Logistic Regression, LDA QDA and KNN

Abbass Al Sharif

October 2, 2014

In this document, I will show you how to run LR, LDA, QDA, and KNN for classification methods on the stock market data set (Smarket). For each method, I will be reading the data seperately because some of these methods require us to input data in different ways (especially the KNN function).

Logistic Regression

The Smarket data set is part if the ISLR package. We'll first read load the ISLR package, and then split the Smarket data set into training and testing data. In this problem, we will use subsetting techniques (vector subsetting). The training data set will contain all observations before 2005 and the testing data set will have all observations in 2005.

The following command attaches the data frame for Smarket to R's working directory (memory). This will make us able to access variables in the data set directly without having to specify the name of the data frame, so instead of typing Smarket$Year we can directly type Year which is a variable in the Smarket data set.

```
library(ISLR)
attach(Smarket)

## find the indecies for the observations in Smarket that consitutes the training
## and testing data

train = Year < 2005
test = !train
```

The above commands creats two boolean vectors. A boolean vector has either TRUE or FALSE as its value. For the train vector, a TRUE will be assigned to the cell that has the same index as the observation in Smarket which has Year < 2005. The test vector is exactly the oposite of train. The '!' negates what is in train.

Select the observations in Smarket that will go into the training and testing data set. Notice that we got rid of the 8th variable `Today` because it is similar to direction. Actually, that's how `Direction` is computed:

```
training_data = Smarket[train, -8]
testing_data = Smarket[test, -8]
```

For model assesment purposes, we are going to create a vector that has all the y values in the testing data set. The model assessment will happen later on, after we create our model using the training data

```
testing_y = Direction[test]
```

we used Direction because this is what we are trying to predict (our y variable). Notice here that we did not put a comma `,` when we indexed Direction because it is a vector (one column) and not a dataframe!

Now it is time to train our model using the training data set:

```
logistic_model = glm(Direction ~ .,
                     data = training_data,
                     family = "binomial")
```

In the above logistic Regression model, we are using the `glm()` function, which is a general linear model. The first argumet is our regression forumla, which specifies that we are predicting `Direction` using all predictor variables in our date set (the `.` means to use all variables). If you want to use specific variables, let's say `Lag1` and `Lag2`, then the formula would be `Direction ~ Lag1 + Lag2`. We are using the training data set to train our model, and we specified the family of the model to be `binomial`, because we are running logistic regression. If we don't specify the family of the linear model, then the model will be regular linear regression

Next, we want to assess our model `logistic_model`. To do so, we will predict the y values for the testing data set, and then compare the predicted y's with the real one that we saved under the name `testing_y` earlier. When the `predict()` function is used in logistic regression, it computes the predicted probablities of being in one class or another (Down or Up in our case).

```
logistic_probs = predict(logistic_model, testing_data, type = "response")
head(logistic_probs)
```

```
##    999   1000   1001   1002   1003   1004
## 0.6385 0.6017 0.6038 0.5962 0.5875 0.5928
```

Since `predict()` computes probablities, then we have to convert them to the actual classes (Up or Down). Unfortunately, in logistic regression, `predict()` function does not produce the categories. So, let's convert those probablities. We will first start by creating a vector to hold those classes. This array will have the same length of the `testing_y` (252 in this example), and we will initialize it to have all of its cells marked as `Down`, and then we will update this vector to have `Up` in cells where the corresponding predicted probablities is greater than `0.5` (this threshold could change based on the application).

```
logistic_pred_y = rep("Down", length(testing_y))
## the function rep(), repeats "Down" 252 times

logistic_pred_y[logistic_probs > 0.5] = "Up"
## R will first evaluate "logistic_probs >0.5", and it will be a vector of TRUE and FALSE.
## TRUE when the value of the cell in logistic_prob > 0.5, otherwise FALSE.
## R will replace all the "Down" values in "logistic_pred_y" vector with "Up"
## when "logistic_probs >0.5" is TRUE.
```

The last few steps in assessment includes finding the confusion matrix for the model.

```
## the following command creates the confusion matrix
table(logistic_pred_y, testing_y)
```

```
##                testing_y
## logistic_pred_y Down  Up
##            Down    2   1
##            Up    109 140
```

Now, lets compute the misclassification error rate:

```
mean(logistic_pred_y != testing_y)
```

```
## [1] 0.4365
```

The missclassification error rate for the logistic regression model we created above is 43.65%, which is considered a high misclassification error rate.

