DSA 6100: Statistical Methods for Data Science and Analytics

Winter 2019

Lab Session 4

We will perform classifiers on the Smarket dataset which is part of the ISLR library. This dataset consists of percentage returns for 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 date in question) and Direction (whether the market was Up or Down on this date).

```
> library (ISLR)
> names(Smarket )
> dim(Smarket )
> summary (Smarket )
```

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

```
> cor(Smarket )
Error in cor(Smarket ) : ’x’ must be numeric
> cor(Smarket [,-9])
```

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 )
```

1. Logistic Regression

Next, we will fit a logistic regression model in order to predict Direction using Lag1 through Lag5 and Volume. The glm() function with the argument family=binomial tells R to run a logistic regression rather than some other type of generalized linear model.

```
> glm.fits=glm(Direction ~Lag1+Lag2+Lag3+Lag4+Lag5+Volume , data=Smarket ,family
=binomial )
> summary (glm.fits)
```

The smallest p-value here is associated with Lag1.
We use the coef() function in order to access just the coefficients for this fitted model. We can also use the summary() function to access particular aspects of the fitted model, such as the p-values for the coefficients.

```
> coef(glm.fits)
> summary (glm.fits)$coef
```

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 the logit. If no data set 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 ten 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 a 1 for Up.

```
> glm.probs =predict (glm.fits,type =" response ")
> glm.probs [1:10]
> 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.

```
> glm.pred=rep ("Down " ,1250)
> glm.pred[glm .probs >.5]=" Up"
```

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.

```
> table(glm .pred ,Direction )
Direction
glm .pred Down Up
Down 145 141
Up 457 507
> (507+145) /1250
[1] 0.5216
> mean(glm.pred== Direction )
[1] 0.5216
```

The diagonal elements of the confusion matrix indicate correct predictions, while the 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. 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. 100− 52.2 = 47.8% is the *training* 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.

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 data set of observations from 2005.

```
> train =(Year <2005)
> Smarket .2005= Smarket [! train ,]
> dim(Smarket .2005)
[1] 252 9
> Direction .2005= Direction [! train]
```

The object train is a vector of 1,250 elements, corresponding to the observations in our data set. The elements of the vector that correspond to observations that occurred before 2005 are set to TRUE.

We now fit a logistic 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 our test set—that is, for the days in 2005.

```
> glm.fits=glm(Direction ~Lag1+Lag2+Lag3+Lag4+Lag5+Volume ,
data=Smarket ,family =binomial ,subset =train )
> glm.probs =predict (glm .fits,Smarket .2005 , type=" response ")
```

Finally, we compute the predictions for 2005 and compare them to the actual movements of the market over that time period.

```
> glm.pred=rep ("Down " ,252)
> glm.pred[glm .probs >.5]=" Up"
> table(glm .pred ,Direction .2005)
Direction .2005
glm .pred Down Up
Down 77 97
Up 34 44
> mean(glm.pred== Direction .2005)
 [1] 0.48
> mean(glm.pred!= Direction .2005)
[1] 0.52
```

Below we have re-fit the logistic regression using just Lag1 and Lag2, which seemed to have the highest predictive power in the original logistic regression model.

```
> glm.fits=glm(Direction ~Lag1+Lag2 ,data=Smarket ,family =binomial , subset =train)
> glm.probs =predict (glm .fits,Smarket .2005 , type=" response ")
> glm.pred=rep ("Down " ,252)
> glm.pred[glm .probs >.5]=" Up"
> table(glm .pred ,Direction .2005)
Direction .2005
         glm .pred   Down Up
Down        35      35
Up          76        106
> mean(glm.pred== Direction .2005)
[1] 0.56
```

2. Linear Discriminant Analysis

Now we will perform LDA on the Smarket data. In R, we fit an LDA model using the lda() function, which is part of the MASS library. We fit the model using only the observations before 2005.

```
> library (MASS)
> lda.fit=lda(Direction~Lag1+Lag2 ,data=Smarket ,subset =train)
> lda.fit
> plot(lda.fit)
```

The *coefficients of linear discriminants* output provides the linear combination of Lag1 and Lag2 that are used to form the LDA decision rule. If −0.642×Lag1−0.514×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 −0.642 × Lag1 − 0.514 × 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 k^{th} column contains the posterior probability that the corresponding observation belongs to the *k*th class. Finally, x contains the linear discriminants

```
> lda.pred=predict (lda.fit , Smarket .2005)
> names(lda .pred)
[1] "class" "posterior " "x"
```

Applying a 50% threshold to the posterior probabilities allows us to recreate the predictions contained in lda.pred$class.

```
> sum(lda.pred$posterior [ ,1] >=.5)
[1] 70
> sum(lda.pred$posterior [,1]<.5)
[1] 182
```

If we wanted to use a posterior probability threshold other than 50% in order to make predictions, then we could easily do so. For instance, suppose that we wish to predict a market decrease only if we are very certain that the market will indeed decrease on that day—say, if the posterior probability is at least 90%.

```
> sum(lda.pred$posterior [,1]>.9)
[1] 0
```

No days in 2005 meet that threshold! In fact, the greatest posterior probability of decrease in all of 2005 was 52.02%.

3. Quadratic Discriminant Analysis

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().

```
> qda.fit=qda(Direction~Lag1+Lag2 ,data=Smarket ,subset =train)
> qda.fit
```

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.

```
> 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
> mean(qda.class == Direction .2005)
[1] 0.599
```

4. K-Nearest Neighbors

We will now perform KNN using the knn() function, which is part of the class library. knn() forms predictions using a single command. The function requires four inputs.

