Assignment Part 2

Group1

2023-01-05

Contents

Abstract

Summary of Methods and Metrics

To transform the initial dataset, we've used the library tidymodels. With tidymodels, it is possible to create a recipe, which can then be "baked", in order to receive a prepared, normalized and one-hot-encoded dataset that can be forwarded into a neural network. Additionally, tidymodels is able to oversample the minority class, which is needed in case the class attribute is imbalanced. To build the neural network, the library Tensorflow together with Keras have been used.

As an optimizer, we started with rmsprop, replacing it with adam to finally use SGD (stochastic gradiant descent). Although SGD is able to use a momentum, results were best without. The loss function needs to be categorical_crossentropy and the activation function of the last layer needs to be softmax for multiclass, single-lable classification tasks. The metric for the neural network is categorical_accuracy, a metric that calculates, how often the predictions match the one-hot encoded class attribute.

Overview of important steps

1. As in every Data Science project, knowledge about data is key. Therefore, the first step included to gain it.
2. Clean the data of unwanted, unusable rows and columns as ID or FLAG_MOBIL or rows with NAME_INCOME_TYPE = "Student"

- ID is a number assigned to each customer after registration. There is no use in using it as a predictor
- FLAG_MOBIL is always 1, no use as predictor
- There are only 4 rows with students and keeping them resulted in an error with the One-Hot-Encoding in tidymodels. Since the frequency was so low, we decided to drop the 4 rows, knowing that our model is not able to learn this predictor

3. Replace C and X in the class attribute with numerical values

- This was done to easily One-Hot-Encode the data with tidymodels

4. Replace N/A in OCCUPATION_TYPE with "Unknown"

- N/A will result in problems with tidymodels. therefore we changed them to "Unknown"
- This could possibly impact our model in a negative way as the model could learn wrongly that "Unknown"-rows belong together although it was just the absence of a value. It might be an idea to remove the rows with N/A or try to use kNN to assign another value

5. Randomly shuffle the dataset

- Training- and validation-accuracy seemed to be fine with > 90%. However, on evaluation with our test-data, the NN has not seen during training, the accuracy was still only around 70-80%. To prevent that some patterns only occur in test-data and therefore cannot be trained, the data-set was shuffled before the split

6. Split data into test/trainingset (80/20)
7. Remove outliers in training-set outside 1.5 of interquartile range
8. Preprocess the data

- We oversampled the class attribute status until all classes are equally distributed. This performed best in our tests.

9. Build a model

- We trained a lot of models, changing the amount of layers, and the size of the model
- We compared the results and tuned the hyperparameters accordingly

10. Train the model

- After setting up the k-fold validation, we trained the model with k = 2 and only 500 epochs to estimate if the current model with its hyperparameters is a good candidate or not. A selection of that training is listed in the table below

11. Evaluate the model

- This was done by comparing the mean of the training, validation and test-accuracy

12. Repeat step 8-10 until the accuracy is satisfying enough

Explorative steps

These are steps that have been tried out, but were discarded again due to bad performance. These steps include:

- Undersampling the majority class
  - As the class attribute is highly imbalanced, we tried to untersample. However, some status only had around 100 examples in the dataset. Therefore we removed the undersampling as it reduced our trainingset to ~1000 rows
- Undersampling with oversampling
  - As undersampling alone resulted in so few rows, we afterwards tried to oversample again to gain enough rows to train the models with. The performance was bad, which can be expected as undersampling removed around 55000 rows of the majority class
- Oversampling the minority classes until minority-classes reach 25% of the majority class
  - Oversampling the minority classes will add artificial rows. As we only had very few examples for some classes, we didn't want to add 10'000s of rows, so our first attempt was to only oversample up to 25%
- Using class weights
  - Using class weights as a test, until we received a satisfying accuracy with oversampling. As oversampling was sufficient, class weights have been removed again

Examples of hyperparameter tuning:

