Insurance customer churn prediction

Nico Henzel

2022-04-25

1 Plan

1.1 Use case

The goal for this project is to train a classification algorithm with anonymized insurance contract data in order to predict the churn of insurance customers (whether the insurance company will lose a customer or not). Since the outcome is categorial (customer stays or leaves) and the churn in the available training data is labeled, we face a supervised learning problem that can be solved with a classification model.

A potential use case, where these insights could be applied, is:
Any insurance company wants to improve (or at least keep) their customer loyalty. If the insurance case handlers could understand why a customer quits a contract and more importantly if a customer is close to quitting an ongoing contract, they could start countermeasures in order to improve customer service and maybe keep the customer. The output from this model could be provided as information within a dashboard, or implemented into existing applications which display contract information to the case handler. This information can then be used to make further decisions by the case handlers. The visualization in this project will be realized through a dashboard made in shiny.

1.2 Metrics

To measure our models performance, we will use the specificity (True negatives / (True negatives + False positives)) and also visualize the predictions with a confusion matrix. The specificity is the most important metric since we want to measure how well our model predicts true negatives (churn) and minimizes false positives (model predicting not churning, when churn would be correct) in order to give the case handlers the best overview of possible churn candidates. We will train different models and compare their performance based on specificity and the confusion matrix.

The model is successful when:

- Reasonable portion (> 30%) of customer churning can be predicted on new data

Additionally we will cover our use case with * Visualized recommendations that allow case handlers to identify potential churn and get in contact with customers.

2 Data understanding

2.1 Import

Load training and test dataset (provided from kaggle - see **https://www.kaggle.com/datasets/k123vinod/insurance-churn-prediction-weekend-hackathon (https://www.kaggle.com/datasets/k123vinod/insurance-churn-prediction-weekend-hackathon)**)

Note: the training dataset will be used to fit the model. The provided test dataset will be treated as new data to simulate the models performance, since the data is unlabeled.

```
LINK_train <-
  "https://raw.githubusercontent.com/NicoHenzel/Insurance-Churn-Prediction/main/Data/Train.csv"

new_training_data <-
  read_csv(LINK_train, show_col_types = FALSE)
```

```
LINK_new <-
  "https://raw.githubusercontent.com/NicoHenzel/Insurance-Churn-Prediction/main/Data/Test.csv"

new_data <-
  read_csv(LINK_new, show_col_types = FALSE)
```

2.2 First look

Look at the first rows from each dataframe.

```
new_training_data %>%
  slice_head(n=5) %>%
  gt() %>%
  tab_header(
    title = md("**Anonymized contract churn data**"),
    subtitle = md("Training set")
  ) %>%
  tab_source_note(
    source_note = "Source: Kaggle - Customer Churn Prediction - Weekend Hackathon"
  ) %>%
    tab_source_note(
    source_note = "https://www.kaggle.com/datasets/k123vinod/insurance-churn-prediction-weekend-hac
kathon"
    )
```

Anonymized contract cl

Training set

feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8
-0.2765146	-0.4244288	1.3449969	-0.01228261	0.07622994	1.0766475	0.1821975	3	0
0.8535731	0.1509913	0.5038918	-0.97917917	-0.56935064	-0.4114531	-0.2519404	4	1

Source: Kaggle - Customer Churn Prediction - Weekend Hackathon

https://www.kaggle.com/datasets/k123vinod/insurance-churn-prediction-weekend-hackathon

Anonymized contract cl

Training set

0.9477471	-0.1738321	1.8256285	-0.70347774	0.07622994	-0.4114531	-0.2519404	6	1
0.8535731	-0.3814037	0.9845233	-0.03946444	-0.56935064	-0.4114531	-0.2519404	4	0
1.3244430	1.5905268	-1.1783185	-0.09771123	-0.24656035	-0.4114531	-0.2519404	0	1

Source: Kaggle - Customer Churn Prediction - Weekend Hackathon

https://www.kaggle.com/datasets/k123vinod/insurance-churn-prediction-weekend-hackathon

Anonymized contract churi

Unlabeled dataset

feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8
0.5710512	0.4068430	0.9845233	0.0110161	-0.56935064	-0.4114531	-0.2519404	0	1
-1.1240804	-0.1669349	0.5038918	-0.3229321	0.72181052	0.5473231	0.1821975	0	2
0.4768772	0.1450794	-0.5775291	-0.6918284	-0.24656035	-0.4114531	-0.2519404	0	1
1.6069650	-0.4474193	1.8256285	-0.9830623	7.17761632	-0.4114531	-0.2519404	1	1
-0.9357324	-0.3646534	-1.1783185	-0.3229321	0.07622994	-0.4114531	-0.2519404	8	2

Source: Kaggle - Customer Churn Prediction - Weekend Hackathon

https://www.kaggle.com/datasets/k123vinod/insurance-churn-prediction-weekend-hackathon

This gives us the following informations:

1. The data seems to be clean already. Further information will be gathered in **Format Data** section.

2. The datasets are divided by a 75/25 split. The dimensions for both dataframes are the following:

- Training set:
  observations = 33908
  variables = 17

- New dataset:
  observations = 11303
  variables = 16

3. The labels column is a boolean (1 or 0) which only appears in the training data set.
   This variable can be seen as the indicator for churning, since the column doesn't appear in the given test data.
   This also means, that the provided test data can be treated as new data to simulate a churn prediction on.

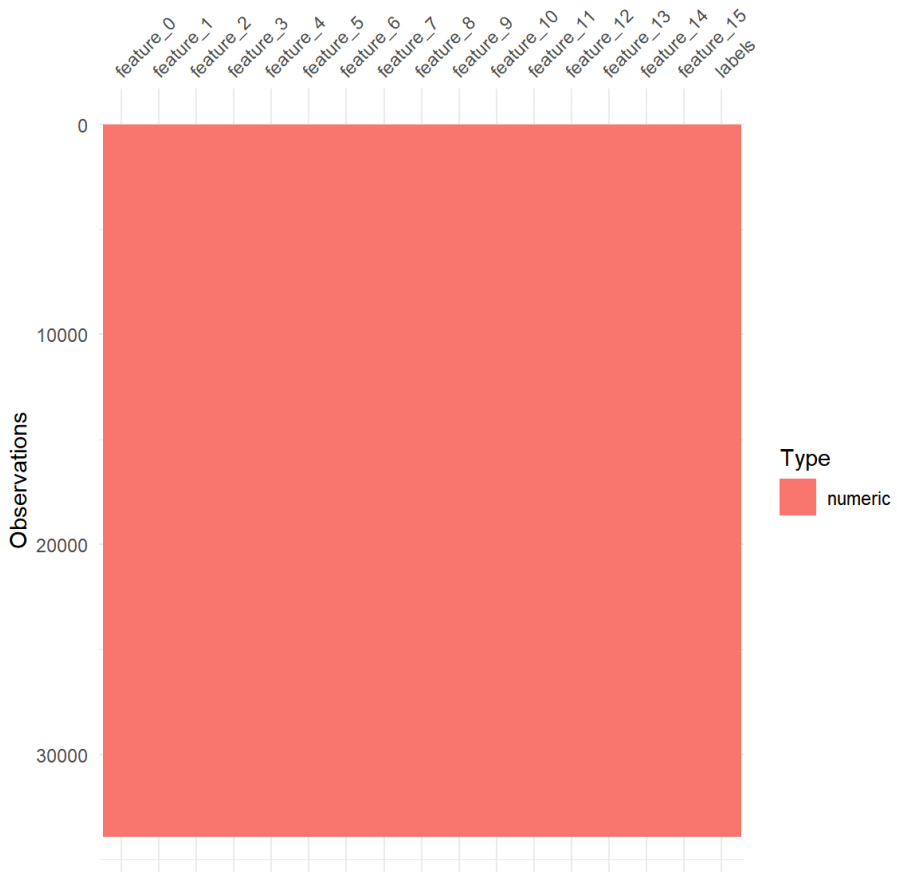
2.3 Clean data

The datasets are already clean (lowercase column names without spaces and no special characters). This makes it easier for us, since no data cleaning needs to be performed.

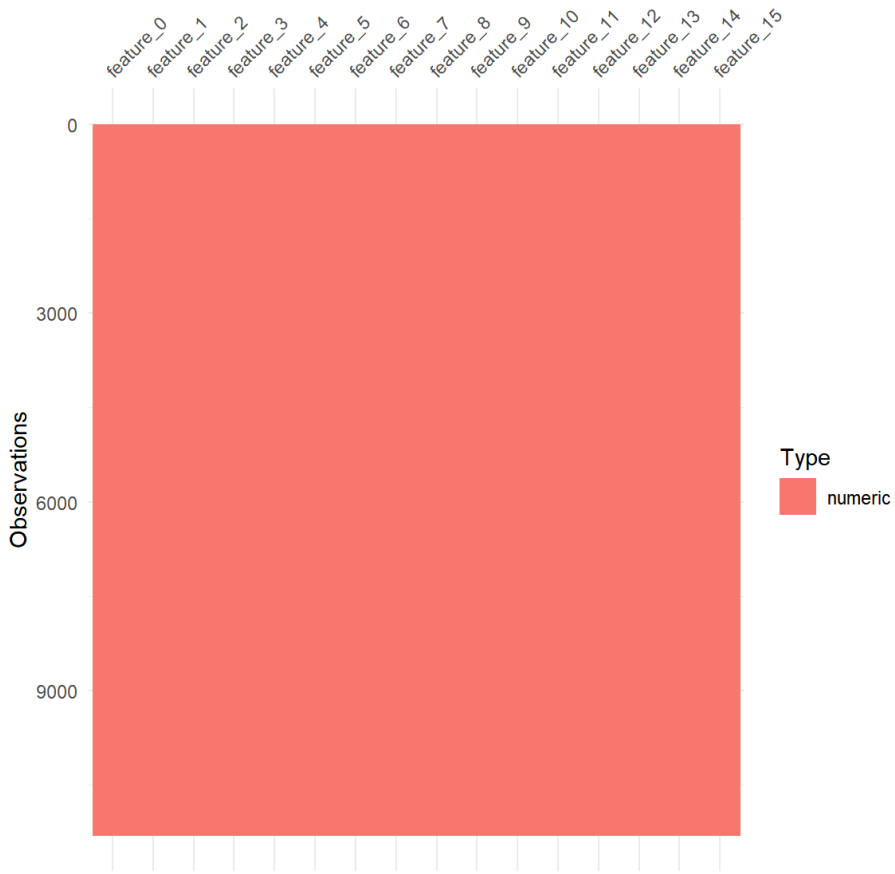
2.4 Format data

Next we will be inspecting the given data structure and check if there are any missing values in both datasets.

```
## Rows: 33,908
## Columns: 17
## $ feature_0  <dbl> -0.276514595, 0.853573137, 0.947747115, 0.853573137, 1.3244~
## $ feature_1  <dbl> -0.42442881, 0.15099126, -0.17383206, -0.38140368, 1.590526~
## $ feature_2  <dbl> 1.34499695, 0.50389181, 1.82562845, 0.98452332, -1.17831846~
## $ feature_3  <dbl> -0.012282614, -0.979179166, -0.703477740, -0.039464445, -0.~
## $ feature_4  <dbl> 0.07622994, -0.56935064, 0.07622994, -0.56935064, -0.246560~
## $ feature_5  <dbl> 1.0766475, -0.4114531, -0.4114531, -0.4114531, -0.4114531, ~
## $ feature_6  <dbl> 0.1821975, -0.2519404, -0.2519404, -0.2519404, -0.2519404, ~
## $ feature_7  <dbl> 3, 4, 6, 4, 0, 4, 9, 9, 8, 7, 1, 1, 11, 10, 9, 7, 6, 9, 8, ~
## $ feature_8  <dbl> 0, 1, 1, 0, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 0, 2, 2, 2, 2, 2,~
## $ feature_9  <dbl> 1, 2, 2, 2, 1, 1, 1, 1, 2, 1, 1, 0, 0, 1, 1, 1, 2, 1, 1, 1,~
## $ feature_10 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ feature_11 <dbl> 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,~
## $ feature_12 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ feature_13 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 2, 0, 0, 2, 0, 0, 0,~
## $ feature_14 <dbl> 10, 0, 5, 5, 8, 10, 5, 5, 8, 8, 6, 5, 5, 8, 1, 0, 6, 0, 8, ~
## $ feature_15 <dbl> 2, 3, 3, 3, 3, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 3,~
## $ labels     <dbl> 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0,~
```



```
## Rows: 11,303
## Columns: 16
## $ feature_0  <dbl> 0.57105120, -1.12408039, 0.47687723, 1.60696496, -0.9357324~
## $ feature_1  <dbl> 0.40684299, -0.16693490, 0.14507941, -0.44741934, -0.364653~
## $ feature_2  <dbl> 0.9845233, 0.5038918, -0.5775291, 1.8256285, -1.1783185, 0.~
## $ feature_3  <dbl> 0.01101610, -0.32293211, -0.69182838, -0.98306229, -0.32293~
## $ feature_4  <dbl> -0.56935064, 0.72181052, -0.24656035, 7.17761632, 0.0762299~
## $ feature_5  <dbl> -0.4114531, 0.5473231, -0.4114531, -0.4114531, -0.4114531, ~
## $ feature_6  <dbl> -0.2519404, 0.1821975, -0.2519404, -0.2519404, -0.2519404, ~
## $ feature_7  <dbl> 0, 0, 0, 1, 8, 1, 0, 4, 4, 10, 4, 0, 8, 1, 10, 3, 10, 4, 9,~
## $ feature_8  <dbl> 1, 2, 1, 1, 2, 2, 2, 0, 1, 1, 2, 0, 2, 1, 1, 0, 2, 2, 0, 1,~
## $ feature_9  <dbl> 1, 1, 1, 0, 1, 3, 1, 1, 2, 1, 2, 1, 1, 1, 0, 1, 1, 2, 2, 2,~
## $ feature_10 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ feature_11 <dbl> 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1,~
## $ feature_12 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ feature_13 <dbl> 0, 0, 0, 0, 2, 2, 0, 2, 0, 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 2,~
## $ feature_14 <dbl> 11, 5, 1, 5, 8, 6, 3, 8, 5, 0, 8, 8, 8, 8, 3, 9, 1, 1, 9, 6~
## $ feature_15 <dbl> 3, 1, 3, 3, 3, 3, 0, 3, 3, 3, 3, 3, 3, 0, 3, 3, 3, 3, 3, 3,~
```



We can see that:

1. The datatype for every variable (feature column) in both datasets is double, although there are variables that only hold integer values (feature_7 to feature_15). In the **Data exploration** section we will inspect the difference between integer and float variables.
2. There are 0 missing values in the training and 0 in the new dataset.
3. The labels column is formatted as numeric (dbl). It should be a factor since it is a categorial variable with two levels (1 or 0).

First the labels column will be renamed to churn.

The column is also transformed into a factor type which is needed for running the classification algorithm.

```
df_train <-
  new_training_data %>%
  # Rename labels
  rename(churn = labels) %>%
  # Change column type to factor
  mutate(
    churn = as.factor(churn)
    )
```

Note:

No new variables will be created, since we work with anonymized data.

The **Data exploration** section covers analysis and interpretation of the relations between the variables.

2.5 Data overview

skim_type	skim_variable	n_missing	complete_rate	factor.ordered	factor.n_unique	factor.top_counts	nu
factor	churn	0	1	FALSE	2	0: 29941, 1: 3967	
numeric	feature_0	0	1	NA	NA	NA	-4
numeric	feature_1	0	1	NA	NA	NA	2
numeric	feature_2	0	1	NA	NA	NA	-2
numeric	feature_3	0	1	NA	NA	NA	-5
numeric	feature_4	0	1	NA	NA	NA	-2
numeric	feature_5	0	1	NA	NA	NA	-4
numeric	feature_6	0	1	NA	NA	NA	-7
numeric	feature_7	0	1	NA	NA	NA	4.
numeric	feature_8	0	1	NA	NA	NA	1.
numeric	feature_9	0	1	NA	NA	NA	1.
numeric	feature_10	0	1	NA	NA	NA	1
numeric	feature_11	0	1	NA	NA	NA	5
numeric	feature_12	0	1	NA	NA	NA	1
numeric	feature_13	0	1	NA	NA	NA	6

skim_type	skim_variable	n_missing	complete_rate	factor.ordered	factor.n_unique	factor.top_counts	nu
numeric	feature_14	0	1	NA	NA	NA	5.
numeric	feature_15	0	1	NA	NA	NA	2.

The overview shows:

- Feature_7 - 15 are discrete and furthermore feature_10 - 12 seem to be binary (1 or 0).
- Every other feature column holds continuous values.
- The maximum and minimum values differ quite a bit, which means the variables have different scales. This issue will be solved in the **Data preparation** section through feature scaling (data normalization).
- Feature 0-6 have a positive skew shown by the histograms. This does not need to be addressed for our classifier models (normal distributions are not needed).
- The shown stats (mean, sd, perecentiles) are nearly impossible to interpret at this point since we don't know what the variables itself mean.

2.6 Data split

Before exploring the data we perform a data split. The **Data exploration** will only be done on the training set. This is crucial so we don't use any information from the training set in our model building phase. Although the data already comes with a 75/25 split the provided test data will not be used for model training since it is unlabeled and better suited to simulate new data.

```
# Seeded split to provide the same split everytime the split is made to make sure we always use the same data for model training.
set.seed(42)
# Split the data with a 75/25 proportion in favor for the training set
data_split <-
  initial_split(
    df_train, # original training dataset
    prop = 3/4, # 75/25 split
    )
# Create the dataframes for training and testing
train_data <- training(data_split)
test_data <- testing(data_split)
```

2.7 Data exploration

In order to get a better understanding of the data and underlying relations between features, we plot the data in different ways.

