In this example, we perform tuning of an MLP using `nnet` and `neuralnet` for a classification problem.

Although the neuralnet package can deal directly with classification problems, including factor-valued variables, this is not possible within the caret framework and we have to perform a one-hot encoding on the data in a preprocessing step. This encoding converts class values to numerical values, equal to either zero or one. In the case of the iris data, where there are 3 possible levels of the `Species` factor—the response variable—we need to add 3 columns, one corresponding to each level/species, whose values will be either 1 or 0 depending on whether the given observation belongs to a given species, or not. Then the classification problem can be treated as a regression problem by caret's `neuralnet` method.

We use the Iris Data Set downloaded from the kaggle site [][https://www.kaggle.com/uciml/iris].

Step1: Read in the required packages and data.

```
#Load required packages
library(neuralnet)
library(caret)
library(reshape2)
library(ggplot2)
library(stringr)
#Read  data
irisData <- read.csv(file="iris.csv", head = TRUE, sep = ",")
```

Step 2: (EDA) Inspect the data.

```
#Get a summary of the data
summary(irisData)
```

```
##        Id         SepalLengthCm    SepalWidthCm   PetalLengthCm
##  Min.   :  1.00   Min.   :4.300   Min.   :2.000   Min.   :1.000
##  1st Qu.: 38.25   1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
##  Median : 75.50   Median :5.800   Median :3.000   Median :4.350
##  Mean   : 75.50   Mean   :5.843   Mean   :3.054   Mean   :3.759
##  3rd Qu.:112.75   3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
##  Max.   :150.00   Max.   :7.900   Max.   :4.400   Max.   :6.900
##   PetalWidthCm     Species
##  Min.   :0.100   Length:150
##  1st Qu.:0.300   Class :character
##  Median :1.300   Mode  :character
##  Mean   :1.199
##  3rd Qu.:1.800
##  Max.   :2.500
```

```
str(irisData)
```

```
## 'data.frame':    150 obs. of  6 variables:
##  $ Id           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ SepalLengthCm: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ SepalWidthCm : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ PetalLengthCm: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ PetalWidthCm : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species      : chr  "Iris-setosa" "Iris-setosa" "Iris-setosa" "Iris-setosa" ...
```

Step 3: (EDA) Check the data for any missing values. There are multiple ways to do this but. Here, we use the `colsums` function to determine how many missing values are in each column, if any.

```
#Compute total number of NA values in each column. Note, this  will be tricked if there are dummy values
colSums(is.na(irisData))
```

