Assignment 2

Classification Task

Sarah Rebecca Meyer

2024-2025

Contents

Assignment Overview

In this assignment, our goal is to develop a predictive model that determines how well a customer will repay their credit card debt ("status"), categorizing into eight distinct classes based on repayment patterns based on historical customer data. The primary objective was to explore various neural network architectures and hyperparameter optimization techniques to improve classification accuracy, not to predict whether the credit card application should be accepted or rejected.

The workflow entailed data exploration, preprocessing, and iterative experimentation with different neural network configurations. Challenges included handling class imbalance, addressing outliers, and ensuring the data was appropriately scaled and transformed for neural network training.

According to the assignment documentation, no extensive feature engineering is necessary, and all preprocessing will aim to ensure data compatibility with neural networks, transforming datasets into appropriate formats. Our approach will involve a systematic search for a "reasonable" model with careful validation and clear justification of the steps taken.

Abstract

Key steps in this classification task included cleaning and transforming the data, addressing imbalances in the target variable, and experimenting with diverse preprocessing pipelines to enhance model performance. Highlights of the approach included:

- Data Preprocessing: We brought the data into a format that would be compatible with neural networks, addressed class imbalances using techniques like SMOTE, normalized features, and encoded categorical

variables.
- Neural Network Architectures: We tried many different neural network architectures in an iterative approach, going through various configurations, including changes in layer depth, neuron count, activation functions, and regularization methods. We logged promising network architectures using a logging function and exported the learning curves to png so we could consult all these previous tries during later iterations and ultimately choose the best model.
- Hyperparameter Tuning: We conducted grid search and cross-validation to optimize parameters such as learning rate, dropout rates, and batch size.
- Evaluation: We mainly measured the performance of our models using the accuracy / categorical accuracy, as this would be the success metric used to evaluate and grade our project.

Among various models tested, a complex neural network with three dense layers and ReLU activations, incorporating dropout regularization and batch normalization, achieved the highest validation accuracy. Ultimately, the final model was re-trained using the complete dataset to maximise the learnings, exportet to a .rds file that gets imported in the reality check file. The best val_accuracy achieved was 88%, the overall accuracy on the whole dataset reached 95%.

Lessons Learned

- Preprocessing Drives Model Performance: Feature transformations, normalization, and handling outliers were critical in enabling the model to learn effectively.
- Domain Knowledge: Especially for Exploratory Data Analysis and Data Preprocessing, domain knowledge is crucial to understand whether the observed values are plausible and how to handle certain features. For the 4 of us – all coming from a medical informatics background - this was an important part of the project to familiarize ourselves with the features available for our neural network and get the maximum understanding of it.
- Class Imbalance: Techniques like SMOTE and class weighting impacted the model's ability to generalize across all categories.
- Iterative Model Development is Key: We experimented a lot with different architectures, layer configurations, and regularization techniques. This helped provide valuable insights into improving performance.
- Hyperparameter Tuning: Strategies like grid search and learning rate adjustments substantially improved model accuracy and convergence.
- Computational Resources: The training of large neural network models requires quite strong computational resources and some of our models required hours to train.
- One-Hot-Encoding: It is important when doing One-Hot Encoding on the training dataset, to save all possible values and one-hot-encode the test dataset using these, so the amount of features will be the same in the end. If we have a rare feature in the training dataset (like for example "student" in the NAME_INCOME_TYPE column, with 4 instances), chances are, it might not be represented in the test dataset. If this is the case, the matrix will be missing a feature and the model will not run through. This of course makes sense when we consider the later usage in production: if we had just one instance we would like to get a prediction for, we wouldn't know other possible values for the One-Hot Encoding, so we would need to pass them to the Encoder so the model could predict for our case.

Network Design

For this assignment, we chose a feed-forward neural network over convolutional or recursive architectures. This decision was based on the nature of the dataset and the task.
This is because the dataset consists of tabular data with numerical and categorical variables, which is best suited for feed-forward networks. As a comparison, image data / data where spatial relationships need to be captured are ideal for convolutional architectures, while sequential data benefits from recurrent layers to capture relationships. Our tabular data requires capturing relationships between features without an inherent order or structure, making feed-forward networks perfect for this task.

Even though we couldn't quite find any logical reason / advantage to use a CNN instead of a normal feed-forward network, we still gave it a try, but as expected, the performance was not satisfactory. The model landed at an accuracy of 78%, predicting the majority classes.

Setting the seed

For reproducibility, we set the seed.

```
set.seed(42)

# tensorflow::install_tensorflow()
tensorflow::set_random_seed(42)
```

Data Loading & Initial Analysis

In the first step, we import the data from the CSV file:

```
# Importing CSV file into R
setwd(dirname(getActiveDocumentContext()$path))
data <- read.csv("Dataset-part-2.csv", header = TRUE, sep = ",")
```

Then we have an outlook over the data:

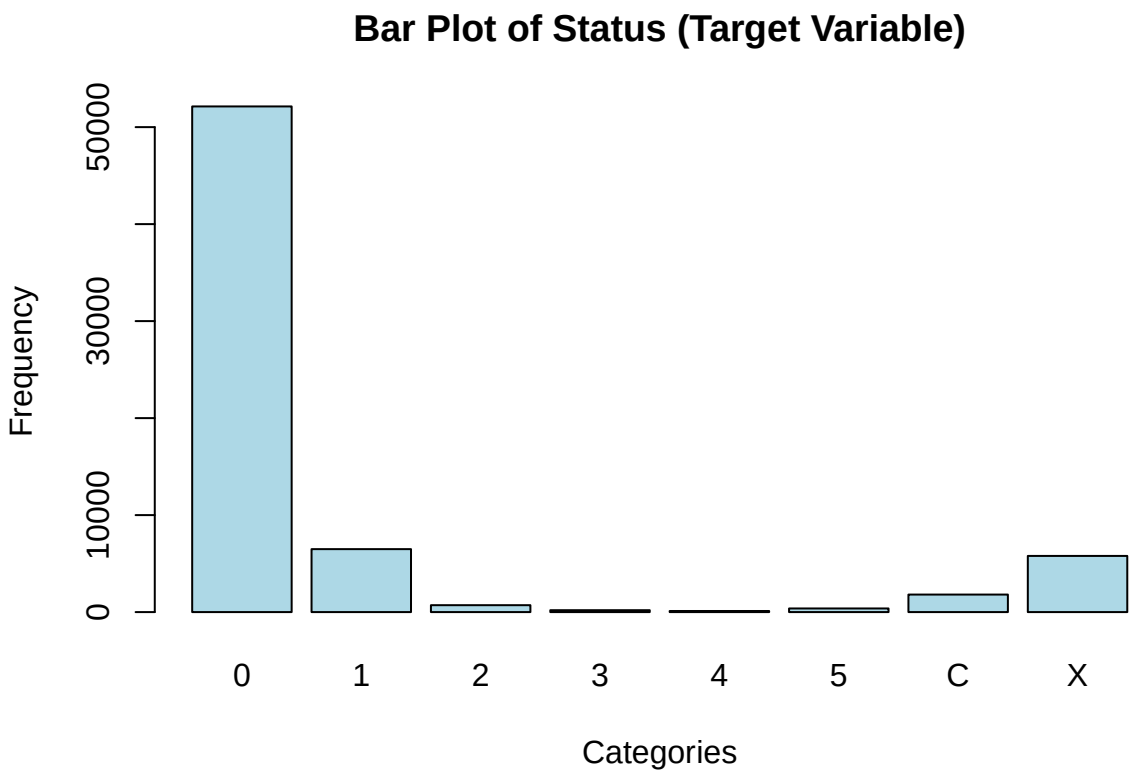
```
head(data)
```

The dataset has 67614 rows and 19 columns / variables.

Target variable

The variable we are trying to predict is the variable status. In the first step, we want to have an overlook over how the target is distributed in our dataset, because this will influence how we train our model later / which other steps we will have to take to make the model better.

```
# Create a bar plot for our target variable
barplot(table(data$status),
        main = "Bar Plot of Status (Target Variable)",
        xlab = "Categories",
        ylab = "Frequency",
        col = "lightblue",
        border = "black")
```



From this Bar Plot, we get the following information:

- Class Imbalance: the target variable of our dataset is highly imbalanced, with a large majority of the dataset falling into the category 0, and very few instances of the categories 2, 3, 4 and 5
- Not all labels of the target are numerical -> we will need to map the names

We have another look at this with the percentages to visualise this better.

```
target_distribution <- table(data$status)
target_percentage <- prop.table(target_distribution) * 100

distribution_df <- data.frame(
  Class = names(target_distribution),
  Count = as.integer(target_distribution),
  Percentage = round(as.numeric(target_percentage), 2)
)

# Print the distribution table
print(distribution_df)
```

```
##   Class Count Percentage
## 1     0 52133      77.10
## 2     1  6491       9.60
## 3     2   712       1.05
## 4     3   195       0.29
## 5     4   114       0.17
## 6     5   374       0.55
## 7     C  1805       2.67
## 8     X  5790       8.56
```

What does this mean for us?

- Potential Bias towards the Majority Class 0: with the class 0 being highly overrepresented in comparision to other classes, the model might lean towards predicting this class, which can lead to poor generalisation for the less represented classes such as 2, 3, 4, and 5.
- Possible Mitigation for this: Resampling Techniques like oversampling or undersampling, SMOTE
- Evaluation Metrics: we use both Accuracy and other metrics such as Precision, Recall, and F1-score. Our assignment will be graded upon the accuracy, which is why we will use this metric, but to ensure that the fewer representated classes are predicted well, we will also occasionally check other metrics.
- Test-Train Split: we will use stratified splitting to ensure that all classes are represented in both the train and test set equally

Input Variables

From the first glimpse on the data we can say:

- ID - to be removed as it does not carry significant information for the prediction
- Occupation Type - NA's
- CNT_CHILDREN might be strongly correlated with CNT_FAM_MEMBERS
- FLAGs (PHONE etc) - probably not important for the prediction

Summary

Using the skimr package to display a nice summary of the data for an initial overlook.

```
skim(data)
```

Table 1: Data summary

Name	data
Number of rows	67614
Number of columns	19
Column type frequency:	
character	9
numeric	10
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
CODE_GENDER	0	1.00	1	1	0	2	0
FLAG_OWN_CAR	0	1.00	1	1	0	2	0
FLAG_OWN_REALTY	0	1.00	1	1	0	2	0
NAME_INCOME_TYPE	0	1.00	7	20	0	5	0
NAME_EDUCATION_TYPE	0	1.00	15	29	0	5	0
NAME_FAMILY_STATUS	0	1.00	5	20	0	5	0
NAME_HOUSING_TYPE	0	1.00	12	19	0	6	0
OCCUPATION_TYPE	20699	0.69	7	21	0	18	0
status	0	1.00	1	1	0	8	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
ID	0	1	5908132.78	517395.25	5008804	5465941	5954270	6289080	7965248	
CNT_CHILDREN	0	1	0.42	0.72	0	0	0	1	19	
AMT_INCOME_TOTAL	0	1	178867.07	103396.62	26100	112500	157500	225000	6750000	
DAYS_BIRTH	0	1	-15914.38	4260.24	-25201	-19438	-15592	-12347	-7489	
DAYS_EMPLOYED	0	1	62022.42	139831.22	-17531	-2886	-1305	-321	365243	
FLAG_MOBIL	0	1	1.00	0.00	1	1	1	1	1	
FLAG_WORK_PHONE	0	1	0.20	0.40	0	0	0	0	1	
FLAG_PHONE	0	1	0.27	0.45	0	0	0	1	1	
FLAG_EMAIL	0	1	0.10	0.30	0	0	0	0	1	
CNT_FAM_MEMBERS	0	1	2.17	0.90	1	2	2	3	20	

Checking for NAs

We want to check, whether there are any NA values in our input features.

```
total_rows <- nrow(data)

na_summary <- sapply(data, function(column) {
  na_count <- sum(is.na(column))
  percentage <- (na_count / total_rows) * 100
  distinct_values_count <- length(unique(column))
  return(c(na_count = na_count, percentage = percentage, distinct_values_count =
   ↪  distinct_values_count))
})

na_summary_df <- as.data.frame(t(na_summary))
colnames(na_summary_df) <- c("NA Count", "Percentage NAs", "Distinct Values Count")

# Print the summary
print(na_summary_df)
```

```
##                     NA Count Percentage NAs Distinct Values Count
## ID                         0        0.00000                  67614
## CODE_GENDER                0        0.00000                      2
## FLAG_OWN_CAR               0        0.00000                      2
## FLAG_OWN_REALTY            0        0.00000                      2
## CNT_CHILDREN               0        0.00000                     10
## AMT_INCOME_TOTAL           0        0.00000                    733
## NAME_INCOME_TYPE           0        0.00000                      5
## NAME_EDUCATION_TYPE        0        0.00000                      5
## NAME_FAMILY_STATUS         0        0.00000                      5
## NAME_HOUSING_TYPE          0        0.00000                      6
## DAYS_BIRTH                 0        0.00000                  16002
## DAYS_EMPLOYED              0        0.00000                   8704
## FLAG_MOBIL                 0        0.00000                      1
## FLAG_WORK_PHONE            0        0.00000                      2
## FLAG_PHONE                 0        0.00000                      2
## FLAG_EMAIL                 0        0.00000                      2
## OCCUPATION_TYPE        20699       30.61348                     19
## CNT_FAM_MEMBERS            0        0.00000                     11
## status                     0        0.00000                      8
```

We can see that the only NA values we have are in the column OCCUPATION_TYPE and that they make up for about 30% of the data.

OCCUPATION_TYPE

We see that a large proportion of the instances where OCCUPATION_TYPE is NA, has a positive value for DAYS_EMPLOYED, meaning that the person is unemployed. We have a closer look at the distribution of all the instances where the OCCUPATION_TYPE is NA to see the distribution.

```
na_occupation_data <- subset(data, is.na(OCCUPATION_TYPE))

# Calculate the number of instances with DAYS_EMPLOYED > 0
days_employed_positive <- sum(na_occupation_data$DAYS_EMPLOYED > 0, na.rm = TRUE)

# Calculate the number of instances with DAYS_EMPLOYED <= 0
days_employed_not_positive <- sum(na_occupation_data$DAYS_EMPLOYED <= 0, na.rm = TRUE)

# Calculate the total number of instances with NA in OCCUPATION_TYPE
total_na_occupation <- nrow(na_occupation_data)

# Calculate percentages
percentage_employed_positive <- (days_employed_positive / total_na_occupation) * 100
percentage_employed_not_positive <- (days_employed_not_positive / total_na_occupation) *
 ↪  100

# Print results
cat("Number with DAYS_EMPLOYED > 0:", days_employed_positive, "\n")
```

```
## Number with DAYS_EMPLOYED > 0: 11855
```

```
cat("Percentage with DAYS_EMPLOYED > 0:", percentage_employed_positive, "%\n\n")
```

```
## Percentage with DAYS_EMPLOYED > 0: 57.2733 %
```

```
cat("Number with DAYS_EMPLOYED <= 0:", days_employed_not_positive, "\n")
```

```
## Number with DAYS_EMPLOYED <= 0: 8844
```

```
cat("Percentage with DAYS_EMPLOYED <= 0:", percentage_employed_not_positive, "%\n")
```

```
## Percentage with DAYS_EMPLOYED <= 0: 42.7267 %
```

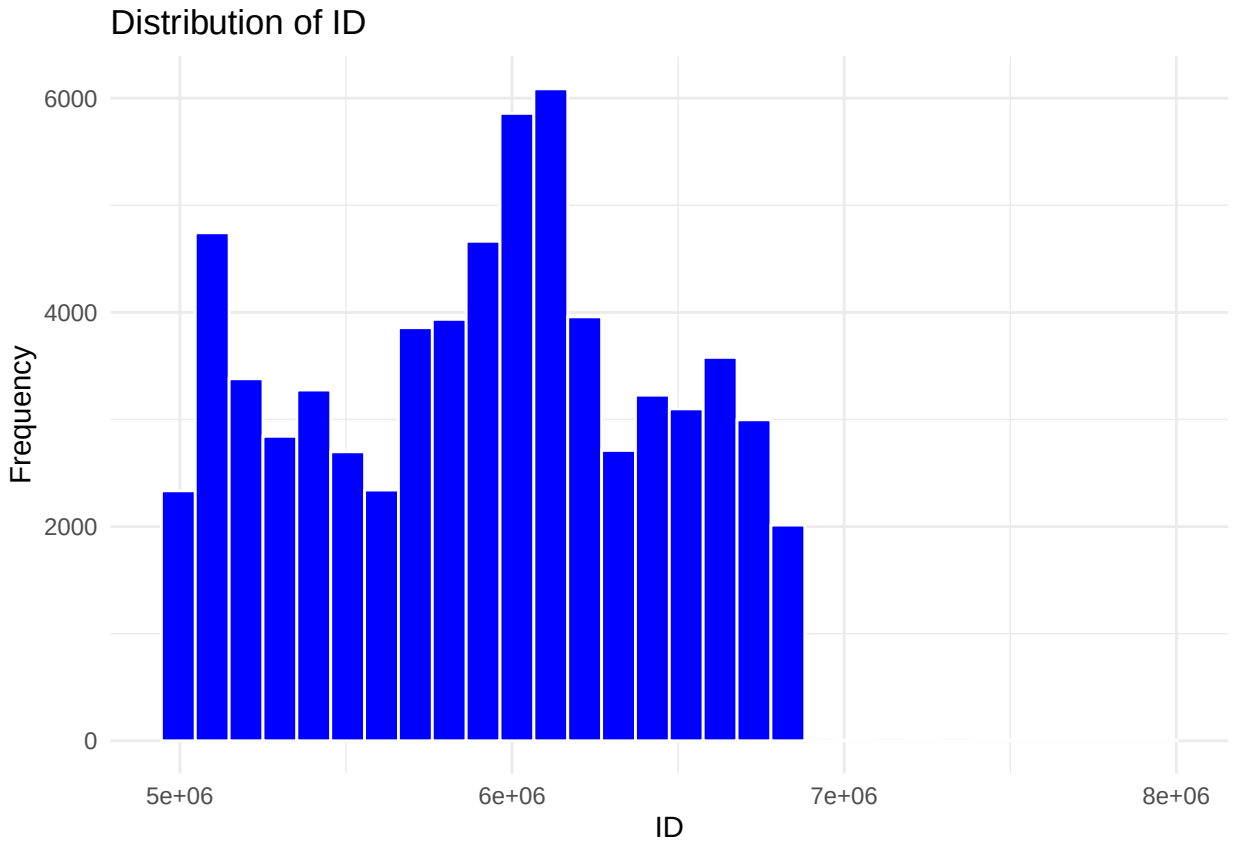
57% of the 30% instances with NA in the OCCUPATION_TYPE column are unemployed. We create a new value in the OCCUPATION_TYPE column for "Unemployed", after which we will only have 8844 instances with NA left, which we can impute later on.

Distribution of numerical columns

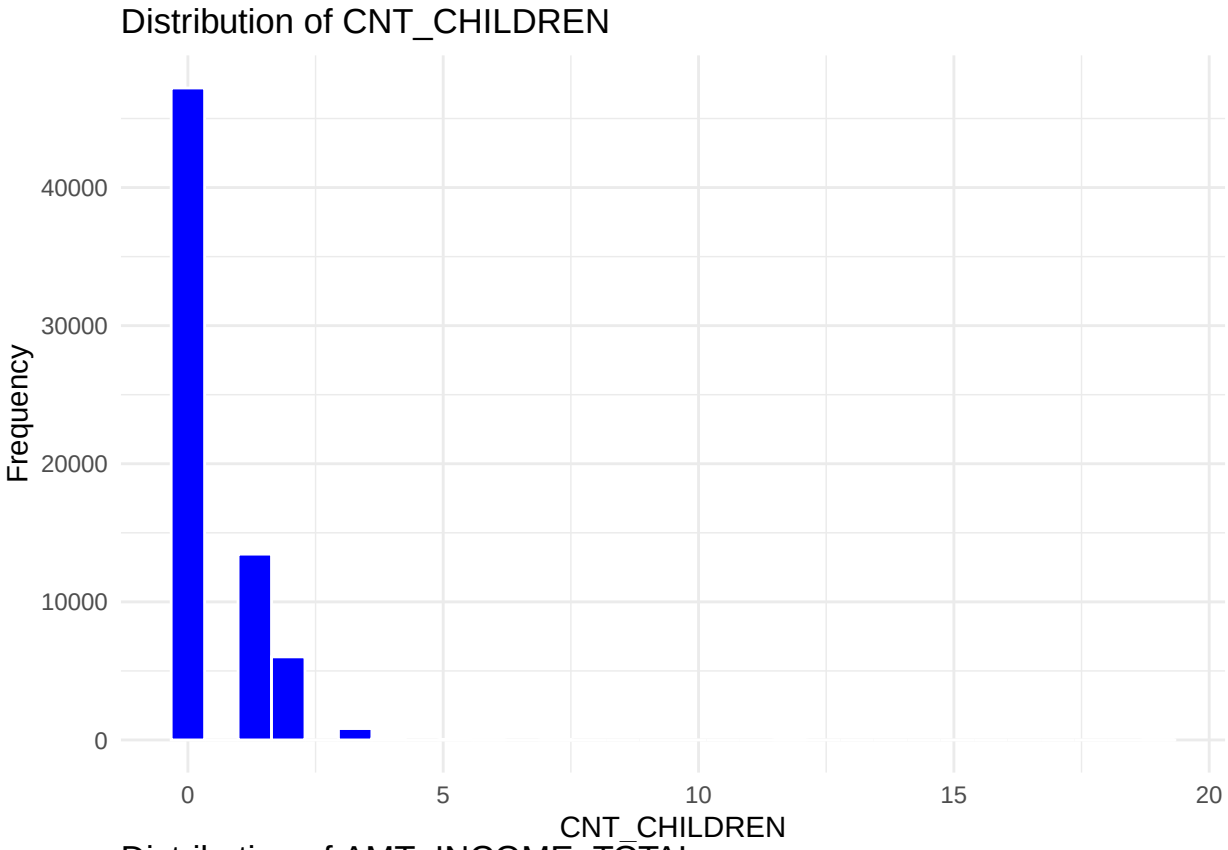
To see the distribution & spread of the numerical columns, we plot them.

