Model Validation

So far we have fitted various models and evaluated their performance using a number of metrics such as RMSE, AIC, BIC, Sensitivity… So far, we fit the model and evaluated its performance on the same dataset. This is perfectly acceptable if you have no intention to use the model for predictions on out sample data. This is the approach used in most scientific studies. For practical purposes however, the prediction out of sample is often the sole purpose of fiting the model in the first place.

The main problem with fitting and evaluating the model performance on a single dataset is that you have no way of knowing the performance of the model in new data. This is a real concern considering the overfitting problem. With any modeling approach we learned thus far, there is the risk of overfitting. An overfitted model may adopt to the idiosyncracies of the sample dataset so well that it may not perform well for out sample data. Using only metrics from one dataset, we may not be able to detect overfitted models.

In this learning activity we will start with how to conduct validation for predictive modeling using simple splits.

Simple Split

One way to adress this problem is to split the data to enable testing of model performance. Say we use only 70% of the data to train the model and use the rest of the data to test its performance. We need a way to split the data into two groups.

I will be using the titanic dataset from the Module 4, Neural Nets learning activity. The titanic package already provides a separate testing and training set. For demonstration purposes however, I will spliting the training dataset.

```r
# Import data as usual
titanic <- titanic::titanic_train
# Standardize continuous columns
# Create a list of columns that should be factors
factorColumns <- c("Survived", "Pclass", "Sex", "Name",  "Ticket", "Cabin", "Embarked")
notFactorColumns <- !(colnames(titanic) %in% factorColumns)
titanicStd <- data.frame(titanic[,factorColumns], scale(titanic[,notFactorColumns]))
# Feed the list of Factors to lapply, to turn these into factors
titanicStd[,factorColumns] <- lapply(titanicStd[,factorColumns], as.factor)
```

Split à la Vanilla R

The idea is to randomly select a portion (typically 30%) of the dataset for the testing. We will use the remainder (70%) for training. Let us see how this can be achieved with vanilla R.

```r
# How many observations does data have?
nrow(titanicStd)
```

```
## [1] 891
```

```r
# What is 30% of this? Round it so that it is an integer.
size <- round(nrow(titanicStd) * .3, digits = 0)
# let us randomly sample that many row numbers
testingIndex <- sample(1:nrow(titanicStd), size)
```

Now we have 267 random numbers ranging from 1 to 891. We can use these as row indices to get a subset of the data for testing purposes.

```r
# Split testing and training
titanicStdTest <- titanicStd[testingIndex,]
titanicStdTrain <- titanicStd[-testingIndex,]
```

Above is how you create a very basic test and training set using only vanilla R functionality. I think above code demonstrates the logic of the process.

Split à la Caret Package

In caret package, there is *createDataPartition()* function family that allows you to split data as well. The benefit of using *createDataPartition()* over vanilla R functionality is that it allows for more complex split (stratified, or time series data, bootstraping, etc.) schemes to be carried out relatively easily.

For example if we wanted the training and testing samples to have same proportion of survivors we can ask *createDataPartition* to split data taking the outcomes into account.

```r
# Create a balanced split based on outcome
# sample sizes are different due to rounding differences
testingIndex <- createDataPartition(titanicStd$Survived, p = .3, list = FALSE)
# Split testing and training
titanicStdTest <- titanicStd[testingIndex,]
titanicStdTrain <- titanicStd[-testingIndex,]
```

Evaluating Out Sample Performance

Once the data is split, we need to fit a model to just the training data and then evaluate the performance of so fitted model in predicting the testing data. You may want to brush up on fitting a neural network for a binary outcome variable from Module 4.

Let us first fit the model to the traning dataset.

```r
# Model Matrix for categorical variables handling
titanicStdTrainMM <- model.matrix(~ Survived + Pclass + Sex + Age , titanicStdTrain)
# Things get easier if we omit the dependent variable here
titanicStdTestMM <- model.matrix(~ Pclass + Sex + Age , titanicStdTest)
```

Note that the sample size went down for both groups. Some observations have missing values. Those are dropped. If the reduction in sample size is worrisome, you may consider imputation as discussed in Module 2.