2.7.1 Setup

We create a new dataframe to avoid altering the training dataset during the data exploration.

```
explore_data <-
  train_data
```

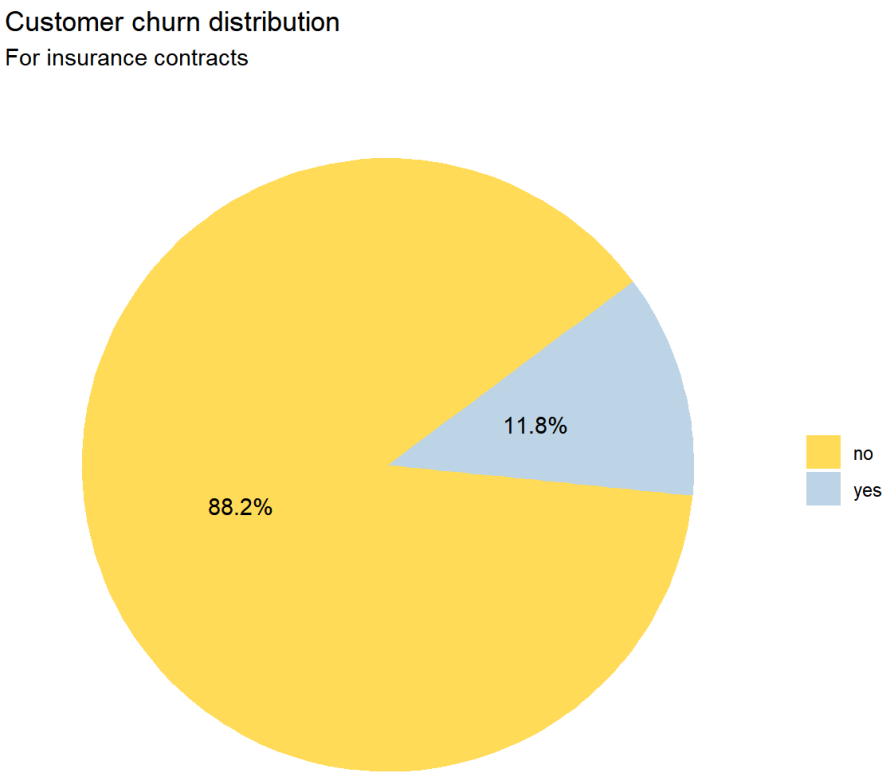
Then we change the binary values to named values to represent the churn in order to make the plots easier to read (Using yes and no instead of 1 and 0).

```
# Change column type to character to change values
explore_data <-
  explore_data %>%
  mutate(
    churn = as.character(churn)
  )
# Change values
explore_data$churn[explore_data$churn == "1"] <-
  "yes"
explore_data$churn[explore_data$churn == "0"] <-
  "no"
# Change column type back to factor
explore_data <-
  explore_data %>%
  mutate(
    churn = as.factor(churn)
  )
```

2.7.2 Churn distribution

The ratio of customers that have churned within the training data set is depicted below.

```
explore_data %>%
  count(churn, name ="churn_total") %>%
  mutate(percent = churn_total/sum(churn_total)*100,
         percent = round(percent, 2)) %>%
  ggplot(
  aes(x="",
    y=percent,
    fill=churn)
  ) +
  geom_bar(
    stat="identity",
    width=1
    ) +
  coord_polar("y", start = -175) +
  theme_classic() +
  theme(
    axis.line = element_blank(),
    axis.text = element_blank(),
    axis.ticks = element_blank()
  ) +
  scale_fill_manual(
    values=c("#ffdb58", "#bcd4e6")
      ) +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
    title = "Customer churn distribution",
    subtitle = "For insurance contracts") +
  geom_text(
    aes(label = paste0(percent, "%")),
    position = position_stack(vjust=0.5)
  )
```




```
explore_data %>%
  count(churn,
        name ="churn_total") %>%
  mutate(percent = churn_total/sum(churn_total)*100,
         percent = round(percent, 2)) %>%
 gt() %>%
  tab_header(
    title = "Insurance customers churn rate ",
    subtitle = "Anonymized training sample"
  ) %>%
  cols_label(
    churn = "Churn",
    churn_total = "Amount",
    percent = "Percent"
  )
```

Insurance customers churn rate

Anonymized training sample

Churn	Amount	Percent
no	22431	88.2
yes	3000	11.8

The data is imbalanced as nearly 90 % are labeled as not churning. We will need to compensate for this before performing hyperparameter tuning.

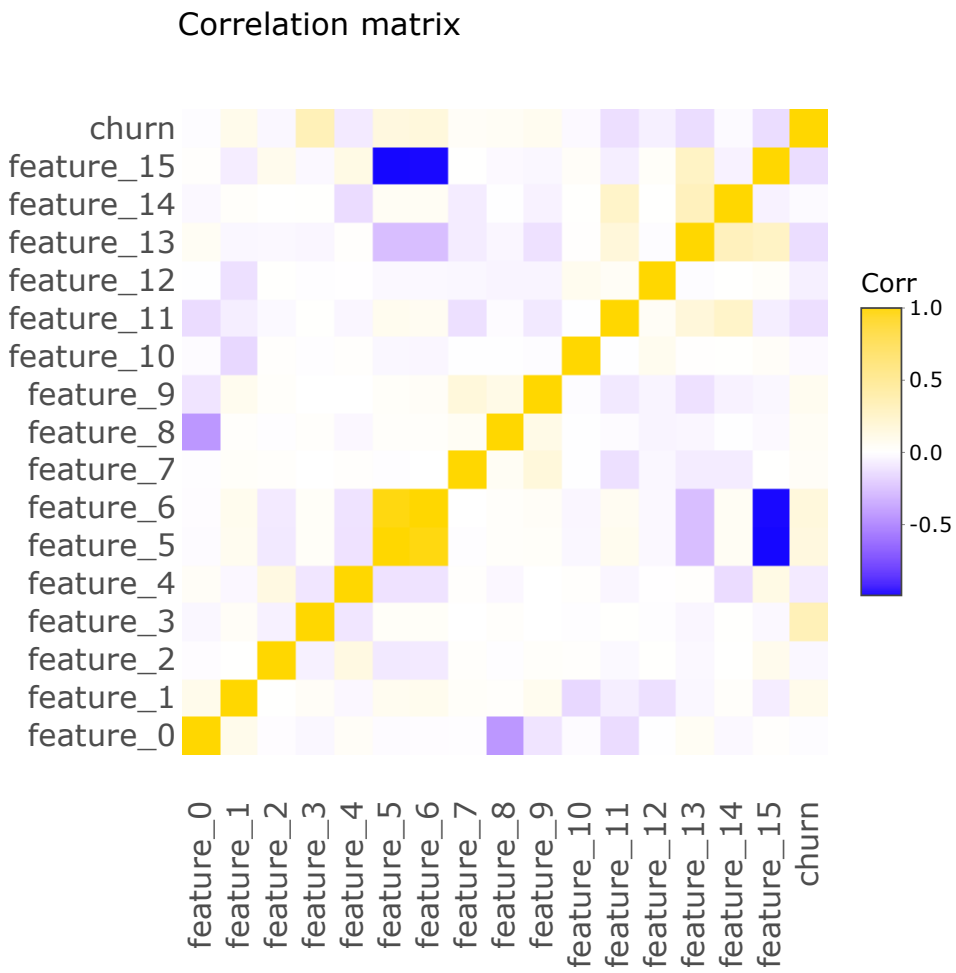
2.7.3 Correlation to churn

The correlation matrix helps us see different relations between the features as it may indicate a predictive relationship between variables (See: https://en.wikipedia.org/wiki/Correlation (https://en.wikipedia.org/wiki/Correlation)). It is formed by calculating the correlation coefficient **r** between each feature. **r** ranges from -1 to +1 and indicates the strength and direction of the linear relation:

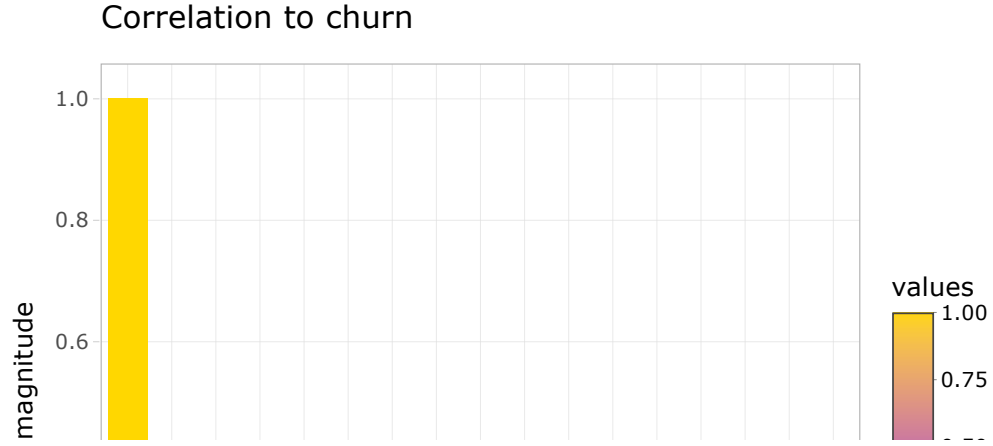
Positive numbers indicate that greater values of one variable lead to greater values of the other variable which also holds true for lower values (similar behavior).

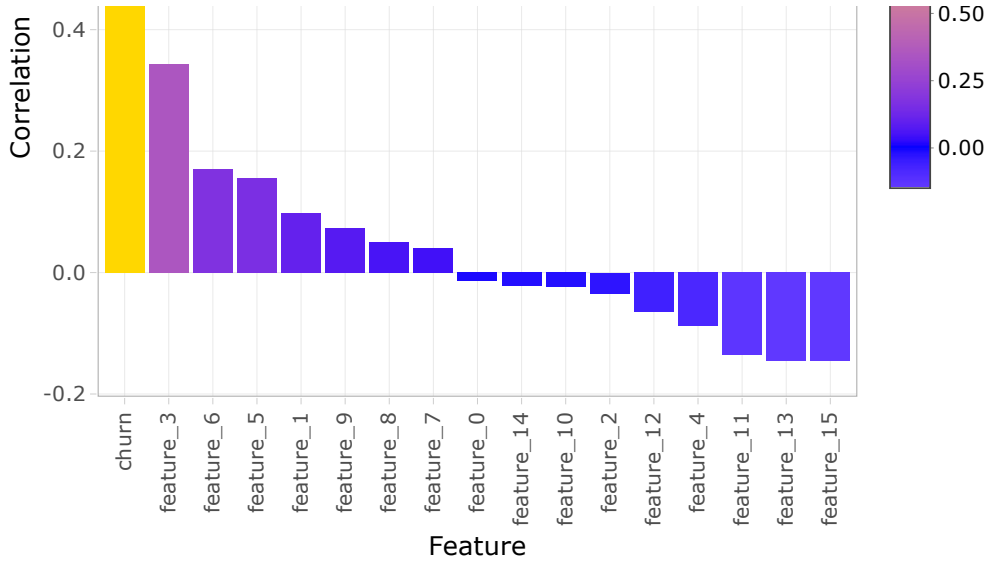
Negative numbers indicate that greater values of one variable correspond to lesser values of the other (opposite behavior).

(See: https://en.wikipedia.org/wiki/Correlation (https://en.wikipedia.org/wiki/Correlation) and https://en.wikipedia.org/wiki/Covariance (https://en.wikipedia.org/wiki/Covariance))



It is also useful to show the specific correlation of each feature to our churn variable:





We can see that the correlation between churn and the other features is the following:

- It does not correlate significantly with most of the features (|r| < 0.1)
- feature_3 has a moderate positive correlation with churn (r > 0.3).
- feature_5 & 6 correlate slightly positive (r > 0.1).
- feature_11 & 13 correlate slightly negative (r < -0.1).
- Additionally feature_5 & 6 correlate nearly perfectly linear with each other (r ~ 1) .

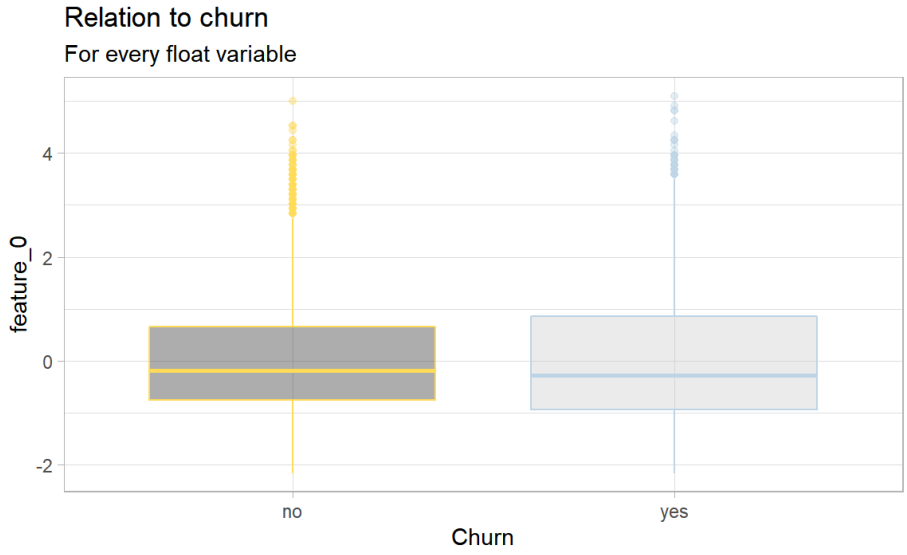
To further inspect the relation between churn and other variables we will create plots for each variable.

2.7.4 Float Variables

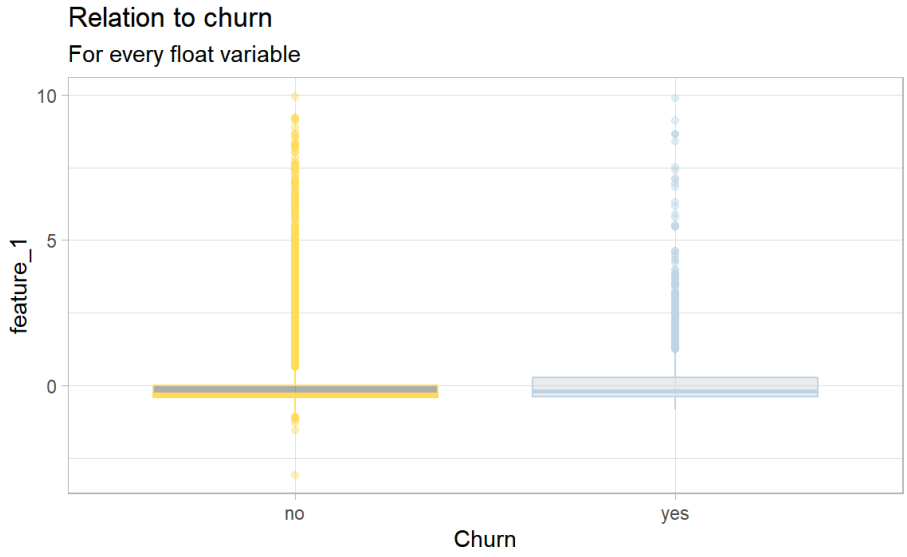
We will start with boxplots and histograms for the float variables.

```
# Credits for the code chunks goes to:
# https://www.kirenz.com/post/2021-02-17-r-classification-tidymodels/#evaluate-models

print_boxplot_float <- function(.y_var){

  # convert strings to variable
  y_var <- sym(.y_var)

  # unquote variables using {{}}
  explore_data %>%
    # Filter out some extreme outliers to better display the plots
    filter(
      feature_1 < 10,
      feature_4 < 8,
      feature_5 < 5,
      feature_6 < 10
      ) %>%
  ggplot(
    aes(x = churn,
        y = {{y_var}},
        fill = churn,
        color = churn
        )) +
    geom_boxplot(alpha=0.4) +
    labs(
      x = "Churn",
      title = "Relation to churn",
      subtitle = "For every float variable"
    ) +
    theme(legend.position = "none") +
    scale_color_manual(values = c("#ffdb58", "#bcd4e6")) +
    scale_fill_grey()
}

y_var_float <-
  explore_data %>%
  # select feature_0 - 6 since those contain float numbers
  select(1:7) %>%
  # obtain names
  variable.names()

map(y_var_float, print_boxplot_float)
```

