Research on Prediction of Credit Card Approval Based on Machine Learning

Bailan He, Xiong Yue, Rui Yang, Jan Alexander Jensen

Contents

1 Introduction

The credit card business has entered a fast-rising development stage after an incremental transformation from the traditional payment to E-payment. Major banks and related financial institutions increase the issuance of credit cards with the growing demand and high profits of credit card business. However, high profits are accompanied by high risks, especially in the cases of facing hundreds of thousands credit card applications from potential customers. In the actual situation, credit risk management system is set up for meeting the tremendous demand of credit card application while the increasing Probability of Default (PD), brought up by individuals who cannot pay back the loans, appears accordingly. In order to control and monitor the credit risk, the traditional credit assessment approach is an artificial credit risk assessment, which is performed by credit risk analysts who review the information submitted by credit card applicants, generally including customer personal data (age, ID card, etc.), work (income, occupation, etc.), individual assets, stability of repayment capacity, and so on. Banks and financial institutions value the repayment ability of cardholders. In other words, the stronger the repayment ability, the easier to issue credit cards will be. Nowadays, there are more and more people who apply for credit cards, and the bank's credit card business system gradually reflects the characteristics of large issuance, frequent transactions, and comprehensive and accurate transaction information. It's even clearer that traditional manual estimation is no longer able to complete these tasks. Under such circumstances, a set of practical prediction models that decide whether or not to approve these applications and monitor the credit risk, will be necessary. The traditional model and techniques cannot satisfy the current situation. For this reason, this report "Research on Prediction of Credit Card Approval Based on Machine Learning" aims to address the problem of forecasting credit risk by the use of an emerging technique "Machine Learning", and this project is based on the public dataset **"credit card approval prediction" from Kaggle**, the U.S. data competition platform. Our ultimate purpose is using machine learning approaches based on **mlr3 R package**, such as logistic regression, KNN, random forest model, etc. to establish a credit card risk assessment model to determine whether the applicant would overdue or not repay the loan and achieved a quantitative analysis of personal credit risk.

The flowchart above illustrates the process of our experiment. First, we prepare the data and define the target variable. Afterwards, we need to take care of the missing values in our data set. Here, we will try three different methods to deal with missing values - Listwise deletion, MICE and missForest. Then we have to encode the nominal variable to a numeric format so that we can use it for training. We first compare the results of all data set between different learner without fine-tuning. Subsequently, we will select one data set and perform further fine-tuning on it. In the end, we will compare the final results of all, and conclude with one method.

2 Data

According to (Bishop, Christopher (2006))[1], In machine learning, a feature is an individual measurable property or characteristic of a phenomenon being observed, and the concept of "feature" is related to that of explanatory variables used in statistic application. Therefore, both concepts mentioned in this report are the same based upon the definition of feature and variable. In this study, all features are classified into three kinds of variables according to their data type: binary variables, categorical variables, and continuous variables. The description of all features and a self-defined response variable will be provided in this section.

2.1 Reading in the data

The raw data, which consists of two data set **application_record.csv** and **credit_record.csv**, given by an Excel file in the csv format.

One of the most straightforward ways to import an Excel file into R is using the **read_csv** function. It is good practice to conduct a preliminary examination after importing the raw data, in order to ensure that the imported data has the expected format:

```
library(readr)
set.seed(2020)
application_data <- read_csv("./data/application_record.csv") # the first data set
record <- read_csv("./data/credit_record.csv") # the second data set
```

2.2 Defining response variable

The following code describes the process of defining the response variable with the name **y**. For some operations we rely on the **dplyr** package.

```
library(dplyr)

opentime_data <- record %>%
  group_by(ID) %>%
  # extract the snapshot date when an account is opened
  filter(MONTHS_BALANCE == min(MONTHS_BALANCE)) %>%
  select(ID, MONTHS_BALANCE) %>%
  rename(opentime = MONTHS_BALANCE) # rename the snapshot date as "opentime"

# merge two dataset based on the common variable "ID"
data_with_day <- left_join(application_data, opentime_data, by = "ID")
```

A payment default is in simple terms an overdue account. In short, in most cases it arises when a debt has become overdue, and based on the Basel Agreement(2007)[2] and Guidance to banks on non-performing loans from European Central Bank(ECB) https://www.bankingsupervision.europa.eu/ecb/pub/pdf/guidance_on_npl.en.pdf, the potential client, whose debt is not paid more than 60 days of the payment date printed on the invoice by the credit provider or the lender has reason to believe that the loan will not be repaid, will be classified as risk users.

```r
unique(record$STATUS)
```

```
## [1] "X" "0" "C" "1" "2" "3" "4" "5"
```

Variable **STATUS** in the data set **record** describes the overdue status, which is used to define the target in this project. The following table shows all values of the varibale $STATUS$ and corresponding meanings:

Value	Meaning
"X"	No loan for the month
"0"	1-29 days past due
"C"	Paid off that month
"1"	30-59 days past due
"2"	60-89 days overdue
"3"	90-119 days overdue
"4"	120-149 days overdue
"5"	Overdue or bad debts, write-offs for more than 150 days

```r
creat_target <- function(x) {
  if (x == "2" | x == "3" | x == "4" | x == "5") {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

The following code shows how to define the target varible **y** by using the pre-defined function "creat_target". As a result, the target varible is a binary variable, "1" means that the credit card application should not be approved.

```r
library(purrr)
# compute variable "target" with function "creat_target", and save it in new_record
new_record <- record %>% mutate(target = map_dbl(record$STATUS, creat_target))
# sum the value of variable "target" for each group(grouped by ID).
data_target <- new_record %>%
  group_by(ID) %>%
  summarise(y = sum(target))
# for each ID, if the target value >0, means there is at least one TRUE under the ID number,
# if an ID has one TURE, means this person has been overdue at least 60 days.
# then we mark this ID as 1,means we will not approve the application.
data_target$y <- map_dbl(data_target$y, function(x) ifelse(x > 0, 1, 0))
```

```r
round(prop.table(table(data_target$y)), digits = 2)
```

```
##
##    0    1
## 0.99 0.01
```

The above result shows that only 1% of records are classified as risk users. From that point of view, due to the imbalanced distribution of the target variable, it is necessary to utilize an appropriate technique to deal with this problem by training the model.

```r
# merge two data with method inner_join.
data <- inner_join(data_with_day, data_target, by = "ID")
# convert all character data into facotr datatype.
final_data <- data %>%
  mutate_if(is.character, as.factor) %>%
  mutate(y = as.factor(y))
# check data
str(final_data)
```

```
## tibble [36,457 x 20] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ ID                 : num [1:36457] 5008804 5008805 5008806 5008808 5008809 ...
##  $ CODE_GENDER        : Factor w/ 2 levels "F","M": 2 2 2 1 1 1 1 1 1 1 ...
##  $ FLAG_OWN_CAR       : Factor w/ 2 levels "N","Y": 2 2 2 1 1 1 1 1 1 1 ...
##  $ FLAG_OWN_REALTY    : Factor w/ 2 levels "N","Y": 2 2 2 2 2 2 2 2 2 2 ...
##  $ CNT_CHILDREN       : num [1:36457] 0 0 0 0 0 0 0 0 0 0 ...
##  $ AMT_INCOME_TOTAL   : num [1:36457] 427500 427500 112500 270000 270000 ...
##  $ NAME_INCOME_TYPE   : Factor w/ 5 levels "Commercial associate",..: 5 5 5 1 1 1 1 2 2 2 ...
##  $ NAME_EDUCATION_TYPE: Factor w/ 5 levels "Academic degree",..: 2 2 5 5 5 5 5 2 2 2 ...
##  $ NAME_FAMILY_STATUS : Factor w/ 5 levels "Civil marriage",..: 1 1 2 4 4 4 4 3 3 3 ...
##  $ NAME_HOUSING_TYPE  : Factor w/ 6 levels "Co-op apartment",..: 5 5 2 2 2 2 2 2 2 2 ...
##  $ DAYS_BIRTH         : num [1:36457] -12005 -12005 -21474 -19110 -19110 ...
##  $ DAYS_EMPLOYED      : num [1:36457] -4542 -4542 -1134 -3051 -3051 ...
##  $ FLAG_MOBIL         : num [1:36457] 1 1 1 1 1 1 1 1 1 1 ...
##  $ FLAG_WORK_PHONE    : num [1:36457] 1 1 0 0 0 0 0 0 0 0 ...
##  $ FLAG_PHONE         : num [1:36457] 0 0 0 1 1 1 1 0 0 0 ...
##  $ FLAG_EMAIL         : num [1:36457] 0 0 0 1 1 1 1 0 0 0 ...
##  $ OCCUPATION_TYPE    : Factor w/ 18 levels "Accountants",..: NA NA 17 15 15 15 15 NA NA NA ...
##  $ CNT_FAM_MEMBERS    : num [1:36457] 2 2 2 1 1 1 1 1 1 1 ...
##  $ opentime           : num [1:36457] -15 -14 -29 -4 -26 -26 -38 -20 -16 -17 ...
##  $ y                  : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
```

```r
# remove useless variable
# ID used to identify applications from indivuduals
# All values of "FLAG_MOBIL" are 1, which plays no role for training the model.
final_data <- final_data %>% select(-ID, -FLAG_MOBIL)

dim(final_data)
```

```
## [1] 36457    18
```

After performing the above code, we can get a final data set with 36,457 observations of 18 variables and we assume that the variable **y** (binary variable with 0 and 1) is our ultimate target variable to decide for the final decision regarding whether or not the issue shall be approved. Also, we may need to turn the data set into data frame in order to facilitate the following data processing procedures.

```r
# turnning the data into data frame
to_imp_data <- final_data %>% as.data.frame()

names(final_data)
```

```
##  [1] "CODE_GENDER"         "FLAG_OWN_CAR"        "FLAG_OWN_REALTY"
##  [4] "CNT_CHILDREN"        "AMT_INCOME_TOTAL"    "NAME_INCOME_TYPE"
##  [7] "NAME_EDUCATION_TYPE" "NAME_FAMILY_STATUS"  "NAME_HOUSING_TYPE"
## [10] "DAYS_BIRTH"          "DAYS_EMPLOYED"       "FLAG_WORK_PHONE"
```

```
## [13] "FLAG_PHONE"          "FLAG_EMAIL"          "OCCUPATION_TYPE"
## [16] "CNT_FAM_MEMBERS"     "opentime"            "y"
```

Three different variable types (e.g. binary, categorical and continuous variable) in the final data set will be introduced in detail as follows.

2.3 Binary variable

Binary variable is a type of disrete random variable which only take two possible values. While many variables and questions are naturally binary e.g. gender and "yes or no" question. The following table shows six binary variables used to record the personal information:

Name of variable in R	Meaning	Value
"CODE_GENDER"	Gender	• "F": Female • "M": Male
"FLAG_OWN_CAR"	Having a car or not	• "Y": Yes • "N": No
"FLAG_OWN_REALTY"	Having house reality or not	• "Y": Yes • "N": No
"FLAG_PHONE"	Having a phone or not	• "1": Yes • "0": No
"FLAG_EMAIL"	Having an email or not	• "1": Yes • "0": No
"FLAG_WORK_PHONE"	Having a Work Phone or not	• "1": Yes • "0": No

2.4 Categorical variable

A categorical variable is a type of statistical variable that can take on one of a finite and usually fixed number of possible values. Examples of categorical variables include different types of occupation (e.g. Core staff, Sales staff, High skill tech staff). Based on previously known qualitative properties, this kind of variable assigns each individual or other single unit of observed objects to a specific group or nominal category. five categorical varibales include the following:

Name of variable in R	Meaning	Value
"NAME_INCOME_TYPE"	Income type (5 types)	• "Pensioner" • "Working" ...

Name of variable in R	Meaning	Value
"OCCUPATION_TYPE"	Occupation Type (18 types)	• "Core staff" • "Sales staff" ...
"NAME_HOUSING_TYPE"	House Type (6 types)	• "With parents" • "House / apartment" ...
"NAME_EDUCATION_TYPE"	Education (5 types)	• "Higher education" • "Incomplete higher" ...
"NAME_FAMILY_STATUS"	Marriage Condition (5 types)	• "Married" • "Separated" ...

2.5 Continuous variable

A continuous variable is a type of numerical variables which may take on infinite values and is always collected in the form of numbers, despite the fact that other types of data also appear in the form of numbers. Examples of continuous variables include the annual income, and the number of children is not classified as categorical varibale but continuous variable in this project. The value of age is not intuitive, because it is calculated based on the actual number of days not years and represented as negative value due to the benchmark setting. The following table describes all continuous varibales used in this project:

Name of variable in R	Meaning	Value
"CNT_CHILDREN"	Children Numbers	• "0" • "1" ...
"AMT_INCOME_TOTAL"	Annual Income	• "427500" • "112500" ...
"DAYS_BIRTH"	Age	• "-12005" • "-21474" ...
"DAYS_EMPLOYED"	Working Days |	• "-4542" • "-1134" ...
"CNT_FAM_MEMBERS"	Famliy Size	• "1" • "2" ...
"opentime"	Duration of accout (Month)	• "-15" • "-14" ...

3 Missing values processing

```
summary(final_data)
```

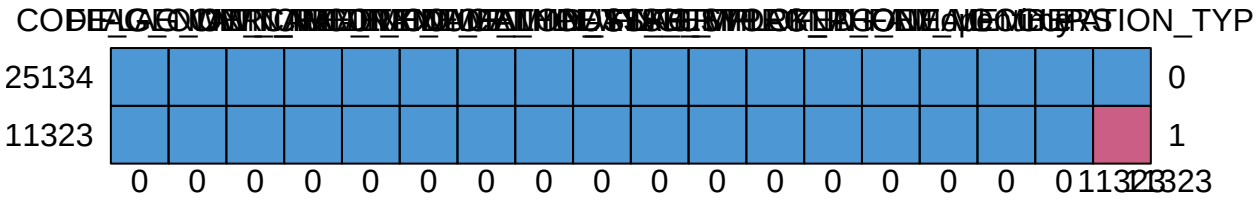
```
##  CODE_GENDER FLAG_OWN_CAR FLAG_OWN_REALTY  CNT_CHILDREN     AMT_INCOME_TOTAL
##  F:24430     N:22614      N:11951         Min.   : 0.0000   Min.   :  27000
##  M:12027     Y:13843      Y:24506         1st Qu.: 0.0000   1st Qu.: 121500
##                                           Median : 0.0000   Median : 157500
##                                           Mean   : 0.4303   Mean   : 186686
##                                           3rd Qu.: 1.0000   3rd Qu.: 225000
##                                           Max.   :19.0000   Max.   :1575000
##
##              NAME_INCOME_TYPE                    NAME_EDUCATION_TYPE
##  Commercial associate: 8490   Academic degree              :   32
##  Pensioner           : 6152   Higher education             : 9864
##  State servant       : 2985   Incomplete higher            : 1410
##  Student             :   11   Lower secondary              :  374
##  Working             :18819   Secondary / secondary special:24777
##
##
##             NAME_FAMILY_STATUS           NAME_HOUSING_TYPE   DAYS_BIRTH
##  Civil marriage      : 2945    Co-op apartment    :  168   Min.   :-25152
##  Married             :25048    House / apartment  :32548   1st Qu.:-19438
##  Separated           : 2103    Municipal apartment: 1128   Median :-15563
##  Single / not married: 4829    Office apartment   :  262   Mean   :-15975
##  Widow               : 1532    Rented apartment   :  575   3rd Qu.:-12462
##                                With parents       : 1776   Max.   : -7489
##
##  DAYS_EMPLOYED    FLAG_WORK_PHONE    FLAG_PHONE       FLAG_EMAIL
##  Min.   :-15713   Min.   :0.0000   Min.   :0.0000   Min.   :0.00000
##  1st Qu.: -3153   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.00000
##  Median : -1552   Median :0.0000   Median :0.0000   Median :0.00000
##  Mean   : 59263   Mean   :0.2255   Mean   :0.2948   Mean   :0.08972
##  3rd Qu.:  -408   3rd Qu.:0.0000   3rd Qu.:1.0000   3rd Qu.:0.00000
##  Max.   :365243   Max.   :1.0000   Max.   :1.0000   Max.   :1.00000
##
##     OCCUPATION_TYPE  CNT_FAM_MEMBERS     opentime      y
##  Laborers   : 6211   Min.   : 1.000   Min.   :-60.00   0:35841
##  Core staff : 3591   1st Qu.: 2.000   1st Qu.:-39.00   1:  616
##  Sales staff: 3485   Median : 2.000   Median :-24.00
##  Managers   : 3012   Mean   : 2.198   Mean   :-26.16
##  Drivers    : 2138   3rd Qu.: 3.000   3rd Qu.:-12.00
##  (Other)    : 6697   Max.   :20.000   Max.   :  0.00
##  NA's       :11323
```

The above result shows that there are 11,323 missing values in the varibale `OCCUPATION_TYPE`, and these missing values are marked with "NA" that cannot be addressed directly. In order to get a considerably better idea of dealing with the missing data and also to minimize the negative impact of the missing values, it is necessary to implement several methods to cover this issue, which are **listwise deletion** and with the help of two powerful R packages (**MICE** and **missForest**).

3.1 Missing pattern and missing data mechanism

In order to choose an appropriate method to deal with missing values, it is necessary to analyse the corresponding **missing pattern** and **missing data mechanism**, because these approach are based on different assumptions. It is easy to distinguish the missing pattern of this data set by the use of the function **md.pattern** from **MICE** package.

