Exercise:
Neural Networks

Introduction to Machine Learning with R

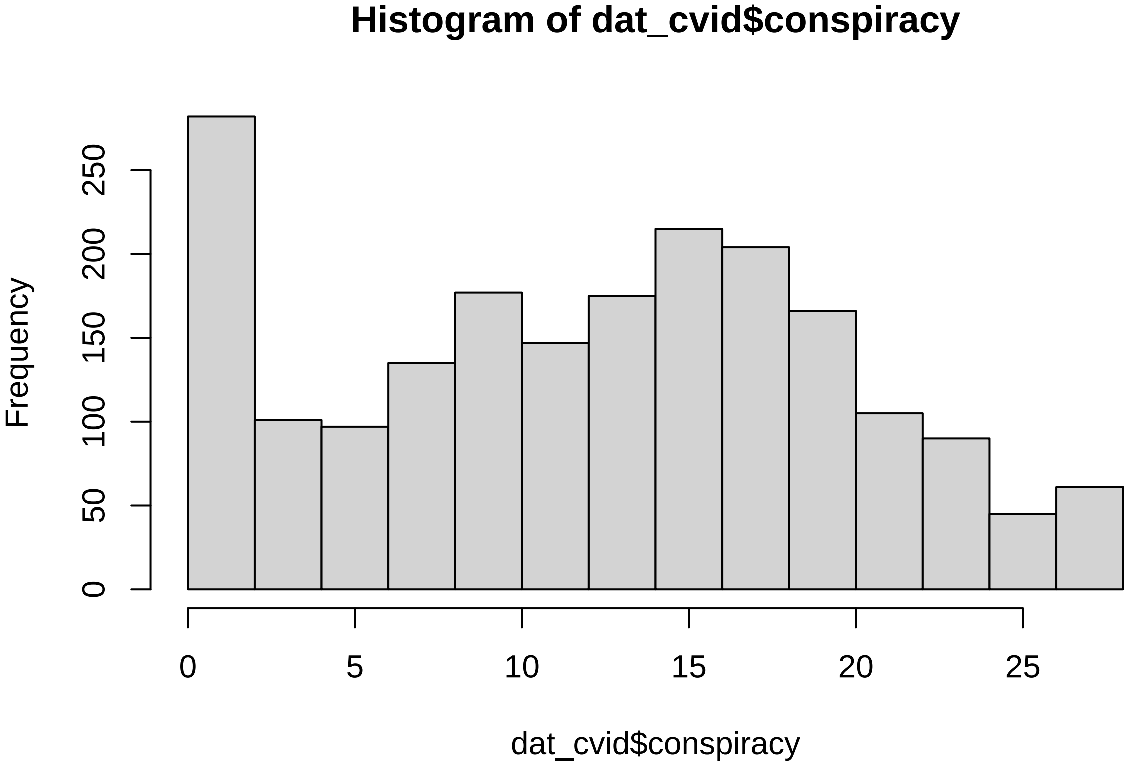
SOLUTION

Exercise 1: Corona stress data

We work again with the corona stress data set (`Covid_stress_data_subs.rda`). Check the previous exercise for a description of the individual variables in the data.

a) Load the data into R using the `load` function. This time, we want to predict the value of the `conspiracy` variable which expresses the tendency of the participants towards conspiracy thinking. Take a look at the distribution of this variable by creating a histogram (**Hint**: `hist()`).

```r
load('Covid_stress_data_subs.rda')
hist(dat_cvid$conspiracy, breaks = 10)
```



Our data includes different degrees of conspiracy thinking. We can see that there is a large group of observations with a value of zero.

b) We want to predict the variable `conspiracy` using a neural network. Since neural networks usually converge much better with standardized predictors, all (numeric) predictors in our data should be standardized (**Hint**: `scale()`). Therefore, we want to create a data frame in which all numeric predictors are standardized but the target variable `conspiracy` and all categorical variables are still in their original form. There are many different ways to perform such a transformation of data in R. You can either try to find a solution yourself or use the code presented in the solution sheet (solving the problem with a for-loop).

```r
### Copy data:
d <- dat_cvid
### Scale only numeric variables (many different ways to do this):
for(i in 1:ncol(d)){
  if(is.numeric(d[,i])){
    d[,i] <- scale(d[,i])
  }
}
### Bring back unscaled response variable:
d$conspiracy <- dat_cvid$conspiracy
```

Note: The response variable could also be standardized but normally this is not necessary. If the response variable is standardized then the predictions of the model will be on the standardized scale, too!

