MODULE 2 UNIT 1
Lesson Video 1 Transcript

IN COLLABORATION WITH getsmarter™

Module 2 Unit 1 Lesson Video 1 Transcript

Fitting KNN and polynomial regression onto data

YINING CHEN: In this video, you will learn about the fitting of two different models to data, and evaluating their performance. The two models that will be explored are the *k*-nearest neighbour classifier and polynomial regression.

Section A: K-nearest neighbour classifier

CHEN: In this example, we will apply the KNN classifier to the MNIST data set.

```
In [ ]: mnist <- readRDS("MNIST.rds")
```

CHEN: The MNIST data set contains a set of handwritten digits between 0 and 9. The purpose of using KNN here is to accurately predict the actual digit based on the handwritten digit.

To start off with, the data is loaded into R. As the original MNIST data does not come as a CSV file, it has already been saved as an R object for our convenience. This R object can be loaded using the “readRDS” function.

```
In [1]: mnist <- readRDS("MNIST.rds")
```

CHEN: With the data loaded, you can now explore the structure of the data using the “str” function.

```
In [2]: str(mnist)
List of 3
 $ n: int [1:5000] 60000 NA NA NA NA NA NA NA NA NA ...
 $ x: int [1:5000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
 $ y: int [1:5000] 5 0 4 1 9 2 1 3 1 4 ...
```

CHEN: You will see that the data is a list object with three items: *n* is the number of the observations, *x* is the value of each of the 784 pixels that make up the image, and *y* is the actual digit represented. Let’s now split object into two variables using the dollar sign operator.

```
In [3]: mnist_train_x <- mnist$x
        mnist_train_y <- mnist$y
```

CHEN: With the data now loaded and split into variables, relevant R packages should be loaded. The package we used for KNN modelling is called caret, which stands for

classification and regression training. For your information, it is possible to write one's own custom R functions, save it as a separate script, and load it in using the "source" function.

For KNN, some standard data pre-processing might make it work better. This step is optional in this example. The details of this part falls outside of the scope of this course, so we just mention it briefly. In essence, our aim here is get rid of the predictors that have variance exactly or very close to zero, such as those that represent the pixels near the corners of the images. The function that does this pre-processing has been saved in the "KNN Function.R" script, which is also now loaded into R.

```
In [*]: library(caret)
        source("KNN Function.R")
```

```
Loading required package: lattice

Loading required package: ggplot2
```

CHEN: With the necessary data and functions loaded, we then begin on preparing the data for it to be used for the KNN classifier. As mentioned earlier, as an optional step, the data pre-processing function, called "data_preprocessing", requires the image data as an input, which then produces an output of pre-processed data. In addition, as this is a classification problem, the actual digits that will be predicted need to be in factor format, and not integer format. This transformation is done by using the "as.factor" function.

```
In [5]: mnist_train_x <- data_preprocessing(mnist_train_x)
        mnist_train_y <- as.factor(mnist_train_y)
        mnist_train_x
```

CHEN: With the data now ready, we use the "knn3" function to estimate the KNN model. The first input required is the pre-processed image data, the second input is the actual digits, and finally, the number of nearest neighbours – *k* – needs to be chosen. For this first example, we will use "*k* = 5".

```
In [6]: set.seed(1)
        knn_model <- knn3(mnist_train_x,
                          mnist_train_y,
                          k = 5)
```

CHEN: The model is estimated and now we need to see how well it performs on the training data. To do this, predictions need to be made using the "predict" function.

```
In [*]: knn_predict <- predict(knn_model, mnist_train_x, type = "class")
```

CHEN: Once this is done, a confusion matrix is generated using the "cm" function. This produces the output, along with some additional statistics.

```
In [8]: confusionMatrix(knn_predict, mnist_train_y)
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1   2   3   4   5   6   7   8   9
         0 476   0   2   0   0   0   3   0   0   2
         1   1 559   8   2  10   1   3   7   8   3
         2   0   2 464   2   0   2   0   0   5   1
         3   1   0   1 469   0   4   0   0   8   3
         4   0   1   0   0 505   0   0   0   1   3
         5   1   1   0   6   0 415   3   0   7   0
         6   0   0   3   2   7   7 492   0   5   4
         7   0   0  10   6   1   2   0 535   1   9
         8   0   0   0   1   0   0   0   0 421   1
         9   0   0   0   5  12   3   0   8   6 469

Overall Statistics

               Accuracy : 0.961
                 95% CI : (0.9553, 0.9662)
    No Information Rate : 0.1126
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9566

 Mcnemar's Test P-Value : NA
```

CHEN: The overall statistics show that the model predicts the correct values on the diagonals, most of the time, with an accuracy of around 96%.

To see how this accuracy rate changes with a different *k*, the same steps as in the first example are repeated, but with the value of *k* decreased to 3.

```
In [9]: set.seed(1)
        knn_model <- knn3(mnist_train_x,
                          mnist_train_y,
                          k = 3)
        knn_predict <- predict(knn_model, mnist_train_x, type = "class")
        confusionMatrix(knn_predict, mnist_train_y)
```

CHEN: From the overall statistics generated here, it is evident that the model accuracy has now increased to 97%. This indicates that the model now fits the data even better than before.

```
Overall Statistics

               Accuracy : 0.9718
                 95% CI : (0.9668, 0.9762)
    No Information Rate : 0.1126
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9686
```

CHEN: However, in this example, we build and evaluate the model based on the same data set, so these accuracy rates might be misleading as a result of overfitting. This issue can be rectified using cross-validation, which we will discuss in the next unit.

Our ultimate goal here is to find the value of *k* where the model performs at an optimal level, avoiding overfitting of the data that leads to inaccurate results.