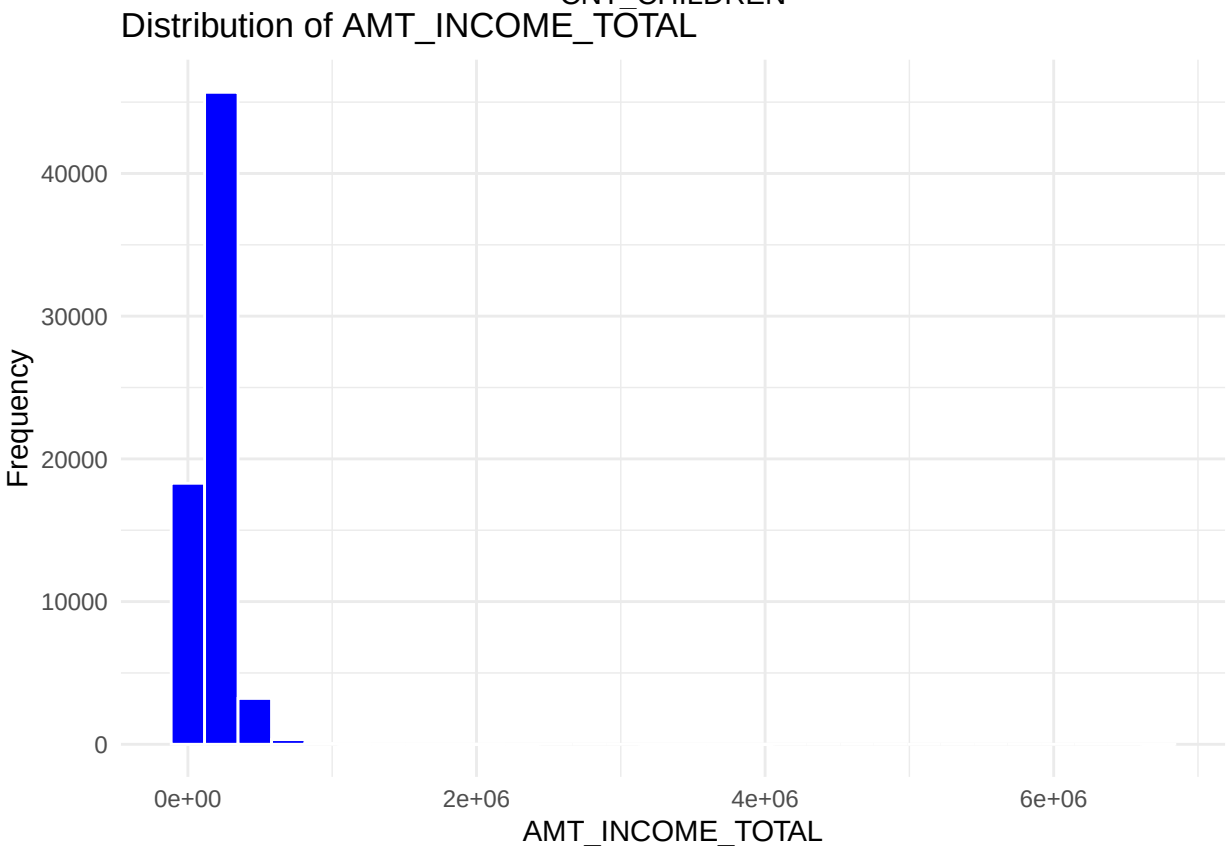
```
# Function
plot_all_distributions <- function(data) {
  for (col in names(data)) {
    if (is.numeric(data[[col]])) {
      print(
        ggplot(data, aes(x = .data[[col]])) +
          geom_histogram(bins = 30, fill = "blue", color = "white") +
```

```
          labs(title = paste("Distribution of", col), x = col, y = "Frequency") +
          theme_minimal()
      )
    } else {
      message(paste("Skipping non-numeric column:", col))
    }
  }
}

# Explore the distributions for all columns
plot_all_distributions(data)
```



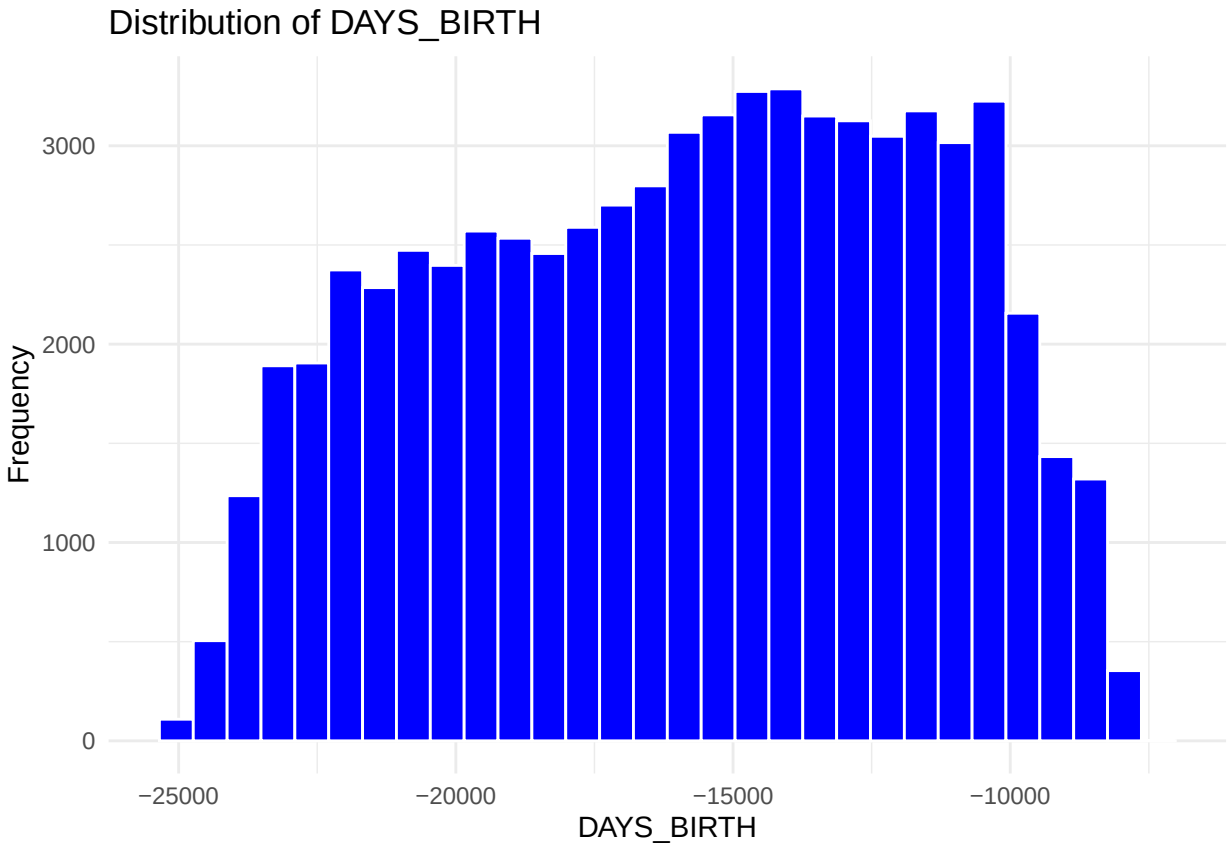

```
## Skipping non-numeric column: CODE_GENDER
## Skipping non-numeric column: FLAG_OWN_CAR
## Skipping non-numeric column: FLAG_OWN_REALTY
```

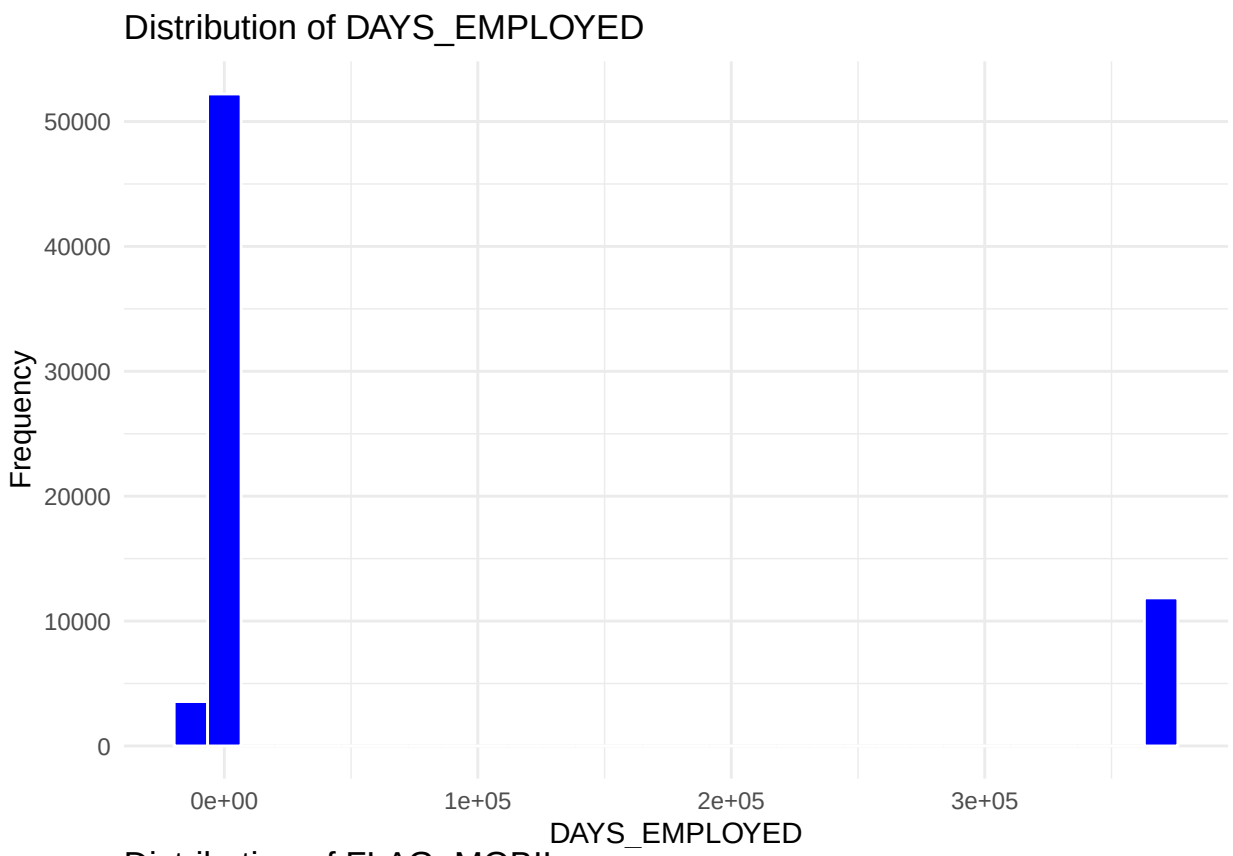


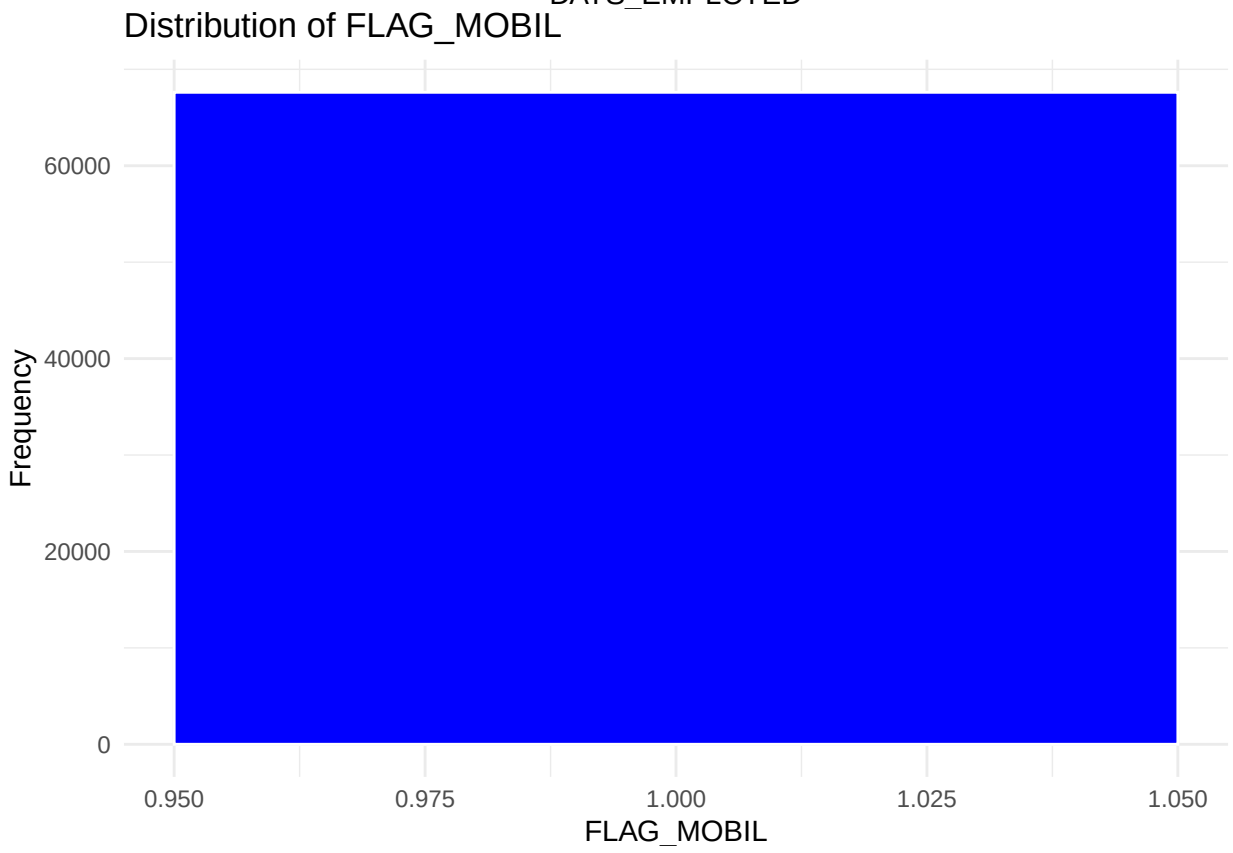


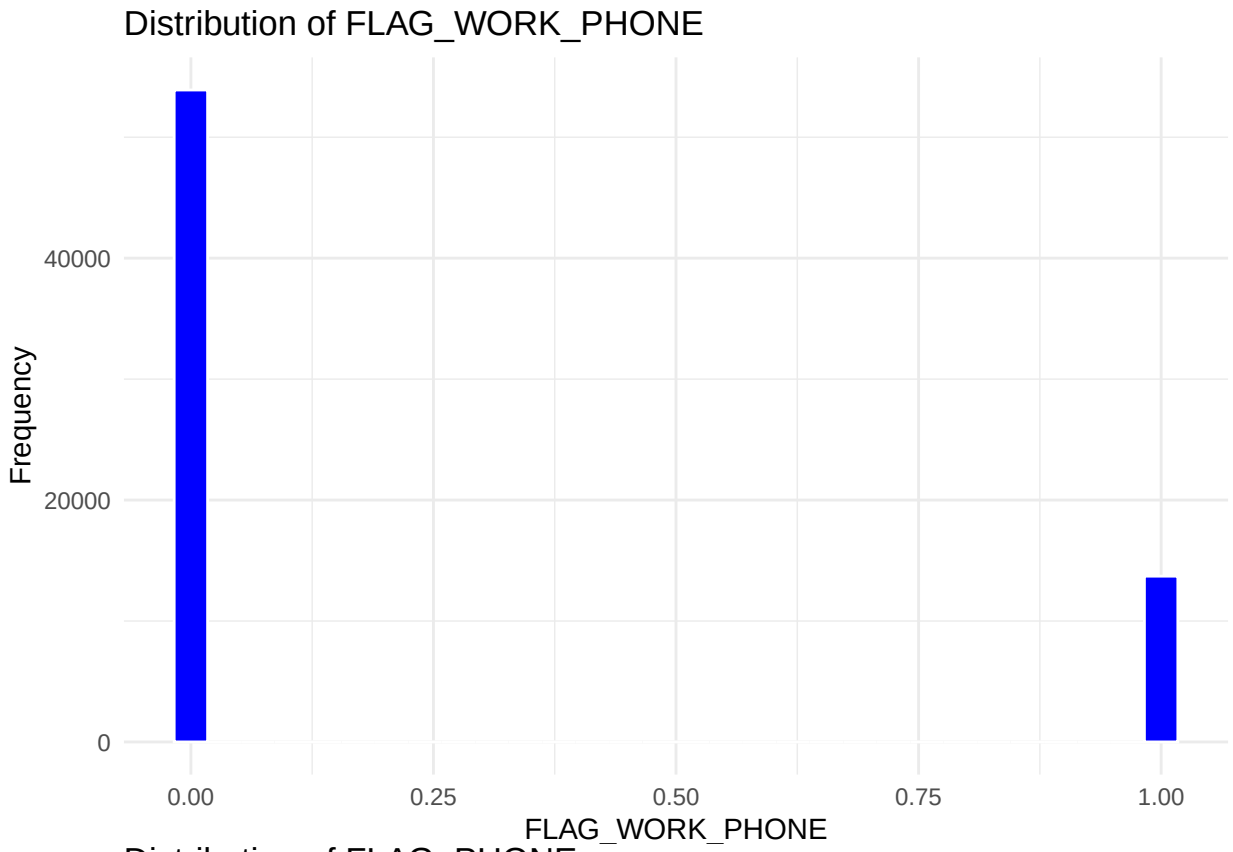

```
## Skipping non-numeric column: NAME_INCOME_TYPE
```

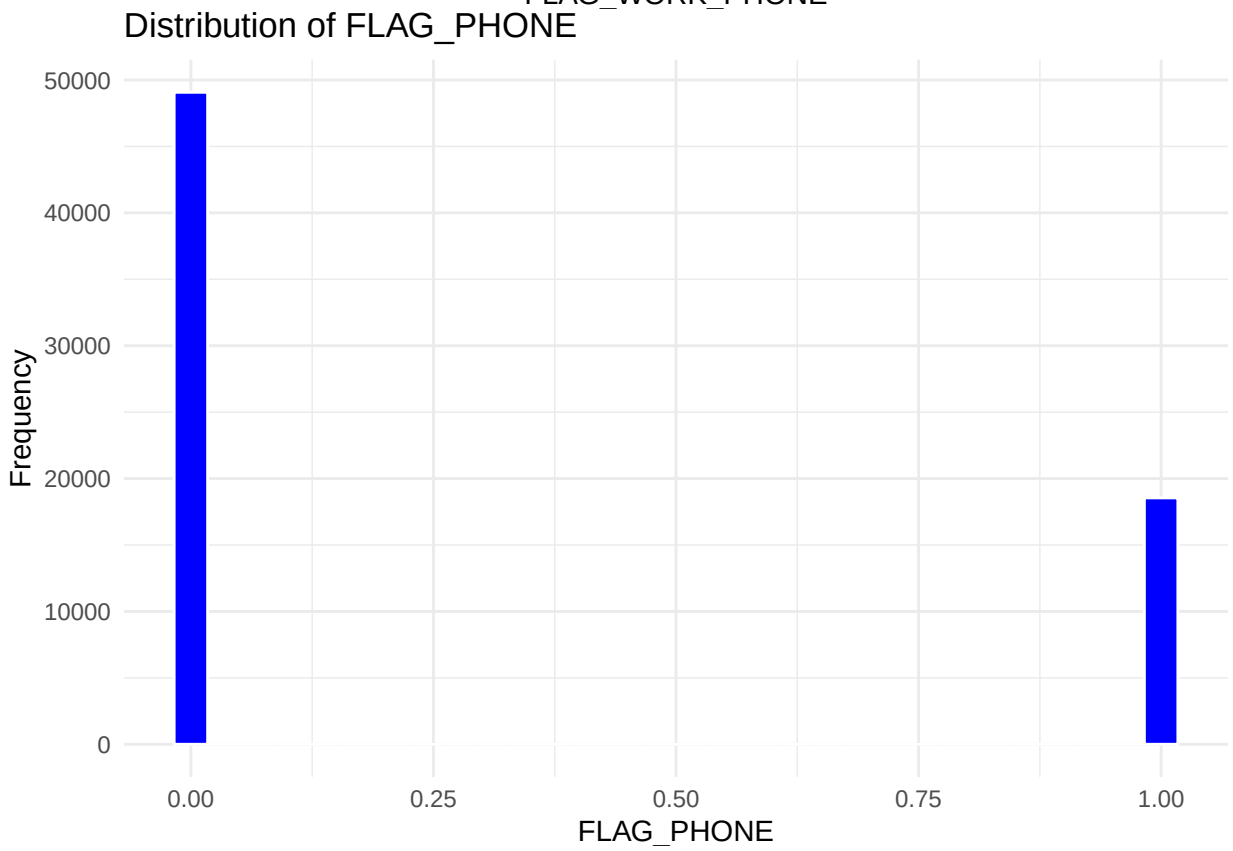
```
## Skipping non-numeric column: NAME_EDUCATION_TYPE
## Skipping non-numeric column: NAME_FAMILY_STATUS
## Skipping non-numeric column: NAME_HOUSING_TYPE
```

