Individual Assignment 5a – Neural Networks

Lucas Hagelstein (559020)

04/12/2020

Dr. Michel van de Velden

Dr. Anastasija Tetereva

Dr. Carlo Cavicchia

Seminar in Data Science for Marketing Analytics

MSc in Data Science and Marketing Analytics

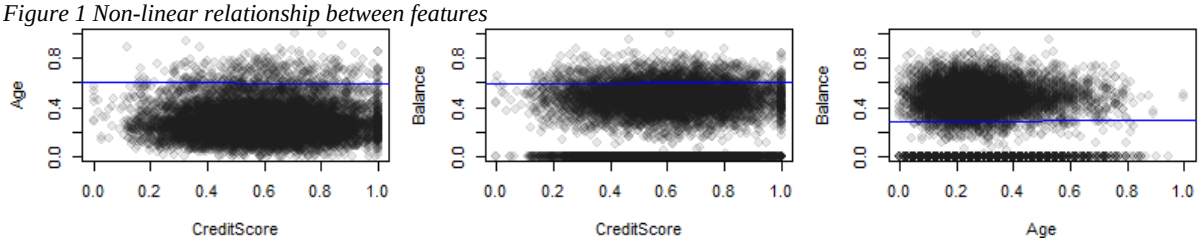
Erasmus School of Economic

1. Introduction

The customer is a national bank that experienced a high rate of churned customers in the last periods. Therefore, they want to predict which customers are likely to churn given eight independent features which describe customer characteristics (e.g. age, credit score, estimated salary, tenure, etc...). Blattberg and Deighton (1996) conclude the costs of attracting new customers to be higher compared to the retention costs. Hence, the goal of the analysis is to determine which customers are likely to churn and subsequently reach out to them by using suitable marketing activities.

A neural network is a suitable research method considering the following: First, numeric features of the dataset are non-linearly related to each other. Figure 1 visualizes the relationships between "CreditScore", "Age" and "Balance". Second, relationships are not only detected in the training data but also in the unseen test data. Third, only "Age" and "CreditScore" are normally distributed whereas all other variables are either binary or uniformly distributed. Hence, linear models may perform poorly and therefore a neural network will be built.

Figure 1 Non-linear relationship between features



2. Method

Multi-layer feed-forward neural networks consist of an input layer, at least one hidden layer and an output layer. Each layer has a specified number of nodes n. In the input layer n is equal to the number of features p. The number of nodes n in hidden layers as well as the number of hidden layers h can be chosen arbitrarily. Svozil et al. (1997), advise to choose $h = 1$ with varying n. For each node i in the hidden layer the mapping function ξ_i assigns the preceded ancestors. The relationship between two nodes i and j is described by the weight coefficient ω_{ij} (reflects the connection between nodes i and j, whereas j is the prior ancestor). Each node i is further described by the threshold coefficient υ_i (also known as bias) and the outcome of node i is described as y_i. Formally, it is written $y_i = f(\varphi_i)$, whereas

$$\varphi_i = \upsilon_i + \sum_{j \epsilon \xi_i^{-1}} \omega_{ij} y_j.$$

The expression $j \in \xi_i^{-1}$ includes all prior ancestors. Hence, the product of the weighted coefficient of nodes i and j and the prior outcome of node j is summed up for all prior ancestors and added to the threshold coefficient (bias) υ_i. Intuitively, φ_i is understood as potential that is lead forward (forward phase) through the neuronal network by the activation function

$$f(\varphi) = \frac{1}{1 + \exp(-\varphi)},$$

where the potential φ_i is non-linearly transformed by the sigmoid function. The optimizer is the binary cross-entropy loss function that is minimized in the backward phase (backpropagation). This can be written as

$$E = \sum_o \hat{y}_o * [-\log(y_o)] + (1 - \hat{y}_o) * [-\log(1 - y_o)] + \lambda,$$