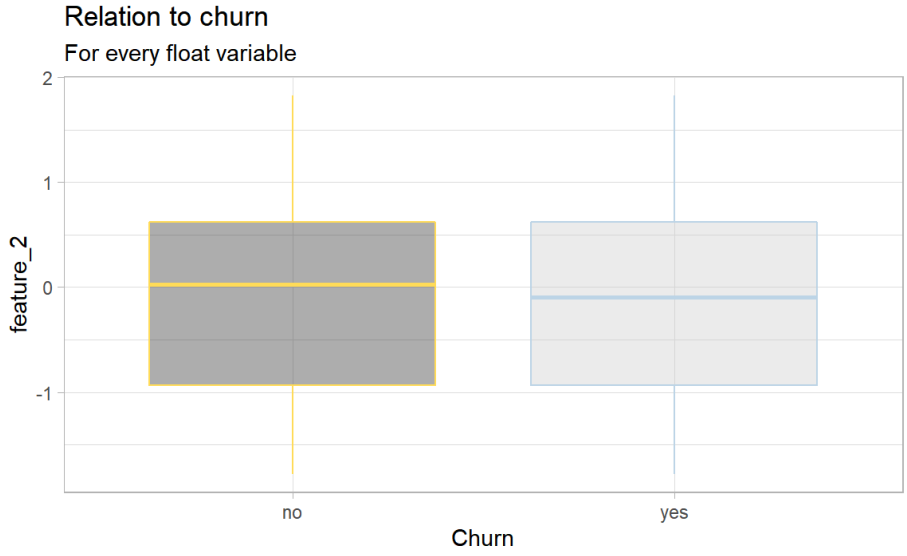
```
## [[1]]
```


Relation to churn
For every float variable
feature_0
4
2
0
-2
no
yes
Churn

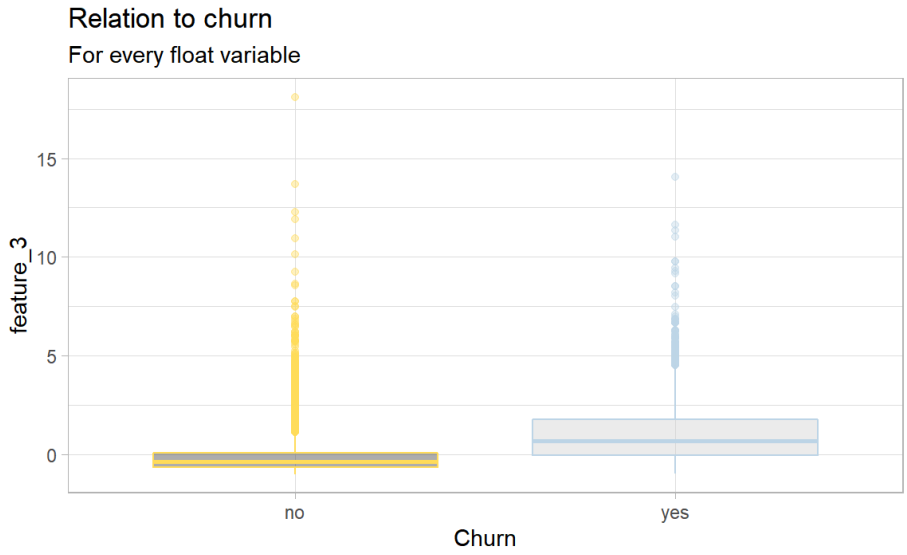
```
##
## [[2]]
```


Relation to churn
For every float variable
feature_1
10
5
0
no
yes
Churn

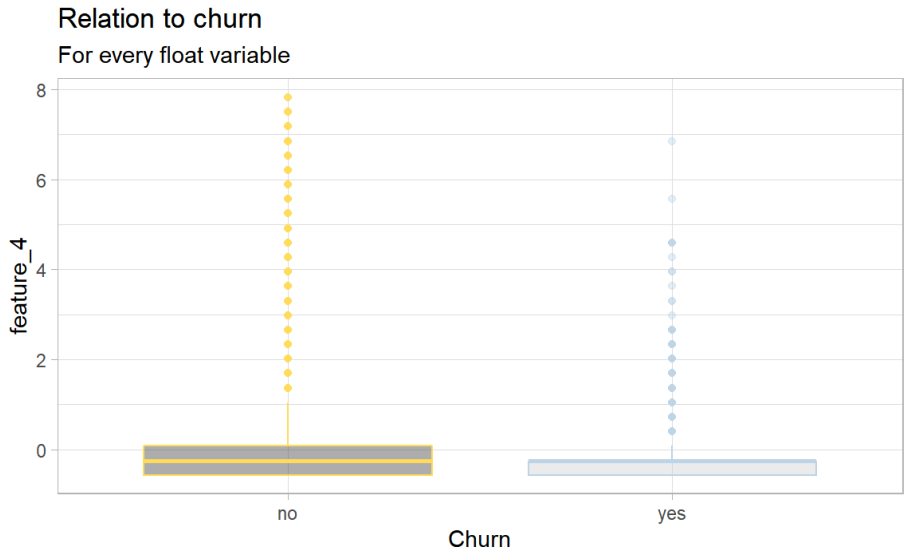
```
##
## [[3]]
```


Relation to churn
For every float variable
feature_2
2
1
0
-1
no
yes
Churn

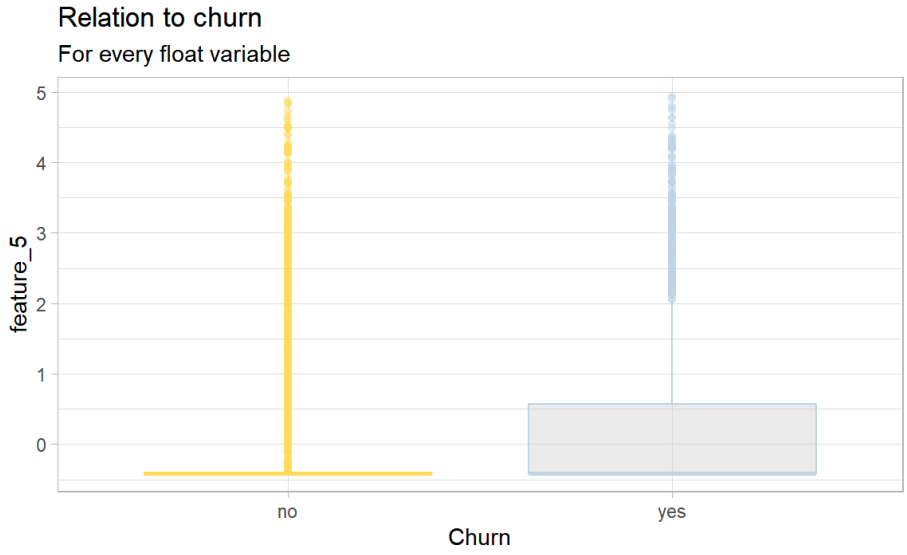
```
##
## [[4]]
```


Relation to churn
For every float variable
feature_3
15
10
5
0
no
yes
Churn

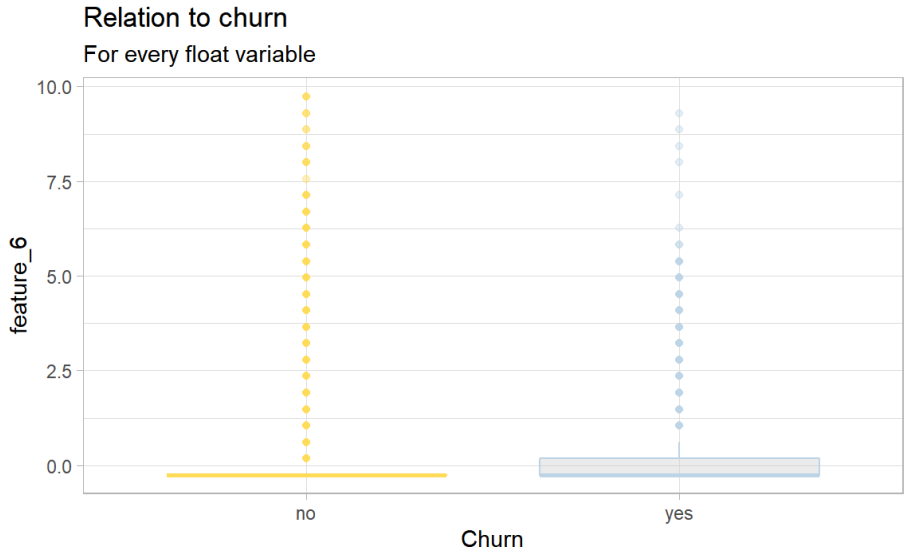
```
##
## [[5]]
```


Relation to churn
For every float variable
feature_4
8
6
4
2
0
no
yes
Churn

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

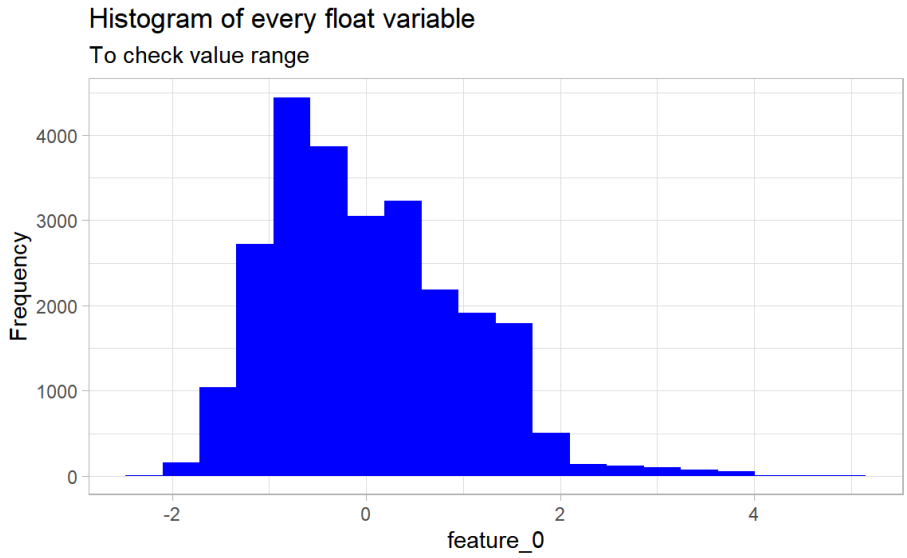

Relation to churn
For every float variable
feature_5
5
4
3
2
1
0
no
yes
Churn

```
##
## [[7]]
```

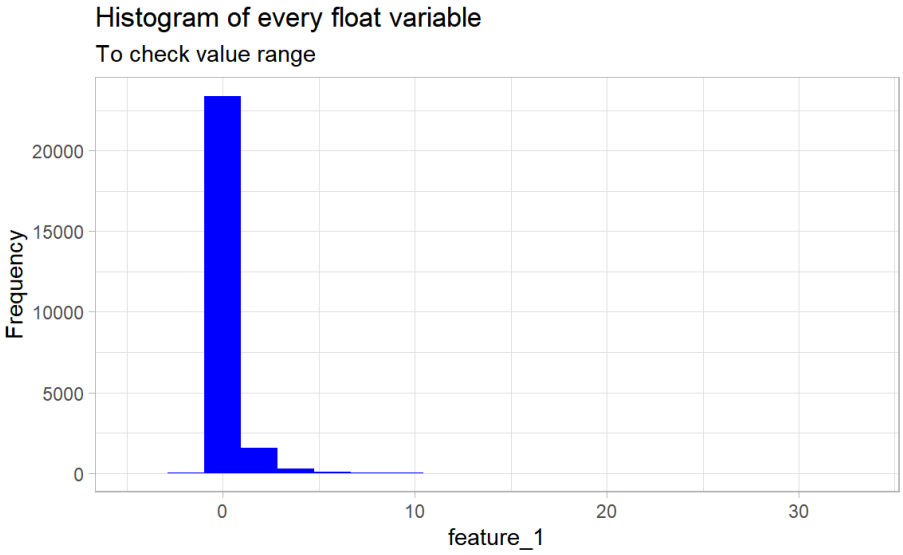

Relation to churn
For every float variable
feature_6
10.0
7.5
5.0
2.5
0.0
no
yes
Churn

```
# Credits for the code chunks goes to:
# https://www.kirenz.com/post/2021-02-17-r-classification-tidymodels/#evaluate-models


print_bar_float <- function(.x_var){

  # convert strings to variable
  x_var <- sym(.x_var)

  # unquote variables using {{}}
  explore_data %>%
  ggplot(
    aes(x = {{x_var}})) +
    geom_histogram(bins = 20, fill = "blue") +
    labs(
      y = "Frequency",
      title = "Histogram of every float variable",
      subtitle = "To check value range"
    )
}

x_var_float <-
  explore_data %>%
  # select feature_0 - 6 since those contain float numbers
  select(1:7) %>%
  # obtain names
  variable.names()

map(x_var_float, print_bar_float)
```

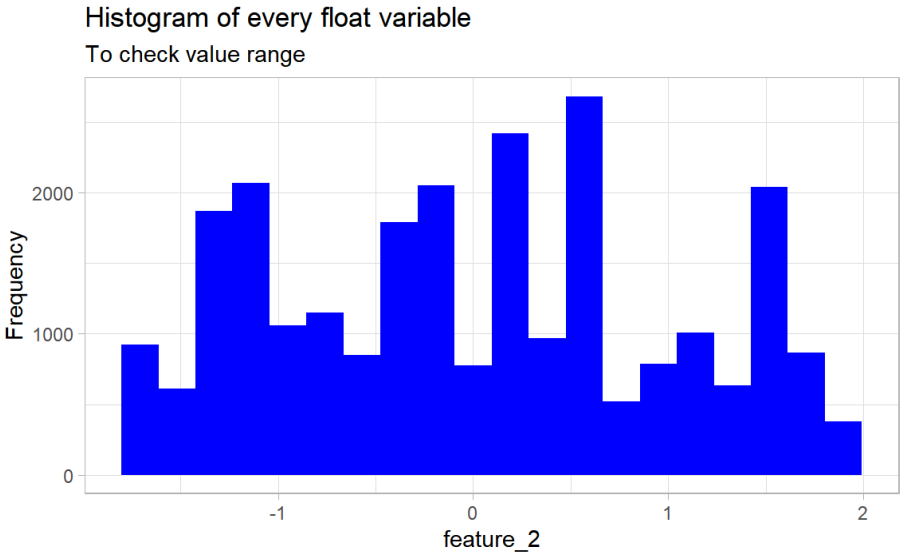
```
## [[1]]
```



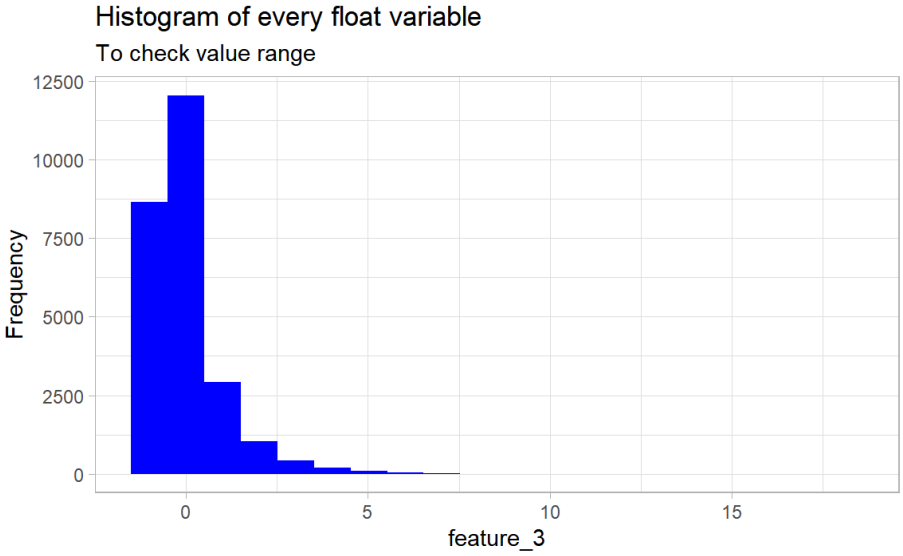

```
##
## [[2]]
```


Histogram of every float variable
To check value range
Frequency
20000
15000
10000
5000
0
0
10
20
30
feature_1

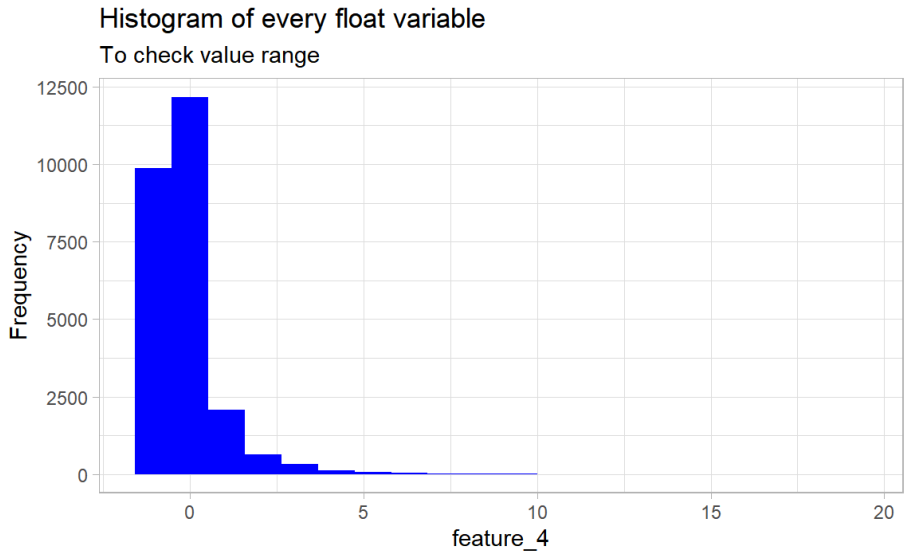
```
##
## [[3]]
```


Histogram of every float variable
To check value range
Frequency
2000
1000
0
-1
0
1
2
feature_2

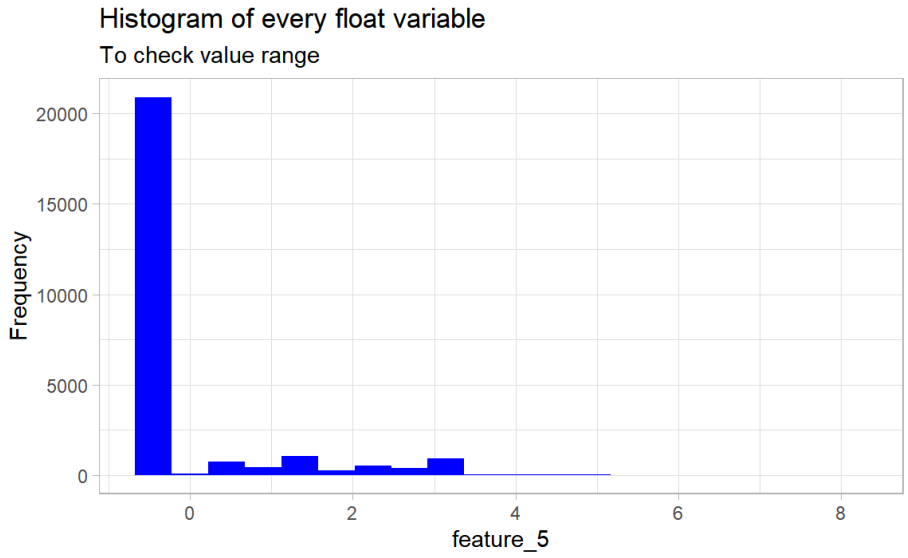
```
##
## [[4]]
```


Histogram of every float variable
To check value range
Frequency
12500
10000
7500
5000
2500
0
0
5
10
15
feature_3

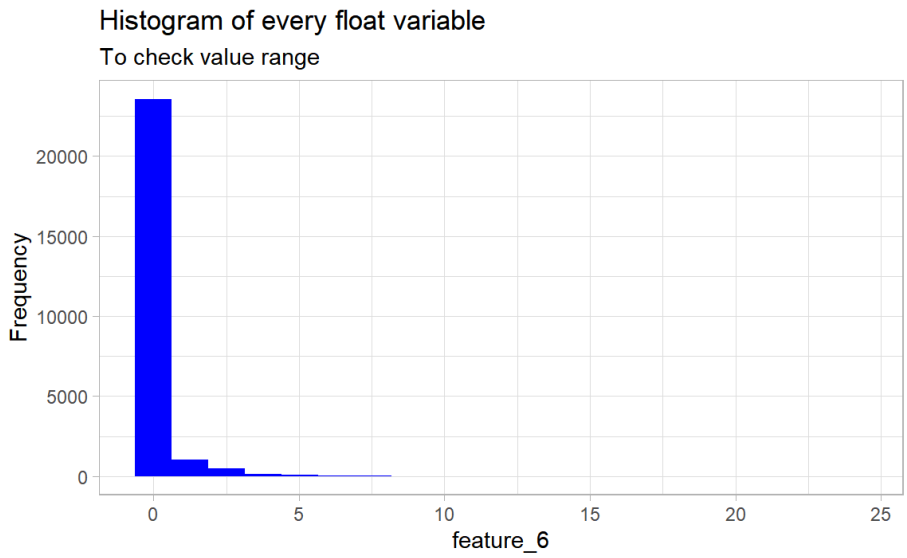
```
##
## [[5]]
```


Histogram of every float variable
To check value range
Frequency
12500
10000
7500
5000
2500
0
0
5
10
15
20
feature_4

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


Histogram of every float variable
To check value range
Frequency
20000
15000
10000
5000
0
0
2
4
6
8
feature_5

```
##
## [[7]]
```



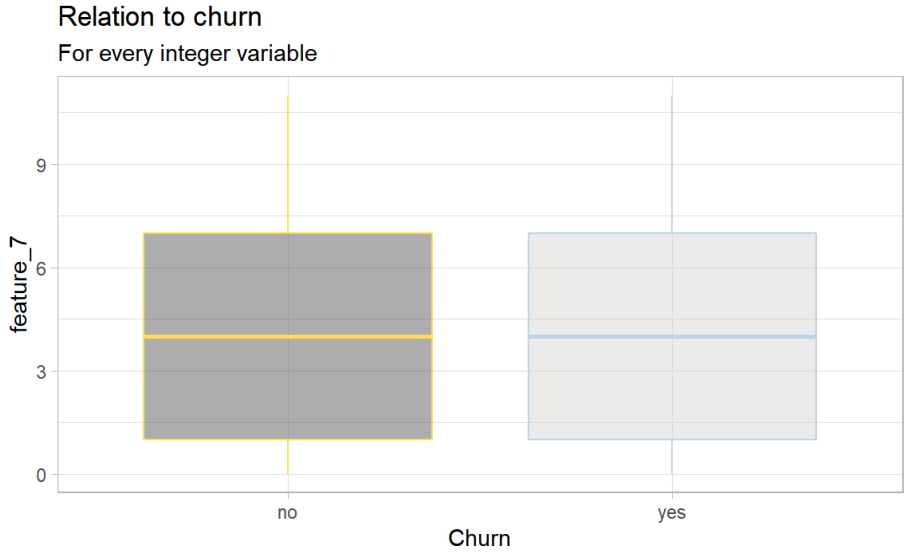
This confirms our previous insights from the **Data overview** and **Correlation to churn** section:

- Most of the variables do not have a significant influence on the churn label.
- feature_3 has a noticeable impact on churn.
- feature_5 & 6 can be interesting candidates for model training. This will be testes in the **Model** section
- There are alot of outliers present for almost every feature. This can be seen in the histograms as well as the barplots (Note that features 1, 4, 5 & 6 have been filtered to better display the respective boxplots).
- The scales for float variables are very different.

