Chapter 7
Lazy Learning: Classification Using Nearest Neighbors

In the next several Chapters, we will concentrate on various progressively advanced machine learning, classification and clustering techniques. There are two categories of learning techniques we wil explore: supervised (human-guided) classification and unsupervised (fully-automated) clustering. In general, supervised *classification* aims to identify or predict predefined classes and label new objects as members of specific classes. Whereas, unsupervised *clustering* attempts to group objects into sets, without knowing a priori labels, and determine relationships between objects.

In the context of machine learning, classification refers to supervised learning and clustering to unsupervised learning.

Unsupervised classification refers to methods where the outcomes (groupings with common characteristics) are automatically derived based on intrinsic affinities and associations in the data without prior human indication of clustering. Unsupervised learning is purely based on input data (X) without corresponding output labels. The goal is to model the underlying structure, affinities, or distribution in the data in order to learn more about its intrinsic characteristics. It is called unsupervised learning because there are no *a priori* correct answers and there is no human guidance. Algorithms are left to their own devises to discover and present the interesting structure in the data. *Clustering* (discovers the inherent groupings in the data) and *association* (discovers association rules that describe the data) represent the core unsupervised learning problems. The **k-means** clustering and the **Apriori association rule** provide solutions to unsupervised learning problems.

Supervised classification methods utilize user provided labels representative of specific classes associated with concrete observations, cases, or units. These training classes/outcomes are used as references for the classification. Many problems can be addressed by decision-support systems utilizing combinations of supervised and unsupervised classification processes. Supervised learning involves input variables (X) and an outcome variable (Y) to learn mapping functions from the input to the output: $Y = f(X)$. The goal is to approximate the mapping function so that when it is applied to new (validation) data (Z) it (accurately) predicts the (expected) outcome variables (Y). It is called supervised learning because the learning process is

© Ivo D. Dinov 2018

I. D. Dinov, *Data Science and Predictive Analytics*,
https://doi.org/10.1007/978-3-319-72347-1_7

Table 7.1 Summary of supervised classification and unsupervised clustering techniques

Inference	Outcome	Supervised	Unsupervised
Classification & prediction	Binary	Classification-rules, OneR, kNN, NaiveBayes, Decision-Tree, C5.0, AdaBoost, XGBoost, LDA/QDA, Logit/Poisson, SVM	*Apriori*, Association-rules, k-Means, NaiveBayes
Classification & prediction	Categorical	Regression modeling & forecasting	*Apriori*, Association-rules, k-Means, NaiveBayes
Regression modeling	Real quantitative	LDA/QDA, SVM, Decision-Tree, NeuralNet	(MLR) Regression modeling, Regression modeling tree, *Apriori*/Association-rules

supervised by initial training labels guiding and correcting the learning until the algorithm achieves an acceptable level of performance.

Regression (output variable is a real value) and *classification* (output variable is a category) problems represent the two types of supervised learning. Examples of supervised machine learning algorithms include *Linear regression* and *Random forest*. Both provide solutions for regression problems, but *Random forest* also provides solutions to classification problems.

Just like categorization of exploratory data analytics (Chap. 4) is challenging, so is systematic codification of machine learning techniques. Table 7.1 attempts to provide a rough representation of common machine learning methods. However, it is not really intended to be a gold-standard protocol for choosing the best analytical method. Before you settle on a specific strategy for data analysis, you should always review the data characteristics in light of the assumptions of each technique and assess the potential to gain new knowledge or extract valid information from applying a specific technique (Table 7.1).

Many of these will be discussed in later Chapters. In this Chapter, we will present step-by-step the *k-nearest neighbor (kNN)* algorithm. Specifically, we will show (1) data retrieval and normalization; (2) splitting the data into *training* and *testing* sets; (3) fitting models on the training data; (4) evaluating model performance on testing data; (5) improving model performance; and (6) determining optimal values of *k*.

In Chap. 14, we will present detailed strategies, and evaluation metrics, to assess the performance of all clustering and classification methods.

7.1 Motivation

Classification tasks could be very difficult when the features and target classes are numerous, complicated, or extremely difficult to understand. In those scenarios where the items of similar class type tend to be homogeneous, nearest neighbor classifying method are well-suited because assigning unlabeled examples to most similar labeled examples would be fairly easy.

Such classification methods can help us to understand the story behind the complicated case-studies. This is because machine learning methods generally have no distribution assumptions. However, this non-parametric manner makes the methods rely heavily on large and representative training datasets.

