Week 8 Practical

K-Nearest Neighbor, Naïve Bayes, and Rule-Based Classifiers

Overview

In this Practical we will revisit rule-based classifiers and learn how to build and interpret k-nearest neighbor (KNN) and Naïve Bayes classifiers.

We will learn about several useful packages (some being used before) and introduce a method for finding an optimal KNN model.

How to use this practical?

The practical provides a set of tasks you should follow to learn a set of skills you will need for you assignments. This is by no means intended to be a complete guide to all the functions and methods available out there for a specific task. It rather focuses on commonly used methods and a minimum you need to understand to be successful in your assignments.

Icon ❓ indicates a challenge you should try to solve. Those challenges are highly recommended.

Objectives

- Build a predictive model using KNN algorithm.
- Choose optimal K for a KNN model.
- Build a Naïve Bayes classification model.
- Build a rule-based classification predictive model.
- Make a prediction using models we built.

Sharing results

While tasks should be self-explanatory, some of the challenges might be more complex. Feel free to share your results or questions in the course discussion forum. Results to all the challenges will be available UPON REQUEST (discussion forum or email).

Datasets

In this practical, we will be using a dataset that was derived from one of the most popular datasets available at UCI Machine Learning Repository (https://archive.ics.uci.edu/). The original dataset has `quality` as the output variable that represents a wine quality score on the scale from 0 to 10. The derived dataset contains only records that belong to a derived `quality_class`, being labeled as 0 – medium wine quality and 1 – high quality of wine.

Table 1. Wine quality dataset description

Column Name	Description
fixed acidity	tartaric acid - g/dm3; most acids involved with wine or fixed or nonvolatile (do not evaporate readily)
volatile acidity	acetic acid - g/dm3; the amount of acetic acid in wine, which at too high of levels can lead to an unpleasant, vinegar taste
citric acid	g/dm3; found in small quantities, citric acid can add 'freshness' and flavor to wines
residual sugar	g/dm3; the amount of sugar remaining after fermentation stops, it's rare to find wines with less than 1 gram/liter and wines with greater than 45 grams/liter are considered sweet
chlorides	sodium chloride - g/dm3; the amount of salt in the wine
free sulfur dioxide	mg/dm3; the free form of SO2 exists in equilibrium between molecular SO2 (as a dissolved gas) and bisulfite ion; it prevents microbial growth and the oxidation of wine
total sulfur dioxide	mg/dm3; amount of free and bound forms of S02; in low concentrations, SO2 is mostly undetectable in wine, but at free SO2 concentrations over 50 ppm, SO2 becomes evident in the nose and taste of wine
density	g/cm3; the density of water is close to that of water depending on the percent alcohol and sugar content
pH	describes how acidic or basic a wine is on a scale from 0 (very acidic) to 14 (very basic); most wines are between 3-4 on the pH scale
sulphates	potassium sulphate - g/dm3; a wine additive which can contribute to sulfur dioxide gas (S02) levels, which acts as an antimicrobial and antioxidant
alcohol	% by volume; the percent alcohol content of the wine
quality class	0 for the medium quality wine (original scale 4-6) and 1 for the high-quality wine (original scale 7-9)

Before getting started

Before going through the tasks, make sure you have a new R Notebook created in RStudio, as shown in Figure 1.

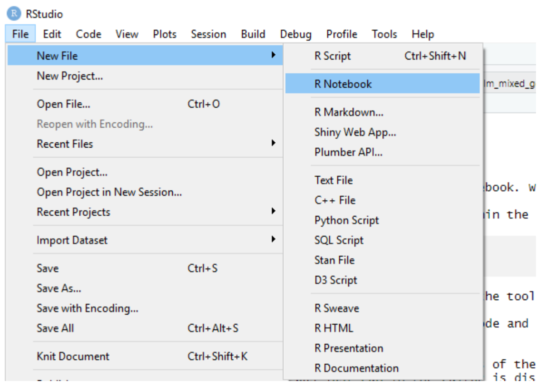


Figure 1. Opening R Notebook in RStudio

Each section of the code (marked by the grey box) should be executed within a single chunk in your Notebook.

