THE LONDON SCHOOL OF ECONOMICS AND POLITICAL SCIENCE

MODULE 2 UNIT 2
Lesson Video 1 Transcript

Module 2 Unit 2 Lesson Video 1 Transcript

Apply KNN and polynomial regression in R

Introduction

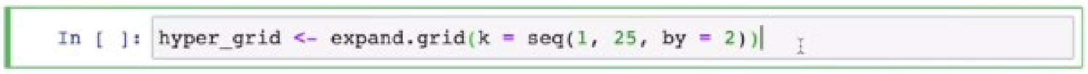
YINING CHEN: In this screencast, the machine learning pipeline is applied to optimise the performance of a KNN classifier and the polynomial regression, respectively.

Section A: KNN classifier

CHEN: The KNN classifier will be applied to the MNIST data set used in Unit 1. To prevent unnecessary repetition from the steps explored previously, the data from the previous environment is loaded into R using the “load” function. Then the caret package is loaded, which is necessary for cross-validation and estimation across tuning parameters.

Step 1: Identify the models and their tuning parameters

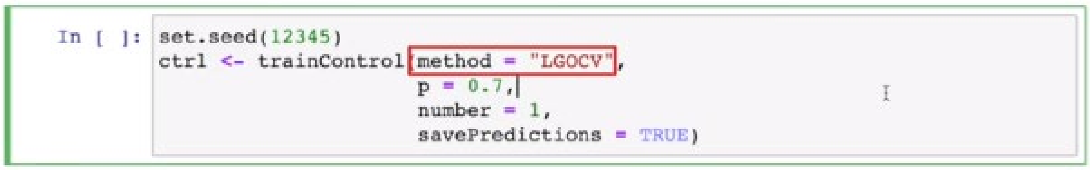
CHEN: At this point, after the MNIST data and caret package has been loaded into R, a model needs to be identified. As explained in Unit 1, due to the nature of pixel positioning of the digits within the grid a KNN classifier can be used to generate an output from the image input. After model has been chosen, the corresponding tuning parameter must be chosen as well. With the KNN classifier, a choice of k is needed. Here, odd numbers are generally used to prevent the likelihood of a tie between points. Therefore, we'll use the values of k the odd numbers between 1 and 25. To do this, we first used the “seq” function to generate a number sequence and then apply the “expand.grid” function to create a corresponding data frame.



```
In [ ]: hyper_grid <- expand.grid(k = seq(1, 25, by = 2))
```

Step 2: Apply cross-validation to estimate the test errors

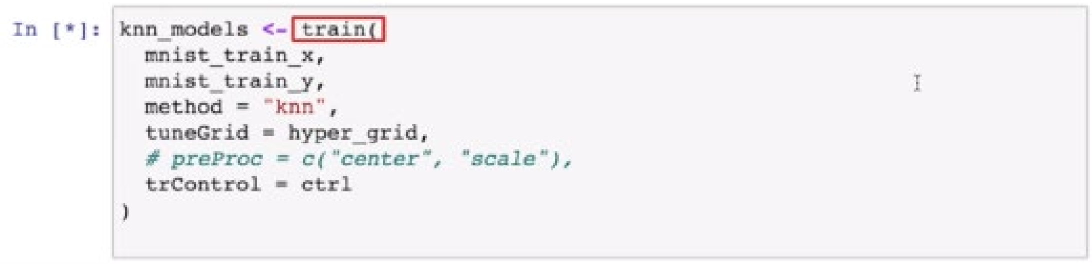
CHEN: With the appropriate model chosen and the plausible tuning parameter values in place, the cross-validation environment is set up. For purposes of this example, leave-one-group-out cross-validation is used without any repeats. This is one of many types of cross-validation that can be applied when training a model. The training percentage chosen for this example is 70%, meaning that 70% of data will be randomly selected for the purpose of training in the model while the remaining 30% will be used solely for the purpose of evaluating or testing, which here is the calculation of the prediction accuracy. Know that here we only split the data randomly into training and testing sets once. Though in practice, this process can be repeated many times before we aggregate the results to make things more stable. Everything can be done using the caret “trainControl” function here.



```
In [ ]: set.seed(12345)
        ctrl <- trainControl(method = "LGOCV",
                             p = 0.7,
                             number = 1,
                             savePredictions = TRUE)
```

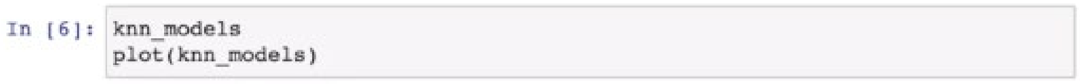
Step 3: Choose the optimal model or tuning parameter

CHEN: Now that Steps 1 and 2 are complete, the model is ready to be trained across various tuning parameters. The training purpose is done by applying caret’s “train” function, which is used to estimate the model across different tuning parameter values.