7.2 The kNN Algorithm Overview

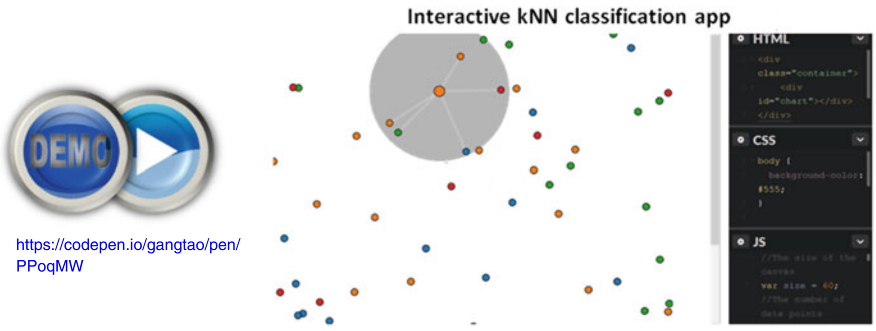
The kNN algorithm involves the following steps:

1. Create a training dataset that has classified examples labeled by nominal variables and different features in ordinal or numerical variables.
2. Create a test dataset containing unlabeled examples with similar features as the training data.
3. Given a predetermined number *k*, match each test record with *k* training records that are "nearest" in similarity.
4. Assigning the class that contains the majority of the *k* training records to the test record.

The Fig. 7.1 demonstration shows the dynamic classification of the mouse location (*x*, *y*) coordinates that are used as new data. You can specify the number of points (*n*) and the number of nearest neighbors (*k*). The app automatically computes the neighborhood size and the corresponding label (color) for the mouse location and draws the connecting edges to the nearest neighbors showing the dynamic classification process.

7.2.1 *Distance Function and Dummy Coding*

How to measure the similarity between records? We can measure the similarity as the geometric distance between the two records. There are many distance functions to choose from. Traditionally, we use *Euclidean distance* as our distance function.


Interactive kNN classification app
DEMO
https://codepen.io/gangtao/pen/PPoqMW
HTML
CSS
JS

```
<div
class="container">
    <div
id="chart"></div>
</div>
```

```
body {
  background-color:
#555;
}
```

```
//The size of the
canvas
var size = 60;
//The number of
data points
```

Fig. 7.1 Live Demo: k-nearest neighbor classification webapp

If we use a line to link the two dots created by the test record and the training record in n dimensional space, the length of the line is the Euclidean distance. Suppose a, b both have n features with coordinates $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$. A simple Euclidian distance could be defined by:

$$dist(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}.$$

When we have nominal features, it requires a little trick to apply the Euclidean distance formula. We could create dummy variables as indicators of the nominal feature. The dummy variable would equal to one when we have the feature and zero otherwise. We show two examples:

$$Gender = \begin{cases} 0 & X = male \\ 1 & X = female \end{cases},$$
$$Cold = \begin{cases} 0 & Temp \geq 37F \\ 1 & Temp < 37F \end{cases}.$$

This allows only binary expressions. If we have multiple nominal categories, just make each one as a dummy variable and apply the Euclidean distance.

7.2.2 *Ways to Determine* k

The parameter k could be neither too large nor too small. If our k is too large, the test record tends to be classified as the most popular class in the training records rather than the most similar one. On the other hand, if the k is too small, outliers or noisy data, like mislabeling the training data, might lead to errors in predictions.

A common practice is to calculate the square root of the number of training examples and use that number as k.

A more robust way would be to choose several k's and select the one with best classifying performance.

7.2.3 *Rescaling of the Features*

Different features might have different scales. For example, we can have a measure of pain scaling from one to ten or one to one hundred. They could be transferred into the same scale. Re-scaling can make each feature contribute to the distance in a relatively equal manner.

7.2.4 Rescaling Formulas

1. *min-max normalization*

$$X_{new} = \frac{X - min(X)}{max(X) - min(X)}.$$

After re-scaling, X_{new} would range between 0 and 1. It measures the distance between each value and its minimum as a percentage. The larger a percentage the further a value is from the minimum. 100% means that the value is at the maximum.

2. *z-Score Standardization*

$$X_{new} = \frac{X - \mu}{\sigma} = \frac{X - Mean(X)}{SD(X)}.$$

This is based on the properties of normal distribution that we have talked about in Chap. 3. After z-score standardization, the re-scaled feature will have unbounded range. This is different from the min-max normalization that has a limited range from 0 to 1. However, after z-score standardization, the new X is assumed to follow a standard normal distribution.

7.3 Case Study

7.3.1 Step 1: Collecting Data

The data we are using for this case study is the "Boys Town Study of Youth Development", which is the second case study, CaseStudy02_Boystown_Data.csv.

Variables:

- **ID**: Case subject identifier.
- **Sex**: dichotomous variable (1 = male, 2 = female).
- **GPA**: Interval-level variable with range of 0–5 (0-"A" average, 1- "B" average, 2- "C" average, 3- "D" average, 4-"E", 5-"F"").
- **Alcohol use**: Interval level variable from 0 to 11 (drink everyday - never drinked).
- **Attitudes on drinking in the household**: Alcatt- Interval level variable from 0 to 6 (totally approve - totally disapprove).
- **DadJob**: 1-yes, dad has a job: and 2- no.
- **MomJob**: 1-yes and 2-no.
- **Parent closeness** (example: In your opinion, does your mother make you feel close to her?)
  - Dadclose: Interval level variable 0–7 (usually-never)
  - Momclose: interval level variable 0–7 (usually-never).

