Classification task: k Nearest Neighbours (kNN)

The prediction task: to predict which mobile phones can be solved for a high price.

The rationale: help a company determine its price policy; more precisely, help it identify which phones it can offer to its customers on a high price.

The algorithm to be used: K Nearest Neighbours (kNN)

Load and inspect the data

The data originates from the Mobile Price Classification dataset, where further information about it can be found (including the description of all the variables).

Notes about the data set:

- The original data set has been adapted for this class.
- The data set used in Naive Bayes class originates from the same data set, but was adapted in a different way than the one used in this class; hence, the data sets used in this and the Naive Bayes class are not identical.

Load the data

```
mobiles_data <- read.csv("data/mobiles.csv")
```

Inspect the data

```
str(mobiles_data)
```

```
## 'data.frame':    2000 obs. of  14 variables:
##  $ battery_power: int  842 1021 563 615 1821 1859 1821 1954 1445 509 ...
##  $ clock_speed  : num  2.2 0.5 0.5 2.5 1.2 0.5 1.7 0.5 0.5 0.6 ...
##  $ dual_sim     : int  0 1 1 0 0 1 0 1 0 1 ...
##  $ fc           : int  1 0 2 0 13 3 4 0 0 2 ...
##  $ int_memory   : int  7 53 41 10 44 22 10 24 53 9 ...
##  $ m_dep        : num  0.6 0.7 0.9 0.8 0.6 0.7 0.8 0.8 0.7 0.1 ...
##  $ mobile_wt    : int  188 136 145 131 141 164 139 187 174 93 ...
##  $ n_cores      : int  2 3 5 6 2 1 8 4 7 5 ...
##  $ pc           : int  2 6 6 9 14 7 10 0 14 15 ...
##  $ ram          : int  2549 2631 2603 2769 1411 1067 3220 700 1099 513 ...
##  $ sc_h         : int  9 17 11 16 8 17 13 16 17 19 ...
##  $ sc_w         : int  7 3 2 8 2 1 8 3 1 10 ...
##  $ talk_time    : int  19 7 9 11 15 10 18 5 20 12 ...
##  $ price_range  : int  1 2 2 2 1 1 3 0 0 0 ...
```

```r
summary(mobiles_data)
```

```
##  battery_power     clock_speed       dual_sim            fc
##  Min.   : 501.0   Min.   :0.500   Min.   :0.0000   Min.   : 0.000
##  1st Qu.: 851.8   1st Qu.:0.700   1st Qu.:0.0000   1st Qu.: 1.000
##  Median :1226.0   Median :1.500   Median :1.0000   Median : 3.000
##  Mean   :1238.5   Mean   :1.522   Mean   :0.5095   Mean   : 4.294
##  3rd Qu.:1615.2   3rd Qu.:2.200   3rd Qu.:1.0000   3rd Qu.: 7.000
##  Max.   :1998.0   Max.   :3.000   Max.   :1.0000   Max.   :16.000
##    int_memory         m_dep           mobile_wt        n_cores
##  Min.   : 2.00   Min.   :0.1000   Min.   : 80.0   Min.   :1.00
##  1st Qu.:16.00   1st Qu.:0.2000   1st Qu.:109.0   1st Qu.:3.00
##  Median :32.00   Median :0.5000   Median :141.0   Median :4.00
##  Mean   :32.05   Mean   :0.5018   Mean   :140.2   Mean   :4.52
##  3rd Qu.:48.00   3rd Qu.:0.8000   3rd Qu.:170.0   3rd Qu.:7.00
##  Max.   :64.00   Max.   :1.0000   Max.   :200.0   Max.   :8.00
##        pc              ram            sc_h            sc_w
##  Min.   : 0.000   Min.   : 256   Min.   : 5.00   Min.   : 0.000
##  1st Qu.: 5.000   1st Qu.:1208   1st Qu.: 9.00   1st Qu.: 2.000
##  Median :10.000   Median :2146   Median :12.00   Median : 5.000
##  Mean   : 9.916   Mean   :2124   Mean   :12.31   Mean   : 5.767
##  3rd Qu.:15.000   3rd Qu.:3064   3rd Qu.:16.00   3rd Qu.: 9.000
##  Max.   :20.000   Max.   :3998   Max.   :19.00   Max.   :18.000
##    talk_time      price_range
##  Min.   : 2.00   Min.   :0.00
##  1st Qu.: 6.00   1st Qu.:0.75
##  Median :11.00   Median :1.50
##  Mean   :11.01   Mean   :1.50
##  3rd Qu.:16.00   3rd Qu.:2.25
##  Max.   :20.00   Max.   :3.00
```