Section B: Polynomial regression

CHEN: In this section, a polynomial regression curve is fitted onto a simulated data set. For this example, the simulated data has been created and is saved as a CSV file titled "Polynomial Data". This data set is loaded into R using the "read.csv" function.

```
In [10]: data <- read.csv("Polynomial Data.csv")
```

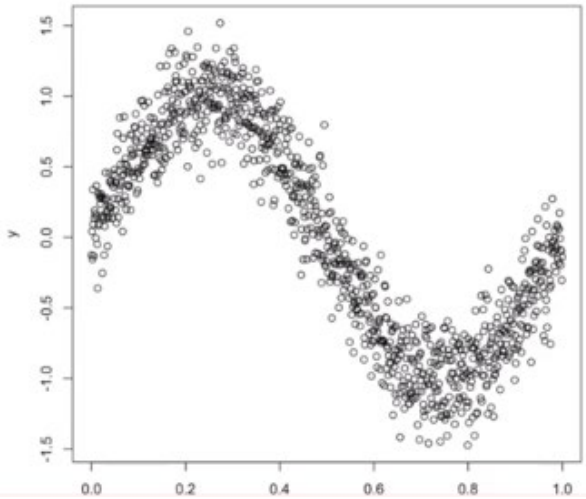
CHEN: With the data loaded into R, the structure of the data can be analysed by making use of the "str" function. Once this function is applied, it becomes clear that the data is a data frame containing two numeric variables, X and Y.

```
In [11]: str(data)

'data.frame':   1000 obs. of  2 variables:
 $ X: num  0 0.001 0.002 0.003 0.004 ...
 $ Y: num  -0.1253 0.043 -0.1545 0.3379 0.0911 ...
```

CHEN: X being the predictor variable used to make a prediction, Y is the actual value with which our prediction should be compared to.

From the data frame, we then extract two variables, X and Y. These variables can now be used to generate a scatter plot from the data frame.

```
In [12]: x <- data$X
         y <- data$Y

         plot(x, y)
```



CHEN: To create polynomial data, the ISLR package must first be loaded into R.

```
In [ ]: library(ISLR)
```

CHEN: This package allows for the creation of polynomial variables. Once the package has been loaded, the “poly” function is used to create the polynomial variables, passing in the X data, and indicating the order of the polynomial required. In this example, the chosen order of the polynomial will be 3.

```
In [ ]: poly_x <- poly(x, 3)
```

CHEN: The created polynomial variables are then fitted onto the Y data by making use of the “lm” function, standing for linear models. We will discuss linear regression in more detail in Module 3. Since it is important to retain the equation format, the formula used for this action is “Y equals the polynomial X data”.

```
In [ ]: poly_reg <- lm(y~poly_x)
```

CHEN: To determine how well the polynomial data X fits the Y data, a summary of the regression model is generated by making use of the “summary” function.

```
In [16]: summary(poly_reg)

Call:
lm(formula = y ~ poly_x)

Residuals:
     Min       1Q   Median       3Q      Max
-0.68342 -0.14911  0.00298  0.14833  0.77690

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.002330   0.006974  -0.334    0.738
poly_x1     -17.571200   0.220536 -79.675   <2e-16 ***
poly_x2       0.218223   0.220536   0.990    0.323
poly_x3      13.909529   0.220536  63.071   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.2205 on 996 degrees of freedom
Multiple R-squared:  0.912,     Adjusted R-squared:  0.9118
F-statistic:  3442 on 3 and 996 DF,  p-value: < 2.2e-16
```

CHEN: In this example, it can be seen that the regression model has an R-squared of around 91%. This means that 91% of the total variability is explained by this regression model. Also, the root mean squared error of this model – or displayed as the “Residual standard error” in R – is 0.22. This is the quantity that indicates the magnitude of the unexplained variability of the data by this model.

As was done in the KNN example shown in Section A, all the previous steps are now repeated, this time choosing the polynomial order of 5.

```
In [17]: poly_x <- poly(x, 5)
         poly_reg <- lm(y~poly_x)
         summary(poly_reg)

         Call:
         lm(formula = y ~ poly_x)

         Residuals:
              Min       1Q   Median       3Q      Max
         -0.59420 -0.13456 -0.00679  0.13631  0.77367

         Coefficients:
                      Estimate Std. Error t value Pr(>|t|)
         (Intercept)  -0.002330   0.006549  -0.356    0.722
         poly_x1     -17.571200   0.207107 -84.841   <2e-16 ***
         poly_x2       0.218223   0.207107   1.054    0.292
         poly_x3      13.909529   0.207107  67.161   <2e-16 ***
         poly_x4      -0.055613   0.207107  -0.269    0.788
         poly_x5      -2.408865   0.207107 -11.631   <2e-16 ***
         ---
         Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

         Residual standard error: 0.2071 on 994 degrees of freedom
         Multiple R-squared:  0.9226,    Adjusted R-squared:  0.9222
         F-statistic:  2369 on 5 and 994 DF,  p-value: < 2.2e-16
```

CHEN: Once these steps are executed, the summary statistics show that the R-squared increases to around 92%, while the root mean squared error decreases to around 0.21, implying that the model with a higher polynomial order fits the data slightly better.

It must be kept in mind that, as the order of polynomial increases, so does the computational power needed to fit the model onto the data. It is therefore important to find the optimal order of the polynomial to ensure the best model fit, bearing the increase in computational power in mind. More importantly, you should also remain extremely mindful of overfitting the model. The concept of overfitting will be discussed in more detail at a later stage.

This video explored two examples of both the KNN and polynomial regression models.

Conclusion

CHEN: You will have the opportunity to practice these steps later in this module when you engage with the content of the IDE Activity.