Logistic Regresion, LDA, QDA and KNN

Laura Cline

23/10/2021

Contents


```
library(MASS) #lda()
library(ISLR)
library(class) #knn()
```

Logistic Regression

We will begin by examining some numerical and graphical summaries of the `Smarket` data, which is part of the `ISLR` library. This dataset consists of percentage returns of the S&P 500 stock index over 1,250 days from the beginning of 2001 until the end of 2005. For each date, we have recorded the percentage returns for each of the five previous trading days, `Lag1` through `Lag5`. We have also recorded `Volume` (the number of shares traded on the previous day, in billions), `Today` (the percentage return on the data in question), and `Direction` (whether the market was `Up` or `Down` on this date).

```
# Begin with a cursory inspection of the data structure
colnames(Smarket) <- tolower(colnames(Smarket))
names(Smarket)
```

```
## [1] "year"      "lag1"      "lag2"      "lag3"      "lag4"      "lag5"
## [7] "volume"    "today"     "direction"
```

```
dim(Smarket)
```

```
## [1] 1250    9
```

```
summary(Smarket)
```

```
##       year           lag1                lag2                lag3
##  Min.   :2001   Min.   :-4.922000   Min.   :-4.922000   Min.   :-4.922000
##  1st Qu.:2002   1st Qu.:-0.639500   1st Qu.:-0.639500   1st Qu.:-0.640000
##  Median :2003   Median : 0.039000   Median : 0.039000   Median : 0.038500
##  Mean   :2003   Mean   : 0.003834   Mean   : 0.003919   Mean   : 0.001716
##  3rd Qu.:2004   3rd Qu.: 0.596750   3rd Qu.: 0.596750   3rd Qu.: 0.596750
##  Max.   :2005   Max.   : 5.733000   Max.   : 5.733000   Max.   : 5.733000
```

```
##       lag4               lag5              volume           today
##  Min.   :-4.922000   Min.   :-4.92200   Min.   :0.3561   Min.   :-4.922000
##  1st Qu.:-0.640000   1st Qu.:-0.64000   1st Qu.:1.2574   1st Qu.:-0.639500
##  Median : 0.038500   Median : 0.03850   Median :1.4229   Median : 0.038500
##  Mean   : 0.001636   Mean   : 0.00561   Mean   :1.4783   Mean   : 0.003138
##  3rd Qu.: 0.596750   3rd Qu.: 0.59700   3rd Qu.:1.6417   3rd Qu.: 0.596750
##  Max.   : 5.733000   Max.   : 5.73300   Max.   :3.1525   Max.   : 5.733000
##  direction
##  Down:602
##  Up  :648
##
##
##
##
```

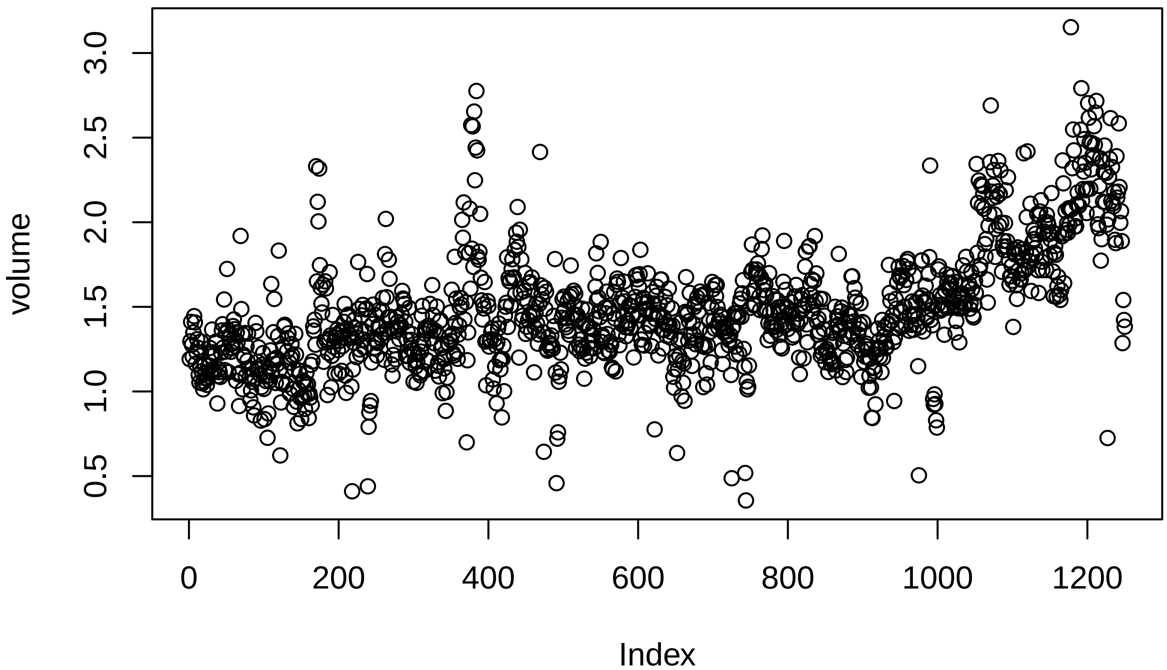
The `cor()` function produces a matrix that contains all of the pairwise correlatons among the predictors in the dataset. The first command below gives an error message because the `Direction` variable is qualitative.

```
cor(Smarket[,-9]) # The 9th column variable, Direction, is qualitative so it should be excluded from th
```

```
##              year         lag1         lag2         lag3         lag4
## year   1.00000000  0.029699649  0.030596422  0.033194581  0.035688718
## lag1   0.02969965  1.000000000 -0.026294328 -0.010803402 -0.002985911
## lag2   0.03059642 -0.026294328  1.000000000 -0.025896670 -0.010853533
## lag3   0.03319458 -0.010803402 -0.025896670  1.000000000 -0.024051036
## lag4   0.03568872 -0.002985911 -0.010853533 -0.024051036  1.000000000
## lag5   0.02978799 -0.005674606 -0.003557949 -0.018808338 -0.027083641
## volume 0.53900647  0.040909908 -0.043383215 -0.041823686 -0.048414246
## today  0.03009523 -0.026155045 -0.010250033 -0.002447647 -0.006899527
##                lag5      volume        today
## year    0.029787995  0.53900647  0.030095229
## lag1   -0.005674606  0.04090991 -0.026155045
## lag2   -0.003557949 -0.04338321 -0.010250033
## lag3   -0.018808338 -0.04182369 -0.002447647
## lag4   -0.027083641 -0.04841425 -0.006899527
## lag5    1.000000000 -0.02200231 -0.034860083
## volume -0.022002315  1.00000000  0.014591823
## today  -0.034860083  0.01459182  1.000000000
```