Task 1. K-Nearest Neighbor classifier

As we already discussed in our lecture, KNN is a supervised machine learning algorithm that classifies a new data point into the target class, depending on the features of its neighboring data points. It is one of the most simple machine learning algorithms and it can be easily implemented for a varied set of problems. The algorithm is mainly based on feature similarity. That is, KNN checks how similar a data point is to its neighbor and classifies the data point into the class it is most similar to.

KNN is a lazy algorithm, this means that it memorizes the training data set instead of learning a discriminative function from the training data. It is applicable in solving both classification and regression problems.

As it was the case in previous practicals, we will start by loading the libraries we need and then a dataset we will be using:

```
# load libraries
library(class)
library(caret)

library(mlr3)
library(mlr3learners)
library(mlr3measures)
library(C50)
```

```
# load data
data <-
read.csv(url("https://raw.githubusercontent.com/sreckojoksimovic/infs5100/
main/wine-data.csv"))

# There might be a problem with column names, that is why we will assign
# column names before going ahead
names(data) <- c("fixed_acidity", names(data)[2:12])

# We should make sure that the output variable is in the right format
data$quality_class <- as.factor(data$quality_class)

# Finally, we will summarize dataset
summary(data)
```

If you take a careful look at Table 1, you will notice that the variables are on different scales. This is something you can also observe from the data summary. For example, while the values for `density` are between 0.990 and 1.004, whereas the range of values for `total sulfur dioxide` goes between 6 and 289. Being a distance-based algorithm, KNN is affected by the scale of the variables. Similar to K-Means, a commonly used clustering algorithm.

There are many ways to scale your variables and we will use a simple, step by step approach, as per below.

```
data$fixed_acidity <- scale(data$fixed_acidity, center = TRUE, scale =
TRUE)
data$volatile_acidity <- scale(data$volatile_acidity, center = TRUE,
scale = TRUE)
data$citric_acid <- scale(data$citric_acid, center = TRUE, scale = TRUE)
data$residual_sugar <- scale(data$residual_sugar, center = TRUE, scale =
TRUE)
data$chlorides <- scale(data$chlorides, center = TRUE, scale = TRUE)
data$free_sulfur_dioxide <- scale(data$free_sulfur_dioxide, center =
TRUE, scale = TRUE)
data$total_sulfur_dioxide <- scale(data$total_sulfur_dioxide, center =
TRUE, scale = TRUE)
data$density <- scale(data$density, center = TRUE, scale = TRUE)
data$pH <- scale(data$pH, center = TRUE, scale = TRUE)
data$sulphates <- scale(data$sulphates, center = TRUE, scale = TRUE)
data$alcohol <- scale(data$alcohol, center = TRUE, scale = TRUE)
```

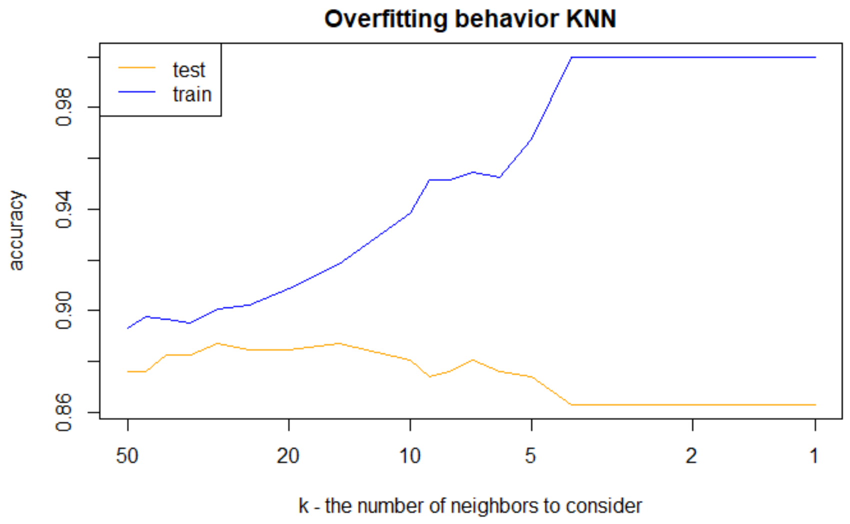
Once we have our dataset ready, we need to split the dataset into training and test sets. As it is commonly used, we will keep 70% for training and 30% for testing.

```
train.size <- .7

train.indices <- sample(x = seq(1, nrow(data), by = 1), size =
ceiling(train.size * nrow(data)), replace = FALSE)

wine.data.train <- data[ train.indices, ]
wine.data.test <- data[ -train.indices, ]
```