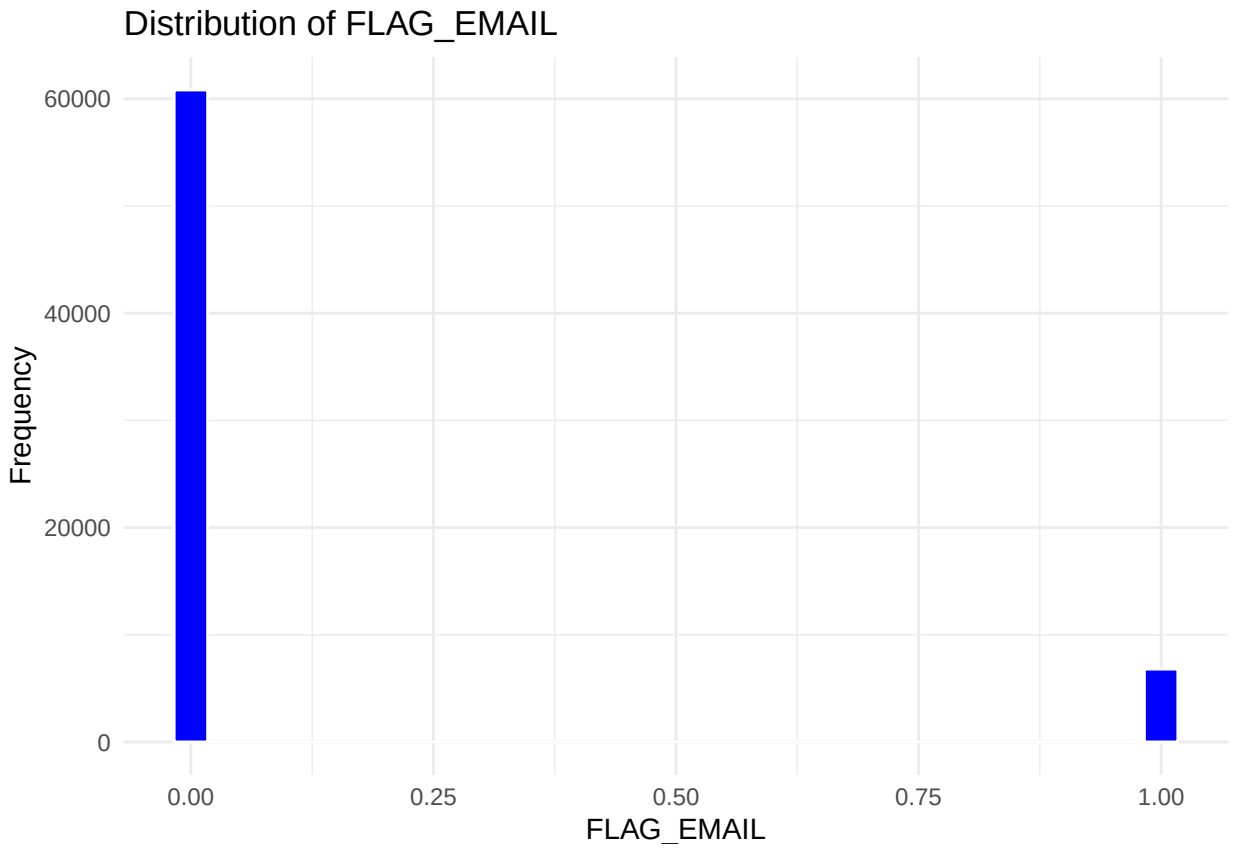



Distribution of DAYS_EMPLOYED
Frequency
50000
40000
30000
20000
10000
0
0e+00
1e+05
2e+05
3e+05
DAYS_EMPLOYED

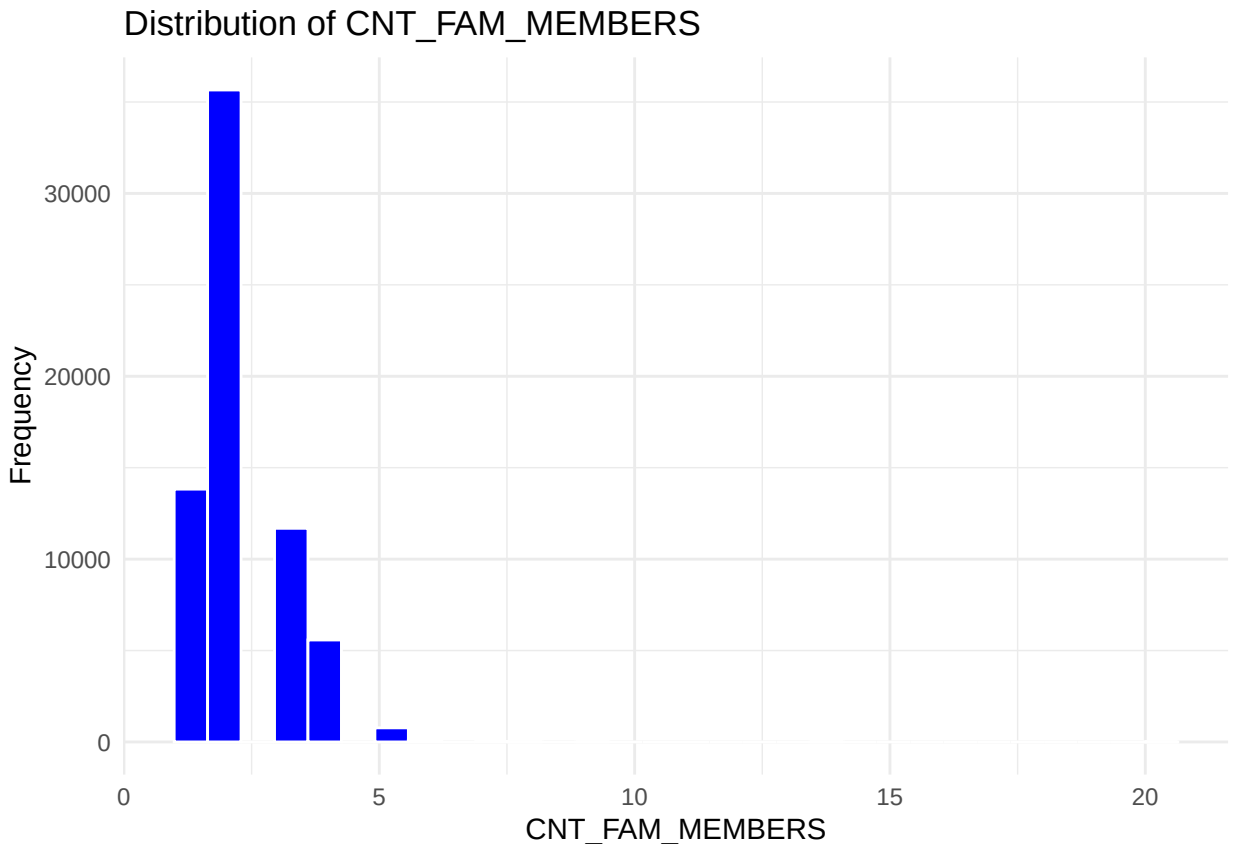

Distribution of FLAG_MOBIL
Frequency
60000
40000
20000
0
0.950
0.975
1.000
1.025
1.050
FLAG_MOBIL


Distribution of FLAG_WORK_PHONE
Frequency
40000
20000
0
0.00
0.25
0.50
0.75
1.00
FLAG_WORK_PHONE


Distribution of FLAG_PHONE
Frequency
50000
40000
30000
20000
10000
0
0.00
0.25
0.50
0.75
1.00
FLAG_PHONE




```
## Skipping non-numeric column: OCCUPATION_TYPE
```




```
## Skipping non-numeric column: status
```

In these distribution we see the following:

- CNT_CHILDREN: outliers
- AMT_INCOME_TOTAL: We see that the column AMT_INCOME_TOTAL contains some outliers.
- DAYS_EMPLOYED: a high amount of extremely high numbers?
- CNT_FAM_MEMBERS: outliers

Identifying Outliers

We have a closer look at the extreme outliers (we used the modified z-score to identify them; a mod z-score of > 3.5 signifies an extreme outlier.)

Mod. Z-Score To identify outliers, we often use the modified z-score, for which we define a function here.

```
calculate_mod_zscore <- function(x) {
  median_x <- median(x, na.rm = TRUE)
  mad_x <- mad(x, constant = 1.4826, na.rm = TRUE) # Adjusted MAD
  mod_zscore <- (x - median_x) / mad_x
  return(mod_zscore)
}
```

IQR Method For some other instances where we cannot use the mod. z-score, we use the IQR*3 method, for which we define the bounds here.

AMT_INCOME_TOTAL This column contains the income. We already know it contains outliers.

```r
summary_stats <- summary(data$AMT_INCOME_TOTAL)
print(summary_stats)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   26100  112500  157500  178867  225000 6750000
```

```r
# Apply the mod. z-score function to AMT_INCOME_TOTAL
data$mod_zscore <- calculate_mod_zscore(data$AMT_INCOME_TOTAL)

threshold <- 3.5

# Identify extreme outliers
extreme_outliers <- data %>%
  filter(abs(mod_zscore) > threshold)

# Print the extreme outliers
print(paste("Number of extreme outliers:", nrow(extreme_outliers)))
```

```
## [1] "Number of extreme outliers: 2063"
```

We see that there are extreme outliers and keep this in mind for when we later train the models - we will use normalisation on the data prior to model creation and this distribution with the extreme outliers could lead to slower convergence during training or harm the generalisation.

To see the effect this has on the data distribution, we look at the skewness.

```r
skewness_value <- skewness(data$AMT_INCOME_TOTAL, na.rm = TRUE)
print(paste("Skewness of AMT_INCOME_TOTAL:", skewness_value))
```

```
## [1] "Skewness of AMT_INCOME_TOTAL: 8.48451114662705"
```

This indicates an extremely high skewness, meaning that it is likely better that we remove the outliers. We keep this in mind and will maybe do this later after our initial model was trained, to see whether it performs better without the outliers. Options to handle this would be:

- remove the outliers
- apply log-transformation on the feature or both..?

CNT_CHILDREN Because the distribution of CNT_CHILDREN is mostly clustered at 0, we cannot use the modified z-score, so we use the IQR * 3 methods here to identify extreme outliers.

```r
data$CNT_CHILDREN <- as.numeric(data$CNT_CHILDREN)

# Calculate Q1, Q3, and IQR
Q1 <- quantile(data$CNT_CHILDREN, 0.25, na.rm = TRUE)
Q3 <- quantile(data$CNT_CHILDREN, 0.75, na.rm = TRUE)
IQR_value <- Q3 - Q1

lower_bound_extreme <- Q1 - 3 * IQR_value
upper_bound_extreme <- Q3 + 3 * IQR_value

# Identify extreme outliers
extreme_outliers <- data %>%
  filter(CNT_CHILDREN < lower_bound_extreme | CNT_CHILDREN > upper_bound_extreme)

# Print the number of extreme outliers and their details
print(paste("Number of extreme outliers:", nrow(extreme_outliers)))
```

```
## [1] "Number of extreme outliers: 22"
```

This shows us 22 extreme outliers of families that have 5 or more children (up to 19 children). Most likely, it will be best to get rid of these outliers in order to have a good spread in the normalization later on.

CNT_FAM_MEMBERS We use the same approach as for CNT_CHILDREN, because most values are quite low and we cannot use the modified z-score, so we use the IQR * 3 methods here to identify extreme outliers.

```
# Calculate Q1, Q3, and IQR
Q1 <- quantile(data$CNT_FAM_MEMBERS, 0.25, na.rm = TRUE)
Q3 <- quantile(data$CNT_FAM_MEMBERS, 0.75, na.rm = TRUE)
IQR_value <- Q3 - Q1

lower_bound_extreme <- Q1 - 3 * IQR_value
upper_bound_extreme <- Q3 + 3 * IQR_value

# Identify extreme outliers
extreme_outliers <- data %>%
  filter(CNT_FAM_MEMBERS < lower_bound_extreme | CNT_FAM_MEMBERS > upper_bound_extreme)

# Print the number of extreme outliers and their details
print(paste("Number of extreme outliers:", nrow(extreme_outliers)))
```

```
## [1] "Number of extreme outliers: 21"
```

This shows us 21 extreme outliers of families that consist of 7 or more family members. These are likely the same as the ones from CNT_CHILDREN, and as mentioned before, it might be best to get rid of these outliers in order to have a good spread in the normalization later on.

DAYS_BIRTH Having the amount of days since birth is not ideal, because it is not very interpretable and can distort the normalization by compressing the values in a certain area. Having the information in a smaller range (e.g. years instead of days) could be beneficial for the model. This is something we might explore later on.

```
data$DAYS_BIRTH <- abs(data$DAYS_BIRTH) # transforming it into positive

summary_stats <- summary(data$DAYS_BIRTH)
print(summary_stats)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    7489   12347   15592   15914   19438   25201
```

```
data$mod_zscore <- calculate_mod_zscore(data$DAYS_BIRTH)

threshold <- 3.5

extreme_outliers <- data %>%
  filter(abs(mod_zscore) > threshold)

# Print the extreme outliers
print(paste("Number of extreme outliers:", nrow(extreme_outliers)))
```

```
## [1] "Number of extreme outliers: 0"
```

There are no extreme outliers.

DAYS_EMPLOYED Here we already know that the value for all unemployed people is 365'243 which will extremely much distort our distribution and harm the normalisation.

```
summary_stats <- summary(data$DAYS_EMPLOYED)
print(summary_stats)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -17531   -2886   -1305   62022    -321  365243
```

```
# Apply the function to DAYS_EMPLOYED
data$mod_zscore <- calculate_mod_zscore(data$DAYS_EMPLOYED)

threshold <- 3.5

# Identify extreme outliers
extreme_outliers <- data %>%
  filter(abs(mod_zscore) > threshold)

# Print the extreme outliers
print(paste("Number of extreme outliers:", nrow(extreme_outliers)))
```

```
## [1] "Number of extreme outliers: 14611"
```

This is definitely a problem that needs to be addressed because it would render the whole column unusable for the model if left like this.

CNT_CHILDREN / CNT_FAM_MEMBERS

These two features seem quite similar and we want to have a closer look at the correlation.

```
data$CNT_CHILDREN <- as.numeric(data$CNT_CHILDREN)
data$CNT_FAM_MEMBERS <- as.numeric(data$CNT_FAM_MEMBERS)

# Calculate correlation
correlation <- cor(data$CNT_CHILDREN, data$CNT_FAM_MEMBERS, method = "pearson")
print(paste("Correlation:", correlation))
```

```
## [1] "Correlation: 0.878420343008209"
```

This correlation is quite high with 0.87, we keep this in mind and will later try to train models both with and without one of the features, to see whether the model performs better or not.

```
data$status_numeric <- as.numeric(as.factor(data$status)) - 1

# Check the correlation for each feature with the target
correlation_children <- cor(data$CNT_CHILDREN, data$status_numeric, use = "complete.obs")
correlation_family <- cor(data$CNT_FAM_MEMBERS, data$status_numeric, use =
 ↪  "complete.obs")

# Print correlations
cat("Correlation between CNT_CHILDREN and payment_status:", correlation_children, "\n")
```

```
## Correlation between CNT_CHILDREN and payment_status: -0.03025593
```

```r
cat("Correlation between CNT_FAM_MEMBERS and payment_status:", correlation_family, "\n")
```

```
## Correlation between CNT_FAM_MEMBERS and payment_status: -0.01656999
```

Both do not seem to have a very high correlation, but this could be because the correlation might not be linear, so we do not know.

Distribution of non-numerical columns

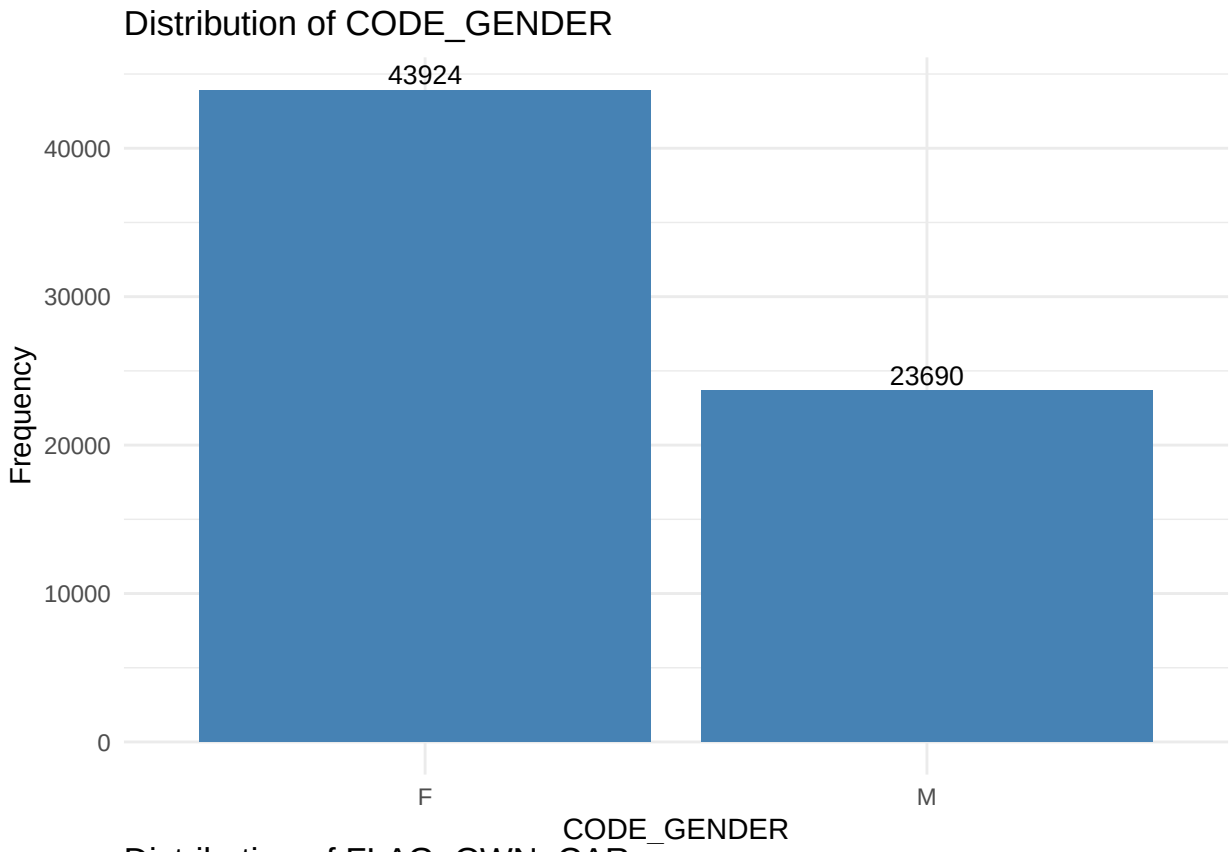
First we transform all non-numerical columns to factors so we can analyse them.