Layers	Split	Epochs	K	Learning Rate	Mean(Training)	Mean(Val)	Mean(Test)
128,1024,1024,128,8	80/20	1500	2	0,01	0.9499	0.9365972	0.7394069
128,2048,DO,2048,DO,128,8	80/20	1500	2	0,02	0.8081568	0.8220032	0.6698957
128,2048,2048,128,8	80/20	1500	2	0,02	0.8361189	0.8279235	0.7154478
128,512,512,128,8	80/20	1500	2	0,03	0.8611527	0.851758	0.7405901
128,512,512,128,8	80/20	1500	2	0,2	0.9526283	0.9347506	0.7614435
128,512,512,64,8	80/20	1500	2	0,2	0.9507743	0.9316875	0.7654366
128,512,512,32,8	80/20	1500	2	0,2	0.9506324	0.9322321	0.7704651
128,512,512,32,8	80/20	1500	2	0,1	0.9302815	0.914666	0.7717223
128,512,512,32,8	80/20	1500	2	0,25	0.9554314	0.93616	0.7595208
128,512,512,64,8	80/20	1500	2	0,25	0.9324356	0.9129541	0.7520521
128,512,512,128,8	80/20	1500	2	0,25	0.9503229	0.9308634	0.7576721
128,512,512,128,8	90/10	1500	2	0,25	0.9567535	0.9383946	0.7685596
128,512,128,8	90/10	1500	2	0,25	0.9450369	0.9285425	0.7698905

Poor results on Testset indicate overfitting.
To check if there is a problem with the Test and Training-Set, data was shuffled before the split into training and test and we continued with the best previous Hyperparameters:

Layers	Split	Epochs	K	Learning Rate	Mean(Training)	Mean(Val)	Mean(Test)
128,512,512,128,8	80/20	1500	10	0,2	0.9643427	0.9460294	0.7660849
128,512,512,128,8	80/20	1500	2	0,15	0.97559553	0.9261869	0.7554356
128,512,512,128,8	80/20	1500	2	0,175	0.9484972	0.9302641	0.7601686
128,1024,1024,256,32,8	80/20	1500	2	0,2	0.9575419	0.9378761	0.7637184
128,1024,1024,256,32,8	80/20	540	2	0,2	0.9854212	0.9646771	0.7728147
128,1024,1024,256,32,8	80/20	540	10	0,2	0.9865129	0.9681746	0.7777696
128,1024,1024,256,32,8	80/20	534	10	0,2	0.9866723	0.9688698	0.7941133

Main questions

Why did you use a feed-forward network, or a convolutional or recursive?

We used a feed-forward network for our solution. CNN are used for pattern recognition and are able to process a 2D array of data, e.g. pixel of an image. As we want to train a 1D vector (one row is one customer) instead of a 2D array, this type of network is wrong. Additionally, a convolutional layer uses 2D filters. If we would use this on our data, we would mix up data from different rows.

A recursive network has a memory and is able to remember other inputs while processing a certain row. This is of great use for NLP or with timeseries, but as the rows in the data for this assignment are independent from each other, a memory is not needed.

For simple predictive analysis, feed-forward networks are the best choice due to the simplicity and ability to process tabular data.

Is our model is reasonable good?

The training and validation-accuracy was high and sufficient with around 98% and 97%. However, using the model on unseen testdata only showed an accuracy of around 79%. This would mean that our model fails in every fifth prediction. It seems to be too low, but as we only estimate the payback behavior and not if the customer receives a credit at all, the model could still be good enough. Usually, a lower value on testdata as in training indicates overfitting. To rule it out, the dataset was shuffled and different splits, e.g. 80/20, 90/10 were tried without success. This could indicate that With available, heavily imbalanced data, it could be that this is the highest accuracy we could receive. As a possible next step, more training data should be gathered in order to further train the model, but with the current accuracy we would advise against using the model productively.