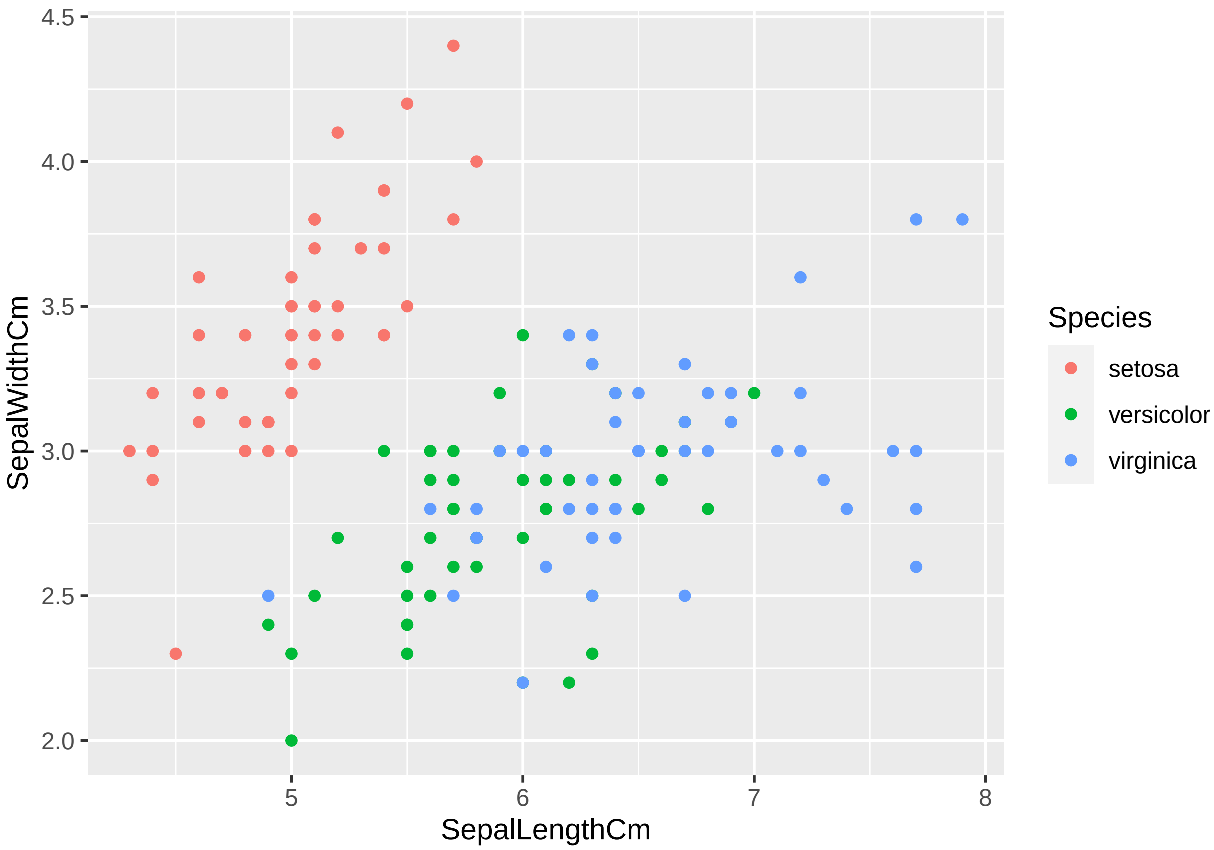
```
##            Id SepalLengthCm  SepalWidthCm PetalLengthCm  PetalWidthCm
##             0             0             0             0             0
##       Species
##             0
```

Step 4: Perform any required data cleaning. Here, we remove the "Iris-" from the species column. Otherwise, if left in, could cause difficulties in constructing the model formula automatically.

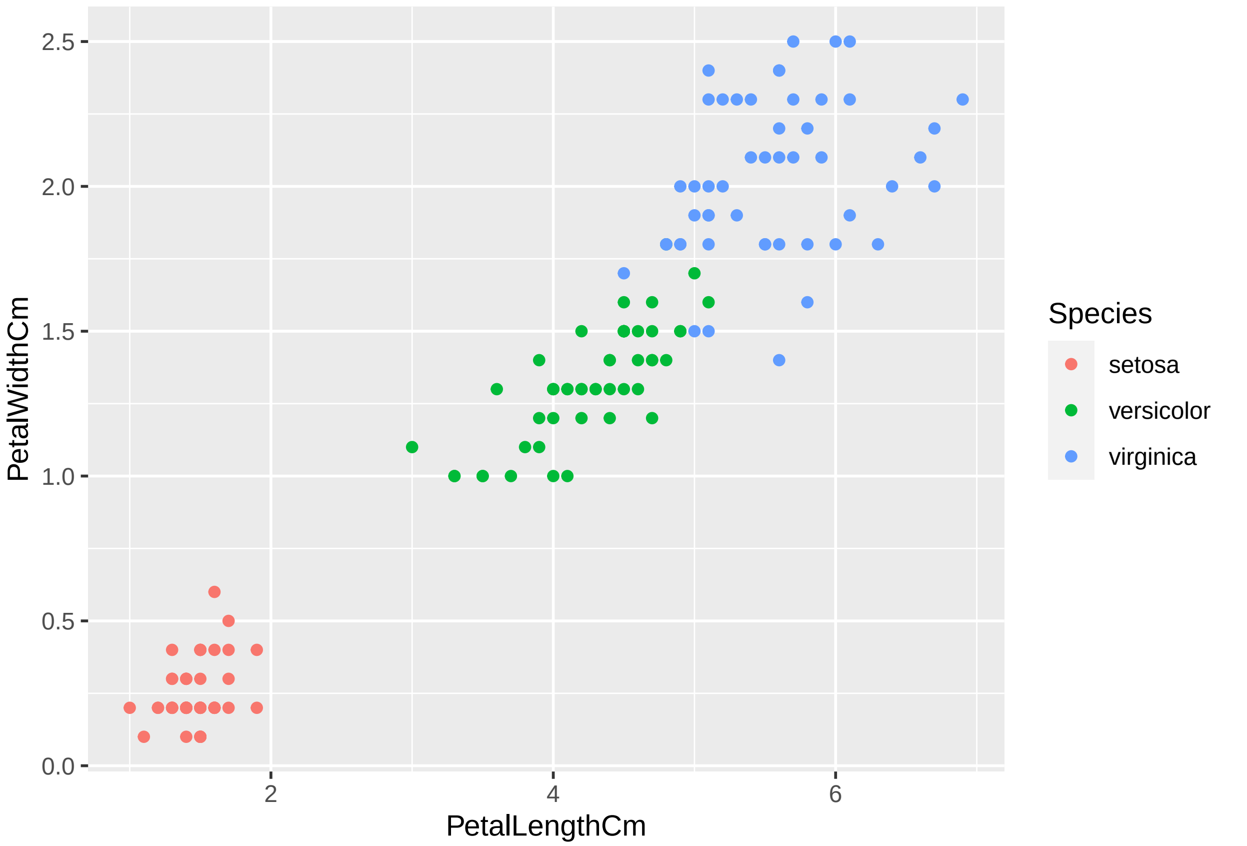
```
irisData$Species <- str_replace(irisData$Species, "Iris-","")
```

