**PRACTICAL**

**Metabolomics using Machine Learning:**

**Part A**

**Tuesday, 14th January 2025**

**Maria Anastasiadi (m.anastasiadi@cranfield.ac.uk)**

**Introduction**

Machine learning is a complex field, but the workflow involved in machine learning tasks remains more or less constant:

Data Exploration

Data Preparation

Model Training

Model Evaluation

Model Improvement

After data collection, we always need to evaluate data quality, both in terms of how the data is represented/encoded (e.g. consistency, missing values, types of variables) and whether they meet our expectations (e.g. do we have observations spanning all classes in our study, or are there any obvious biases?) The quality of any project is based largely on the quality of its input data.

Then, data preparation involves dealing with missing or unexpected values, eliminating unnecessary data and sometimes recoding data, to conform to expected inputs (e.g. R functions may treat the same values differently depending on whether they are numbers or factors).

Model training is then performed using an appropriate algorithm for the data at hand. In R, this is as straightforward as choosing an algorithm, selecting and installing a package and calling the relevant functions with suitable arguments.

Models are always biased, so it’s important to evaluate how well the algorithm learns from the available data and how well it may perform in the future. For such evaluations, it is standard practice to use separate training and test datasets.

Finally, depending on model performance, improvement may be required. Sometimes, this can be achieved by tweaking the parameters of the current model and re-training it. In other case, switching to a different type of model may be necessary.

**Dataset**

Our dataset comes from “Ensemble-based support vector machine classifiers as an efficient tool for quality assessment of beef fillets from electronic nose data”, by Fady Mohareb *et al.*, *Analytical Methods*, 2016, **8**, 3711

E-noses use an array of electronic chemical sensors to allow the detection of odours. The data e-noses generate are too abstract to use without some kind of processing. It is through statistical or machine learning techniques that e-noses can find real-world applications, such as spoilage profiling.

You have been supplied with two separate CSV files: EnoseAllSamples.csv and SensoryAllSamples.csv .

Table 1: EnoseAllSamples.csv (sample)



Table 2: SensoryAllSamples.csv (sample)



Sample ids (e.g. 48A0\_7) are unique and can be used to match rows across the two files. DF1, DF2, DF3 etc. correspond to e-nose sensor responses, whereas ‘sensory’ is a categorical variable. In this case, sensory class 1, 2, 3 denotes fresh, semi-fresh, spoiled samples respectively.

In this practical, we’ll focus on classification tasks (predicting the sensory class from e-nose sensor responses) using two different machine learning algorithms: knn and decision trees. You will need to install and load the following packages: **mixOmics, class, gmodels, caret, rpart, rpart.plot**.

**Data Loading**

Load the data for the two csv files and create two datasets “enose” and “sensory”. To load the data into R, you can use the **read.table()** function. For example:

**enose <- read.table('csv-file', sep=",", header=TRUE, row.names=1)**

In order to combine all rows from EnoseAllSamples.csv and SensoryAllSamples.csv, you can use the **merge()** function. This way you will ensure that the sample labels for the predictor (enose measurements) and target variables (sensory sore), as there is the chance that there is a mismatch between the labels or the number of rows in the two files. This could be a handling mistake from the technician recording the data, or the samples measured with enose may be different from the samples evaluated from the sensory panel.