whereas λ is a constant penalty term and y_o and $\hat{y}_o$ are vectors of the predicted and the actual outcomes of the output layer nodes, respectively. As each outcome y_i depends on the bias ϑ_i and each weight coefficient ω_{ij}, the optimal coefficients are obtained by calculating the partial derivatives $\frac{\partial E}{\partial \omega_{ij}}$ and $\frac{\partial E}{\partial \vartheta_i}$ to minimize the loss function. This process is also known as gradient descent. The hidden layer can therefore be understood as translator from the input to the output layer and vice versa. The penalty term $\lambda \geq 0$, also called weight decay, shrinks the weight coefficients as well as the threshold

coefficients. The tuneable hyperparameters, in the context of this report, are given by the number of nodes n in the hidden layer and the weight decay λ.

Opening the black-box of a neural network is possible using the variable importance that is deconstructing model weights and identifying weighted connections between the nodes of interest (Zhang et al., 2018).

3. Data Preparation and Results

First, numeric features have been normalized and take values between 0-1 to improve the training efficiency. Additionally, this avoids large features to be dominant in the gradient descent. Second, a common problem when evaluating classification results is the accuracy paradox. In other words, imbalanced classes will bias the model towards the majority class as only little data is available for the minority class. Especially if the data is split randomly. Additionally, the accuracy is misguiding as the TPs (unimportant majority class) possibly average out the FNs (minority class). Given these two reasons, a partitioning split (Stratify), oversampling and a combination of over-/ undersampling (ROSE) is used to improve the performance on the minority class. For comparability of the methods, the over- /undersampling techniques are applied on the training data which is obtained after partitioning the original dataset regarding the underlying balance of 80/20 ("Not_Exited"/"Exited") between the classes. Finally, the data consist of 70% training and 30% testing.

Models are built, tuned and evaluated with the Caret-Package in R. The model is considered as "nnet"-classifier. Each model is tuned by performing 10 iterations of 10-fold cross-validated grid-search on the hyperparameters $n \in \{7, 8, 9, 10, 12, 14, 16, 20\}$ and $\lambda \in \{0, 0.0001, 0.0001, 0.0003, 0.001, 0.01, 0.1\}$.

Cohen's Kappa is used for model evaluation as it is robust to imbalanced data. Hence, the final model with 20 nodes and a weight decay of 0.0001 yields 83% accuracy and Cohen's Kappa of 0.47 which is a fair agreement. Furthermore, by applying resampling methods instead of a simple "partitioning split" the Cohen's Kappa increases by 0.03 at the costs of 2% in accuracy (see Table 1).

Figure 2 Variable Importance: Garson Algorithm

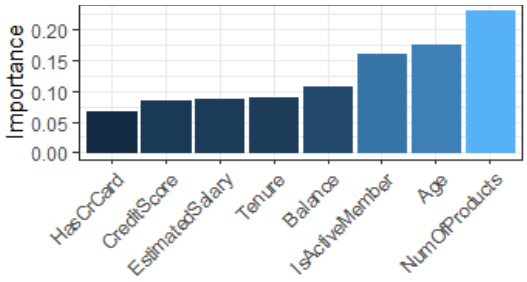


Figure 2 shows which variables most influence whether a customer will remain. Recall, "Not_Exited" is determined as positive value. Number of products, age, and if a customer is active or not are considered as important variables. However, nothing can be said about the direction of the variable influence.