As one would expect, the correlation between the lag variables and today's returns are close to zero. In other words, there appears to be little correlation between today's returns and the previous day's returns. The only substantial correlation is between `Year` and `Volume`. By plotting the data we see that `Volume` is increasing over time. In other words, the average number of shares traded daily increased from 2001 to 2005.

```
attach(Smarket)
plot(volume)
```



Next, we fit a logistic regression model in order to predict `Direction` using `Lag1` through `Lag5` and `Volume`. The `glm()` function fits *generalized linear models*, a class of models that includes logistic regression. The syntax of the `glm()` function is similiar to that of `lm()`, except that we must pass in the argument `family=binomial` in order to tell `R` to run a logistic regression rather than some other type of generalized linear model.

```
# We now fit a logit model in order to predict Direction using Lag1-Lag5 and Volume with the glm() func
glm.fit <- glm(direction ~ lag1 + lag2 + lag3 + lag4 + lag5 + volume,
               data=Smarket, family="binomial")
summary(glm.fit)
```

```
##
## Call:
## glm(formula = direction ~ lag1 + lag2 + lag3 + lag4 + lag5 +
##     volume, family = "binomial", data = Smarket)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -1.446  -1.203   1.065   1.145   1.326
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.126000   0.240736  -0.523    0.601
## lag1        -0.073074   0.050167  -1.457    0.145
## lag2        -0.042301   0.050086  -0.845    0.398
## lag3         0.011085   0.049939   0.222    0.824
## lag4         0.009359   0.049974   0.187    0.851
## lag5         0.010313   0.049511   0.208    0.835
## volume       0.135441   0.158360   0.855    0.392
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1731.2  on 1249  degrees of freedom
## Residual deviance: 1727.6  on 1243  degrees of freedom
```

```
## AIC: 1741.6
##
## Number of Fisher Scoring iterations: 3
```

```r
summary(glm.fit)$coef
```

```
##                 Estimate Std. Error    z value  Pr(>|z|)
## (Intercept) -0.126000257 0.24073574 -0.5233966 0.6006983
## lag1        -0.073073746 0.05016739 -1.4565986 0.1452272
## lag2        -0.042301344 0.05008605 -0.8445733 0.3983491
## lag3         0.011085108 0.04993854  0.2219750 0.8243333
## lag4         0.009358938 0.04997413  0.1872757 0.8514445
## lag5         0.010313068 0.04951146  0.2082966 0.8349974
## volume       0.135440659 0.15835970  0.8552723 0.3924004
```

The smallest p-value here is associated with `Lag1`. The negative coefficient for this predictor suggests that if the market had a positive return yesterday, then it is less likely to go up today. However, at a value of 0.145, the p-value is still relatively large, and there is no clear evidence of a real association between `Lag1` and `Direction`.

Take a look at the coefficient for `Lag1` - it is negative, which implies that positive returns yesterday mean less change of stock increase today. However, the p-value of `Lag1` (0.145) is kind of large, so there's not clear evidence of this relation being true.

We used the `coef()` function in order to access the coefficients for this fitted model. We can also use the `summary()` function to access particular aspects of a fitted model, such as the p-values for the coefficients.