Linear Discriminant Analysis (LDA)

To run LDA in R, we will be using the `lda()` function which is part of a package called `MASS`. This package comes with the core distribution of R, so we just need to load it into R's working directory before we use the `lda()` function.

```
library(MASS)

## create an LDA model. The lda() function takes a forumla and the name of the training data set
## as its argument
lda_model = lda(Direction~., data = training_data)
```

Next, we will assess the model, so again we will use `predict()` fuction on our testing data set.

```
lda_pred = predict(lda_model, testing_data)
names(lda_pred)
```

```
## [1] "class"     "posterior" "x"
```

```
lda_pred_y = lda_pred$class
```

The good news is that when using an LDA model in the `predict()` fuction, the output is the categories themselves(classes), unlike what happened when we used logistic regression model (the output was probabilities).

Alright, now it is time to assess the model. We create the confusion matrix and compute the misclassification error.

```
## compute the confusion matrix
table(lda_pred_y, testing_y)
```

```
##           testing_y
## lda_pred_y Down  Up
##       Down    2   1
##       Up    109 140
```

```
## compute the misclassification error rate
mean(lda_pred_y != testing_y)
```

```
## [1] 0.4365
```

The misclassification error rate for the LDA model turned out to be the same as the one for the logistic regression model. Let's check if a QDA model would do better in terms of misclassification error rate.

Quadratic Discriminant Analysis

Training and assessing a QDA model is very similar in syntax to training and assessing a QDA model. The only difference is in the function name `lda()`.

```
library(MASS)

qda_model = qda(Direction~., data = training_data)
qda_pred = predict(qda_model, testing_data)
qda_pred_y = qda_pred$class

table(qda_pred_y, testing_y)
```

```
##           testing_y
## qda_pred_y Down Up
##       Down   43 51
##       Up     68 90
```

```
mean(qda_pred_y != testing_y)
```

```
## [1] 0.4722
```

The miscallisifcation error rate for the QDA model is 47.22% which is higher than the ones we got from both logistic regression and LDA models.

KNN for Classification

To train a KNN model for classification, we will be using the function `knn()`, which is part of the `class` R package. Make sure to install and load this library.

```
## load the class R package.
library(class)
```

The splitting of data here will be different from what we did for logistic regression, LDA, and QDA. This is because the `knn()` fuction is built to take different arguements compared to `glm()`, `lda()`, and `qda()`.

For `knn()`, we have to have our `y` variable in a seperate column from the training and testing data. In addition to this issue, we have to scale or standerdize our numerical variables because the KNN method classifies observations using distance measures. For more information about this issue, please refer to a previous document on "KNN for Classification".

To standardize the dataset, we can use the function `scale()` as follows:

```
## load this package to use Smarket data
library(ISLR)

## scale the Smarket data without the 8th variable (Today), and the 9th variable (Direction).
## we got rid of Today variable because is is highly correlated with Direction
## we got rid of Direction because it is our response variable. Make sure to exclude all
## categorical variables should be excluded from scaling. We can't scale categorical variables!

data = scale(Smarket[,-c(8,9)])
```

Now let's split the data. Remember there it is going to be a bit different from what we did earlier:

```
## the following two steps are similar to earlier steps
train = Year < 2005
test = !train

## split into train and testing data

## the following two steps looks similar to what we did earlier, but they are actually not!!
## Remember that we got rid of the response variable "Direction" when we scaled the data!
## So, our training and testing data has only the predictors! That's how KNN() function work.

training_data = data[train,]
testing_data = data[test,]

## KNN take the training response variable seperately
training_y = Smarket$Direction[train]

## we also need the have the testing_y seperately for assesing the model later on
testing_y = Smarket$Direction[test]
```

Now the stage is ready for us to train a KNN model. The `knn()` function uses a random number generator to train the model. In order to get the same output for your analysis everytime you run your R code, then make sure to set the seed for the random number generator in R to a number of your choice. But you have to stick to the same seed every time you run your R code.