A critical aspect with KNN (again, similar to K-Means) is finding an optimal value for `K`, the number of neighbors to consider. One way to find an optimal value is to try a range of options. The code below creates a new Task (as we will be using `mlr3` package), prepares a range of K values to be tested and plots accuracy of the models obtained with different K-values.

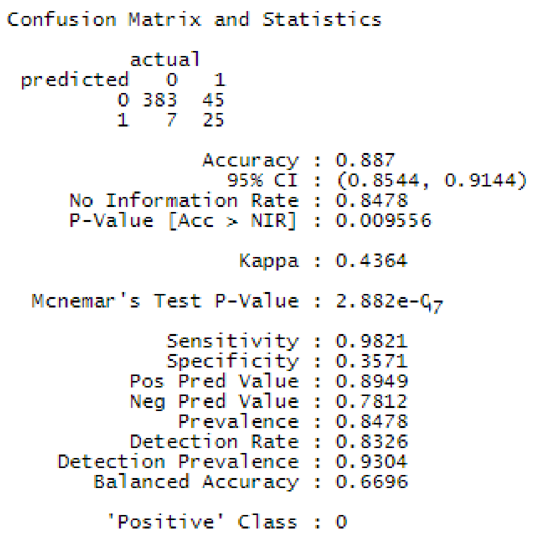
```
wine.task <- TaskClassif$new(id = "wine", backend = wine.data.train,
target = "quality_class")

# run experiment
k.values <- rev(c(1:10, 15, 20, 25, 30, 35, 40, 45, 50))
storage <- data.frame(matrix(NA, ncol = 3, nrow = length(k.values)))
colnames(storage) <- c("acc_train", "acc_test", "k")

for (i in 1:length(k.values)) {
  wine.learner <- lrn("classif.kknn", k = k.values[i])
  wine.learner$train(task = wine.task)
  # test data
  # choose additional adequate measures from: mlr3::mlr_measures
  wine.pred <- wine.learner$predict_newdata(newdata = wine.data.test)
  storage[i, "acc_test"] <- wine.pred$score(msr("classif.acc"))
  # train data
  wine.pred <- wine.learner$predict_newdata(newdata = wine.data.train)
        storage[i, "acc_train"] <- wine.pred$score(msr("classif.acc"))
        storage[i, "k"] <- k.values[i]
}
```

```
storage <- storage[rev(order(storage$k)), ]

plot(
  x = storage$k, y = storage$acc_train, main = "Overfitting behavior
KNN",
  xlab = "k - the number of neighbors to consider", ylab = "accuracy",
col = "blue", type = "l",
  xlim = rev(range(storage$k)),
  ylim = c(
  min(storage$acc_train, storage$acc_test),
  max(storage$acc_train, storage$acc_test)
  ),
  log = "x"
)
lines(x = storage$k, y = storage$acc_test, col = "orange")
legend("topleft", c("test", "train"), col = c("orange", "blue"), lty =
1)}
```



This is the type of plot we discussed in previous weeks. What we can see is that for smaller values of K (less than 10), the obtained models tend to overfit the data. It seems, from the plot above, that value of 30 is the optimal value for the number of neighbors to consider. Please note that you are more than welcome to try other options as well.

```
# Fit KNN with K=30
wine.learner.knn <- lrn("classif.kknn", k = 30)
wine.learner.knn$train(task = wine.task)
wine.pred.knn <- wine.learner.knn$predict_newdata(newdata = wine.data.test)

knn.results = confusionMatrix(table(predicted = wine.pred.knn$response,
                                    actual = wine.data.test$quality_class))

knn.results
```



```
Confusion Matrix and Statistics

          actual
predicted   0   1
        0 383  45
        1   7  25

               Accuracy : 0.887
                 95% CI : (0.8544, 0.9144)
    No Information Rate : 0.8478
    P-Value [Acc > NIR] : 0.009556

                  Kappa : 0.4364

 Mcnemar's Test P-Value : 2.882e-07

            Sensitivity : 0.9821
            Specificity : 0.3571
         Pos Pred Value : 0.8949
         Neg Pred Value : 0.7812
             Prevalence : 0.8478
         Detection Rate : 0.8326
   Detection Prevalence : 0.9304
      Balanced Accuracy : 0.6696

       'Positive' Class : 0
```