**merged <- merge(enose, sensory, by="row.names")**

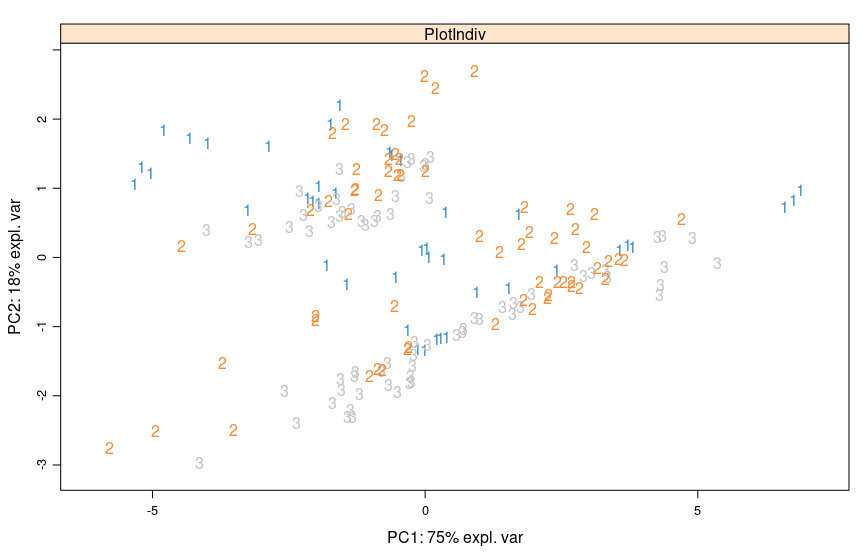
**rownames(merged) = merged[,1]**

When the two data sets were merged, an additional column was added to the merged dataset containing the row names. Remove the row names column from the merged dataset and call it **AllData**.

**AllData<-as.data.frame(merged[,-1])**

**Data Exploration**

Once the CSV data is loaded into R, you can use the **mixOmics** package to perform exploratory data analysis with PCA (**pca()** function). Do you observe any clustering of samples per class? You can plot PCA results with **plotIndiv()**:



**Figure 1**. PCA score plot of e-nose data.

**Data Preparation**

Many R machine learning classifiers require that the target feature is coded as a factor, so we will need to recode the **sensory** variable:

**AllData$sensory <- factor(AllData$sensory, levels = c("1", "2", "3"))**

Next, check the proportion of each class in the dataset to see if they are equally represented or if some classes are underrepresented. What do you observe?

**round(prop.table(table(AllData$sensory)) \* 100, digits = 1)**

Before training any models, you’ll need to partition the data into a training set (70% of observations) and a test set (30% of observations). The selection of observations for each set should be done randomly. However, it also needs to be done in a way that ensures all classes are represented fairly in the two sets and the correspondence between numerical (predictor) and categorical (response) data is preserved. You can do this easily with **createDataPartition()** as it returns an array of response vector positions as the partition (name the vector **trainIndex**). Before you partition the data set a seed so the results are reproducible.

**set.seed(8)**

**trainIndex <- createDataPartition(AllData$sensory, p = .7,**

**list = FALSE,**

**times = 1)**

You can then use these indexes to produce both the training and the test set. e.g.

**trainSet <- AllData[trainIndex,]; testSet <- AllData[-trainIndex,]**

Next define two vectors containing the training and test class names:

**trainCl <- trainSet[,ncol(trainSet)]**

**testCl <- testSet[,ncol(testSet)]**

**Task 1-K Nearest Neighbours (knn)**

**Model Training - knn**

The first machine learning algorithm you will use is **knn**. To apply knn you will use the function **knn()**, from the **class** package. You can start with **k=3**, to see how it works. This function requires only the predictor variables in the training and test set, so you need to remove the class variable from the trainSet and testSet.

**trainSet.knn <-trainSet[, -ncol(trainSet)]**

**testSet.knn <- testSet[, -ncol(testSet)]**

**model.k3 <- knn(trainSet.knn, testSet.knn, trainCl, k=3)**

Take a look at the output of the function using **summary(model.k3)**. Does **knn()** create a model? What does the **knn()** function return?

**Model Evaluation - knn**

For classification tasks, the most obvious way to evaluate a model is through the accuracy of class predictions when it comes to the test set.

Prediction accuracy, aka success rate, is the sum of all true positive and true negative predictions divided by the total number of predictions. To calculate this, we need an easy way to count true/false positive/negatives.

The **CrossTable()** function from the **gmodels** package is a convenient way to obtain these counts.

**cross.table <- CrossTable(testCl, model.k3, prop.chisq=FALSE, prop.t=FALSE, prop.c=FALSE, prop.r=FALSE)**

| model.k3

testCl | 1 | 2 | 3 | Row Total |

-------------|-----------|-----------|-----------|-----------|

1 | 8 | 4 | 0 | 12 |

-------------|-----------|-----------|-----------|-----------|

2 | 5 | 12 | 1 | 18 |

-------------|-----------|-----------|-----------|-----------|

3 | 6 | 4 | 11 | 21 |

-------------|-----------|-----------|-----------|-----------|

Column Total | 19 | 20 | 12 | 51 |

-------------|-----------|-----------|-----------|-----------|

By choosing prop.t=FALSE it also outputs the proportions of test samples attributed to each class.