- Delinquency:
  - larceny (how many times have you taken things >$50?): Interval level data 0–4 (never - many times),
  - vandalism: Interval level data 0–7 (never - many times).

7.3.2 Step 2: Exploring and Preparing the Data

First, we need to load in the data and do some data manipulation. We are using the Euclidean distance, so dummy variable should be used. The following code transfers `sex`, `dadjob` and `momjob` into dummy variables.

```
boystown<-read.csv("https://umich.instructure.com/files/399119/download?down
load_frd=1", sep=" ")
boystown$sex<-boystown$sex-1
boystown$dadjob<--1*(boystown$dadjob-2)
boystown$momjob<--1*(boystown$momjob-2)
str(boystown)

## 'data.frame':    200 obs. of  11 variables:
##  $ id        : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ sex       : num  0 0 0 0 1 1 0 0 1 1 ...
##  $ gpa       : int  5 0 3 2 3 3 1 5 1 3 ...
##  $ Alcoholuse: int  2 4 2 2 6 3 2 6 5 2 ...
##  $ alcatt    : int  3 2 3 1 2 0 0 3 0 1 ...
##  $ dadjob    : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ momjob    : num  0 0 0 0 1 0 0 0 1 1 ...
##  $ dadclose  : int  1 3 2 1 2 1 3 6 3 1 ...
##  $ momclose  : int  1 4 2 2 1 2 1 2 3 2 ...
##  $ larceny   : int  1 0 0 3 1 0 0 0 1 1 ...
##  $ vandalism : int  3 0 2 2 2 0 5 1 4 0 ...
```

The `str()` function reports that we have 200 observations and 11 variables. However, the ID variable is not important in this case study so we can delete it. The variable of most interest is GPA. We can classify it into two categories. Whoever gets a "C" or higher will be classified into the "above average" category; Students who have average score below "C" will be in the "average or below" category. These two are the classes of interest for this case study.

```
boystown<-boystown[, -1]
table(boystown$gpa)

##
##  0  1  2  3  4  5
## 30 50 54 40 14 12

boystown$grade<-boystown$gpa %in% c(3, 4, 5)
boystown$grade<-factor(boystown$grade, levels=c(F, T), labels = c("above_avg
", "avg_or_below"))
table(boystown$grade)

##
##    above_avg avg_or_below
##          134           66
```

Let's look at the proportions for the two categorizes.

```
round(prop.table(table(boystown$grade))*100, digits=1)

##    above_avg avg_or_below
##           67           33
```

We can see that most of the students are above average (67%).

The remaining ten features are all numeric but with different scales. If we use these features directly, the ones with larger scale will have a greater impact on the classification performance. Therefore, re-scaling is needed in this scenario.

```
summary(boystown[c("Alcoholuse", "larceny", "vandalism")])

##    Alcoholuse        larceny        vandalism
##  Min.   : 0.00   Min.   :0.00   Min.   :0.0
##  1st Qu.: 2.00   1st Qu.:0.00   1st Qu.:1.0
##  Median : 4.00   Median :1.00   Median :2.0
##  Mean   : 3.87   Mean   :0.92   Mean   :1.9
##  3rd Qu.: 5.00   3rd Qu.:1.00   3rd Qu.:3.0
##  Max.   :11.00   Max.   :4.00   Max.   :7.0
```

7.3.3 Normalizing Data

First let's create a function of our own using the min-max normalization formula. We can check the function using some trial vectors.

```
normalize<-function(x){
     # be careful, the denominator may be trivial!
  return((x-min(x))/(max(x)-min(x)))
}

# some test examples:
normalize(c(1, 2, 3, 4, 5))
## [1] 0.00 0.25 0.50 0.75 1.00

normalize(c(1, 3, 6, 7, 9))

## [1] 0.000 0.250 0.625 0.750 1.000
```

After confirming that it is working properly, we use the `lapply()` function to apply the normalization to each element in a "list." First, we need to make our dataset into a list. The `as.data.frame()` function converts our data into a data frame, which is a list of equal-length column vectors. Thus, each feature is an element in the list that we can apply the normalization function to.

```
boystown_n<-as.data.frame(lapply(boystown[-11], normalize))
```

Let's see one of the features that have been normalized.

```
summary(boystown_n$Alcoholuse)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.1818  0.3636  0.3518  0.4545  1.0000
```

This looks great! Now we can move to the next step.