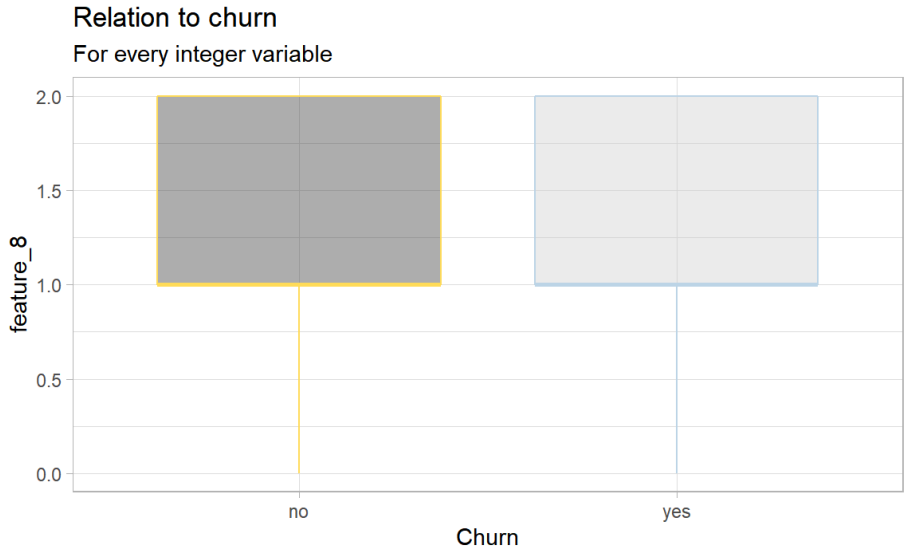
2.7.5 Integer Variables

Next up are the boxplots and histograms for the integer variables.

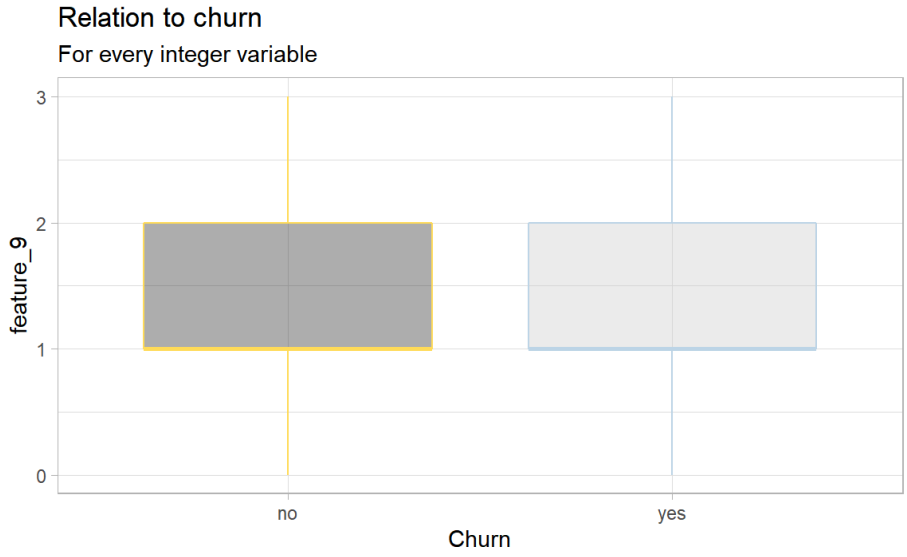
```
## [[1]]
```



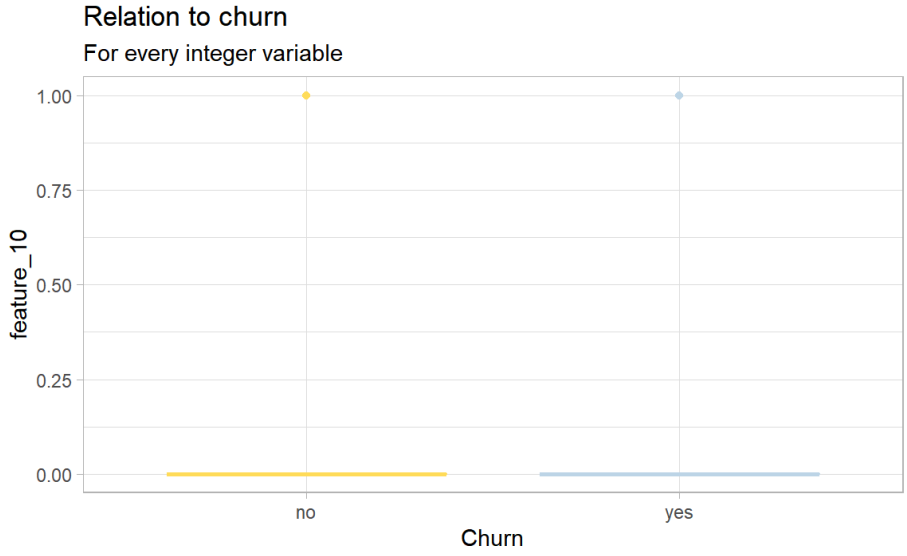

```
##
## [[2]]
```


Relation to churn
For every integer variable
feature_8
2.0
1.5
1.0
0.5
0.0
no
yes
Churn

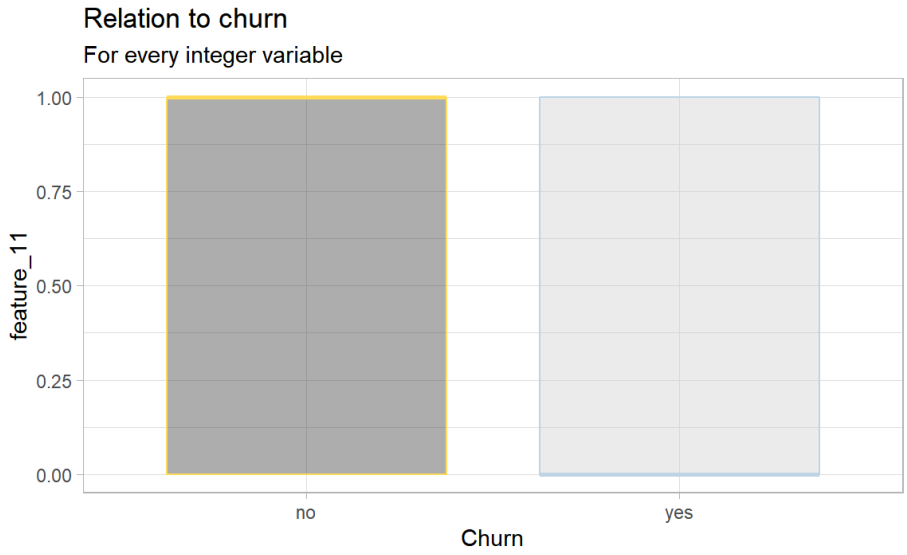
```
##
## [[3]]
```


Relation to churn
For every integer variable
feature_9
3
2
1
0
no
yes
Churn

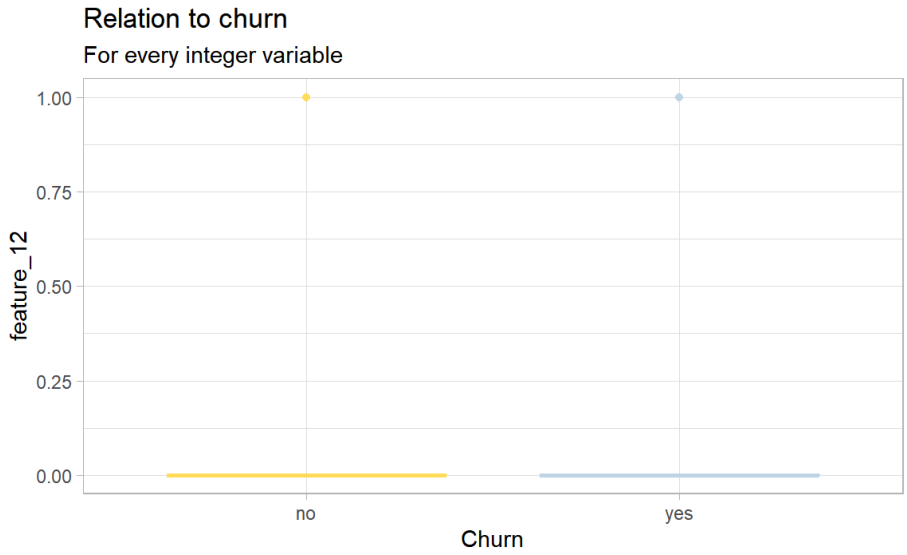
```
##
## [[4]]
```


Relation to churn
For every integer variable
feature_10
1.00
0.75
0.50
0.25
0.00
no
yes
Churn

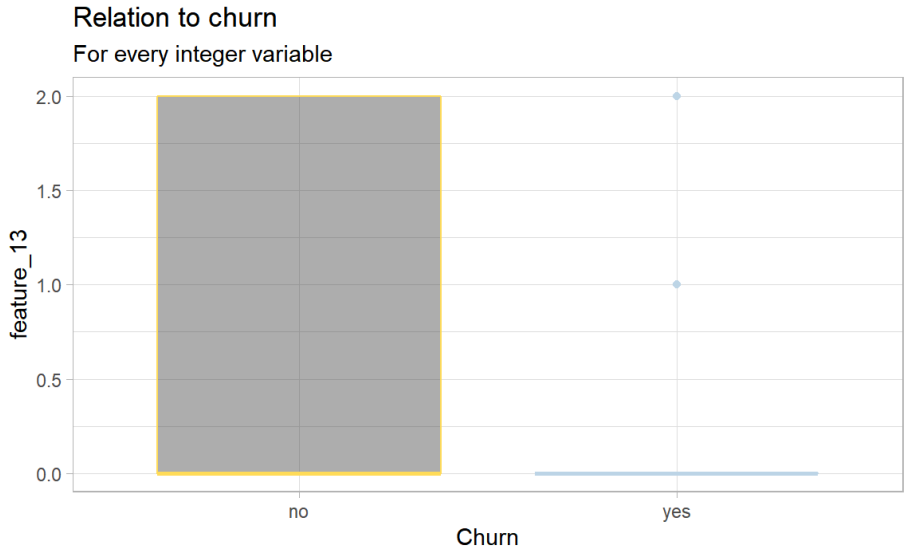
```
##
## [[5]]
```


Relation to churn
For every integer variable
1.00
0.75
0.50
0.25
0.00
feature_11
no
yes
Churn

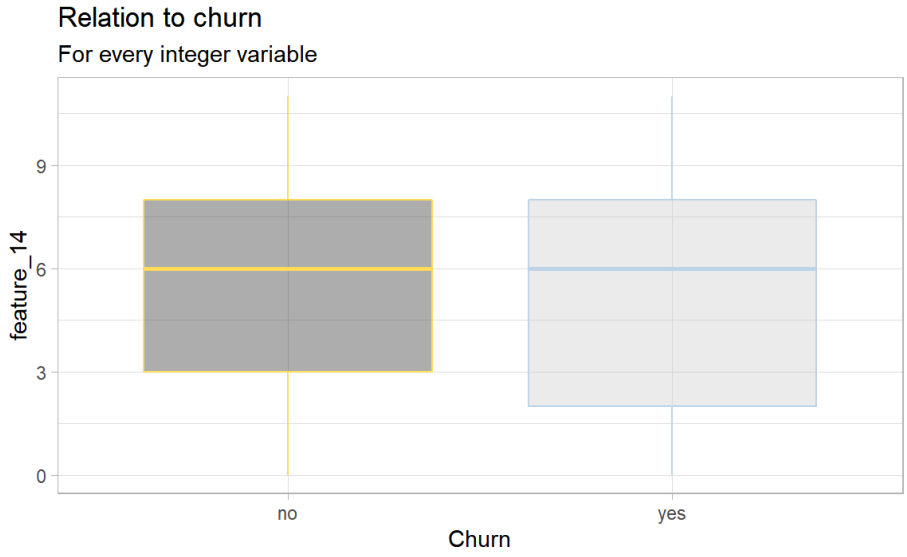
```
##
## [[6]]
```


Relation to churn
For every integer variable
1.00
0.75
0.50
0.25
0.00
feature_12
no
yes
Churn

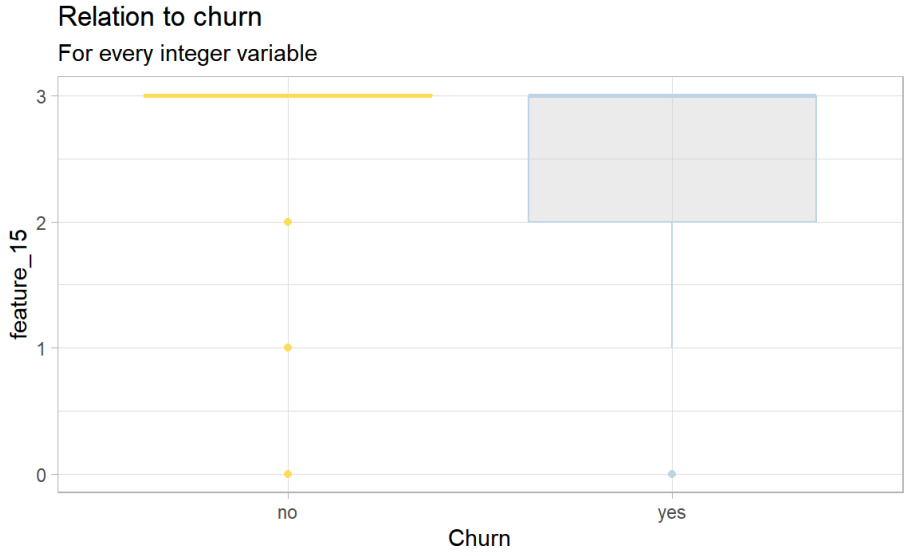
```
##
## [[7]]
```


Relation to churn
For every integer variable
feature_13
2.0
1.5
1.0
0.5
0.0
no
yes
Churn

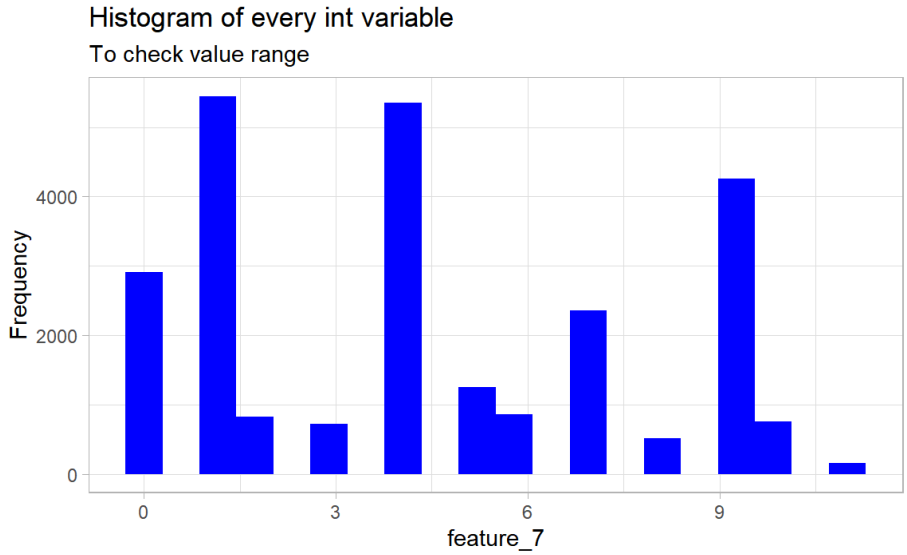
```
##
## [[8]]
```


Relation to churn
For every integer variable
feature_14
9
6
3
0
no
yes
Churn

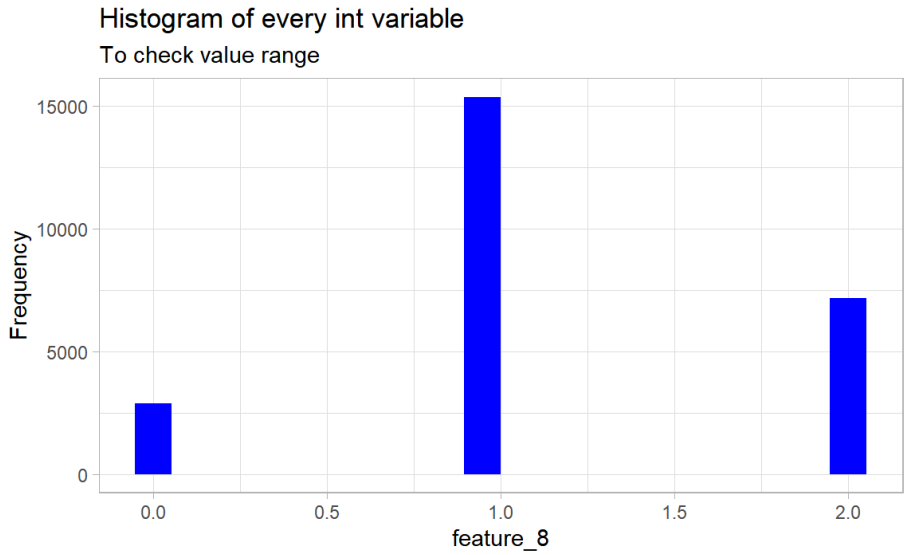
```
##
## [[9]]
```


Relation to churn
For every integer variable
feature_15
3
2
1
0
no
yes
Churn

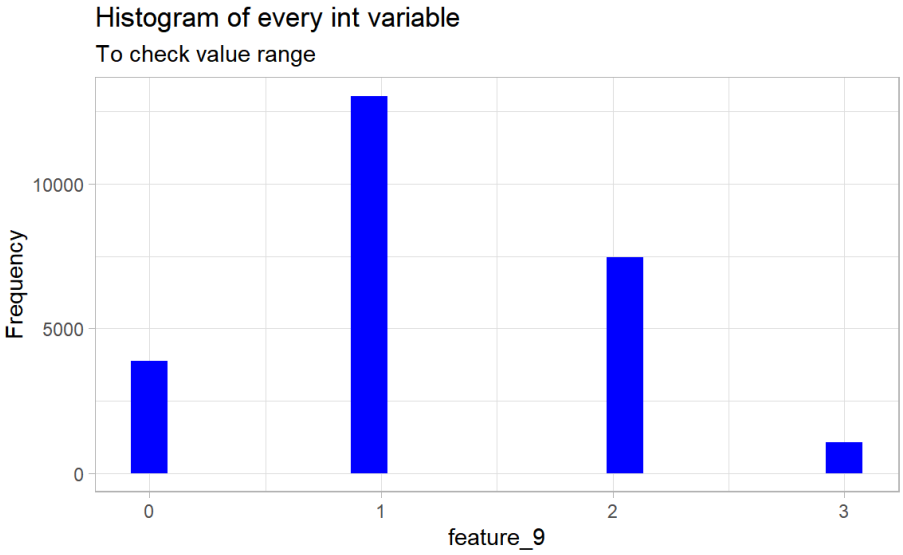
```
## [[1]]
```


Histogram of every int variable
To check value range
Frequency
4000
2000
0
0
3
6
9
feature_7

