Project2: Predicting Multiple Discrete Values of Airline Data with Multinomial, Neural Networks and performing repeated Cross-Validation

**Structure of Analysis:**

**1-Reason for choosing Multinomial Model**

**2-Working with Neural Network Algorithm.**

**3-Repeated Cross-Validation**

**4-Conclusion**

(I have used the Airline Data of Project 1, thus skipped the steps to illustrate importing and pre-processing of data.)

Packages Used in this project

**{nnet}** - neural network multinomial modeling

**{caret}** - dummyVars and postResample function

**1-Reason for choosing Multinomial model**: Multinomial logistic regression is a simple extension of binary logistic regression that allows for more than two categories of the dependent or outcome .

library(nnet): Fits multinomial log-linear models via neural networks. In a nutshell, this allows to predict a factor of multiple levels , in one shot with the power of neural networks . **Neural networks** are great at working through multiple combinations and also great with linear models, so it's an ideal combination.

In the Airline data, we have re coded the factor ArrDelay as Delay with levels such as “On-Time”, “Short-Delay”, “Intermediate Delay”, “Long Delay”, “Uncertain Delay”. So ,instead of using multiple models and doing A versus B, B versus C ,and C versus A, and finally going through the hassle of concatenating the resulting probabilities the multinomial function predicts the probabilities for each level, all in one shot.

**2-Neural Network Algorithm**: We created a variable Delay, by breaking the “ ArrDelay” into five ranges , hence we create a multilevel categories for target class.

attach(d)

Delay <- cut(ArrDelay,breaks=c(-50,0, 15,30,45,1000),labels=c("On\_Time", "Almost\_On\_time", "Short\_Delay","Much\_delay", "Uncertain\_Delay"),right=FALSE)

d<-cbind(Delay,d)

We already have a random data, so we go on to split the data .The first half is assigned to dTrain and later to dTest. We will then assign the results of the multinomial model to DelayModel. The maxit is how may iterations you want neural network to do.

Since we have kept the trace on, we can see that the error value is decreasing until it reaches its minimum.

Let's put **nnet** to work and predict Delay. The maxiter variable defaults to 100 when omitted so let's start with a large number ie 500 during the first round to make sure we find the lowest possible error level (i.e. global minimum - solution with the lowest error possible.

d<-cbind(Delay,d)

split <- floor(nrow(d)/2)

dTrain <- d[0:split,]

dTest <- d[(split+1):nrow(d),]

DelayModel <- multinom(Delay~DepDelay+Distance+Dest+Origin +DepTime +DayOfWeek+DayofMonth, data=dTrain, maxit=500, trace=T)

# weights: 40 (28 variable)

initial value 1035658.811719

iter 10 value 813828.824163

iter 20 value 779026.063263

iter 30 value 668624.717000

iter 40 value 616285.818512

iter 50 value 616241.848993

iter 50 value 616241.846809

iter 50 value 616241.846732

final value 616241.846732

When you see the word **converged** in the log output, you know the model went as far as it could. Here we see that although we set a large value, it needed only 50 iterations to converge.

**Finding most influential variable**: Let's find the most influential variables by using **caret's** varImp function. We assign the Delay model to the varImp function ,which sorts the most important variable. We have a look at the first five.

mostImportantVariables <- varImp(DelayModel)

mostImportantVariables$Variables <- row.names(mostImportantVariables)

mostImportantVariables<-mostImportantVariables[order(mostImportantVariables$Overall),]

print(head(mostImportantVariables))

|  |
| --- |
| Overall Variables  DepDelay 1.282667963 DepDelay  DayOfWeek 0.343681376 DayOfWeek  DayofMonth 0.016184061 DayofMonth  Dest 0.009240655 Dest  Origin 0.005518291 Origin  Distance 0.001016410 Distance |
|  |
| |  | | --- | |  | |

The above output shows the Most Influential variable in modelling the Delay is DepDelay, followed by DayOfWeek ,DayOfMonth, Dest,Origin,Distance.