Overall statistics and evaluation metrics, such as **Kappa, Sensitivity** and **Specificity** can be obtained easily using the **confusionMatrix()** function from the **caret** package. The **Balanced Accuracy** is an important metric as it provides the prediction accuracy for each classed based on its representation in the test set. **Note**: The predicted vs real columns in Crosstable and confusion matrix outputs are the other way-round.

**confusion.matrix <- confusionMatrix(model.k3, testCl, positive="3")**

What is the prediction accuracy of your **knn()** model?

**Model Improvement - Knn**

The first obvious improvement for our **knn()** approach is to find the optimal value of **k** for our data. You’ll need to write a function

**k.results <- function(n, trS, tstS, trCl, tstCl){}**

where:

n= the highest number of nearest neighbours you want to test,

trS = the training set

tstS = the test set

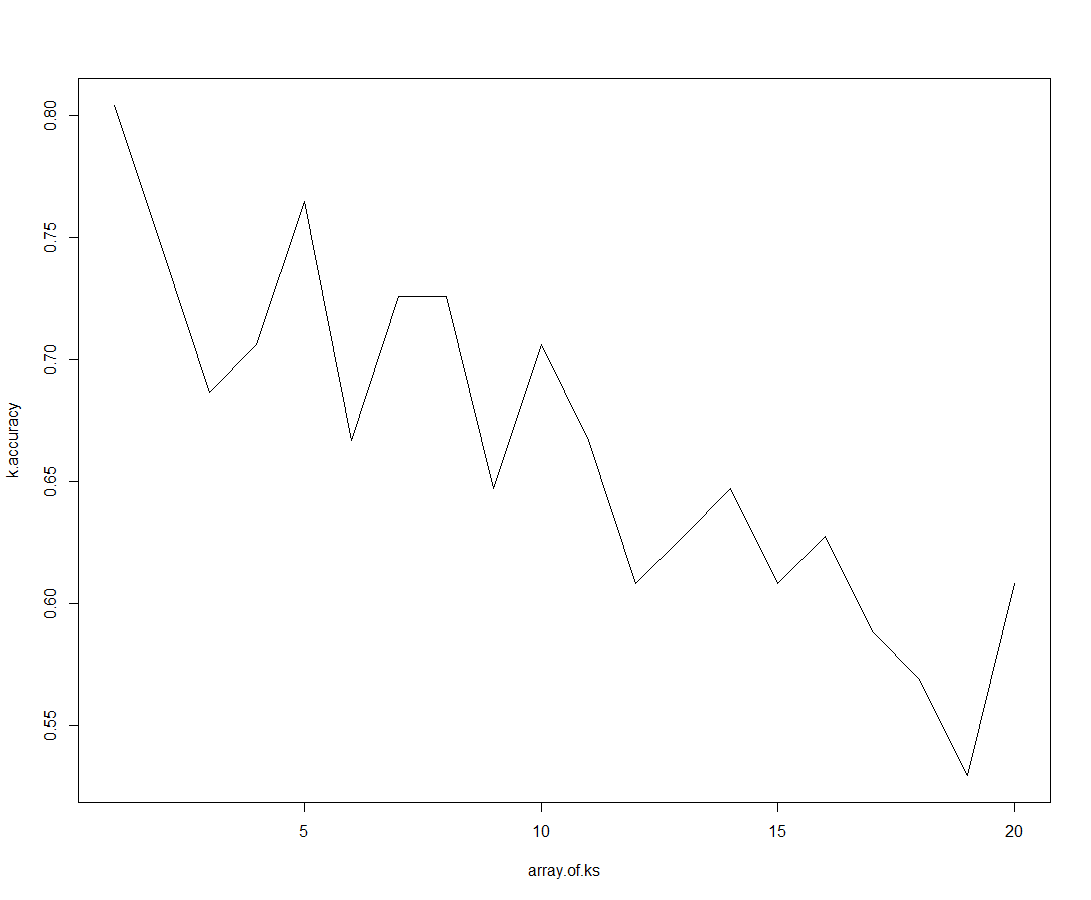
trCl = the training class

tstCl = the test class

Use a loop inside the function and perform the same analysis for values of k ranging from 1 to n. Also, ensure you store the accuracy results, as you are going to plot them later. You can retrieve the prediction accuracy programmatically from a confusion matrix object this way:

**confusion.matrix$overall[1]**

Use the **k.results()** function with n=20 and plot the accuracy results for each k.