We will use the `price_range` variable to create the outcome variable:

```r
table(mobiles_data$price_range)
```

```
##
##   0   1   2   3
## 500 500 500 500
```

Create the outcome variable - `top_price` - by considering that `price_range` value of 3 denotes top price:

```r
mobiles_data$top_price = ifelse(test=mobiles_data$price_range == 3,
                                yes="Yes", no="No")
mobiles_data$top_price[1:10]
```

```
##  [1] "No"  "No"  "No"  "No"  "No"  "No"  "Yes" "No"  "No"  "No"
```

Transform the outcome variable into factor, as required for the classification task

```r
mobiles_data$top_price = as.factor(mobiles_data$top_price)
mobiles_data$top_price[1:10]
```

```
##  [1] No  No  No  No  No  No  Yes No  No  No
## Levels: No Yes
```

Remove from the data set the variable used for creating the outcome variable.

```r
mobiles_data$price_range = NULL
```

Data preparation

kNN algorithm typically works with numerical data. *Note*: kNN can be used with categorical data, as well, but in that case it requires a distance measure that is appropriate for categorical data, such as Hamming distance. Such a use of the kNN algorithm is out of the scope of this course.

In the current data set, all input variables except one (dual_sim) are numerical variables (either int or real value numbers). As such, they can be used for model building.

If we had an ordinal variables encoded as a factor - for example, satisfaction of the customers w/ the phone, expressed on the 5-point scale from `Not All` to `Fully Satisfied` - we could transform it into an int variable and use for model building.

On the other hand, if we had a true categorical variable encoded as a factor - for example, the color of the phone - we would not use it. As noted above, it is generally possible, but is more complicated and we will not consider such cases.

For computing similarity among instances, we will use the Euclidean distance (the most commonly used distance measure). Since this distance measure is very sensitive to the value range of variables (variables with larger value range tend to dominate over those with smaller range), we should check the range of the input variables and if their value ranges differ significantly do the scaling of variables.

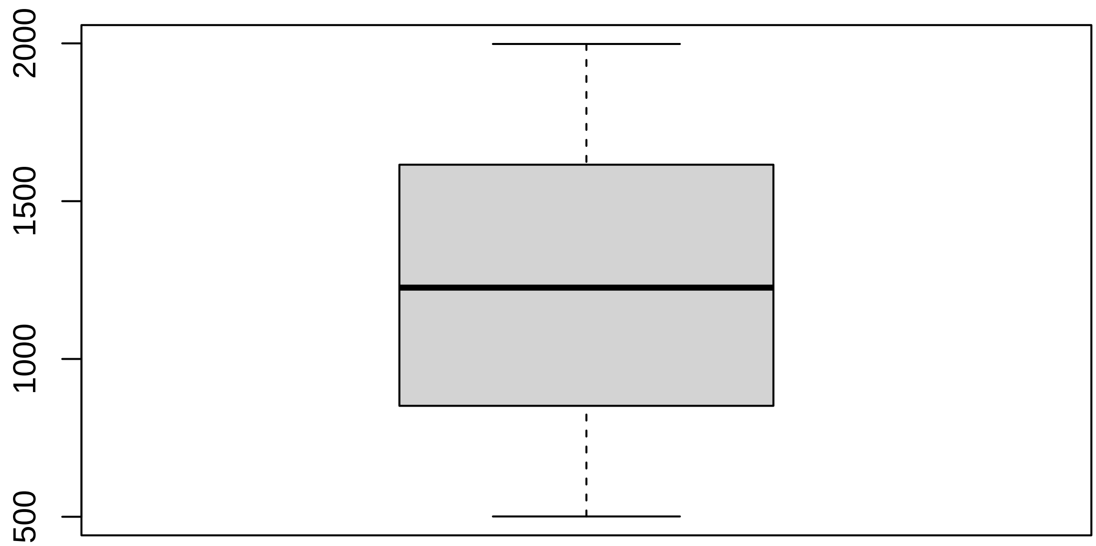
Scaling of input variables The purpose of scaling is to change the value range of variables, so that all variables have the same or at least similar range. Two frequently used approaches for scaling variables are:

- *Normalisation*: it reduces variables to the same value range (typically 0-1). Its advantage: all variables have the same value range; its disadvantage: should not be applied in the presence of outliers
- *Robust standardisation*: it reduces the difference in value ranges by:
  - subtracting the median from the variable values, so that the median of the transformed variable is 0, and
  - dividing the result with the interquartile range (as a measure of variability), so that variability is 1

When possible, the use of normalisation is preferred. However, if outliers are present, robust standardisation should be used instead. In the latter case, the transformed variables will not have the same value range, but the difference will be reduced.