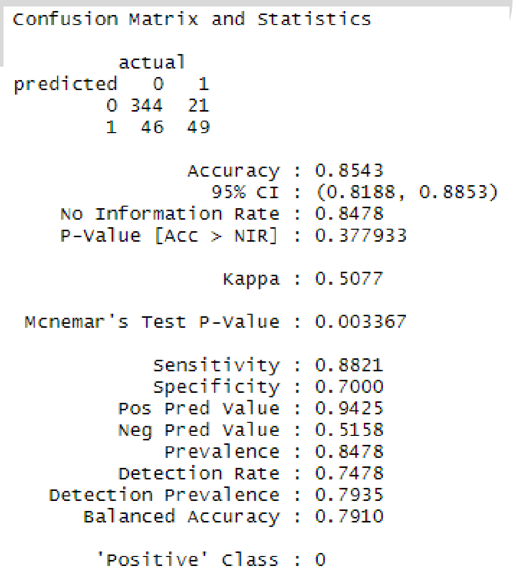
The results seem promising, given that the accuracy is 0.89 and Kappa 0.44. We will discuss those results in detail later.

Task 2. Naïve Bayes classifier

The Naïve Bayes classifier is a simple probabilistic classifier which is based on Bayes theorem. This technique became popular with applications in email filtering, and spam detection, among other similar problems. Despite the naïve design and oversimplified assumptions, this classifier can perform well in many real-world problems.

Fitting a Naïve Bayes classifier using `mlr3` package is rather straightforward. All we technically have to do is to use a different learner, compared to our previous example.

```
# Fit a Naïve Bayes classifier
wine.learner.nb <- lrn("classif.naive_bayes")

wine.learner.nb$train(task = wine.task)

wine.pred.nb <- wine.learner.nb$predict_newdata(newdata = wine.data.test)

nb.results = confusionMatrix(table(predicted = wine.pred.nb$response,

                                   actual = wine.data.test$quality_class))

nb.results
```



```
Confusion Matrix and Statistics

          actual
predicted   0   1
        0 344  21
        1  46  49

               Accuracy : 0.8543
                 95% CI : (0.8188, 0.8853)
    No Information Rate : 0.8478
    P-Value [Acc > NIR] : 0.377933

                  Kappa : 0.5077

 Mcnemar's Test P-Value : 0.003367

            Sensitivity : 0.8821
            Specificity : 0.7000
         Pos Pred Value : 0.9425
         Neg Pred Value : 0.5158
             Prevalence : 0.8478
         Detection Rate : 0.7478
   Detection Prevalence : 0.7935
      Balanced Accuracy : 0.7910

       'Positive' Class : 0
```

Although the accuracy is somewhat lower in this case (compared to KNN), Kappa is considerably higher in the case of Naïve Bayes classifier, having a value of 0.51.