Lessons learned

- How to create a Neural Network in RStudio
- How to use RMarkdown
- How to use Git
- How to test in an ordered manner
- If training takes oddly long, check if it might run on your CPU instead of GPU

In general, the task was a good exercise to familiarise ourselves with neural networks. To solve the task, we had to do a lot of reading and understanding. The only disadvantage for us was the uncertainty whether the model was the best we could possibly create or whether there would be a better model with a few adjustments.

Data import

```
library(here)
library(tidyverse)
library(ggplot2)
library(dplyr)
library(tensorflow)
library(tfdatasets)
library(tidymodels)
library(keras)
library(caret)
library(themis)
#LOAD DATA
setwd(getwd())
dataIn = "../Data/Dataset-part-2.csv"
data_in <- read.csv(dataIn,header = TRUE, sep =',')
#View(data_in)
data <- data.frame(data_in)
summary(data)
plot(data$status)
```

Initial data cleaning

```
# Check for duplicates
sum(duplicated(data))
# No duplicates found

#Remove ID (irrelevant for predction) and FLAG_MOBIL (always 1)
data <- data %>% select(-ID, -FLAG_MOBIL)

# Remove students as there are only 4 and it will result in
# problems with One-Hot-Encoding in later stage
data <- filter(data, NAME_INCOME_TYPE != "Student")

# Factor variables
cols <- c("CODE_GENDER","FLAG_OWN_CAR","FLAG_OWN_REALTY","NAME_INCOME_TYPE","NAME_EDUCATION_TYPE", "NAMI
cols
data[cols] <- lapply(data[cols],factor)

# Replacing empty values with "Unknown" in Occupation_Type
levels(data$OCCUPATION_TYPE) <- c(levels(data$OCCUPATION_TYPE), "Unknown")
data$OCCUPATION_TYPE[is.na(data$OCCUPATION_TYPE)] <- "Unknown"

# Replacing C and X in Status with numeric values
levels(data$status)[levels(data$status)=="C"] <- "6"
#data$status[data$status == "X"] <- 7
levels(data$status)[levels(data$status)=="X"] <- "7"

# Summarize the class distribution
percentage <- prop.table(table(data$status)) * 100
cbind(freq=table(data$status), percentage=percentage, amount=table(data$status))
# Imbalanced dataset

summary(data)
```

Preprocessing and split of dataset

Shuffle data to prevent that we only train on the first 80% of our data and leave out patterns that only occur in the last 20%.
Afterwards split the data into training- and testset, while using stratification to preserve the percentage of samples for each class.

```r
set.seed(1)
# Shuffle data
data <- data[sample(1:nrow(data)), ]
# Split the data into train and test
# prop = 0.8 will result in 80/20 split
# strata = status will use stratification on variable status
trainIndex <- initial_split(data, prop = 0.8, strata = status)
trainingSet <- training(trainIndex)
testSet <- testing(trainIndex)

# Try to train with whole dataset instead of 80/20 split
# We did this to check if our model is overfitted as our evaluation on
# Test-Data was always around 70-80%
# As the result was the same, around 75%, we removed it again
# trainingSet <- data
```

Remove outliers

We identified outliers on CNT_CHILDREN and AMT_INCOME_TOTAL, so we remove them, if they are farther away than 1.5 the interquartile range.

```
# CNT_CHILDREN
boxplot(trainingSet$CNT_CHILDREN, horizontal=TRUE, main="CNT_CHILDREN")
Q1_Child <- quantile(trainingSet$CNT_CHILDREN, .25)
Q3_Child <- quantile(trainingSet$CNT_CHILDREN, .75)
IQR_Child <- IQR(trainingSet$CNT_CHILDREN)
# Now we keep the values within 1.5*IQR of Q1 and Q3
trainingSet <- subset(trainingSet, trainingSet$CNT_CHILDREN > (Q1_Child - 1.5*IQR_Child) & trainingSet$C
#dim(trainingSet)

# AMT_INCOME_TOTAL
boxplot(trainingSet$AMT_INCOME_TOTAL, horizontal=TRUE, main="AMT_INCOME_TOTAL")
Q1_AIT <- quantile(trainingSet$AMT_INCOME_TOTAL, .25)
Q3_AIT <- quantile(trainingSet$AMT_INCOME_TOTAL, .75)
IQR_AIT <- IQR(trainingSet$AMT_INCOME_TOTAL)
# Now we keep the values within 1.5*IQR of Q1 and Q3
trainingSet <- subset(trainingSet, trainingSet$AMT_INCOME_TOTAL > (Q1_AIT - 1.5*IQR_AIT) & trainingSet$A
#dim(trainingSet)
```