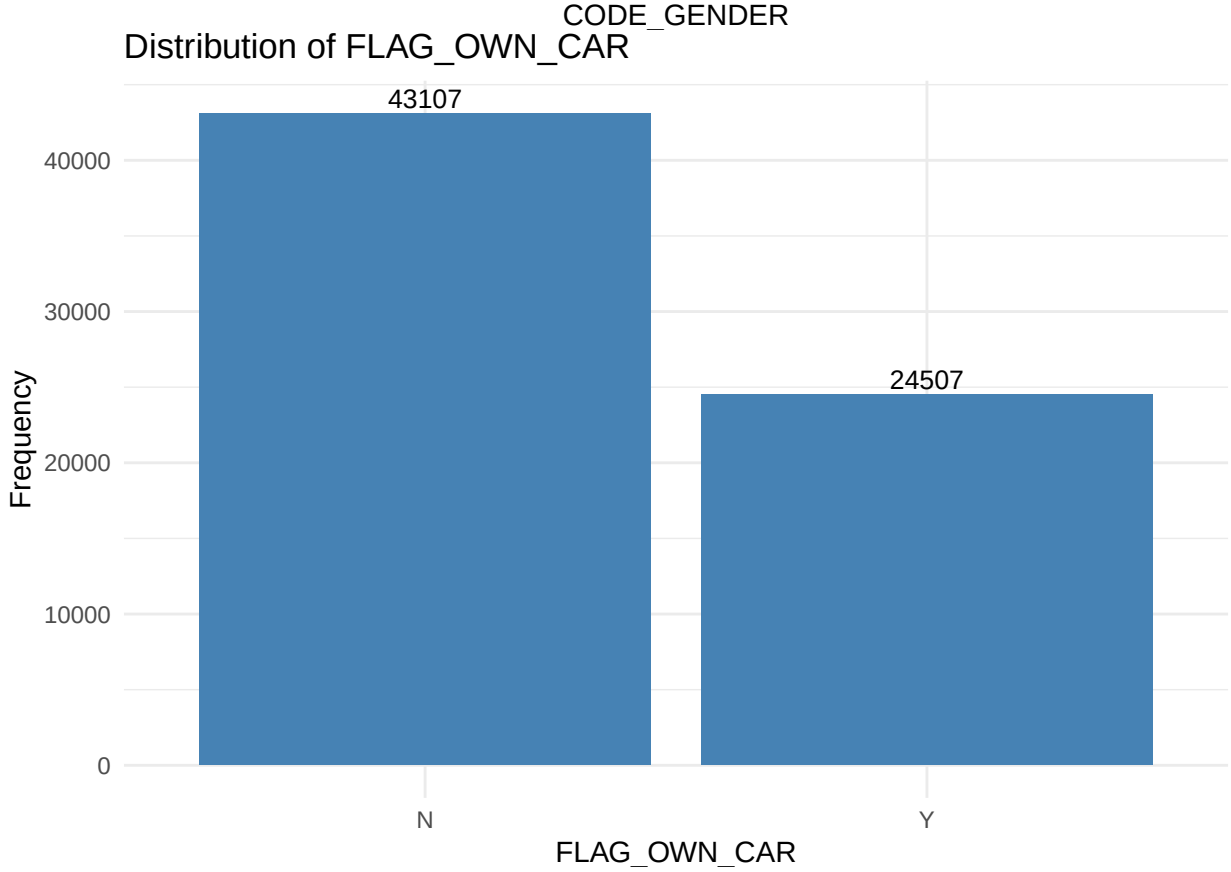
```r
data$CODE_GENDER <- as.factor(data$CODE_GENDER)
data$FLAG_OWN_CAR <- as.factor(data$FLAG_OWN_CAR)
data$FLAG_OWN_REALTY <- as.factor(data$FLAG_OWN_REALTY)
data$NAME_INCOME_TYPE <- as.factor(data$NAME_INCOME_TYPE)
data$NAME_EDUCATION_TYPE <- as.factor(data$NAME_EDUCATION_TYPE)
data$NAME_FAMILY_STATUS <- as.factor(data$NAME_FAMILY_STATUS)
data$NAME_HOUSING_TYPE <- as.factor(data$NAME_HOUSING_TYPE)
data$OCCUPATION_TYPE <- as.factor(data$OCCUPATION_TYPE)
```

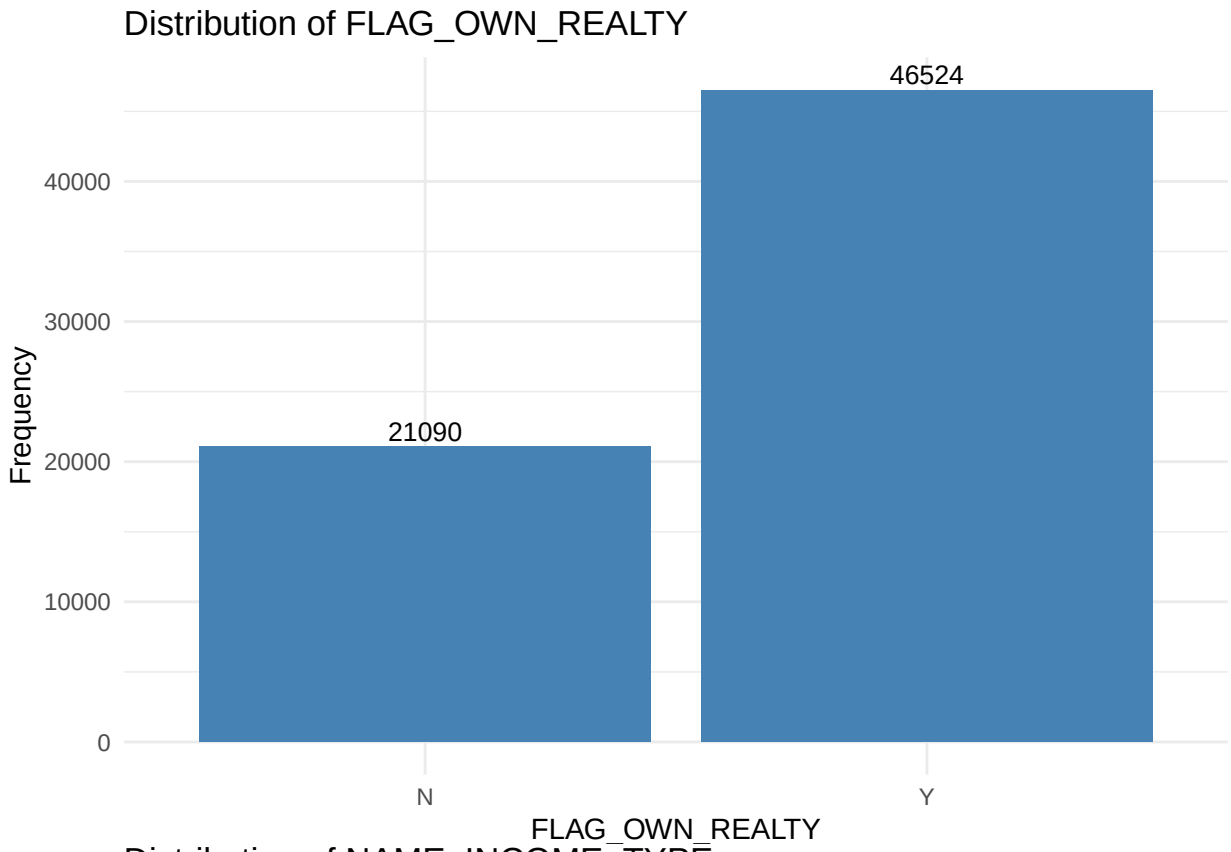
Now we visualize the distribution of the non-numerical columns.

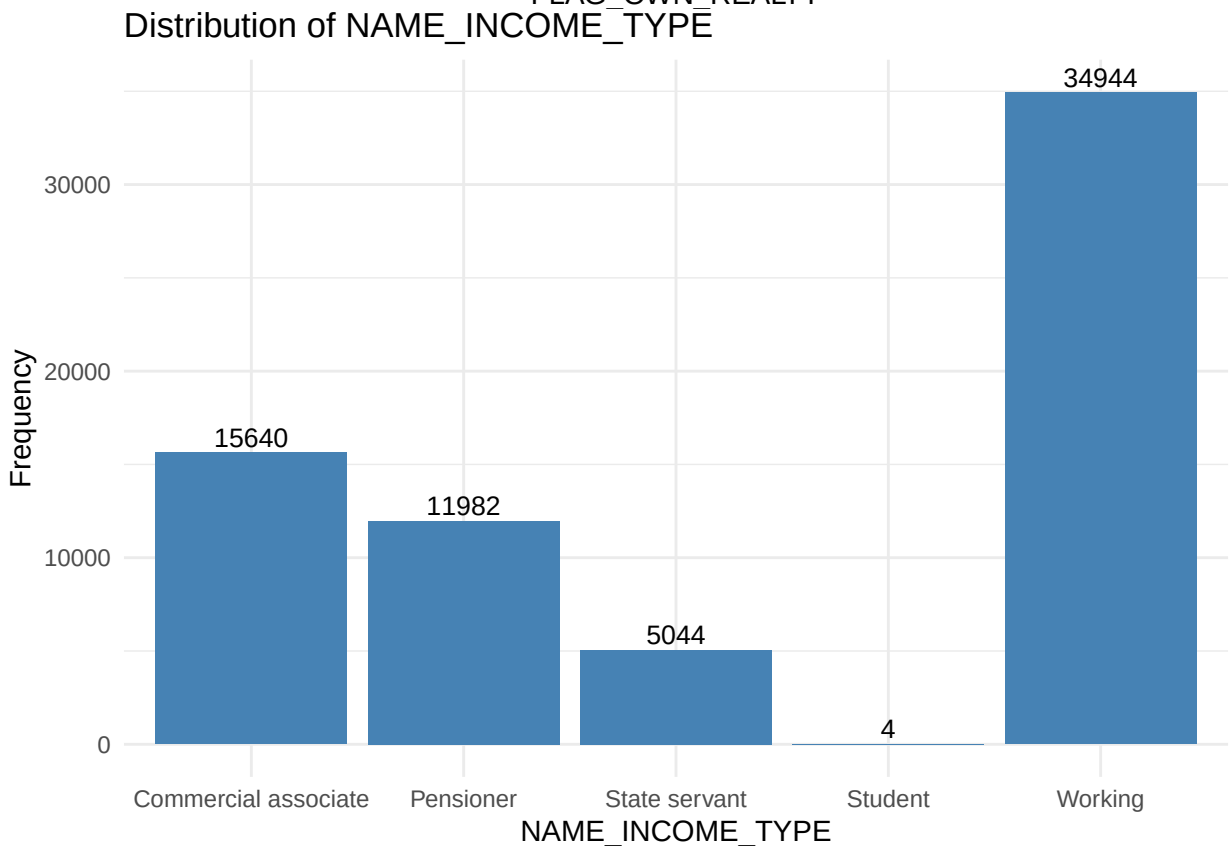
```r
non_numerical_columns <- data %>%
  select(where(~ is.character(.) || is.factor(.)))

# Loop through each non-numerical column and create a bar plot for its distribution
for (col_name in colnames(non_numerical_columns)) {
  # Create a frequency table for the column
  plot_data <- data %>%
    count(!!sym(col_name)) %>%
    rename(Frequency = n)

  # Create the bar chart for the non-numerical column with counts as labels
  p <- ggplot(plot_data, aes_string(x = col_name, y = "Frequency")) +
    geom_bar(stat = "identity", fill = "steelblue") +
    geom_text(aes_string(label = "Frequency"), vjust = -0.3, size = 3.5) + # Add count
     ↪  labels
    labs(title = paste("Distribution of", col_name),
         x = col_name,
         y = "Frequency") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    theme_minimal()

  # Explicitly print the plot
  print(p)
}
```

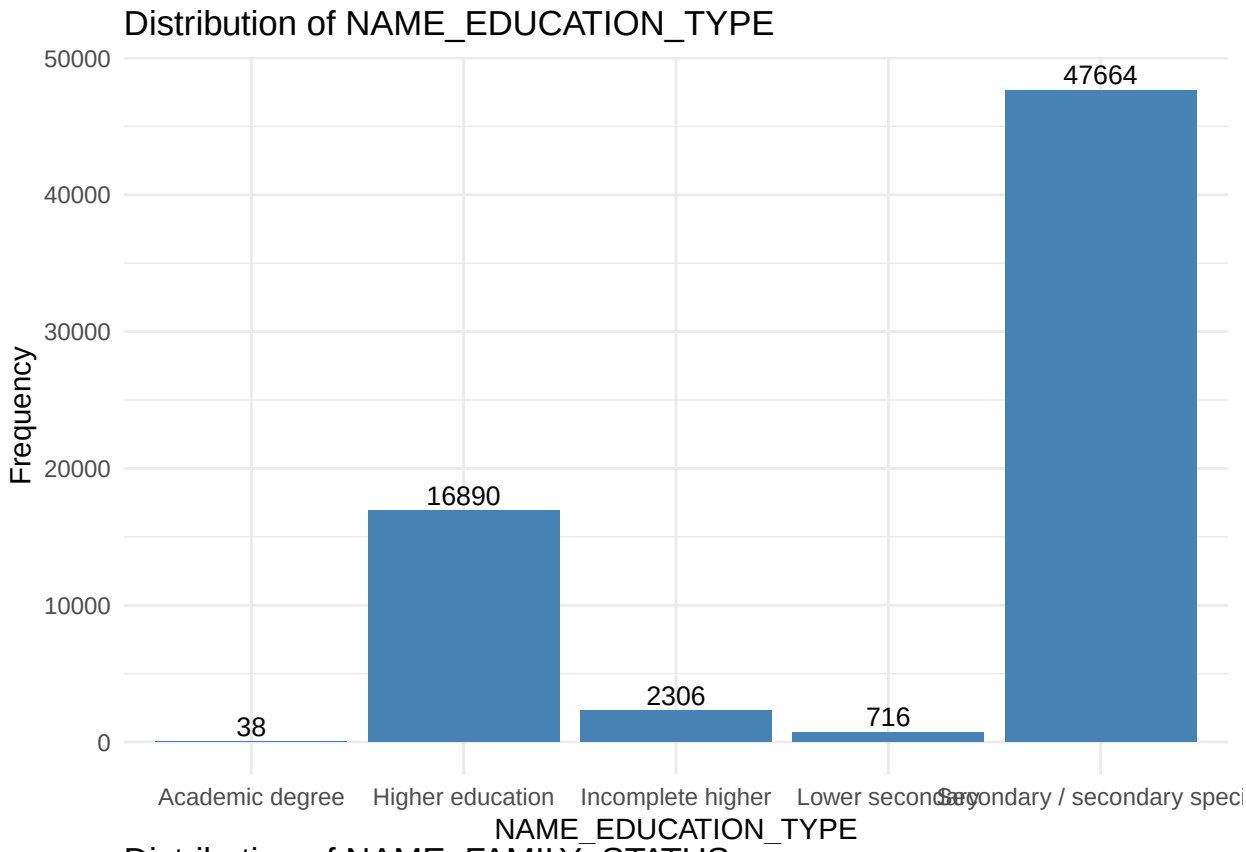
```
## Warning: `aes_string()` was deprecated in ggplot2 3.0.0.
## i Please use tidy evaluation idioms with `aes()`.
## i See also `vignette("ggplot2-in-packages")` for more information.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

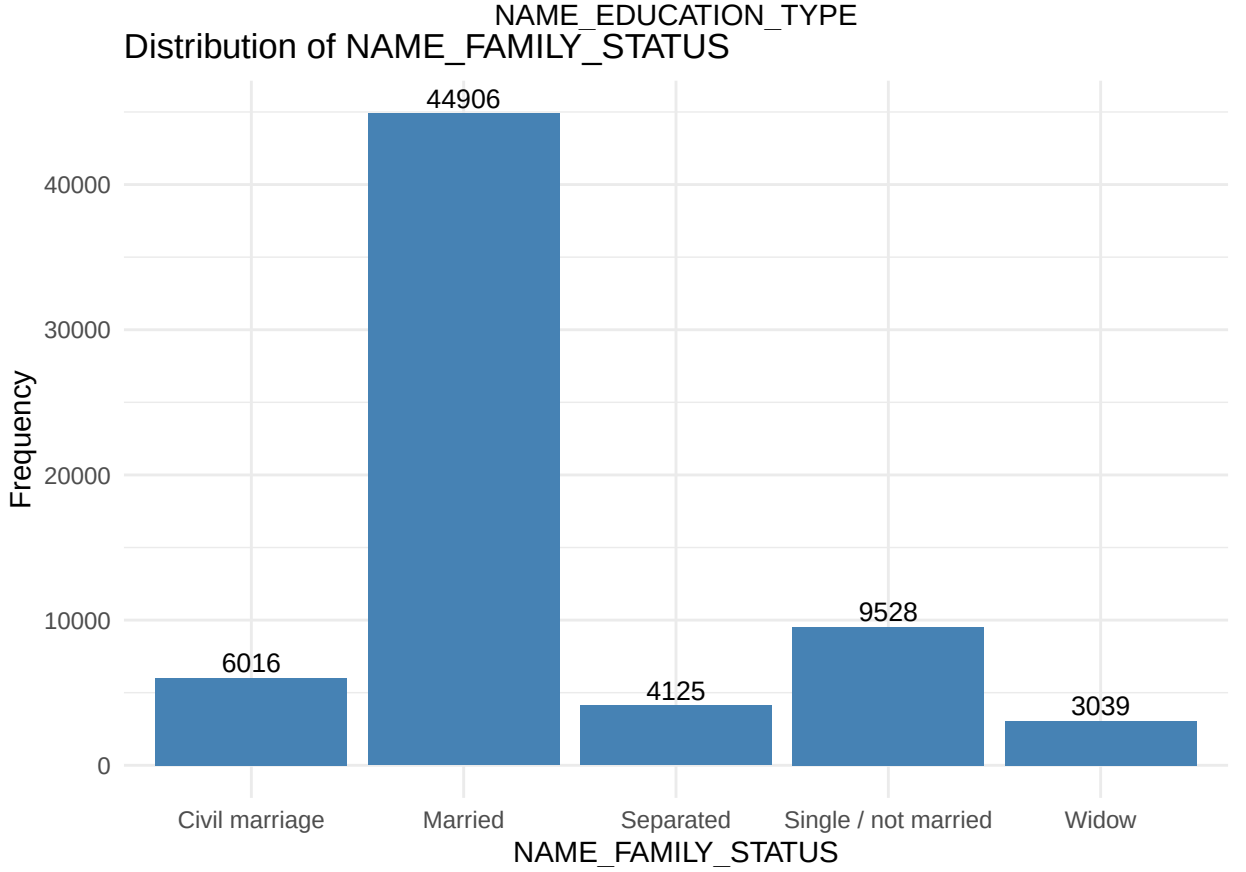

Distribution of CODE_GENDER
43924
23690
40000
30000
20000
10000
0
Frequency
F
M
CODE_GENDER

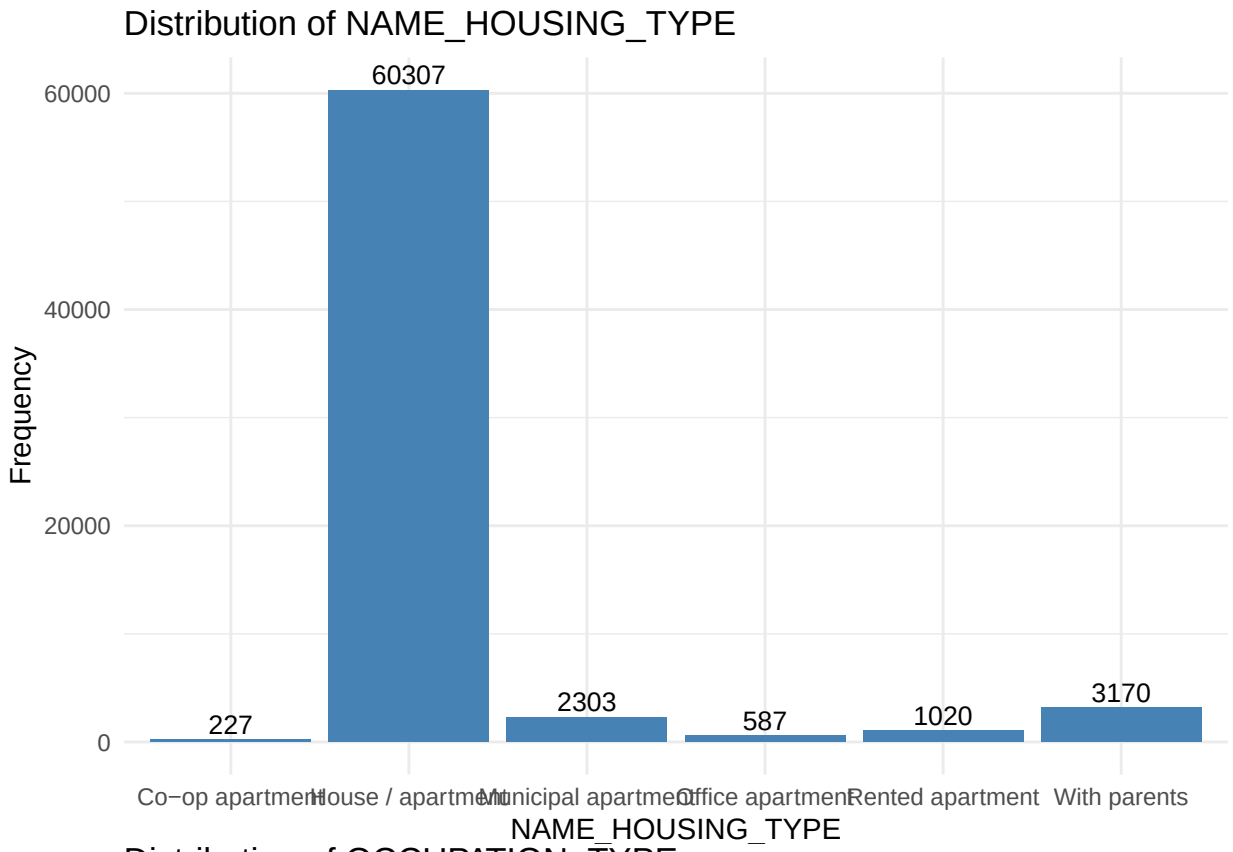

Distribution of FLAG_OWN_CAR
43107
24507
40000
30000
20000
10000
0
Frequency
N
Y
FLAG_OWN_CAR

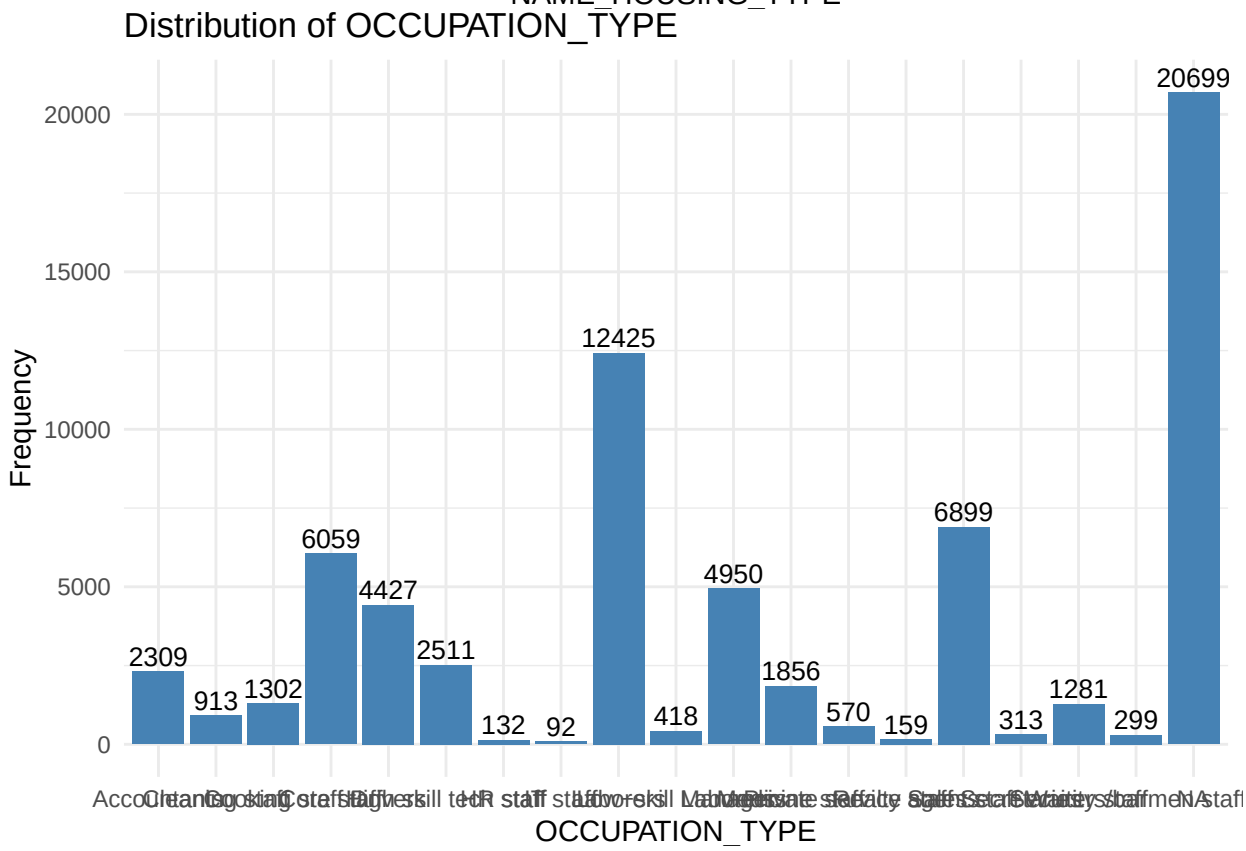

Distribution of FLAG_OWN_REALTY
46524
21090
40000
30000
20000
10000
0
Frequency
N
Y
FLAG_OWN_REALTY

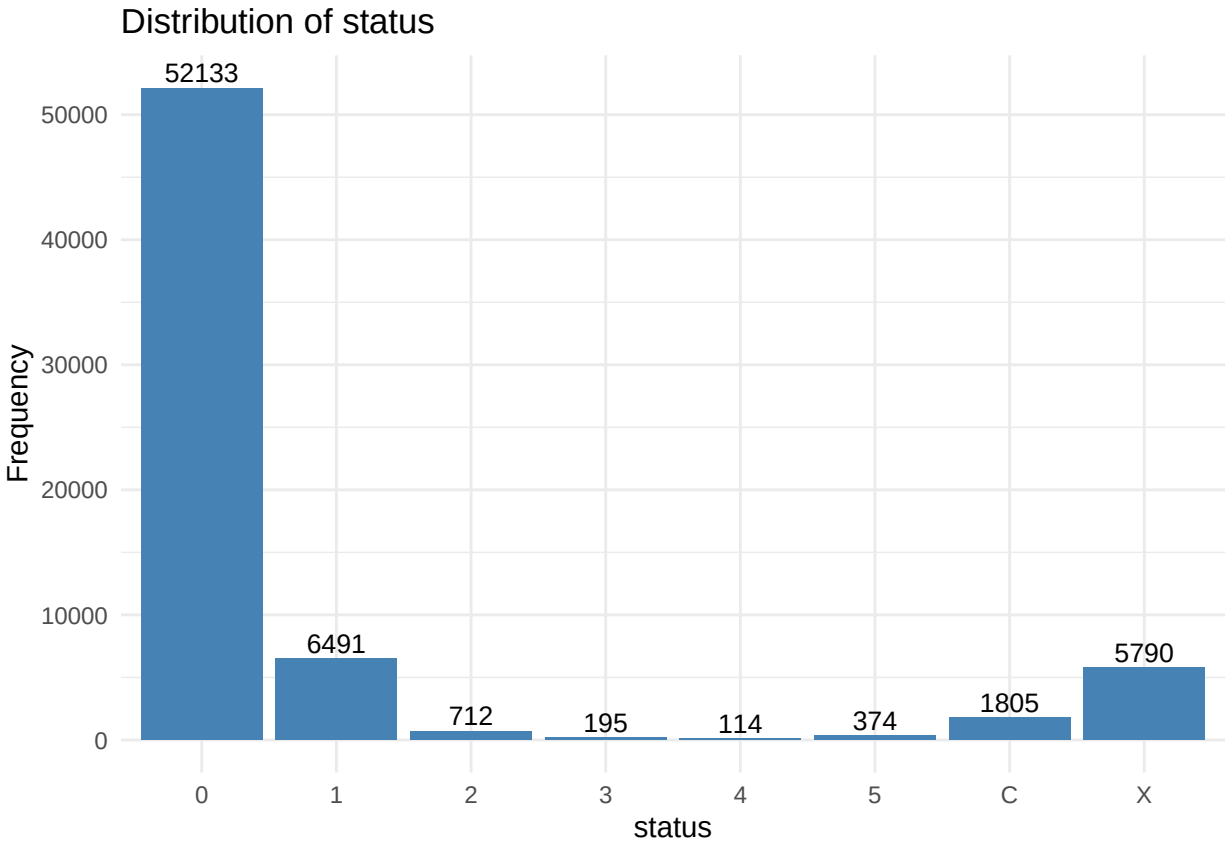

Distribution of NAME_INCOME_TYPE
34944
15640
11982
5044
4
30000
20000
10000
0
Frequency
Commercial associate
Pensioner
State servant
Student
Working
NAME_INCOME_TYPE


Distribution of NAME_EDUCATION_TYPE
Frequency
50000
40000
30000
20000
10000
0
38
16890
2306
716
47664
Academic degree
Higher education
Incomplete higher
Lower secondary
Secondary / secondary speci
NAME_EDUCATION_TYPE


Distribution of NAME_FAMILY_STATUS
Frequency
40000
30000
20000
10000
0
6016
44906
4125
9528
3039
Civil marriage
Married
Separated
Single / not married
Widow
NAME_FAMILY_STATUS


Distribution of NAME_HOUSING_TYPE
Frequency
60000
40000
20000
0
227
60307
2303
587
1020
3170
Co−op apartment
House / apartment
Municipal apartment
Office apartment
Rented apartment
With parents
NAME_HOUSING_TYPE


Distribution of OCCUPATION_TYPE
Frequency
20000
15000
10000
5000
0
2309
913
1302
6059
4427
2511
132
92
12425
418
4950
1856
570
159
6899
313
1281
299
20699
OCCUPATION_TYPE



Summary

What preprocessing needs to be done?

- Drop: ID (not significant), FLAG_MOBIL (only one distinct value)
- Mutate: OCCUPATION_TYPE (-> "unemployed" for rows with positive DAYS_EMPLOYED)
- Label Encode: CODE_GENDER, FLAG_OWN_CAR, FLAG_OWN_REALTY, FLAG_WORK_PHONE, FLAG_PHONE, FLAG_EMAIL
- One-Hot Encode: NAME_INCOME_TYPE, NAME_EDUCATION_TYPE, NAME_FAMILY_STATUS, NAME_HOUSING_TYPE, OCCUPATION_TYPE
- Impute: OCCUPATION_TYPE (the rest of the 8844 values) (possibly, could also be left like this)
- then we check the correlation

We decided to label-encode all columns where there are just two distinct values (to make it less computationally intensive compared to one-hot encoding all of them) and will try to one-hot encode all other columns with > 2 distinct variables.

Other suggestions we might try later on:

- remove outliers:
  - mod-z-score: AMT_INCOME_TOTAL, DAYS_EMPLOYED
  - IQR *3 method: CNT_CHILDREN, CNT_FAM_MEMBERS
- non-numeric columns: might bin several categories into one, makes little sense to keep all the "infrequent" categories by themselves!
  - e.g. academic degree, higher education into one class, lower secondary and secondary into another

For our initial models, we will not do too much data preprocessing to get a benchmark, but we might give this a try later on.

Data Preparation Pipeline

In this first step of data preparation, we only do the necessary steps like encoding or imputing the Occupation Type column. ## Input Variables - Version 1 This is the first version of data prepraration that we used. It is deprecated and we will not use this for the reality check file (as the second version performed better due to the removal of outliers), but we will keep it for completeness.