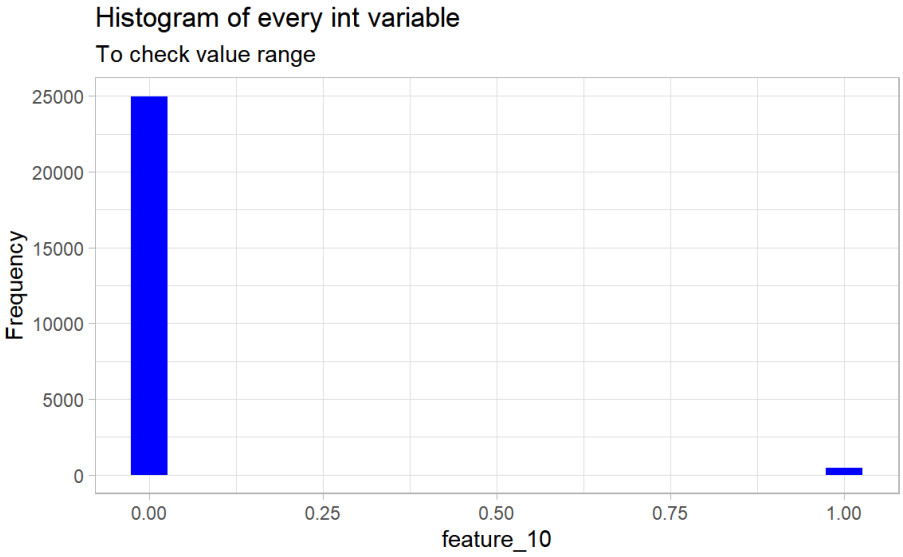
```
##
## [[2]]
```


Histogram of every int variable
To check value range
Frequency
15000
10000
5000
0
0.0
0.5
1.0
1.5
2.0
feature_8

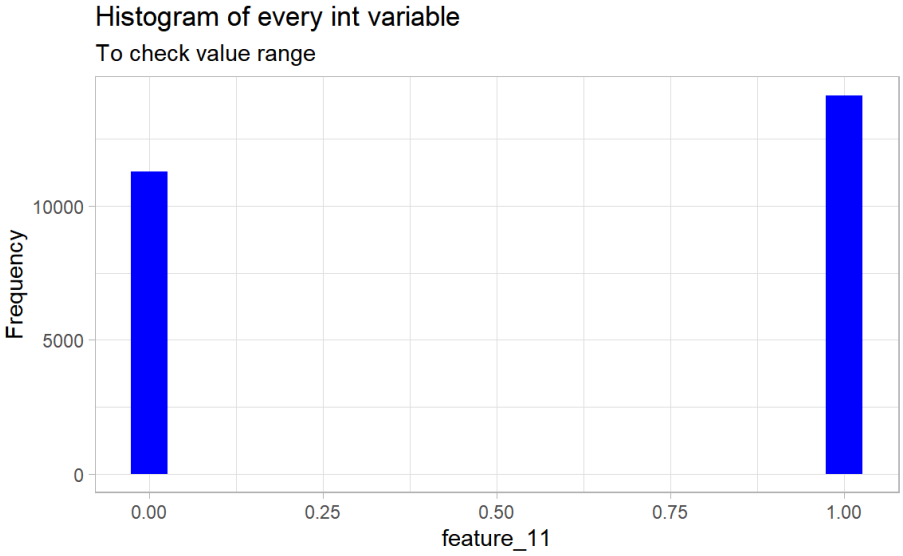
```
##
## [[3]]
```


Histogram of every int variable
To check value range
Frequency
10000
5000
0
0
1
2
3
feature_9

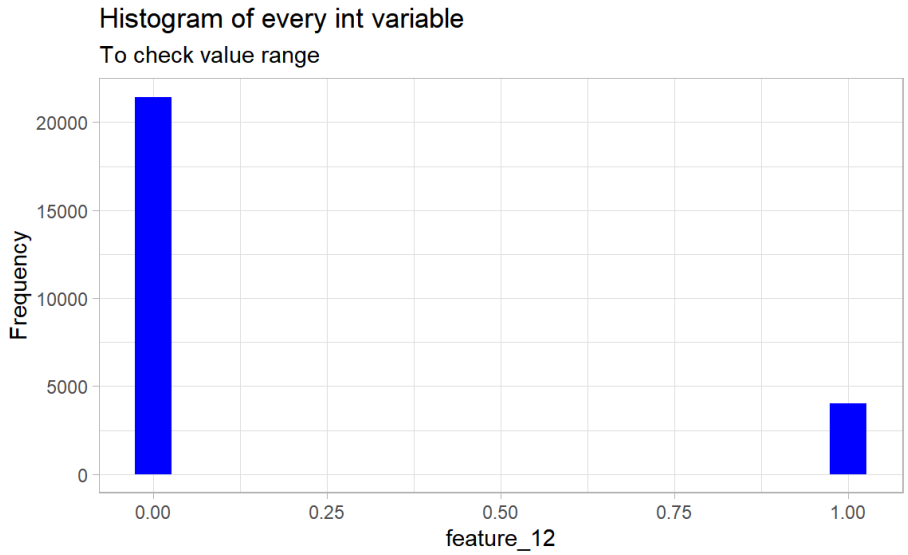
```
##
## [[4]]
```


Histogram of every int variable
To check value range
Frequency
25000
20000
15000
10000
5000
0
0.00
0.25
0.50
0.75
1.00
feature_10

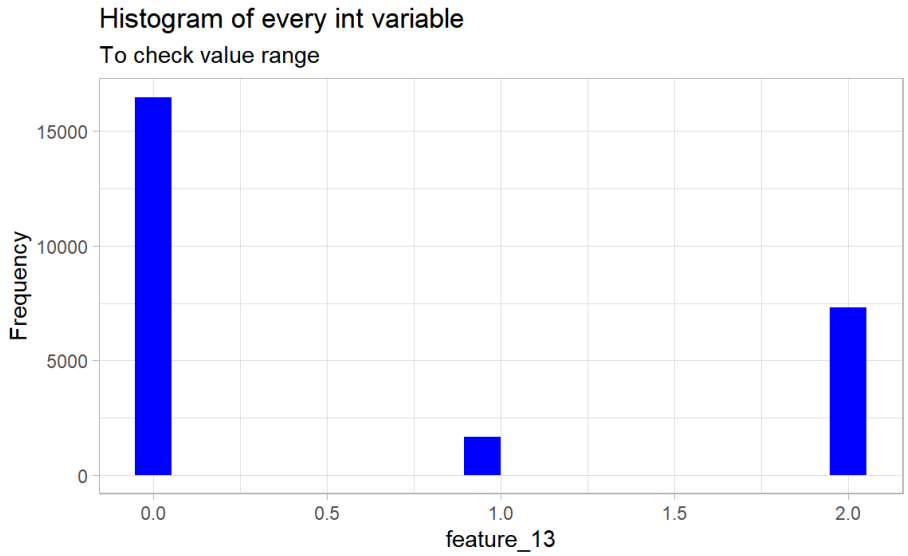
```
##
## [[5]]
```


Histogram of every int variable
To check value range
Frequency
10000
5000
0
0.00
0.25
0.50
0.75
1.00
feature_11

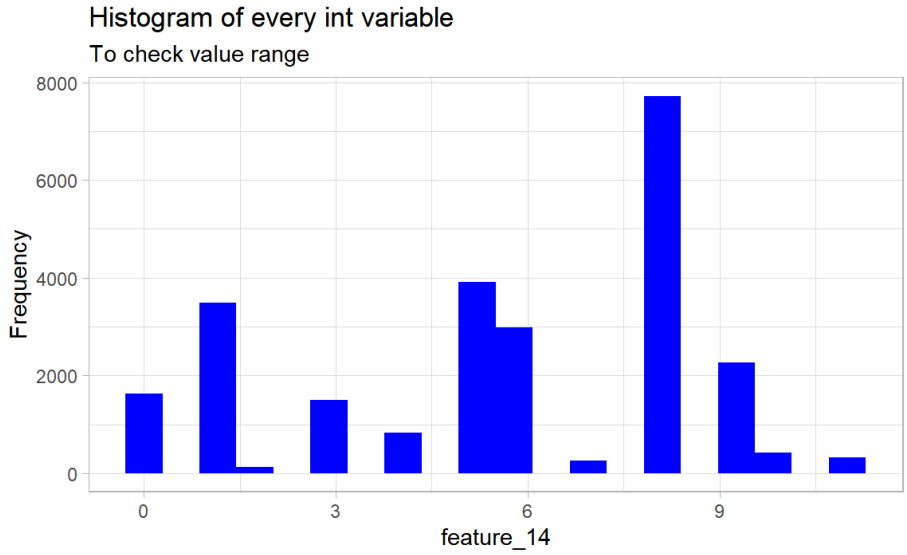
```
##
## [[6]]
```


Histogram of every int variable
To check value range
Frequency
20000
15000
10000
5000
0
0.00
0.25
0.50
0.75
1.00
feature_12

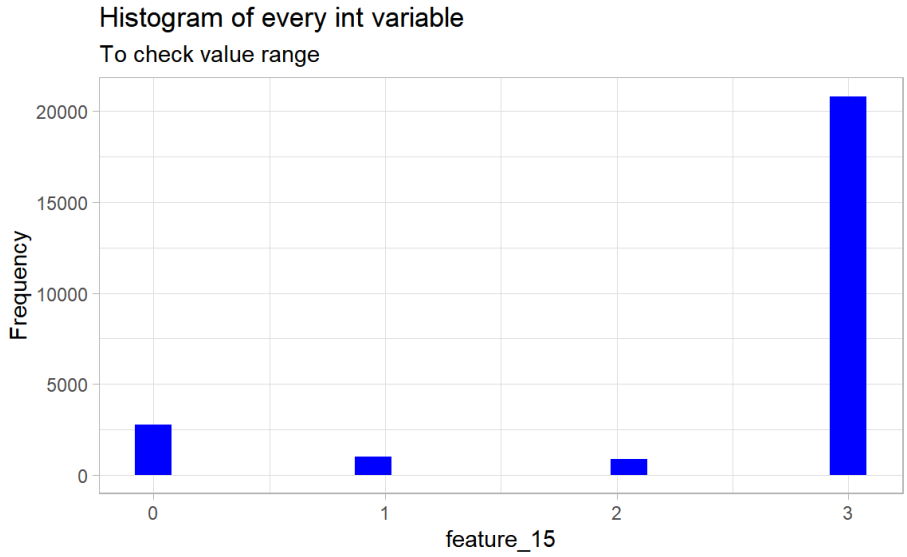
```
##
## [[7]]
```


Histogram of every int variable
To check value range
Frequency
15000
10000
5000
0
0.0
0.5
1.0
1.5
2.0
feature_13

```
##
## [[8]]
```




```
##
## [[9]]
```



Again, this is a confirmation of what we have seen before:

- feature 11 has a noticeable impact on the churn variable.
- feature 13 can also be an interesting candidate for model training. This will be tested in the **Model** section.
- feature 7 - 15 contain discrete integer values, in different levels. feature 10 - 12 contain binary values (0 and 1) for example.
- The scales for integer variables are also different (although not to the same extent as for the float variables).

As a conclusion, we expect the following features to have the most impact on our model training:

- feature 5
- feature 6
- feature 11
- feature 13

3 Data preparation

Before model training we will:

- create a validation set
- create a recipe to remove outliers and perform feature scaling

We will train our model with all available features in order to gain insights on potentially relevant features selected by the algorithm itself. This will be done using the vip() function after performing the last evaluation.

3.1 Validation set

For the validation set a k-fold crossvalidation with a set of 5 validation folds will be used. The validation set is used in order to check the models performance on the training dataset.

```
# Fix random generation, also see data split
set.seed(42)
# generate the cross validation set
cv_folds <-
  vfold_cv(train_data,
           v=5, # number of folds
           )
```

3.2 Data preprocessing recipe

First we create a recipe for our model that will apply the same steps to all data that we feed into our model.

```
# Create a recipe for our model
churn_rec <-
  recipe(
    # outcome ~ predictor
    churn ~ .,
    data = train_data) %>%
  # perform z-standardization: normalize the numeric variables to have a standard deviation of one
 and a mean of zero.
  step_normalize(all_numeric(), -all_outcomes()) %>%
  # removes numeric variables that have no variance.
  step_zv(all_numeric(), -all_outcomes()) %>%
  # remove predictor variables that have large correlations with other predictor variables.
  step_corr(all_predictors(), threshold = 0.7, method = "spearman")
```

These variables with their respective roles will be used by our recipe:

```
summary(churn_rec)
```

variable	type	role	source
<chr>	<chr>	<chr>	<chr>
feature_0	numeric	predictor	original
feature_1	numeric	predictor	original

variable	type	role	source
<chr>	<chr>	<chr>	<chr>
feature_2	numeric	predictor	original
feature_3	numeric	predictor	original
feature_4	numeric	predictor	original
feature_5	numeric	predictor	original
feature_6	numeric	predictor	original
feature_7	numeric	predictor	original
feature_8	numeric	predictor	original
feature_9	numeric	predictor	original

1-10 of 17 rows Previous **1** 2 Next

We check how the recipe will affect our training data

```
prepped_data <-
  churn_rec %>% # use the recipe object
  prep() %>% # apply recipe
  juice() # show the processed data


glimpse(prepped_data)
```

```
## Rows: 25,431
## Columns: 15
## $ feature_0  <dbl> -1.407411688, 0.470540284, 0.188847488, 1.785106664, 0.0010~
## $ feature_1  <dbl> 0.48091487, 1.71019034, -0.44476778, -0.28989333, -0.020645~
## $ feature_2  <dbl> -1.53996994, 0.74295494, 0.62280100, 0.38249312, -0.2182765~
## $ feature_3  <dbl> 0.71516152, -0.55801928, 0.74967546, -0.33176124, 1.1830171~
## $ feature_4  <dbl> -0.5622292, -0.5622292, -0.2434036, -0.2434036, -0.5622292,~
## $ feature_6  <dbl> -0.3056233, 0.2359254, -0.3056233, 4.0267660, -0.3056233, -~
## $ feature_7  <dbl> 0.5058304, -0.1052752, -0.1052752, -1.3274864, -1.3274864, ~
## $ feature_8  <dbl> 1.3713533, 1.3713533, -0.2774995, -0.2774995, -0.2774995, 1~
## $ feature_9  <dbl> 1.0339792, 1.0339792, 1.0339792, 2.3659849, -0.2980265, -0.~
## $ feature_10 <dbl> -0.1358728, -0.1358728, 7.3595332, -0.1358728, -0.1358728, ~
## $ feature_11 <dbl> -1.1179823, -1.1179823, -1.1179823, -1.1179823, 0.8944333, ~
## $ feature_12 <dbl> -0.4322727, -0.4322727, -0.4322727, -0.4322727, -0.4322727,~
## $ feature_13 <dbl> -0.7124594, -0.7124594, -0.7124594, -0.7124594, 1.5176538, ~
## $ feature_14 <dbl> -0.1760558, 0.8240803, -0.1760558, -0.1760558, 0.8240803, 0~
## $ churn      <fct> 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,~
```

The recipe is prepared such that the churn variable serves as the outcome which the model should predict. We can see that feature_5 has been removed. This happened by the step_corr() function in the recipe meaning feature_5 correlates strongly with another feature and is not providing additional useful information in model training.

4 Model - XGBoost

4.1 Model specification

We will set up the models with the needed specification.

```
xgb_spec <-
  # uses empirically best parameters, if none are specified
  boost_tree() %>%
  # model engine
  set_engine("xgboost") %>%
  # model mode
  set_mode("classification")

# show model specification
xgb_spec
```

```
## Boosted Tree Model Specification (classification)
##
## Computational engine: xgboost
```

4.2 Workflow

Next we create a workflow object to combine the recipe churn_rec with our model in **Model execution**.

```
# workflow pipeline
xgb_wflow <-
  workflow() %>%
  # specifiy our recipe
  add_recipe(churn_rec) %>%
  # and our model
  add_model(xgb_spec)

# show workflow
xgb_wflow
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_normalize()
## * step_zv()
## * step_corr()
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Computational engine: xgboost
```

4.3 Model execution

Finally we can execute our model and fit it to our training data. We save the predictions made so we can evaluate the performance metrics, especially the specificity.

```
# Create fit to the model and check with validation folds

xgb_res <-
  xgb_wflow %>%
  fit_resamples(
    # Estimate performance for specified metrics on all folds by fitting model to each resample whi
le holding out a portion from each resample to evaluate.
    resamples = cv_folds,
    # Define metrics
    metrics = metric_set(
              spec,
              accuracy,
              roc_auc,
              recall,
              precision
      ),
    # Save predictions
    control = control_resamples(save_pred = TRUE)
  )
```

4.4 Model evaluation

We can evaluate our metrics with collect_metrics().