Step 5: Look at the data graphically, to choose the best predictors for a model.

```
ggplot(irisData, aes(SepalLengthCm, SepalWidthCm, color = Species)) + geom_point()
```




```
ggplot(irisData, aes(PetalLengthCm, PetalWidthCm, color = Species)) + geom_point()
```



Step 6: To use the measurements in the neural network, it is indispensable to normalize or scale them. If it is not scaled, the model will either be difficult to train—will not converge—or will result in uninterpretable results. We only scale the feature columns.

```
irisData[,2:5] <- scale(irisData[,2:5])
```

Step 7: Because the `neuralnet` method within `caret` does not accept factors, the outcome variable needs to be adjusted. We use the `cbind` command to perform one-hot encoding, thus adding 3 columns with transformations of the outcome variable into target vectors.

```
irisData <- cbind(irisData, model.matrix(~ 0 + Species, irisData))
# check the one-hot encoding
head(irisData[,6:9])
```

```
##   Species Speciessetosa Speciesversicolor Speciesvirginica
## 1  setosa             1                 0                0
## 2  setosa             1                 0                0
## 3  setosa             1                 0                0
## 4  setosa             1                 0                0
## 5  setosa             1                 0                0
## 6  setosa             1                 0                0
```

```
tail(irisData[,6:9])
```

```
##       Species Speciessetosa Speciesversicolor Speciesvirginica
## 145 virginica             0                 0                1
## 146 virginica             0                 0                1
## 147 virginica             0                 0                1
## 148 virginica             0                 0                1
```

```
## 149 virginica              0                  0                 1
## 150 virginica              0                  0                 1
```

Step 8: Split the data into training and testing data in the proportion 75% for training and 25% for testing.

```
#set the seed to get reproducible results
set.seed(12345)
inTrain <- createDataPartition(y=irisData$Species, p=0.75, list=FALSE)
train.set <- irisData[inTrain,]
test.set  <- irisData[-inTrain,]
```

Step 9: Create the model formula.

```
#Get the variable names for the input and output
predictorVars <- names(irisData)[2:5]
outcomeVars   <- names(irisData)[7:9]
#Paste together the formula
modFormula <- as.formula(paste(paste(outcomeVars, collapse = "+"),
                               "~", paste(predictorVars, collapse = " + ")))
modFormula
```

```
## Speciessetosa + Speciesversicolor + Speciesvirginica ~ SepalLengthCm +
##     SepalWidthCm + PetalLengthCm + PetalWidthCm
```