```
# 1. Dropping Rows, mutate OCCUPATION_TYPE, label encoding
data_clean <- data %>%
  select(-c(ID, FLAG_MOBIL)) %>%

  # Mutate OCCUPATION_TYPE to "unemployed" for positive DAYS_EMPLOYED
  mutate(OCCUPATION_TYPE = ifelse(DAYS_EMPLOYED > 0, "Unemployed", OCCUPATION_TYPE)) %>%

  # changing NAs in the OCCUPATION_TYPE column to "NA"
  mutate(OCCUPATION_TYPE = ifelse(is.na(OCCUPATION_TYPE), "NA", OCCUPATION_TYPE)) %>%

  mutate(
    CODE_GENDER = case_when(
      CODE_GENDER == 'M' ~ 0,
      CODE_GENDER == 'F' ~ 1,
      TRUE ~ NA_real_
    ),
    FLAG_OWN_CAR = case_when(
      FLAG_OWN_CAR == 'N' ~ 0,
      FLAG_OWN_CAR == 'Y' ~ 1,
      TRUE ~ NA_real_
    ),
    FLAG_OWN_REALTY = case_when(
      FLAG_OWN_REALTY == 'N' ~ 0,
      FLAG_OWN_REALTY == 'Y' ~ 1,
      TRUE ~ NA_real_
    )
  )
```

In the next step, we perform the kNN imputation for the NAs in the OCCUPATION_TYPE column. We chose k = 4 because the computer wasn't performant enough for k= 5 but we still wanted the highest amount of neighbors possible.

```
# 2. kNN Imputation of OCCUPATION_TYPE
data_clean <- as.data.frame(lapply(data_clean, as.character))
data_clean <- kNN(data_clean, variable = "OCCUPATION_TYPE", k = 5, imp_var = FALSE)  #
 ↪  Apply kNN imputation (k = 5)
```

In the last step of the preprocessing of the input variables we apply the one-hot encoding to the columns that need encoding and have more than two distinct values.

```
# 3. One-hot encode the specified categorical variables
columns_to_encode <- c("NAME_INCOME_TYPE", "NAME_EDUCATION_TYPE", "NAME_FAMILY_STATUS",
 ↪  "NAME_HOUSING_TYPE", "OCCUPATION_TYPE")

# Create one-hot encoded columns using dummyVars from Caret
dummy_model <- dummyVars(~ ., data = data_clean[, columns_to_encode])
encoded_data <- predict(dummy_model, newdata = data_clean[, columns_to_encode])  # Apply
 ↪  model
```

```r
data_clean <- data_clean %>%
  select(-all_of(columns_to_encode)) %>%  # Remove original columns
  bind_cols(as.data.frame(encoded_data))  # Add one-hot encoded columns
```

Exporting the possible values for One-Hot Encoding in the Reality Check file

We realised, that the reality check secret dataset might not contain at least one instance with every possible value. This would mean that we miss a feature and the mdoel would not run. To mitigate this, we export all possible values for each column here, so we can import them again in the reality check file.

```r
columns_to_encode <- c("NAME_INCOME_TYPE", "NAME_EDUCATION_TYPE", "NAME_FAMILY_STATUS",
 ↪  "NAME_HOUSING_TYPE", "OCCUPATION_TYPE")

# Function to standardize the category name
standardize_category_name <- function(value) {
  # Replace spaces with underscores and non-alphanumeric characters with dots
  gsub("\\s+", "_", gsub("[^A-Za-z0-9_]", ".", value))
}

# Create the encoding categories with standardized names
encoding_categories <- lapply(columns_to_encode, function(col) {
  unique_categories <- unique(secret_data[[col]]) # Extract unique categories from the
 ↪  column
  sapply(unique_categories, standardize_category_name) # Standardize the names
})

# Assign column names to the list
names(encoding_categories) <- columns_to_encode

# Manually add "Unemployed" to OCCUPATION_TYPE
if ("OCCUPATION_TYPE" %in% names(encoding_categories)) {
  encoding_categories[["OCCUPATION_TYPE"]] <- unique(c(
    encoding_categories[["OCCUPATION_TYPE"]],
    standardize_category_name("Unemployed")
  ))
}

# Save the encoding categories to an RDS file
saveRDS(encoding_categories, "encoding_categories2.rds")
```

Edit: We couldn't get this to run, but still wanted to leave it in, as we think it was an important consideration.

Target Variable

Our Target Variable 'status' contains pay-back behavior of the customer. We see that the numerical categories 0-1 are the customers who payed past the due date and character categories 'C' and 'X' means that they do not have an overdue.

This is a chunk of code that we initially used but later didn't anymore (we kept the values as is for a longer time and then performed it later), but still decided to keep it here for completeness.

```r
data_clean <- data_clean %>%
  mutate(
    status = recode(status,
```

```
                    'C' = 0,
                    'X' = 1,
                    '0' = 2,
                    '1' = 3,
                    '2' = 4,
                    '3' = 5,
                    '4' = 6,
                    '5' = 7)
  )
```

Correlation Matrix

To generate a correlation matrix of the dataset data_final, we can follow these steps in R. The correlation matrix shows the relationships between numeric variables in your dataset, which can help identify any strong correlations among the features.

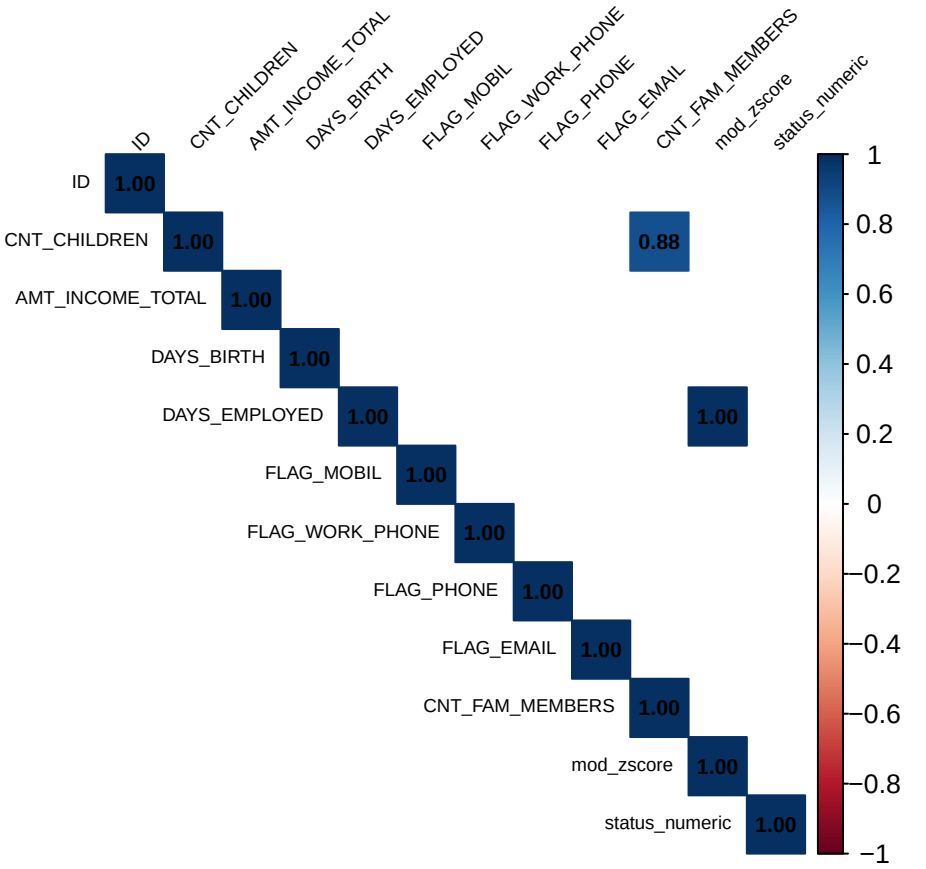
Here, we filter for correlations > 0.7 to see whether we have any strongly correlated features.

Before Encoding

```
correlation_matrix <- cor(data[sapply(data, is.numeric)], use = "complete.obs")
```

```
## Warning in cor(data[sapply(data, is.numeric)], use = "complete.obs"): the
## standard deviation is zero
```

```
# Filter correlations: set values below 0.7 to NA (to hide them in the plot)
filtered_correlation_matrix <- ifelse(abs(correlation_matrix) > 0.7, correlation_matrix,
 ↪  NA)

# Visualize the filtered correlation matrix
corrplot(filtered_correlation_matrix, method = "color", type = "upper",
         tl.col = "black", tl.srt = 45, addCoef.col = "black",
         number.cex = 0.7, tl.cex = 0.6, na.label = " ")
```



As we can see, there is quite a high correlation between CNF_FAM_MEMBERS and CNT_CHILDREN.

After Encoding

```
correlation_matrix <- cor(data[sapply(data, is.numeric)], use = "complete.obs")

# Filter correlations: set values below 0.7 to NA (to hide them in the plot)
filtered_correlation_matrix <- ifelse(abs(correlation_matrix) > 0.7, correlation_matrix,
 ↪  NA)

# Visualize the filtered correlation matrix
corrplot(filtered_correlation_matrix, method = "color", type = "upper",
         tl.col = "black", tl.srt = 45, addCoef.col = "black",
         number.cex = 0.7, tl.cex = 0.6, na.label = " ")
```

We saw that we have correlations greater than 0.7, so we have a closer look at them in a table.

```
threshold <- 0.7
high_corr <- which(abs(correlation_matrix) > threshold & abs(correlation_matrix) < 1,
 ↪  arr.ind = TRUE)

# Format the results into a data frame
correlation_results <- data.frame(
  Variable1 = rownames(correlation_matrix)[high_corr[, 1]],
  Variable2 = colnames(correlation_matrix)[high_corr[, 2]],
  Correlation = correlation_matrix[high_corr]
```

```
)

# Remove duplicate pairs (since correlation matrices are symmetric)
correlation_results <- correlation_results[correlation_results$Variable1 <
 ↪  correlation_results$Variable2, ]

# Display the results
print(correlation_results)
```

```
##       Variable1       Variable2 Correlation
## 3  CNT_CHILDREN CNT_FAM_MEMBERS   0.8784203
## 4 DAYS_EMPLOYED      mod_zscore   1.0000000
```

As we can see, we have an extremely high correlation between the OCCUPATION_TYPEunemployed (which we introduced earlier because it was NA in all those instances + DAYS_EMPLOYED was positive) and NAME_INCOME_TYPEPensioner. We decide to drop this feature we introduced.

```
# Data Preparation
data_final <- data_clean[, !colnames(data_clean) %in% "OCCUPATION_TYPEunemployed"]
```

So we won't have to run the preprocessing respectively the kNN imputation again (computational resources / time), we export this data_final to a csv file, so we could import it from there at a later point again.