Table 1 Confusion Matrix of each Model

Sampling Method	Partitioning (80/20) (Stratify)		Oversampling of minority class		ROSE (Over-/Undersampling)	
Optimal Parameters	$n^* = 12, \lambda^* = 0.1$		$n^* = 20, \lambda^* = 0.001$		$n^* = 20, \lambda^* = 0.0001$	
	Exited	N_Exited	Exited	N_Exited	Exited	N_Exited
Exited (Negative)	250	85	410	463	345	259
N_Exited (Positive)	361	2303	201	1925	266	2129
Accuracy	85%		78%		83%	
Cohen's Kappa	0.44		0.41		0.47	

4. Discussion

The best model was chosen by the highest kappa-statistic but in real-life applications the choice depends on the associated costs and revenues of the client. As the acquisition costs of new customers may be higher than the retention costs the client might prefer a model that has neither the highest accuracy nor the highest kappa-statistic. The preferred solution is obtained by assigning costs to FPs (acquisition costs), FNs (inadvertent retention costs) and TNs (justified retention costs) and the subsequent trade-off between costs and revenues. Finally, the client can reach out to customers who are predicted to churn. Depending on the resulting cost-revenue trade-off the client chooses a model and decides which customers he wants to reach through marketing activities. Next, based on the variable importance the client may elaborate the most influenceable variables and decide which marketing activities to choose.

5. References

Blattberg, R. C., & Deighton, J. (1996). Manage marketing by the customer equity test. Harvard business review, 74(4), 136.

Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. Chemometrics and intelligent laboratory systems, 39(1), 43-62.

Zhang, Z., Beck, M. W., Winkler, D. A., Huang, B., Sibanda, W., & Goyal, H. (2018). Opening the black box of neural networks: methods for interpreting neural network models in clinical applications. Annals of translational medicine, 6(11).