Step 10: Train the network. There are no fixed rules for the hyperparameters, threshold, steps and hidden layers. These can be tuned in order to improve the network performance.

```
irisNNet <- neuralnet(formula = modFormula,
                      data = train.set,
                      hidden = c(4),
                      linear.output = FALSE,
                      threshold = .05,
                      stepmax = 5000)
```

Step 11: See how well the network did by computing its classification accuracy.

```
classes <- compute(irisNNet, test.set)
#Get the classification results out of classes
classRes <- classes$net.result
#Using the apply funciton in conjunction with the 'which.max' function to get the max index
#for each row of the classRes matrix and the test rows of original data
nnClass <- apply(classRes, MARGIN = 1, which.max)
origClass <- apply(test.set[, c(7:9)], MARGIN = 1, which.max)
#Compute the percentage of correct classification for the neural network:
# 'round' is testing pr > 0.5 ?
# 'mean' gives the probability of each class
# '100' gives the percentage
paste("The classification accuracy of the network is",
      round(mean(nnClass == origClass) * 100, digits = 2), "%")
```

```
## [1] "The classification accuracy of the network is 97.22 %"
```

Although these seem like good results this may simply be a result of the partition into training and testing data, so it is important to test the model performance further. Here, we manually perform k-fold cross validation using 10 folds (10 fold cross-validation).

Step 12: Validate the Model

```
#Compute the testing indices using the createFolds function of caret
folds <- createFolds(irisData$Species, k = 10)
```

```r
#'results' is a vector that will contain the accuracy for each fold
# of the network training and testing
results <- c()
for (fld in folds){
  # train the network
  nn <- neuralnet(formula = modFormula, data = irisData[-fld,], hidden = c(4),
                  linear.output = FALSE, threshold = .05, stepmax = 5000)
  # extract the classifications from the network
  classes <- compute(nn, irisData[fld ,-c(1,6:9)])
  # check the accuracy of the network using the same procedure as above
  classRes <- classes$net.result
  nnClass <- apply(classRes, MARGIN = 1, which.max)
  origClass <- apply(irisData[fld , c(7:9)], MARGIN = 1, which.max)
  results <- c(results, mean(nnClass == origClass) * 100)
}
# outoput the result
paste("After", length(results),
      "validation loops the mean accuracy of the network is",
      paste0(round(mean(results),2), "%"))
```

```
## [1] "After 10 validation loops the mean accuracy of the network is 96%"
```

