Notes for Tutorial 5 (ML techniques for prediction)

Starting from Lecture 5 onwards, we introduce basic machine learning (ML) techniques for prediction, classification and clustering. Tutorials 5-6 cover prediction two methods:

- k-nearest neighbours (kNN) regression
- linear regression models

included in Lecture 5 and their implementation in R. Packages **"MASS"** and **"ggplot2"** are required.

After this tutorial, you expected to be able to explain at which circumstances the kNN is applied, describe the pros and cons of the method; reproduce what has been demonstrated in the tutorial, interpret the outputs.

1. K-nearest neighbors regression

KNN modeling is conceptually one of the simplest machine learning methods. In order to predict a continuous outcome value for given new observation (new_obs) of predictors, the KNN algorithm computes the (**inverse distance weighted**) **average outcome** value of the k training observations (i.e., neighbors) that are the most similar to the new_obs, and returns this value as the predicted outcome value of the new_obs.

- **Similarity measures**:

In the kNN algorithm, the similarity between observations is generally measured by Euclidean distance measure, which is very sensitive to the scale on which predictor variable measurements are made. It's generally recommended to normalize the predictor variables to make their scales comparable.

kNN can be applied for both regression prediction and classification (to be explained in Lecture 6) problems. It is widely used due to its ease of interpretation and low computational time.

2. How to predict a continuous variable using KNN?

In general, there are 7 steps in a procedure of prediction (either using kNN or linear regression):
Step 1: Prepare the data
Step 2: Check the data and calculating the data summary
Step 3: Split the data into training and test data sets
Step 4: Write the function (model formula) and fit the model on the training set
Step 5: Evaluate the model
Step 6: Interpret the outputs.

Let's look at the Boston data set, which contains information collected by the U.S Census Service about housing in the area of Boston Mass. It is available in R package "MASS". Comparing with flights dataset, this dataset is small in size with only 506 cases and 14 variables:

- CRIM - per capita crime rate by town
- ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS - proportion of non-retail business acres per town.
- CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- NOX - nitric oxides concentration (parts per 10 million)
- RM - average number of rooms per dwelling
- AGE - proportion of owner-occupied units built prior to 1940
- DIS - weighted distances to five Boston employment centres
- RAD - index of accessibility to radial highways
- TAX - full-value property-tax rate per $10,000
- PTRATIO - pupil-teacher ratio by town
- B - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT - % lower status of the population
- MEDV - Median value of owner-occupied homes in $1000's

We focus on the **housing price**, which was defined by the median value (mdev) of a home, and aim to predict the median house value in Boston Suburbs using various predictor variables. The dataset is clean, we may ignore step 1 in this example.

```
# Load the data
data("Boston", package = "MASS")
# Inspect the data and summary
head(Boston)
dim(Boston)
summary(Boston)
```

Split the data into training and test sets

80/20 is a commonly used splitting rule, which randomly selects 80% of data into the training set for building a predictive model and treats the remaining 20% of data as a test set for evaluating the model. Use R base function **sample()** to randomly draw a sample of the specified size from the elements of x without replacement (by default).

Syntax:

```
    sample(x, size, replace = FALSE, prob = NULL)
```

Make sure to set seed for reproducibility.

```
set.seed(100)
training.idx <- sample(1: nrow(Boston), nrow(Boston)*0.8)
train.data  <- Boston[training.idx, ]
test.data <- Boston[-training.idx, ]
```

Fit the model

Use the function train() in package caret. Syntax

```
train(formula, data, method = "knn",
. . ., trControl = trainControl(),
preProcess = c("center","scale"),
 tuneLength = 3)
```

where

- preProcess, to normalize the data as the distance measure in kNN is sensitive to predictors' scale.
- trControl, a list of values to control parameters for training model using resampling methods. (A heuristically optimal number *k* of nearest neighbors based on RMSE. This is done using cross validation). For example, we set up 10-fold cross validation (cv).
- tuneLength, to specify the number of possible k values to evaluate (select the best one among them)

"cv" is a method to let train() select the best value of k among all possible k values specified in tuneLength. See more details at
https://www.rdocumentation.org/packages/caret/versions/4.47/topics/train.

