Data Science - Regression Task

Group1

2022-12-09

Contents

Abstract

In this study, pre-cleaned data was further processed for training using one-hot encoding, removal of near zero variance variables, and scaling between 0 and 1.

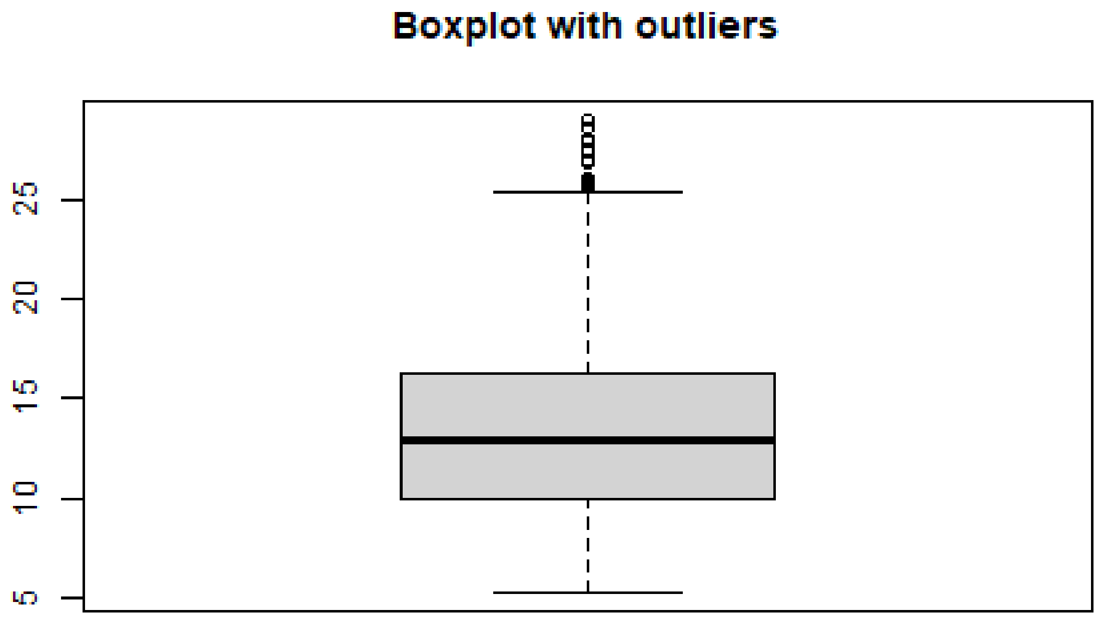
Initially, linear regression, random forest, and gradient boosting models were applied. For each model, several commonly used hyperparameters were utilized to obtain a preliminary assessment of performance. After determining that XGboost achieved the best results, the XGboost approach was selected for further optimization through hyperparameter tuning.

The optimized XGboost model was trained on the training set using the optimal set of hyperparameters and evaluated using standard metrics (MSE, RMSE, MAE, RSQ). The fit of the model was also assessed through visualization.

Based on the results of the assignment, it can be concluded that the model is a viable solution for the prediction of the interest rate. However, several potential areas for future optimizations have been identified. These may include optimizing model hyperparameters, incorporating additional data or features, and experimenting with alternative model architectures. Additionally, it may be beneficial to perform a thorough error analysis to identify specific areas where the model is underestimating the interest rate and prioritize efforts to address these issues. By implementing these recommendations, it is hoped that the model will continue to evolve and improve its performance.

General preparation of the dataset for machine learning

To begin the analysis, a boxplot of the “int_rate” class attribute can be created. This will allow for the visualization of the distribution of values within the attribute, including the minimum and maximum values, the median, and the upper and lower quartiles. This can provide valuable insights into the range and distribution of values within the attribute, as well as identify any potential outliers or anomalies.