```
# Exporting the data_final object to a CSV file
write.csv(data_clean, "data_final_new.csv", row.names = FALSE)
```

Feature Selection / Engineering

We worked with different versions of the dataset and tried the models with a different combination of features as well as different versions and transformations of the feature. For completeness, we include here some of the different versions we tried. ## Changes: AMT_INCOME_TOTAL / DAYS_EMPLOYED We were not satisfied with the initial results of the models created and remembered that there were some extreme outliers in the column AMT_INCOME_TOTAL, which we thought we might have to get rid off. Here we tried the following things:

For the DAYS_EMPLOYED column, which is also quite "distorted" by all the extremely high positive numbers for the unemployed people (e.g. pensioneers), we try setting all the positive entries to 0, for the same reason as the AMT_INCOME_TOTAL so the normalisation will be less "condensed" to the low numbers.

In the consecutive steps, we try different methods of data preprocessing to improve the model's performance.

Main

Outlier Removal

Upon trying the first models, we decided to ultimately remove the outliers we discussed during data exploration to improve the performance. After removing them, we use kNN imputation to set the NAs. In this section, we include the code for this, which we only ran once and then exported to a csv. It includes steps to remove the outliers we identified during the data exploration and sets all the positive values in the DAYS_EMPLOYED column (which represent unemployed people) to 0.

```
data_final <- read.csv("data_final_new.csv", header = TRUE, sep = ",")

# Replace all values > 0 in DAYS_EMPLOYED with 0
data_final <- data_final %>%
```

```r
  mutate(DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED))

# Function to calculate Modified Z-Score
calc_mod_z <- function(column) {
  median_val <- median(column, na.rm = TRUE)
  mad_val <- mad(column, na.rm = TRUE)
  abs(0.6745 * (column - median_val) / mad_val)
}

# Replace outliers (> 3.5 Modified Z-Score) with NA in AMT_INCOME_TOTAL and DAYS_EMPLOYED
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = ifelse(calc_mod_z(AMT_INCOME_TOTAL) > 3.5, NA, AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(calc_mod_z(DAYS_EMPLOYED) > 3.5, NA, DAYS_EMPLOYED)
  )

# Function to replace extreme outliers based on 3*IQR
remove_outliers_iqr <- function(column) {
  q1 <- quantile(column, 0.25, na.rm = TRUE)
  q3 <- quantile(column, 0.75, na.rm = TRUE)
  iqr <- q3 - q1
  lower_bound <- q1 - 3 * iqr
  upper_bound <- q3 + 3 * iqr
  ifelse(column < lower_bound | column > upper_bound, NA, column)
}

# Replace extreme outliers (3*IQR) in CNT_CHILDREN and CNT_FAM_MEMBERS with NA
data_final <- data_final %>%
  mutate(
    CNT_CHILDREN = remove_outliers_iqr(CNT_CHILDREN),
    CNT_FAM_MEMBERS = remove_outliers_iqr(CNT_FAM_MEMBERS)
  )

# View summary of the processed data
summary(data_final)
```

Then in the next step, we ran the imputation using k=5.

```r
data_final_imputed <- kNN(data_final, k = 5)

# Drop the additional attributes added by kNN (which include indicators for imputed
 ↪  values)
data_final_imputed <- data_final_imputed %>% select(-ends_with("_imp"))

# Export the imputed dataset to a CSV file
write.csv(data_final_imputed, "data_outlier_removed_new.csv", row.names = FALSE)

# Print message to confirm export
cat("Imputed dataset exported to 'data_outlier_removed2.csv'")
```

Feature Engineering

This is the final version of the feature engineering that we used for the later steps.

```r
data_final <- read.csv("data_outlier_removed.csv", header = TRUE, sep = ",")

data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )
```

Experiment with CNT_CHILDREN binned

We experimented with using bins to transform the CNT_CHILDREN and CNT_FAMILY_MEMBERS, it did not make a significant difference, so we leave it out, but still include the code here.

```r
data_final <- read.csv("data_outlier_removed.csv", header = TRUE, sep = ",")

data_final <- data_final %>%
  mutate(
    #AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )

# Create bins for CNT_CHILDREN
data_final$CNT_CHILDREN_BIN <- cut(
  data_final$CNT_CHILDREN,
  breaks = c(-Inf, 0, 1, Inf), # Define bin boundaries
  labels = c("No children", "1 child", "2+ children"), # Bin labels
  right = TRUE # Include the right endpoint in the bin
)

# Check the distribution of the bins
print(table(data_final$CNT_CHILDREN_BIN))

# One-hot encode the binned feature
library(caret)

# Create dummy variables
dummies <- dummyVars("~ CNT_CHILDREN_BIN", data = data_final)
data_encoded <- predict(dummies, newdata = data_final)

# Convert to data frame
data_encoded <- as.data.frame(data_encoded)

# Bind the one-hot encoded variables back to the original dataset
data_final <- cbind(data_final, data_encoded)

# Remove the original CNT_CHILDREN column and the binned CNT_CHILDREN_BIN column
data_final$CNT_CHILDREN <- NULL
data_final$CNT_CHILDREN_BIN <- NULL
```

Earlier Main

This is the earlier main version we used before we implemented the removal of the outliers.

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")

data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )
```

Experiment 1

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    CNT_CHILDREN = log1p(CNT_CHILDREN),
    CNT_FAM_MEMBERS = log1p(CNT_FAM_MEMBERS),
    AGE = -DAYS_BIRTH / 365
  ) %>%
  select(-DAYS_BIRTH, -CNT_CHILDREN)
```

Experiment 2

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    CNT_CHILDREN = log1p(CNT_CHILDREN),
    CNT_FAM_MEMBERS = log1p(CNT_FAM_MEMBERS),
    AGE = -DAYS_BIRTH / 365,
    AGE_BIN = cut(
      AGE,
      breaks = quantile(AGE, probs = seq(0, 1, by = 0.2), na.rm = TRUE),
      labels = c("lowest", "low", "medium", "high", "highest"),
      include.lowest = TRUE
    )
  ) %>%
  select(-DAYS_BIRTH, -CNT_CHILDREN)
```

Experiment 3

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    CNT_CHILDREN = log1p(CNT_CHILDREN),
    CNT_FAM_MEMBERS = log1p(CNT_FAM_MEMBERS),
```

```
    AGE = -DAYS_BIRTH / 365,
    EMPLOYMENT_YEARS = -DAYS_EMPLOYED / 365,
  ) %>%
  select(-DAYS_BIRTH, -CNT_CHILDREN, -DAYS_EMPLOYED)
```

Experiment 4

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),  # Adjusting
     ↪  DAYS_EMPLOYED to replace positive values
    CNT_CHILDREN = log1p(CNT_CHILDREN),
    CNT_FAM_MEMBERS = log1p(CNT_FAM_MEMBERS),
    AGE = -DAYS_BIRTH / 365,
    EMPLOYMENT_YEARS = -DAYS_EMPLOYED / 365,
    IS_EMPLOYED = ifelse(EMPLOYMENT_YEARS > 0, 1, 0)  # Create IS_EMPLOYED binary feature
  ) %>%
  select(-DAYS_BIRTH, -CNT_CHILDREN, -DAYS_EMPLOYED)  # Remove unnecessary columns
```

Experiment 5

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),  # Adjusting
     ↪  DAYS_EMPLOYED to replace positive values
    CNT_CHILDREN = log1p(CNT_CHILDREN),
    CNT_FAM_MEMBERS = log1p(CNT_FAM_MEMBERS),
    AGE = -DAYS_BIRTH / 365,
    EMPLOYMENT_YEARS = -DAYS_EMPLOYED / 365
  ) %>%
  select(-DAYS_BIRTH, -CNT_CHILDREN, -DAYS_EMPLOYED)  # Remove unnecessary columns
```

Experiment 6

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    AGE = -DAYS_BIRTH / 365,
    CNT_CHILDREN = ifelse(CNT_CHILDREN > 5, 5, CNT_CHILDREN),
    CNT_CHILDREN = log1p(CNT_CHILDREN),
  ) %>%
  select(-DAYS_BIRTH, -CNT_FAM_MEMBERS)
```

Experiment 7

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, NA, DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH),
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    CNT_CHILDREN = ifelse(CNT_CHILDREN > 5, 5, CNT_CHILDREN),
    CNT_CHILDREN = log1p(CNT_CHILDREN),
  ) %>%
  select(-CNT_FAM_MEMBERS)
```

Experiment 8

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED == 365243, 0, DAYS_EMPLOYED),
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH),
    #DAYS_BIRTH = log1p(DAYS_BIRTH),
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    CNT_CHILDREN = ifelse(CNT_CHILDREN > 5, 5, CNT_CHILDREN),
    CNT_CHILDREN = log1p(CNT_CHILDREN)
  ) %>%
  select(-CNT_FAM_MEMBERS)

# Remove NA values to prepare for Box-Cox transformation
days_employed_numeric <- data_final$DAYS_EMPLOYED
days_employed_numeric_no_na <- days_employed_numeric[!is.na(days_employed_numeric)]

# Perform Box-Cox transformation
lambda <- car::powerTransform(days_employed_numeric_no_na + 1)$lambda  # Determine
 ↪  optimal lambda
days_employed_transformed <- bcPower(days_employed_numeric_no_na + 1, lambda = lambda)

# Replace original values in data_final
data_final$DAYS_EMPLOYED_BOXCOX <- NA  # Initialize with NA
data_final$DAYS_EMPLOYED_BOXCOX[!is.na(data_final$DAYS_EMPLOYED)] <-
 ↪  days_employed_transformed

# Optionally, drop the DAYS_EMPLOYED_BOXCOX column if it's no longer needed
data_final <- data_final %>%
  select(-DAYS_EMPLOYED_BOXCOX)
```

Experiment 9

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
# Create numeric vectors without NA values for DAYS_EMPLOYED, DAYS_BIRTH, etc.
days_employed_numeric_no_na <- na.omit(as.numeric(abs(data_final$DAYS_EMPLOYED)))
days_birth_numeric_no_na <- na.omit(as.numeric(abs(data_final$DAYS_BIRTH)))
amt_income_total_no_na <- na.omit(data_final$AMT_INCOME_TOTAL + 1)
cnt_children_no_na <- na.omit(data_final$CNT_CHILDREN + 1)

# Calculate optimal lambdas using powerTransform from car package
lambda_days_employed <- car::powerTransform(days_employed_numeric_no_na + 1)$lambda
lambda_days_birth <- car::powerTransform(days_birth_numeric_no_na + 1)$lambda
lambda_amt_income_total <- car::powerTransform(amt_income_total_no_na)$lambda
lambda_cnt_children <- car::powerTransform(cnt_children_no_na)$lambda

# Apply BoxCox transformations using the computed lambdas
data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED == 365243, 0, DAYS_EMPLOYED),
    DAYS_EMPLOYED = BoxCox(DAYS_EMPLOYED + 1, lambda = lambda_days_employed),
    DAYS_BIRTH = abs(DAYS_BIRTH),
    DAYS_BIRTH = BoxCox(DAYS_BIRTH + 1, lambda = lambda_days_birth),
    AMT_INCOME_TOTAL = BoxCox(AMT_INCOME_TOTAL + 1, lambda = lambda_amt_income_total),
    CNT_CHILDREN = ifelse(CNT_CHILDREN > 5, 5, CNT_CHILDREN),
    CNT_CHILDREN = BoxCox(CNT_CHILDREN + 1, lambda = lambda_cnt_children)
  ) %>%
  select(-CNT_FAM_MEMBERS)
```

Experiment 10

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED == 365243, 0, DAYS_EMPLOYED),
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
    YEARS_EMPLOYED = DAYS_EMPLOYED / 365,
    DAYS_BIRTH = abs(DAYS_BIRTH),
    AGE = DAYS_BIRTH / 365,
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    CNT_CHILDREN = ifelse(CNT_CHILDREN > 5, 5, CNT_CHILDREN),
    CNT_CHILDREN = log1p(CNT_CHILDREN)
  ) %>%
  select(-CNT_FAM_MEMBERS, -DAYS_EMPLOYED, -DAYS_BIRTH)
```

Experiment 11

```r
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
```

```
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED == 365243, 0, DAYS_EMPLOYED),
    DAYS_EMPLOYED = as.numeric(DAYS_EMPLOYED),
    YEARS_EMPLOYED = DAYS_EMPLOYED / 365,
    DAYS_BIRTH = abs(DAYS_BIRTH),
    AGE = DAYS_BIRTH / 365,
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    CNT_CHILDREN = ifelse(CNT_CHILDREN > 5, 5, CNT_CHILDREN),
    CNT_CHILDREN = log1p(CNT_CHILDREN)
  ) %>%
  select(-CNT_FAM_MEMBERS, -DAYS_EMPLOYED, -DAYS_BIRTH)

days_employed_numeric <- data_final$YEARS_EMPLOYED
days_employed_numeric_no_na <- days_employed_numeric[!is.na(days_employed_numeric)]
age_numeric <- data_final$AGE
age_numeric_no_na <- age_numeric[!is.na(age_numeric)]

# Perform the Box-Cox transformation
lambda <- car::powerTransform(days_employed_numeric_no_na + 1)$lambda
days_employed_transformed <- bcPower(days_employed_numeric_no_na + 1, lambda = lambda)

# Update the original YEARS_EMPLOYED column with transformed values
data_final$YEARS_EMPLOYED[!is.na(data_final$YEARS_EMPLOYED)] <- days_employed_transformed

# Perform the Box-Cox transformation
lambda <- car::powerTransform(age_numeric_no_na + 1)$lambda
age_transformed <- bcPower(age_numeric_no_na + 1, lambda = lambda)

# Update the original YEARS_EMPLOYED column with transformed values
data_final$AGE[!is.na(data_final$AGE)] <- age_transformed
```

Experiment 12

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    #AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )
```

Experiment 13

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
# Compute the 99th percentile for DAYS_EMPLOYED
percentile_99 <- quantile(data_final$DAYS_EMPLOYED[data_final$DAYS_EMPLOYED > 0], 0.99,
 ↪  na.rm = TRUE)

# Update the pipeline to include the 99th percentile capping
data_final <- data_final %>%
  mutate(
```

```
    # Transform DAYS_EMPLOYED to 0 if > 0
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, 0, DAYS_EMPLOYED),
    # Convert DAYS_EMPLOYED and DAYS_BIRTH to absolute values
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )
summary(data_final$DAYS_EMPLOYED)
percentile_99 <- quantile(data_final$DAYS_EMPLOYED[data_final$DAYS_EMPLOYED > 0], 0.99,
 ↪  na.rm = TRUE)
data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > percentile_99, percentile_99, DAYS_EMPLOYED)
  )
summary(data_final$DAYS_EMPLOYED)


# Create a histogram to visualize the distribution
ggplot(data_final, aes(x = DAYS_EMPLOYED)) +
  geom_histogram(binwidth = 500, fill = "blue", color = "black", alpha = 0.7) +
  labs(title = "Distribution of DAYS_EMPLOYED",
       x = "DAYS_EMPLOYED",
       y = "Count") +
  theme_minimal()

# Create a density plot
ggplot(data_final, aes(DAYS_EMPLOYED)) +
  geom_density(fill = "green", alpha = 0.5) +
  labs(title = "Density Plot of DAYS_EMPLOYED",
       x = "DAYS_EMPLOYED",
       y = "Density") +
  theme_minimal()

# Boxplot for a quick look at outliers
ggplot(data_final, aes(y = DAYS_EMPLOYED)) +
  geom_boxplot(fill = "orange", color = "black", alpha = 0.7) +
  labs(title = "Boxplot of DAYS_EMPLOYED",
       y = "DAYS_EMPLOYED") +
  theme_minimal()
```

Experiment 14

```
data_final <- read.csv("data_final2.csv", header = TRUE, sep = ",")
data_final <- data_final %>%
  mutate(
    AMT_INCOME_TOTAL = log1p(AMT_INCOME_TOTAL),
    DAYS_EMPLOYED = ifelse(DAYS_EMPLOYED > 0, NA, DAYS_EMPLOYED),
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )
```

Train - Test Split

Because we found an imbalance in our target variable when we looked at the distribution earlier, we will be performing a stratified train test split to make sure the less common target classes will be equally represented in both the train and the val / test set.

We perform a 70-15-15 split. ## Stratified Split

```
# Perform stratified split (70% training, 15% validation, 15% test)
train_indices <- createDataPartition(data_final$status, p = 0.7, list = FALSE)
train_data <- data_final[train_indices, ]
temp_data <- data_final[-train_indices, ]

# Split temporary data into validation and test sets
val_indices <- createDataPartition(temp_data$status, p = 0.5, list = FALSE)
val_data <- temp_data[val_indices, ]
test_data <- temp_data[-val_indices, ]

# Verify split sizes
cat("Training set size:", nrow(train_data), "\n")
```

```
## Training set size: 47333
```

```
cat("Validation set size:", nrow(val_data), "\n")
```

```
## Validation set size: 10143
```

```
cat("Test set size:", nrow(test_data), "\n")
```

```
## Test set size: 10138
```

Processing after Train-Test-Split

In some later versions, we also tried using BoxCox Transformation on the AMT_INCOME_TOTAL. It didn't make a significant difference, so we did not include it in the final pipeline-

```
bc_params <- BoxCoxTrans(train_data$AMT_INCOME_TOTAL)
train_data$AMT_INCOME_TOTAL <- predict(bc_params, train_data$AMT_INCOME_TOTAL)
val_data$AMT_INCOME_TOTAL <- predict(bc_params, val_data$AMT_INCOME_TOTAL)
test_data$AMT_INCOME_TOTAL <- predict(bc_params, test_data$AMT_INCOME_TOTAL)
```

Trying Robust scaling, because we have some extreme outliers in the AMT_INCOME_TOTAL column:

```
median_income <- median(train_data$AMT_INCOME_TOTAL, na.rm = TRUE)
iqr_income <- IQR(train_data$AMT_INCOME_TOTAL, na.rm = TRUE)

train_data$AMT_INCOME_TOTAL <- (train_data$AMT_INCOME_TOTAL - median_income) / iqr_income
val_data$AMT_INCOME_TOTAL <- (val_data$AMT_INCOME_TOTAL - median_income) / iqr_income
test_data$AMT_INCOME_TOTAL <- (test_data$AMT_INCOME_TOTAL - median_income) / iqr_income
```

Normalization

In the next step, we normalize the data in this project to ensure all features are on a similar scale (between 0 to 1), which is important for neural networks. Neural networks perform better when input features are scaled consistently because it helps the optimization process converge faster because NNs use gradient-based optimization methods. Without normalization, features with larger ranges could dominate the learning process,

leading to biased results (in this case for example AMT_INCOME_TOTAL or DAYS_EMPLOYED).

This is done after the train-test-split so we can make sure that the normalization parameters, such as the minimum and maximum values, are calculated from the training data only.