c) Fit a single hidden layer neural network to the (scaled) corona data with only one neuron in the hidden layer and no weight decay. Throughout this exercise, we use `maxit=10000` to make sure the NN can converge (**Hint**: `nnet(..., decay=0, maxit=10000)`). What is the meaning of the `linout` option in the `nnet` function and how do we have to set it in this case?

```r
library("nnet")
nn_cvid1 <- nnet(conspiracy ~ ., data=d, size=1, decay=0, linout=TRUE, maxit=10000)
```

```
## # weights:  29
## initial  value 463255.281067
## iter  10 value 126377.701517
## iter  20 value 117895.755961
## iter  30 value 117848.299721
## final  value 117847.980368
## converged
```

The linout option defines whether a linear activation function is used for the output layer. Since the response variable conspiracy is numeric, we have to set it to TRUE. Otherwise only values in the range of 0-1 would be predicted due to the default logistic activation function.

d) How many input neurons does our neural network have? Can you think of a reason why this number is not equal to the number of predictor variables?

```r
nn_cvid1
```

```
## a 26-1-1 network with 29 weights
## inputs: age educationUpto6years educationUpto9years educationUpto12years educationUniversity_without
## output(s): conspiracy
## options were - linear output units
```

The neural network has 26 input neurons. This number is larger than our number of predictors (16) because the categorical predictors are represented in the network through multiple dummy variables.

e) Using only one neuron in the hidden layer might be a too simple structure for the corona data. We now want to let the R package `caret` tune a neural network for us. First, one needs to define a search grid with the possible parameter values. We only want to tune the size of the network and not use any weight decay. Generate a small search grid with the possible sizes 1, 2, 3, 4 and 5. (**Hint**: `expand.grid(..., decay=0)`)

```
t_grid <- expand.grid(size=1:5, decay=0)
t_grid
```

```
##   size decay
## 1    1     0
## 2    2     0
## 3    3     0
## 4    4     0
## 5    5     0
```

f) Use the `train` function of `caret` to train a single hidden layer neural network along our search grid (this will take approx. 3 minutes). Note: Since `caret` will do the standardization of the variables for us, we can use the original data set (`dat_cvid`). What structure was finally selected? What was the RMSE of this model? Fix the random seed prior to running the code (`set.seed()`). Note: Even when using the same seed as in the solution sheet (`set.seed(2332)`), it can happen that your results differ from ours. There are various possible reasons for this, such as for example having a different R-version installed.

```
set.seed(2332)
library("caret")
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

```
tune_caret <- train(conspiracy ~ ., data=dat_cvid,
                    method='nnet', tuneGrid=t_grid, maxit=10000, trace=FALSE,
                    linout=TRUE, preProcess=c("center","scale"))
# trace=FALSE suppresses progress message
tune_caret
```

```
## Neural Network
##
## 2000 samples
##   16 predictor
##
## Pre-processing: centered (26), scaled (26)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2000, 2000, 2000, 2000, 2000, 2000, ...
## Resampling results across tuning parameters:
##
##   size  RMSE      Rsquared   MAE
##   1     5.544309  0.4721302  4.399577
##   2     5.973417  0.5024934  4.210346
##   3     5.152090  0.5552054  3.995546
##   4     4.997080  0.5851607  3.833391
##   5     5.105029  0.5703178  3.900728
##
## Tuning parameter 'decay' was held constant at a value of 0
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were size = 4 and decay = 0.
```

As can be seen the neural network with size 4 performed best with an RMSE of 4.997 and was, therefore, selected.

Note: The displayed warning about 'missing values in resampled performance measures' is not further problematic in this case. It signals that in some evaluation steps for the most simple NN (size=1) certain performance measures (in this case the R-squared value) could not be computed. The reason is that the neural network did possibly not have a good fit and predicted the same value for all cases of the resampling step. For predictions with no variance the R-squared value cannot be computed which triggers this warning.

Exercise 2: MNIST data (handwritten digits)

The MNIST data set contains a collection of 28x28 pixel images of handwritten digits (grayscale). It is a commonly used data set to train and test machine learning methods for visual processing.

a) As described in the lecture, grayscale pictures are often represented by tables with three axes. In R, tables of higher dimensions are represented by multi-way arrays. We can create a multi-way array with the `array` function, which requires a vector of numbers to fill the array (`data` argument) and a second vector indicating the dimensions of the array (`dim` argument). Create a three-dimensional array including the numbers 1 to 24 with the dimensions c(2,4,3). Look at the created array in R.

```r
x <- array(data = 1:24, dim = c(2,4,3))
x
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22   24
```

```r
dim(x)
```

```
## [1] 2 4 3
```

We can see that the array has three dimensions, R plots the first two dimensions separately along the third dimension. These are like three ‘slices’ of a cube.

b) We can access individual elements in multi-way arrays like for normal tables using the square brackets (`[ ]`). Display only the element of the second row, of the third column of the first slice. What happens if you select one entire slice, e.g. the second one?