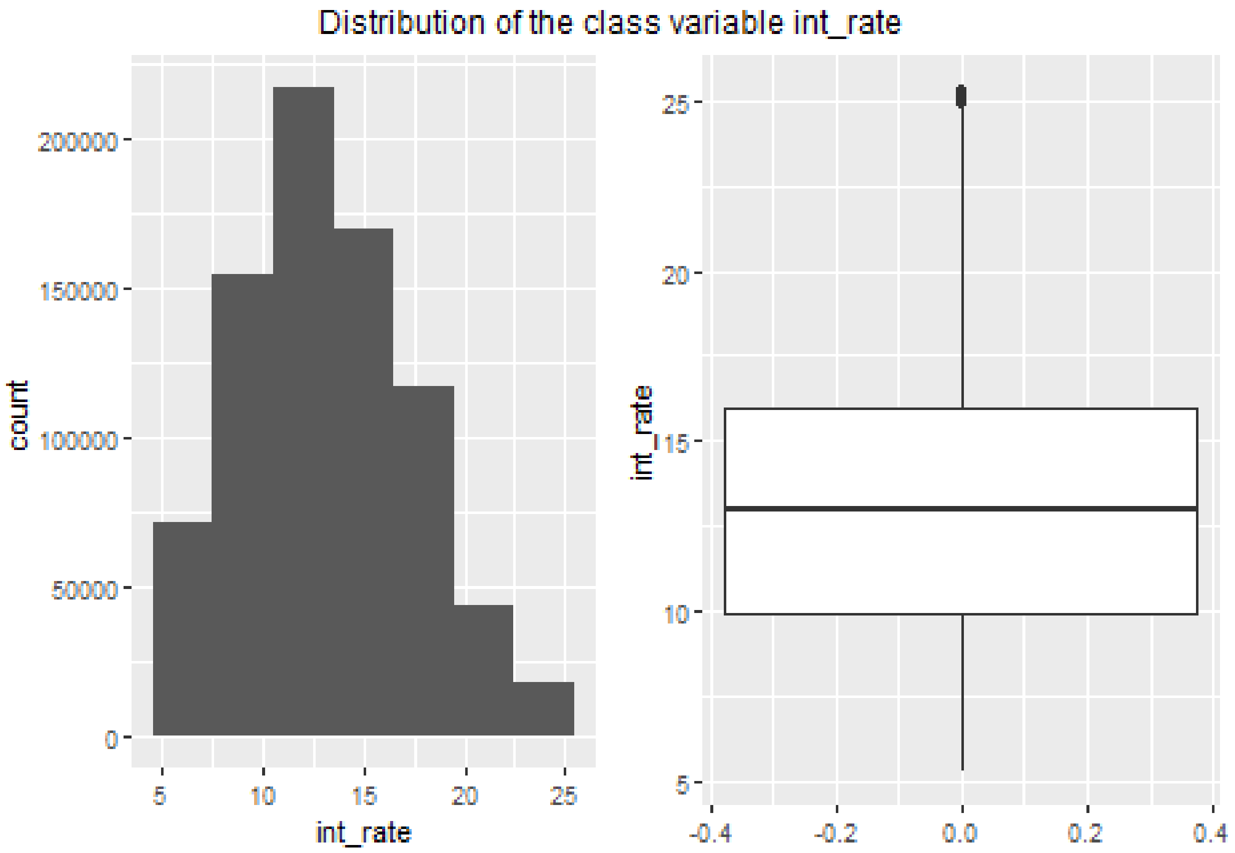


It is observed that there are several outliers at the upper end of the data, which can be identified in the boxplot through the $out column.The outliers are removed to improve the predictions on high interest rates. Nor removing the outliers results in an under prediction. These outliers can be removed by eliminating the $out column from the boxplot. The analysis also reveals that the class attribute is skewed to the right, which will be addressed through stratified sampling in a later stage of the assessment.

```r
#remove outliers in class variable
outliers  <- boxplot(lcdata$int_rate, plot=FALSE)$out
lcdata<- lcdata[-which(lcdata$int_rate %in% outliers),]

cols <- c( "home_ownership", "purpose")
lcdata[cols] <- lapply(lcdata[cols], factor)
```

A summary of the data set after initial cleaning steps:



Create a training and a test set

In order to evaluate the performance of our model, we utilize the initial_split(), training(), and testing() functions to partition our data into a training set and a test set using an 80/20 ratio. The training set is used to fit the model, while the test set is used to validate the model's ability to generalize to unseen data. This process, known as train-test split, allows us to assess the model's performance on a held-out portion of the data.

```
set.seed(1)
trainIndex <- initial_split(lcdata, prop = 0.8, strata = int_rate) #stratify the class
#attribute since it is right skewed.This bins the data into quartiles
trainingSet <- training(trainIndex)
testSet <- testing(trainIndex)
```

Pre-process the data to be used with the models

Since many predictive models perform better with numeric features, one-hot encoding is applied to the factor variables to transform them into a format where each level has its own column (using the step_dummy() function). Near zero variance predictors are also removed (using the step_nzv() function, although no predictors are affected in this case) and the predictors are normalized (using the step_range() function) to fall between 0 and 1, as the average waiting time cannot be negative. This process helps to improve the performance of the predictive models.

```
set.seed(1)
preprocRecipe <-
  recipe(int_rate ~., data = trainingSet) %>%
  step_dummy(all_nominal()) %>%
  step_nzv(all_predictors(), -all_nominal()) %>%
  step_range(all_predictors(), -all_nominal(), min = 0, max = 1)
```

In this process, we first utilize the prep() function to extract a recipe that defines the steps for data transformation. We then use the bake() function to apply this recipe to a given set of data, effectively transforming the data according to the specified steps.

```
trainingSet_processed <- preprocRecipe %>%
  prep(trainingSet) %>%
  bake(trainingSet)
testSet_processed <- preprocRecipe %>%
  prep(testSet) %>%
  bake(testSet)

#write.csv(trainingSet_processed, "../Data/Out/trainingset.csv")
#write.csv(testSet_processed, "../Data/Out/testset.csv")
```

We have now used the recipe steps to create fully processed training and test sets. We are ready to train machine learning algorithms.

Create different regression models to compare

For the given regression task, three different machine learning models have been selected for exploration:

- **multiple linear regression:** Multiple linear regression is a widely used model for regression tasks, as it is simple to implement and can be used to model the relationship between a continuous response variable and one or more explanatory variables.
- **Random Forest (bagging):** Random forest is an ensemble method that combines the predictions of multiple decision trees to make a final prediction. It is known to be robust and can handle a large number of features, making it a strong choice for regression tasks.
- **Gradient boosted decision trees (gradient boosting):** another ensemble method, use the idea of boosting to improve the performance of decision trees. It is a powerful method for regression tasks and has been shown to achieve strong results in many applications.

To evaluate the performance of these models, the default parameters for each model will be used as a starting point. This will allow for an initial comparison of the models' performance and provide a baseline for further optimization if necessary.

```
lmModel <-
  linear_reg(mode = "regression") %>%
  set_engine("lm")

rfModel <-
  rand_forest(mode = "regression",
              mtry = 3,
              trees = 500,
              min_n = 5)%>%
 set_engine("ranger")

xgModel <-
  boost_tree(mode = "regression",
            mtry = 3,
            trees = 500,
            min_n = NULL,
            tree_depth = 8,
            learn_rate = 0.3,
            loss_reduction = NULL,
            sample_size = NULL,
            stop_iter = 20) %>%
  set_engine("xgboost")
```

Each of the defined models is now fit to the training data set.

```
lmFit <- fit(lmModel, int_rate ~ ., data = trainingSet_processed)
 #rfFit <- fit(rfModel, int_rate ~ ., data = trainingSet_processed)
#saveRDS(rfFit, "../Data/Models/rfFit.rds")
rfFit <- readRDS("../Data/Models/rfFit.rds")
#xgFit <- fit(xgModel, int_rate ~ ., data = trainingSet_processed)
#saveRDS(xgFit, "../Data/Models/xgFit.rds")
xgFit <- readRDS("../Data/Models/xgFit.rds")
```