Find Normalization Parameters

In this step, we find the normalization parameters from the test set.

```
columns_to_normalize <- c("AMT_INCOME_TOTAL", "CNT_FAM_MEMBERS", "DAYS_EMPLOYED",
 ↪  "CNT_CHILDREN", "DAYS_BIRTH")

# Compute min and max values for normalization from train_data
train_min <- sapply(train_data[columns_to_normalize], min, na.rm = TRUE)
train_max <- sapply(train_data[columns_to_normalize], max, na.rm = TRUE)
```

Here we define a function to export our train min and max to a csv so we can import them in the reality check file.

```
# Save train_min and train_max as a data frame
normalization_params <- data.frame(
  column = names(train_min),
  min_val = train_min,
  max_val = train_max
)

# Export normalization parameters to a CSV file
write.csv(normalization_params, "normalization_params2.csv", row.names = FALSE)
```

Apply Normalization to Val and Test

Here we apply the normalization with the parameters of the train set to the validation and test set.

```
normalize_with_params <- function(x, min_val, max_val) {
  return((x - min_val) / (max_val - min_val))
}
```

```
normalize_columns_with_train_params <- function(dataset, columns_to_normalize, train_min,
 ↪  train_max) {
  dataset[columns_to_normalize] <- Map(function(col, min_val, max_val) {
    normalize_with_params(dataset[[col]], min_val, max_val)
  }, columns_to_normalize, train_min, train_max)
  return(dataset)
}

train_data <- normalize_columns_with_train_params(train_data, columns_to_normalize,
 ↪  train_min, train_max)
val_data <- normalize_columns_with_train_params(val_data, columns_to_normalize,
 ↪  train_min, train_max)
test_data <- normalize_columns_with_train_params(test_data, columns_to_normalize,
 ↪  train_min, train_max)
```

```
summary(train_data[columns_to_normalize])
```

```
##  AMT_INCOME_TOTAL CNT_FAM_MEMBERS  DAYS_EMPLOYED      CNT_CHILDREN
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.00000   Min.   :0.0000
```

```
##  1st Qu.:0.4891   1st Qu.:0.2000   1st Qu.:0.09334   1st Qu.:0.0000
##  Median :0.6044   Median :0.2000   Median :0.19370   Median :0.0000
##  Mean   :0.6034   Mean   :0.2345   Mean   :0.23100   Mean   :0.1047
##  3rd Qu.:0.7267   3rd Qu.:0.4000   3rd Qu.:0.32586   3rd Qu.:0.2500
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.00000   Max.   :1.0000
##    DAYS_BIRTH
##  Min.   :0.0000
##  1st Qu.:0.2745
##  Median :0.4573
##  Mean   :0.4767
##  3rd Qu.:0.6770
##  Max.   :1.0000
```

```
summary(val_data[columns_to_normalize])
```

```
##  AMT_INCOME_TOTAL CNT_FAM_MEMBERS DAYS_EMPLOYED      CNT_CHILDREN
##  Min.   :0.0000   Min.   :0.000   Min.   :0.00000   Min.   :0.0000
##  1st Qu.:0.4891   1st Qu.:0.200   1st Qu.:0.09129   1st Qu.:0.0000
##  Median :0.6044   Median :0.200   Median :0.19370   Median :0.0000
##  Mean   :0.6051   Mean   :0.233   Mean   :0.23262   Mean   :0.1033
##  3rd Qu.:0.7267   3rd Qu.:0.400   3rd Qu.:0.32586   3rd Qu.:0.2500
##  Max.   :0.9969   Max.   :1.000   Max.   :0.99707   Max.   :1.0000
##    DAYS_BIRTH
##  Min.   :0.0133
##  1st Qu.:0.2796
##  Median :0.4607
##  Mean   :0.4780
##  3rd Qu.:0.6746
##  Max.   :0.9952
```

```
summary(test_data[columns_to_normalize])
```

```
##  AMT_INCOME_TOTAL   CNT_FAM_MEMBERS  DAYS_EMPLOYED        CNT_CHILDREN
##  Min.   :-0.01162   Min.   :0.0000   Min.   :0.0003901   Min.   :0.0000
##  1st Qu.: 0.48911   1st Qu.:0.2000   1st Qu.:0.0909490   1st Qu.:0.0000
##  Median : 0.60443   Median :0.2000   Median :0.1936994   Median :0.0000
##  Mean   : 0.60494   Mean   :0.2359   Mean   :0.2318605   Mean   :0.1068
##  3rd Qu.: 0.72667   3rd Qu.:0.4000   3rd Qu.:0.3280747   3rd Qu.:0.2500
##  Max.   : 0.99690   Max.   :1.0000   Max.   :0.9990247   Max.   :1.0000
##    DAYS_BIRTH
##  Min.   :0.01607
##  1st Qu.:0.27357
##  Median :0.46041
##  Mean   :0.47622
##  3rd Qu.:0.67588
##  Max.   :1.00238
```

Data to Matrix Conversion

Input Variables

We can simply transform the dataset to a matrix.

```
x_train <- train_data %>% select(-all_of("status")) %>% as.matrix()
x_val <- val_data %>% select(-all_of("status")) %>% as.matrix()
```

```
x_test <- test_data %>% select(-all_of("status")) %>% as.matrix()
```

Target Variables

We One-Hot encode the target variable using the fastDummies. Then we transform the datasets to matrizes for the neural networks to use. First cell is executed for normal without SMOTE, second for SMOTE.

```
create_one_hot_matrix_fastDummies <- function(data, column_name) {
  one_hot_encoded <- dummy_cols(data, select_columns = column_name, remove_first_dummy =
 ↪  FALSE, remove_selected_columns = TRUE)

  dummy_columns <- grep(paste0(column_name, "_"), colnames(one_hot_encoded), value =
 ↪  TRUE)
  one_hot_matrix <- as.matrix(one_hot_encoded[, dummy_columns])

  return(one_hot_matrix)
}

# Apply the function to the datasets
y_train <- create_one_hot_matrix_fastDummies(train_data, "status")
y_val <- create_one_hot_matrix_fastDummies(val_data, "status")
y_test <- create_one_hot_matrix_fastDummies(test_data, "status")

# Check the results
cat("One-hot encoded matrices created using fastDummies:")
```

```
## One-hot encoded matrices created using fastDummies:
```

```
print(dim(y_train))
```

```
## [1] 47333     8
```

```
print(dim(y_val))
```

```
## [1] 10143     8
```

```
print(dim(y_test))
```

```
## [1] 10138     8
```

SMOTE

In some models, we also used or tried a rebalanced dataset that we generated using SMOTE with the smotefamily package and the code below. The models were not besser with SMOTE, so we left it out for the final model.

```
label_encoder <- train_data$status %>%
  factor() %>%
  as.integer() - 1

# Apply encoding to y_train, y_val, y_test
y_train <- as.integer(factor(train_data$status)) - 1
y_val <- as.integer(factor(val_data$status, levels = levels(factor(train_data$status))))
 ↪  - 1
y_test <- as.integer(factor(test_data$status, levels =
 ↪  levels(factor(train_data$status)))) - 1
```

```r
# Ensure that x_train is a data frame
x_train <- as.data.frame(x_train)

# Ensure that y_train is a factor
y_train <- as.factor(y_train)

# Apply SMOTE
smote_train <- SMOTE(X = x_train, target = y_train, K = 5, dup_size = 10)

# Extract the resampled data
smote_resampled_data <- smote_train$data

# Separate features and target
# Drop the 'class' column from smote_features
smote_features <- smote_resampled_data[, !names(smote_resampled_data) %in% 'class']
smote_target <- smote_resampled_data$class

# Check the new class distribution
table(smote_target)
```

```r
train_data$status <- as.numeric(as.factor(train_data$status)) - 1

# Check the encoded target variable
print(unique(train_data$status))

smote_features <- as.matrix(smote_features)

str(smote_features)
```

Applying SMOTE to train_data After the target needed to be label encoded for the SMOTE, we now One-Hot-encode the label. We also need to do it for y_val (so they have the same shape), because we don't execute the chunk further above where it would be done for y_val.

```r
# One-hot encode the target variable
smote_target <- to_categorical(smote_target)
y_train <- to_categorical(y_train, num_classes = 8)
y_val <- to_categorical(y_val, num_classes = 8)
y_test <- to_categorical(y_test, num_classes = 8)
# Check the structure of the encoded target
str(smote_target)
```

Function to save models

Here we define a function to log our models to the model log file so we can later find the best model.

```r
save_model_details <- function(try_number, history, model_description,
 ↪  data_preprocessing, train_acc, val_acc, test_acc, train_loss, val_loss, test_loss,
 ↪  output_file = "all_models_details.txt") {
  # Open file in append mode
  con <- file(output_file, open = "a")
```

```
  # Write content to the file
  writeLines(c(
    "--------------------",
    paste0("Try Number: ", try_number),
    "",
    "Accuracy Metrics:",
    paste0("Train Accuracy: ", train_acc),
    paste0("Validation Accuracy: ", val_acc),
    paste0("Test Accuracy: ", test_acc),
    "---",
    "Loss Metrics:",
    paste0("Train Loss: ", train_loss),
    paste0("Validation Loss: ", val_loss),
    paste0("Test Loss: ", test_loss),
    "",
    "Model Description:",
    model_description,
    "Data Preprocessing:",
    data_preprocessing,
    "--------------------\n"
  ), con)

  # Close the connection
  close(con)
}
```

The results are saved in the file all_models_details, where you will be able to follow through some of the steps we did. We only logged promising models. From almost all models found in the log, there are also png files in the model_outputs folder that contain the learning curves of the models during training.

Example Usage of Function

```
results <- model %>% evaluate(x_test, y_test)
test_loss <- results[[1]]  # First value is the loss
test_accuracy <- results[[2]]  # Second value is the accuracy

model %>% evaluate(x_val, y_val)
model %>% evaluate(x_train, y_train)

save_model_details(
  try_number = 48,
  history = history,
  model_description = "4 relu layers: 1024(0.005 l2
   ↪  regularizer)/0.25/512/0.15/512/0.1/128; softmax activation function, nadam
   ↪  optimizer, loss_categorigal_crossentropy; 500 epochs 256 batch size CORRECTED
   ↪  NORMALIZATION!!!!
  adjusted class weights *0.5 (corrected!)
  learning rate 0.001
  callback_reduce_lr <- callback_reduce_lr_on_plateau(
  monitor = val_loss,   # Monitor validation loss
  factor = 0.5,           # Reduce learning rate by this factor
  patience = 10,          # Number of epochs with no improvement before reducing LR
```

```
  min_lr = 1e-6           # Minimum learning rate to prevent it from becoming too small
)
early stopping! stopped after 93 epochs",
  data_preprocessing = "
with outliers removed prior to this!!!

data_final <- data_final %>%
  mutate(
    #DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    AGE_YEARS = as.integer(abs(DAYS_BIRTH) / 365.25),
    EMPLOYED_YEARS = as.integer(DAYS_EMPLOYED / 365.25)
  ) %>%
  select(-DAYS_BIRTH, -DAYS_EMPLOYED)

  with imputed days employed
",
  train_acc = max(history$metrics$accuracy),
  val_acc = max(history$metrics$val_accuracy),
  test_acc = test_accuracy,
  train_loss = min(history$metrics$loss),
  val_loss = min(history$metrics$val_loss),
  test_loss = test_loss
)
```

Model Creation

Benchmark Model

We start by creating a relatively simple model to get a benchmark to compare against. ### Model Definition

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 8, activation = "softmax")
```

Model Compilation

```
model %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = list("accuracy")
)
```

Model Training

```
history <- model %>% fit(
  x = x_train, y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 500,                # Number of epochs
  batch_size = 32,            # Batch size
```

```
  verbose = 1                 # Verbosity level
)

# Evaluate the model on the test set
model %>% evaluate(x_test, y_test, verbose = 2)
```

The performance of our initial benchmark model is 0.77 accuracy on the test set and 0.79 test loss. From the plots of the accuracy and val_accuracy as well as loss and val_loss during training we learn the following things:

- Accuracy:
  - appear to converge to a value around 0.78 or 0.79 for both train and val -> this is close to the percentage of the largest class, meaning that it likely just predicts that class for most instances
  - flat curves -> does not learn well, quickly reaches plateau -> could be underfitting, model is probably not complex enough
- Loss:
  - training loss is steadily decreasing
  - validation loss is very chaotic / noisy and inconsistant

Next steps:

- Increase Model Complexity: more layers/more neurons to increase the capacity of model
- Regularization Techniques: even though we already included 2 dropout layers.
- Learning Rate Adjustment

Most likely, this model just learns to predict the majority class.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = 'relu', input_shape = ncol(x_train)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.3) %>%  # Regularization to prevent overfitting
  layer_dense(units = 64, activation = 'relu') %>%
  layer_batch_normalization() %>%  # Batch normalization again for better convergence
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 8, activation = 'softmax')  # Output layer (assuming 8 classes)

# Model Compilation
model %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.001),  # Lower learning rate for smoother
   ↪  convergence
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

# Model Training
history <- model %>% fit(
  x = x_train, y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 300,               # Lower epochs initially to avoid overfitting
  batch_size = 32,
  callbacks = list(callback_early_stopping(patience = 20),  # Early stopping to prevent
   ↪  overfitting
  callback_reduce_lr_on_plateau(patience = 10, factor = 0.5)),  # Reduce learning rate if
   ↪  the model stops improving
```

```
  verbose = 2
)
```

```
# Evaluate the model on the test set
model %>% evaluate(x_test, y_test, verbose = 2)
```

This model has a test accuracy 0f 0.77 and loss of 0.73. Judging from the plots, this is slightly better, but nowhere where we'd like to be. In the next step, we attempt to make the neural net wider, meaning we will increase the units.

Model 2: increased width, early stopping, reducing learning rate

This is the next model we tried in an increased width. We also implement early stopping and learning rate reduction.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = 'relu', input_shape = ncol(x_train)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 8, activation = 'softmax')  # Output layer

# Model Compilation
model %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.001),
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

# Model Training
history <- model %>% fit(
  x = x_train, y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 300,
  batch_size = 32,
  callbacks = list(callback_early_stopping(patience = 20),  # Early stopping
  callback_reduce_lr_on_plateau(patience = 10, factor = 0.5)),
  verbose = 1
)

# Evaluate the model on the test set
model %>% evaluate(x_test, y_test, verbose = 2)
```

This is not a satisfactory improvement, so we try further.

Experimenting with Learning Rate Decay

We also try a manual learing rate scheduler.

```
learning_rate_schedule <- learning_rate_schedule_exponential_decay(
  initial_learning_rate = 0.01, # Start with a higher learning rate
```

```
  decay_steps = 50,              # Decay the learning rate every 50 steps
  decay_rate = 0.9,              # Multiply the learning rate by 0.9 every 50 steps
  staircase = TRUE               # Decay happens in discrete steps
)

# Updated Optimizer with Decayed Learning Rate
model %>% compile(
  optimizer = optimizer_adam(learning_rate = learning_rate_schedule),
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

# Model Training
history <- model %>% fit(
  x = x_train, y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 300,
  batch_size = 64,
  callbacks = list(
    callback_early_stopping(patience = 20)
  ),
  verbose = 1
)
```

Not better.

Model 3: more complex

We try a more complex model, with a bit lower dropout rate and use the categorical loss for the measurement of metric.

```
model <- keras_model_sequential() %>%
    layer_dense(units = 256, activation = "relu", input_shape = ncol(x_train)) %>%
    layer_dropout(rate=0.25) %>%
    layer_batch_normalization() %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dropout(rate=0.15) %>%
    layer_batch_normalization() %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dropout(rate=0.10) %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dense(units = 8, activation = "softmax")


opti <- optimizer_nadam()
loss <- loss_categorical_crossentropy()

model %>% compile(
  optimizer = opti,
  loss = loss,
  metrics = c(metric_categorical_accuracy, "accuracy")
)

history <- model %>% fit(
```

```
  x_train,
  y_train,
  epochs = 250,
  batch_size = 128,
  validation_data = list(x_val, y_val),
  verbose = 1
)
```

In the next step, we added the learning rate that we found with the grid search in the grid search passage furhter below. We added it here because this was our best model and we further refined it with the learning rate.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 2048, activation = "relu", input_shape = 52, kernel_regularizer =
   ↪  regularizer_l2(l = 0.03)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate=0.3) %>%
  layer_dense(units = 1024, activation = "relu", kernel_regularizer = regularizer_l2(l =
   ↪  0.01)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate=0.30) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate=0.20) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 8, activation = "softmax")

opti <- optimizer_nadam(learning_rate = 0.0001)
loss <- loss_categorical_crossentropy()

model %>% compile(
  optimizer = opti,
  loss = loss,
  metrics = c(metric_categorical_accuracy, "accuracy")
)

history <- model %>% fit(
  x_train,
  y_train,
  epochs = 500,
  batch_size = 256,
  validation_data = list(x_val, y_val),
  verbose = 1
)
```

We also tried to use Leaky ReLU in the layers.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 256, input_shape = 51, kernel_regularizer = regularizer_l2(l =
   ↪  0.001)) %>%
  layer_activation_leaky_relu(alpha = 0.01) %>%  # Use Leaky ReLU activation with
   ↪  alpha=0.01
  layer_dropout(rate = 0.1) %>%
  layer_batch_normalization() %>%
```

```
  layer_dense(units = 256, kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_activation_leaky_relu(alpha = 0.01) %>%  # Use Leaky ReLU activation with
   ↪  alpha=0.01
  layer_dropout(rate = 0.1) %>%
  layer_batch_normalization() %>%

  layer_dense(units = 256, kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_activation_leaky_relu(alpha = 0.01) %>%  # Use Leaky ReLU activation with
   ↪  alpha=0.01
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.1) %>%

  layer_dense(units = 256) %>%
  layer_activation_leaky_relu(alpha = 0.01) %>%  # Use Leaky ReLU activation with
   ↪  alpha=0.01
  layer_dense(units = 8, activation = "softmax")

opti <- optimizer_nadam(learning_rate = 0.0001)
loss <- loss_categorical_crossentropy()

model %>% compile(
  optimizer = opti,
  loss = loss,
  metrics = c(metric_categorical_accuracy, "accuracy")
)

history <- model %>% fit(
  x_train,
  y_train,
  epochs = 500,
  batch_size = 128,
  validation_data = list(x_val, y_val),
  verbose = 1
)
```

This is by far the best so far, even though it seems to plateau especially for the val_accuracy and then overfit on the training data.

Model 4: Model 3 with class weights

We try to improve the model 3 by introducing class weights to balance out the imbalance of the classes.

```
class_frequencies <- table(y_train_labels) # Get frequency of each class
total_instances <- length(y_train_labels) # Total number of instances

# Calculate class weights by taking the inverse of class frequencies
class_weights <- sapply(class_frequencies, function(freq) total_instances /
 ↪  (length(class_frequencies) * freq))

# Weighted averaging to reduce class weight extremity
alpha <- 0.2  # A value between 0 and 1; lower values make weights less aggressive
uniform_weights <- rep(1, length(class_weights))
class_weights <- alpha * class_weights + (1 - alpha) * uniform_weights
```

```
# Print the adjusted class weights
print(class_weights)
```

We will now try to include the class weights in the model creation to see whether it performs better. This was an iterative process and the following changes were made over the process of the exercise:

- We tried several times and saw that it starts at a much lower accuracy during training (around 0.25), so we realised we will need more epochs to get to a point where the model learns successfully, so we went up to 1000 epochs of training.
- We saw that the performance was not satisfactory with the class weights as they were because the effect was too extreme, so we decided to do weight averaging to adjust the effect the weights had. With a weight average of about 0.4, the performance was quite good.

```
model <- keras_model_sequential() %>%
    layer_dense(units = 256, activation = "relu", input_shape = ncol(x_train)) %>%
    layer_dropout(rate=0.25) %>%
    layer_batch_normalization() %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dropout(rate=0.15) %>%
    layer_batch_normalization() %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dropout(rate=0.10) %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dense(units = 8, activation = "softmax")


opti <- optimizer_nadam()
loss <- loss_categorical_crossentropy()

model %>% compile(
  optimizer = opti,
  loss = loss,
  metrics = c(metric_categorical_accuracy, "accuracy")
)

history <- model %>% fit(
  x_train,
  y_train,
  epochs = 1000,
  batch_size = 128,
  validation_data = list(x_val, y_val),
  class_weight = class_weights, # this is what we changed to model 3
  verbose = 1
)
```

Evaluating the performance:

```
results <- model %>% evaluate(x_test, y_test)

model %>% evaluate(x_val, y_val)
model %>% evaluate(x_train, y_train)
```

Model 5: trying different optimizers

In the next step we try different optimizers and settings on the optimizer.

```
model <- keras_model_sequential() %>%
    layer_dense(units = 256, activation = "relu", input_shape = ncol(x_train)) %>%
    layer_dropout(rate=0.3) %>%
    layer_batch_normalization() %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dropout(rate=0.2) %>%
    layer_batch_normalization() %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dropout(rate=0.10) %>%
    layer_dense(units = 128, activation = "relu") %>%
    layer_dense(units = 8, activation = "softmax")


opti <- optimizer_nadam(learning_rate = 0.001, beta_1 = 0.9, beta_2 = 0.999)
loss <- loss_categorical_crossentropy()

model %>% compile(
  optimizer = opti,
  loss = loss,
  metrics = c(metric_categorical_accuracy, "accuracy")
)

history <- model %>% fit(
  x_train,
  y_train,
  epochs = 300,
  batch_size = 128,
  validation_data = list(x_val, y_val),
  class_weight = class_weights,
  verbose = 2
)
```

Model 4: with 10-fold CV and grid search

Here we use a 10-fold crossvalidation as well as a grid search with the model from the previous step (Model 3). To do so, we must first convert y_train from a matrix to a vector.