Appendix: Code

```
library(nnet)
library(readr)
library(caret)
library(ROSE)
library(tidyverse)
library(DMwR)
#############################################################################
#### 1. Data Preparation
## Read Dataframe
Churn <- read_csv("Churn.csv")
## Inspect Data
summary(Churn)
#> Booleans: HasCrCard, IsActiveMember, Exited
## Read Dataframe and change Datatypes
Churn <- read_csv("Churn.csv",
                  col_types = cols(Exited = col_logical(),
                                   HasCrCard = col_logical(),
                                   IsActiveMember = col_logical()))
## Convert columns to integers
Churn[, c(2:4,6)] <- lapply(Churn[,c(2:4,6)], as.integer)
## Convert Dependent Variable to a factor
Churn$Exited <- as.factor(ifelse(Churn$Exited == TRUE, "Exited","Not_Exited"))
Churn$HasCrCard <- as.factor(ifelse(Churn$HasCrCard == TRUE, 1,0))
Churn$IsActiveMember <- as.factor(ifelse(Churn$IsActiveMember == TRUE, 1,0))
## Drop first column
Churn <- Churn[,-1]
#############################################################################
#### 2. Preprocessing
summary(Churn$Exited)
#Min-Max Scaler
normalize <- function(x){
  return((x- min(x)) /(max(x)-min(x)))}
## Normalize numeric values
Churn[,c(1:5,8)] <- sapply(Churn[,c(1:5,8)],normalize)
colnames(Churn)
## Correlationplot
par(mfrow=c(1,3), mar=c(4,4,1,1))
plot(Churn$CreditScore, Churn$Age, xlab="CreditScore", ylab="Age", col = rgb(red =
0.1, green = 0.1, blue = 0.1, alpha = 0.1), pch=20, cex=2)
abline(lm(Churn$CreditScore ~ Churn$Age), col="blue")
plot(Churn$CreditScore, Churn$Balance,xlab="CreditScore", ylab="Balance",col = rgb
(red = 0.1, green = 0.1, blue = 0.1, alpha = 0.1), pch=20, cex=2)
abline(lm(Churn$CreditScore ~ Churn$Balance), col="blue")
plot(Churn$Age, Churn$Balance,xlab="Age", ylab="Balance",col = rgb(red = 0.1, gree
n = 0.1, blue = 0.1, alpha = 0.1), pch=20, cex=2)
abline(lm(Churn$Age ~ Churn$Balance), col="blue")
## Sort the Dataframe and create formula
Churn_sorted <- data.frame(Churn[,9],Churn[,1:8])
formula <-as.formula('Exited~CreditScore+Age+Tenure+Balance+NumOfProducts+HasCrCar
d+IsActiveMember+EstimatedSalary')
## Stratify Data --> train/test (80/20)
set.seed(1)
train.index <- createDataPartition(Churn_sorted[,1], p = .7, list = FALSE)
train_data <- Churn_sorted[train.index,]
test_data  <- Churn_sorted[-train.index,]
colnames(train_data) <- c("Exited", "CreditScore", "Age", "Tenure", "Balance", "Nu
mOfProducts", "HasCrCard", "IsActiveMember", "EstimatedSalary")
#Check the split 70% train data (80% NO, 20% YES), 30% test data (80% NO, 20% YES)
table(train_data[,1])
#############################################################################
```

```
#### 3. Analysis
# part a: set range of tuning parameters (layer size and weight decay)
tune_grid_neural <- expand.grid(size = c(7,8,9,10,12,14,16,20),
                                decay = c(0, 0.00001, 0.0001, 0.0003, 0.001,0.01,0.1))
# part b: constraint calculation
max_size_neural <- max(tune_grid_neural$size)
max_weights_neural <- max_size_neural*(nrow(train_data) + 1) + max_size_neural + 1
# ------------------------------------------------------------------------------
# STEP 2: SELECT TUNING METHOD
# set up train control object, which specifies training/testing technique
train_control_neural <- trainControl(method = "LGOCV",
                                     number = 10,
                                     classProbs = TRUE,
                                     verboseIter = TRUE)
# ------------------------------------------------------------------------------
# STEP 0: set seed, so that statistics don't keep changing for every analysis
set.seed(1)
# ------------------------------------------------------------------------------
# STEP 1: decide how many times to run the model
rounds <- 10
# ------------------------------------------------------------------------------
# STEP 2: set up object to store results
# part a: create names of results to store
result_cols <- c("model_type", "round", "accuracy", "kappa", "accuracy_LL", "accur
acy_UL",
                 "sensitivity", "specificity", "precision", "npv","F1", "n")
# part b: create matrix
results <-
  matrix(nrow = rounds,
         ncol = length(result_cols))
# part c: actually name columns in results marix
colnames(results) <- result_cols
# part d: convert to df (so multiple variables of different types can be stored)
results <- data.frame(results)
# ------------------------------------------------------------------------------
# STEP 2: start timer
start_time <- Sys.time()
# ------------------------------------------------------------------------------
# STEP 3: create rounds number of models, and store results each time
for (i in 1:rounds){
  # part c: use caret "train" function to train logistic regression model
  model <- train(form = formula,