How does the choice of k affect the accuracy?

**Figure 2.** Prediction accuracy of the knn algorithm for k values between 1 and 20 for unscaled data.

**Optional Task**

**Apply scaling prior to knn**

The second potential improvement is scaling the data before applying **knn()**since this is a distance-based algorithm. You can use the **preProcess()** function available with the caret package to do so. First you need to define which type of scaling you will be using and find the scaling parameters from the training set. Use the **preProcess()** function and define the scaling method as autoscaling:

**preProcValues <- preProcess(trainset.knn, method = c("center", "scale"))**

Then you can use the **predict()** function to apply the scaling to both the training set and test set using the same scaling parameters (i.e. mean and sd) for both.

**trainTransformed <- predict(preProcValues, trainset.knn)**

**testTransformed <- predict(preProcValues, testSet.knn)**

Now use the new training and test sets and create a knn model with k=3, as done for the unscaled data. Evaluate the model. Can you see any improvement compared to the model created with the raw data?

Finally use the **k.results()** function with n=20 on the scaled data and plot the effect of k on accuracy as before. Can you see any improvement after scaling?

**Task 2-Decision Trees**

**Model Training – Decision Trees**

The second machine learning algorithm you will use is **decision trees**. To do so, you will use the **rpart()** function from the **rpart** package.

The first argument in the **rpart()** function is a formula expression. The left-hand-side of the formula (response/target variables) should be either a numerical vector when a regression tree will be fitted (see regression methods on Wednesday) or a factor, when a classification tree is produced, as it is the case in this instance. The right-hand-side of the formula should be a series of numeric or factor variables (predictor variables) separated by +; there should be no interaction terms. Eg. you can include all the variables in the model by providing the generic formula: **trnCl~.** or you can choose to include/exclude variables by writing a formula such as:

**trainingClasses ~ DF1+DF3+DF4**

For this practical use the generic formula: **trnCl ~.**You also need to provide the dataset the model will be trained upon.

**model.tree <-rpart(sensory ~ ., data=trainSet)**

The **rpart()** function produces a model, which can be visualised with the **rpart.plot()**function.

e.g. **x11()**

**rpart.plot(model.tree, box.palette = "BuBn", type = 5)**

The leaf nodes in the plot tell us which class is classified by that node, and the proportions of each class in that node.

**Model evaluation – Decision Trees**

To see the prediction results for the **rpart()** model, you’ll need to **predict()** the classes of the test set first.

**predicted <- predict(model.tree, testSet, type="class")**

Other evaluation metrics, such as **Kappa, Sensitivity** and **Specificity** can be obtained easily using the **confusionMatrix()** function from the **caret** package. What is the accuracy of the tree model?

**Model Improvement – Decision Trees**

When it comes to decision trees, there are many ways to improve the model, so that it performs better when classifying other sets, countering the tendency of decision trees to over-fit. One such technique is pruning, which simplifies the decision tree, making less prone to overfitting.

The **rpart** package supports cost-complexity pruning through **prune()**. The **prune()** function prunes a fitted model object of class **rpart** to the **cp** value. The **cp** value (or complexity parameter) defines the complexity of the tree and takes values between 0.01 to 0.1. Explore the effect of pruning your existing tree model to prediction accuracy for **cp = 0.04** and assess the model as previously done. Has the prediction accuracy improved?

**pruned\_tree <-prune(model.tree, cp = 0.04)**

Finally, create a function which can calculate the prediction accuracy for different values of **cp** and stores the results in a vector.

**accuracy.cp <- function(model.tr, tstS, tstCl) {}**

where:

model.tr = the tree model you want to apply pruning to.

tstS = the test set

tstCl = the test class

Write a loop inside the function which evaluates accuracy for all **cp** values between 0.01-0.1.

Use the **accuracy.cp()** for the model.tree model we created initially and plot the accuracy results against the values of **cp**. Does pruning improve the prediction accuracy? Why?