```
library(mice)
# check the missing pattern by using md.pattern function
mice::md.pattern(to_imp_data, plot = TRUE)
```




```
##       CODE_GENDER FLAG_OWN_CAR FLAG_OWN_REALTY CNT_CHILDREN AMT_INCOME_TOTAL
## 25134           1            1               1            1                1
## 11323           1            1               1            1                1
##                 0            0               0            0                0
##       NAME_INCOME_TYPE NAME_EDUCATION_TYPE NAME_FAMILY_STATUS NAME_HOUSING_TYPE
## 25134                1                   1                  1                1
## 11323                1                   1                  1                1
##                      0                   0                  0                0
##       DAYS_BIRTH DAYS_EMPLOYED FLAG_WORK_PHONE FLAG_PHONE FLAG_EMAIL
## 25134          1             1               1          1          1
## 11323          1             1               1          1          1
##                0             0               0          0          0
##       CNT_FAM_MEMBERS opentime y OCCUPATION_TYPE
## 25134               1        1 1               1     0
## 11323               1        1 1               0     1
##                     0        0 0           11323 11323
```

The above figure and result illustrate that the corresponding missing pattern is **Univariate Nonresponse**, whereby a single variable `OCCUPATION_TYPE` has missing values, and therefore, the corresponding missing data mechanism could be **Missing Completely At Random (MCAR)**. a brief description of these three missing data mechanism is provided below:

Missing Completely At Random (MCAR): If the events that lead to any specific data item being missing are indepen- dent not only of observable variables but also of unobservable parameters of interest, and if they occur completely at random, then the corresponding missing values in a data set are MCAR

Missing At Random (MAR): In contrast to MCAR data, the MAR mechanism occurs when the missing- ness is not completely random, and can be explained by at least one other variable with complete information. In this case, the missingness proba- bility is related to some of the observed data instead of the missing data itself.

Not Missing At Random (NMAR): When the missing data are NMAR, the missingness has an exclusive relationship with the missing data. In other words, the missingness probability is allowed to be dependent on the missing values themselves.

3.2 Listwise Deletion

In this section, we simply delete the missing values by using the **na.omit()** function. With this function, we would delete the whole row where the missing values appear.

```
# by using na.omit, we reomve the whole row in which the missing data is included
completeData <- na.omit(final_data)
# check the dataset after deleting
str(completeData)
```

After deleting, there are only 25,134 rows left for data analysis which means we have removed for more than 30 percent of the raw data. Therefore, a significant negative impact of listwise deletion is that it reduces the the sample size. However, the benifit of it is that it does not lead to the reduction of variability in the data, which implies that the standard deviations and variance estimates are likely to be the same, if the missing data mechanism is MCAR.

An initial guess without comparison with these methods is that listwise deletion is probably better than the other two **Multiple Imputation (MI)** methods, the possible reason is that the missing data mechanism in this case is MCAR, based on which the potential distribution by the use of listwise deletion does not change very much. In addition to this, imputing the missing values could also introduce additional errors in the original data set. The comparison results will also be provided in the subsequent section.

3.3 Multiple imputation with MICE package

In contrast to listwise deletion, MI considers statistical uncertainty when imputing missing values. One of the two powerful R-packages that handle missing data is multivariate imputation by chained equations (MICE), also called "sequential regression multiple imputation" or "fully conditional specification." This is one of the most important methods used to address and impute missing data. In consideration of the flexibility of chained equations, MICE can handle various types of variables in the data set, such as continuous variables, categorical variables, and mixed-type variables. If the distribution of each variable in the data set is already established, this method is more applicable. For example, if a variable fits the normal distribution then specific approaches can be defined in advance to impute the missing values of this variable by using the `mice()` function. Even if no appropriate multivariate distribution can be found, MICE remains an applicable option; this implies that MICE is suitable for data sets composed of mixed-type data. In conclusion, for the application of MICE the specific distribution of each variable in the data set should be defined in advance, which is based on a univariate distribution. The R-package MICE uses the FCS algorithm, which imputes each variable with missing values in the data set by conducting several repetitions.

Regarding the application of MICE, two assumptions should be taken into consideration. The first assumption is that the missing data mechanism is MAR. If data are not MAR, biased results are likely to be obtained when applying MICE. However, in order to compare the performance of MICE under the

circumstances of different missing data mechanisms, MICE is also implemented in the cases of MCAR and NMAR data. The second assumption concerns the size of the data set. In practice, data sets tend to be large in size, which implies that they include thou- sands of observations and hundreds of variables (He et al.(2009); Stuart et al.(2009)). Furthermore, in these large data sets a high variety of variables often exists. Based on the large size of data sets, a large joint model for all of the various types of variables should be fitted. With the help of the flex- ibility of MICE a series of regression models is run for each variable with missing data, which are based on the distribution of each variable. For the purposes of operability and objective comparison between different impu- tation methods, the data set in this simulation study is not large.

When using the **MICE** function, there are 2 prime parameters that need to be configured. One parameter is **m**, intended for the number of imputed dataset that we would like to generate. As a result, 1 is enough. And the other is **meth**, designed for the methods we have chosen to undergo the regression. Here, we are using the "polyreg", Bayesian polytomous regression, due to reason that these missing values all belong to factor variables.

```
### imputed data with mice function
tempData <- mice(to_imp_data, m = 1, meth = "polyreg", seed = 2020)
# to extract the imputed data with the assigned column name "OCCUPATION_TYPE"
tempData$imp$OCCUPATION_TYPE
```

In order to get a whole dataset, we can complete the data frame with the imputed ones.

```
# complete the whole dataset with the imputed data
completeData <- complete(tempData)
```

3.4 Multiple imputation with missForest package

In MICE, parametric regression models are needed and assumptions about the distribution of data are considered as prior knowledge. In the applica- tion of MICE it is necessary to specify the appropriate approach for each imputed variable in advance. If the assumptions are not correct then biased imputation results are likely to be produced. For example, it is assumed that a continuous variable fits the normal distribution; in fact this variable cannot perfectly fit normal distribution, which would lead to the estimation of problematic parameters. In addition, if there are complicated interactions, nonlinear relation structures, or high correlation between the regression model variables in the data sets, then predictions made using MICE tend to be less accurate. Consequently, the quality of the imputation would be decreased. missForest is a nonparametric method, which implies that it does not need to make assumptions about structural aspects of the data. Therefore, biased imputation results would not be caused by improper assumptions when applying missForest.

Regarding the prime parameter selection, there are three parameters that need to be taken into consideration. Firstly, the dataset with missing values **xmins**. Then, **maxiter**, which helps decide the number of iterations that need to be done during the imputation. The last parameter is **ntree** which implies the number of random trees to grow in each forest.

```
### Missing value imputation with missForest

library(missForest)
# generate final data after imputation while using missForest
imputed_Data <- missForest(to_imp_data)
# check imputation with OCCUPATION_TYPE
imputed_Data$ximp$OCCUPATION_TYPE
# get complete data with the imputed data inserted in
completeData <- imputed_Data$ximp
```

finally we get three datasets based different imputation methods: **dl_na_data**, **mf_na_data** and **mice_na_data**.

4 Feature Engineering

4.1 WOE Encoding

After running several model based on original features, we find that the positive samples(y=1) are not able to be good classified. We suppose that our model are overfitting to negative samples, because some features have too many categories, and there is a traditional variables transformation that use WOE to replace features. This is basically a technique that can be applied if we have a binary response variable.

4.1.1 Introduction

Weight of evidence (WOE) and Information value (IV) are simple but powerful techniques to perform variable transformation and selection. It is widely used in credit scoring to measure the separation of good vs bad customers. They help to explore data and screen variables. It is also used in marketing analytics project such as customer attrition model, campaign response model etc.

- What is Weight of Evidence (WOE)?
  The weight of evidence tells the predictive power of an independent variable in relation to the dependent variable.
- Steps of Calculating WOE

1. For a continuous variable, split data into several parts.
2. Calculate the number of events and non-events in each group (bin)
3. Calculate the % of events and % of non-events in each group.
4. Calculate WOE by taking natural log of division of % of non-events and % of events.

$$WOE = ln\frac{\%of\ nonevents}{\%of\ events}$$

- Benefits of WOE

1. It can treat outliers. Suppose you have a continuous variable such as annual salary and extreme values are more than 500 million dollars. These values would be grouped to a class of (let's say 250-500 million dollars). Later, instead of using the raw values, we would be using WOE scores of each classes.
2. It can handle missing values as missing values can be binned separately.
3. Since WOE Transformation handles categorical variable so there is no need for dummy variables.
4. WoE transformation helps you to build strict linear relationship with log odds. Otherwise it is not easy to accomplish linear relationship using other transformation methods such as log, square-root etc. In short, if you would not use WOE transformation, you may have to try out several transformation methods to achieve this.

- What is Information Value (IV)?
  Information value is one of the most useful technique to select important variables in a predictive model. It helps to rank variables on the basis of their importance.

$$IV = \sum(\%of\ nonevents\ -\ \%of\ event) * WOE$$

- Important Points
  Information value is not an optimal feature (variable) selection method when you are building a classification model other than binary logistic regression (for eg. random forest or SVM). So we only use IV to judge which categories should be combine together. For example: when we merge two categories into one categories within one feature and the IV does not change a lot, which means the distribution of feature with respect to target is not rapidly changed, we can accept this mergence.

4.1.2 Implementation

In this section, we will do features engineering based on three principles:
As for binary factor variables, we will convert them into binary numeric type for next modelling.
As for categorical variables, we will try to reduce the types of them without influence their IV.
As for continuous variables, By using **woebin** function of **scorecard** package, we will replace them with binning data.
Since we use 3 different strategies to deal with the missing data, we now have three dataset and for all the datasets we will excute the same process to finish features engineering. Using **dl_na_data** as an example.

Binary variables

After loading data, we inspect our factory variables as mentioned above, there are 8 factory variables, i.e. `CODE_GENDER`, `FLAG_OWN_CAR`, `FLAG_OWN_REALTY`, `NAME_INCOME_TYPE` `NAME_EDUCATION_TYPE`, `NAME_FAMILY_STATUS`, `NAME_HOUSING_TYPE`, `OCCUPATION_TYPE`, only `CODE_GENDER`, `FLAG_OWN_CAR`, `FLAG_OWN_REALTY` are binary factory variables.

```
# load final_data
dl_data <- read.csv2("./data/dl_na_data.csv")
final_data <- dl_data
binary_data <- final_data %>% select_if(is.factor)
glimpse(binary_data)
```

```
## Rows: 25,134
## Columns: 8
## $ CODE_GENDER         <fct> M, F, F, F, F, M, M, M, M, M, M, M, M, F, F, F,...
## $ FLAG_OWN_CAR        <fct> Y, N, N, N, N, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, N,...
## $ FLAG_OWN_REALTY     <fct> Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, N, N, Y,...
## $ NAME_INCOME_TYPE    <fct> Working, Commercial associate, Commercial assoc...
## $ NAME_EDUCATION_TYPE <fct> Secondary / secondary special, Secondary / seco...
## $ NAME_FAMILY_STATUS  <fct> Married, Single / not married, Single / not mar...
## $ NAME_HOUSING_TYPE   <fct> House / apartment, House / apartment, House / a...
## $ OCCUPATION_TYPE     <fct> Security staff, Sales staff, Sales staff, Sales...
```

As for binary factor variables, we write a function **convert_binary_variable** to convert them into binary numeric type for next modelling.

```
# function to convert binary variable into numeric.
convert_binary_variable <- function(feature) {
  # save condition for succinct expression
  # check for defensive programm
  if (check_class(feature, "factor") & nlevels(feature) == 2) {
    levels(feature) <- seq(nlevels(feature))
    feature <- as.numeric(feature)
    return(feature)
  }
}
# convert all binary variable into binary number
converted_data <- final_data %>% mutate_if(is.factor, convert_binary_variable)
# inspect binary variables
converted_binary_data <- binary_data %>% mutate_if(is.factor, convert_binary_variable)
glimpse(converted_binary_data)
```

```
## Rows: 25,134
## Columns: 3
## $ CODE_GENDER     <dbl> 2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, ...
## $ FLAG_OWN_CAR    <dbl> 2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, ...
## $ FLAG_OWN_REALTY <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, ...
```

Categorical variables After converting binary factor variables, five Categorical variables are remained, i.e. `NAME_INCOME_TYPE`, `NAME_EDUCATION_TYPE`, `NAME_FAMILY_STATUS`, `NAME_HOUSING_TYPE`, `OCCUPATION_TYPE`,

```
# find which variables have several values
dif_variable <- setdiff(names(final_data), names(converted_data))
dif_variable
```

```
## [1] "NAME_INCOME_TYPE"    "NAME_EDUCATION_TYPE" "NAME_FAMILY_STATUS"
## [4] "NAME_HOUSING_TYPE"   "OCCUPATION_TYPE"
```

As for `NAME_INCOME_TYPE`, we write the function **calc_iv** to inspect its WOE and IV.

```
calc_iv(dif_variable[1])
```

```
## [1] 0.0100533
```

```
## # A tibble: 5 x 11
##   feature val     all  good   bad   share bad_rate good_dis bad_dis      woe
##   <chr>   <chr> <dbl> <dbl> <dbl>   <dbl>    <dbl>    <dbl>   <dbl>    <dbl>
## 1 NAME_I~ Work~ 15622 15361   261 6.22e-1   0.0167 0.622     0.618   0.00503
## 2 NAME_I~ Comm~  7052  6933   119 2.81e-1   0.0169 0.281     0.282  -0.00511
## 3 NAME_I~ Stat~  2437  2408    29 9.70e-2   0.0119 0.0974    0.0687  0.349
## 4 NAME_I~ Stud~    10    10     0 3.98e-4   0      0.000405  0       0
## 5 NAME_I~ Pens~    13     0    13 5.17e-4   1      0         0.0308  0
## # ... with 1 more variable: iv <dbl>
```

We noticed that factor type `Student` and `Pensioner` are remarkably useful to determine the target. Factor `Pensioner` can be regarded as "refuse mark", as they doesn't work and have less resilient financial condition. On the other hand, `Student` can be regarded as "accept mark", perhaps because they can get finantial support from their parents. Although the two factor are relatively strange, we will not merge them into other types.