Let's look at the output:

Define evaluation metrics

Next, we need to use these model fit objects to predict classes for the test set. After that, we will calculate and interpret the metrics describing the performance of the classifiers we trained. We need to start by creating a data frame including just the class outcome and the prediction.

```
set.seed(1)

lmClassWithPredictions <- testSet_processed %>%
  dplyr::select(int_rate) %>%
  bind_cols(predict(lmFit, testSet_processed))

rfClassWithPredictions <- testSet_processed %>%
  dplyr::select(int_rate) %>%
  bind_cols(predict(rfFit, testSet_processed))

xgClassWithPredictions <- testSet_processed %>%
  dplyr::select(int_rate) %>%
  bind_cols(predict(xgFit, testSet_processed))
```

The `metric_set()` function from the "yardstick" package is used to define the metrics. For this project the following metrics have been chosen:

- mean squared error (MSE)

  - MSE is our main metric as MSE is sensitive to outliers. In regression tasks, it is important to consider the impact of outliers on the model performance. MSE gives more weight to larger errors, which makes it more sensitive to outliers and can help identify issues with the model.

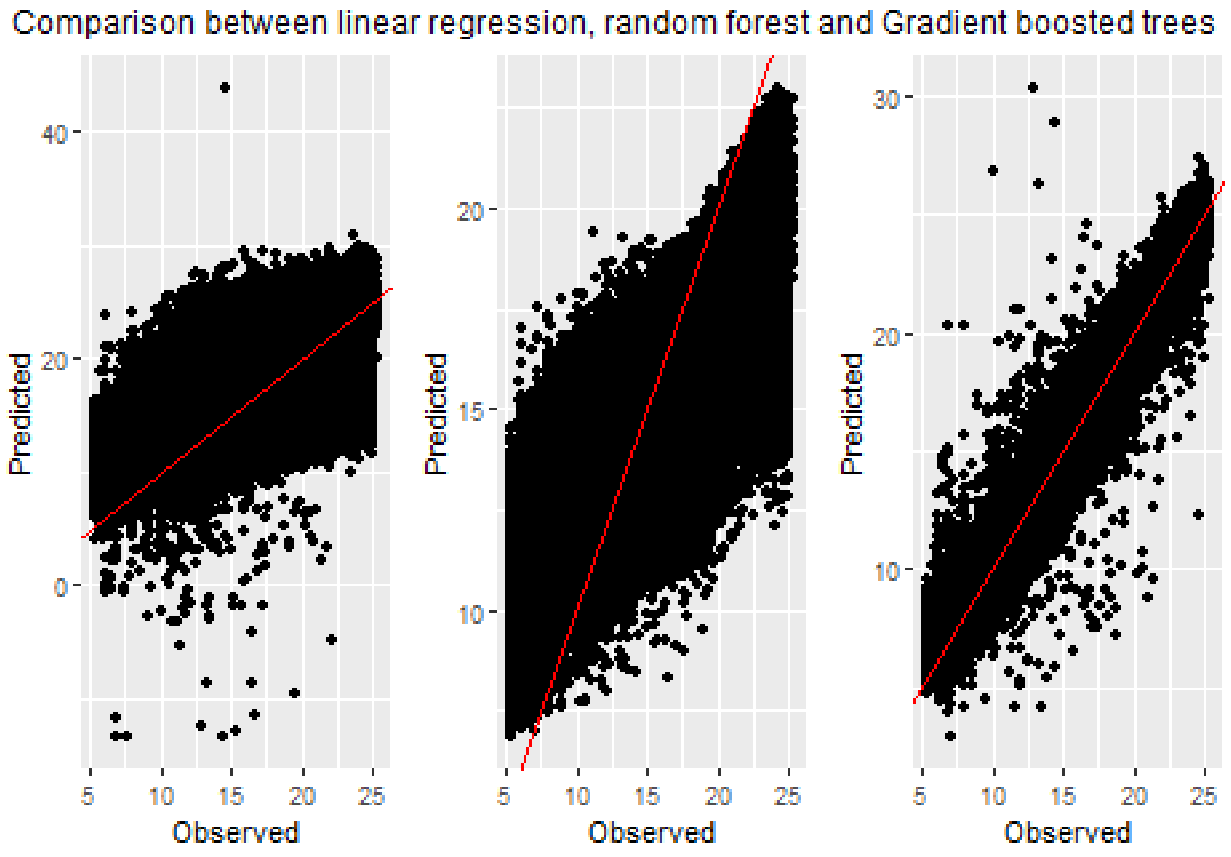
- root mean squared error (RMSE)
  - RMSE is our main metric as it puts a higher penalty on large errors. As we want to adjust our staffing according to the predicted calls a large offset would mean that we are either vastly over- or under staffed.

- root squared error (RSQ)
  - RSQ is used as a second reference. It's purpose is to tell in relative terms how much the predictor variables account for the output.

- mean absolute error (MAE)
  - MAE is used as a third validation step as it is the the metric which is most easily interpreted

```
metricSet <- metric_set(rmse, rsq, mae)
```

The metrics are now extracted and shown. Further there is a scatter plot to visualize the fit of the different models.



We can see that the gradient boosted trees with xgBoost seem to perform best. XGBoost has the best RMSE 0.8179185 as well as the best fitting plot. Is is notable that all models overestimate the number of callers in times where a low number of callers was observed.

In order to be able to also perform a hyperparamter tuning it was decided by the project to only move forward with the xgBoost model. It might be worth in the future to revisit also the linear regression and the random forest to see how well they can perform on the data given a more in depth configuration.