The `predict()` function can be used to predict the probability that the market will go up, given values of the predictors. The `type="response"` option tells `R` to output probabilities of the form $P(Y = 1|X)$, as opposed to other information such as logit. If no dataset is supplied to the `predict()` function, then the probabilities are computed for the training data that was used to fit the logistic regression model. Here we have printed only the first 10 probabilities. We know that these values correspond to the probability of the market going up, rather than down, because the `contrasts()` function indicates that `R` has created a dummy variable with 1 for `Up`.

```r
#Because our output is log-odds, we need to use the option type="response" when making predictions if w
glm.probs <- predict(glm.fit, type="response")
glm.probs[1:10]
```

```
##         1         2         3         4         5         6         7         8
## 0.5070841 0.4814679 0.4811388 0.5152224 0.5107812 0.5069565 0.4926509 0.5092292
##         9        10
## 0.5176135 0.4888378
```

```r
# To make sure that these probabilities are for the market going Up, check how R codes direction - we n
contrasts(direction)
```

```
##      Up
## Down  0
## Up    1
```

In order to make a prediction as to whether the market will go up or down on a particular day, we must convert these predicted probabilities into class labels, `Up` or `Down`. The following two commands create a vector of class predictions based on whether the predicted probability of a market increase is greater than or less than 0.5.

```r
# Instead of looking at these probabilities one by one, we can set a threshold to tell us what our obse
glm.pred <- rep("Down", 1250)
glm.pred[glm.probs>0.5] = "Up"
```

In order to make a prediction as to whether the market will go up or down on a particular day, we must convert these predicted probabilities into class labels, `Up` or `Down`. The following two commands create a vector of class predictions based on whether the predicted probability of a market increase is greater than or less than 0.5.

The first command creates a vector of 1,250 `Down` elements. The second line transforms to `Up` all of the elements for which the predicted probability of a market increase exceeds 0.5. Given these predictions, the `table()` function can be used to produce a confusion matrix in order to determine how many observations were correctly or incorrectly classified.

```
# Now that we've got predicted values and true values of direction, we can create a confusion matrix to
table(glm.pred, direction)
```

```
##         direction
## glm.pred Down  Up
##     Down  145 141
##     Up    457 507
```

```
mean(glm.pred==direction)
```

```
## [1] 0.5216
```

The diagonal elements of the confusion matrix indicate correct predictions while of off-diagonals represent incorrect predictions. Hence, our model correctly predicted that the market would go up on 507 days and that it would go down on 145 days, for a total of 507 + 145 = 652 correct predictions. The `mean()` function can be used to compute the fraction of days for which the prediction was correct. In this case, logistic regression correctly predicted the movement of the market 52.2% of the time.

The output of the `mean()` function tells us the fraction of days that our model predicted correctly - 52.16% of the time. That means that our model has a 47.84% training error rate - hardly better than flipping a coin. Furthermore, the training error rate generally underestimates the test error rate, so the actual accuracy may be worst.

At first glance, it appears that the logistic regression model is working a little better than random guessing. However, this result is misleading because we trained and tested the model on the same set of 1,250 observations. In other words, 100 - 53.2 = 47.8% is the *training* error rate. As we have seen previously, the training error rate is often overly optimistic - it tends to underestimate the test error rate. In order to better assess the accuracy of the logistic regression model in this setting, we can fit the model using part of the data, and then examine how well it predicts the *held out* data. This will yield a more realistic error rate, in the sense that in practice we will be interested in our model's performance not on the data that was used to fit the model, but rather on days in the future for which the market's movements are unknown.

To implement this strategy, we will first create a vector corresponding to the observations from 2001 through 2004. We will then use this vector to create a held out dataset of observations from 2005.

```
# The training data for the model is the entire dataset, leaving nothing for testing. Let's split up th
train <- (year < 2005) # Create training set indicator for the years 2001-2004
Smarket.2005 <- Smarket[!train,] # Subset the original data to keep data with train == 0
direction.2005 <- Smarket.2005[,"direction"] # True values of the test set
```

Instead of splitting the original data (Smarket) into two new sets, training and testing, we simply use a new indicator variable for training observations. This is useful when working with extremely large dataset and will keep RAM usage down.

The object `train` is a vector of 1,250 elements, corresponding to the observations in our dataset. The elements of the vector that correspond to observations that occured before 2005 are set to `TRUE`, whereas those that correspond to observations in 2005 are set to `FALSE`. The object `train` is a *Boolean* vector, since its elements are `TRUE` and `FALSE`. Boolean vectors can be used to obtain a subset of the rows or columns of a matrix. For instance, the command `Smarket[train,]` would pick out a submatrix of the stock market dataset, corresponding only to the dates before 2005, since those are the ones for which the elements of `train` are `TRUE`. The ! symbol can be used to reverse all of the elements of a Boolean vector. That is, `!train` is a vector similar to `train`, expect that the elements that are `TRUE` in `train` get swapped to `FALSE` in `!train`. Therefore, `Smarket[!train,]` yields a submatrix of the stock market data containing only the observations for which `train` is `FALSE` - that is, the observations with dates in 2005. The output above indicates that there are 252 such observations.