Fit the Model.

```r
titanicStdTrainMM_nn_0 <- neuralnet(Survived1 ~ Age + Sexmale + Pclass2 + Pclass3, data = titanicStdTra
```

Predict outcomes of test data.

```r
# The Model Matrix has intercept as the first column, remove it.
# You only want the predictors to go into the compute function.
# The probabilities estimated are going to be in net.result
predicted <- compute(titanicStdTrainMM_nn_0, titanicStdTestMM[,-1])

# Remember the reduction in sample size? Let us drop missing values to match predicted
titanicStdTested <- na.exclude(titanicStdTest[,c("Survived", "Age", "Sex", "Pclass")])
# Let's put estimated probabilities in
titanicStdTested$PredictedProbability <- predicted$net.result
# Code probabilities so we get estimated outcome
titanicStdTested$Predicted <- round(titanicStdTested$PredictedProbability, digits = 0)
```

Finally, you evaluate this prediction's performance.

```
titanicTestConfusionMatrix <- table(titanicStdTested$Survived, titanicStdTested$Predicted)
confusionMatrix(titanicTestConfusionMatrix)
```

```
## Confusion Matrix and Statistics
##
##
##       0   1
##   0 102  19
##   1  54  42
##
##                Accuracy : 0.6635945
##                  95% CI : (0.5965136, 0.7261344)
##     No Information Rate : 0.718894
##     P-Value [Acc > NIR] : 0.9687476
##
##                   Kappa : 0.2914523
##  Mcnemar's Test P-Value : 0.0000690897
##
##             Sensitivity : 0.6538462
##             Specificity : 0.6885246
##          Pos Pred Value : 0.8429752
##          Neg Pred Value : 0.4375000
##              Prevalence : 0.7188940
##          Detection Rate : 0.4700461
##    Detection Prevalence : 0.5576037
##       Balanced Accuracy : 0.6711854
##
##        'Positive' Class : 0
##
```

The model looks medicore at best when tested on an out sample.

Here is a simple exercise for you. Compare the output of *confusionMatrix()* above to the output of the confusion matrix for the predictions on the training set. Comment on results. Why would this be?

What you want to do at this stage is to try to configure your neural network to increase accuracy. You can change number of hidden neurons and layers. You can change variables that go into the model. If you suspect over fitting, you may adjust threshold parameter so the fit would not be as precise…

Solutions to Exercises

1 - Compare the output of *confusionMatrix()* above to the output of the confusion matrix for the predictions on the training set.

```
# Missing values need to be excluded
titanicStdTrained <- na.exclude(titanicStdTrain[,c("Survived", "Age", "Sex", "Pclass")])
titanicStdTrained$PredictedProbability <- titanicStdTrainMM_nn_0$net.result[[1]]
# Code probabilities so we get estimated outcome
titanicStdTrained$Predicted <- round(titanicStdTrained$PredictedProbability, digits = 0)
# Create table
titanicTrainedConfusionMatrix <- table(titanicStdTrained$Survived, titanicStdTrained$Predicted)
confusionMatrix(titanicTrainedConfusionMatrix)
```

```
## Confusion Matrix and Statistics
```

```
##
##
##       0   1
##   0 288  15
##   1  72 122
##
##                Accuracy : 0.8249497
##                  95% CI : (0.788629, 0.8573486)
##     No Information Rate : 0.7243461
##     P-Value [Acc > NIR] : 0.000000103707048
##
##                   Kappa : 0.6116874
##  Mcnemar's Test P-Value : 0.000000001927163
##
##             Sensitivity : 0.8000000
##             Specificity : 0.8905109
##          Pos Pred Value : 0.9504950
##          Neg Pred Value : 0.6288660
##              Prevalence : 0.7243461
##          Detection Rate : 0.5794769
##    Detection Prevalence : 0.6096579
##       Balanced Accuracy : 0.8452555
##
##        'Positive' Class : 0
##
```

Here, we have a model that fits the training set perfectly, yet fails dismally in the testing set. It is natural that the model fits the training data better than the testing data. This is essentially the main reason why validation is such an important concept.