7.3.4 Data Preparation: Creating Training and Testing Datasets

We have 200 observations in this dataset. The more data we use to train the algorithm, the more precise the prediction would be. We can use 3/4 of the data for training and the remaining 1/4 for testing.

```
# Ideally, we want to randomly split the raw data into training and testing
# For example: 80% training + 20% testing
# subset_int <- sample(nrow(boystown_n), floor(nrow(boystown_n)*0.8))
# bt_train<- boystown_n [subset_int, ]; bt_test<-boystown_n[-subset_int, ]
# Below, we use a simpler 3:1 split for simplicity
bt_train<-boystown_n[1:150, -11]
bt_test<-boystown_n[151:200, -11]
```

The following step is to extract the labels or classes (column = 11, Delinquency in terms of reoccurring `vandalism`) for our two subsets.

```
bt_train_labels<-boystown[1:150, 11]
bt_test_labels<-boystown[151:200, 11]
```

7.3.5 Step 3: Training a Model On the Data

We are using the `class` package for the kNN algorithm in R.

```
#install.packages('class', repos = "http://cran.us.r-project.org")
library(class)
```

The function `knn()` has following components:

```
p<-knn(train, test, class, k)
```

- train: data frame containing numeric training data (features)
- test: data frame containing numeric testing data (features)
- class/cl: class for each observation in the training data
- k: predetermined integer indication the number of nearest neighbors

The first k we chose shoud be the square root of our number of observations: $\sqrt{200} \approx 14$.

```
bt_test_pred<-knn(train=bt_train, test=bt_test, cl=bt_train_labels, k=14)
```

7.3.6 Step 4: Evaluating Model Performance

We utilize the `CrossTable()` function in Chap. 3 to evaluate the kNN model. We have two classes in this example. The goal is to create a 2 × 2 table that shows the matched true and predicted classes, as well as the unmatched ones. However chi-square values are not needed, so we use option `prop.chisq=False' to suppress reporting them.

```
# install.packages("gmodels", repos="http://cran.us.r-project.org")
library(gmodels)
CrossTable(x=bt_test_labels, y=bt_test_pred, prop.chisq = F)

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  50
##
##
##                | bt_test_pred
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           30 |            0 |           30 |
##                |        1.000 |        0.000 |        0.600 |
##                |        0.769 |        0.000 |              |
##                |        0.600 |        0.000 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |            9 |           11 |           20 |
##                |        0.450 |        0.550 |        0.400 |
##                |        0.231 |        1.000 |              |
##                |        0.180 |        0.220 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           39 |           11 |           50 |
##                |        0.780 |        0.220 |              |
## ---------------|--------------|--------------|--------------|
```

From the table, the diagonal first row first cell and the second row second cell contain the counts for records that have predicted classes matching the true classes. The other two cells are the counts for unmatched cases. The accuracy in this case is calculated by: $\frac{cell[1,1]+cell[2,2]}{total}$. This accuracy will vary each time we run the algorithm. In this situation, we got $accuracy = \frac{cell[1,1]+cell[2,2]}{total} = \frac{41}{50} = 0.82$, however, a previous run generated an $accuracy = \frac{cell[1,1]+cell[2,2]}{total} = \frac{38}{50} = 0.76$.

7.3.7 Step 5: Improving Model Performance

The above Normalization may be suboptimal. We can try an alternative standardization method, e.g., standard Z-score centralization and normalization (via `scale()` method). Let's give it a try:

```
bt_z<-as.data.frame(scale(boystown[, -11]))
summary(bt_z$Alcoholuse)

##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.04800 -0.98960  0.06879  0.00000  0.59800  3.77300
```

The `summary()` shows the re-scaling is working properly. Then, we can proceed to next steps (retraining the kNN, predicting and assessing the accuracy of the results):

```
bt_train<-bt_z[1:150, -11]
bt_test<-bt_z[151:200, -11]
bt_train_labels<-boystown[1:150, 11]
bt_test_labels<-boystown[151:200, 11]
bt_test_pred<-knn(train=bt_train, test=bt_test,
                  cl=bt_train_labels, k=14)
CrossTable(x=bt_test_labels, y=bt_test_pred, prop.chisq = F)

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  50
##
##
##                | bt_test_pred
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           30 |            0 |           30 |
##                |        1.000 |        0.000 |        0.600 |
##                |        0.769 |        0.000 |              |
##                |        0.600 |        0.000 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |            9 |           11 |           20 |
##                |        0.450 |        0.550 |        0.400 |
##                |        0.231 |        1.000 |              |
##                |        0.180 |        0.220 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           39 |           11 |           50 |
##                |        0.780 |        0.220 |              |
## ---------------|--------------|--------------|--------------|
```

Under the z-score method, the prediction result is similar to the previous run.

