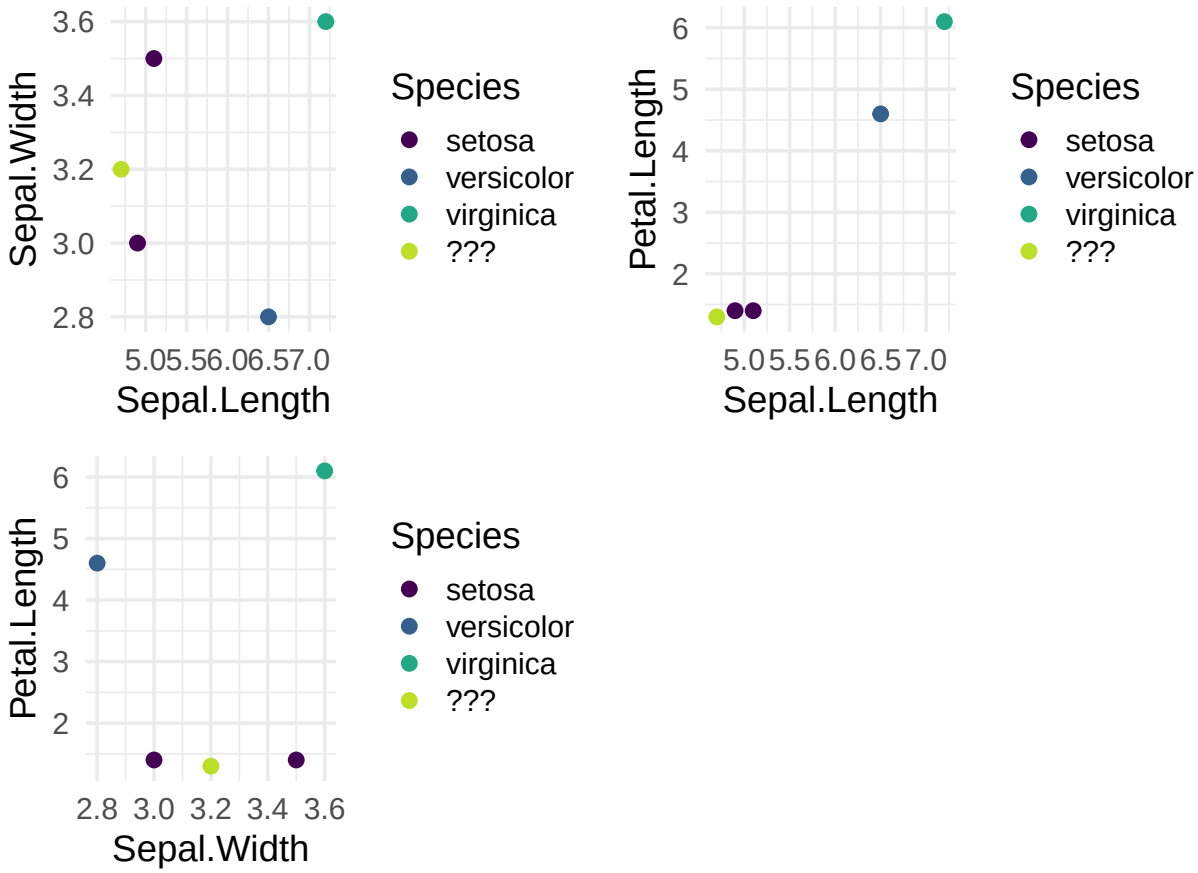
Imagine five data points:

```r
data_snippet <- iris[c(1, 2, 3, 55, 110), -4]
levels(data_snippet[, "Species"]) <- c(levels(data_snippet[, "Species"]), "???")
data_snippet[3, "Species"] <- "???"
print(data_snippet)
```

```
##     Sepal.Length Sepal.Width Petal.Length    Species
## 1            5.1         3.5          1.4     setosa
## 2            4.9         3.0          1.4     setosa
## 3            4.7         3.2          1.3        ???
## 55           6.5         2.8          4.6 versicolor
## 110          7.2         3.6          6.1  virginica
```