Prepare the xgBoost model for hyperparameter tuning

Ah new model for the xgBoost is defined where all parameters are replaced with the 'tune()' function. This allows to have them defined by the tuning grid introduced in a later step.

```
## Boosted Tree Model Specification (regression)
##
## Main Arguments:
##   mtry = tune()
##   trees = tune()
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

For the tuning grid a space-filling design is used to cover the hyperparameter space as well as possible. "Experimental designs for computer experiments are used to construct parameter grids that try to cover the parameter space such that any portion of the space has an observed combination that is not too far from it." As size we used 30 but this should be increased to get better results. computation time did not allow for a bigger value here.

Setup the workflow for later use.

```
xgb_wf <- workflow() %>%
  add_recipe(preprocRecipe)  %>%
  add_model(xgbFit)
xgb_wf
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_dummy()
## * step_nzv()
## * step_range()
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (regression)
##
## Main Arguments:
##   mtry = tune()
##   trees = tune()
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

Ee implemented a 5-fold cross validation procedure for training the model. The choice of a 5-fold cross validation was made to balance the trade-off between computational efficiency and stability of the model. It is known that increasing the number of folds in cross validation can improve the stability of the model, but at the cost of longer computation time. We also applied stratified sampling on the class variable to ensure that the relative proportions of each class were maintained in each fold, given the skewed distribution of the data. If the model exhibits high bias during evaluation, it may be necessary to consider increasing the number of folds in the cross validation procedure.

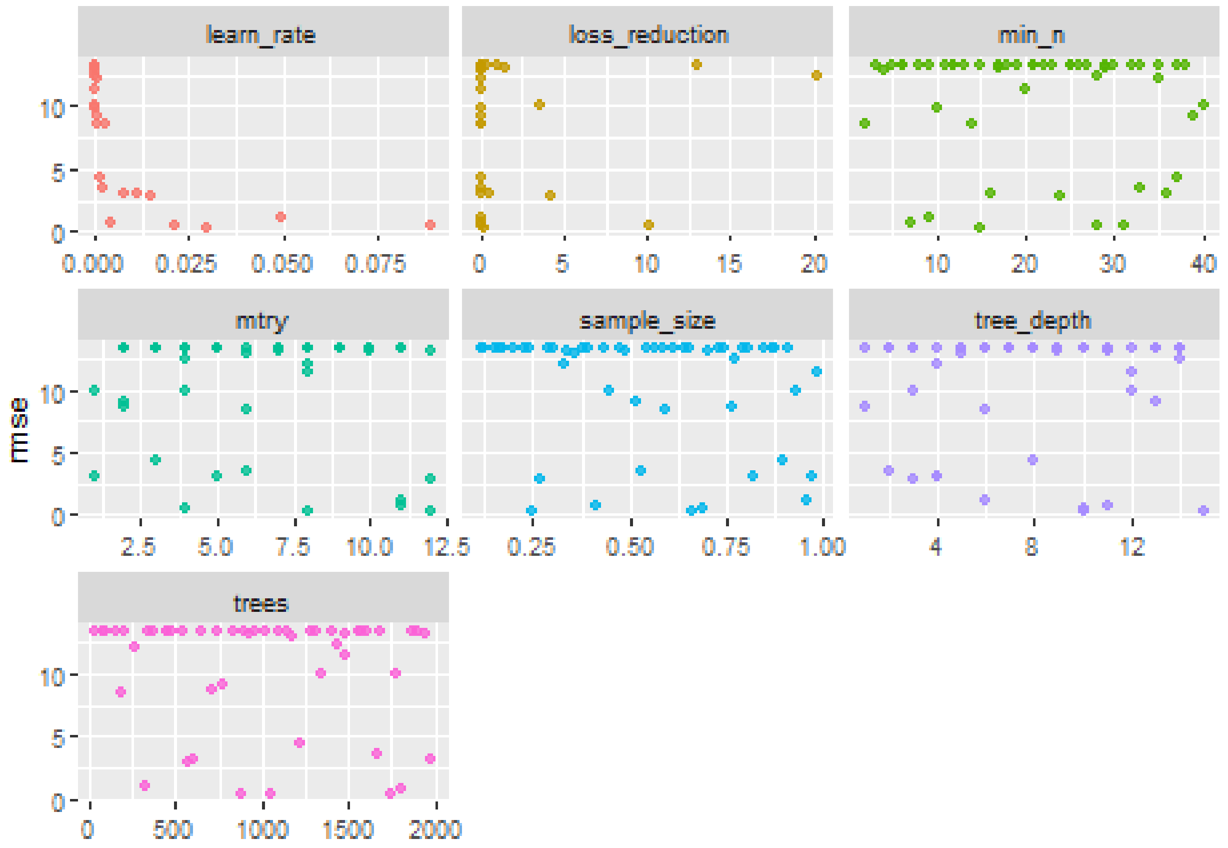
```
set.seed(1)
vb_folds <- vfold_cv(trainingSet, v = 5, strata = int_rate)
```

With the tuning grid, cross validation procedure, and evaluation metrics defined, we can now proceed to search for the optimal hyperparameters for the model. The selected hyperparameters will be those that result in the best performance, as measured by the chosen metrics, across the folds of the cross validation procedure.

```
# all_cores <- parallel::detectCores(logical = FALSE)
#
# library(doParallel)
# cl <- makePSOCKcluster(all_cores)
# registerDoParallel(cl)
#
# set.seed(123)
# xgb_res <- tune_grid(
#   xgb_wf,
#   resamples = vb_folds,
#   grid = xgb_grid,
#   metrics = metricSet,
#   control = control_grid(save_pred = TRUE)
# )
# stopCluster(cl)
# # Save a single object to a file
# saveRDS(xgb_res, "../Data/Models/xgb_res.rds")
xgb_res <- readRDS("../Data/Models/xgb_res.rds")
m <- collect_metrics(xgb_res)
```

The following visualization illustrates the results of the tuning grid runs. The Y-axis represents the root mean squared error (RMSE) and the X-axis shows the values of the tuned hyperparameter. It can be observed that increasing the learning rate leads to a reduction in RMSE. No clear correlation is evident for the other hyperparameters. There are a few notable data points, such as the min_n at 10 and 30. Increasing the size of the tuning grid may reveal more patterns in the data.

```
xgb_res %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  select(mean, mtry:sample_size) %>%
  pivot_longer(mtry:sample_size,
               values_to = "value",
               names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) +
  facet_wrap(~parameter, scales = "free_x") +
  labs(x = NULL, y = "rmse")
```



Since root mean squared error (RMSE) was chosen as the primary evaluation metric, the model with the lowest RMSE value will be selected as the best model. This model will then be saved for future use.