So, let's check if the input variables have outliers. This is often done using the boxplot. For example, we can check the presence of outliers in the `battery_power` attribute:

```r
boxplot(mobiles_data$battery_power)
```



We can also use R's method that compute the statistics for a boxplot - `boxplot.stats` - to directly get outliers, if any:

```r
boxplot.stats(mobiles_data$battery_power)
```

```
## $stats
## [1]  501.0  851.5 1226.0 1615.5 1998.0
##
## $n
## [1] 2000
##
## $conf
## [1] 1199.008 1252.992
##
## $out
## integer(0)
```

Apply this check, now, to all the attributes

```r
apply(mobiles_data[,-14], 2, function(x) length(boxplot.stats(x)$out))
```

```
## battery_power   clock_speed      dual_sim            fc    int_memory
##             0             0             0             0             0
```

```
##          m_dep      mobile_wt        n_cores              pc             ram
##              0              0              0               0               0
##           sc_h           sc_w      talk_time
##              0              0              0
```

None of the input variables have outliers => we can apply normalisation

Normalisation is typically done using the following formula: (x - min(x))/(max(x) - min(x))

```
normalise_var <- function(var) {
  (var - min(var, na.rm = T))/(max(var, na.rm = T) - min(var, na.rm = T))
}
```

Apply it to all input variables

```
apply(mobiles_data[,-14], 2, normalise_var) |> as.data.frame() -> mobiles_norm
```

Check that the value ranges are really the same

```
apply(mobiles_norm, 2, range) |> as.data.frame()
```

```
##   battery_power clock_speed dual_sim fc int_memory m_dep mobile_wt n_cores pc
## 1             0           0        0  0          0     0         0       0  0
## 2             1           1        1  1          1     1         1       1  1
##   ram sc_h sc_w talk_time
## 1   0    0    0         0
## 2   1    1    1         1
```

Add the output variable to the transformed input variables

```
mobiles_norm$top_price = mobiles_data$top_price
```

We can now proceed to splitting the data into training and test sets and building models.

IMPORTANT NOTE: for now, we are considering all variables as potentially relevant for building a prediction model. In a week or two, we will learn how to select variables that are truly relevant candidates for model building.

Train-test split

We will split the data into training and test sets in the same way we did before, using 80% of the data for the training set.

```
library(caret)
```

```
set.seed(2024)
training_indices = createDataPartition(mobiles_norm$top_price, p = 0.75, list = FALSE)
train_ds = mobiles_norm[training_indices, ]
test_ds = mobiles_norm[-training_indices, ]
```

Make predictions using the kNN algorithm

We will start by loading the *class* library that offers a function (knn) for doing the classification task using the kNN algorithm. Note: the `knn` function can be used for the regression task, as well.

```
library(class)
```

Initial knn classification We will first do the classification with a randomly chosen value for k.

```
set.seed(2024)
rand_k = sample(seq(3,21,2), 1) ## reminder: we consider only odd values for k, to avoid ties

knn1_pred = knn(train = train_ds[,-14],
                test = test_ds[,-14],
                cl = train_ds$top_price,
                k = rand_k)
```

Reminder: the kNN algorithm does not build a model, but directly makes prediction on the test data based on the training data. Hence, the result of the knn function are predicted values of the outcome variable on the test set.

We will now create the confusion matrix and compute the evaluation measures.

```
cm1 = table(true = test_ds$top_price, predicted=knn1_pred)
cm1
```

```
##      predicted
## true   No Yes
##   No  357  18
##   Yes  55  70
```

Get the function for computing the evaluation measures from the util.R script

```
source("util.R")
```

```
compute_eval_measures_v1(cm1)
```

```
##  accuracy precision    recall        F1
##    0.8540    0.7955    0.5600    0.6573
```

There are two things that primarily affect the results of knn-based classification:

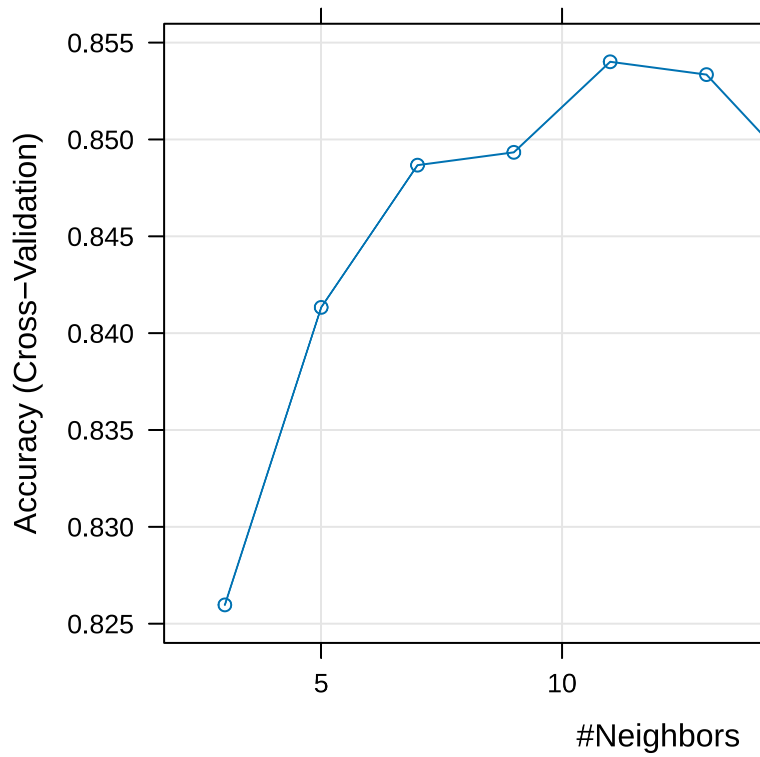
- the value of k, that is, the number of nearest neighbours to consider
- the distance measure used when identifying neighbours

However, the knn function uses the Euclidean metric as the distance measure and does not allow for an alternative metric. So, we will seek to improve the results by finding the optimal value for `k`.

```r
train_setup = trainControl(method = "cv", number = 10)
k_grid = expand.grid(.k = seq(from=3, to=21, by=2))

set.seed(2024)
knn_cv = train(x = train_ds[,-14],
               y = train_ds$top_price,
               method = "knn",
               tuneGrid = k_grid,
               trControl = train_setup)
```

```r
plot(knn_cv)
```



Finding optimal k value through cross-validation

```r
knn_cv
```

```
## k-Nearest Neighbors
##
## 1500 samples
##   13 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
```

```
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1349, 1350, 1350, 1350, 1351, 1350, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    3  0.8259703  0.4933926
##    5  0.8413304  0.5209428
##    7  0.8486728  0.5301134
##    9  0.8493351  0.5273533
##   11  0.8540108  0.5429066
##   13  0.8533484  0.5385156
##   15  0.8480151  0.5058304
##   17  0.8433483  0.4827105
##   19  0.8386948  0.4576339
##   21  0.8373526  0.4526622
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 11.
```

Create new predictions using the optimal `k` value

```r
optimal_k = knn_cv$bestTune$k

knn2_pred = knn(train = train_ds[,-14],
                test = test_ds[,-14],
                cl = train_ds$top_price,
                k = optimal_k)
```

Note: another option to consider is using k=13 since the performance is very similar, while the variance is smaller (at the expense of bias that starts increasing - remember the bias/variance trade-off)

```r
cm2 = table(true = test_ds$top_price, predicted = knn2_pred)
cm2
```

```
##      predicted
## true   No Yes
##   No  365  10
##   Yes  56  69
```

```r
compute_eval_measures_v1(cm2)
```

```
##  accuracy precision    recall        F1
##    0.8680    0.8734    0.5520    0.6765
```

Compare the results

```r
rbind(compute_eval_measures_v1(cm1),
      compute_eval_measures_v1(cm2)) |>
  round(digits = 4) |>
  as.data.frame()
```

```
##   accuracy precision recall     F1
## 1    0.854    0.7955  0.560 0.6573
## 2    0.868    0.8734  0.552 0.6765
```

The two models are very similar in terms of performance - the first has better recall, at the expense of weaker precision compared to the 2nd model

Final notes:

- How improper selection of K may lead to under-fitting or over-fitting: in general, very low `k` values tend to lead to high variance and over-fitting, whereas very high `k` values tend to lead to high bias and under-fitting. A nice explanation is given in this video.
- How to choose distance measure: while we will use Euclidean measure in this course, you may want to learn a bit about alternative measures and how to make the choice which one to use; in that case, this short video might be useful.