Which class would you select as prediction for the unknown observation?

```r
library(ggplot2)
library(gridExtra)

proj_1 <- ggplot(
  data_snippet,
  aes(
    x = Sepal.Length,
    y = Sepal.Width,
    color = Species,
  )) +
  geom_point(size = 3)
proj_2 <- ggplot(
  data_snippet,
  aes(
    x = Sepal.Length,
    y = Petal.Length,
    color = Species
  )) +
  geom_point(size = 3)
proj_3 <- ggplot(
  data_snippet,
  aes(
    x = Sepal.Width,
    y = Petal.Length,
    color = Species
  )) +
  geom_point(size = 3)

grid.arrange(proj_1, proj_2, proj_3, ncol = 2, nrow = 2)
```



Implementation

The function takes a vector of training data class labels, the training data, the data on which we want to perform k-NN classification as well as the k parameter, and the distance measure we want to use. We also include the option to standardize the features. (Why is that important?)

```
distance_euclidean <- function(a, b) {
  sqrt(sum((a - b)^2))
}

# function to classify data with knn
get_knn <- function(train_labels, train_data, to_classify_data, k,
                    distance = distance_euclidean, standardize = FALSE) {
  # TODO: for a real implementation, we would need to add input checks here to
  #       make our implicit assumptions about the inputs explicit:
  #  train_labels: vector with no NAs, same length as rows of train_data
  #  train_data: data.frame or matrix without NAs, same columns as to_classify_data
  #  to_classify_data: data.frame or matrix without NAs
  #  k: single positive integer, at most number of rows of train_data
  #  distance: a function taking two arguments (hard to check beyond that...)

  num_preds <- nrow(to_classify_data)
  pred <- rep(as.character(NA), num_preds)
```

```
  # transform data to speedup computation
  train_data <- data.matrix(train_data)
  to_classify_data <- data.matrix(to_classify_data)

  # standardize the feature vectors if desired
  if (standardize == TRUE) {
    train_data <- scale(train_data)
    to_classify_data <- scale(
      to_classify_data,
      center = attr(train_data, "scaled:center"),
      scale = attr(train_data, "scaled:scale")
    )
  }
  for (i in seq_len(num_preds)) {
    # compute distance to all training data points
    distance_to_train <- apply(
      train_data,
      MARGIN = 1,
      FUN = function(x) distance(x, to_classify_data[i, ])
    )
    # extract row indices of k nearest ones
    nearest_neighbors <- order(distance_to_train)[1:k]
    # compute frequencies of classes in neighborhood
    class_frequency <- table(train_labels[nearest_neighbors])
    most_frequent_classes <-
      names(class_frequency)[class_frequency == max(class_frequency)]
    # random tie breaking if more than 1 class has maximal frequency.
    # (causes the occasionally weird, fuzzy class boundaries in the figs. below)
    pred[i] <- sample(most_frequent_classes, 1)
  }

  list(prediction = pred, levels = levels(train_labels))
}
```

Let's check how well the training data is classified with our implementation for $k = 5$:

```
to_check_result <- get_knn(
  train_labels = iris_train$Species,
  train_data = iris_train[, 1:4],
  to_classify_data = iris_train[, 1:4],
  k = 5, standardize = FALSE
)
table(actual = iris_train$Species, predicted = to_check_result$prediction)
```

```
##             predicted
## actual       setosa versicolor virginica
##   setosa         28          0         0
##   versicolor      0         19         0
##   virginica       0          0        28
```

```
correct <- (iris_train$Species == to_check_result$prediction)
print(paste(
```

```
  100 * round(mean(correct), 4),
  "% correctly classified"
))
```

```
## [1] "100 % correctly classified"
```

Since the model above is based on a $p = 4$ dimensional feature vector, it's hard to visualize what k-NN actually does.