Now, perform CV using the `caret` package.

```r
# Define the tuning grid
grid <-  expand.grid(size=c(2,4), decay=2^(-3:-1))
# define the CV strategy
tr.nnet <- trainControl(method = "cv",
                        number = 5,
                        verboseIter = TRUE)
# Train with `caret`
nn1 <- train(Species ~., #formula = modFormula,
            data = train.set[,1:6],
            method="nnet",#method = "neuralnet",
            tuneGrid = grid,
            preProc = c("center", "scale", "nzv"),
            trControl = tr.nnet)
```

```
## + Fold1: size=2, decay=0.125
## # weights:  21
## initial  value 111.704526
## iter  10 value 22.888788
## iter  20 value 18.663312
## final  value 18.661892
## converged
## - Fold1: size=2, decay=0.125
## + Fold1: size=4, decay=0.125
## # weights:  39
## initial  value 98.988268
## iter  10 value 19.480327
## iter  20 value 17.078020
## iter  30 value 17.072665
## final  value 17.072665
## converged
## - Fold1: size=4, decay=0.125
```

```
## + Fold1: size=2, decay=0.250
## # weights:  21
## initial  value 105.585293
## iter  10 value 35.050588
## iter  20 value 29.152157
## iter  30 value 29.105871
## final  value 29.105871
## converged
## - Fold1: size=2, decay=0.250
## + Fold1: size=4, decay=0.250
## # weights:  39
## initial  value 129.079239
## iter  10 value 27.820488
## iter  20 value 24.089594
## iter  30 value 23.811317
## iter  40 value 23.810869
## iter  40 value 23.810869
## iter  40 value 23.810869
## final  value 23.810869
## converged
## - Fold1: size=4, decay=0.250
## + Fold1: size=2, decay=0.500
## # weights:  21
## initial  value 110.784584
## iter  10 value 48.150471
## iter  20 value 44.088273
## final  value 44.082612
## converged
## - Fold1: size=2, decay=0.500
## + Fold1: size=4, decay=0.500
## # weights:  39
## initial  value 114.648922
## iter  10 value 40.405027
## iter  20 value 37.760951
## iter  30 value 37.755432
## final  value 37.755416
## converged
## - Fold1: size=4, decay=0.500
## + Fold2: size=2, decay=0.125
## # weights:  21
## initial  value 101.039006
## iter  10 value 24.215441
## iter  20 value 18.680372
## iter  30 value 18.599284
## final  value 18.599114
## converged
## - Fold2: size=2, decay=0.125
## + Fold2: size=4, decay=0.125
## # weights:  39
## initial  value 104.890567
## iter  10 value 18.002740
## iter  20 value 14.957767
## iter  30 value 14.876416
## iter  40 value 14.876384
```

```
## final  value 14.876384
## converged
## - Fold2: size=4, decay=0.125
## + Fold2: size=2, decay=0.250
## # weights:  21
## initial  value 104.384683
## iter  10 value 37.346807
## iter  20 value 29.154016
## final  value 29.033976
## converged
## - Fold2: size=2, decay=0.250
## + Fold2: size=4, decay=0.250
## # weights:  39
## initial  value 100.027669
## iter  10 value 35.057910
## iter  20 value 24.348119
## iter  30 value 23.827182
## iter  40 value 23.825065
## final  value 23.825055
## converged
## - Fold2: size=4, decay=0.250
## + Fold2: size=2, decay=0.500
## # weights:  21
## initial  value 104.032192
## iter  10 value 47.285594
## iter  20 value 43.284413
## final  value 43.283151
## converged
## - Fold2: size=2, decay=0.500
## + Fold2: size=4, decay=0.500
## # weights:  39
## initial  value 117.160549
## iter  10 value 38.694214
## iter  20 value 36.632848
## iter  30 value 36.600448
## final  value 36.600434
## converged
## - Fold2: size=4, decay=0.500
## + Fold3: size=2, decay=0.125
## # weights:  21
## initial  value 105.844547
## iter  10 value 21.997533
## iter  20 value 18.428592
## final  value 18.421533
## converged
## - Fold3: size=2, decay=0.125
## + Fold3: size=4, decay=0.125
## # weights:  39
## initial  value 104.304705
## iter  10 value 21.587076
## iter  20 value 14.796379
## iter  30 value 14.396387
## iter  40 value 14.361763
## final  value 14.360886
```

```