Create a recipe for preprocessing of tidymodels

In this step we use the library tidymodels. We define a "recipe" to preprocess the data in a pipeline.
* status is considered as model outcome, the . indicates all other variables as predictors
* step_dummy one-hot-encodes all nominal variables, but not status
* step_range will normalize numeric variables
* step_smote is oversampling the minority classes to get a ratio of 1 between all classes
We also tried a ratio of 0.25 and 0.5, not leading to better results.
step_downsample Undersampling the majority class was tried out, but as minority classes only have a few hundreds of examples, this will remove most training data. Therefore, we decided against it.
We also tried undersampling followed by oversampling, also not leading to better results. We decided against it as well.

```
set.seed(1)
preprocRecipe <-
  recipe(status ~., data = data) %>%
  step_dummy(all_nominal(), -status,  one_hot = TRUE) %>%
  step_range(all_predictors(), -all_nominal(), min = 0, max = 1) %>%
  step_smote(status, over_ratio = 1) %>%
 # step_downsample(status, under_ratio = 1, skip=TRUE) %>%
 # step_smote(status, over_ratio = 1, skip=TRUE) %>%
  step_dummy(status, one_hot = TRUE)# %>%
```

Bake the recipe to transform data

In this step the above defined receipt is extracted using the `prep()` function, and then use the `bake()` function to transform a set of data based on that recipe.

```
# retain = TRUE and new_data = NULL ensures that pre-processed trainingSet is
# returned
trainingSet_processed <- preprocRecipe %>%
  prep(trainingSet, retain = TRUE) %>%
  bake(new_data = NULL)
testSet_processed <- preprocRecipe %>%
  prep(testSet) %>%
  bake(new_data =testSet)

#summary(trainingSet_processed)
```

Split training and Test into data and target

Cutting off the last 8 columns. These 8 columns are the one-hot-encoded class attribute.

```r
# Turn data frame into data matrix to feed it into NN
matrix_data <- trainingSet_processed %>% select(-tail(names(trainingSet_processed), 8))
matrix_targets <- trainingSet_processed %>% select(tail(names(trainingSet_processed), 8))

matrix_data_test  <- testSet_processed %>% select(-tail(names(testSet_processed), 8))
matrix_targets_test  <- testSet_processed %>% select(tail(names(testSet_processed), 8))

#Subset only 100 entries for testing
#matrix_data <- matrix_data[1:100, ]
#matrix_targets <- matrix_targets[1:100, ]
```

Build Model

Function to build the model We tried a lot of different combinations:
* Different layers, also with dropout as layer_dropout(0.1) * Different unitsize
* Different learning_rate (0,5 to 0,01) with 0.2 showing best results while not being too “jumpy”
* Different optimizers (Adam, SGD, rmsprop)
We also tried to use momentum, not leading to better results. To check if the “dying ReLU”-problem is present, we’ve used elu, not leading to better results. Therefore we kept relu as activation function for all but the last layer, where it needs to be softmax

```
#train_data <- matrix_data
train_data <- data.matrix(matrix_data)
test_data <- data.matrix(matrix_data_test)
train_targets <- data.matrix(matrix_targets)
test_targets <- data.matrix(matrix_targets_test)


build_model <- function() {
  model <- keras_model_sequential() %>%
    layer_dense(units = 128, activation = "relu", input_shape = dim(train_data)[[2]]) %>%
    layer_dense(units = 1024, activation = "relu") %>%
    layer_dense(units = 1024, activation = "relu") %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dense(units = 32, activation = "relu") %>%
    layer_dense(units = 8, activation = "softmax")

  model %>% compile(
    optimizer = optimizer_sgd(learning_rate = 0.2),
    loss = "categorical_crossentropy",
    metrics = "categorical_accuracy"
  )

}
```