```
# analyse the details of first multi-factors variable
# found the variable "NAME_INCOME_TYPE" has 5 types of values
final_data["less_factor_income"] <- final_data %>%
  pull(dif_variable[[1]])
```

As for `NAME_EDUCSTION_TYPE`, we also inspect this variables at first.

```
calc_iv(dif_variable[2])
```

```
## [1] 0.01043881
```

```
## # A tibble: 5 x 11
##   feature val     all  good   bad   share bad_rate good_dis bad_dis     woe
##   <chr>   <chr> <dbl> <dbl> <dbl>   <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
## 1 NAME_E~ Seco~ 16808 16541   267 6.69e-1   0.0159 0.669     0.633   0.0563
## 2 NAME_E~ High~  7132  7004   128 2.84e-1   0.0179 0.283     0.303  -0.0678
## 3 NAME_E~ Inco~   993   972    21 3.95e-2   0.0211 0.0393    0.0498 -0.235
## 4 NAME_E~ Lowe~   187   181     6 7.44e-3   0.0321 0.00732   0.0142 -0.663
## 5 NAME_E~ Acad~    14    14     0 5.57e-4   0      0.000567  0       0
## # ... with 1 more variable: iv <dbl>
```

We noticed that factor `Academic degree` also can be regarded as "accept mark", `Higher education` and `Incomplete higher` have the same distribution, so we merge them together. IV slightly decreases from **0.01043881** to **0.00935902**, which we can accpet.

```
# analyse the details of second multi-factors variable
# found the variable "NAME_EDUCSTION_TYPE" has 5 types of values. we integrate
# "Incomplete higher" with "Higher education"
final_data["less_factor_edu"] <- final_data %>%
  pull(dif_variable[[2]]) %>%
  recode("Higher education" = "Incomplete higher")
# analyse how the IV of the variable has been changed.
calc_iv("less_factor_edu")
```

```
## [1] 0.00935902
```

```
## # A tibble: 4 x 11
##   feature val     all  good   bad   share bad_rate good_dis bad_dis     woe
##   <chr>   <chr> <dbl> <dbl> <dbl>   <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
## 1 less_f~ Seco~ 16808 16541   267 6.69e-1   0.0159 0.669     0.633   0.0563
## 2 less_f~ Inco~  8125  7976   149 3.23e-1   0.0183 0.323     0.353  -0.0898
## 3 less_f~ Lowe~   187   181     6 7.44e-3   0.0321 0.00732   0.0142 -0.663
## 4 less_f~ Acad~    14    14     0 5.57e-4   0      0.000567  0       0
## # ... with 1 more variable: iv <dbl>
```

We noticed `NAME_FAMILY_STATUS` feature is relatively balanced, no need to change.

```
# analyse the details of third multi-factors variable
calc_iv(dif_variable[3])
```

```
## [1] 0.04313712
```

```
## # A tibble: 5 x 11
##   feature val     all  good   bad  share bad_rate good_dis bad_dis     woe
##   <chr>   <chr> <dbl> <dbl> <dbl>  <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
## 1 NAME_F~ Marr~ 17509 17232   277 0.697    0.0158   0.697   0.656   0.0605
## 2 NAME_F~ Sing~  3445  3362    83 0.137    0.0241   0.136   0.197  -0.369
## 3 NAME_F~ Civi~  2133  2101    32 0.0849   0.0150   0.0850  0.0758  0.114
## 4 NAME_F~ Sepa~  1467  1452    15 0.0584   0.0102   0.0588  0.0355  0.503
## 5 NAME_F~ Widow   580   565    15 0.0231   0.0259   0.0229  0.0355 -0.441
## # ... with 1 more variable: iv <dbl>
```

```
final_data["less_factor_status"] <- final_data %>% pull(dif_variable[3])
# found this variable seems relatively balanced, no need to change.
```

We noticed the variable `NAME_HOUSING_TYPE` has 6 types of values. Factor `Co-op apartment` has the same distribution with `Office apartment`. so we merge them into one factor.

```
calc_iv(dif_variable[4])
```

```
## [1] 0.007327503
```

```
## # A tibble: 6 x 11
##   feature val     all  good   bad   share bad_rate good_dis bad_dis     woe
##   <chr>   <chr> <dbl> <dbl> <dbl>   <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
## 1 NAME_H~ Hous~ 22102 21738   364 0.879     0.0165  0.880   0.863    0.0196
```

```
## 2 NAME_H~ Rent~   439   433     6 0.0175    0.0137  0.0175  0.0142   0.209
## 3 NAME_H~ Muni~   812   793    19 0.0323    0.0234  0.0321  0.0450  -0.339
## 4 NAME_H~ With~  1430  1405    25 0.0569    0.0175  0.0569  0.0592  -0.0411
## 5 NAME_H~ Co-o~   152   149     3 0.00605   0.0197  0.00603 0.00711 -0.165
## 6 NAME_H~ Offi~   199   194     5 0.00792   0.0251  0.00785 0.0118  -0.412
## # ... with 1 more variable: iv <dbl>
```

After merge `Co-op apartment` into `Office apartment`, IV slightly decreases from **0.007327503** to **0.007086979**,which we can accpet.

```
# analyse the details of fourth multi-factors variable
# found the variable "NAME_HOUSING_TYPE" has 6 types of values. we integrate
# "Co-op apartment" with "Office apartment"
final_data["less_factor_house"] <- final_data %>%
  pull(dif_variable[[4]]) %>%
  recode("Co-op apartment" = "Office apartment")
# analyse how the IV of the variable has been changed.
calc_iv("less_factor_house")
```

```
## [1] 0.007086979
```

```
## # A tibble: 5 x 11
##   feature val     all  good   bad  share bad_rate good_dis bad_dis     woe
##   <chr>   <chr> <dbl> <dbl> <dbl>  <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
## 1 less_f~ Hous~ 22102 21738   364 0.879    0.0165   0.880   0.863   0.0196
## 2 less_f~ Rent~   439   433     6 0.0175   0.0137   0.0175  0.0142  0.209
## 3 less_f~ Muni~   812   793    19 0.0323   0.0234   0.0321  0.0450 -0.339
## 4 less_f~ With~  1430  1405    25 0.0569   0.0175   0.0569  0.0592 -0.0411
## 5 less_f~ Offi~   351   343     8 0.0140   0.0228   0.0139  0.0190 -0.312
## # ... with 1 more variable: iv <dbl>
```

We found the variable `NAME_HOUSING_TYPE` has too manny categories. We integrate some of them with different working status. i.e. `Cleaning staff`, `Cooking staff` can be classified as `Labor`, `Accountants`, `Core staff` can be classified as `Office`, `IT staff`, `High skilltech staff` can be classified as `higher`,which means they earn a higher salary.

```
calc_iv(dif_variable[5])
```

```
## [1] 0.08805434
```

```
## # A tibble: 18 x 11
##    feature val     all  good   bad   share bad_rate good_dis bad_dis     woe
##    <chr>   <chr> <dbl> <dbl> <dbl>   <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
##  1 OCCUPA~ Secu~   592   579    13 0.0236   0.0220   0.0234  0.0308  -0.274
##  2 OCCUPA~ Sale~  3485  3440    45 0.139    0.0129   0.139   0.107    0.267
##  3 OCCUPA~ Acco~  1241  1218    23 0.0494   0.0185   0.0493  0.0545  -0.101
##  4 OCCUPA~ Labo~  6211  6112    99 0.247    0.0159   0.247   0.235    0.0529
##  5 OCCUPA~ Mana~  3012  2965    47 0.120    0.0156   0.120   0.111    0.0744
##  6 OCCUPA~ Driv~  2138  2089    49 0.0851   0.0229   0.0845  0.116   -0.317
##  7 OCCUPA~ Core~  3591  3517    74 0.143    0.0206   0.142   0.175   -0.209
##  8 OCCUPA~ High~  1383  1353    30 0.0550   0.0217   0.0548  0.0711  -0.261
##  9 OCCUPA~ Clea~   551   546     5 0.0219   0.00907  0.0221  0.0118   0.623
## 10 OCCUPA~ Priv~   344   342     2 0.0137   0.00581  0.0138  0.00474  1.07
## 11 OCCUPA~ Cook~   655   646     9 0.0261   0.0137   0.0261  0.0213   0.204
## 12 OCCUPA~ Low-~   175   167     8 0.00696  0.0457   0.00676 0.0190  -1.03
```

```
## 13 OCCUPA~ Medi~  1207  1197    10 0.0480   0.00829  0.0484  0.0237   0.715
## 14 OCCUPA~ Secr~   151   149     2 0.00601  0.0132   0.00603 0.00474  0.241
## 15 OCCUPA~ Wait~   174   172     2 0.00692  0.0115   0.00696 0.00474  0.384
## 16 OCCUPA~ HR s~    85    84     1 0.00338  0.0118   0.00340 0.00237  0.361
## 17 OCCUPA~ Real~    79    79     0 0.00314  0        0.00320 0        0
## 18 OCCUPA~ IT s~    60    57     3 0.00239  0.05     0.00231 0.00711 -1.13
## # ... with 1 more variable: iv <dbl>
```

```
# analyse the details of fifth multi-factors variable
# found the variable "NAME_HOUSING_TYPE" has too many categories.

final_data["less_factor_work"] <- final_data %>%
  pull(dif_variable[[5]]) %>%
  recode(
    "Cleaning staff" = "Labor",
    "Cooking staff" = "Labor",
    "Drivers" = "Labor",
    "Laborers" = "Labor",
    "Low-skill Laborers" = "Labor",
    "Security staff" = "Labor",
    "Waiters/barmen staff" = "Labor"
  ) %>%
  recode(
    "Accountants" = "Office",
    "Core staff" = "Office",
    "HR staff" = "Office",
    "Medicine staff" = "Office",
    "Private service staff" = "Office",
    "Realty agents" = "Office",
    "Sales staff" = "Office",
    "Secretaries" = "Office"
  ) %>%
  recode(
    "Managers" = "higher",
    "High skill tech staff" = "higher",
    "IT staff" = "higher"
  )
# analyse how the IV of the variable has been changed.

calc_iv("less_factor_work")
```

```
## [1] 0.004820472
```

```
## # A tibble: 3 x 11
##   feature val     all  good   bad share bad_rate good_dis bad_dis     woe
##   <chr>   <chr> <dbl> <dbl> <dbl> <dbl>    <dbl>    <dbl>   <dbl>   <dbl>
## 1 less_f~ Labor 10496 10311   185 0.418   0.0176    0.417   0.438 -0.0494
## 2 less_f~ Offi~ 10183 10026   157 0.405   0.0154    0.406   0.372  0.0867
## 3 less_f~ high~  4455  4375    80 0.177   0.0180    0.177   0.190 -0.0684
## # ... with 1 more variable: iv <dbl>
```

Then encode multi-factors variables as one-hot variables.

```
# encode multi-factors variables into one-hot
factor_data <- final_data %>%
  select(starts_with("less")) %>%
  as.data.table()
oh_data <- one_hot(factor_data, dropCols = T)
# cbind oh_data with converted_data
converted_data <- final_data %>% mutate_if(is.factor, convert_binary_variable)
converted_data <- converted_data %>% add_column(oh_data)
# print oh_data
glimpse(oh_data)
```

```
## Rows: 25,134
## Columns: 22
## $ `less_factor_edu_Academic degree`               <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_edu_Incomplete higher`             <int> 0, 0, 0, 0, 0, 1, 1...
## $ `less_factor_edu_Lower secondary`               <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_edu_Secondary / secondary special` <int> 1, 1, 1, 1, 1, 0, 0...
## $ `less_factor_house_House / apartment`           <int> 1, 1, 1, 1, 1, 1, 1...
## $ `less_factor_house_Municipal apartment`         <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_house_Office apartment`            <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_house_Rented apartment`            <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_house_With parents`                <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_income_Commercial associate`       <int> 0, 1, 1, 1, 1, 0, 0...
## $ less_factor_income_Pensioner                    <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_income_State servant`              <int> 0, 0, 0, 0, 0, 0, 0...
## $ less_factor_income_Student                      <int> 0, 0, 0, 0, 0, 0, 0...
## $ less_factor_income_Working                      <int> 1, 0, 0, 0, 0, 1, 1...
## $ `less_factor_status_Civil marriage`             <int> 0, 0, 0, 0, 0, 0, 0...
## $ less_factor_status_Married                      <int> 1, 0, 0, 0, 0, 1, 1...
## $ less_factor_status_Separated                    <int> 0, 0, 0, 0, 0, 0, 0...
## $ `less_factor_status_Single / not married`       <int> 0, 1, 1, 1, 1, 0, 0...
## $ less_factor_status_Widow                        <int> 0, 0, 0, 0, 0, 0, 0...
## $ less_factor_work_Labor                          <int> 1, 0, 0, 0, 0, 0, 0...
## $ less_factor_work_Office                         <int> 0, 1, 1, 1, 1, 1, 1...
## $ less_factor_work_higher                         <int> 0, 0, 0, 0, 0, 0, 0...
```

And we discretizate the continuous variables with the help of **woebin** function from **scorecard** package. When using the **woebin** function, there is one parameter **method** that need to be configured, desiged for which method will based on to discretizate variables. **method** has two options, **tree** and **chimerge**. We choose **chimerge** because we noticed it brings better performance for modelling.

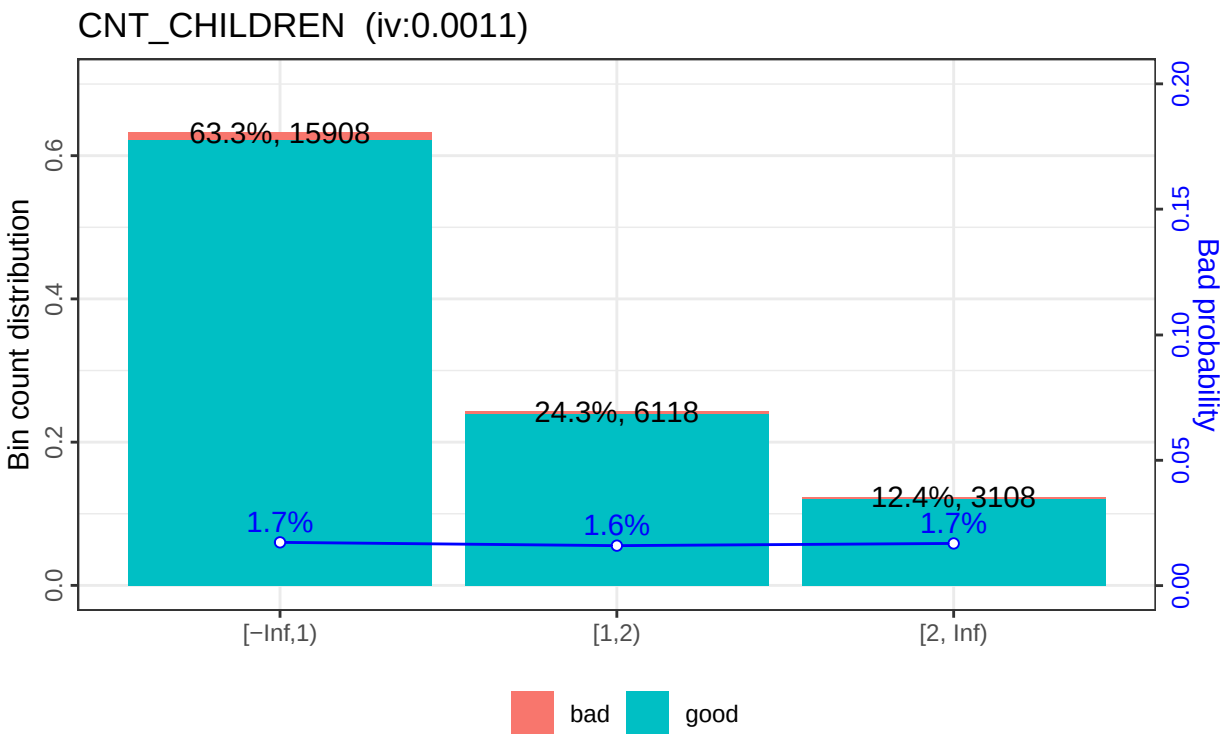
```
# Binning continuous variable to improve performance.
dl_bin <- woebin(converted_data,
  y = "y",
  x = c("CNT_CHILDREN", "AMT_INCOME_TOTAL", "DAYS_BIRTH", "DAYS_EMPLOYED", "CNT_FAM_MEMBERS"),
  method = "chimerge"
)
```

```
## [INFO] creating woe binning ...
```

Now we inspect the Binning variables. As for `CNT_CHILDREN` , we notice that it is discretizated into 3 categories, i.e. `[-inf, 1)`,`[1,2)`,`[2, inf)`. and the bad ratios for categories is linear, which means this binnning is good.