7.3.8 *Testing Alternative Values of* k

Originally, we used the square root of 200 as our *k*. However, this might not be the best *k* in this case study. We can test different *k*'s for their predicting performance.

```
bt_train<-boystown_n[1:150, -11]
bt_test<-boystown_n[151:200, -11]
bt_train_labels<-boystown[1:150, 11]
bt_test_labels<-boystown[151:200, 11]
bt_test_pred1<-knn(train=bt_train, test=bt_test,
                  cl=bt_train_labels, k=1)
bt_test_pred5<-knn(train=bt_train, test=bt_test,
                  cl=bt_train_labels, k=5)
bt_test_pred11<-knn(train=bt_train, test=bt_test,
                  cl=bt_train_labels, k=11)
bt_test_pred21<-knn(train=bt_train, test=bt_test,
                  cl=bt_train_labels, k=21)
bt_test_pred27<-knn(train=bt_train, test=bt_test,
                  cl=bt_train_labels, k=27)
ct_1<-CrossTable(x=bt_test_labels, y=bt_test_pred1, prop.chisq = F)
```

```
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  50
##
##
##                | bt_test_pred1
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           27 |            3 |           30 |
##                |        0.900 |        0.100 |        0.600 |
##                |        0.818 |        0.176 |              |
##                |        0.540 |        0.060 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |            6 |           14 |           20 |
##                |        0.300 |        0.700 |        0.400 |
##                |        0.182 |        0.824 |              |
##                |        0.120 |        0.280 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           33 |           17 |           50 |
##                |        0.660 |        0.340 |              |
## ---------------|--------------|--------------|--------------|
##
```

```
ct_5<-CrossTable(x=bt_test_labels, y=bt_test_pred5,
           prop.chisq = F)
```

```
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
##

##  Total Observations in Table:  50
##
##
##                 | bt_test_pred5
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           30 |            0 |           30 |
##                |        1.000 |        0.000 |        0.600 |
##                |        0.857 |        0.000 |              |
##                |        0.600 |        0.000 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |            5 |           15 |           20 |
##                |        0.250 |        0.750 |        0.400 |
##                |        0.143 |        1.000 |              |
##                |        0.100 |        0.300 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           35 |           15 |           50 |
##                |        0.700 |        0.300 |              |
## ---------------|--------------|--------------|--------------|
##

ct_11<-CrossTable(x=bt_test_labels, y=bt_test_pred11,
           prop.chisq = F)

##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
## Total Observations in Table:  50
##
##                 | bt_test_pred11
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           30 |            0 |           30 |
##                |        1.000 |        0.000 |        0.600 |
##                |        0.769 |        0.000 |              |
##                |        0.600 |        0.000 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |            9 |           11 |           20 |
##                |        0.450 |        0.550 |        0.400 |
##                |        0.231 |        1.000 |              |
##                |        0.180 |        0.220 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           39 |           11 |           50 |
##                |        0.780 |        0.220 |              |
## ---------------|--------------|--------------|--------------|
ct_21<-CrossTable(x=bt_test_labels, y=bt_test_pred21,
           prop.chisq = F)

##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  50
##
##
##                | bt_test_pred21
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           30 |            0 |           30 |
##                |        1.000 |        0.000 |        0.600 |
##                |        0.714 |        0.000 |              |
##                |        0.600 |        0.000 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |           12 |            8 |           20 |
##                |        0.600 |        0.400 |        0.400 |
##                |        0.286 |        1.000 |              |
##                |        0.240 |        0.160 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           42 |            8 |           50 |
##                |        0.840 |        0.160 |              |
## ---------------|--------------|--------------|--------------|
##
##

ct_27<-CrossTable(x=bt_test_labels, y=bt_test_pred27,prop.chisq = F)

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  50
##
##
##                | bt_test_pred27
## bt_test_labels |    above_avg | avg_or_below |    Row Total |
## ---------------|--------------|--------------|--------------|
##      above_avg |           30 |            0 |           30 |
##                |        1.000 |        0.000 |        0.600 |
##                |        0.682 |        0.000 |              |
##                |        0.600 |        0.000 |              |
## ---------------|--------------|--------------|--------------|
##   avg_or_below |           14 |            6 |           20 |
##                |        0.700 |        0.300 |        0.400 |
##                |        0.318 |        1.000 |              |
##                |        0.280 |        0.120 |              |
## ---------------|--------------|--------------|--------------|
##   Column Total |           44 |            6 |           50 |
##                |        0.880 |        0.120 |              |
## ---------------|--------------|--------------|--------------|
```

The choice of k in kNN clustering is very important.

```
# install.packages("e1071")
library(e1071)
knntuning = tune.knn(x= bt_train, y = bt_train_labels, k = 1:30)
knntuning
```

```
##
## Parameter tuning of 'knn.wrapper':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   k
##   9
##
## - best performance: 0.1733333
```

```
summary(knntuning)
```

```
## Parameter tuning of 'knn.wrapper':
## - sampling method: 10-fold cross validation
## - best parameters:
##   k
##   9
## - best performance: 0.1733333
## - Detailed performance results:
##     k     error dispersion
## 1   1 0.2400000 0.08432740
## 2   2 0.2800000 0.10795518
## 3   3 0.2133333 0.10327956
## 4   4 0.2466667 0.04499657
## 5   5 0.1866667 0.10327956
## 6   6 0.1866667 0.11243654
## 7   7 0.1800000 0.10446808
## 8   8 0.1866667 0.12090196
## 9   9 0.1733333 0.10976968
## 10 10 0.2133333 0.12090196
## 11 11 0.2266667 0.12649111
## 12 12 0.2066667 0.11088867
## 13 13 0.2133333 0.11243654
## 14 14 0.2266667 0.13033670
## 15 15 0.2133333 0.12090196
## 16 16 0.2133333 0.09838197
## 17 17 0.2200000 0.10909278
## 18 18 0.2266667 0.11842589
## 19 19 0.2200000 0.10909278
## 20 20 0.2333333 0.11439589
## 21 21 0.2333333 0.11439589
## 22 22 0.2200000 0.08916623
## 23 23 0.2533333 0.10327956
## 24 24 0.2466667 0.10446808
## 25 25 0.2466667 0.11779874
## 26 26 0.2600000 0.11088867
## 27 27 0.2533333 0.11674600
## 28 28 0.2666667 0.10886621
## 29 29 0.2866667 0.11352924
## 30 30 0.2800000 0.11674600
```

It's useful to visualize the `error rate` against the value of *k*. This can help us select a *k* parameter that minimizes the cross-validation (CV) error (Fig. 7.2).