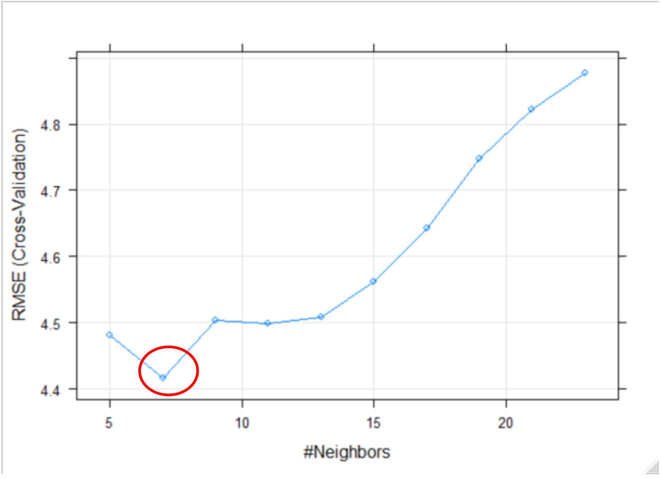
```
# Fit the model on the training set
library(caret)
set.seed(101)
model <- train(
  medv~., data = train.data, method = "knn",
  trControl = trainControl("cv", number = 10),
  preProcess = c("center","scale"),
  tuneLength = 10
)
```

In general, a formula for a model is written as y~x1+x2+…. Here, the outcome variable y is medv, and "." after "~" refers to that all variables other than medv contained in data will be used as predictor variables in regression.

Select value of k

The best k is the one that minimize the prediction error RMSE (root mean squared error). The RMSE corresponds to the square root of the average difference between the observed known outcome values $y = (y_1, \cdots, y_n)^T$ and the corresponding predicted values $\hat{y}$, RMSE = sqrt($(y - \hat{y})^T(y - \hat{y})$/n). The lower the RMSE, the better fit of the model. Train() in the above code chooses a value of k that minimizes the cross-validation RMSE.

```
# Plot model error RMSE vs different values of k
plot(model)
```



```
# Best tuning parameter k that minimize the RMSE
model$bestTune
  k
2 7
```

3. Assess prediction performance of the kNN regression on the test data set

```
predictions <- predict(model, test.data)
head(predictions)
[1] 30.08571 18.61429 19.35714 16.84286 15.20000 17.00000
```

where function predict() obtains predictions and optional estimates of those predictions from a fitted model object. Syntax:

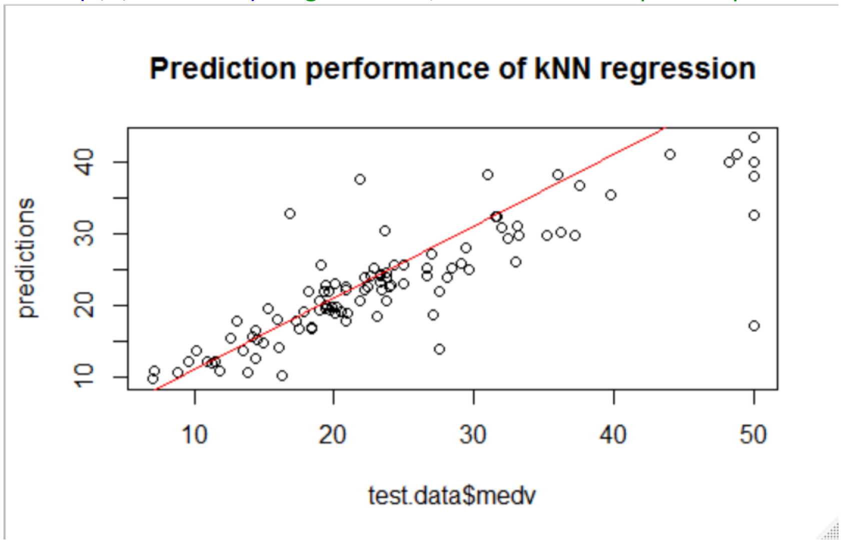
```
predict(object, newdata = NULL,...)
```

Compute the prediction error RMSE:

```
RMSE(predictions, test.data$medv)
[1] 5.714944
```

To visualize the performance of kNN reg, we plot predicted medv vs medv in the test data.

```
plot(test.data$medv, predictions,main="Prediction performance of kNN regression")
#add a reference line x=y
abline(0,1, col="red") #arguments 0,1 refer to intercept & slope of the line
```



It shows that most data points are close to the 45 degree line, implying a very good prediction.

4. Is it sufficient to use only one predictor age to predict medv?

To answer this question, we perform kNN regression with an updated model formula "medv~age" in train(). Results show that the prediction error RMSE=10.26, which is much bigger than that in the kNN regression with all possible explanatory variables. So the prediction using a single explanatory variable is obviously insufficient to provide a good prediction.

To summary, issues have to be considered when using kNN:

- Normalize the data when performing the kNN analysis.

- Test multiple k-values to decide an optimal value for your data. This can be done automatically using train() in the caret package.
- Assess the model performance on test data.

- **Pros**: pretty intuitive and simple; non-parametric algorithm without assumptions. Parametric models like linear regression have lots of assumptions to be met by data before it can be implemented.
- **Cons: no explicit model** (we are not able to interpret the effects of predictor variables on the outcome); not good prediction with high-dim or binary predictors