          data = train_data,
          method = "nnet",
          tuneGrid = tune_grid_neural,
          trControl = train_control_neural,
          metric = "Kappa", # how to select among models
          trace = FALSE,
          maxit = 100,
          MaxNWts = max_weights_neural)
  # part d: make predictions
  preds <- predict(object = model,
            newdata = test_data,
            type = "raw")
  # part e: store model performance
  conf_m <- confusionMatrix(data = as.factor(preds),
                    reference = test_data$Exited,
                    positive ="Exited")
  # part f: store model results
  # model type
  results[i, 1] <- "neural"
```

```
  # round
  results[i, 2] <- i
  # accuracy
  results[i, 3] <- conf_m$overall[1]
  # Kappa
  results[i, 4] <- conf_m$overall[2]
  # accuracy LL
  results[i, 5] <- conf_m$overall[3]
  # accuracy UL
  results[i, 6] <- conf_m$overall[4]
  # sensitivity
  results[i, 7] <- conf_m$byClass[1]
  # specificity
  results[i, 8] <- conf_m$byClass[2]
  # precision
  results[i, 9] <- conf_m$byClass[3]
  # negative predictive value
  results[i, 10] <- conf_m$byClass[4]
  # F1 Score
  results[i, 11] <- conf_m$byClass[7]
  # sample size (of test set)
  results[i, 12] <- sum(conf_m$table)
  # part g: print round and total elapsed time so far
  cumul_time <- difftime(Sys.time(), start_time, units = "mins")
  print(paste("round #", i, ": cumulative time ", round(cumul_time, 2), " mins",
              sep = ""))
  print("---------------------------------------")
}
#> Imbalanced Classification problem
##############################################################################################
## Oversampling
#> Keep majority class equal, oversample (4x) minority class
# ------------------------------------------------------------------------------
# STEP 0: set seed, so that statistics don't keep changing for every analysis
set.seed(1)
data_balanced_over <- ovun.sample(formula, data = train_data, method = "over",N =
11000)$data
table(data_balanced_over$Exited)
# ------------------------------------------------------------------------------
# STEP 1: decide how many times to run the model
rounds <- 10
# ------------------------------------------------------------------------------
# STEP 2: set up object to store results
# part a: create names of results to store
result_cols <- c("model_type", "round", "accuracy", "kappa", "accuracy_LL", "accur
acy_UL",
                 "sensitivity", "specificity", "precision", "npv","F1", "n")
# part b: create matrix
results_over <- matrix(nrow = rounds,
         ncol = length(result_cols))
# part c: actually name columns in results marix
colnames(results_over) <- result_cols
# part d: convert to df (so multiple variables of different types can be stored)
results_over <- data.frame(results_over)
# ------------------------------------------------------------------------------
# STEP 2: start timer
start_time <- Sys.time()
for (i in 1:rounds){
  # part c: use caret "train" function to train logistic regression model
  model_over <- train(form = formula,
          data = data_balanced_over,
```

```
          method = "nnet",
          tuneGrid = tune_grid_neural,
          trControl = train_control_neural,
          metric = "Kappa", # how to select among models
          trace = FALSE,
          maxit = 100,
          MaxNWts = max_weights_neural)
  # part d: make predictions
  preds_over <- predict(object = model_over,
            newdata = test_data,
            type = "raw")
  # part e: store model performance
  conf_m_over <- confusionMatrix(data = as.factor(preds_over),
                    reference = test_data$Exited,
                    positive = "Exited")
  # part f: store model results
  # model type
  results_over[i, 1] <- "neural"
  # round
  results_over[i, 2] <- i
  # accuracy
  results_over[i, 3] <- conf_m_over$overall[1]
  # Kappa
  results_over[i, 4] <- conf_m_over$overall[2]
  # accuracy LL
  results_over[i, 5] <- conf_m_over$overall[3]
  # accuracy UL
  results_over[i, 6] <- conf_m_over$overall[4]
  # sensitivity
  results_over[i, 7] <- conf_m_over$byClass[1]
  # specificity
  results_over[i, 8] <- conf_m_over$byClass[2]
  # precision
  results_over[i, 9] <- conf_m_over$byClass[3]
  # negative predictive value
  results_over[i, 10] <- conf_m_over$byClass[4]
  # F1 Score
  results_over[i, 11] <- conf_m_over$byClass[7]
  # sample size (of test set)
  results_over[i, 12] <- sum(conf_m_over$table)
  # part g: print round and total elapsed time so far
  cumul_time <- difftime(Sys.time(), start_time, units = "mins")
  print(paste("round #", i, ": cumulative time ", round(cumul_time, 2), " mins",
              sep = ""))
  print("---------------------------------------")
}
#####################################################################