```
# mapping from one-hot-endoced names to numeric labels
class_mapping <- c(
  "status_0" = 0,
  "status_1" = 1,
  "status_2" = 2,
  "status_3" = 3,
  "status_4" = 4,
  "status_5" = 5,
  "status_C" = 6,
  "status_X" = 7
)

class_names <- colnames(y_train)
```

```r
y_train_labels <- apply(y_train, 1, function(row) {
  class_name <- class_names[which.max(row)]  # Get the class name for the row
  class_mapping[class_name]  # Map it to a numeric label
})
```

CV

```r
create_model <- function(size, activation, learning_rate) {
  model <- keras_model_sequential() %>%
    layer_dense(units = size, activation = activation, input_shape = 52,
    ↪  kernel_regularizer = regularizer_l2(l = 0.002)) %>%
    layer_batch_normalization() %>%
    layer_dropout(rate = 0.15) %>%
    layer_dense(units = size, activation = activation, kernel_regularizer =
    ↪  regularizer_l2(l = 0.002)) %>%
    layer_batch_normalization() %>%
    layer_dropout(rate = 0.15) %>%
    layer_dense(units = size, activation = activation) %>%
    layer_batch_normalization() %>%
    layer_dropout(rate = 0.15) %>%
    layer_dense(units = size, activation = activation) %>%
    layer_dense(units = 8, activation = "softmax")

  optimizer <- optimizer_nadam(learning_rate = learning_rate)

  model %>% compile(
    optimizer = optimizer,
    loss = loss_categorical_crossentropy(),
    metrics = c(metric_categorical_accuracy(), 'accuracy')
  )

  return(model)
}

# Grid search hyperparameters
grid <- expand.grid(
  size = c(256, 512),
  activation = c("relu", "tanh"),
  learning_rate = c(0.001, 0.01),
  batch_size = c(128, 256),
  epochs = c(250, 500)
)

# Setting up 5-fold cross-validation
train_control <- trainControl(method = "cv", number = 5, verboseIter = TRUE)

# Custom training function
train_model <- function(size, activation, learning_rate, batch_size, epochs) {
  model <- create_model(size, activation, learning_rate)
  history <- model %>% fit(
    x_train, y_train,
    epochs = epochs,
    batch_size = batch_size,
```

```
    validation_data = list(x_val, y_val),
    callbacks = list(callback_early_stopping(monitor = 'val_loss', patience = 10)),
    verbose = 0
  )

  # Return the validation accuracy of the final epoch
  return(max(history$metrics$val_accuracy))
}

# Performing grid search
results <- lapply(1:nrow(grid), function(i) {
  params <- grid[i, ]
  cat("Training with params:", paste(params, collapse = ", "), "
")
  val_accuracy <- train_model(params$size, params$activation, params$learning_rate,
 ↪  params$batch_size, params$epochs)
  return(c(params, val_accuracy = val_accuracy))
})

# Convert results to data frame
results_df <- do.call(rbind, results)
print(results_df)
```

The best params from this grid search were: [26,] 512 relu 0.001 256 500 0.8412699

We run another grid search based on those hyperparameters:

```
create_model <- function(size, activation, learning_rate) {
  model <- keras_model_sequential() %>%
    layer_dense(units = size, activation = activation, input_shape = 52,
     ↪  kernel_regularizer = regularizer_l2(l = 0.002)) %>%
    layer_batch_normalization() %>%
    layer_dropout(rate = 0.15) %>%
    layer_dense(units = size, activation = activation, kernel_regularizer =
     ↪  regularizer_l2(l = 0.002)) %>%
    layer_batch_normalization() %>%
    layer_dropout(rate = 0.15) %>%
    layer_dense(units = size, activation = activation) %>%
    layer_batch_normalization() %>%
    layer_dropout(rate = 0.15) %>%
    layer_dense(units = size, activation = activation) %>%
    layer_dense(units = 8, activation = "softmax")

  optimizer <- optimizer_nadam(learning_rate = learning_rate)

  model %>% compile(
    optimizer = optimizer,
    loss = loss_categorical_crossentropy(),
    metrics = c(metric_categorical_accuracy(), 'accuracy')
  )

  return(model)
}
```

```
grid <- expand.grid(
  size = c(512),
  activation = c("relu"),
  learning_rate = c(0.00001, 0.00005, 0.0001),
  batch_size = c(256, 512),
  epochs = c(500, 1000)
)


# Setting up 5-fold cross-validation
train_control <- trainControl(method = "cv", number = 5, verboseIter = TRUE)

# Custom training function
train_model <- function(size, activation, learning_rate, batch_size, epochs) {
  model <- create_model(size, activation, learning_rate)
  history <- model %>% fit(
    x_train, y_train,
    epochs = epochs,
    batch_size = batch_size,
    validation_data = list(x_val, y_val),
    callbacks = list(callback_early_stopping(monitor = 'val_loss', patience = 10)),
    verbose = 0
  )

  # Return the validation accuracy of the final epoch
  return(max(history$metrics$val_accuracy))
}

# Performing grid search
results <- lapply(1:nrow(grid), function(i) {
  params <- grid[i, ]
  cat("Training with params:", paste(params, collapse = ", "), "
")
  val_accuracy <- train_model(params$size, params$activation, params$learning_rate,
 ↪  params$batch_size, params$epochs)
  return(c(params, val_accuracy = val_accuracy))
})

# Convert results to data frame
results_df <- do.call(rbind, results)
print(results_df)
```

The best params in the first round are: [7,] 512 relu 5e-04 512 500 0.8530021

Second round: grid <- expand.grid(size = c(512), activation = c("relu"), learning_rate = c(0.0001, 0.0005, 0.001), batch_size = c(256, 512), epochs = c(500, 750, 1000))

[1,] 512 relu 1e-04 256 500 0.8665089 [4,] 512 relu 1e-04 512 500 0.8643399 [13,] 512 relu 1e-04 256 1000 0.8634526 [16,] 512 relu 1e-04 512 1000 0.8635512 (actually, all with the smaller learning rate! might use that for later models?..?)

Experimenting with Exponential Learning Rate Decay

We do more exploration of learning rate decay, this time using exponential decay.

```
learning_rate_schedule <- learning_rate_schedule_exponential_decay(
  initial_learning_rate = 0.001, # Start with a higher learning rate
  decay_steps = 50,              # Decay the learning rate every 50 steps
  decay_rate = 0.9,              # Multiply the learning rate by 0.9 every 50 steps
  staircase = TRUE               # Decay happens in discrete steps
)

# Updated Optimizer with Decayed Learning Rate
model %>% compile(
  optimizer = optimizer_adam(learning_rate = learning_rate_schedule),
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

# Model Training
history <- model %>% fit(
  x = x_train, y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 300,
  batch_size = 64,
  callbacks = list(
    callback_early_stopping(patience = 20)
  ),
  verbose = 1
)
```

This does not provide a satisfactory improvement, but is something we might explore more later on.

Oversampling / Undersampling

Oversampling Minority Classes

```
oversample_minority <- function(data, target) {
  target_col <- enquo(target)

  # Count the number of samples in each class
  class_counts <- data %>%
    count(!!target_col) %>%
    arrange(n)

  # Determine the majority class size
  majority_size <- max(class_counts$n)

  # Oversample minority classes
  oversampled_data <- data %>%
    group_by(!!target_col) %>%
    group_modify(~ {
      if (nrow(.x) < majority_size) {
        # Calculate how many additional samples are needed
        additional_samples <- majority_size - nrow(.x)

        # Resample with slight modifications
```

```
        new_samples <- .x %>%
          sample_n(additional_samples, replace = TRUE) %>%
          mutate(across(where(is.numeric), ~ . + rnorm(n(), mean = 0, sd = sd(.x) *
           ↪  0.05))) # Add slight changes to numeric columns

        bind_rows(.x, new_samples)
      } else {
        .x
      }
    }) %>%
    ungroup()

  return(oversampled_data)
}

# Apply the function to your dataset
oversampled_train_data <- oversample_minority(train_data, status)

# Check class distribution
table(oversampled_train_data$status)
```

```
# Separate the features (X) and target (Y)
x_train_oversampled <- oversampled_train_data %>%
  select(-status) # Select all columns except the target variable

y_train_oversampled <- oversampled_train_data %>%
  pull(status) # Extract the target variable as a vector

y_train_oversampled_df <- data.frame(status = y_train_oversampled)

# Perform one-hot encoding
y_train_one_hot <- dummy_cols(y_train_oversampled_df, select_columns = "status",
 ↪  remove_selected_columns = TRUE)

# Convert x_train_oversampled to a matrix
x_train_matrix <- as.matrix(x_train_oversampled)

# Ensure all values are numeric
x_train_matrix <- apply(x_train_matrix, 2, as.numeric)

y_train_one_hot <- as.matrix(y_train_one_hot)
```

Combined Over/ Undersampling

```
# Function for balanced resampling with size constraint
balanced_resample <- function(data, target, total_target_size = 67000) {
  target_col <- enquo(target)

  # Count the number of samples in each class
  class_counts <- data %>%
    count(!!target_col) %>%
    arrange(desc(n))
```

```r
  # Calculate the average target size per class
  num_classes <- n_distinct(data[[quo_name(target_col)]])
  target_class_size <- total_target_size / num_classes

  # Initialize a list to hold the resampled data
  resampled_data <- list()

  for (class in unique(data[[quo_name(target_col)]])) {
    class_data <- data %>% filter(!!target_col == class)

    if (nrow(class_data) > target_class_size) {
      # Undersample majority classes
      sampled_data <- class_data %>% sample_n(ceiling(target_class_size))
    } else if (nrow(class_data) < target_class_size) {
      # Oversample minority classes
      additional_samples <- ceiling(target_class_size - nrow(class_data))
      sampled_data <- class_data %>%
        bind_rows(
          class_data %>%
            slice_sample(n = additional_samples, replace = TRUE) %>%
            mutate(across(where(is.numeric), ~ . + rnorm(n(), mean = 0, sd = sd(.x) *
             ↪  0.05))) # Add slight changes
        )
    } else {
      # Retain as is
      sampled_data <- class_data
    }

    resampled_data[[as.character(class)]] <- sampled_data
  }

  # Combine all resampled classes
  balanced_data <- bind_rows(resampled_data)

  # Ensure the final size is close to the target size by randomly sampling if necessary
  if (nrow(balanced_data) > total_target_size) {
    balanced_data <- balanced_data %>% sample_n(total_target_size)
  }

  return(balanced_data)
}

# Apply the function to your dataset
balanced_train_data <- balanced_resample(train_data, status, total_target_size = 67000)

# Split into x_train and y_train
x_train <- balanced_train_data %>% select(-status)  # Features
y_train <- balanced_train_data$status               # Target variable

# One-hot encode y_train
y_train_one_hot <- model.matrix(~ y_train - 1)       # Create one-hot encoding matrix
```

```r
# Convert x_train and y_train to matrices
x_train_matrix <- as.matrix(x_train)
y_train_matrix <- as.matrix(y_train_one_hot)

# Check class distribution and total size
cat("Class Distribution:\n")
print(table(balanced_train_data$status))
cat("\nTotal Dataset Size:", nrow(balanced_train_data), "\n")
```

```r
x_val <- val_data %>% select(-all_of("status")) %>% as.matrix()
x_test <- test_data %>% select(-all_of("status")) %>% as.matrix()

create_one_hot_matrix_fastDummies <- function(data, column_name) {
  one_hot_encoded <- dummy_cols(data, select_columns = column_name, remove_first_dummy =
 ↪  FALSE, remove_selected_columns = TRUE)

  dummy_columns <- grep(paste0(column_name, "_"), colnames(one_hot_encoded), value =
 ↪  TRUE)
  one_hot_matrix <- as.matrix(one_hot_encoded[, dummy_columns])

  return(one_hot_matrix)
}

y_val <- create_one_hot_matrix_fastDummies(val_data, "status")
y_test <- create_one_hot_matrix_fastDummies(test_data, "status")
```

```r
# Convert one-hot encoded y_train to original class indices
original_y_train <- apply(y_train, 1, which.max) - 1  # Subtract 1 for zero-indexing
```

Model With Class Weights

Computing Class Frequencies

We try again to improve our model with class weights. For this, we calculated the class frequencies. As the initial model with full class frequencies performed worse, we included the smothing_factor to apply to the class weights. This value of 0.5 We used in the end, we established through the CV featured below.

```r
class_counts <- colSums(y_train)  # Sum each column to get counts for one-hot encoded
 ↪  classes
total_samples <- sum(class_counts)  # Total number of samples
class_weights <- as.list(total_samples / (length(class_counts) * class_counts))

smoothing_factor <- 0.5  # Adjust this value as needed
adjusted_class_weights <- lapply(class_weights, function(w) w^(1 - smoothing_factor))


adjusted_class_weights <- setNames(adjusted_class_weights, colnames(y_train))
adjusted_class_weights
```

```
## $status_0
## [1] 0.4026487
##
## $status_1
## [1] 1.141085
```

```
##
## $status_2
## [1] 3.443394
##
## $status_3
## [1] 6.571685
##
## $status_4
## [1] 8.599873
##
## $status_5
## [1] 4.752109
##
## $status_C
## [1] 2.163533
##
## $status_X
## [1] 1.208228
```

CV to find best smoothing factor

In the next step, I want to try setting up a CV to find the optimal smoothing factor for the class weights.

```r
# Generate smoothing factor grid
smoothing_factors <- seq(0.1, 1.0, by = 0.1)

# Create folds for cross-validation
k <- 5  # Number of folds
folds <- createFolds(seq_len(nrow(y_train)), k = k)

# Placeholder to store results
results <- data.frame(SmoothingFactor = numeric(), Fold = integer(), ValAccuracy =
 ↪  numeric())

# Cross-validation loop
for (smoothing_factor in smoothing_factors) {
  for (fold in seq_along(folds)) {
    # Split data into training and validation for this fold
    train_indices <- folds[[fold]]
    val_indices <- setdiff(seq_len(nrow(x_train)), train_indices)

    x_train_fold <- x_train[train_indices, , drop = FALSE]
    y_train_fold <- y_train[train_indices, , drop = FALSE]
    x_val_fold <- x_train[val_indices, , drop = FALSE]
    y_val_fold <- y_train[val_indices, , drop = FALSE]

    # Calculate class weights with smoothing
    class_counts <- colSums(y_train_fold)  # Sum columns of one-hot encoded labels
    total_samples <- sum(class_counts)
    class_weights <- as.list(total_samples / (length(class_counts) * class_counts))
    adjusted_class_weights <- lapply(class_weights, function(w) w^(1 - smoothing_factor))
    adjusted_class_weights <- setNames(adjusted_class_weights, colnames(y_train))  #
 ↪  Assign names to weights

    # Define model
```

```r
    model <- keras_model_sequential() %>%
      layer_dense(units = 512, activation = "relu", input_shape = ncol(x_train),
       ↪  kernel_regularizer = regularizer_l2(l = 0.005)) %>%
      layer_batch_normalization() %>%
      layer_dropout(rate = 0.2) %>%
      layer_dense(units = 256, activation = "relu") %>%
      layer_batch_normalization() %>%
      layer_dropout(rate = 0.1) %>%
      layer_dense(units = 256, activation = "relu") %>%
      layer_dense(units = ncol(y_train), activation = "softmax")

    # Compile model
    opti <- optimizer_nadam(learning_rate = 0.0001)
    loss <- loss_categorical_crossentropy()
    model %>% compile(
      optimizer = opti,
      loss = loss,
      metrics = c(metric_categorical_accuracy, "accuracy")
    )

    # Callback to reduce learning rate on plateau
    callback_reduce_lr <- callback_reduce_lr_on_plateau(
      monitor = "val_loss",
      factor = 0.5,
      patience = 10,
      min_lr = 1e-6
    )

    # Train model
    history <- model %>% fit(
      x_train_fold,
      y_train_fold,
      epochs = 50,  # Reduced for CV
      batch_size = 256,
      validation_data = list(x_val_fold, y_val_fold),
      verbose = 0,
      callbacks = list(callback_reduce_lr),
      class_weight = adjusted_class_weights
    )

    # Store validation accuracy
    val_accuracy <- max(history$metrics$val_accuracy)
    results <- rbind(results, data.frame(SmoothingFactor = smoothing_factor, Fold = fold,
 ↪  ValAccuracy = val_accuracy))
  }
}

# Summarize results and find the best smoothing factor
summary <- aggregate(ValAccuracy ~ SmoothingFactor, data = results, mean)
best_smoothing_factor <- summary$SmoothingFactor[which.max(summary$ValAccuracy)]

print(summary)
cat("Best smoothing factor:", best_smoothing_factor, "\n")
```

CNN

We also gave CNNs a try, even though we were not quite sure whether it made sense for this data. The models both showed an accuracy 77% and this means it was likely just predicting the majority class for every instance, so we didn't further inverstigate into this.

Model 1

```
model_cnn <- keras_model_sequential() %>%

  layer_reshape(target_shape = c(ncol(x_train), 1), input_shape = c(ncol(x_train))) %>%

  layer_conv_1d(filters = 32, kernel_size = 3, activation = "relu", padding = "same") %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(rate = 0.3) %>%

  layer_conv_1d(filters = 64, kernel_size = 3, activation = "relu", padding = "same") %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(rate = 0.3) %>%

  layer_flatten() %>%

  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.4) %>%

  layer_dense(units = ncol(y_train), activation = "softmax")


model_cnn %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.001),
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

history_cnn <- model_cnn %>% fit(
  x = x_train,
  y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 500,
  batch_size = 32
)
```

```
# Evaluate the model on the test set
model_cnn %>% evaluate(x_test, y_test, verbose = 2)
```

```
# Plot training history
plot(history_cnn)
```

Model 2

```
model_cnn2 <- keras_model_sequential() %>%

  layer_reshape(target_shape = c(ncol(x_train), 1), input_shape = c(ncol(x_train))) %>%
```

```
  layer_conv_1d(filters = 32, kernel_size = 3, activation = "relu", padding = "same") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(rate = 0.3) %>%

  layer_conv_1d(filters = 64, kernel_size = 3, activation = "relu", padding = "same") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(rate = 0.4) %>%

  layer_conv_1d(filters = 128, kernel_size = 3, activation = "relu", padding = "same")
   ↪  %>%
  layer_batch_normalization() %>%
  layer_max_pooling_1d(pool_size = 2) %>%
  layer_dropout(rate = 0.5) %>%

  layer_flatten() %>%

  layer_dense(units = 256, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.5) %>%

  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.4) %>%

  layer_dense(units = ncol(y_train), activation = "softmax")

# Compile the model
model_cnn2 %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.0005),
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

# Train the model
history_cnn2 <- model_cnn2 %>% fit(
  x = x_train,
  y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 500,
  batch_size = 32
)
```

```
# Evaluate the model on the test set
model_cnn2 %>% evaluate(x_test, y_test, verbose = 2)
```

```
# Plot training history
plot(history_cnn2)
```

Final Model

This is our final model we use to predict the data for our assignment. For the final model training, we took out the early stopping, because we wanted to gain a small bit more on the train data to maximize our points there, but still leave the code in that we used for early stopping.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 1024, activation = "relu", input_shape = 52, kernel_regularizer =
   ↪  regularizer_l2(l = 0.005)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.15) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 8, activation = "softmax")

opti <- optimizer_nadam(0.001)
loss <- loss_categorical_crossentropy()

model %>% compile(
  optimizer = opti,
  loss = loss,
  metrics = c(metric_categorical_accuracy, "accuracy")
)

# Callback to reduce learning rate on plateau
callback_reduce_lr <- callback_reduce_lr_on_plateau(
  monitor = "val_loss",    # Monitor validation loss
  factor = 0.5,            # Reduce learning rate by this factor
  patience = 10,           # Number of epochs with no improvement before reducing LR
  min_lr = 1e-6            # Minimum learning rate to prevent it from becoming too small
)

# callback_early_stop <- callback_early_stopping(
#   monitor = "val_loss",    # Monitor validation loss
#   patience = 100,           # Number of epochs with no improvement before stopping
#   restore_best_weights = TRUE # Restore the best model weights
# )

history <- model %>% fit(
  x_train,
  y_train,
  epochs = 500,
  batch_size = 256,
  validation_data = list(x_val, y_val),
  verbose = 1,
  callbacks = list(callback_reduce_lr)
)
```

The final model performed better without the classs weights so we ended up taking this code out of the model.

Exporting the Final Model

To use the model in the reality check file, we tried to export it to hd5 like it was required in the project description. This no longer works in keras3, instead, we save it to a .keras model

```
model |> save_model('model.keras')
```

Loading and Evaluating the Model

Here we load our saved model and use it to predict X (the whole dataset) and evaluate it's accuracy on the whole dataset to measure the performance on the public dataset.

```
model <- load_model('model.keras')
```

Creating X and y

We create X and y out of the data_final object to be able to evaluate our model.

```
data_final <- read.csv("data_outlier_removed.csv", header = TRUE, sep = ",")

data_final <- data_final %>%
  mutate(
    DAYS_EMPLOYED = abs(DAYS_EMPLOYED),
    DAYS_BIRTH = abs(DAYS_BIRTH)
  )

data_final <- data_final

# Normalization
columns_to_normalize <- c("AMT_INCOME_TOTAL", "CNT_FAM_MEMBERS", "DAYS_EMPLOYED",
 ↪  "CNT_CHILDREN", "DAYS_BIRTH")
normalize_columns_with_train_params <- function(dataset, columns_to_normalize, train_min,
 ↪  train_max) {
  # Iterate through each column to normalize
  for (col in columns_to_normalize) {
    # Apply normalization using train_min and train_max
    dataset[[col]] <- (dataset[[col]] - train_min[col]) / (train_max[col] -
 ↪  train_min[col])
  }
  return(dataset)
}

#importing the normalization parameters we got from the training data
normalization_params <- read.csv("normalization_params.csv")


columns_to_normalize <- normalization_params$column

# Extract min and max values as named vectors for easy use
train_min <- setNames(normalization_params$min_val, normalization_params$column)
train_max <- setNames(normalization_params$max_val, normalization_params$column)

data_final <- normalize_columns_with_train_params(data_final, columns_to_normalize,
 ↪  train_min, train_max)
```

```r
# clipping the values after normalization to ensure values between 0 snd 1
data_final[columns_to_normalize] <- lapply(
  data_final[columns_to_normalize],
  function(col) pmin(pmax(col, 0), 1)
)


X <- as.matrix(data_final %>% select(-status))
y <- as.matrix(data_final$status)

colnames(y) <- "status"
create_one_hot_matrix_fastDummies <- function(data, column_name) {
  one_hot_encoded <- dummy_cols(data, select_columns = column_name, remove_first_dummy =
 ↪  FALSE, remove_selected_columns = TRUE)

  dummy_columns <- grep(paste0(column_name, "_"), colnames(one_hot_encoded), value =
 ↪  TRUE)
  one_hot_matrix <- as.matrix(one_hot_encoded[, dummy_columns])

  return(one_hot_matrix)
}


y <- create_one_hot_matrix_fastDummies(y, "status")
```

Evaluating our Model on the Whole Dataset

Here, we use the model we imported to predict the whole dataset and get an evaluation of the performance on the complete set.

```r
results <- model %>% evaluate(data.matrix(X), data.matrix(y))
```

```
## 2113/2113 - 2s - 768us/step - accuracy: 0.9471 - categorical_accuracy: 0.9471 - loss: 0.2377
```

Our model achieved 95% accuracy on the whole dataset. We are aware, that this model is likely overfit, but we want to achieve optimal points on the public dataset. It did however also perform well on the validation and test data, achieving 87% on both, so we chose this model.