**Pedict Delay using class and probs :**Next we predict Delay using the predict function on the testing data set. There are two ways to compute predictions, class or probs: class is for the classification and probs is for probabailities.

We predict Delay using the predict function on the testing data set.

preds1 <- predict(DelayModel, type="probs", newdata=dTest)

head(preds1)

On\_Time Almost\_On\_time Short\_Delay Much\_delay Uncertain\_Delay

503 1.285534e-02 5.040531e-01 0.4498489291 0.02881229 0.004430371

504 1.838406e-20 2.826704e-11 0.0011106850 0.18440912 0.814480194

505 3.281893e-03 3.680932e-01 0.5777159616 0.04379303 0.007115955

506 5.830915e-05 4.166850e-02 0.7446768259 0.15409648 0.059499888

507 9.758503e-21 1.762627e-11 0.0008607714 0.13299468 0.866144550

508 6.772498e-10 8.364004e-05 0.2158876105 0.39704115 0.386987599

Here, we get the probabilities the model assigns to each of the five delay levels.

The highest probability is assigned to “Almost On Time”, similarly we can see the

probabilities for the second highest classes ie On Time comes next.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| preds2 <- predict(DelayModel, type="class", newdata=dTest)  > head(preds2)   |  | | --- | | [1] Almost\_On\_time On\_Time On\_Time On\_Time On\_Time On\_Time  Levels: On\_Time Almost\_On\_time Short\_Delay Much\_delay Uncertain\_Delay  As we can see, using class we predict our flight Delay is “Almost on time” for row1.  “On Time” for row 2 , “On Time” for row3 and so on. | |  | | |  | | --- | |  | | |
|  |
| |  | | --- | |  | |

Choosing which of the two predictions will depend on your needs. If you just want your

Delay predictions, use class, if you need to do anything more complex, like measure

the conviction of each prediction, use the probs option (every row will add up to 1).

To check the accuracy of the model, we call the postResample function from caret. For

numeric vectors, it uses the mean squared error and R-squared and for factors, the

overall agreement rate and Kappa:

postResample(dTest$Delay,preds2)

Accuracy Kappa

0.608871 0.365616

**3.Cross validation**: This method randomly splits the dataset into training and validation data. For each such split, the model is fit to the training data, and predictive accuracy is assessed using the validation data. The results are then averaged over the splits. The advantage of this method (over *k*-fold cross validation) is that the proportion of the training/validation split is not dependent on the number of iterations (folds).

The Caret package has some very good tools for cross-validation , bootstrapping, leave one out etc.

Lets do some simple repeated cross validation to get a more comprehensive mean accuracy score and understand convergence. The code below will divide the dataset into 10 portions. The loop below will iterate through the dataset 10 times ,each time it will take one-tenth of dataset as testing and rest as training. This will allow us to test our model on ten different portions of data, giving every variable a chance of being test and train data sets. This helps us alleviate the very serious problem of “over fitting” The first time around we set maxit to only 100:

totalAccuracy <- c()

cv <- 10

cvDivider <- floor(nrow(d) / (cv+1))  
> for (cv in seq(1:cv)) {

+ # assign chunk to data test

+ dataTestIndex <- c((cv \* cvDivider):(cv \* cvDivider + cvDivider))

+ dataTest <- d[dataTestIndex,]

+ # everything else to train

+ dataTrain <- d[-dataTestIndex,]

+

+ DelayModel <- multinom(Delay~., data=dataTrain, maxit=100, trace=T)

+

+ pred <- predict(DelayModel, newdata=dataTest, type="class")

We check our classification error as a distance between the actual vs predicted below

+ # classification error

+ cv\_ac <- postResample(dataTest$Delay, pred)[[1]]

+ print(paste('Current Accuracy:',cv\_ac,'for CV:',cv))

+ totalAccuracy <- c(totalAccuracy, cv\_ac)