The performance over all folds is:

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
accuracy	binary	0.8985885	5	0.0014934523	Preprocessor1_Model1

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
precision	binary	0.9209114	5	0.0014228347	Preprocessor1_Model1
recall	binary	0.9681675	5	0.0008099776	Preprocessor1_Model1
roc_auc	binary	0.9143461	5	0.0015210758	Preprocessor1_Model1
spec	binary	0.3780741	5	0.0082370377	Preprocessor1_Model1

5 rows

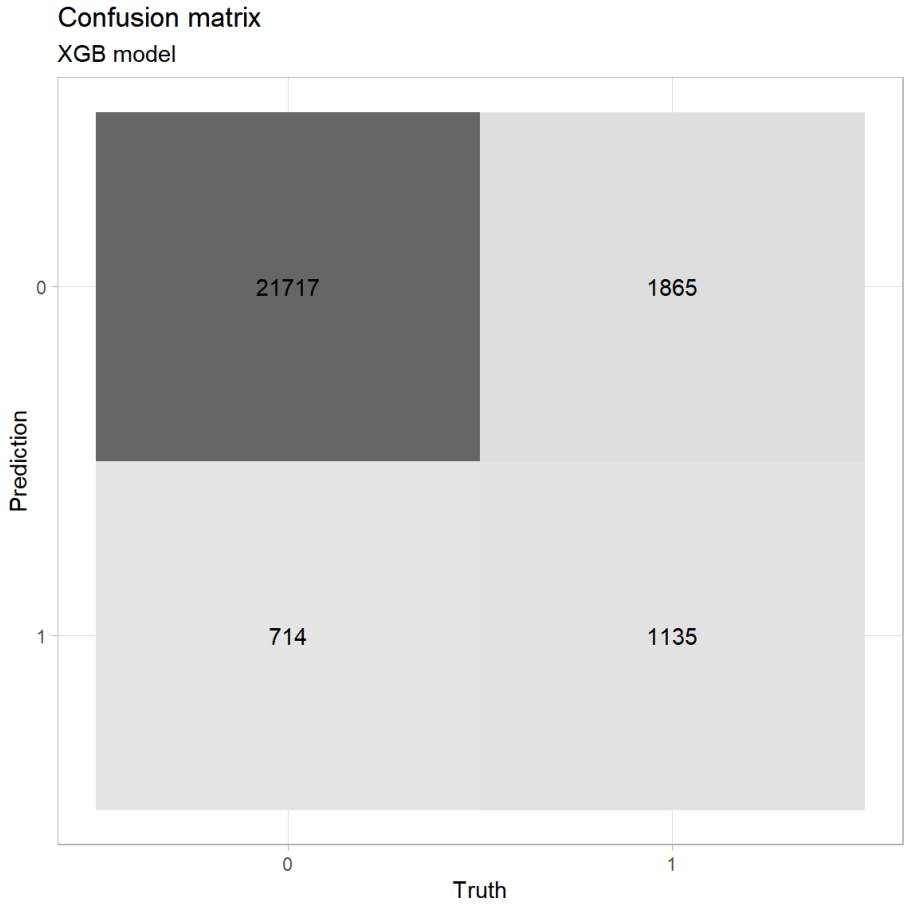
Our models performance overall seems to be good. The only problem is that the specificity is low.

We then collect our predictions. This will be needed to create the confusion matrix and ROC curve

```
xgb_pred <-
  xgb_res %>%
  collect_predictions()
```

Next we create the confusion matrix and plot it.

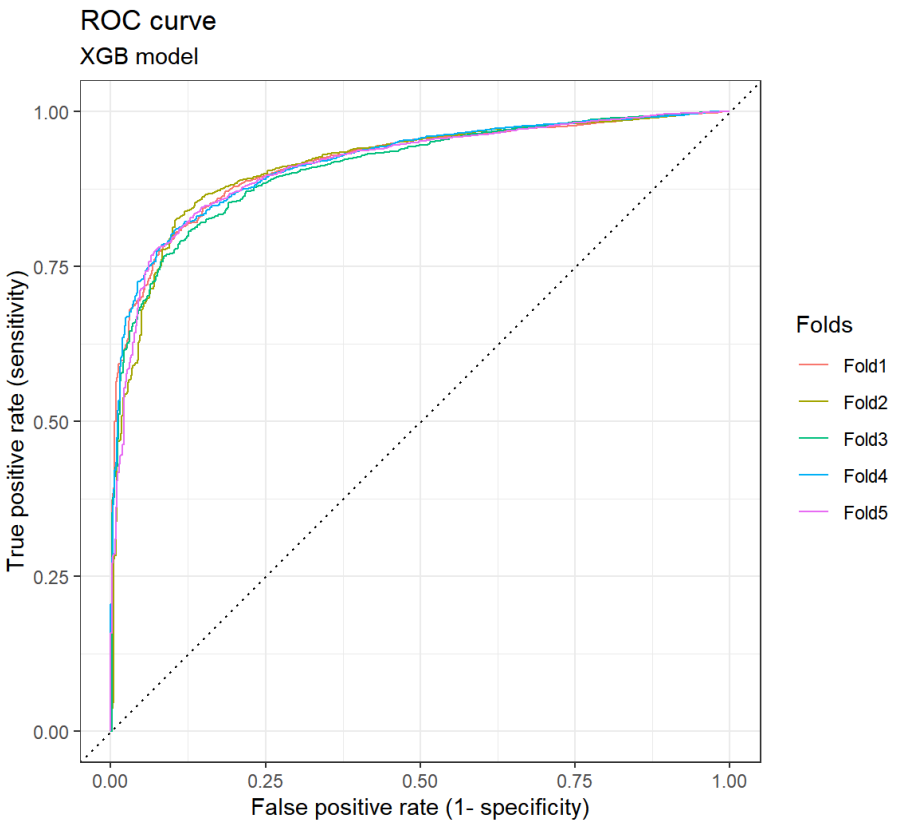
```
xgb_pred %>%
  conf_mat(churn, .pred_class) %>%
  autoplot(type = "heatmap") +
  labs(
    title = "Confusion matrix",
    subtitle = "XGB model",
    x = "Truth",
    y = "Prediction"
  )
```



Overall our model has a high rate of true positives but (as we have seen already) a low specificity - rate of true negatives.

We also visualize the ROC:

```
roc_xgb_pred <-
xgb_pred %>%
  group_by(id) %>% # id contains our folds
  roc_curve(churn, .pred_0)

roc_xgb_pred %>%
  autoplot() +
  labs(
    title = "ROC curve",
    subtitle = "XGB model",
    x = "False positive rate (1- specificity)",
    y = "True positive rate (sensitivity)",
    color = "Folds")
```



The ROC curve looks really good, but we need to remember that the rate of predicting true negatives (specificity) is still low.

In order to find a better model we will train and evaluate some other classifying algorithms and compare all models.

5 Additional model training

We will use the following algorithms

- Logistic regression
- Random forest
- K-nearest neighbor

We repeat all our steps from above to create specifications, workflows and train the models.

```
# Logistic regression
log_spec <-
  # will use best empirical hyperparamters
  logistic_reg() %>%
  set_engine(engine = "glm") %>%
  set_mode("classification")

# Random forest
rf_spec <-
  # will use best empirical hyperparamters
  rand_forest() %>%
  set_engine("ranger") %>%
  set_mode("classification")

# K-nearest neighbor
knn_spec <-
  # will use best empirical hyperparamters
  nearest_neighbor(neighbors = 4) %>% # can be adjusted
  set_engine("kknn") %>%
  set_mode("classification")
```

```
# Logistic regression
log_wflow <-
 workflow() %>%
 add_recipe(churn_rec) %>%
 add_model(log_spec)

# Random forest
rf_wflow <-
 workflow() %>%
 add_recipe(churn_rec) %>%
 add_model(rf_spec)

# K-nearest neighbor
knn_wflow <-
 workflow() %>%
 add_recipe(churn_rec) %>%
 add_model(knn_spec)
```

```
# Logistic regression
log_res <-
  log_wflow %>%
  fit_resamples(
    resamples = cv_folds,
    # Define metrics
    metrics = metric_set(
              spec,
              accuracy,
              roc_auc,
              recall,
              precision
      ),
    # Save predictions
    control = control_resamples(save_pred = TRUE)
  )


# Random forest
rf_res <-
  rf_wflow %>%
  fit_resamples(
    resamples = cv_folds,
    # Define metrics
    metrics = metric_set(
      spec,
      accuracy,
      roc_auc,
      recall,
      precision
      ),
    # Save predictions
    control = control_resamples(save_pred = TRUE)
  )

# K-nearest neighbor
knn_res <-
  knn_wflow %>%
  fit_resamples(
    resamples = cv_folds,
    # Define metrics
    metrics = metric_set(
              spec,
              accuracy,
              roc_auc,
              recall,
              precision
      ),
    # Save predictions
    control = control_resamples(save_pred = TRUE)
  )
```

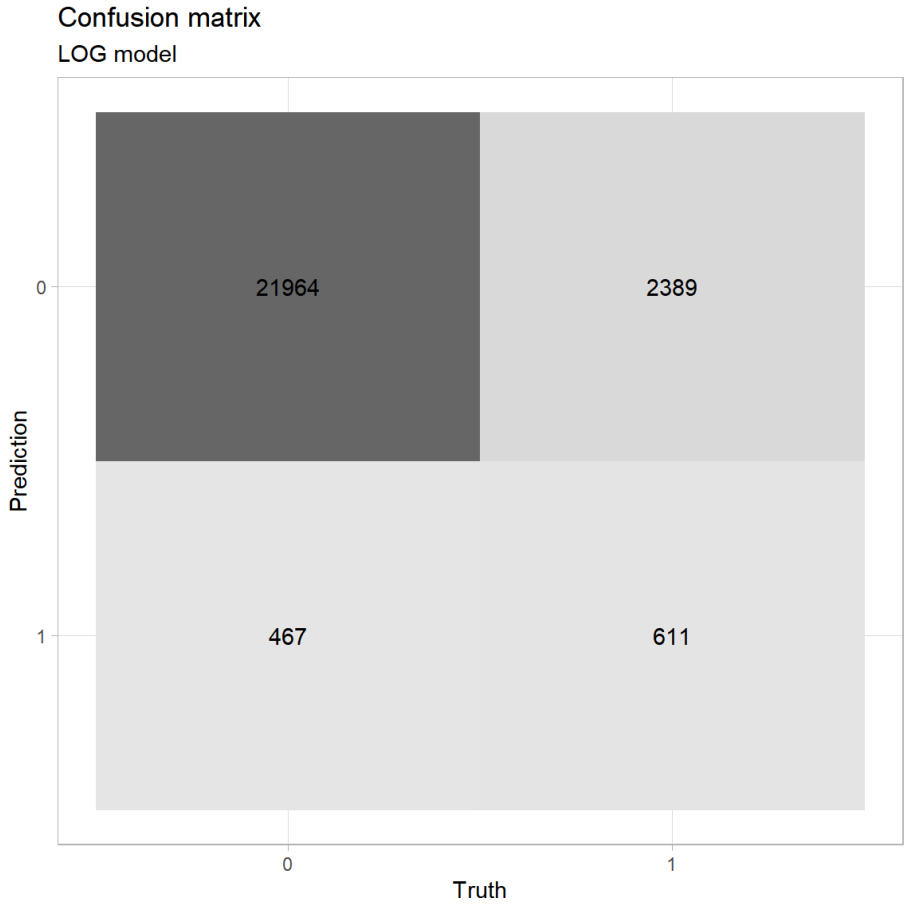
5.1 Model comparison

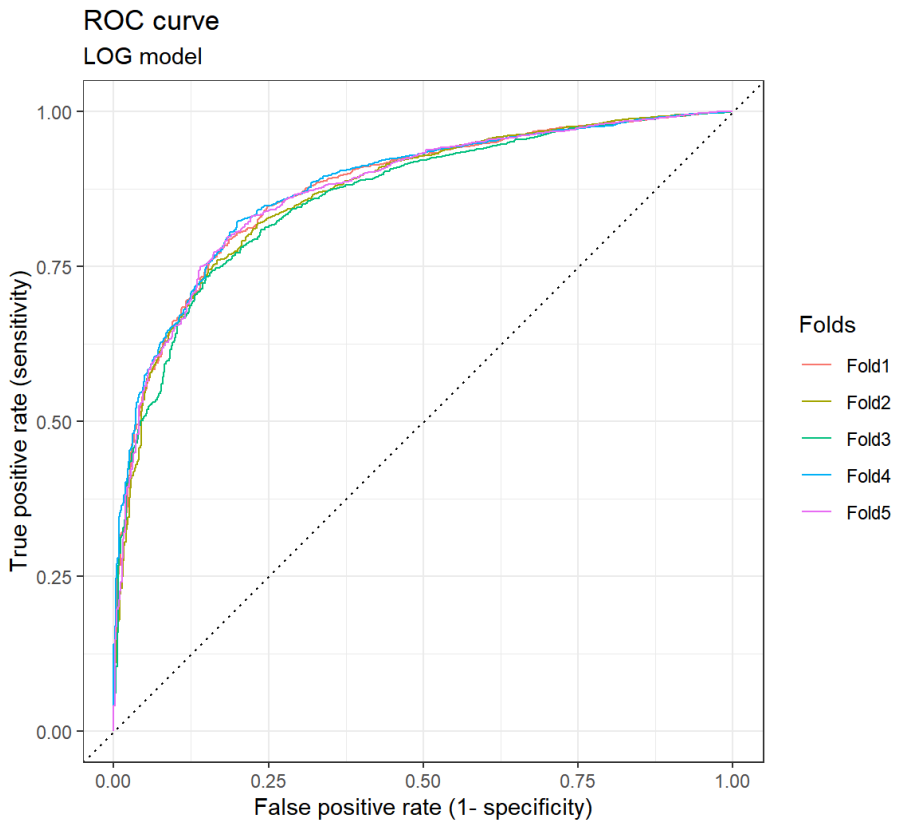
We compare the models by first looking at the different metrics and the confusion matrices.

- Logistic regression

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
accuracy	binary	0.8876962	5	0.001998861	Preprocessor1_Model1
precision	binary	0.9019031	5	0.002145901	Preprocessor1_Model1
recall	binary	0.9791822	5	0.001009754	Preprocessor1_Model1
roc_auc	binary	0.8704290	5	0.002875341	Preprocessor1_Model1
spec	binary	0.2038449	5	0.005455154	Preprocessor1_Model1

5 rows

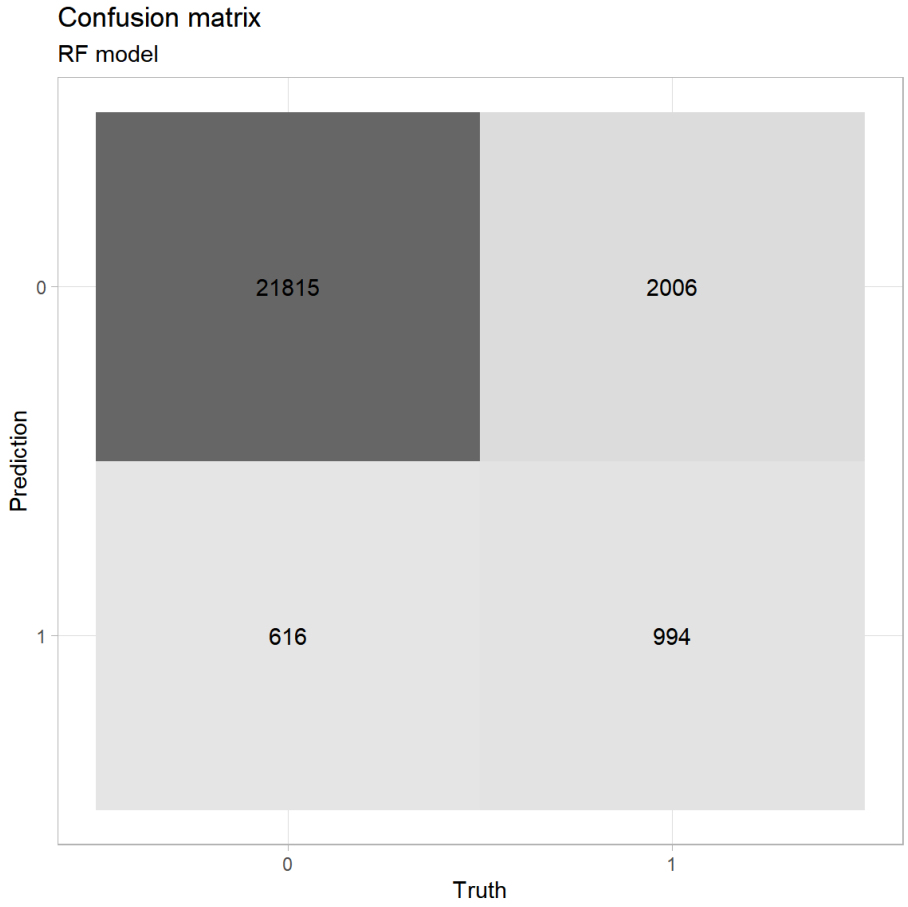


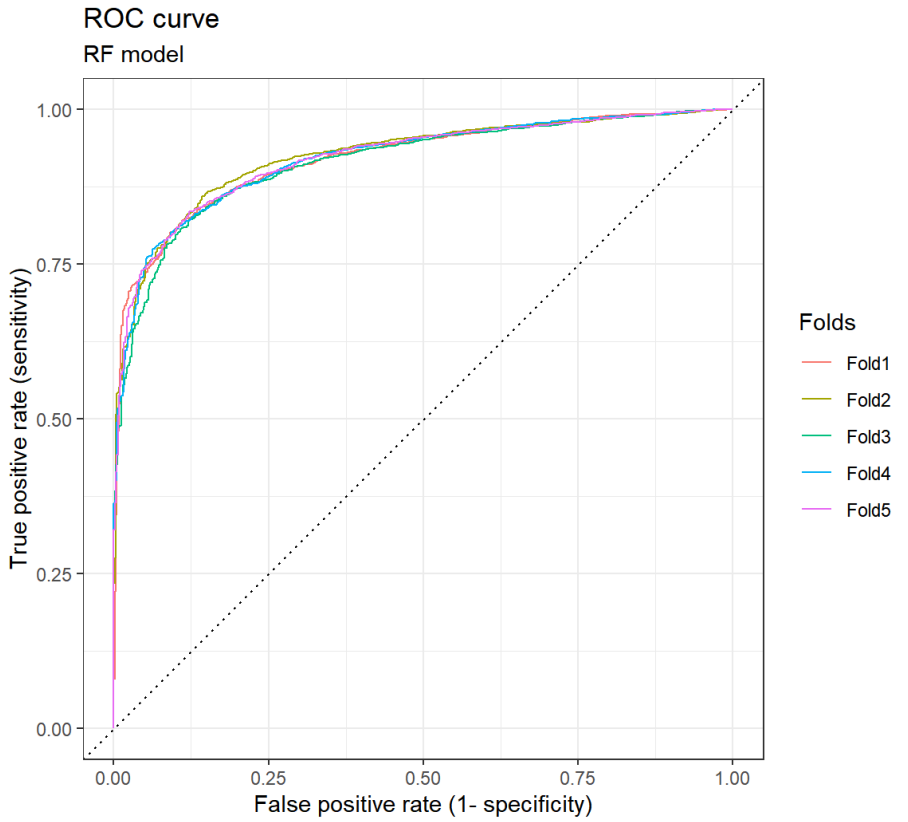


- Random forest

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
accuracy	binary	0.8968976	5	0.0009338776	Preprocessor1_Model1
precision	binary	0.9157923	5	0.0017306180	Preprocessor1_Model1
recall	binary	0.9725462	5	0.0009274773	Preprocessor1_Model1
roc_auc	binary	0.9198774	5	0.0015712604	Preprocessor1_Model1
spec	binary	0.3313702	5	0.0061078767	Preprocessor1_Model1