We not fit a logisic regression model using only the subset of the observations that correspond to dates before 2005, using the `subset` argument. We then obtain predicted probabilities of the stock market going up for each of the days in the test set - that is, for the days in 2005.

```
# Fit a new logistic regression using only the subset of observations for which train == 1, and then fi
glm.fit <- glm(direction ~ lag1 + lag2 + lag3 + lag4 + lag5 + volume, data=Smarket,
               family="binomial", subset=train)
glm.probs <- predict(glm.fit, Smarket.2005, type="response")
```

Notice that we have trained and tested our model on two completely seperate datasets: training was performed using only the dates before 2005, and testing was performed using only dates in 2005. Finally, we compute the predictions for 2005 and compare them to the actual movements of the market over that time period.

```
# Create Yes/No predictions just as we did with the previous model
glm.pred = rep("Down", length(glm.probs))
glm.pred[glm.probs> 0.5] = "Up"
```

For better reproducibility of your code, it's usually a good idea to refrain from using explicit numbers, like we did earlier with `glm.pred`. Instead, try to find ways to generalize what you are trying to do so that it can ve done again without many edits to the original code. For instance, when we create the `Down` vector, instead of using the number 252 in the second argument we just say `length(glm.probs)`. Both are equivalent line of code, but the second allows us to easily change the size of the test set and not worry about running into errors.

```
table(glm.pred, direction.2005)
```

```
##          direction.2005
## glm.pred Down Up
##     Down   77 97
##     Up     34 44
```

```
mean(glm.pred != direction.2005)
```

```
## [1] 0.5198413
```

The results are not that great - the test error rate is 52%. We noticed earlier that the p-values are pretty lackluster, so it might be worth investigating a similar model that excludes the terms with the highest p-values.

The `!=` notation means *not equal to*, and so the last command computes the test error rate. The results are rather disappointing: the test error rate is 52%, which is worse than random guessing! Of course this result is not surprising, given that one would not generally expect to be able to use previous days' returns to predict future market performance.

We recall that the logistic regression model had very underwhelming p-values associated with all of the predictors, and that the smallest p-value (although not very small) corresponding to `Lag1`. Perhaps by removing the variables that appear not be helpful in predicting `Direction`, we can obtain a more effective model. After all, using predictors that have no relationship with the response tends to cause deterioration in the test error rate (since such predictors cause an increase in variance without a corresponding decrease in bias), and so removing such predictors may in turn yield an improvement. Below we have refit the logistic regression using just `Lag1` and `Lag2`, which seemed to have the highest predictive power in the original logistic regression model.

```r
# Let's fit a new model, only using lag1 and lag2 as predictors
glm.fit <- glm(direction ~ lag1 + lag2, data=Smarket,
               family="binomial", subset = train)
glm.probs = predict(glm.fit, Smarket.2005, type="response")
```

```r
glm.pred <- rep("Down", length(glm.probs))
glm.pred[glm.probs > 0.5] = "Up"
```

```r
table(glm.pred, direction.2005)
```

```
##          direction.2005
## glm.pred Down  Up
##     Down   35  35
##     Up     76 106
```

```r
mean(glm.pred == direction.2005)
```

```
## [1] 0.5595238
```

Results are a little better - 55.95% of our observations were predicted correctly by the model. Further still, it has a 58% rate of correctly predicting an upward movement in the market (True Positives / (True Positives + False Positives)).

Now the results appear to be a little better: 56% of the daily movements have been correctly predicted. It is worth noting that in this case, a much simpler strategy of the predicting that the market will increase every day will also be correct 56% of the time! Hence, in terms of overall error rate, the logistic regression method is no better than the naive approach. However, the confusion matrix shows that on days when logistic regression predicts an increase in the market, it has a 58% accuracy rate. This suggests a possible trading strategy of buying on days when the model predicts an increasing market, and avoiding trades on days when a decrease is predicted. Of course one would need to investifate more carefully whether this small improvement was real or just due to random chance.

Linear Discriminant Analysis (LDA)

Now we will perform LDA on the `Smarket` data. In `R`, we fit an LDA using the `lda()` function, which is part of the `MASS` library. Notice that the syntax for the `lda()` function is identical to that of `lm()`, and to that of `glm()` except for the absence of the `family` option. We fit the model using only the observations before 2005.