With the optimal hyperparameters determined, the model development process can be completed. The final model, including the chosen preprocessing steps and the set of hyperparameters, is displayed below. This model can now be evaluated on the test data to assess its generalization performance and determine its suitability for the intended application.

```
final_xgb <- finalize_workflow(
  xgb_wf,
  best_rmse
)

final_xgb
```

```
## == Workflow ==================================================================
## Preprocessor: Recipe
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_dummy()
## * step_nzv()
## * step_range()
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (regression)
##
## Main Arguments:
##   mtry = 8
##   trees = 875
##   min_n = 15
##   tree_depth = 15
##   learn_rate = 0.0298705078586578
##   loss_reduction = 0.231005266767385
##   sample_size = 0.658825560561847
##
## Computational engine: xgboost
```

The final model has now been fit to the training and test data sets. The split rate was adjusted to 95% training data in order to get the best fit possible. Further the stratification was removed to have a more accurate predictive results on the test data.

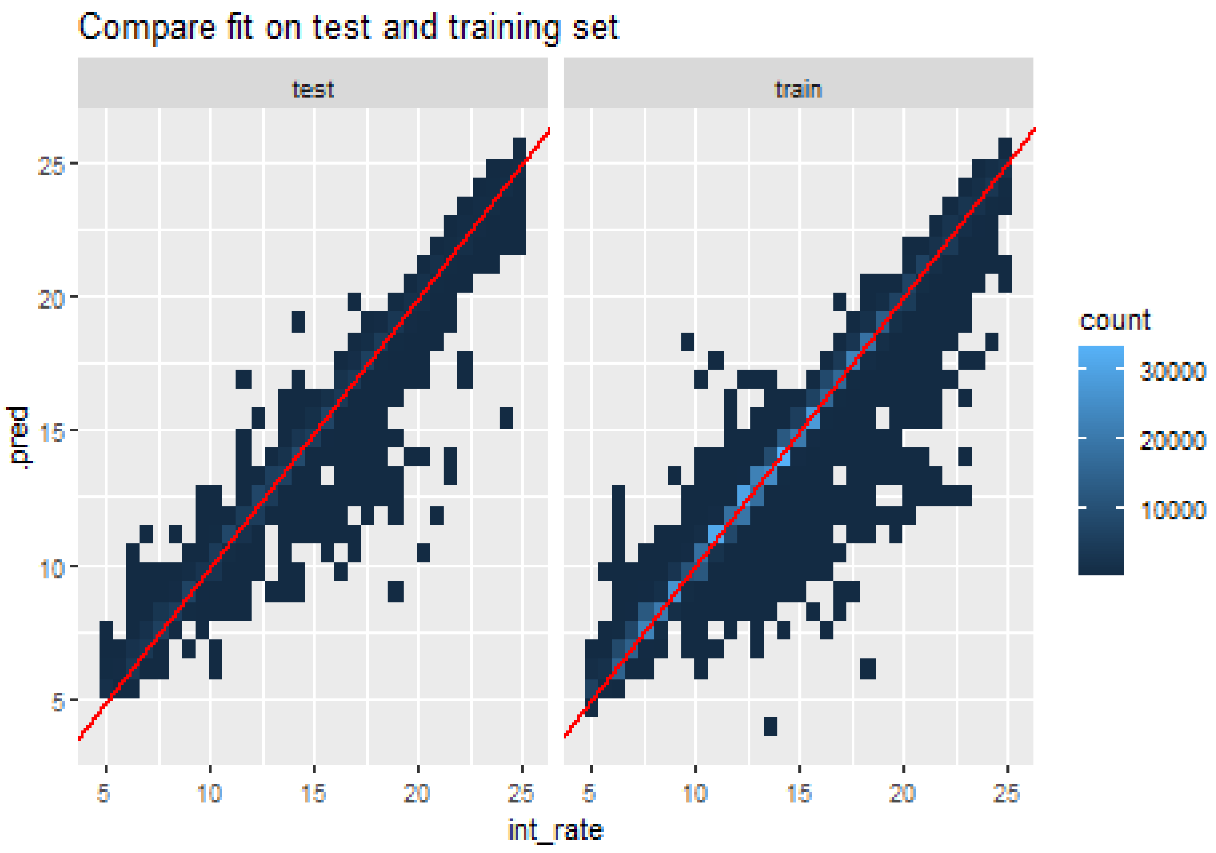
```
set.seed(33)
trainIndex2 <- initial_split(lcdata, prop = 0.95)
#final_res <- last_fit(final_xgb, trainIndex2, metrics=metricSet)
#saveRDS(final_res, "../Data/Models/final_res.rds")
final_res <- readRDS("../Data/Models/final_res.rds")
knitr::kable(collect_metrics(final_res))
```

.metric	.estimator	.estimate	.config
rmse	standard	0.3233214	Preprocessor1_Model1
rsq	standard	0.9943727	Preprocessor1_Model1
mae	standard	0.1802209	Preprocessor1_Model1

Interpreting the results

In the final step of the modeling process, we will analyze and interpret the model to gain a better understanding of its behavior and make informed decisions about its use. This can be done through various methods, such as examining the model coefficients, calculating feature importances, or visualizing the model's decision boundaries. By analyzing the model, we can identify important features, understand how they influence the predictions, and identify any potential limitations or biases in the model.