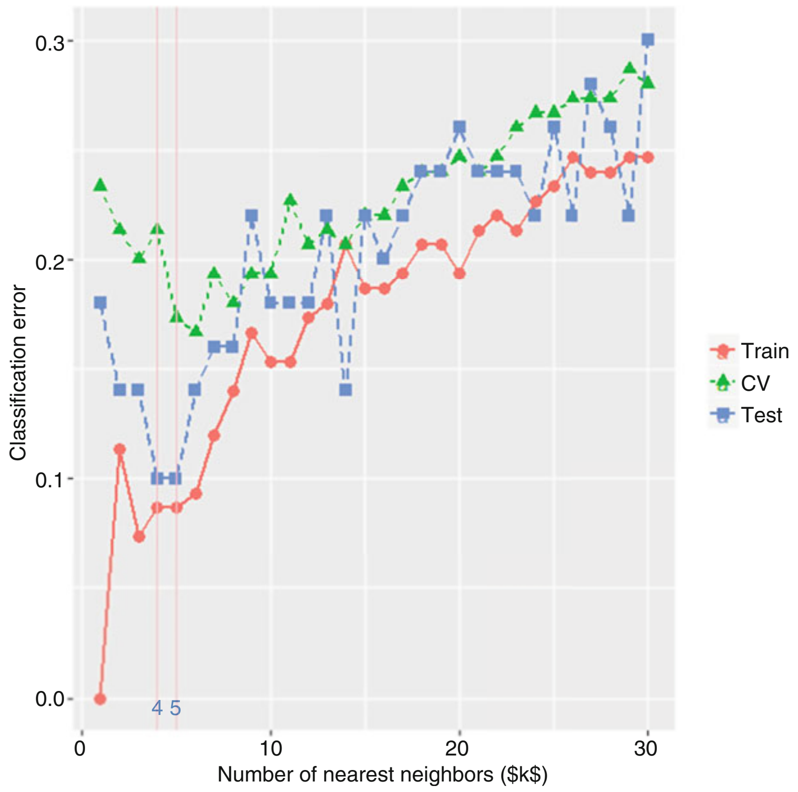


Fig. 7.2 Classification error plots (y-axis) for training data (red), internal statistical cross-validation (green) and external out of box data (blue) against different *k*-parameters of the kNN method

```r
library(class)
library(ggplot2)

# define a function that generates CV folds
cv_partition <- function(y, num_folds = 10, seed = NULL) {
  if(!is.null(seed)) {
    set.seed(seed)
  }
  n <- length(y)

  folds <- split(sample(seq_len(n), n), gl(n = num_folds, k=1, length=n))
  folds <- lapply(folds, function(fold) {
    list(
      training = which(!seq_along(y) %in% fold),
      test = fold
    )
  })
  names(folds) <- paste0("Fold", names(folds))
  return(folds)
}
# Generate 10-folds of the data
folds = cv_partition(bt_train_labels, num_folds = 10)
```

```
# Define a trainingset_CV_error calculation function
train_cv_error = function(K) {
  #Train error
  knnbt = knn(train = bt_train, test = bt_train,
               cl = bt_train_labels, k = K)
  train_error = mean(knnbt != bt_train_labels)

  #CV error
  cverrbt = sapply(folds, function(fold) {
    mean(bt_train_labels[fold$test] != knn(train=bt_train[fold$training,],
cl = bt_train_labels[fold$training], test = bt_train[fold$test,], k=K))
    }
  )

  cv_error = mean(cverrbt)

  #Test error
  knn.test = knn(train = bt_train, test = bt_test,
            cl = bt_train_labels, k = K)
  test_error = mean(knn.test != bt_test_labels)
  return(c(train_error, cv_error, test_error))
}

k_err = sapply(1:30, function(k) train_cv_error(k))
df_errs = data.frame(t(k_err), 1:30)
colnames(df_errs) = c('Train', 'CV', 'Test', 'K')

require(ggplot2)
library(reshape2)

dataL <- melt(df_errs, id="K")
ggplot(dataL, aes_string(x="K", y="value", colour="variable",
   group="variable", linetype="variable", shape="variable")) +
   geom_line(size=0.8) + labs(x = "Number of nearest neighbors ($k$)",
           y = "Classification error",
           colour="", group="",
           linetype="", shape="") +
  geom_point(size=2.8) +
  geom_vline(xintercept=4:5, colour = "pink")+
  geom_text(aes(4,0,label = "4", vjust = 1)) +
  geom_text(aes(5,0,label = "5", vjust = 1))
```

7.3.9 *Quantitative Assessment (Tables 7.2 and 7.3)*

The reader should first review the fundamentals of hypothesis testing inference. Table 7.2 shows the basic components of binary classification, and Table 7.3 reports the results of the classification for several k values.