To get a better understanding how k-NN works, at least for just two features, we can write a simple plot function, where we divide the two dimensional feature space into a grid and color all grid cells according to the class that k-NN would assign to their midpoint based on a given training data set. Data points that are misclassified are shown in white, the others in gray.

```
# function for a general classification method with two features to visualize
# class boundaries X1 and X2 are the names of the two features to use.
plot_2D_classify <- function(to_classify_labels,
                             to_classify_data,
                             classify_method,
                             X1, X2,
                             lengthX1 = 100, lengthX2 = 100,
                             title = "",
                             plot_class_rate = TRUE) {
  gridX1 <- seq(
    min(to_classify_data[, X1]),
    max(to_classify_data[, X1]),
    length.out = lengthX1
  )
  gridX2 <- seq(
    min(to_classify_data[, X2]),
    max(to_classify_data[, X2]),
    length.out = lengthX2
  )

  # compute grid coordinates with cartesian product
  grid_data <- expand.grid(gridX1, gridX2)
  names(grid_data) <- c(X1, X2)

  # assign grid cells to classes based on classification rule:
  grid_result <- classify_method(
    to_classify_data = grid_data
  )
  grid_data$prediction <- grid_result$prediction

  # assign data to be classified based on classification rule & check these
  # "predictions"
  to_check_result <- classify_method(
    to_classify_data = to_classify_data[, c(X1, X2)]
  )
  to_classify_data$class <- to_classify_labels
  to_classify_data$correct <- (to_check_result$prediction == to_classify_labels)
```

```
  ggplot() +
    geom_raster(
      data = grid_data,
      aes_string(x = X1, y = X2, fill = "prediction"),
      alpha = .8
    ) +
    geom_point(
      data = to_classify_data,
      aes_string(x = X1, y = X2, shape = "class", color = "correct"),
      alpha = .8
    ) +
    scale_color_manual(
      values = c("TRUE" = "darkgray", "FALSE" = "white"),
      guide = FALSE
    ) +
    labs(
      fill = "Class", shape = "Class",
      title =  parse(text = if(plot_class_rate) paste0(

        title, '~ ": " ~',
        100 * round(mean(to_classify_data$correct), 4),
        '~ "% correctly classified"'
      )
        else title
      )
    ) +
    theme(plot.title = element_text(size = 16))
}
```

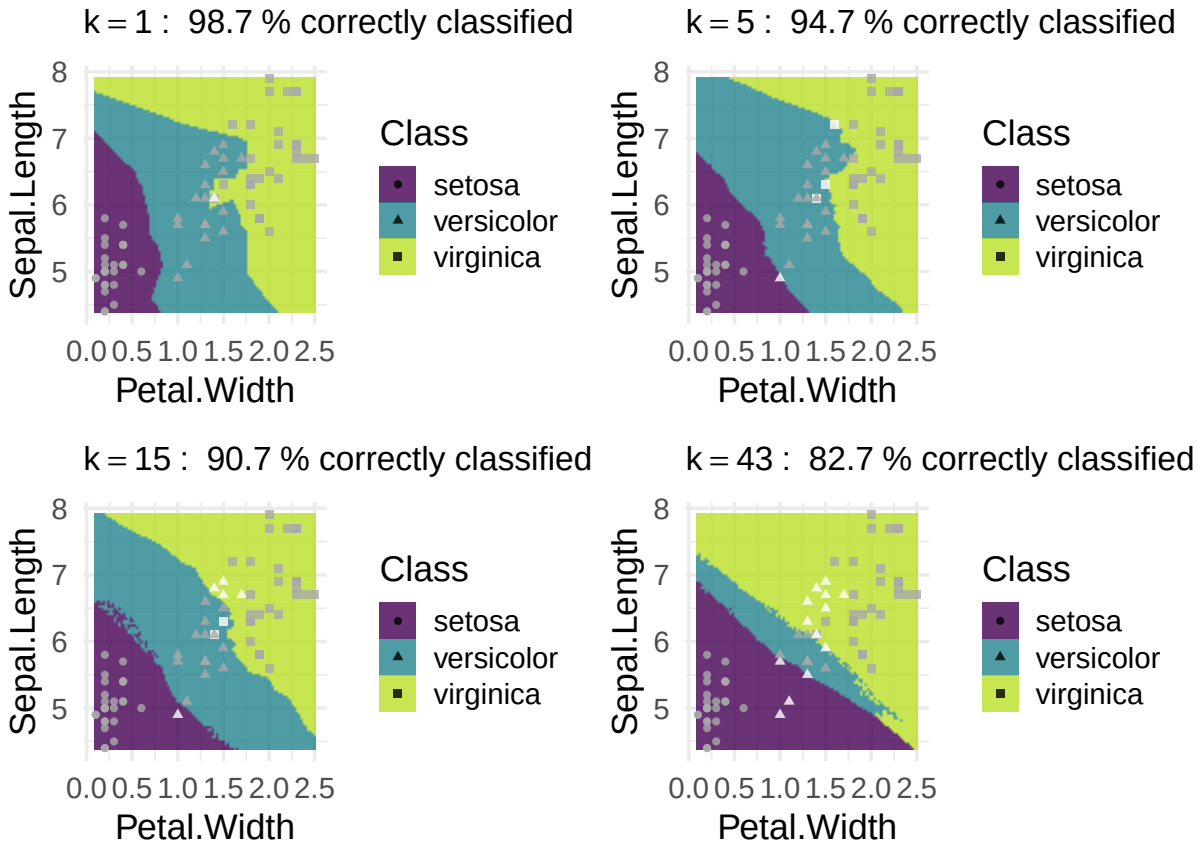
```
# function for simple k-NN with two features to visualize class boundaries
# X1 and X2 are the names of the two features to use.
plot_2D_knn <- function(train_labels, train_data, k, X1, X2,
                        distance = distance_euclidean, standardize = FALSE,
                        # by default, we "predict" the class of the training data:_
                        to_classify_labels = train_labels,
                        to_classify_data = train_data,
                        lengthX1 = 100, lengthX2 = 100,
                        plot_class_rate = TRUE) {

  plot_2D_classify(to_classify_labels = to_classify_labels,
                   to_classify_data = to_classify_data,
                   classify_method = function(to_classify_data)
                       get_knn(train_labels = train_labels,
                               train_data = train_data[, c(X1, X2)],
                               to_classify_data = to_classify_data,
                               k = k, distance = distance,
                               standardize = standardize),
                   X1, X2,
                   lengthX1 = lengthX1, lengthX2 = lengthX2,
                   title = paste0("k == ", k),
                   plot_class_rate = plot_class_rate)

}
```