1. A matrix containing the predictors associated with the training data
2. A matrix containing the predictors associated with the data for which we wish to make predictions
3. A vector containing the class labels for the training observations
4. A value for *K*, the number of nearest neighbors to be used by the classifier.

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.

```
> library (class)
> 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 as nearest neighbors, then R will randomly break the tie. Therefore, a seed must be set in order to ensure reproducibility of results.

```
> 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
> (83+43) /252
[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 $K = 3$.

```
> 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
> mean(knn.pred== Direction .2005)
[1] 0.536
```

The results have improved slightly. But increasing K further turns out to provide no further improvements.

5. Support Vector Classifier

We use the e1071 library in R to demonstrate the support vector classifier and the SVM. We use the svm() function to fit the support vector classifier for a given value of the cost parameter.

We begin by generating the observations, which belong to two classes, and checking whether the classes are linearly separable.

```
> set.seed (1)
> x=matrix (rnorm (20*2) , ncol =2)
> y=c(rep (-1,10) , rep (1 ,10) )
> x[y==1 ,]= x[y==1,] + 1
> plot(x, col =(3-y))
```

They are not. Next, we fit the support vector classifier. Note that in order for the svm() function to perform classification, we must encode the response as a factor variable. We now create a data frame with the response coded as a factor.

```
> dat=data.frame(x=x, y=as.factor (y))
> library (e1071)
> svmfit =svm(y∼., data=dat , kernel =" linear ", cost =10, scale =FALSE )
```

The argument scale=FALSE tells the svm() function not to scale each feature to have mean zero or standard deviation one; depending on the application, one might prefer to use scale=TRUE. We can now plot the support vector classifier obtained:

```
> plot(svmfit , dat)
```

The region of feature space that will be assigned to the −1 class is shown in light blue, and the region that will be assigned to the +1 class is shown in purple. The decision boundary between the two classes is linear (because we used the argument kernel="linear")

The support vectors are plotted as crosses and the remaining observations are plotted as circles; we see here that there are seven support vectors. We can determine their identities as follows:

```
> svmfit$index
[1] 1 2 5 7 14 16 17
```

We can obtain some basic information about the support vector classifier fit using the summary() command:

```
> summary (svmfit )
```

The e1071 library includes a built-in function, tune(), to perform cross-validation. By default, tune() performs ten-fold cross-validation on a set of models of interest. In order to use this function, we pass in relevant information about the set of models that are under consideration. The following command indicates that we want to compare SVMs with a linear kernel, using a range of values of the cost parameter.

```
> set.seed (1)
> tune.out=tune(svm ,y∼.,data=dat ,kernel =" linear ", ranges =list(cost=c(0.001 , 0.01, 0.1,
1,5,10,100) ))
```

We can easily access the cross-validation errors for each of these models using the summary() command:

```
> summary (tune.out)
```

We see that cost=0.1 results in the lowest cross-validation error rate. The tune() function stores the best model obtained, which can be accessed as follows:

```
> bestmod =tune.out$best .model
> summary (bestmod )
```

The predict() function can be used to predict the class label on a set of test observations, at any given value of the cost parameter. We begin by generating a test data set.

```
> xtest=matrix (rnorm (20*2) , ncol =2)
> ytest=sample (c(-1,1) , 20, rep=TRUE)
> xtest[ytest ==1 ,]= xtest[ytest ==1,] + 1
> testdat =data.frame (x=xtest , y=as.factor (ytest))
```

Now we predict the class labels of these test observations. Here we use the best model obtained through cross-validation in order to make predictions.

```
> ypred=predict (bestmod ,testdat )
> table(predict =ypred , truth= testdat$y )
truth
predict -1 1
-1 11 1
1 0 8
```

6. Support Vector Machine

In order to fit an SVM using a non-linear kernel, we once again use the svm() function. However, now we use a different value of the parameter kernel. To fit an SVM with a polynomial kernel we use kernel="polynomial", and to fit an SVM with a radial kernel we use kernel="radial". In the former case we also use the degree argument to specify a degree for the polynomial kernel, and in the latter case we use gamma to specify a value of γ for the radial basis kernel.

We first generate some data with a non-linear class boundary, as follows:

```
> set.seed (1)
> x=matrix (rnorm (200*2) , ncol =2)
> x[1:100 ,]=x[1:100 ,]+2
> x[101:150 ,]= x[101:150 ,] -2
> y=c(rep (1 ,150) ,rep (2 ,50) )
> dat=data.frame(x=x,y=as.factor (y))
```

Plotting the data makes it clear that the class boundary is indeed nonlinear:

```
> plot(x, col=y)
```

The data is randomly split into training and testing groups. We then fit the training data using the svm() function with a radial kernel and $\gamma = 1$:

```
> train=sample (200 ,100)
> svmfit =svm(y∼., data=dat [train ,], kernel =" radial ", gamma =1, cost =1)
> plot(svmfit , dat[train ,])
```

The plot shows that the resulting SVM has a decidedly non-linear boundary. The summary() function can be used to obtain some information about the SVM fit:

```
> summary (svmfit )
```

If we increase the value of cost, we can reduce the number of training errors. However, this comes at the price of a more irregular decision boundary that seems to be at risk of overfitting the data. We can perform cross-validation using tune() to select the best choice of γ and cost for an SVM with a radial kernel:

```
> set.seed (1)
> tune.out=tune(svm , y∼., data=dat[train ,], kernel =" radial ", ranges =list(cost=c(0.1 ,1 ,10
,100 ,1000), gamma=c(0.5,1,2,3,4) ))
> summary (tune.out)
```