+ }

# weights: 80 (60 variable)

initial value 1461.369624

iter 10 value 1125.594455

iter 20 value 907.143128

iter 30 value 815.265155

iter 40 value 784.836972

iter 50 value 668.561224

iter 60 value 346.138032

iter 70 value 65.972074

iter 80 value 28.814165

iter 90 value 7.464957

iter 100 value 3.177820

final value 3.177820

stopped after 100 iterations

[1] "Current Accuracy: 0.945054945054945 for CV: 1"

stopped after 100 iterations

:

:

:

:

:

:

# weights: 80 (60 variable)

initial value 1461.369624

iter 10 value 1107.023948

iter 20 value 817.208849

iter 30 value 696.354690

iter 40 value 667.944415

iter 50 value 532.086689

iter 60 value 385.768911

iter 70 value 76.584741

iter 80 value 18.418497

iter 90 value 8.361572

iter 100 value 3.270726

final value 3.270726

stopped after 100 iterations

[1] "Current Accuracy: 0.945054945054945 for CV: 10"

mean(totalAccuracy)

[1] 0.9681319

The mean accuracy of 0.96 is much higher than the accuracy that we got with the original

simple split(.60). We notice that the log output never prints the word converged.

This means the model never reaches the lowest error or global minima and therefore

isn't the best fit.

Let's try this again and let the model converge by setting the maxit to a large number

Ie 1000 this time. That means the model will go through 1000 iterations (instead of 100

Last time)

totalAccuracy <- c()

cv <- 10

cvDivider <- floor(nrow(d) / (cv+1))

for (cv in seq(1:cv)) {

+ # assign chunk to data test

+ dataTestIndex <- c((cv \* cvDivider):(cv \* cvDivider + cvDivider))

+ dataTest <- d[dataTestIndex,]

+ # everything else to train

+ dataTrain <- d[-dataTestIndex,]

+

+ DelayModel <- multinom(Delay~., data=dataTrain, maxit=1000, trace=T)

+

+ pred <- predict(DelayModel, newdata=dataTest, type="class")

+

+ # classification error

+ cv\_ac <- postResample(dataTest$Delay, pred)[[1]]

+ print(paste('Current Accuracy:',cv\_ac,'for CV:',cv))

+ totalAccuracy <- c(totalAccuracy, cv\_ac)

+ }

# weights: 80 (60 variable)

initial value 1461.369624

iter 10 value 1125.594455

iter 20 value 907.143128

iter 30 value 815.265155

iter 40 value 784.836972

iter 50 value 668.561224

iter 60 value 346.138032

iter 70 value 65.972074

iter 80 value 28.814165

iter 90 value 7.464957

iter 100 value 3.177820

iter 110 value 0.124361

iter 120 value 0.000231

final value 0.000006

converged

[1] "Current Accuracy: 0.945054945054945 for CV: 1"

:

:

:

:

:

:

# weights: 80 (60 variable)

initial value 1461.369624

iter 10 value 1107.023948

iter 20 value 817.208849

iter 30 value 696.354690

iter 40 value 667.944415

iter 50 value 532.086689

iter 60 value 385.768911

iter 70 value 76.584741

iter 80 value 18.418497

iter 90 value 8.361572

iter 100 value 3.270726

iter 110 value 0.068401

final value 0.000055

converged

[1] "Current Accuracy: 0.927065934065934 for CV: 10"

mean(totalAccuracy)

[1] 0.9692308

**4-Conclusion**:Within the logistic regression, we found several variables that were statistically significant, like Distance, Day Of Week, Origin, Destination, Departure-time, Arrival-time, Day

Of Month and etc.

Letting the model iterate before it converges will clearly give less errors.

The score using the repeated cross validation code is a better than the original simple split of 0.6084 and we let each loop converge. The point of using the repeated cross

Validation code isn't that it will return a higher accuracy score (and it doesn't always)

but that it will give a much more accurate score as it uses all of the data.

Repeated cross validation, leave one out, Bootstrapping are the methods that are very important to get the least possible error prediction, and the model is as close as possible to real data and at anticipating future data. Like in our dataset, we got a little better accuracy of .969 in the repeated cross validation than in the initial cross validation .927.