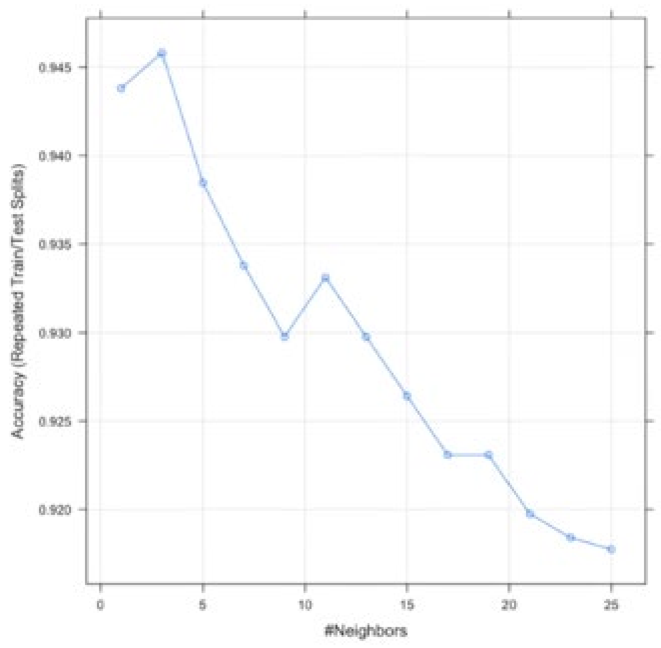
```
In [*]: knn_models <- train(
          mnist_train_x,
          mnist_train_y,
          method = "knn",
          tuneGrid = hyper_grid,
          # preProc = c("center", "scale"),
          trControl = ctrl
        )
```

CHEN: Take note that due to the computational intensity of this process, it may take some time to complete the run. Now that the model has been trained, an output is generated. First, a summary of the various accuracy rates across different tuning parameters is viewed using the “knn_models” function.



```
In [6]: knn_models
        plot(knn_models)
```

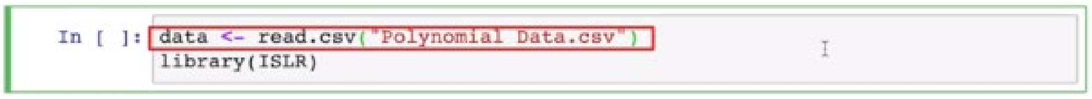
CHEN: These accuracies across different values of k can be plotted in order to visually explore the results.



CHEN: In plotting, the accuracy rates of the models across different values of k, it appears that the value of k equals three is the optimal parameter, with an accuracy of approximately 93 to 94%. Actually, k equals one seems to perform reasonably well too in this example.

Section B: Polynomial regression

CHEN: Like in the KNN example above, we will be using the same data set as in the previous unit. Before applying the steps of the machine learning pipeline to the data, the CSV file is loaded into R. As in Section A, the relevant caret and ISLR package are also loaded for cross-validation and estimation across the tuning parameters.



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

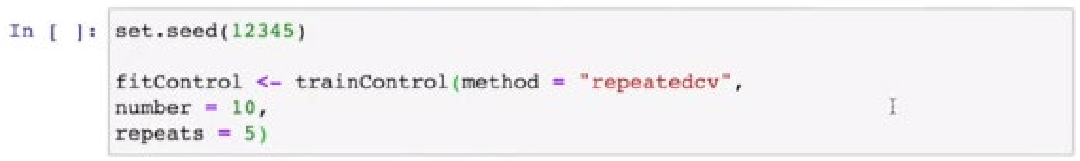
CHEN: Once the data and the packages have been loaded, the steps of the machine learning pipeline can be applied to the data.

Step 1: Identify the models and their tuning parameters

CHEN: As you may recall from Unit 1, the nature of the simulated data used for this example is suitable for the application of the polynomial regression model due to the curve it creates when plotted on a graph. The tuning parameter to be chosen for this example will be that of the polynomial order, which we should take values between 1 and 20. When polynomial regression is applied to data, it is important to note that as the order of polynomial increases, the root mean square errors of the model tends to drop if you were to build and evaluate the model on the same data, which could result in overfitting.

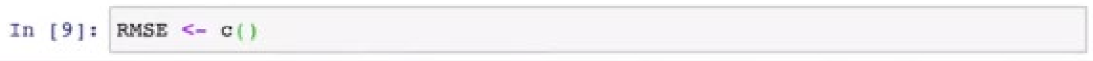
Step 2: Apply cross-validation to estimate the test errors

CHEN: In this example, we are going to use repeated k-fold cross-validation to estimate the test error. Like in the previous example, this is done with the caret's "trainControl" function.