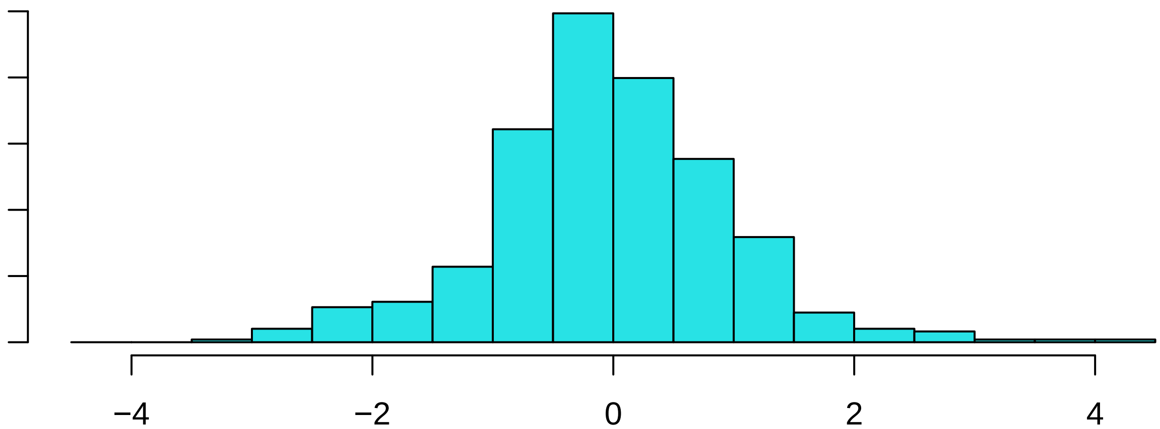
```r
# We will use the lda() function on the training data that we previously created; notice that the synta
lda.fit <- lda(direction ~ lag1 + lag2, data=Smarket, subset=train)
lda.fit
```

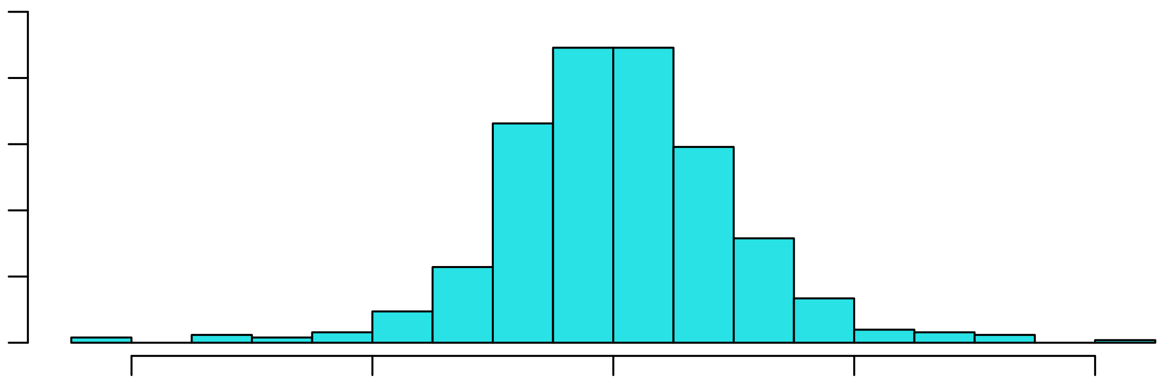
```
## Call:
## lda(direction ~ lag1 + lag2, data = Smarket, subset = train)
##
## Prior probabilities of groups:
##     Down       Up
## 0.491984 0.508016
##
## Group means:
##             lag1        lag2
## Down  0.04279022  0.03389409
## Up   -0.03954635 -0.03132544
##
## Coefficients of linear discriminants:
##             LD1
```

```
## lag1 -0.6420190
## lag2 -0.5135293
```

The output first tells us that 49.19% of our training data corresponds to days where the market went down, and 50.8% went up. The group means is a cross-tabulation of the means of the predictors for each level of the dependent variable, `direction`. Lastly, the coefficients of the linear discriminants gives us the linear combination of lag1 and lag2 that create the decision boundary. If the expression -0.64*lag1 - 0.5135*lag2 is large, the LDA classifier will predict a market increase. Likewise, if the above expression is small, the LDA classifier will predict a market decrease.

```
# The plot() function produces plots of the linear discriminants, obtained by plugging in values of lag
par(mar=c(1,1,1,1))
plot(lda.fit)
```





Observe the distribution of the plots for `Up` and `Down` - it is centered mostly around 0. What does this tell us? Because there are few extreme values in these distributions, the model has a tough time predicting market movement in either class.

The LDA output indicates that $\hat{\pi}_1 = 0.492$ and $\hat{\pi}_2 = 0.508$; in other words, 49.2% of the training observations correspond to days during which the market went down. It also provides the group means; these are the average of each predictor within each class, and are used by LDA as estimates of μ_k. These suggest that there is a tendency for the previous 2 days' returns to be negative on days when the market increases, and a tendency for the previous days' returns to be positive on days when the market declines. The *coefficients of linear discriminants* output provides the linear combination of `Lag1` and `Lag2` that are used to form the LDA decision rule. In other words, there are the multipliers of the elements of X = x. If -0.642x`Lag1` - 0.514x`Lag2` is large, then the LDA classifier will predict a market increase, and if it is small, then the LDA classifier will predict a market decline. The `plot()` function produces plots of the *linear discriminants*, obtained by computing -0.642x`Lag1` - 0.514x`Lag2` for each of the training observations.