```
# plot binning variable
plotlist <- woebin_plot(dl_bin)
plotlist[1]
```

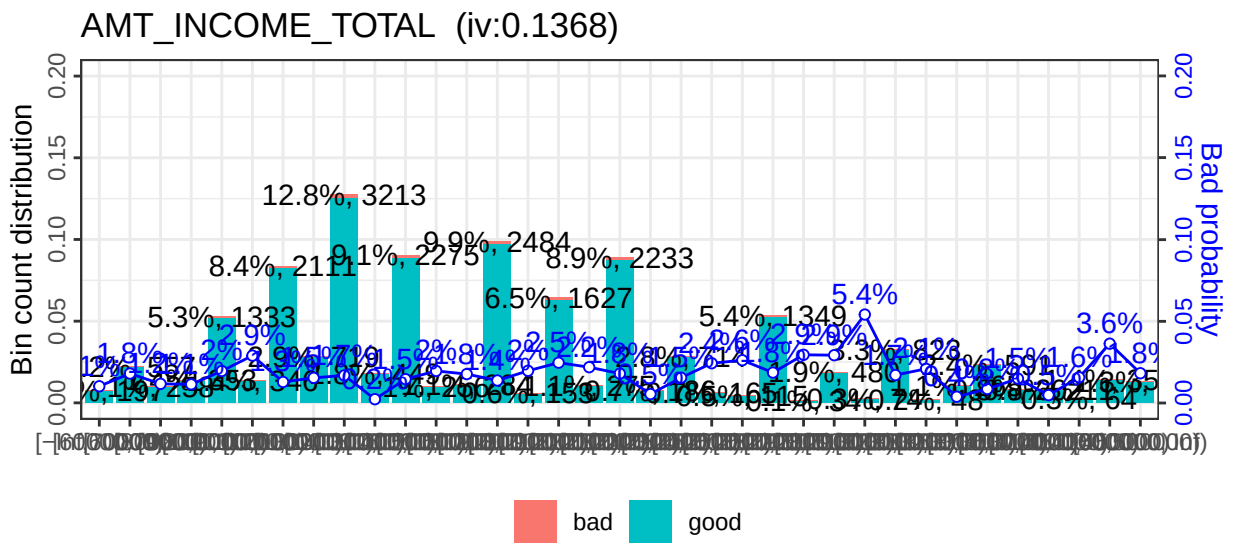
```
## $CNT_CHILDREN
```



As for `AMT_INCOME_TOTAL` , we notice that it is discretizated into several categories, although the bad ratios for categories is not strictly linear, this binning has second highest IV 0.1368.

```
# plot binning variable
plotlist <- woebin_plot(dl_bin)
plotlist[2]
```

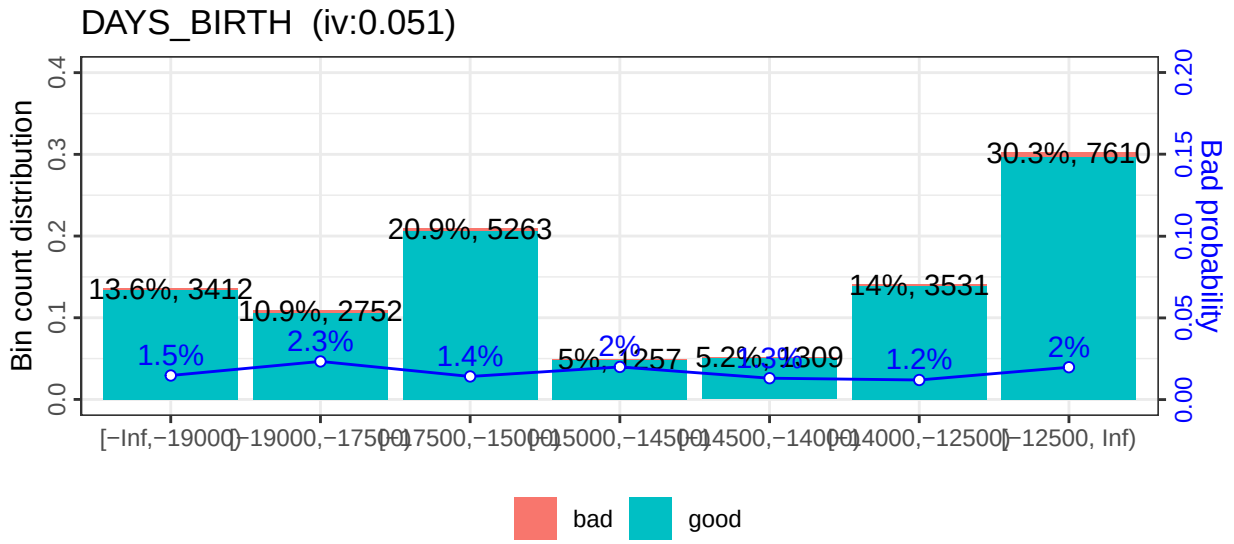
```
## $AMT_INCOME_TOTAL
```



As for `DAYS_BIRTH` , we notice that it is discretizated into 7 categories, i.e. `[-inf, -19000]`,`[-19000, -175000]` etc. and the bad ratios for categories is linear, which means this binnning is good.

```
# plot binning variable
plotlist <- woebin_plot(dl_bin)
plotlist[3]
```

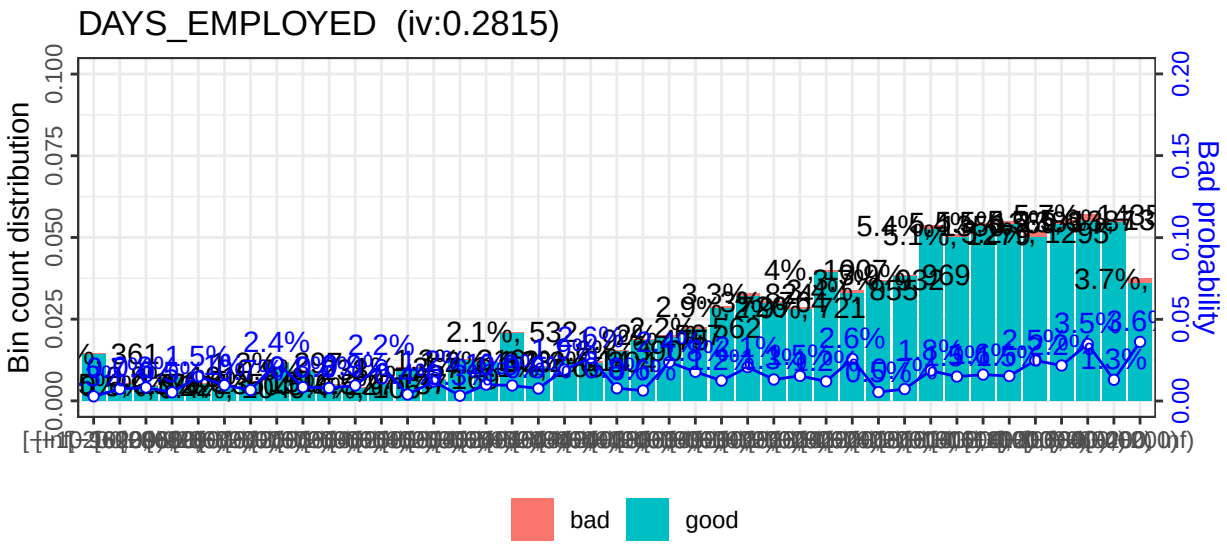
```
## $DAYS_BIRTH
```



As for `DAYS_EMPLOYED` , we notice that it is discretizated into several categories, although the bad ratios for categories is not strictly linear, this binning has highest IV 0.2815.

```
# plot binning variable
plotlist <- woebin_plot(dl_bin)
plotlist[4]
```

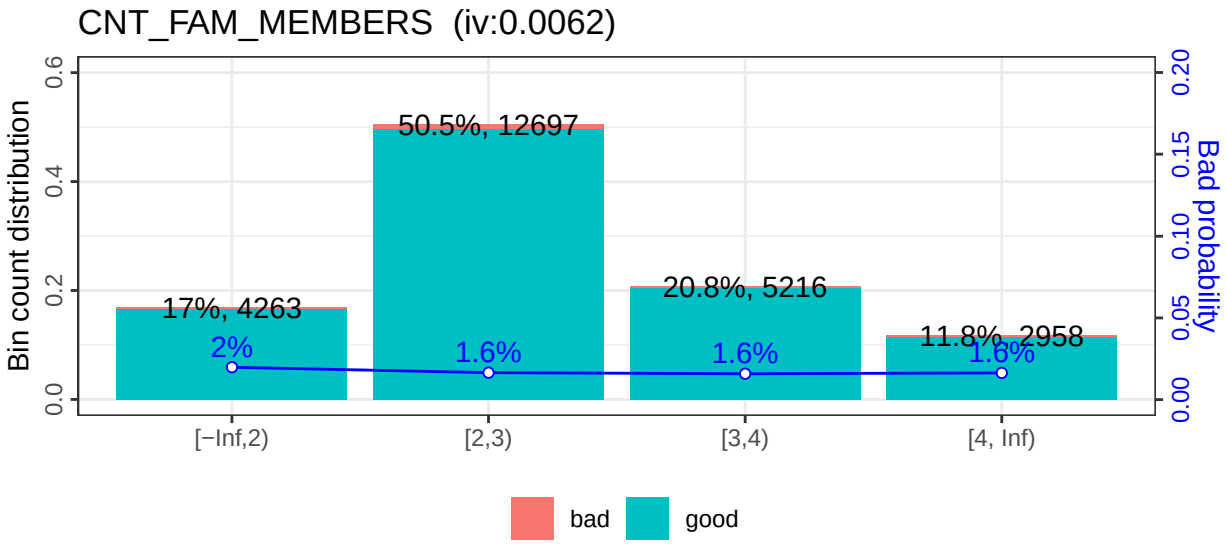
```
## $DAYS_EMPLOYED
```



As for `CNT_FAM_MEMBERS` , we notice that it is discretizated into 4 categories, i.e. `[-inf, 2)`,`[2, 3)`,`[3, 4)` and `[4, inf)`. and the bad ratios for categories is linear, which means this binnning is good.

```
# plot binning variable
plotlist <- woebin_plot(dl_bin)
plotlist[5]
```

```
## $CNT_FAM_MEMBERS
```



Finally we combine binning variables, binary factor variables, and category variables with original data to produce `dl_iv_data` and For the consistency of features engineering we repeat the whole process for mf_data and mice_data to produce `mf_iv_data` and `mf_mice_data`.

```
## [INFO] converting into woe values ...
```

```
## Rows: 25,134
## Columns: 35
## $ CODE_GENDER                                    <dbl> 2, 1, 1, 1, 1, 2, 2, ...
```

```
## $ FLAG_OWN_CAR                                  <dbl> 2, 1, 1, 1, 1, 2, 2, ...
## $ FLAG_OWN_REALTY                               <dbl> 2, 2, 2, 2, 2, 2, 2, ...
## $ FLAG_WORK_PHONE                               <int> 0, 0, 0, 0, 0, 1, 1, ...
## $ FLAG_PHONE                                    <int> 0, 1, 1, 1, 1, 1, 1, ...
## $ FLAG_EMAIL                                    <int> 0, 1, 1, 1, 1, 1, 1, ...
## $ opentime                                      <int> -29, -4, -26, -26, -3...
## $ y                                             <fct> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_edu_Academic.degree               <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_edu_Incomplete.higher             <int> 0, 0, 0, 0, 0, 1, 1, ...
## $ less_factor_edu_Lower.secondary               <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_edu_Secondary...secondary.special <int> 1, 1, 1, 1, 1, 0, 0, ...
## $ less_factor_house_House...apartment           <int> 1, 1, 1, 1, 1, 1, 1, ...
## $ less_factor_house_Municipal.apartment         <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_house_Office.apartment            <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_house_Rented.apartment            <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_house_With.parents                <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_income_Commercial.associate       <int> 0, 1, 1, 1, 1, 0, 0, ...
## $ less_factor_income_Pensioner                  <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_income_State.servant              <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_income_Student                    <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_income_Working                    <int> 1, 0, 0, 0, 0, 1, 1, ...
## $ less_factor_status_Civil.marriage             <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_status_Married                    <int> 1, 0, 0, 0, 0, 1, 1, ...
## $ less_factor_status_Separated                  <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_status_Single...not.married       <int> 0, 1, 1, 1, 1, 0, 0, ...
## $ less_factor_status_Widow                      <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_work_Labor                        <int> 1, 0, 0, 0, 0, 0, 0, ...
## $ less_factor_work_Office                       <int> 0, 1, 1, 1, 1, 1, 1, ...
## $ less_factor_work_higher                       <int> 0, 0, 0, 0, 0, 0, 0, ...
## $ CNT_CHILDREN_woe                              <dbl> 0.022243446, 0.022243...
## $ AMT_INCOME_TOTAL_woe                          <dbl> -0.276168616, 0.10050...
## $ DAYS_BIRTH_woe                                <dbl> -0.1382294, -0.138229...
## $ DAYS_EMPLOYED_woe                             <dbl> -0.1021481, 0.2099286...
## $ CNT_FAM_MEMBERS_woe                           <dbl> -0.02502654, 0.163028...
```

As for **mf__na__data** and **mice__na__data**, we produce the same process to keep the consistency of data, therefor we will get three datasets: **dl__iv__data**, **mf__iv__data** and **mice__iv__data**.

4.2 One-hot Encoding

Having a look at the whole dataset after deleting or imputation, we can figure out there are 8 nominal variables. And since all the nominal variables are categorically arranged, we can use the *one-hot-encoding* to deal with different kinds of nominals. The general idea of using one-hot-encoding is to mark the nominals(categories) into vectors(0 and 1 composed), in which if this particular category exists then mark the element with 1. When the entire encoding is done, there would be newly generated columns with category marked with name in place for the nominal variables. For example, when dealing with categorical variables "OCCUPATION_TYPE" which includes different types of occupation (e.g. Core staff, Sales staff, High skill tech staff, etc.), one particular type would be extracted out as a single vector with only 0 and 1 entered in where element 1 is marked for potential users corresponding to the same job. Similarily, the data processing with binary variables such as "GENDER" would be much easier to address.

```
library(dataPreparation)

# Compute encoding with category names
```

```
encoding <- build_encoding(completeData,
  cols = c(
    "CODE_GENDER", "FLAG_OWN_CAR", "FLAG_OWN_REALTY",
    "NAME_INCOME_TYPE", "NAME_EDUCATION_TYPE",
    "NAME_FAMILY_STATUS", "NAME_HOUSING_TYPE",
    "OCCUPATION_TYPE"
  ), verbose = TRUE
)

# Apply one hot encoding to turn the records of categories into 0,1 composed vectors
completeData <- one_hot_encoder(completeData, encoding = encoding, drop = TRUE)
```

And in order to generate the task for mlr3, we need to transform the target variable into factor type.