Table 7.2 Basic evaluation metrics of binary classification

kNN fails to reject	TN	FN
kNN rejects	FP	TP
	Specificity: TN/(TN + FP)	**Sensitivity: TP/(TP + FN)**

Table 7.3 Summary results of the kNN classification for different values of the parameter k

k value	Total unmatched counts	Accuracy
1	9	0.82
5	**5**	**0.90**
11	9	0.82
21	12	0.76
27	14	0.72

Suppose we want to evaluate the kNN model (5) as to how well it predicts the **below-average** boys. Let's report manually some of the accuracy metrics for model5. Combining the results, we get the following sensitivity and specificity:

```
# bt_test_pred5<-knn(train=bt_train, test=bt_test, cl=bt_train_labels, k=5)
# ct_5<-CrossTable(x=bt_test_labels, y=bt_test_pred5, prop.chisq = F)
mod5_TN <- ct_5$prop.row[1, 1]
mod5_FP <- ct_5$prop.row[1, 2]
mod5_FN <- ct_5$prop.row[2, 1]
mod5_TP <- ct_5$prop.row[2, 2]

mod5_sensi <- mod5_TN/(mod5_TN+mod5_FP)
mod5_speci <- mod5_TP/(mod5_TP+mod5_FN)
print(paste0("mod5_sensi=", mod5_sensi))

## [1] "mod5_sensi=1"

print(paste0("mod5_speci=", mod5_speci))

## [1] "mod5_speci=0.75"
```

Therefore, **model5** yields a good choice for the number of clusters $k = 5$. Nevertheless, we can always examine further near 5 to get potentially better choices of k.

Another strategy for model validation and improvement involves the use of the `confusionMatrix()` method, which reports several complementary metrics quantifying the performance of the prediction model.

Let's focus on model5 power to predict `Delinquency` in terms of reoccurring **vandalism**.

```
corr5 <- cor(as.numeric(bt_test_labels), as.numeric(bt_test_pred5))
corr5

## [1] 0.8017837

# plot(as.numeric(bt_test_labels), as.numeric(bt_test_pred5))

# install.packages("caret")
library("caret")

## Loading required package: lattice

# compute the accuracy, LOR, sensitivity/specificity of 3 kNN models

# Model 1: bt_test_pred1
confusionMatrix(as.numeric(bt_test_labels), as.numeric(bt_test_pred1))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2
##          1 27  3
##          2  6 14
##
##                Accuracy : 0.82
##                  95% CI : (0.6856, 0.9142)
##     No Information Rate : 0.66
##     P-Value [Acc > NIR] : 0.009886
##
##                   Kappa : 0.6154
##  Mcnemar's Test P-Value : 0.504985
##
##             Sensitivity : 0.8182
##             Specificity : 0.8235
##          Pos Pred Value : 0.9000
##          Neg Pred Value : 0.7000
##              Prevalence : 0.6600
##          Detection Rate : 0.5400
##    Detection Prevalence : 0.6000
##       Balanced Accuracy : 0.8209
##
##        'Positive' Class : 1
##
# Model 5: bt_test_pred5
confusionMatrix(as.numeric(bt_test_labels), as.numeric(bt_test_pred5))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2
##          1 30  0
##          2  5 15
##
##                Accuracy : 0.9
##                  95% CI : (0.7819, 0.9667)
##     No Information Rate : 0.7
##     P-Value [Acc > NIR] : 0.0007229
##
##                   Kappa : 0.7826
##  Mcnemar's Test P-Value : 0.0736383
##
```

```
##             Sensitivity : 0.8571
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.7500
##              Prevalence : 0.7000
##          Detection Rate : 0.6000
##    Detection Prevalence : 0.6000
##       Balanced Accuracy : 0.9286
##
##        'Positive' Class : 1
##
```

```
# Model 11: bt_test_pred11
confusionMatrix(as.numeric(bt_test_labels), as.numeric(bt_test_pred11))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2
##          1 30  0
##          2  9 11
##
##                Accuracy : 0.82
##                  95% CI : (0.6856, 0.9142)
##     No Information Rate : 0.78
##     P-Value [Acc > NIR] : 0.313048
##
##                   Kappa : 0.5946
##  Mcnemar's Test P-Value : 0.007661
##
##             Sensitivity : 0.7692
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.5500
##              Prevalence : 0.7800
##          Detection Rate : 0.6000
##    Detection Prevalence : 0.6000
##       Balanced Accuracy : 0.8846
##
##        'Positive' Class : 1
##
```