5 rows



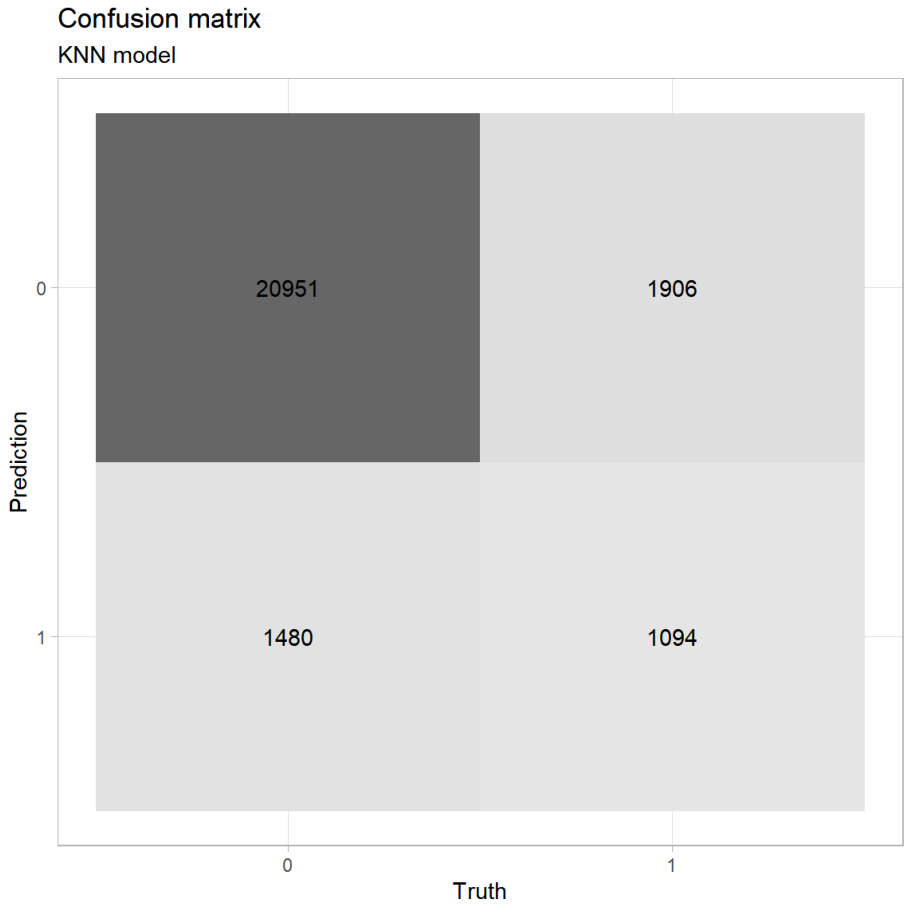


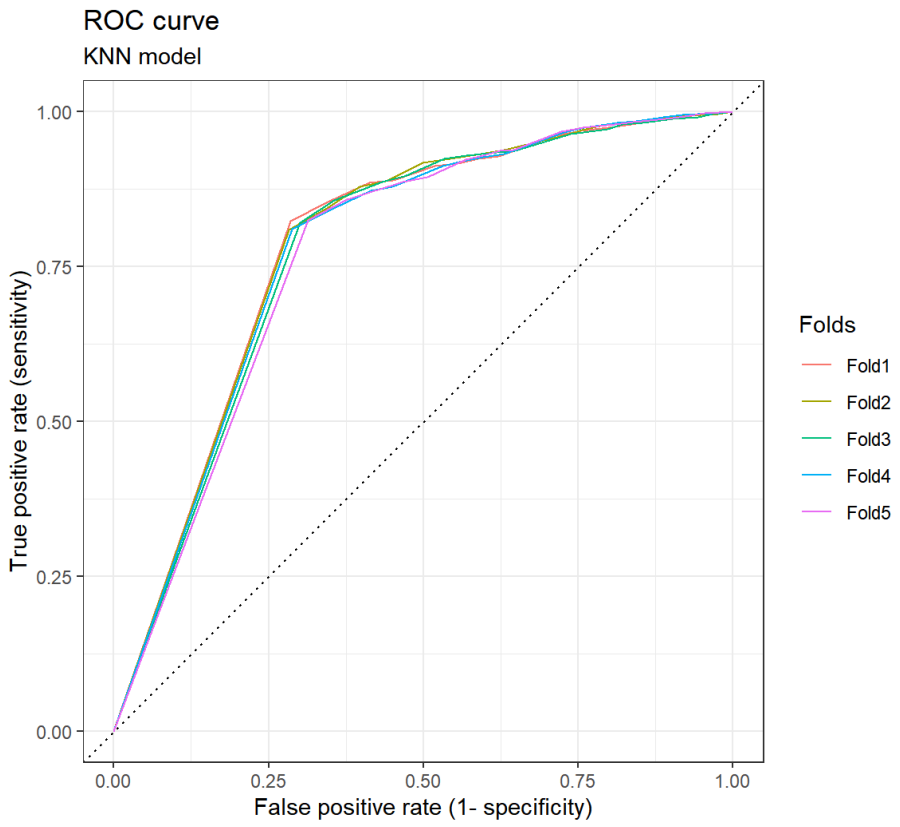
- K-nearest neighbor

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
accuracy	binary	0.8668557	5	0.002258094	Preprocessor1_Model1
precision	binary	0.9166020	5	0.001228233	Preprocessor1_Model1
recall	binary	0.9340093	5	0.002199318	Preprocessor1_Model1
roc_auc	binary	0.7789646	5	0.002451022	Preprocessor1_Model1
spec	binary	0.3643629	5	0.005101067	Preprocessor1_Model1

5 rows





We can see that our xgboost classification algorithm performed the best out of all models so far, considering specificity and the confusion matrix.

We will use that model and fine tune it in order to gain a better specificity and thus a higher prediction for churning.

6 Model fine tuning

To adjust our imbalanced data, we will use the synthetic minority oversampling technique smote().

```
smote_data <-
  # Generate more samples of the minority label (churn) while also decreasing samples of the majori
ty label (no churn)
  smote(churn ~., train_data, perc.over = 3, perc.under = 2)

table(smote_data$churn)
```

```
##
##     0     1
## 18000 12000
```

The data is still imbalanced, but not as much as before.

We will also need to create a new evaluation set with the adjusted dataset.

```
# Fix random generation, also see data split
set.seed(42)
# generate the cross validation set
cv_folds <-
  vfold_cv(smote_data,
           v=5, # number of folds
           )
```

Now we will perform hyperparameter tuning on our xgboost algorithm to select the best values to maximize the specificity.

6.1 Model specification

We have to specify the hyperparameters that we want to tune. We also adjust for the rest of the imbalance by using scale_pos_weight.

```
xgb_tuned_spec <-
  boost_tree(
      mtry = tune(), # number (or proportion) of predictors
      trees = 100, # number of decision trees.
      min_n = tune(), # minimum number of data points in a node required to split
      tree_depth = tune(), # maximum splits (depth of decision trees)
      learn_rate = tune(), # adaption rate over iterations
      loss_reduction = tune(), # reduction in loss required to split
      sample_size = tune(), # number (or proportion) of data exposed to fitting
  ) %>%
  # model engine
  # specifiy scale_pos_weight to account for the imbalanced dataset
  set_engine("xgboost", scale_pos_weight = tune()) %>%
  # model mode
  set_mode("classification")

# show model specification
xgb_tuned_spec
```

```
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = 100
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Engine-Specific Arguments:
##   scale_pos_weight = tune()
##
## Computational engine: xgboost
```

6.2 Workflow

Next we create a new workflow object.

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_normalize()
## * step_zv()
## * step_corr()
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = 100
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Engine-Specific Arguments:
##   scale_pos_weight = tune()
##
## Computational engine: xgboost
```

6.3 Grid search

In order to tune the parameters we first set up possible values for them. For efficiency reasons we use grid_latin_hypercube().

For more information see:

https://htmlpreview.github.io/?https://github.com/kirenz/tidymodels-in-r/blob/main/05-tidymodels-xgboost-tuning.html (https://htmlpreview.github.io/?https://github.com/kirenz/tidymodels-in-r/blob/main/05-tidymodels-xgboost-tuning.html)

```
xgb_grid <- grid_latin_hypercube(
  scale_pos_weight(),
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(), # needs to be a proportion
  finalize(mtry(), smote_data), # different approach since mtry depends on number of predictors in
 data
  learn_rate(),
  size = 40
)
```

6.4 Hyperparameter tuning

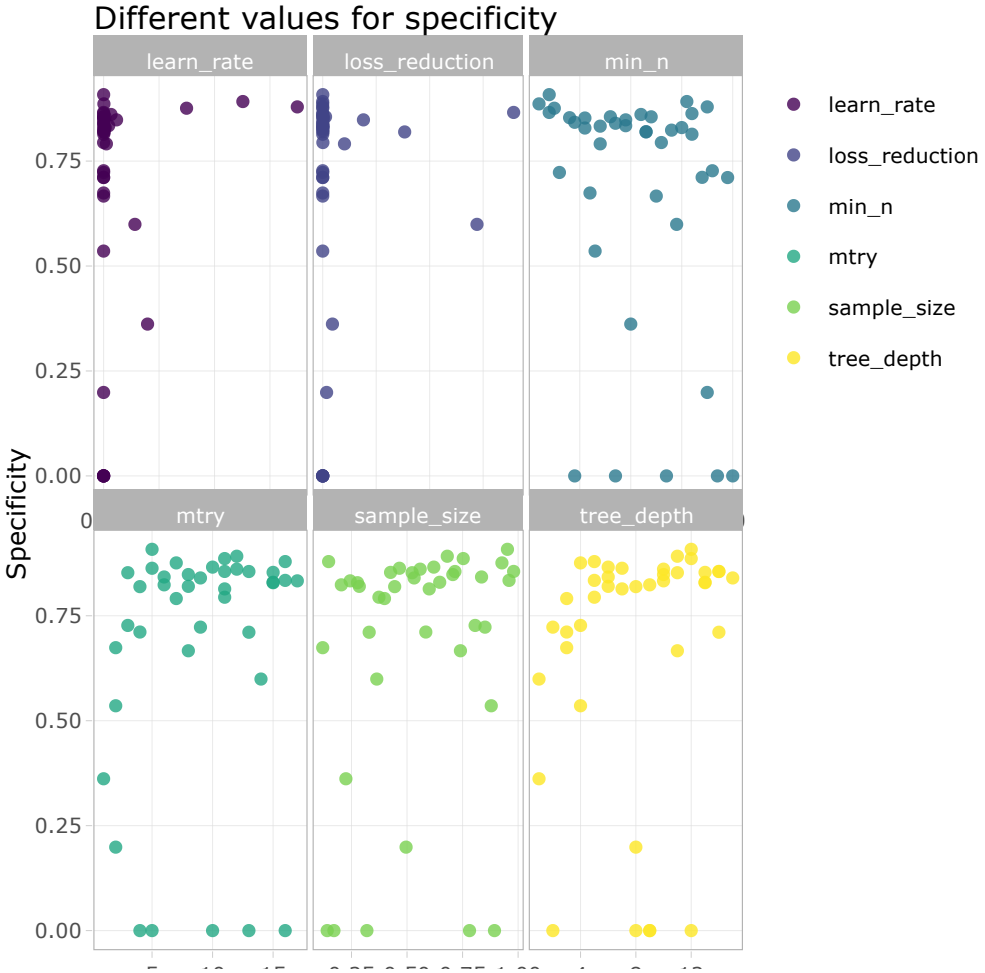
The tune_grid function computes the performance metrics for us, which we defined earlier.

```
# Seeded to make sure we can reproduce the tuning
set.seed(42)

xgb_tuned_res <- tune_grid(
  xgb_tuned_wflow,
  resamples = cv_folds,
  grid = xgb_grid,
  # save the predictions to visualize the data
  metrics = metric_set(
              spec,
              accuracy,
              roc_auc,
              recall,
              precision
      ),
  control = control_grid(save_pred = TRUE)
)

metrics <-
xgb_tuned_res %>%
  collect_metrics(summarize = TRUE)
```

6.5 Explore results

We visualize the specificty for every hyperparameter which gives us a first impression of how well specific values perform.

```
# Credits for the code chunk goes to:
# https://htmlpreview.github.io/?https://github.com/kirenz/tidymodels-in-r/blob/main/05-tidymodels-xgboost-tuning.html

# Pivot xgb_tuned_res to longer data format while only filtering for spec
xgb_res_spec <-
  xgb_tuned_res %>%
  collect_metrics() %>%
  filter(.metric == "spec") %>%
  select(mean, mtry:sample_size) %>%
  pivot_longer(mtry:sample_size,
               values_to = "value",
               names_to = "parameter"
  )
spec_tuning_comp <-
# show specificity side by side for every hyperparameter
xgb_res_spec %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) +
  facet_wrap(~parameter, scales = "free_x") +
  scale_color_viridis_d()+
  labs(x = NULL,
       y = "Specificity",
       title = "Different values for specificity",
       subtitle = "For each hyperparameter after tuning")

ggplotly(spec_tuning_comp)
```



5 10 15 0.25 0.50 0.75 1.00 4 8 12

6.6 Finding the best parameters

```
show_best(xgb_tuned_res,
          "spec")
```

...	m...	tree_de...	learn_rate	loss_reduction	sample_s...	scale_pos_wei...	.met...	.estimator
<int>	<int>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
5	4	12	1.412341e-05	5.230322e-03	0.9520000	0.8085457	spec	binary
12	31	11	5.023160e-02	2.386869e-09	0.6811584	1.0461585	spec	binary
11	2	12	1.047533e-04	5.184604e-06	0.7529731	1.1794240	spec	binary
16	35	5	6.986843e-02	1.251271e-06	0.1469394	0.8394978	spec	binary
7	5	4	2.992083e-02	7.855803e-05	0.9271984	0.9126205	spec	binary

5 rows | 1-9 of 13 columns

```
best_spec <-
  select_best(
    xgb_tuned_res,
    "spec"
    )
best_spec
```

...	m...	tree_de...	learn_rate	loss_reduction	sample_s...	scale_pos_wei...	.config
<int>	<int>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>
5	4	12	1.412341e-05	0.005230322	0.952	0.8085457	Preprocessor1_Model05

1 row

This set of values will be chosen for our hyperparameters as it overall provides the best specificity values.

6.7 Adding best parameters to Workflow

We create a final workflow which uses our best set of hyperparameters.

```
final_xgb_Wf <- finalize_workflow(
  xgb_tuned_wflow,
  best_spec
)