```
# convert all character y data into factor data type.
completeData <- data %>%
  mutate_if(is.character, as.factor) %>%
  mutate(y = as.factor(y))
head(completeData)
```

After the above preprocessing procedures with nominal variable addressed with one-hot encoding, the corresponding 3 datasets were generated accordingly which are named as *dl_oh_data*, *mf_oh_data* and *mice_oh_data*.

When finishing with the nominal variable encoding process, there are 6 datasets in general named with 2 groups: the *iv* group and the *one-hot* group. There are *dl_iv_data*, *mf_iv_data* and *mice_iv_data* included in the *iv* group while *dl_oh_data*, *mf_oh_data* and *mice_oh_data* included in the *one-hot* group.

5 Introduction to methods and metrices

5.1 Logistic regression

Logistic Regression is a Machine Learning classification algorithm that is used to predict the probability of a categorical dependent variable. In logistic regression, the dependent variable is a binary variable that contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.). In other words, the logistic regression model predicts P(Y=1) as is shown in formular of X.

$$P = (y = 1|x, \beta) = \frac{1}{1 + e^{-\beta^T x}}$$

5.2 KNN

KNN (k-nearest neighbors) is a method used for classification. In simple words, to classify one specific data point, it takes k neighbors with the shortest distance. Furthermore, based on how the neighbors are classified, we will assign the new input to the most popular category in the k neighbors.
The **KNN** package included in **mlr3** has the following parameters: **k** (the number of neighbors considered), **distance** (Parameter of Minkowski distance), **kernel** (kernel functions used to weight the neighbors). Here the number of k and the method of calculating the distances between data points are crucial. We will first evaluate how KNN performs between different data, including different encoding, and different missing value handling. Then we will explore different methods to get better results. First, we investigate them separately and then combine the knowledge to perform further fine-tuning to improve the model.

5.3 SVM

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimentional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

Correspondingly, we will use the SVM algorithm embedded in the **mlr3** learner and analyze the prediction performances with AUC (Area under the ROC Curve) values.

When it comes to model tuning for SVM, we consider the **kernel** function, **cost** (also known as **C-Regularization** parameter) and **gamma** to tune.

kernels: The main function of the kernel is to take low dimensional input space and transform it into a higher-dimensional space. It is mostly useful in non-linear separation problem.

cost (Regularisation): Cost is the penalty parameter, which represents misclassification or error term. The misclassification or error term tells the SVM optimisation how much error is bearable. This is how you can control the trade-off between decision boundary and misclassification term.

Gamma: It defines how far influences the calculation of plausible line of separation.

5.4 Random Forest

Random Forest, also known as Random Decision Forest, is an algorithm mainly used for dealing with classification problems. It consists of multiple decision trees and harnesses bagging and feature randomness in order to create an uncorrelated forest of trees whose prediction performance is more accurate than of any individual tree. With regards to model tuning for Random Rorest, there are mainly four variables to take into consideration. One of the crucial hyperparameters of RF is **mtry**, defined as how many variables to select at a node split when growing a decision tree. Generally, lower values of mtry will lead to more different, less correlated trees, arousing better stability while aggregating. As mentioned by Probst in the article Hyperparameters and tuning strategies for random forest, mtry values are sampled from the space $[0, p]$ with p being the number of predictor variables. Accordingly, we should set the range of mtry values to meet the above requirements. Secondly, the sample size parameter, **sample.fraction**, determines how many observations are drawn for the training of each tree. It has a similar effect as the mtry parameter. Decreasing the sample size leads to more diverse trees and thereby lower correlation between the trees, which

has a positive effect on the prediction accuracy when aggregating the trees. However, the accuracy of the single trees decreases, since fewer observations are used for training. Hence, similarly to the mtry parameter, the choice of the sample size can be seen as a trade-off between stability and accuracy of the trees. Therefore, sample size values shall be sampled from $[0.2n, 0.9n]$ with n being the number of observations. Moreover, the nodesize parameter, **min.node.size**, specifies the minimum number of observations in a terminal node. Setting it lower leads to trees with a larger depth which means that more splits are performed until the terminal nodes. Considering tuning with it (see Hyperparameters and tuning strategies for random forest), node size values should be sampled from with higher probability (in the initial design) for smaller values by sampling x from $[0, 1]$ and transforming the value by the formula $[(0.2n)^x]$. Last but not least, the number of trees in a forest is a parameter that is not tunable in the classical sense but should be set sufficiently high. According to measures based on the mean quadratic loss such as the mean squared error (in case of regression) or the Brier score (in case of classification), however, more trees are always better, as theoretically proved by Probst and Boulesteix (2017). Thence, we have decided to set it to be fixed with 1000 to meet the considerably latge number of obsearvations of records. Also, it can be vividly seen from the target dataset we have generated from the data preparation step that the distribution with target 1 and 0 are so imbalanced that we must train the learner with SMOTE function to get considerably balanced training data. Fortunately, in mlr3pipelines, there is a classbalancing and a smote pipe operator that can be combined with any learner. We have decided to combine the fixed tree number of learner with SMOTE function learner in order to get the optimal parameter settings for smote.K and smote.dup_size.

5.5 Metrics

5.5.1 ROC curve and AUC score

A receiver operating characteristic curve, or ROC curve, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate $TPR = \frac{TP}{TP+FN}$ against the false positive rate $FPR = \frac{FP}{FP+TN}$ at various threshold settings. The true-positive rate is also known as sensitivity, reacall or probablitly of detection in machine learning. The false-positive rate is also known as probability of false alarm and can be calculated as (1 - specificity). It can also be thought of as a plot of the power as fa cuntion of the Type I Error of the decision rule. Auc stands for “Area under the Roc Curve”, it measures the entire two-dimensional area underneath the entire ROC curve form (0,0) to (1,1). It tells how much model is capable of distinguishing between classes. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s.

5.5.2 Why choose AUC and ROC

In our experiment, we looked at several different measures to evaluate our result, including fbeta, ce, acc and AUC. Ingeneral, using fbeta would be ideal when facing imbalanced data[recite]. However, our data is highly imbalanced, and the fbeta performs in favour of predicting all as the majority category. It outputs, in most cases with a very similar value close to 0.99. Thus, we decided to stick with AUC. By comparing the AUC value, we can better distinguish the improvement in models.

6 Choosing the training task

As for the process concerning the general model tuning, we would firstly go with the 6 different datasets corresponding to 6 training tasks and train the 4 models with default parameter settings and see which task to choose according to the ultimate performance output based on the best AUC level. By doing these, the training workload for the following tuning procedures would be tremendously decreased as we have considerably more learners to train the task with.

6.1 Models results

first we load all the datasets. As the requests of **mlr3**, all integer numeric should be converted as double numeric, and target **y** should be converted as type factor.

```
# loading library
library(tidyverse)
# set seed
set.seed(2020)
# load data
dl_iv_data <- read.csv2("./data/dl_iv_data.csv") %>%
  mutate_if(is.integer, as.numeric) %>%
  mutate(y = as.factor(y))
mf_iv_data <- read.csv2("./data/mf_iv_data.csv") %>%
  mutate_if(is.integer, as.numeric) %>%
  mutate(y = as.factor(y))
mice_iv_data <- read.csv2("./data/mice_iv_data.csv") %>%
  mutate_if(is.integer, as.numeric) %>%
  mutate(y = as.factor(y))
dl_oh_data <- read.csv("./data/dl_oh_data.csv") %>%
  mutate_if(is.integer, as.numeric) %>%
  mutate(y = as.factor(y))
mf_oh_data <- read.csv("./data/mf_oh_data.csv") %>%
  mutate_if(is.integer, as.numeric) %>%
  mutate(y = as.factor(y))
mice_oh_data <- read.csv("./data/mice_oh_data.csv") %>%
  mutate_if(is.integer, as.numeric) %>%
  mutate(y = as.factor(y))
```

then we creat and combine all the tasks into a list.

```
library(mlr3)
# creat task for all the data
# creat task without dm
task_all <- list(
  TaskClassif$new("dl_iv", backend = dl_iv_data, target = "y"),
  TaskClassif$new("mf_iv", backend = mf_iv_data, target = "y"),
  TaskClassif$new("mice_iv", backend = mice_iv_data, target = "y"),
  TaskClassif$new("dl_oh", backend = dl_oh_data, target = "y"),
  TaskClassif$new("mf_oh", backend = mf_oh_data, target = "y"),
  TaskClassif$new("mice_oh", backend = mice_oh_data, target = "y")
)
```

for each model, we work the same procedures.
we all use **benchmark_grid** to creat design, setting with **5 folds cv**, **predict_type = "prob"** and **measure = msr("classif.auc")** for computing AUC. In addition, we fix random seed as **2020** for reproducing.

6.1.1 Logistic Regression:

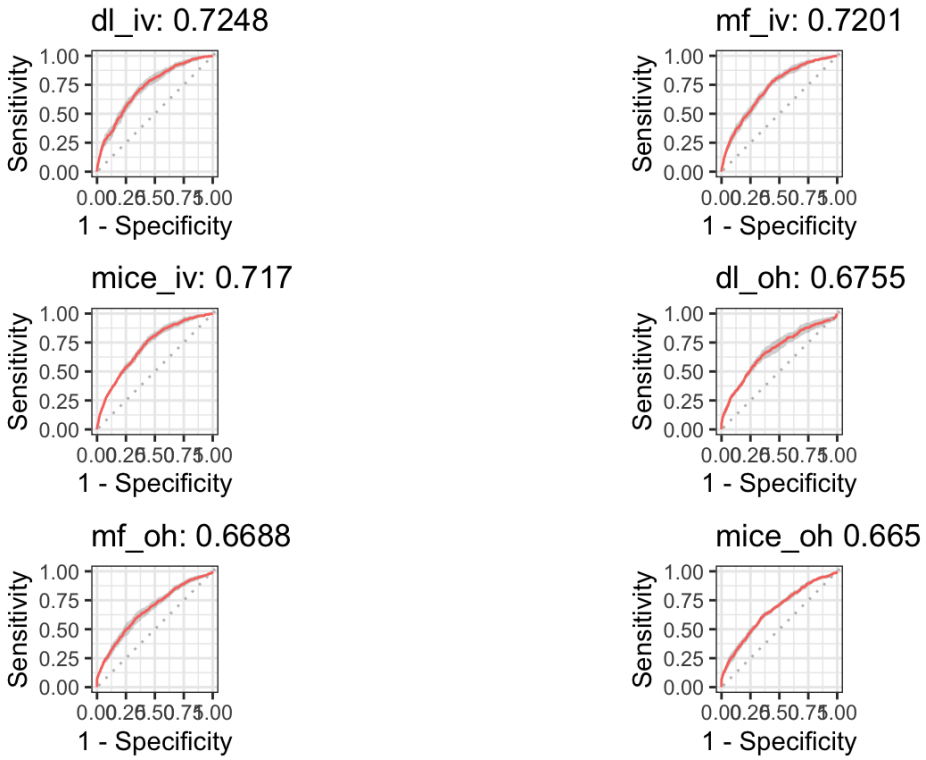
We will first evaluate how Logistic Regression performs between different data, including different encoding, and different missing value handling. As mentioned above, we pass arguments to **benchmark** function from **mlr3** to train logsitic regression.

```
# load library
library(mlr3learners)
library(mlr3tuning)
library(paradox)

# creat a benchmark
design <- benchmark_grid(
  tasks = task_all,
  learners = lrn("classif.log_reg", predict_type = "prob"),
  resampling = rsmp("cv", folds = 5L)
)

# set measure
all_measures <- msr("classif.auc")
# run the benchmark
set.seed(2020)
lg_bmr <- benchmark(design, store_models = TRUE)
# save the results
lg_results <- lg_bmr$aggregate(measures = msr("classif.auc"))
```

For better comparing with each task, we write **multiplot_roc** function to plots results.

```
library(mlr3viz)

# plot for the 6 data tasks
multiplot_roc <- function(models, type = "roc") {
  # set a null list
  plots <- list()
  # clone results
  model <- models$clone()$filter(task_id = "dl_iv")
  # format AUC
  auc <- round(model$aggregate(msr("classif.auc"))[[7]], 4)
  # plot
  plots[[1]] <- autoplot(model, type = type) + ggtitle(paste("dl_iv:", auc))

  model <- models$clone()$filter(task_id = "mf_iv")
  auc <- round(model$aggregate(msr("classif.auc"))[[7]], 4)
  plots[[2]] <- autoplot(model, type = type) + ggtitle(paste("mf_iv:", auc))

  model <- models$clone()$filter(task_id = "mice_iv")
  auc <- round(model$aggregate(msr("classif.auc"))[[7]], 4)
  plots[[3]] <- autoplot(model, type = type) + ggtitle(paste("mice_iv:", auc))

  model <- models$clone()$filter(task_id = "dl_oh")
  auc <- round(model$aggregate(msr("classif.auc"))[[7]], 4)
  plots[[4]] <- autoplot(model, type = type) + ggtitle(paste("dl_oh:", auc))

  model <- models$clone()$filter(task_id = "mf_oh")
  auc <- round(model$aggregate(msr("classif.auc"))[[7]], 4)
```

```
  plots[[5]] <- autoplot(model, type = type) + ggtitle(paste("mf_oh:", auc))

  model <- models$clone()$filter(task_id = "mice_oh")
  auc <- round(model$aggregate(msr("classif.auc"))[[7]], 4)
  plots[[6]] <- autoplot(model, type = type) + ggtitle(paste("mice_oh", auc))

  do.call("grid.arrange", plots)
}
multiplot_roc(lg_bmr)
```



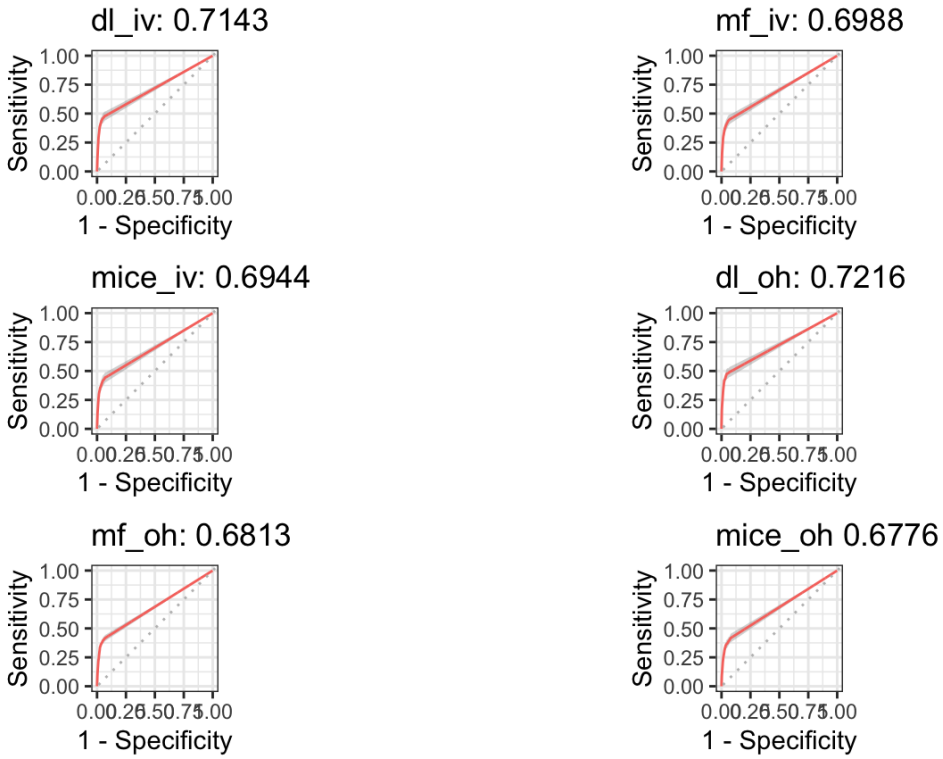
From the ROC plots above, we can see that as for different encoding methods, tasks tackled with WOE encoding generally perform 0.05 better with the AUC score than the corresbonding counterpart binary encoding group, as for different missing data handling methods, Logistic Regression has better prediction performance on datasets by deleting missing values with whole row, i.e. **dl_iv_data**, **dl_oh_data**. Above all, we choose **dl_iv_data** as final datasets.

6.1.2 KNN

We will first evaluate how KNN performs between different data, including different encoding, and different missing value handling. Then we will explore different methods to get better results.