`knn()` predicts the categories for the response viriable, so in this case we don't need to use the `predict()` function as we did in the previous models. The following code, trains a KNN model with `k=1`.

```
set.seed(1)
knn_pred_y = knn(training_data, testing_data, training_y, k = 1)
table(knn_pred_y, testing_y)
```

```
##           testing_y
## knn_pred_y Down Up
##       Down   42 53
##       Up     69 88
```

```
mean(knn_pred_y != testing_y)
```

```
## [1] 0.4841
```

The misclassifcation error rate when `k=1` is `48.41%`. It is not better than the previous regression, lda, and qda models. Let's see what value of `k` would give us the lowest misclassification error rate. We will have a `for` loop for this purpose:

```
  knn_pred_y = NULL
  error_rate = NULL

for(i in 1:300){
  set.seed(1)
  knn_pred_y = knn(training_data,testing_data,training_y,k=i)
```

```
  error_rate[i] = mean(testing_y != knn_pred_y)
}

### find the minimum error rate
min_error_rate = min(error_rate)
print(min_error_rate)
```

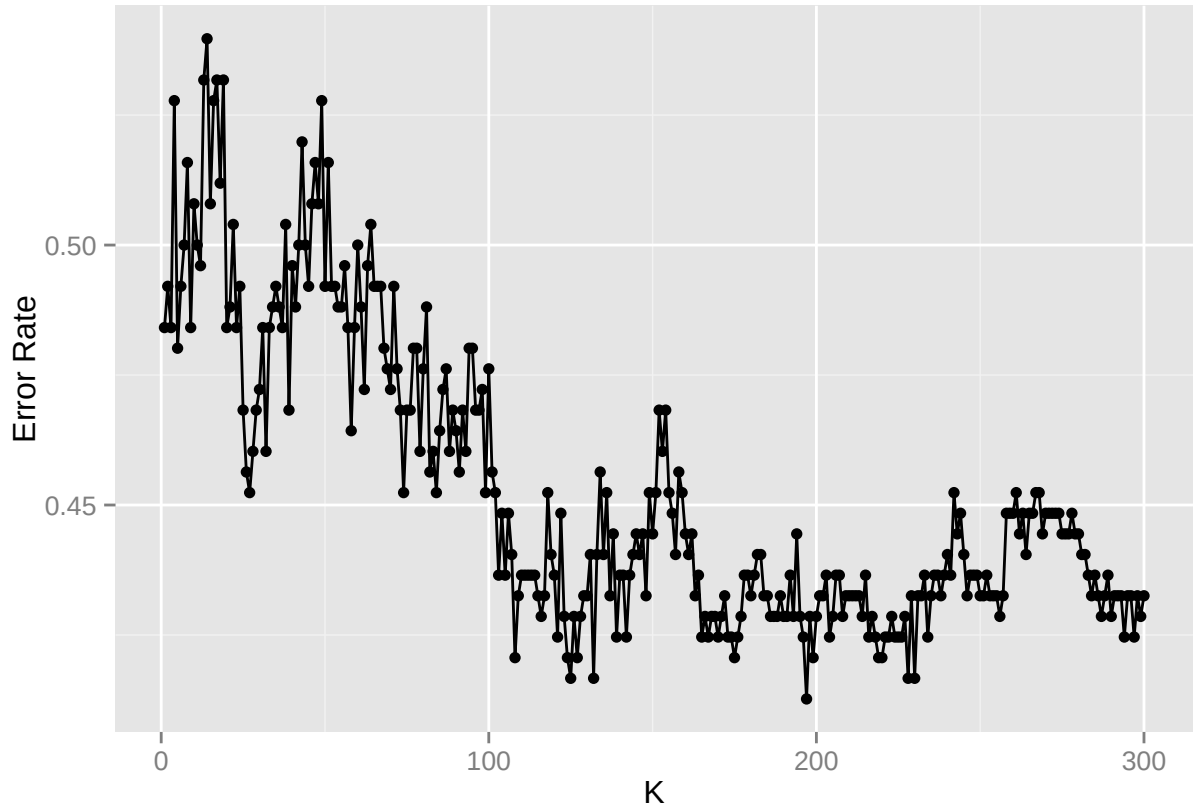
```
## [1] 0.4127
```

```
### get the index of that error rate, which is the k
K = which(error_rate == min_error_rate)
print(K)
```

```
## [1] 197
```

To visualize how the misclassification error rate is affected when we increase `k`:

```
library(ggplot2)
qplot(1:300, error_rate, xlab = "K",
      ylab = "Error Rate",
      geom=c("point", "line"))
```



When we train a KNN model with `k=27`, then we get the lowest misclassification error rate of `41.27%`. This will lead us to conlude that the best model for this data set would be either the logistic regression or LDA model because they have the least misclasslifcation error. But be careful, this might be overfitting! What do you think?