## ROSE
#> Undersample majority class, oversample (2x) minority class
# ---------------------------------------------------------------------------
# STEP 0: set seed, so that statistics don't keep changing for every analysis
set.seed(1)
data_rose <- ROSE(formula, data = train_data, seed = 1)$data
table(data_rose$Exited)
# ---------------------------------------------------------------------------
# STEP 1: decide how many times to run the model
rounds <- 10
# ---------------------------------------------------------------------------
# STEP 2: set up object to store results
# part a: create names of results to store
result_cols <- c("model_type", "round", "accuracy", "kappa", "accuracy_LL", "accur
```

```
acy_UL",
                  "sensitivity", "specificity", "precision", "npv","F1", "n")
# part b: create matrix
results_rose <-
  matrix(nrow = rounds,
         ncol = length(result_cols))
# part c: actually name columns in results marix
colnames(results_rose) <- result_cols
# part d: convert to df (so multiple variables of different types can be stored)
results_rose <- data.frame(results_rose)
# ------------------------------------------------------------------------------
# STEP 2: start timer
set.seed(1)
start_time <- Sys.time()
for (i in 1:rounds){
  # part c: use caret "train" function to train logistic regression model
  model_rose <-
    train(form = formula,
          data = data_rose,
          method = "nnet",
          tuneGrid = tune_grid_neural,
          trControl = train_control_neural,
          metric = "Kappa", # how to select among models
          trace = FALSE,
          maxit = 100,
          MaxNWts = max_weights_neural)
  # part d: make predictions
  preds_rose <- predict(object = model_rose,
                        newdata = test_data,
                        type = "raw")
  # part e: store model performance
  conf_m_rose <- confusionMatrix(data = as.factor(preds_rose),
                                 reference = test_data$Exited,
                                 positive = "Exited")
  # part f: store model results
  # model type
  results_rose[i, 1] <- "neural"
  # round
  results_rose[i, 2] <- i
  # accuracy
  results_rose[i, 3] <- conf_m_rose$overall[1]
  # Kappa
  results_rose[i, 4] <- conf_m_rose$overall[2]
  # accuracy LL
  results_rose[i, 5] <- conf_m_rose$overall[3]
  # accuracy UL
  results_rose[i, 6] <- conf_m_rose$overall[4]
  # sensitivity
  results_rose[i, 7] <- conf_m_rose$byClass[1]
  # specificity
  results_rose[i, 8] <- conf_m_rose$byClass[2]
  # precision
  results_rose[i, 9] <- conf_m_rose$byClass[3]
  # negative predictive value
  results_rose[i, 10] <- conf_m_rose$byClass[4]
  # F1 Score
  results_rose[i, 11] <- conf_m_rose$byClass[7]
  # sample size (of test set)
  results_rose[i, 12] <- sum(conf_m_rose$table)
  # part g: print round and total elapsed time so far
  cumul_time <- difftime(Sys.time(), start_time, units = "mins")
```

```
  print(paste("round #", i, ": cumulative time ", round(cumul_time, 2), " mins",
              sep = ""))
  print("-------------------------------------")
}
model$bestTune
#> size = 12, decay = 0.1
model_over$bestTune
#> size = 20, decay = 0.001
model_rose$bestTune
#> size = 20, decay = 0.0001
##############################################################################
#### Final Model
# part a: set range of tuning parameters (layer size and weight decay)
grid_final <- expand.grid(size = c(model_rose$bestTune[1,1]),
                          decay = c(model_rose$bestTune[1,2]))
set.seed(1)
# ----------------------------------------------------------------------------
# STEP 2: SELECT TUNING METHOD
# set up train control object, which specifies training/testing technique
train_control_neural <- trainControl(method = "LGOCV",
                                     number = 50,
                                     classProbs = TRUE,
                                     verboseIter = TRUE)
fit_final <- train(formula,
               data = data_rose, method = 'nnet',
               trControl = train_control_neural,
               tuneGrid= grid_final,
               metric = "Kappa",
               trace = FALSE,
               maxit = 100,
               MaxNWts = max_weights_neural)
results_test <- predict(fit_final, newdata=test_data)
conf_test <- confusionMatrix(results_test, test_data$Exited)
##############################################################################
## Plot Variable Importance
library(devtools)
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505
ff044412703516c34f1a4684a5/nnet_plot_update.r')
gar <- garson(fit_final) +
       theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust=1))
##############################################################################
## Plot Neural Network
plotnet(fit_final)
```