final_xgb_Wf
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_normalize()
## * step_zv()
## * step_corr()
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 5
##   trees = 100
##   min_n = 4
##   tree_depth = 12
##   learn_rate = 1.41234058236518e-05
##   loss_reduction = 0.00523032184525034
##   sample_size = 0.951999950091704
##
## Engine-Specific Arguments:
##   scale_pos_weight = 0.808545743301511
##
## Computational engine: xgboost
```

6.8 Tuned Model execution

Fit the model with our finalized workflow.

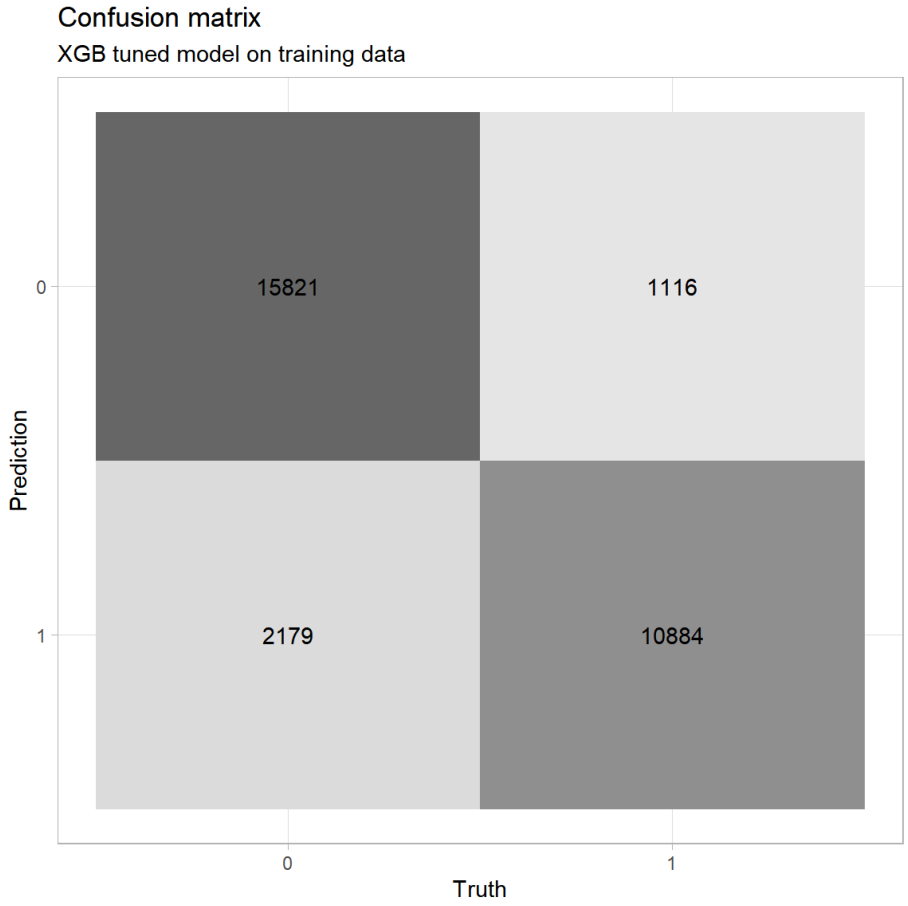
```
xgb_tuned_res <-
  final_xgb_Wf %>%
  fit_resamples(
    resamples = cv_folds,
    metrics = metric_set(
              spec,
              accuracy,
              roc_auc,
              recall,
              precision
      ),
    # Save predictions
    control = control_resamples(save_pred = TRUE)
  )
```

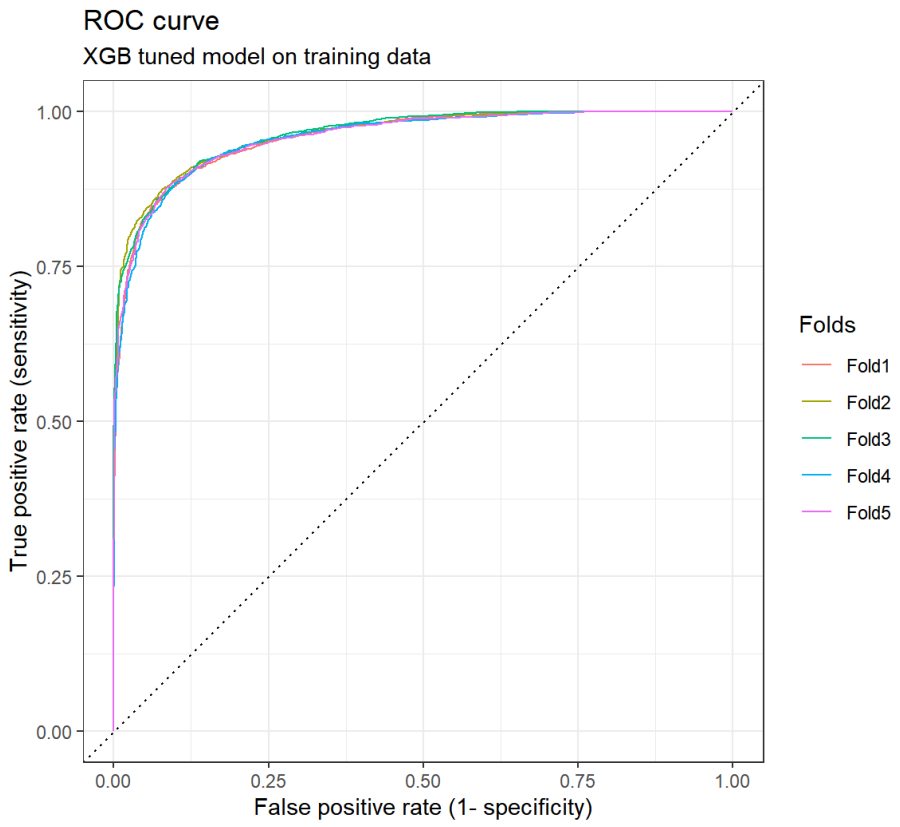
6.9 Tuned model evaluation

Evaluate the metrics over all folds and show the confusion matrix and ROC curve.

.metric	.estimator	mean	n	std_err	.config
<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
accuracy	binary	0.8901667	5	0.0010354816	Preprocessor1_Model1
precision	binary	0.9341233	5	0.0020851215	Preprocessor1_Model1
recall	binary	0.8789442	5	0.0004941674	Preprocessor1_Model1
roc_auc	binary	0.9601103	5	0.0011475852	Preprocessor1_Model1
spec	binary	0.9070110	5	0.0030008988	Preprocessor1_Model1

5 rows





We can see that our tuned model outperformed every other model regarding the specificity. Note that the value seems to be good which is a direct result of using the smote() function to rebalance the given data. The performance regarding specificity will change on the test data since it will still be imbalanced. The reason we used the smote() function is to see which hyperparameters perform the best.

7 Last evaluation

We can now perform our last fit to the test data we split of earlier.

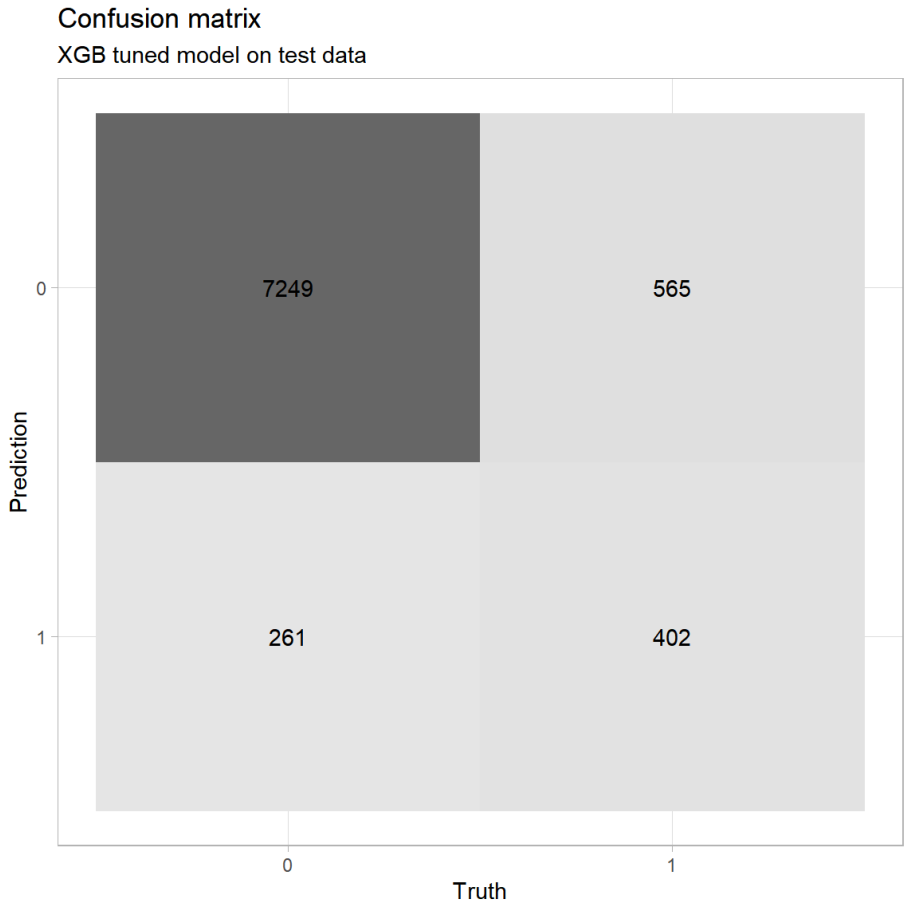
```
# perform last fit of our model on test data
last_fit_xgb <-
  last_fit(final_xgb_Wf,
           split = data_split,
           metrics = metric_set(
             spec,
             accuracy,
             roc_auc,
             recall,
             precision
             )
           )
```

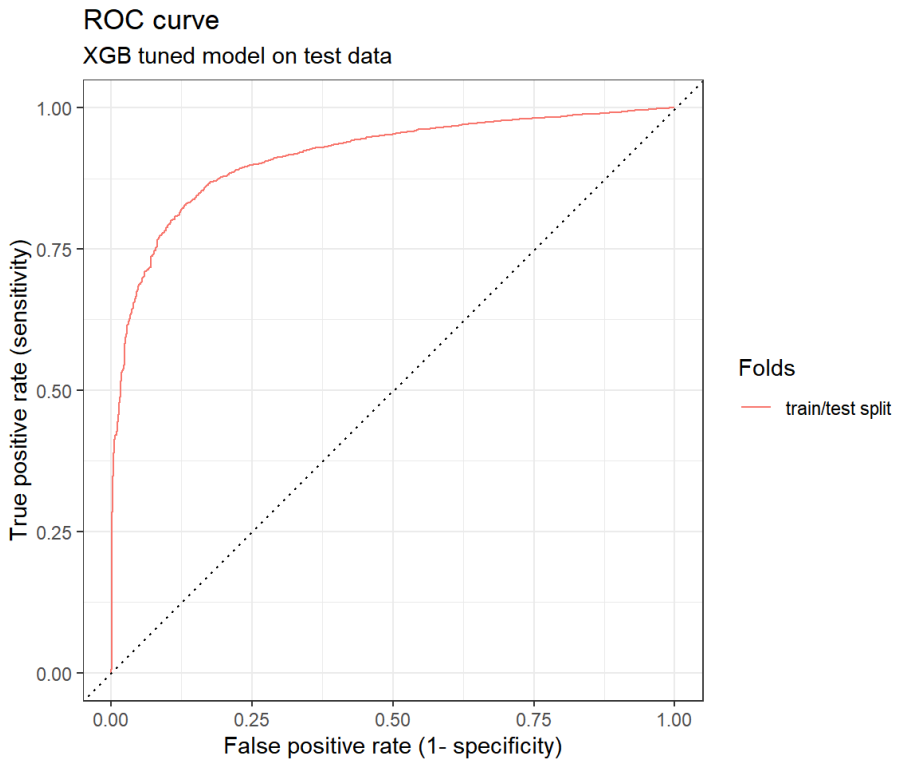
To see how our model performed we check the performance metrics

.metric	.estimator	.estimate	.config
<chr>	<chr>	<dbl>	<chr>

.metric	.estimator	.estimate	.config
<chr>	<chr>	<dbl>	<chr>
spec	binary	0.4157187	Preprocessor1_Model1
accuracy	binary	0.9025599	Preprocessor1_Model1
recall	binary	0.9652463	Preprocessor1_Model1
precision	binary	0.9276939	Preprocessor1_Model1
roc_auc	binary	0.9144662	Preprocessor1_Model1

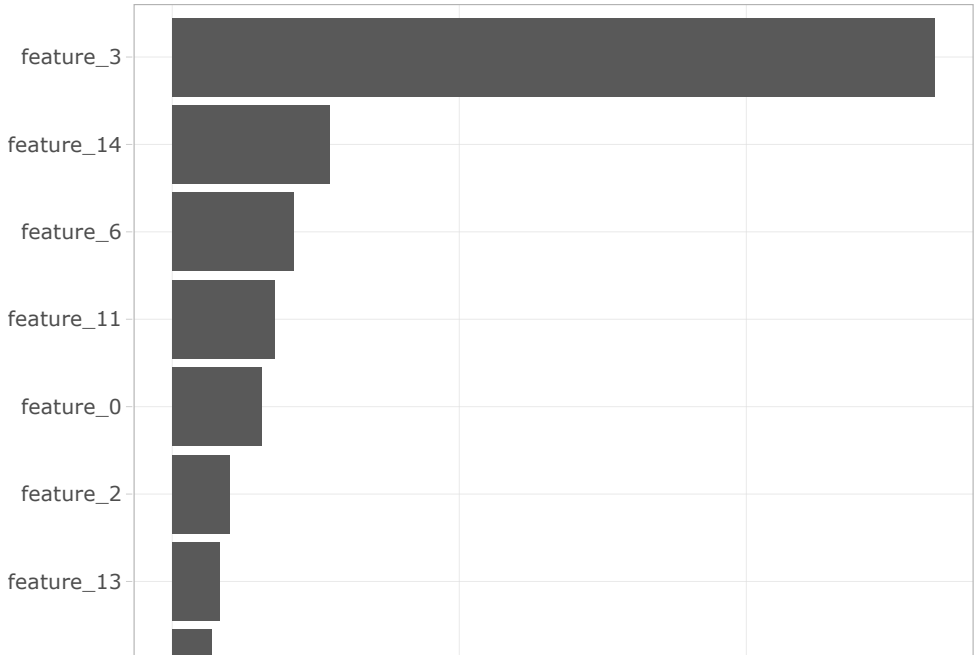
5 rows

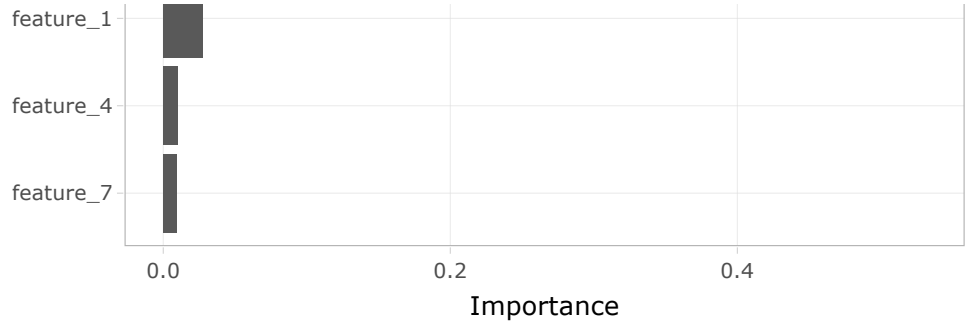




As expected the specificity is not as high as it was with our training data. It still improved from 0.378 to 0.415 by using hyperparamater tuning. Since it is still at a low value, our model did not predict the churn label well regarding the test data.

```
vip_features <-
last_fit_xgb %>%
  pluck(".workflow", 1) %>%
  pull_workflow_fit() %>%
  vip(num_features = 10)

ggplotly(vip_features)
```





We can see that the most important feature for the prediction was feature_3. Interesting is the relevance of feature_14 which we did identify as relevant before. The others have a relatively low importance which is an indicator that feature engineering would have given better results on the test data.

The results point towards an underfitting scenario where our model is not complex enough to accurately predict the churn with the given data. In order to improve the performance further, custom performance metrics or new features could be generated. Another step would be to use different models (like neural networks) or an ensemble of models.

Even if the model predicts only about 40% of the actual churning customers, We can still use it for our use case since the cost for not guessing correctly is low. Also any contract labelled as **churning** is better than no predictive information.

8 Predictions on new, unlabeled data

We will use our model on new data, which has not been labeled yet. We fit it to the second dataset provided by kaggle which we stored in our new_data dataframe.

In order to make predictions, we need to create an object of class model_fit of our final_xgb_wf. This is done using the fit function.

```
# create final model
final_model <-
  fit(final_xgb_Wf, df_train)
```

```
## [13:32:29] WARNING: amalgamation/../src/learner.cc:1115: Starting in XGBoost 1.3.0, the default
evaluation metric used with the objective 'binary:logistic' was changed from 'error' to 'logloss'.
Explicitly set eval_metric if you'd like to restore the old behavior.
```

```
# create predictions
predictions <-
predict(final_model, new_data)

# create new dataframe
new_prediction_data <-
  new_data

# add predictions to new_data_prediction
new_prediction_data$churn_prediction <-
  predictions$.pred_class

# put predictions as first column
new_prediction_data <-
  new_prediction_data %>%
  select(churn_prediction, everything())
```

We change the prediction labels to better represent the churn.

```
new_prediction_data <-
  new_prediction_data %>%
  mutate(
    churn_prediction = as.character(churn_prediction)
  )


new_prediction_data$churn_prediction[new_prediction_data$churn_prediction == 0] <- "no"
new_prediction_data$churn_prediction[new_prediction_data$churn_prediction == 1] <- "yes"
```

Let's take a look at the transformed data

```
datatable(new_prediction_data,
              extensions = list(
                'Buttons' = NULL,
                "Responsive" = NULL
              ),
              options = list(
                dom = 'Bfrtip',
                buttons = c( 'csv', 'excel', 'pdf')
              )
            )
```

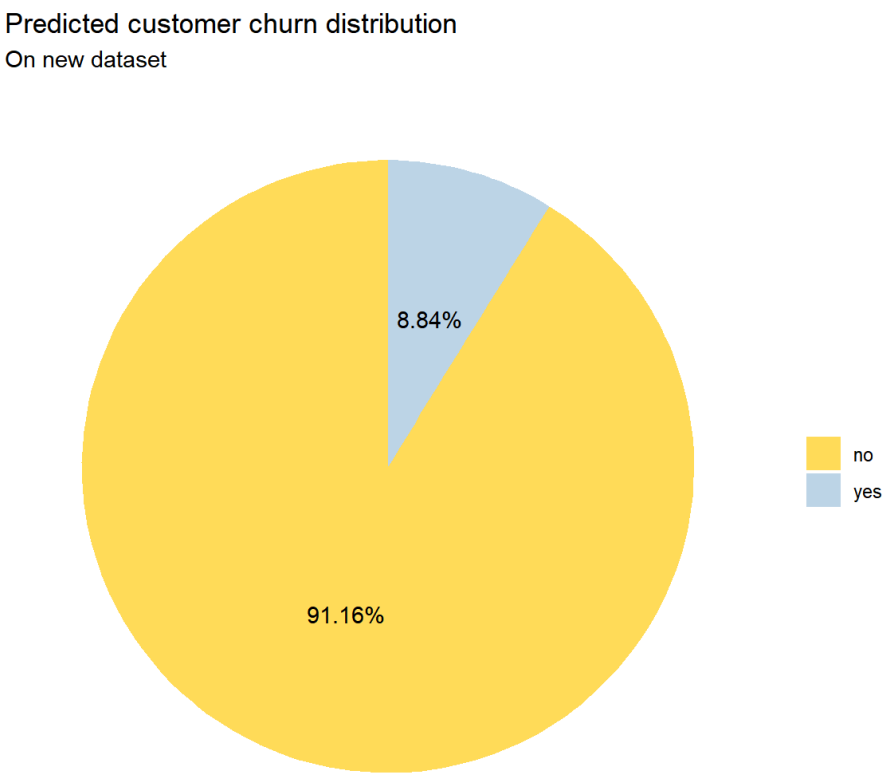
CSV Excel PDF Search:

	churn_prediction	feature_0	feature_1	feature_2	feature_3
+ 1	yes	0.571051203933711	0.406842992680743	0.984523316600528	0.0110160979054011
+ 2	no	-1.12408039431121	-0.166934899892671	0.503891811721719	-0.322932108912729
+ 3	no	0.476877226253437	0.145079409177016	-0.5775290742556	-0.691828383886244

	churn_prediction	feature_0	feature_1	feature_2	feature_3
+ 4	no	1.60696495841672	-0.447419341940579	1.82562845013844	-0.983062285181125
+ 5	no	-0.935732438950666	-0.364653441008409	-1.17831845535411	-0.322932108912729
+ 6	no	0.28852927089289	0.974709035187573	0.263576059282315	-0.940347979657876
+ 7	no	-0.841558461270392	-0.236563356232432	-1.29847633157381	-0.800555707036333
+ 8	no	1.23026904769563	-0.396511744144998	-0.217055445596494	-0.711243977305903
+ 9	no	-0.276514595188751	0.424906978995304	-0.938002702914707	-0.109360581296483
+ 10	no	1.79531291377727	0.209452887679815	-0.938002702914707	-0.777256994932743

Showing 1 to 10 of 11,303 entries Previous 1 2 3 4 5 … 1,131 Next

Next we recreate the pie chart for our predictions



churn_prediction	churn_total	percent
<chr>	<int>	<dbl>
no	10304	91.16
yes	999	8.84

2 rows

We then save the predictions on the new data, our pie chart and the plotly graph of our vip features to our Dashboard folder in order to use it in our dashboard.

```
# save data
write.csv(new_prediction_data, "Dashboard\\Results.csv", row.names=FALSE)

# save plotly chart vip-features as html
fig <-
ggplotly(vip_features)

htmlwidgets::saveWidget(fig,
                        file="Dashboard\\insurance-churn-dashboard\\www\\plotly.html",
                        selfcontained = TRUE)

# save pie chart as png
png("Dashboard\\insurance-churn-dashboard\\www\\pie-chart.png")
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs="i", yaxs="i")
pie_chart
dev.off()
```

```
## png
##   2
```

9 Conclusion

This report shows how to build a classifier algorithm for anonymized data. We have seen that our model still has room for improvement due to underfitting. The model does not predict churning at a high rate. Depending on the use case this performance has to be enhanced. I suggested, that additional feature engineering and model stacking could lead to better results.

The created dashboard is a mockup in order to display the models predictions to case handlers of contract data. You can access the source code for this report and the dashboard here: https://github.com/NicoHenzel/Insurance-Churn-Prediction (https://github.com/NicoHenzel/Insurance-Churn-Prediction)