The `predict()` function returns a list with three elements. The first element, `class` contains LDA's predictions about the movement of the market. The second element, `posterior`, is a matrix whose kth column contains the posterior probability that the corresponding observation belongs to the kth class. Finally, `x` contains the linear discriminants described earlier.

```
# Now we make some predictions, and see how our results compare to those of the logistic regression
lda.pred = predict(lda.fit, Smarket.2005)
names(lda.pred)
```

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

Notice the output of `lda.pred`:

- "class" tells us the prediction of the LDA
- "posterior" is a matrix of probabilities of the observations belonging to that class
- "x" contains the linear discriminants

As we observed before, the LDA and logistic regression predictions are almost identical.

```
lda.class = lda.pred$class
table(lda.class, direction.2005)
```

```
##          direction.2005
## lda.class Down  Up
##      Down   35  35
##      Up     76 106
```

```
mean(lda.class == direction.2005)
```

```
## [1] 0.5595238
```

The mean is pretty much identical to that of the logistic regression.

Quadratic Discriminant Analysis (QDA)

We will now fit a QDA model to the `Smarket` data. QDA is implemented in `R` using the `qda()` function, which is also part of the `MASS` library. The syntax is identical to that of `lda()`.

```
# The syntax for the qda() function is identical to that of lda(), so we can quickly replicate what we
qda.fit <- qda(direction ~ lag1 + lag2, data=Smarket, subset=train)
qda.fit
```

```
## Call:
## qda(direction ~ lag1 + lag2, data = Smarket, subset = train)
##
## Prior probabilities of groups:
##     Down       Up
## 0.491984 0.508016
##
## Group means:
##             lag1        lag2
## Down  0.04279022  0.03389409
## Up   -0.03954635 -0.03132544
```

Notice that the `qda()` function does not report the coefficients of the linear discriminants.

The output contains the group means. But it does not contain the coefficients of the linear discriminants, because the QDA classifier involves a quadratic, rather than a linear, function of the predictors. The `predict()` function works in exactly the same fashion as for LDA.

```r
# Make the same usual predictors
qda.class <- predict(qda.fit, Smarket.2005)$class
table(qda.class, direction.2005)
```

```
##          direction.2005
## qda.class Down  Up
##      Down   30  20
##      Up     81 121
```

```r
mean(qda.class == direction.2005)
```

```
## [1] 0.5992063
```

Our mean has increased to about 60%, which suggests the quadratic form assumed by QDA better fits the relationship.

Interestingly, the QDA predictors are accurate almost 60% of the time, even though the 2005 data was not used to fit the model. This level of accuracy is quite impressive for stock market data, which is known to be quite hard to model accurately. This suggests that the quadratic form assumed by QDA may capture the true relationship more accurately than the linear forms assumed by LDA and logistic regression. However, we recommend evaluating these method's performance on a larger test set before betting that this approach will consistently beat the market.

K-Nearest Neighbours (KNN)

We will now perform KNN using the `knn()` function, which is part of the `class` library. This function works rather differentlyy from the other model-fitting functions that we have encountered so far. Rather than a two-step approach in which we first fit the model and then we use the model to make predictions, `knn()` forms predictions using a single command. The function requires four inputs:

1. A matrix containing the predictors associated with the training data, labeled `train.X` below.
2. A matrix containing the predictors associated with the data for which we wish to make predictions, labeled `test.X` below.
3. A vector containing the class labels for the training observations, labeled `train.Direction` below.
4. A value for K, the number of nearest neighbours to be used by the classifier.

The syntax for `knn()` is different from the commands that we have used in the past: `knn(train, test, cl, k)`:

- `train` - the matrix or data frame of training set cases
- `test` - the matrix or data frame of test set cases
- `cl` - a vector containing the class labels for the training observations
- `k` - the number of neighbours considered

We use the `cbind()` function, short for *column bind*, to bind the `Lag1` and `Lag2` variables together into two matrices, one for the training set and the other for the test set.

```r
# Create the training and testing sets
train.X <- cbind(lag1, lag2)[train,]
test.X <- cbind(lag1, lag2)[!train,]
train.direction <- direction[train]
```

Now the `knn()` function can be used to predict the market's movement for the dates in 2005. We set a random seed before we apply `knn()` because if several observations are tied to nearest neighbours, then `R` will randomly break the tie. Therefore, a seed must be set in order to ensure reproducability of results.

```r
# Now use knn() to predict the market's movement for the dates in 2005 (test set)
set.seed(1)
knn.pred <- knn(train.X, test.X, train.direction, k=1)
table(knn.pred, direction.2005)
```

```
##          direction.2005
## knn.pred Down Up
##     Down   43 58
##     Up     68 83
```

```r
mean(knn.pred == direction.2005)
```

```
## [1] 0.5
```