Let's plot the classification we did before, but now with only two features:

```
plot_2D_knn(
  train_labels = iris_train$Species,
  train_data = iris_train[, -5],
  k = 3,
  X1 = "Petal.Width", X2 = "Sepal.Length",
)
```

Visualize results for different k:

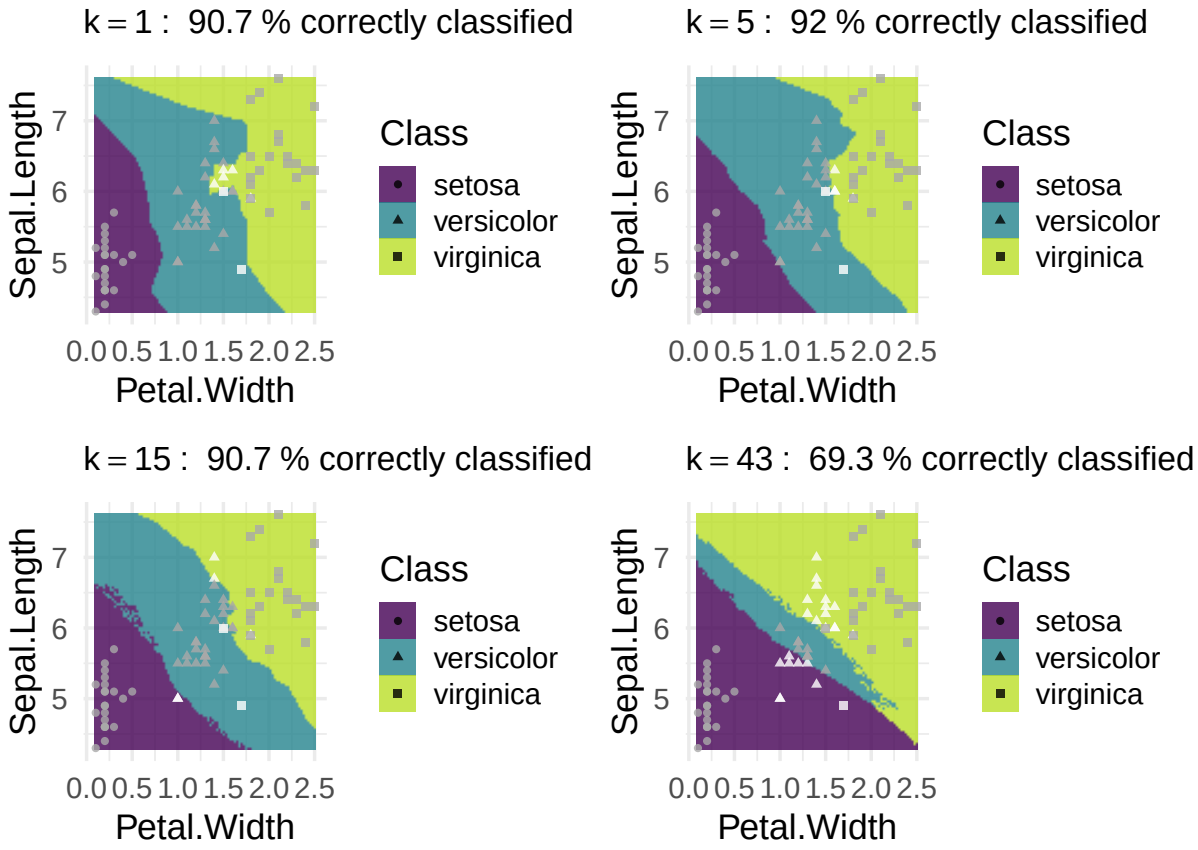
```
k_values <- c(1, 5, 15, 43)
ggplot_list <- lapply(
  k_values,
  function(k) {
    plot_2D_knn(
      train_labels = iris_train$Species,
      train_data = iris_train[, -5],
      k = k,
      X1 = "Petal.Width", X2 = "Sepal.Length"
    )
  }
)
do.call(grid.arrange, ggplot_list)
```