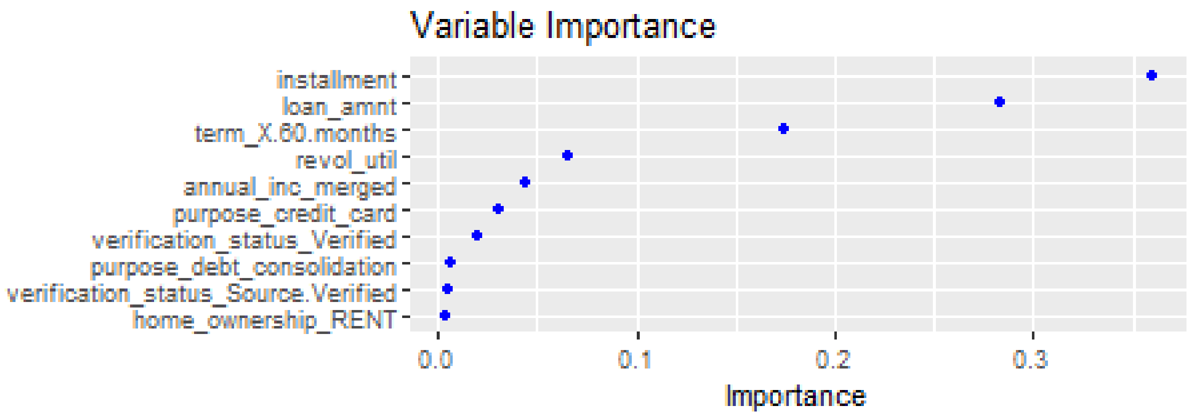
Before evaluating the model on unseen data, it is important to ensure that it is not overfitting to the training data. To do this, we compare the performance of the model on the training and test data. The results show that the model performs slightly better on the training data, but there is no significant bias against the test data. This can be observed in both the graph, where the predicted values for the two data sets are similar, and in the individual evaluation metrics. However, it appears that the model tends to slightly underestimate the int_rate, as indicated by the larger residuals below the red line in the plot.

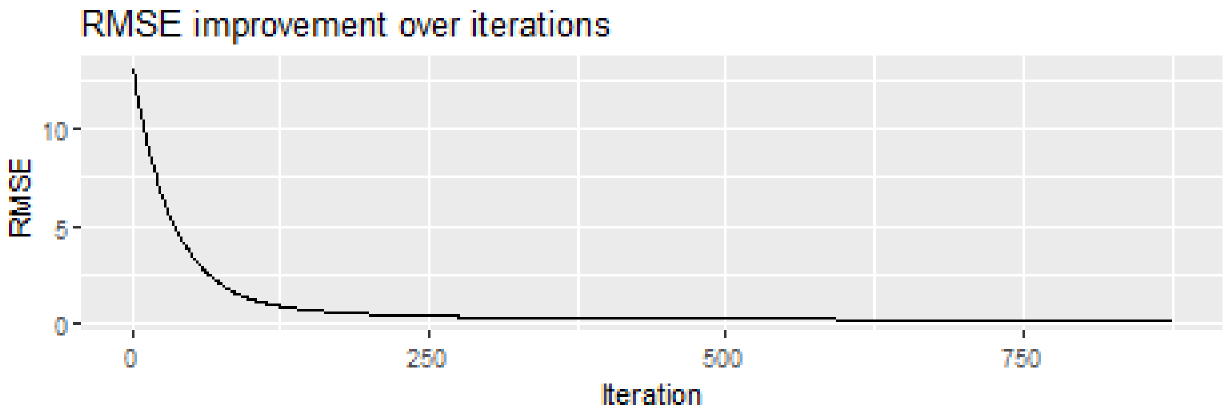


We also take a look at the variable improtance as well as the improvment with training iterations:

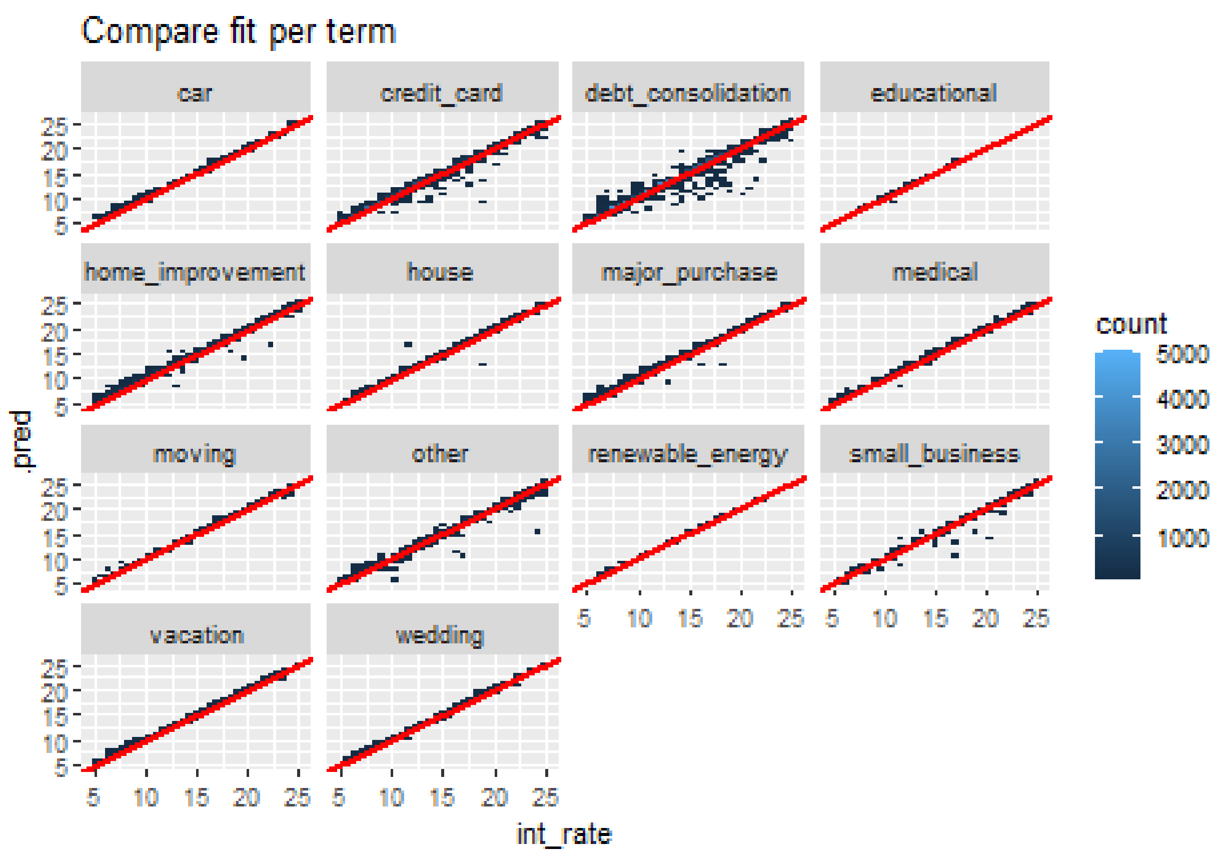
```
v1 <-extract_fit_parsnip(final_res) %>%
vip(geom = "point", aesthetics = list(color = "blue", size = 1.5))+ ggtitle("Variable Importance")

fit <- extract_fit_parsnip(final_res)
data_mod <- data.frame(Iteration = fit$fit$evaluation_log$iter,
                       RMSE = fit$fit$evaluation_log$training_rmse)
v2 <- ggplot(data_mod, aes(x=Iteration, y=RMSE)) + geom_line() + ggtitle("RMSE improvement over iterati
grid.arrange(v1,v2)
```

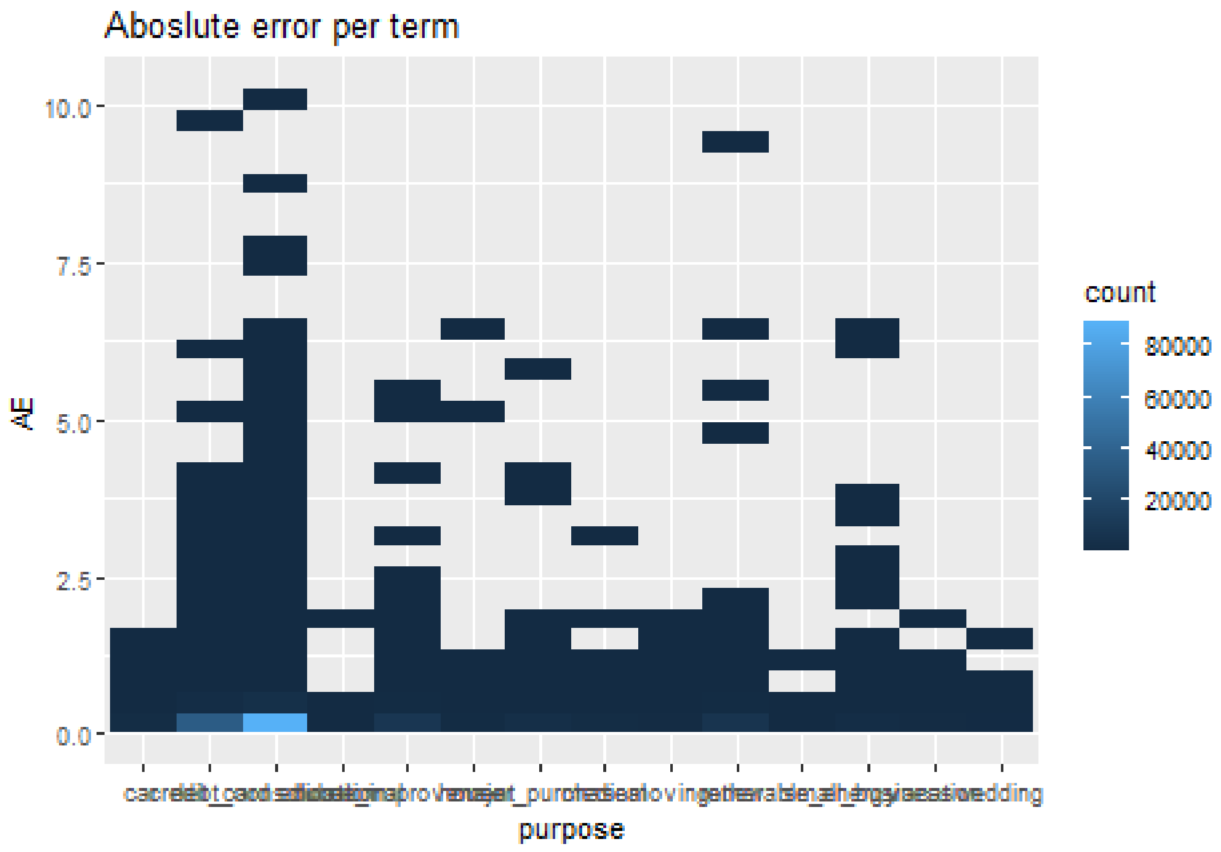




We can see that this error is very pronounced on Mondays, Tuesdays and Fridays. Further we can see that on weekends we can expect less calls in general.