The results using K = 1 are not very good, since only 50% of the observations are correctly predicted. Of course, it may be that K = 1 results in an overly flexible fit to the data. Below, we repeat the analysis using K = 3.

```r
# We only used one nearest neighbour, so obviously the predictive power is not great
knn.pred <- knn(train.X, test.X, train.direction, k=3)
table(knn.pred, direction.2005)
```

```
##          direction.2005
## knn.pred Down Up
##     Down   48 54
##     Up     63 87
```

```r
mean(knn.pred == direction.2005)
```

```
## [1] 0.5357143
```

The results have improved slightly. But increasing K further turns out to provide no further improvements. It appears that for this data, QDA provides the best results of the methods that we have examined so far.

An Application to Caravan Insurance Data

Finally, we will apply the KNN approach to the `Caravan` dataset, which is part of the `ISLR` library. This dataset includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is `Purchase`, which indicates whether or not a given individual purchases a caravan insurance policy. In this dataset, only 6% of people purchased caravan insurance.

```r
dim(Caravan)
```

```
## [1] 5822   86
```

```r
colnames(Caravan) <- tolower(colnames(Caravan))
attach(Caravan)
sapply(Caravan, class) # check the class of each variable to see which are categorical and which are co
```

```
##   mostype  maanthui   mgemomv  mgemleef  moshoofd    mgodrk    mgodpr    mgodov
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##    mgodge    mrelge    mrelsa    mrelov  mfalleen  mfgekind  mfwekind  moplhoog
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##  moplmidd  mopllaag  mberhoog  mberzelf  mberboer  mbermidd  mberarbg  mberarbo
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      mska     mskb1     mskb2      mskc      mskd    mhhuur    mhkoop     maut1
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##     maut2     maut0   mzfonds    mzpart   minkm30  mink3045  mink4575  mink7512
```

```
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##  mink123m   minkgem  mkoopkla   pwapart   pwabedr   pwaland  ppersaut   pbesaut
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##   pmotsco   pvraaut  paanhang  ptractor     pwerkt     pbrom    pleven  ppersong
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##   pgezong   pwaoreg    pbrand   pzeilpl  pplezier    pfiets   pinboed  pbystand
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##   awapart   awabedr   awaland  apersaut   abesaut   amotsco   avraaut  aaanhang
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##  atractor    awerkt     abrom    aleven  apersong   agezong   awaoreg    abrand
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##   azeilpl  aplezier    afiets   ainboed  abystand  purchase
## "numeric" "numeric" "numeric" "numeric" "numeric"  "factor"
```

```r
attach(Caravan)
```

```
## The following objects are masked from Caravan (pos = 3):
##
##     aaanhang, abesaut, abrand, abrom, abystand, afiets, agezong,
##     ainboed, aleven, amotsco, apersaut, apersong, aplezier, atractor,
##     avraaut, awabedr, awaland, awaoreg, awapart, awerkt, azeilpl,
##     maanthui, maut0, maut1, maut2, mberarbg, mberarbo, mberboer,
##     mberhoog, mbermidd, mberzelf, mfalleen, mfgekind, mfwekind,
##     mgemleef, mgemomv, mgodge, mgodov, mgodpr, mgodrk, mhhuur, mhkoop,
##     mink123m, mink3045, mink4575, mink7512, minkgem, minkm30, mkoopkla,
##     moplhoog, mopllaag, moplmidd, moshoofd, mostype, mrelge, mrelov,
##     mrelsa, mska, mskb1, mskb2, mskc, mskd, mzfonds, mzpart, paanhang,
##     pbesaut, pbrand, pbrom, pbystand, pfiets, pgezong, pinboed, pleven,
##     pmotsco, ppersaut, ppersong, pplezier, ptractor, purchase, pvraaut,
##     pwabedr, pwaland, pwaoreg, pwapart, pwerkt, pzeilpl
```

```r
summary(purchase)
```

```
##   No  Yes
## 5474  348
```

The KNN classifier relies on identifying observations that are near one another, so the scale of the predictors is very important. Any variable that are on a large scale will have a much larger effect on the distance between the observations. For example, a difference of $1000 in salary has a much larger effect on the classifier than say a difference in 50 years of age. As a result, classifiers tend to overstate the importance of variables with large scales and understate smaller ones.

Because the KNN classifier predicts the class of a given test observations by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the *distance* between the observations, and hence on the KNN classifier, than variables that are on a small scale.

A good way to handle this problem is to *standardize* the data so that all the variables are given a mean of zero and a standard deviation of one. Then all variables will be on a comparable scale. The `scale()` function does just this. In standardizing the data, we exclude column 86 because tha tis the qualitative `purchase` variable.

```r
# All variables excluing purchase are numeric variables, so when we standardize the data frame, we igno
standardized.x <- scale(Caravan[,-86])
var(Caravan[,1])
```

```
## [1] 165.0378
```

```
var(Caravan[,2])
```

```
## [1] 0.1647078
```

```
var(standardized.x[,1])
```

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

```
var(standardized.x[,2])
```

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

Now every column of `standardized.X` has a standard deviation of one and a mean of zero.