K-Fold-Validation

```r
# K-Fold valudation with k=10, started with 1500 epochs
k <- 10
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)

num_epochs <- 1500
all_acc_histories <- NULL
all_train_histories <- NULL
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,] #test_data#
  val_targets <- train_targets[val_indices,] #test_targets#

  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices,]
  model <- build_model()

  # Train the model (in silent mode, verbose=0)

  # One epoch = one forward pass and one backward pass of all the
  # training examples
  # Batch size = the number of training examples in one forward/backward pass.
  # The higher the batch size, the more memory space you'll need.
  # Number of iterations = number of passes, each pass using [batch size] number
  # of examples. To be clear, one pass = one forward pass + one backward pass
  # (we do not count the forward pass and backward pass as two different passes)
  # Batch size 32 much faster than 1, also the smaller the batch the less
  # accurate the estimate of the gradient will be.
  history <- model %>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 8192, verbose = 0
  )
  acc_history <- history$metrics$val_categorical_accuracy
  all_acc_histories <- rbind(all_acc_histories, acc_history)
  train_history <- history$metrics$categorical_accuracy
  all_train_histories <- rbind(all_train_histories, train_history)
}

#reticulate::py_last_error()
```

Evaluation

We can then compute the average of the per-epoch ACC scores for all folds.
Check on the data history for all k. We take the mean to estimate the best amount of epochs.

```r
average_acc_history <- data.frame(
  epoch = seq(1:ncol(all_acc_histories)),
  validation_acc = apply(all_acc_histories, 2, mean)
)

average_train_history <- data.frame(
  epoch = seq(1:ncol(all_train_histories)),
  train_acc = apply(all_train_histories, 2, mean)
)

head(max(average_train_history$train_acc))
head(max(average_acc_history$validation_acc))

#Plot the accuracy
ggplot(average_acc_history, aes(x = epoch, y = validation_acc)) + geom_smooth()

# Evaluate on Testset
eval <- evaluate(model, test_data, test_targets, verbose = 1)
head(eval)

eval <- evaluate(model, train_data, train_targets, verbose = 1)
head(eval)
```

Train the final model and save it

Mean-Results in k-fold validation showed that we have a peak at around 540 epochs with a maximum at 534 epochs. Therefore we train our model with 540 and 534 epochs again and compare the results.

```
# Train the final model
model <- build_model()
model %>% fit(train_data, train_targets,
          epochs = 540, batch_size = 8192, verbose = 0)
result <- model %>% evaluate(test_data, test_targets)
result

# Save model and history, please change the name
 write.csv(average_acc_history, "../Doc/Hand in/Average_Acc_history.csv", row.names=FALSE)
 save_model_hdf5(model, "../Doc/Hand in/Group 1 model.hfd5", overwrite = TRUE, include_optimizer = TRUE

# Save Training, Testing and Validation Data
 write.csv(train_data, "../Doc/Hand in/train_data.csv", row.names=FALSE)
 write.csv(test_data, "../Doc/Hand in/test_data.csv", row.names=FALSE)
 write.csv(train_targets, "../Doc/Hand in/train_targets.csv", row.names=FALSE)
 write.csv(test_targets, "../Doc/Hand in/test_targets.csv", row.names=FALSE)


# Load model
# Use model_history as precaution
# model_history <- load_model_hdf5("../Doc/Hand in/Group 1 model.hfd5", custom_objects = NULL, compile
```

Epochs	Data	Loss	Categorical_accuracy
534	Train	0.04170835	0.98532724
534	Test	1.8139522	0.7749593
540	Train	0.03299291	0.98883730
540	Test	1.7114468	0.7877533

As the model trains slightly better with 540 epochs, this is the model we kept for our Reality check.