```
# creat a benchmark
design <- benchmark_grid(
  tasks = task_all,
  learners = lrn("classif.kknn", predict_type = "prob"),
  resampling = rsmp("cv", folds = 5L)
)

# set measure
all_measures <- msr("classif.auc")
```

```r
# run the benchmark
set.seed(2020)
knn_bmr <- benchmark(design, store_models = TRUE)
# save the results
knn_results <- knn_bmr$aggregate(measures = msr("classif.auc"))

multiplot_roc(knn_bmr)
```

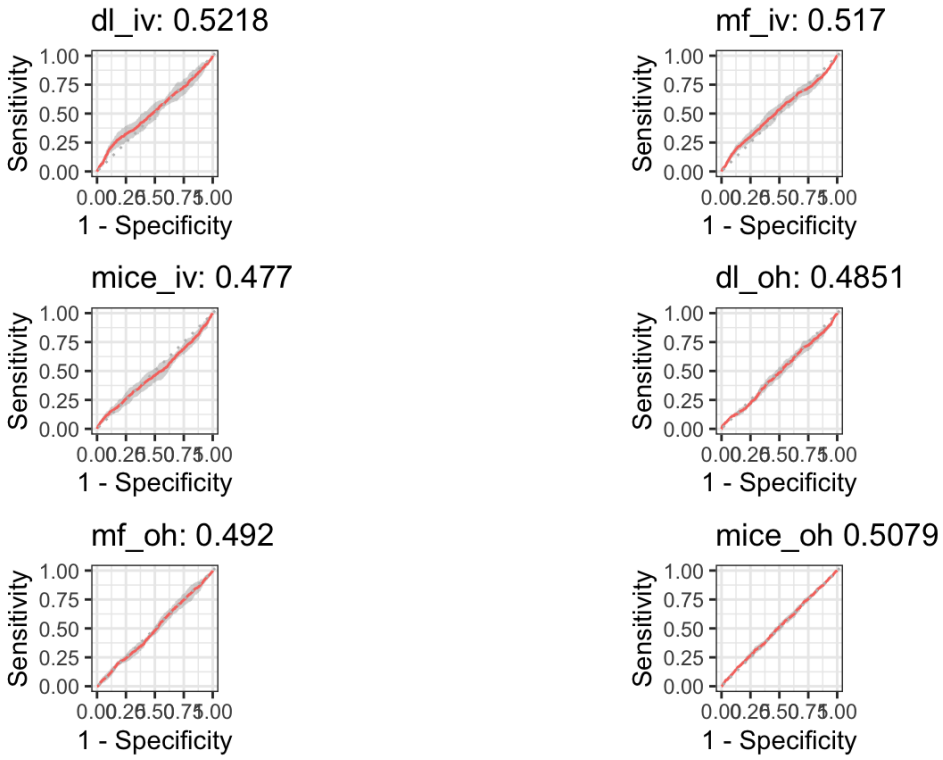


From the ROC plots above, we can see that KNN performs with no significant difference between different encoding and missing data handling methods. Moreover, To reduce our computation cost, we decided to take the task with the highest AUC value in this step, being dl_iv. Now, we will focus on the task dl_iv, and fine-tune the parameters.

6.1.3 SVM

```r
# create a benchmark to train a list of tasks with default hyperparameter settings
design <- benchmark_grid(
  tasks = task_all,
  learners = lrn("classif.svm", predict_type = "prob", kernel = "linear"),
  resampling = rsmp("cv", folds = 5L)
)
# set measure
all_measures <- msr("classif.auc")

# run the benchmark to  train the model
svm_bmr <- benchmark(design, store_models = TRUE)
# save the results
svm_results <- svm_bmr$aggregate(all_measures)
```

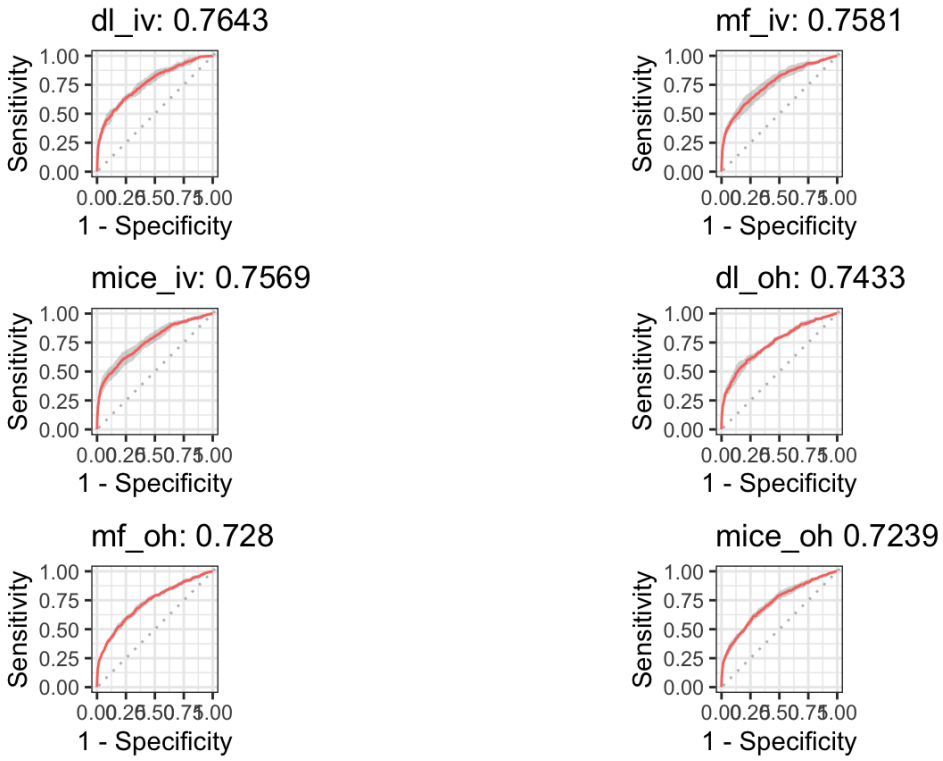
```
multiplot_roc(svm_bmr)
```



Similarly, we go with the **dl_iv_data** training task.

6.1.4 Random Forest

```
# create a benchmark to train a list of tasks with default hyperparameter settings
design <- benchmark_grid(
  tasks = task_all,
  learners = lrn("classif.ranger", predict_type = "prob"),
  resampling = rsmp("cv", folds = 5L)
)
# set measure
all_measures <- msr("classif.auc")

# run the benchmark to  train the model
rf_bmr <- benchmark(design, store_models = TRUE)
# save the results
rf_results <- rf_bmr$aggregate(all_measures)

multiplot_roc(rf_bmr)
```



Similarly, we go with the **dl_iv_data** training task.

6.1.5 Conclusion

according to the results above, **dl_iv_data** generally has the best prediction performance among all the six tasks with regards to 4 different models. Hence **dl_iv_data** is chosen as the final training dataset.

7 Finetunning

Based on the imbalanced target dataset we already have, it is crucial that we need to firstly address the imbalance problem. When it comes to deal with imbalanced data, there are two options to choose. The first method is the **smote** and the other is the **oversampling**. In mlr3pipelines, there is a **classbalancing** and a **smote** pipe operator that can be combined with any learner. Note that **smote** has two hyperparameters **K** and **dup_size**. While **K** changes the behavior of the SMOTE algorithm, **dup_size** will affect oversampling rate of SMOTE algorithm. As for **oversampling** oeperator, there is only one hyperparameter **oversample.ratio**, which determines the oversampling rate of **oversampling** operator.

To focus on the effect of the oversampling rate on the performance, we will consider different parameter settings with regards to the **oversampling** pipe operator which can be adapted to different training models.

when it comes to hyperparameter tuning, we need to set different learners based on the **mlr3 package** to tune tasks. The next step is to set for the hyperparameters which are used to tune with different ranges or factor levels.

We create a list of AutoTuner classes of the 4 different learners (Random Forest, SVM, Logistic and KNN) to tune the graph (4 different model learners + imbalance data correction method) based on a 3-fold CV using the classif.AUC as the performance measure. To keep runtime low, we define the resolution with grid_search to 5. However, one can also jointly tune the hyperparameter of the learner along with the imbalance correction method by extending the search space with the learner's hyperparameters. Note that SMOTE has two hyperparameters **K** and **dup_size**. While **K** changes the behavior of the SMOTE algorithm, **dup_size** will affect oversampling rate. To focus on the effect of the oversampling rate on the performance, we will consider different parameter settings with regards to the **oversampling** pipe operator which can be adapted to different training models.

Therefore, we will need to firstly find the optimal settings for **smote.K** and **smote.dup_size** to correct the imbalanced target data distribution. Hence, we use grid search with the corresponding hyperparameter configurations for the oversampling method and each SMOTE variant for tuning. The **AutoTuner** is a fully tuned graph that behaves like a usual learner. For the innner tuning, a 3-fold CV is used. Now, we use the **benchmark** function to compare the tuned class imbalance pipeline graphs based on a 3-fold CV for the outer evaluation. After all these tuning procedures mentioned above, we can therefore draw different parameter combinations with **ggplot** and see what the general trend is with different parameter ranges or levels to choose the optimal hyperparameter combinations for the best prediction performance. Also one thing to mention, we would generally use the y pixel as the AUC value, x pixel as the tuning parameter and the color parameter as well to show the different tendencies if necessary. Ultimately, after all these step-to-step tuning procedures, we can grab the best performance (measured by AUC value) with the recommended parameter settings and therefore improve our default models.

7.1 Logistic Regression

7.1.1 Oversampling

At first we define the oversampling PipeOps.

```
# load library
library(mlr3learners)
library(mlr3tuning)
library(mlr3pipelines)
library(paradox)
task <- TaskClassif$new("dl_iv", backend = dl_iv_data, target = "y")
# logistic learner
lg_learner <- lrn("classif.log_reg", predict_type = "prob")
# po_smote = po("smote", dup_size = 6)
po_over <- po("classbalancing",
  id = "oversample", adjust = "minor",
  reference = "minor", shuffle = FALSE, ratio = 6
)
```

```
# creat oversample
lg_over_learner <- GraphLearner$new(po_over %>>% lg_learner, predict_type = "prob")
```

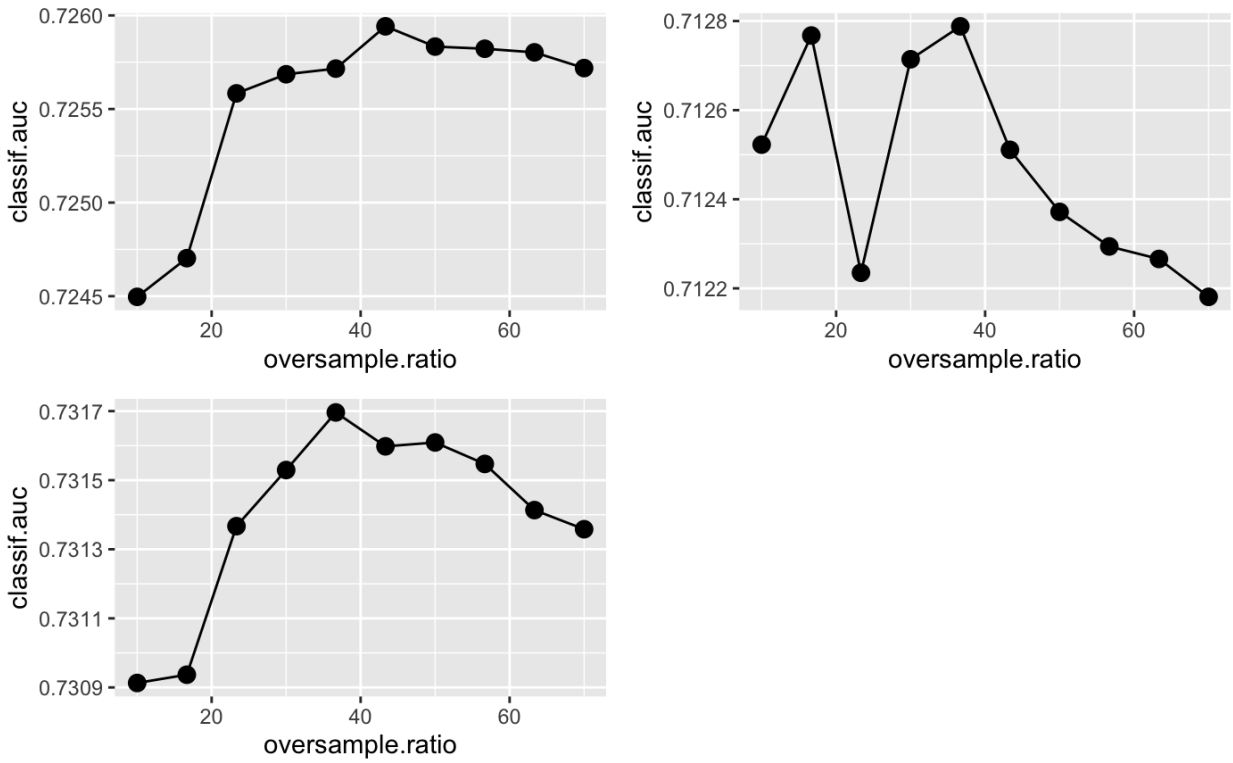
As for **classbalancing**, **oversample.ratio** is the hyperparamter we must set, ranging from 10 to 70. We define the search space in order to tune the hyperparameters of the class imbalance methods. We create an AutoTuner class from the learner to tune the graph based on a 3-fold CV using the classif.auc as performance measure.

```
lg_over_param_set <- ParamSet$new(list(ParamDbl$new("oversample.ratio", lower = 10, upper = 70)))
terms <- term("none")
inner_rsmp <- rsmp("cv", folds = 5L)
lg_over_auto <- AutoTuner$new(
  learner = lg_over_learner, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = lg_over_param_set,
  terminator = terms, tuner = tnr("grid_search", resolution = 10)
)
# set outer_resampling, and creat a design with it
outer_rsmp <- rsmp("cv", folds = 3L)
lg_over_design <- benchmark_grid(
  tasks = task,
  learners = lg_over_auto,
  resamplings = outer_rsmp
)
```

With **store_models = TRUE** we allow the benchmark function to store each single model that was computed during tuning. Therefore, we can plot the tuning path of the best learner from the subsampling iterations:

```
set.seed(2020)
lg_over_bmr <- benchmark(lg_over_design, store_models = TRUE)
lg_over_results <- lg_over_bmr$aggregate(measures = msr("classif.auc"))
# inspect AUC value.
lg_over_results$classif.auc

# load library
library(ggplot2)
library(ggpubr)
# plot path for each cv
over_path1 <- lg_over_bmr$data$learner[[1]]$archive("params")
over_gg1 <- ggplot(over_path1, aes(
  x = oversample.ratio,
  y = classif.auc
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

over_path2 <- lg_over_bmr$data$learner[[2]]$archive("params")
over_gg2 <- ggplot(over_path2, aes(
  x = oversample.ratio,
  y = classif.auc
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")
```

```
over_path3 <- lg_over_bmr$data$learner[[3]]$archive("params")
over_gg3 <- ggplot(over_path3, aes(
  x = oversample.ratio,
  y = classif.auc
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")
# summary plots
ggarrange(over_gg1, over_gg2, over_gg3, common.legend = TRUE, legend = "bottom")
```



shown on the plots above, it is better to choose **oversample.ratio** with value 40 in order to show a considerably good prediction performance measured around 0.73.
In comparison with defalut **dl_iv_data**, there is no significant increment. Thence, oversample method seems not suitable for Logistic Regression.

7.1.2 Smote

At first we define the smote PipeOps.