## converged
## - Fold3: size=4, decay=0.125
## + Fold3: size=2, decay=0.250
## # weights:  21
## initial  value 101.594902
## iter  10 value 42.842600
## iter  20 value 28.488282
## iter  30 value 28.466019
## final  value 28.465965
## converged
## - Fold3: size=2, decay=0.250
## + Fold3: size=4, decay=0.250
## # weights:  39
## initial  value 101.645135
## iter  10 value 25.947652
## iter  20 value 23.199019
## iter  30 value 23.140566
## final  value 23.140435
## converged
## - Fold3: size=4, decay=0.250
## + Fold3: size=2, decay=0.500
## # weights:  21
## initial  value 102.870560
## iter  10 value 54.361393
## iter  20 value 42.701354
## final  value 42.695284
## converged
## - Fold3: size=2, decay=0.500
## + Fold3: size=4, decay=0.500
## # weights:  39
## initial  value 108.580974
## iter  10 value 39.889151
## iter  20 value 37.255652
## iter  30 value 37.150848
## final  value 37.150837
## converged
## - Fold3: size=4, decay=0.500
## + Fold4: size=2, decay=0.125
## # weights:  21
## initial  value 110.299877
## iter  10 value 20.140186
## iter  20 value 18.689777
## iter  30 value 18.682193
## final  value 18.682190
## converged
## - Fold4: size=2, decay=0.125
## + Fold4: size=4, decay=0.125
## # weights:  39
## initial  value 106.396875
## iter  10 value 17.297860
## iter  20 value 15.918761
## iter  30 value 15.906395
## final  value 15.905483
## converged
```

```
## - Fold4: size=4, decay=0.125
## + Fold4: size=2, decay=0.250
## # weights:  21
## initial  value 109.165417
## iter  10 value 34.378097
## iter  20 value 29.616229
## final  value 29.605795
## converged
## - Fold4: size=2, decay=0.250
## + Fold4: size=4, decay=0.250
## # weights:  39
## initial  value 109.314739
## iter  10 value 32.209350
## iter  20 value 25.366350
## iter  30 value 25.245459
## final  value 25.244170
## converged
## - Fold4: size=4, decay=0.250
## + Fold4: size=2, decay=0.500
## # weights:  21
## initial  value 101.645990
## iter  10 value 48.505996
## iter  20 value 43.402033
## final  value 43.394254
## converged
## - Fold4: size=2, decay=0.500
## + Fold4: size=4, decay=0.500
## # weights:  39
## initial  value 136.869115
## iter  10 value 38.865227
## iter  20 value 36.875254
## iter  30 value 36.735868
## final  value 36.735388
## converged
## - Fold4: size=4, decay=0.500
## + Fold5: size=2, decay=0.125
## # weights:  21
## initial  value 100.813309
## iter  10 value 23.571977
## iter  20 value 18.898555
## iter  30 value 18.892759
## final  value 18.892757
## converged
## - Fold5: size=2, decay=0.125
## + Fold5: size=4, decay=0.125
## # weights:  39
## initial  value 114.829664
## iter  10 value 19.404490
## iter  20 value 14.743382
## iter  30 value 14.600919
## iter  40 value 14.596945
## final  value 14.596938
## converged
## - Fold5: size=4, decay=0.125
```

```
## + Fold5: size=2, decay=0.250
## # weights:  21
## initial  value 107.851205
## iter  10 value 29.313300
## final  value 29.147210
## converged
## - Fold5: size=2, decay=0.250
## + Fold5: size=4, decay=0.250
## # weights:  39
## initial  value 110.584053
## iter  10 value 29.561501
## iter  20 value 23.817242
## iter  30 value 23.428513
## final  value 23.424196
## converged
## - Fold5: size=4, decay=0.250
## + Fold5: size=2, decay=0.500
## # weights:  21
## initial  value 111.858424
## iter  10 value 45.177292
## iter  20 value 43.653529
## final  value 43.652173
## converged
## - Fold5: size=2, decay=0.500
## + Fold5: size=4, decay=0.500
## # weights:  39
## initial  value 90.978647
## iter  10 value 38.250156
## iter  20 value 37.238378
## iter  30 value 37.233916
## final  value 37.233911
## converged
## - Fold5: size=4, decay=0.500
## Aggregating results
## Selecting tuning parameters
## Fitting size = 2, decay = 0.5 on full training set
## # weights:  21
## initial  value 139.779488
## iter  10 value 56.260964
## iter  20 value 47.783685
## iter  30 value 47.718207
## final  value 47.718205
## converged
```

```
# display best model
nn1
```

```
## Neural Network
##
## 114 samples
##   5 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## Pre-processing: centered (5), scaled (5)
## Resampling: Cross-Validated (5 fold)
```

```
## Summary of sample sizes: 92, 91, 92, 91, 90
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##   2     0.125  1.0000000  1.0000000
##   2     0.250  1.0000000  1.0000000
##   2     0.500  1.0000000  1.0000000
##   4     0.125  1.0000000  1.0000000
##   4     0.250  1.0000000  1.0000000
##   4     0.500  0.9909091  0.9863777
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 2 and decay = 0.5.
```

```r
# predict and display confusion matrix
pred_nnet <- predict(nn1,test.set)
table(pred_nnet, test.set$Species)
```

```
##
## pred_nnet    setosa versicolor virginica
##   setosa         12          0         0
##   versicolor      0         12         0
##   virginica       0          0        12
```

The predictive precision is excellent!

We could also tune and cross-validate with the `neuralnet` method that provides more options for the network architecture.