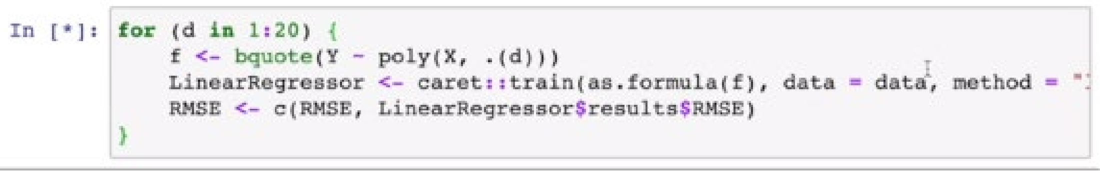
```
In [ ]: set.seed(12345)

        fitControl <- trainControl(method = "repeatedcv",
        number = 10,
        repeats = 5)
```

CHEN: Then an object for the root mean square error is created.



```
In [9]: RMSE <- c()
```

CHEN: Unfortunately, caret does not provide a function to estimate across different tuning parameters for polynomial regression. So, it is manually done using a for loop. A for loop essentially repeats the same process chosen number of times, 20 times in this case. Caret's "train" function is used to generate a new formula for each loop. So, the "bquote" function is combined with the "poly" function used when this example was illustrated in Unit 1. Essentially, this tells caret to train the model on polynomial data of a growing order. This is done by increasing the order parameter in "poly", the function that creates the polynomial data by i. The number of times the loop has already run. The measure of accuracy should also be stored – in this case, the root mean square error – by creating a variable for it and then appending it with the "c" function.

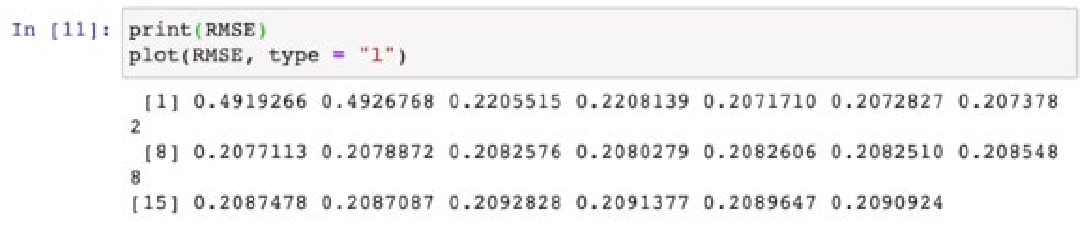


```
In [*]: for (d in 1:20) {
            f <- bquote(Y ~ poly(X, .(d)))
            LinearRegressor <- caret::train(as.formula(f), data = data, method = "
            RMSE <- c(RMSE, LinearRegressor$results$RMSE)
        }
```

CHEN: After completing all these actions, the optimal tuning parameter can be identified and chosen in the next step.

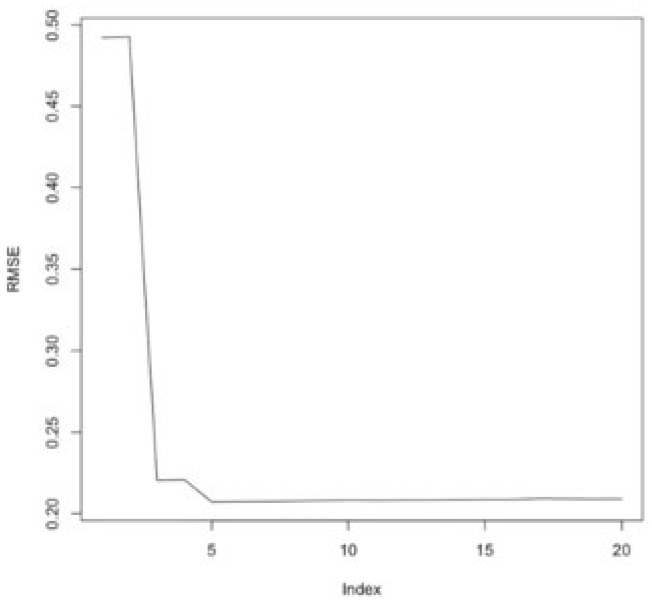
Step 3: Choose the optimal model or tuning parameter

CHEN: With different tuning parameters estimated in Step 2, the variables that contains that root mean square error values are printed to ensure that order of polynomial, which has the lowest root mean square error, can be chosen from the list.



```
In [11]: print(RMSE)
         plot(RMSE, type = "l")

 [1] 0.4919266 0.4926768 0.2205515 0.2208139 0.2071710 0.2072827 0.207378
2
 [8] 0.2077113 0.2078872 0.2082576 0.2080279 0.2082606 0.2082510 0.208548
8
[15] 0.2087478 0.2087087 0.2092828 0.2091377 0.2089647 0.2090924
```

CHEN: As you can see from the values in the output, the best model has a polynomial order of five



Conclusion

CHEN: This video explored the application of the machine learning pipeline on two examples using the KNN classifier and polynomial regression. In the next component, you will have the opportunity to execute those steps on different data sets.