```
po_smote <- po("smote", dup_size = 50)
# creat smote
lg_smote_learner <- GraphLearner$new(po_smote %>>% lg_learner, predict_type = "prob")
```

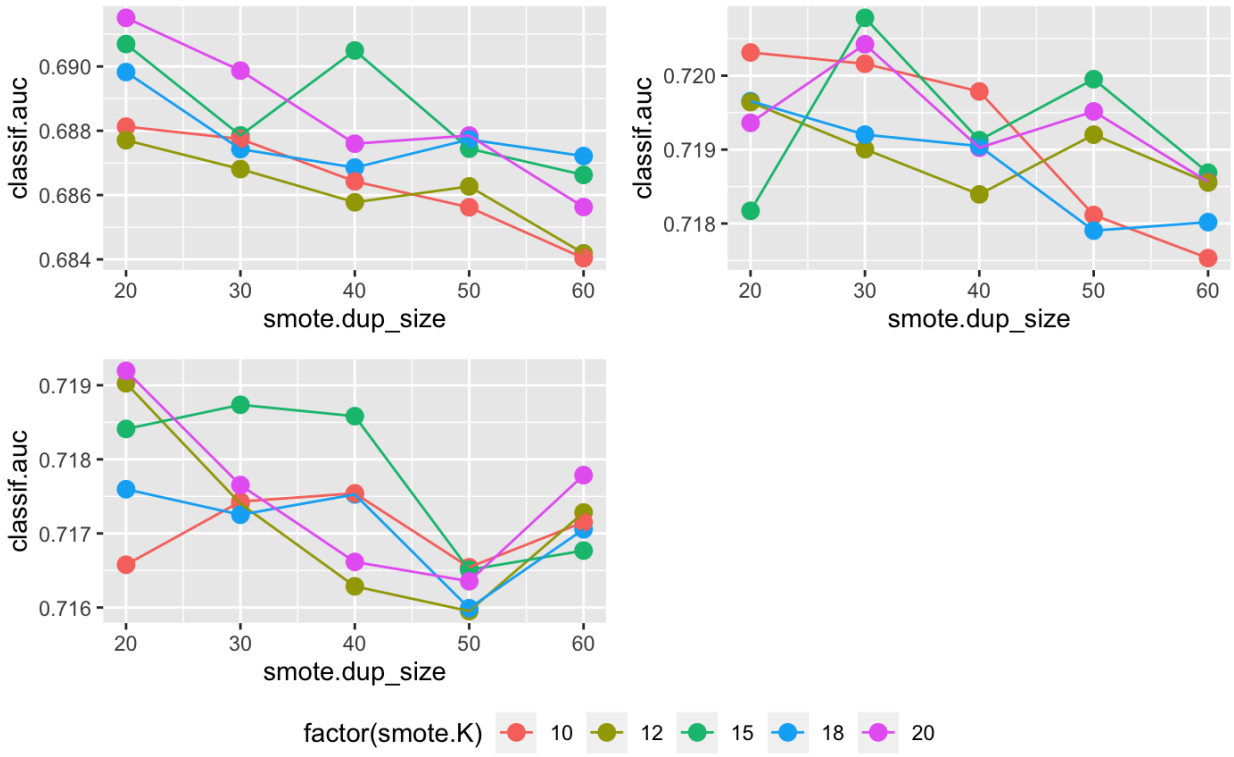
As for **smote**, we set hyperparamters **smote.K** ranging from 20 to 60 and **smote.dup_size** ranging from 10 to 20. We define the search space in order to tune the hyperparameters of the class imbalance methods. other configurations remain mentioned above.

```
lg_smote_param_set <- ParamSet$new(params = list(
  ParamInt$new("smote.dup_size", lower = 20, upper = 60),
  ParamInt$new("smote.K", lower = 10, upper = 20)
```

```
))
# set outer_resampling, and creat a design with it
terms <- term("none")
inner_rsmp <- rsmp("cv", folds = 3L)
lg_smote_auto <- AutoTuner$new(
  learner = lg_smote_learner, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = lg_smote_param_set,
  terminator = terms, tuner = tnr("grid_search", resolution = 5)
)

# set outer_resampling, and creat a design with it
outer_rsmp <- rsmp("cv", folds = 3L)
lg_smote_design <- benchmark_grid(
  tasks = task,
  learners = lg_smote_auto,
  resamplings = outer_rsmp
)
```

With **store_models = TRUE** we allow the benchmark function to store each single model that was computed during tuning. Therefore, we can plot the tuning path of the best learner from the subsampling iterations:

```
set.seed(2020)
# run benchmark
lg_smote_bmr <- benchmark(lg_smote_design, store_models = TRUE)
lg_smote_results <- lg_smote_bmr$aggregate(measures = msr("classif.auc"))
lg_smote_results$classif.auc
# plot path
over_path1 <- lg_smote_bmr$data$learner[[1]]$archive("params")
over_gg1 <- ggplot(over_path1, aes(
  x = smote.dup_size,
  y = classif.auc, col = factor(smote.K)
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

over_path2 <- lg_smote_bmr$data$learner[[2]]$archive("params")
over_gg2 <- ggplot(over_path2, aes(
  x = smote.dup_size,
  y = classif.auc, col = factor(smote.K)
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

over_path3 <- lg_smote_bmr$data$learner[[3]]$archive("params")
over_gg3 <- ggplot(over_path3, aes(
  x = smote.dup_size,
  y = classif.auc, col = factor(smote.K)
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

ggarrange(over_gg1, over_gg2, over_gg3, common.legend = TRUE, legend = "bottom")
```



shown on the plots above, it is better to choose **smote.K** with value 40 and **smote.dup_size** with value 15 in order to show a considerably good prediction performance measured around **0.72**.
In comparison with defalut **dl_iv_data**, there is no significant increment. Thence, smote method seems not suitable for Logistic Regression.

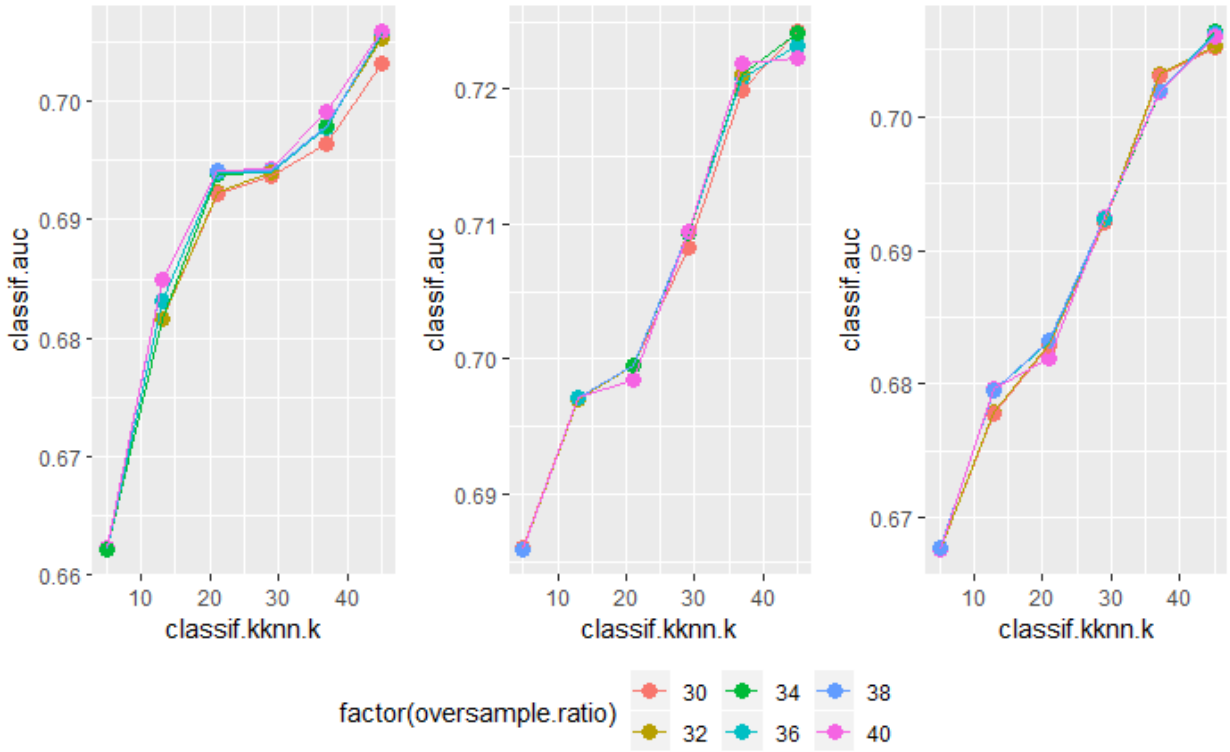
7.2 KNN

7.2.1 Oversampling

Since our data is highly unbalanced, we want to try two different methods to solve this issue - Smote and oversampling and see if this increases the performance of the model.

```
knn_learner <- lrn("classif.kknn", predict_type = "prob")
po_over = po("classbalancing",
             id = "oversample", adjust = "minor",
             reference = "minor", shuffle = FALSE, ratio = 6)

lrn_over <- GraphLearner$new(po_over %>>% knn_learner, predict_type = "prob")

# setting the tunning for parameters, and terminator
knn_param_set <- ParamSet$new(list(ParamInt$new("classif.kknn.k", lower = 5, upper = 45),
                           ParamDbl$new("oversample.ratio", lower = 30, upper = 40)))

terms <- term("none")

# creat autotuner, using the inner sampling and tuning parameter with grid_search
inner_rsmp <- rsmp("cv",folds = 5L)
knn_auto <- AutoTuner$new(learner = lrn_over, resampling = inner_rsmp,
                          measures = msr("classif.auc"), tune_ps = knn_param_set,
                          terminator = terms, tuner = tnr("grid_search", resolution = 6))
```

```r
# set outer_resampling, and creat a design with it
outer_rsmp <- rsmp("cv", folds = 3L)
design = benchmark_grid(
  tasks = task,
  learners = knn_auto,
  resamplings = outer_rsmp
)

# set seed before traing, then run the benchmark
# save the results afterwards
set.seed(2020)
knn_bmr <- benchmark(design, store_models = TRUE)

# plot results from 3 outer sampling rounds
library(ggplot2)

over_path1 = knn_bmr$data$learner[[1]]$archive("params")
over_gg1 = ggplot(over_path1, aes(
  x = classif.kknn.k,
  y = classif.auc, col = factor(oversample.ratio))) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

over_path2 = knn_bmr$data$learner[[2]]$archive("params")
over_gg2 = ggplot(over_path2, aes(
  x = classif.kknn.k,
  y = classif.auc, col = factor(oversample.ratio))) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

over_path3 = knn_bmr$data$learner[[3]]$archive("params")
over_gg3 = ggplot(over_path3, aes(
  x = classif.kknn.k,
  y = classif.auc, col = factor(oversample.ratio))) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")

library(ggpubr)
ggarrange(over_gg1, over_gg2, over_gg3, common.legend = TRUE, legend="bottom", nrow=1)
```

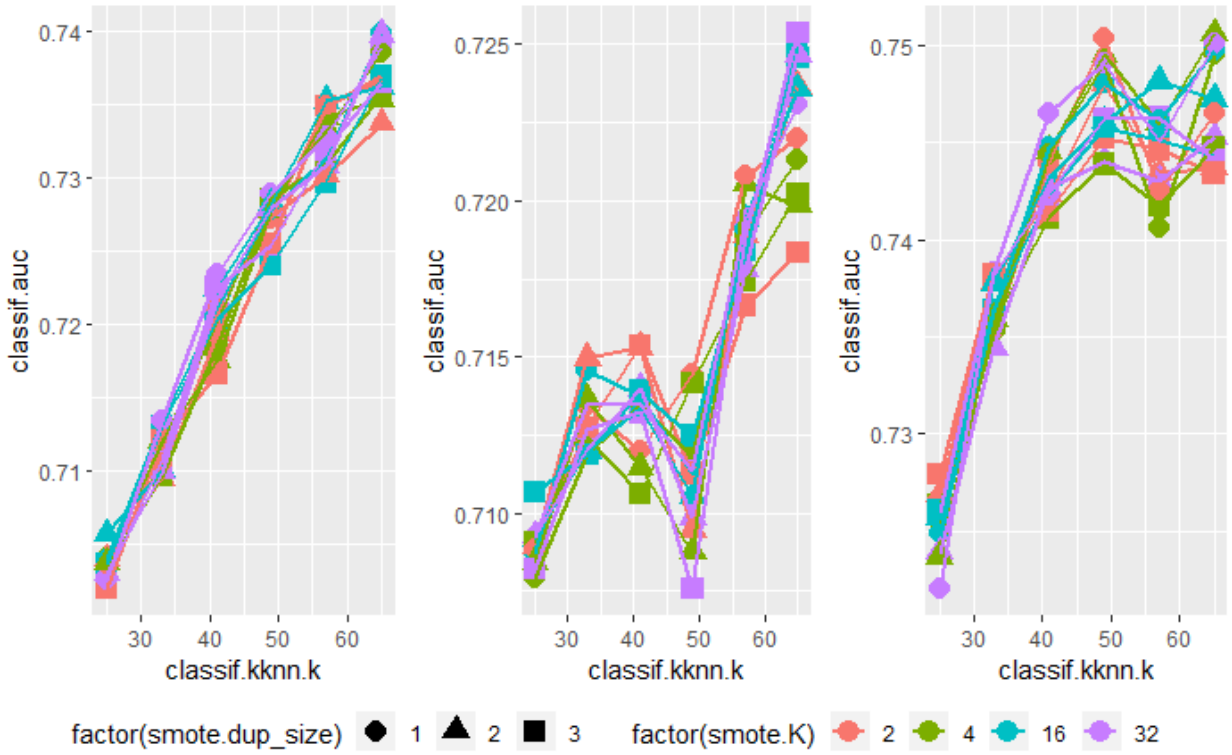


7.2.2 Smote

In this section, we will use the package **SMOTE** that is included in **mlr3**, to see whethere balancing the data with smote may improve the prediction of the model.