Task 3. Rule-based classifier

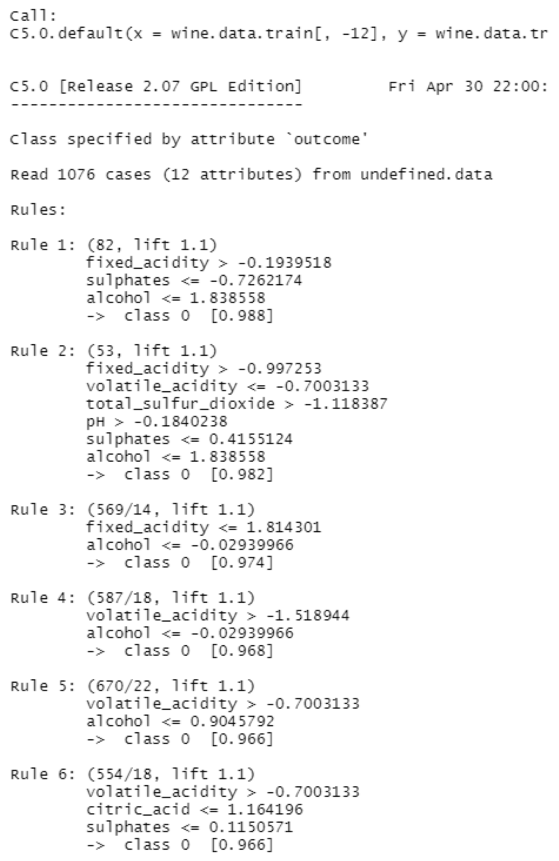
In the previous week, we covered rule-based classifiers in detail. In this task, we will introduce a rather straightforward approach to fitting a rule-based classifier using `C5.0` package.

```
rule.class <- C5.0(x = wine.data.train[,-12], y =
wine.data.train$quality_class, rules = TRUE)
```

That is pretty much everything we need to do.

We can also run a summary on the obtained model to get a list of rules we learned. The output is probably easier to review in your RStudio, so please note that below is displayed only a part of the entire output.

```
summary(rule.class)
```



```
Call:
C5.0.default(x = wine.data.train[, -12], y = wine.data.tr


C5.0 [Release 2.07 GPL Edition]        Fri Apr 30 22:00:
-------------------------------

Class specified by attribute `outcome'

Read 1076 cases (12 attributes) from undefined.data

Rules:

Rule 1: (82, lift 1.1)
        fixed_acidity > -0.1939518
        sulphates <= -0.7262174
        alcohol <= 1.838558
        ->  class 0  [0.988]

Rule 2: (53, lift 1.1)
        fixed_acidity > -0.997253
        volatile_acidity <= -0.7003133
        total_sulfur_dioxide > -1.118387
        pH > -0.1840238
        sulphates <= 0.4155124
        alcohol <= 1.838558
        ->  class 0  [0.982]

Rule 3: (569/14, lift 1.1)
        fixed_acidity <= 1.814301
        alcohol <= -0.02939966
        ->  class 0  [0.974]

Rule 4: (587/18, lift 1.1)
        volatile_acidity > -1.518944
        alcohol <= -0.02939966
        ->  class 0  [0.968]

Rule 5: (670/22, lift 1.1)
        volatile_acidity > -0.7003133
        alcohol <= 0.9045792
        ->  class 0  [0.966]

Rule 6: (554/18, lift 1.1)
        volatile_acidity > -0.7003133
        citric_acid <= 1.164196
        sulphates <= 0.1150571
        ->  class 0  [0.966]
```

Finally, let's run `confusionMatrix()` to get the model parameters.

```
wine.pred.rule <- predict(rule.class, newdata = wine.data.test)
rule.results = confusionMatrix(table(predicted = wine.pred.rule,
                                     actual = wine.data.test$quality_class))

rule.results
```

```
Confusion Matrix and Statistics

         actual
predicted   0   1
        0 372  39
        1  18  31

               Accuracy : 0.8761
                 95% CI : (0.8425, 0.9048)
    No Information Rate : 0.8478
    P-Value [Acc > NIR] : 0.049476

                  Kappa : 0.4524

 Mcnemar's Test P-Value : 0.008071

            Sensitivity : 0.9538
            Specificity : 0.4429
         Pos Pred Value : 0.9051
         Neg Pred Value : 0.6327
             Prevalence : 0.8478
         Detection Rate : 0.8087
   Detection Prevalence : 0.8935
      Balanced Accuracy : 0.6984

       'Positive' Class : 0
```

Which of the classifiers showed the best performance in this case?

Can you comment on the confusion matrix for each of the examples? Which classifier has the lowest number of false positives and false negatives?

Challenge 1. We talked in previous weeks about feature selection. Also, we implemented feature selection in the previous practical.

Can you add feature selection to the pipeline outlined in this practical?

That is, before fitting each of the classifiers, try to apply feature selection, using one of the methods we introduced in previous practicals.