```r
x[2,3,1]
```

```
## [1] 6
```

```r
### Select entire slice:
x[,,2]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
```

When we select the entire second slice we see that the output is the corresponding 2-dimensional table.

c) Read in the MNIST data containing the 28x28 pixel images. The data is stored in the `Mnist_training_and_test_data.rda` file. Once loaded, there should be four new objects in your environment: `test_x.0`, `test_y.0`, `train_x.0` and `train_y.0`. These objects store the pictures and the corresponding labels of the training and test set. Look at the dimensions of the arrays. How many pictures are in the training and test set? Also look at the target variable of the data (`test_y.0` and `train_y.0`).

```r
load('Mnist_training_and_test_data.rda')
dim(train_x.0)
```

```
## [1]    28    28 60000
```

```r
dim(train_y.0)
```

```
## [1] 60000
```

```r
dim(test_x.0)
```

```
## [1]    28    28 10000
```

```r
dim(test_y.0)
```

```
## [1] 10000
```

```r
### Look at labels (only done for test set):
test_y.0[1:20]
```

```
##  [1] 7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4
```

Each picture has 28x28 pixels. We can see that the arrays are organized so that the third dimension indicates the different pictures, while the first and second dimensions indicate the pixel positions. There are 60000 pictures in the training set and 10000 pictures in the test set. The test_y.0 and train_y.0 objects simply list the numbers shown on the individual pictures.

d) In the second image of the training data, what is the grayscale value of the pixel in the top right corner of the image?

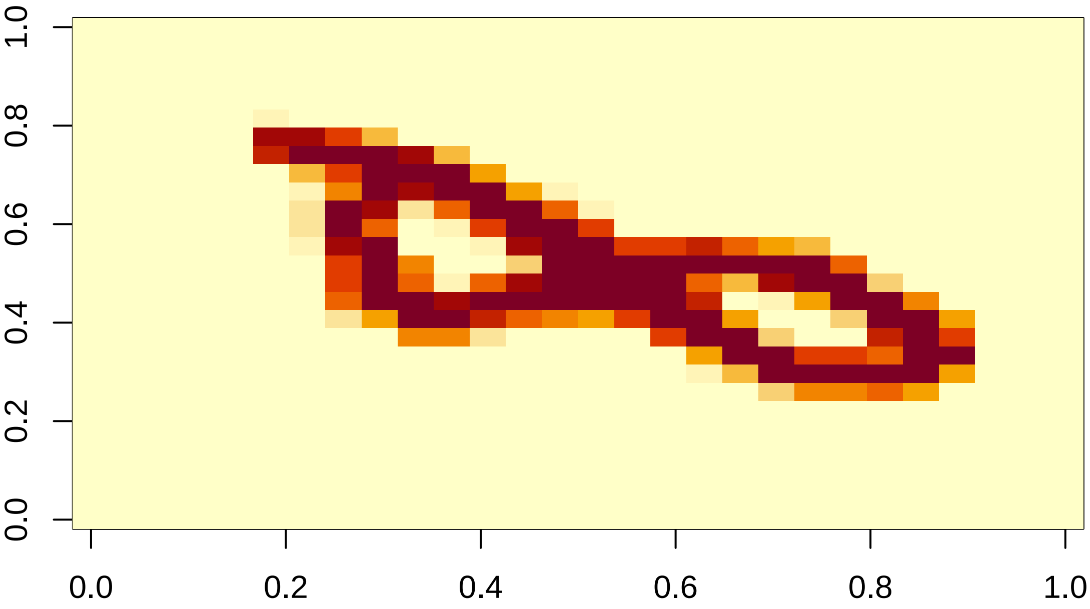
```
train_x.0[1, 28, 2]
```

```
## [1] 0
```

The pixel in the top row on the far right has a value of 0. In the MINST data 0 corresponds to white while 255 corresponds to black pixels. So this is a white pixel.

e) Each slice (3rd dimension) in the `train_x.0` array corresponds to one picture. We can plot one picture by selecting only one slice in the array and feed it into the `image` function. Select the 18th picture of `train_x.0` and visualize it with `image()`.

```
pc18 <- train_x.0[,,18]
image(pc18)
```



As we can see the 18th image is a handwritten 8. The image() function in its default setting plots the image 90-degree rotated and adds colour to the grayscale image.

f) To directly feed the MNIST pictures to a standard neural network, we need to "flatten" the images, so that each image is one row and the columns represent all the pixels of a picture. Run the code below to flatten the images of the training and test set. The code additionally scales the grayscale values to a range of 0-1 by dividing the values by 255 (better for NN). The target variable is turned into a factor since predicting the written number on a picture is a classification task. Finally, the pixel values are combined with the target variable into data frames. Look at the final data frames.