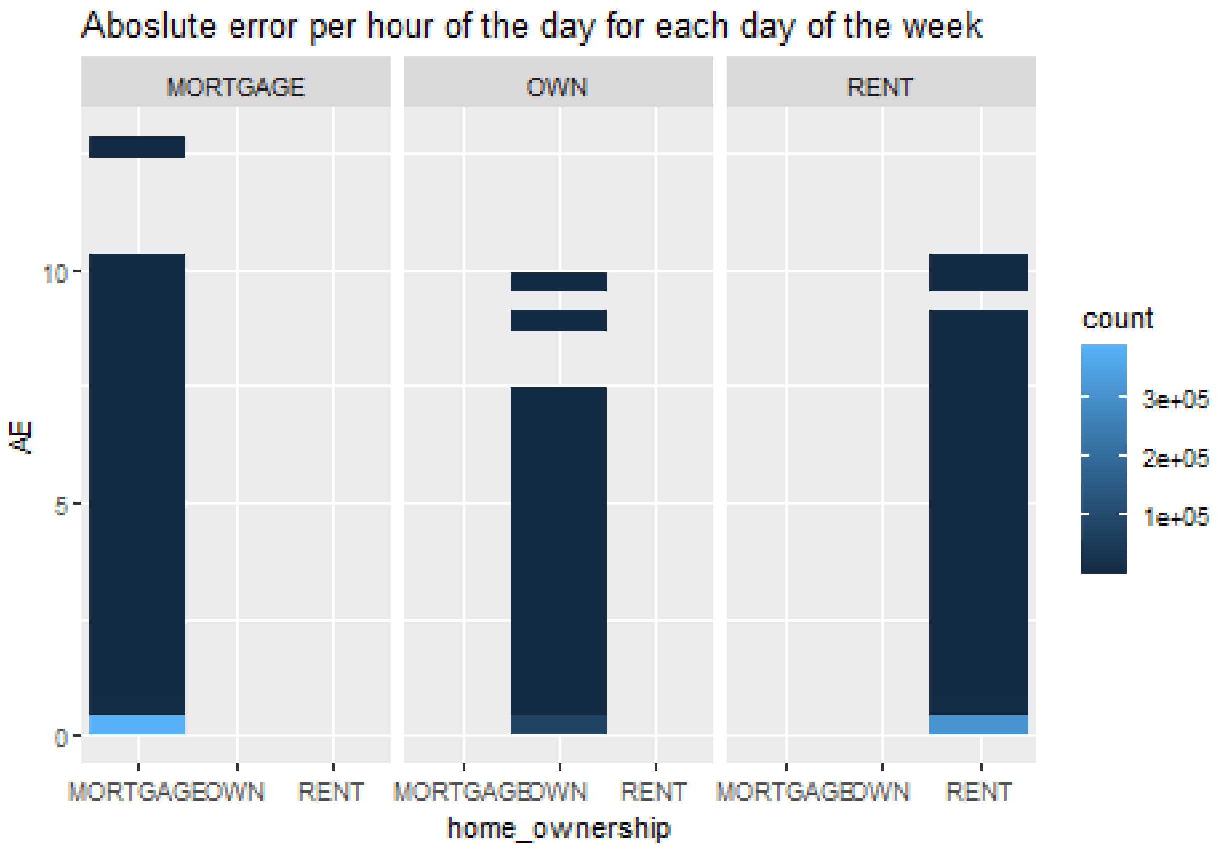
We want to analyse the weak predictions when the number of calls is low in a little more detail. We know that we have fewer calls at night. This shows also in graph below, which shows that the absolute error is higher at night when the number of calls is low.



We can observe this pattern also on the individual days of the week.

Table 2: Metrics comparison with final MSE

x	type	.metric	.estimator	.estimate	x	type	.metric	.estimator	.estimate
train	train	rmse	standard	0.2074040	test	test	rmse	standard	0.2068648
	train	rsq	standard	0.9976370		test	rsq	standard	0.9976483
	train	mae	standard	0.1249555		test	mae	standard	0.1255006
	train	mse	manual	0.0430164		test	mse	manual	0.0427930



If we now compare the metrics for the model during day time hours (07:00 -19:00), RMSE: 0.207404 with the night time hours (19:00 -07:00,RMSE: 0.2068648) we see that the model is more than 6 times more accurate during the day. We can conclude that our model performs very good during the day but has potential for improvement during the night.

Verdict and next steps

The presented model can predict the interest rate with a reasonable level of accuracy but tends underestimate a bit escpecially on higher interest rates. To further improve the model the following options are available:

- Use more source data.
- Include and experiment with additional features in the model. Especially if they can be linked with the understimation of datapoints.
- Explore other types of models in more detail: Random Forest, Neural networks etc.

- Spend more time on the tuning of the hyper parameters for the xgBoost model.
- It would also be possible to use an enriched input data set. The data can be enriched with external loan data that could improve the results even furhter.

Also to make the model more robust it would be preferable to not only use a test and training split but also a validation split. More data and increased number of folds in the cross-validation would also be preferable but was not performed for this model due to time constraints.

Sources

The following tutorials where used as a reference for this report:

- Ross, J. (n.d.). Tidymodels Tutorial. RPubs. Retrieved December 9, 2022, from https://rpubs.com/jwross83/933687
- Barter, R. (2020, April 15). Tidymodels: Tidy Machine Learning in R. Hugo Future Imperfect. Retrieved December 9, 2022, from https://www.rebeccabarter.com/blog/2020-03-25_machine_learning/
- RichardOnData. (n.d.). Youtube-scripts/R tutorial (ML) - tidymodels.rmd at master · RICHARDONDATA/youtube-scripts. GitHub. Retrieved December 9, 2022, from https://github.com/RichardOnData/YouTube-Scripts/blob/master/R%20Tutorial%20(ML)%20-%20tidymodels.Rmd