We now split the observations into a test set, containing the first 1,000 observations and a training set, containing the remaining observations. We fit a KNN model on the training data using K = 1, and evaluate its performance on the test data.

```
# Now we split the observations into a test set, containing the first 1,000 observations, and a trainin
test <- 1:1000
train.x <- standardized.x[-test,]
test.x <- standardized.x[test,]
```

```
train.y <- purchase[-test]
test.y <- purchase[test]
```

```
set.seed(1)
knn.pred <- knn(train.x, test.x, train.y, k=1)
mean(test.y!=knn.pred)
```

```
## [1] 0.118
```

```
mean(test.y!="No")
```

```
## [1] 0.059
```

It seems like we have got a solid classifier - the test error rate is only 11.8%! However, only 6% of people actually bough insurance, so we could have an error rate down to 6% if we changed the classifier to always predict "No".

A vector `test` is numeric, which values from 1 through 1,000. Typing `standardized.X[test,]` yields the submatrix of the data containing the observations whose indices range from 1 to 1,000, whereas typing `standardized.X[-test,]` yields the submatrix containing the observations whose indices do *not* range from 1 to 1,000. The KNN error rate on the 1,000 test observations is just under 12%. At first glance, thiss may appear to be fairly good. However, since only 6% of our customers purchased insurance, we could get the error rate down to 6% by always predicting `No` regardless of the values of the predictors!

Suppose that there is some non-trivial cost to trying to sell insurance to a given individual. For instance, perhaps a salesperson must visit each potential customer. If the company tries to sell insurance to a random number of customers, then the success rate will only be 6%, which may be far too low given the costs involved. Instead, the company would like to try to sell insurance only to customers who are likely to buy it. So the overall error rate is not of interest. Instead, the fraction of individuals that are correctly predicted to buy insurance is of interest.

It turns out that KNN with K=1 does far better than random guessing among the customers that are predicted to buy insurance. Among 77 such customers, 9 or 11.8% actually do purchase insurance. This is double the rate that one would obtain from random guessing.

```
table(knn.pred, test.y)
```

```
##         test.y
```

```
## knn.pred  No Yes
##      No  873  50
##      Yes  68   9
```

```
mean(knn.pred != test.y)
```

```
## [1] 0.118
```

Using K = 3, the success rate increases to 19%, and with K = 5 the rate is 26.7%. This is over four times the rate that results from random guessing. It appears that KNN is finding some real patterns in a difficult dataset.

```
# Let's repeat the process using more neighbours to see if we get a more accurate classifier
knn.pred <- knn(train.x, test.x, train.y, k=3)
table(knn.pred, test.y)
```

```
##         test.y
## knn.pred  No Yes
##      No  920  54
##      Yes  21   5
```

```
mean(knn.pred == test.y)
```

```
## [1] 0.925
```

```
5/26
```

```
## [1] 0.1923077
```

```
knn.pred <- knn(train.x, test.x, train.y, k=5)
table(knn.pred, test.y)
```

```
##         test.y
## knn.pred  No Yes
##      No  930  55
##      Yes  11   4
```

```
mean(knn.pred == test.y)
```

```
## [1] 0.934
```

```
4/15
```

```
## [1] 0.2666667
```

As a comparison, we can also fit a logistic regression model to the data. If we use 0.5 as the predicted probability cut-off for the classifier, then we have a problem: only seven of the test observations are predicted to purchase insurance. Even worse, we are wrong about all of these! However, we are not required to do a cut-off of 0.5. If we instead predict a purchase any time the predicted probability of purchase exceeds 0.35, we get much better results: we predict that 33 people will purchase insurance, and we are correct for about 33% of these people. This is over five times better than random guessing!

```
# Now that we have our knn predictions, let's compare it to a logistic regression model using 0.5 as a
glm.fit <- glm(purchase ~., data=Caravan, family=binomial, subset=-test)
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
glm.probs <- predict(glm.fit, Caravan[test,], type="response")
glm.pred <- rep("No", 1000)
glm.pred[glm.probs>0.5] <- "Yes"
```

```r
table(glm.pred, test.y)
```

```
##         test.y
## glm.pred  No Yes
##      No  934  59
##      Yes   7   0
```

```r
mean(glm.pred == test.y)
```

```
## [1] 0.934
```

```r
# We actually did a terrible job predicting people buying the insurance, they are all wrong! Change the
glm.pred <- rep("No", 1000)
glm.pred[glm.probs>0.25] <- "Yes"
```

```r
table(glm.pred, test.y)
```

```
##         test.y
## glm.pred  No Yes
##      No  919  48
##      Yes  22  11
```

```r
mean(glm.pred == test.y)
```

```
## [1] 0.93
```