Spring 2018 IS451B Lab 1

TA: Zhijin Zhou

4/12/2018

In class Practice: Sales of Riding Mowers

Step 1: read the data

The first step is to load up and view the data. We can use `View` function to view data in a seperate window.

```
# read and view data
mower.df <- read.csv("RidingMowers.csv")
```

We can use function `names` to print out the names of the variables that we have in our dataset.

```
names(mower.df)
```

```
## [1] "Income"    "Lot_Size"  "Ownership"
```

Step 2: Partition data

Before builing up the model, we always need to partition our data into training and validation set. To do this, we need to randomly select 60% out of all the observations to be included in to the training set and 40% our of all the observations.

We can call function `nrow` to get the total number of observations in our data.

```
nrow(mower.df)
```

```
## [1] 24
```

Then we sample 60% of all the observations into training set.

```
set.seed(111)
# partition data
train.index <- sample(1:nrow(mower.df), 0.6*nrow(mower.df))
train.df <- mower.df[train.index, ]
valid.df <- mower.df[-train.index,]
```

```
new.df <- data.frame(Income = 60, Lot_Size = 20)
```

Step 2: Normalizing data

The second step is to normalize the data. Why do we need to do this?

The k-nearest neighbor algorithm relies on majority voting based on class membership of 'k' nearest samples for a given test point. The nearness of samples is typically based on Euclidean distance.

Suppose you had a dataset, and all but one feature dimension had values strictly between (0,1), while a single feature dimension had values that range between (-1000000,1000000). When taking the euclidean distance between pairs of "examples", the values of the feature dimensions that range between 0 and 1 may become *uninformative* and the algorithm would essentially rely on the single dimension whose values are substantially larger, leading to incorrect classification.(!)

Therefore, to avoid this miss classification, the second step we need to do is to *normalize* the feature variables.

To do this, we first need to make copies of the original dataset. We will do the normalization on the copies.

```
# initialize normalized training, validation data, complete data frames to originals
train.norm.df <- train.df
valid.norm.df <- valid.df
mower.norm.df <- mower.df
```

We can use `preProcess` function in the `caret` package to do the normalization. First let's load the library. If the package is not installed, run `install.packages("caret", dependencies = TRUE)` to install it.

```
# use preProcess() from the caret package to normalize Income and Lot_Size.
# install.packages("caret", dependencies = TRUE)
library(caret)
```

```
## Loading required package: lattice

## Loading required package: ggplot2

## Warning in as.POSIXlt.POSIXct(Sys.time()): unknown timezone 'zone/tz/2018c.
## 1.0/zoneinfo/America/Los_Angeles'
```

Call `preProcess` function to do the normalization.

```
norm.values <- preProcess(train.df[, 1:2], method=c("center", "scale"))
train.norm.df[, 1:2] <- predict(norm.values, train.df[, 1:2])
valid.norm.df[, 1:2] <- predict(norm.values, valid.df[, 1:2])
mower.norm.df[, 1:2] <- predict(norm.values, mower.df[, 1:2])
new.norm.df <- predict(norm.values, new.df)
```

Step 3: KNN classification

Now we are ready to do the KNN classification. We can easily do this by calling function `knn` in pacakge `FNN`. First load the library. Make sure you have installed the package before you load it.

```
#install.packages("FNN")
library(FNN)
```

```
accuracy.df <- data.frame(k = 1:10, accuracy = rep(0, 10))
# compute knn for different k on validation.
for(i in 1:10) {
  # use ?knn to find out more information
  # It worth noting that the input argument cl must be a factor!
  knn.pred <- knn(train.norm.df[, 1:2], valid.norm.df[, 1:2],
                  cl = train.norm.df[, 3], k = i)
  confusionMatrix(knn.pred, valid.norm.df[, 3])
  accuracy.df[i, 2] <- confusionMatrix(knn.pred, valid.norm.df[, 3])$overall[1]
}
accuracy.df
```

```
##     k accuracy
## 1   1      0.7
## 2   2      0.7
## 3   3      0.8
## 4   4      0.9
## 5   5      0.8
## 6   6      0.9
## 7   7      0.9
## 8   8      1.0
## 9   9      0.9
## 10 10       0.9
```