For $k = 1$ we can only ever get a misclassification if there exist observations with identical feature values but different labels in our training data set. So why should we use a $k \neq 1$?

Let's look how we perform when we apply our k-NN classification rules to the entries in the `iris` data set which were not used for training:

```
# new, previously unseen data to classify:
iris_to_check <- iris[-train_indices, ]
k_values <- c(1, 5, 15, 43)
ggplot_list <- lapply(
  k_values,
  function(k) {
    plot_2D_knn(
      train_labels = iris_train$Species,
      train_data = iris_train[, -5],
      k = k,
      to_classify_labels = iris_to_check$Species,
      to_classify_data = iris_to_check[, -5],
      X1 = "Petal.Width",
      X2 = "Sepal.Length"
    )
  }
)
do.call(grid.arrange, ggplot_list)
```



Now we observe that, on data we didn't use as training data, the value of k for which we get the best classification accuracy seems to be higher than 1.

Models very often perform much better on the training data than on other data sets, this is called **overfitting** and will be discussed in detail in the chapters on performance evaluation and tuning.

Standardization

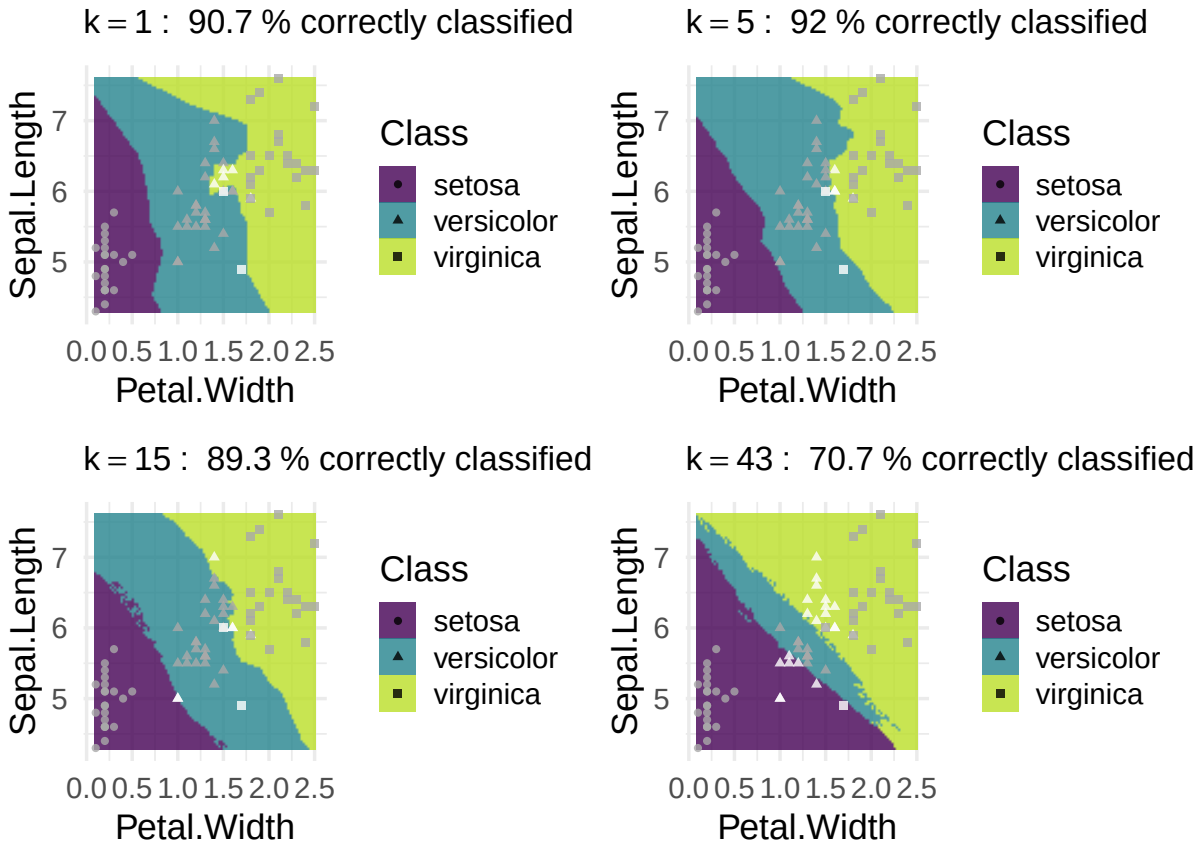
We included the option to standardize features by subtracting their mean and dividing by their standard deviation so that the value of the i-th value of the j-th feature is:

$$x^{(i)}_{j,\text{standardized}} = \frac{x^{(i)}_j - \overline{x}_j}{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x^{(i)}_j - \overline{x}_j\right)^2}}.$$

This makes sense if features are on totally different scales (e.g. millimeter vs. kilometer) or even in completely different units (e.g. kilogram vs Euro). In such cases we would compare apples with oranges and the distances would be weighted unequally. After standardization, all features are dimensionless and the numbers simply represent how many standard deviations an observation of a feature is above or below its mean. (Can you figure out under which conditions the Mahalaobis distance and the Euclidean distance of the standardized features are identical?)

Does standardization improve performance for the `iris` data?

```r
k_values <- c(1, 5, 15, 43)
ggplot_list <- lapply(
  k_values,
  function(k) {
    plot_2D_knn(
      train_labels = iris_train$Species,
      train_data = iris_train[, -5],
      k = k,
      ##
      standardize = TRUE,
      ##
      to_classify_labels = iris_to_check$Species,
      to_classify_data = iris_to_check[, -5],
      X1 = "Petal.Width",
      X2 = "Sepal.Length"
    )
  }
)
do.call(grid.arrange, ggplot_list)
```



`mlr3` implementation

The `mlr3` package offers a unified interface to many different machine learning algorithms making complicated, inefficient and error-prone implementations as above unnecessary.
Check the tutorial. It uses a simple syntax, as used below:

```r
library(mlr3)
library(mlr3learners)

# define task
iris_task <- TaskClassif$new(id = "iris_train", backend = iris_train,
                             target = "Species")
# define learner and check possible models on mlr3 homepage
knn_learner <- lrn("classif.kknn", k = 5)
# check available parameter settings
knn_learner$param_set
```

```
## ParamSet:
##          id    class lower upper
## 1:        k ParamInt     1   Inf
## 2: distance ParamDbl     0   Inf
## 3:   kernel ParamFct    NA    NA
## 4:    scale ParamLgl    NA    NA
##                                                                levels
## 1:
## 2:
## 3: rectangular,triangular,epanechnikov,biweight,triweight,cos,...
## 4:                                                       TRUE,FALSE
##    default value
## 1:       7     5
## 2:       2
## 3: optimal
## 4:    TRUE
```

```r
# train the model
knn_learner$train(iris_task)
# predict on test data
iris_pred <- knn_learner$predict_newdata(newdata = iris_to_check)
# check "confusion" matrix
iris_pred$confusion
```

```
##             truth
## response     setosa versicolor virginica
##   setosa         22          0         0
##   versicolor      0         26         2
##   virginica       0          5        20
```

```r
iris_pred$score()
```

```
## classif.ce
##     0.0933
```