```
### Turn into 2d matrices (each row one picture):
train_x <- array(as.numeric(aperm(train_x.0, perm = c(3, 1, 2))), dim = c(60000, 784))
test_x <- array(as.numeric(aperm(test_x.0, perm = c(3, 1, 2))), dim = c(10000, 784))
```

```
### Scale to 0-1 range:
train_x <- train_x/255
test_x <- test_x/255
### Turn number labels to factor:
train_y <- as.factor(train_y.0)
test_y <- as.factor(test_y.0)
### Combine to dataframe:
mnist.dtra <- data.frame("y"=train_y, train_x)
mnist.dte <- data.frame("y"=test_y, test_x)
### Look at dimensions:
dim(mnist.dtra)
```

```
## [1] 60000   785
```

```
dim(mnist.dte)
```

```
## [1] 10000   785
```

We see that the training data is now a normal data frame with the dimensions 60000 x 785. This corresponds to 60000 pictures, each with 28 x 28 = 784 pixels. Each column stores the value of one pixel and the first column stores the target variable (1 + 784 = 785 columns).

g) Now the data is ready to be fed to a classifier, e.g. a neural network. We want to train a single hidden layer NN on the training data and check its performance on the test data. Since training a neural network with `nnet` of `size=20` on the MNIST data takes approximately 20 hours, you can use the already fitted NN stored in the `NN_mnist.rda` file. The file contains a fitted NN (`nne.mnist` object). It was created with the command:

```
nne.mnist <- nnet(y~., mnist.dtra, size=20, decay=0, maxit=10000, MaxNWts = 16000)
```

Predict the written numbers in the test set with the neural network and create a confusion matrix showing how well it performed (**Hint**: `predict(..., type='class')`). Calculate the accuracy of the predictions (accuracy = 1 - missclassification rate).

```
load('NN_mnist.rda')
### Confusion matrix:
conf.nn <- table(test_y, predict(nne.mnist, newdata = mnist.dte, type='class'))
conf.nn
```

```
##
## test_y    0    1    2    3    4    5    6    7    8    9
##      0  936    1    6    2    4    6   11    1    6    7
##      1    3 1099    5    2    2    2    4    3   13    2
##      2    8    2  927   22   16    9    7   17   21    3
##      3    5    8   21  894    3   39    3   16   12    9
##      4    2    4    2    6  908    6   14   12    5   23
##      5    5   18    5   35   11  767   15    5    8   23
##      6    9    5   11    0   10   23  888    2    9    1
##      7    3    6   22   17   10    3    1  942    2   22
##      8   12    1   14   25    8   21   18    7  859    9
##      9    4    5    2   11   44   14    1   18    5  905
```

```
### Accuracy (one minus missclassification rate):
cc <- conf.nn
diag(cc) <- 0
1 - sum(cc)/sum(conf.nn)
```

```
## [1] 0.9125
```

The neural network perfoms okay but not too great, reaching an accuracy of 0.912. For a better performance the network would have to be tuned properly. Normally, for picture recognition a convolutional neural network is used, which can easily reach an accuracy of 99.7% on the MNIST data set.

h) Compare how a Random Forest performs in the MNIST classification task. Fitting a Random Forest with `cforest` to the MNIST training data and generating predictions for the test data also takes a long time (approx. 3 hours). Therefore, we have already fitted a Random Forest (consisting of 500 trees) to the training data and used it to create predictions for the test data. The predictions have been created with the following commands:

```r
library("partykit")
set.seed(111)
### Create random forest:
rf_mnist <- cforest(y~., mnist.dtra, ntree=500)
### Generate predictions for test data:
prds_rf <- predict(rf_minst, newdata = mnist.dte)
```

The created predictions are stored in the `RF500_mnistPreds.rda` file. Load the predictions in R and investigate the predictive accuracy of the Random Forest.

```r
load('RF500_mnistPreds.rda')
### Confusion matrix:
conf.rf <- table(mnist.dte$y, prds_rf)
conf.rf
```

```
##    prds_rf
##        0    1    2    3    4    5    6    7    8    9
##   0  970    0    0    0    0    4    2    1    3    0
##   1    0 1119    3    3    1    1    4    1    3    0
##   2    8    1  978    6    9    1    8   13    7    1
##   3    2    0   16  950    0   14    1   12   11    4
##   4    1    1    2    0  927    0    7    0    5   39
##   5    8    4    1   15    5  838   10    2    6    3
##   6   14    3    0    0    4    3  930    0    4    0
##   7    2   10   26    1    3    0    0  957    6   23
##   8    5    1    7    4    6    4   12    5  916   14
##   9    7    7    2   14   17    4    2    4    8  944
```

```r
### Accuracy:
cc <- conf.rf
diag(cc) <- 0
1 - sum(cc)/sum(conf.rf)
```

```
## [1] 0.9529
```

As we can see the Random Forest performs quite well on the MNIST data, reaching an accuracy of 0.953.