Finally, we can use a 3D plot to display the results of model5 (mod5_TN, mod5_FN, mod5_FP, mod5_TP), Fig. 7.3.

```
# install.packages("scatterplot3d")
library(scatterplot3d)
grid_xy <- matrix(c(0, 1, 1, 0), nrow=2, ncol=2)
intensity <- matrix(c(mod5_TN, mod5_FN, mod5_FP, mod5_TP), nrow=2, ncol=2)

# scatterplot3d(grid_xy, intensity, pch=16, highlight.3d=TRUE, type="h",
main="3D Scatterplot")

s3d.dat <- data.frame(cols=as.vector(col(grid_xy)),
      rows=as.vector(row(grid_xy)),
      value=as.vector(intensity))
scatterplot3d(s3d.dat, pch=16, highlight.3d=TRUE, type="h", xlab="real",
ylab="predicted", zlab="Agreement", main="3D Scatterplot: Model5 Results
(FP, FN, TP, TN)")
```

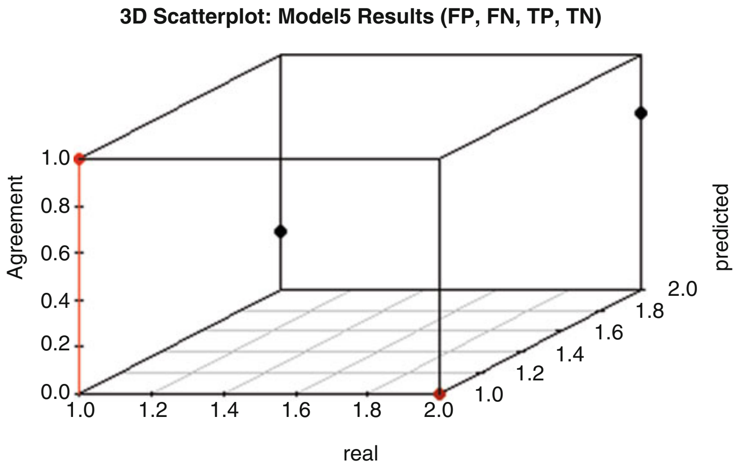


Fig. 7.3 5-NN classification metrics

```
# scatterplot3d(s3d.dat, type="h", lwd=5, pch=" ", xlab="real", ylab="predic
ted", zlab="Agreement", main="Model5 Results (FP, FN, TP, TN)")
```

7.4 Assignments: 7. Lazy Learning: Classification Using Nearest Neighbors

7.4.1 Traumatic Brain Injury (TBI)

Use the kNN algorithm to provide a classification of the data in the TBI case study, (CaseStudy11_TBI). Determine an appropriate *k*, train, and evaluate the performance of the classification model on the data. Report some model quality statistics for a couple of different values of *k* and use these to rank-order (and perhaps plot the classification results of) the models.

7.4.2 Parkinson's Disease

Use 05_PPMI_top_UPDRS_Integrated_LongFormat1 data to practice KNN classification.

7.4.3 KNN Classification in a High Dimensional Space

- **Preprocess the data**: delete the `index` and `ID` columns; convert the response variable `ResearchGroup` to binary 0-1 factor; detect `NA` (missing) values (impute if necessary)
- **Summarize the dataset**: use `str`, `summary`, `cor`, `ggpairs`
- **Scale/Normalize the data**: As appropriate, scale to [0, 1]; transform $log(x + 1)$; discretize (0 or 1), etc.
- **Partition data into training and testing sets**: use `set.seed` and random `sample`, train:test = 2:1
- **Select the optimal** k **for each of the scaled data**: Plot an error graph for k, including three lines: training_error, cross-validation error, and testing error, respectively
- **What is the impact of** k**?** Formulate a hypothesis about the relation between k and the error rates. You can try to use `knn.tunning` to verify the results (Hint: select the same folds, all you may obtain a result slightly different)
- **Interpret the results**: Hint: Considering the number of dimension of the data, how many points are necessary to obtain the same density result for 100 dimensional space compared to a 1 dimensional space?
- **Report the error rates** for both the training and the testing data. What do you find?

7.4.4 KNN Classification in a Lower Dimensional Space

Try all the above again but select only the variables: `UPDRS_Part_I_Summary_Score_Baseline`, `UPDRS_Part_I_Summary_Score_Month_24`, `UPDRS_Part_II_Patient_Questionnaire_Summary_Score_Baseline`, `UPDRS_Part_II_Patient_Questionnaire_Summary_Score_Month_24`, `UPDRS_Part_III_Summary_Score_Baseline`, `UPDRS_Part_III_Summary_Score_Month_24`, as predictors. Now, what about the specific k you select and the error rates for each kind of data (original data, normalized data, log-transformed data, and binary data). Comment on any interesting observations.

References

Kidwell , David A. (2013) *Lazy Learning*, Springer Science & Business Media, ISBN 9401720533, 9789401720533

Interactive kNN webapp: https://codepen.io/gangtao/pen/PPoqMW

Aggarwal, Charu C. (ed.) (2015) *Data Classification: Algorithms and Applications*, Chapman & Hall/CRC, ISBN 1498760589, 9781498760584