```
# knn learner
knn_learner <- lrn("classif.kknn", predict_type = "prob", distance=1, kernel= "inv")
po_smote = po("smote", dup_size = 6)
lrn_smote <- GraphLearner$new(po_smote %>>% knn_learner, predict_type = "prob")

# setting the tunning for parameters
knn_param_set <- ParamSet$new(params = list(
  ParamInt$new("classif.kknn.k", lower = 25, upper = 65),
  ParamInt$new("smote.dup_size", lower = 1, upper = 3),
  ParamInt$new("smote.K", lower = 1, upper = 5)))

# make grow smote.k exponential 2^n
knn_param_set$trafo = function(x, param_set) {
  x$smote.K = round(2^(x$smote.K))
  x
}

# creat autotuner, using the inner sampling and tuning parameter with grid_search
inner_rsmp <- rsmp("cv",folds = 5L)
terms <- term("none")
knn_auto <- AutoTuner$new(learner = lrn_smote, resampling = inner_rsmp,
                          measures = msr("classif.auc"), tune_ps = knn_param_set,
                          terminator = terms,
                          tuner = tnr("grid_search", resolution = 6))
```

```r
# set outer_resampling, and creat a design with it
outer_rsmp <- rsmp("cv", folds = 3L)
design = benchmark_grid(
  tasks = task,
  learners = knn_auto,
  resamplings = outer_rsmp
)

# set seed before traing, then run the benchmark
# save the results afterwards
set.seed(2020)
knn_bmr <- benchmark(design, store_models = TRUE)

# plot results from 3 outer sampling rounds
library(ggplot2)
stune_path1 = knn_bmr$data$learner[[1]]$archive("params")
stune_gg1 = ggplot(stune_path1, aes(
  x = classif.kknn.k,
  y = classif.auc, col = factor(smote.K), shape = factor(smote.dup_size))) +
  geom_point(size = 4) +
  geom_line(size=1)

stune_path2 = knn_bmr$data$learner[[2]]$archive("params")
stune_gg2 = ggplot(stune_path2, aes(
  x = classif.kknn.k,
  y = classif.auc, col = factor(smote.K), shape = factor(smote.dup_size))) +
  geom_point(size = 4) +
  geom_line(size=1)

stune_path3 = knn_bmr$data$learner[[3]]$archive("params")
stune_gg3 = ggplot(stune_path3, aes(
  x = classif.kknn.k,
  y = classif.auc, col = factor(smote.K), shape = factor(smote.dup_size))) +
  geom_point(size = 4) +
  geom_line(size=1)

library(ggpubr)
ggarrange(stune_gg1, stune_gg2, stune_gg3, common.legend = TRUE, legend="bottom", nrow=1)
```



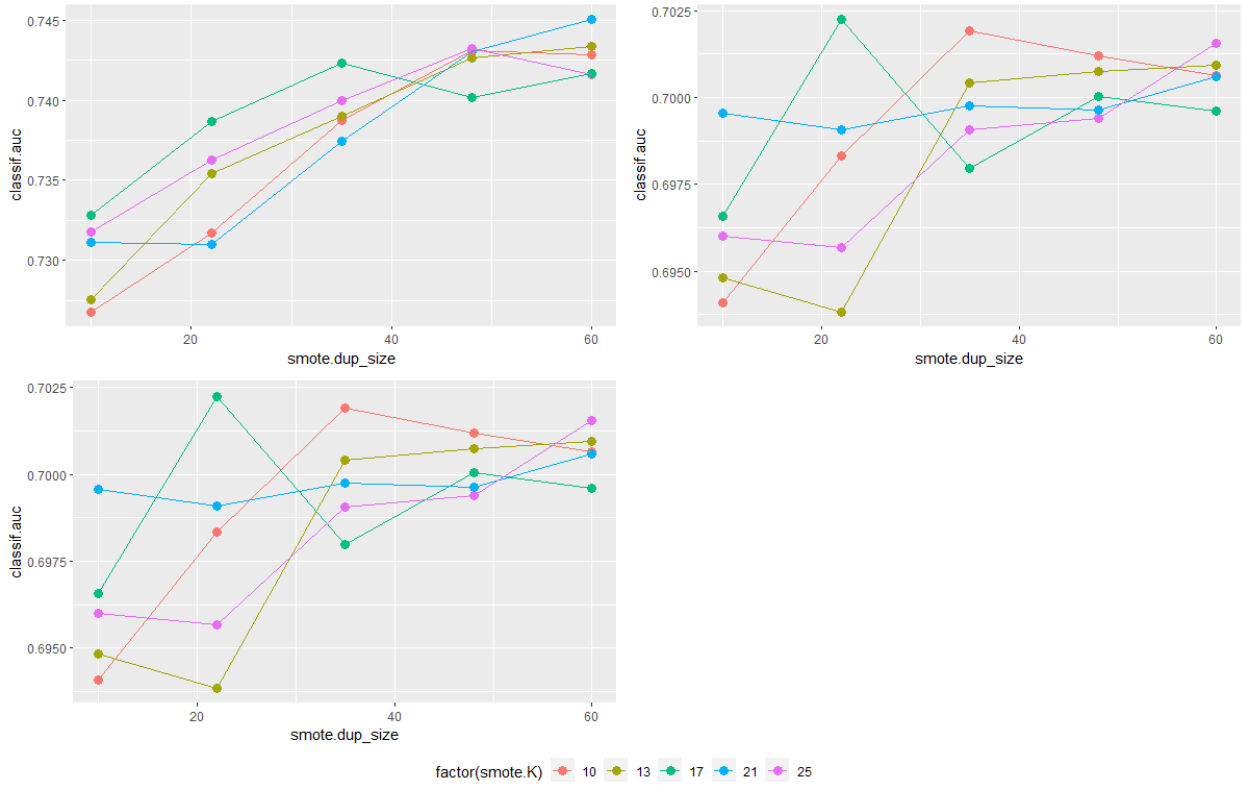
As we can see, there is no significant improvement after trying different methods to balance the data. Since we used a binary variable to indicate whether a category is present or not, the max distance can only be 1 or 0. Moreover, other numeric variables have a more significant distance, meaning that they have a more substantial impact on the distance than the categorical data without having a significant correlation with our target variable. To get better results, it would be necessary to either use other ways to handle categorical data better for distance calculation or using different training methods to perform classification instead of KNN.

7.3 SVM (smote)

Considering the super imbalanced target dataset to process, it is also necessary to implement the **SMOTE** function to firstly balance the whole dataset. Therefore, to determine the two crucial parameters **smote.dup_size** and **smote.K** is the first step. With **smote.dup_size** defined from 10 to 60 and **smote.K** from 10 to 25, it is better to implement a **grid_search** to figure out what is the optimal parameter settings for the 2 parameters based on the **svm** learner and the **dl_iv_data** training task.

```
# suppress svm package
suppressPackageStartupMessages(library(e1071))
## test for best matches with smote function's parameter with SVM learner
svm_lrn <- lrn("classif.svm", predict_type = "prob")
# train with smote function's 2 parameters
param_smote <- ParamSet$new(params = list(
  ParamInt$new("smote.dup_size", lower = 10, upper = 60),
  ParamInt$new("smote.K", lower = 10, upper = 25)
))
# inner resampling set
inner_rsmp <- rsmp("cv", folds = 3L)
po_smote <- po("smote", dup_size = 50)
# create smote learner fixed in svm learner
svm_smote_lrn <- GraphLearner$new(po_smote %>>% svm_lrn, predict_type = "prob")
```

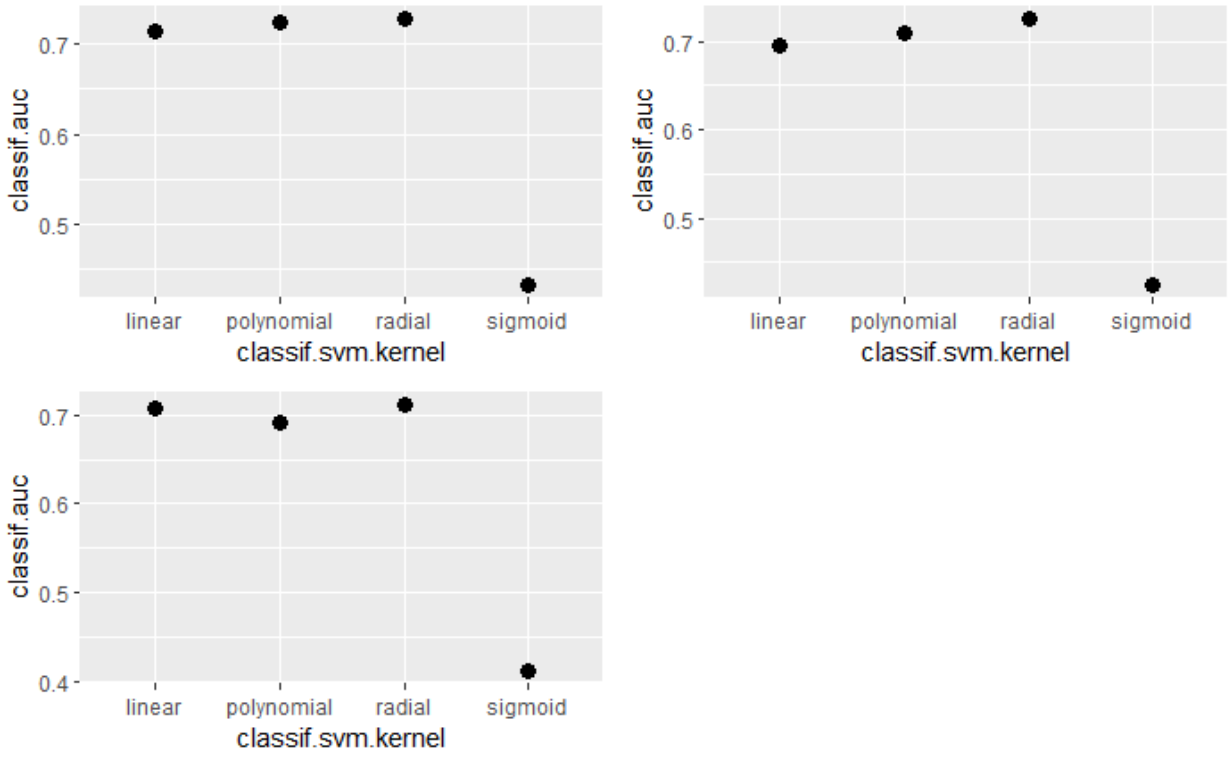
```
# create Autolearner
svm_auto_smote <- AutoTuner$new(
  learner = svm_smote_lrn, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = param_smote,
  terminator = term("none"), tuner = tnr("grid_search", resolution = 5)
)
```



As can be seen from the plots above, **smote.dup_size** is showing better prediction performance when defined as 40 and **smote.K** shall be defined as 21 which is having more stable training performance with color blue correspondingly.

Then the next step would be to determine which **kernel** function type to use with fixed **smote.dup_size** and **smote.K**. Regarding the choice of suitable kernel function with SVM, there are in total four options to choose: **linear**, **polynomial**, **radial**, **sigmoid**. The default value for the mlr3 SVM learner is the **radical** function. Thence, we need to implement the other three kernel functions on the previous training models with the **dl_iv_data** task and again use the **grid_search** to check which **kernel** function will generate better prediction performance shown with higher AUC value.

```
# train with kernel type
kernel_type <- c("linear", "polynomial", "radial", "sigmoid")
param_kernel <- ParamSet$new(params = list(ParamFct$new("classif.svm.kernel", levels = kernel_type)))
po_smote <- po("smote", dup_size = 45, K = 21)
# create smote learner fixed in rf_trees_fixed_lrn
svm_smote_lrn <- GraphLearner$new(po_smote %>>% svm_lrn, predict_type = "prob")
# create Autolearner for step 1
svm_auto_kernel <- AutoTuner$new(
  learner = svm_smote_lrn, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = param_kernel,
  terminator = term("none"), tuner = tnr("grid_search", resolution = 5)
)
```



As you can seen from the plots which refer to the comparison with different **kernel** functions, the best performance is given by the **radial** function for the **dl_iv_data** task. Nevertheless, we still need to figure out the optimal choice with parameter **cost** and **gamma** and as for the tuning range for **cost**, the first try would be to set with 0.1, 1, 10 and 100.

```
svm_lrn <- lrn("classif.svm", predict_type = "prob", kernel = "radial", type = "C-classification")
# train with different set with parameter cost
param_cost <- ParamSet$new(params = list(ParamDbl$new("classif.svm.cost", lower = 0, upper = 3)))
param_cost$trafo = function(x, param_set){
  x$classif.svm.cost = 10**(x$classif.svm.cost)/10
  x
}
# create smote learner fixed un svm learner
svm_smote_lrn <- GraphLearner$new(po_smote %>>% svm_lrn, predict_type = "prob")
# create Autolearner
svm_auto_smote <- AutoTuner$new(
  learner = svm_smote_lrn, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = param_cost,
  terminator = term("none"), tuner = tnr("grid_search", resolution = 3)
)
```

Shown on the plots above, it is better to choose **cost** with value 0.1. Hence, the last step would be to tune **gamma** for the **dl_iv_data** training task. Ranging from xx to xx with **gamma**, we would like to see the trend of the tuning performance and determine for the ultimate **gamma** accordingly.

```
svm_lrn <- lrn("classif.svm", predict_type = "prob", kernel = "radial", cost = 0.1, type = "C-classifica
# train with gamm
param_gamma <- ParamSet$new(params = list(ParamDbl$new("classif.svm.gamma", lower = 0, upper = 3)))
param_cost$trafo = function(x, param_set){
  x$classif.svm.gamma = 10 ** (x$classif.svm.gamma) / 10**3
```

```
  x
}
po_smote <- po("smote", dup_size = 45, K = 21)
# create smote learner
svm_smote_lrn <- GraphLearner$new(po_smote %>>% svm_lrn, predict_type = "prob")
# create Autolearner for step 1
svm_auto_smote <- AutoTuner$new(
  learner = svm_smote_lrn, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = param_gamma,
  terminator = term("none"), tuner = tnr("grid_search", resolution = 4)
)
```

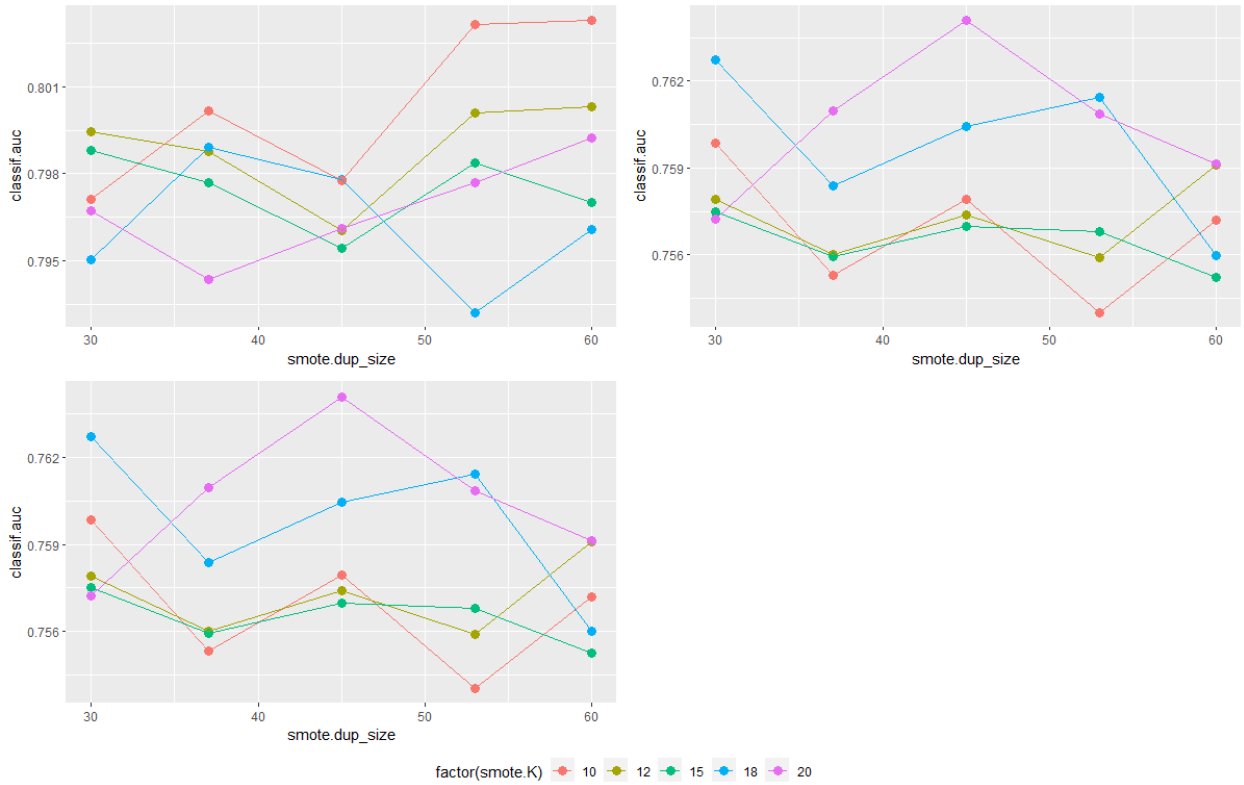
In conclusion, it is better to go with *radial* kernel function, *cost* being set to 0.1 and *gamma* being set to 0.75 on the condition that *smote.dup_size* is set to 40 and *smote.K* to 21. And the ultimate prediction performance measured by AUC is therefore 0.7839 with an increase of 0.0x in regards to the previous svm model with default parameter settings.

7.4 Random Forest (smote)

As mentioned in chapter 3, we fix the parameter **num.trees** with value 1000 and set training range 30 to 60 for **smote.dup_size** and 10 to 20 for **smote.K** seperately in order to find the best training pairs with the two parameters, reaching the best prediction performance.

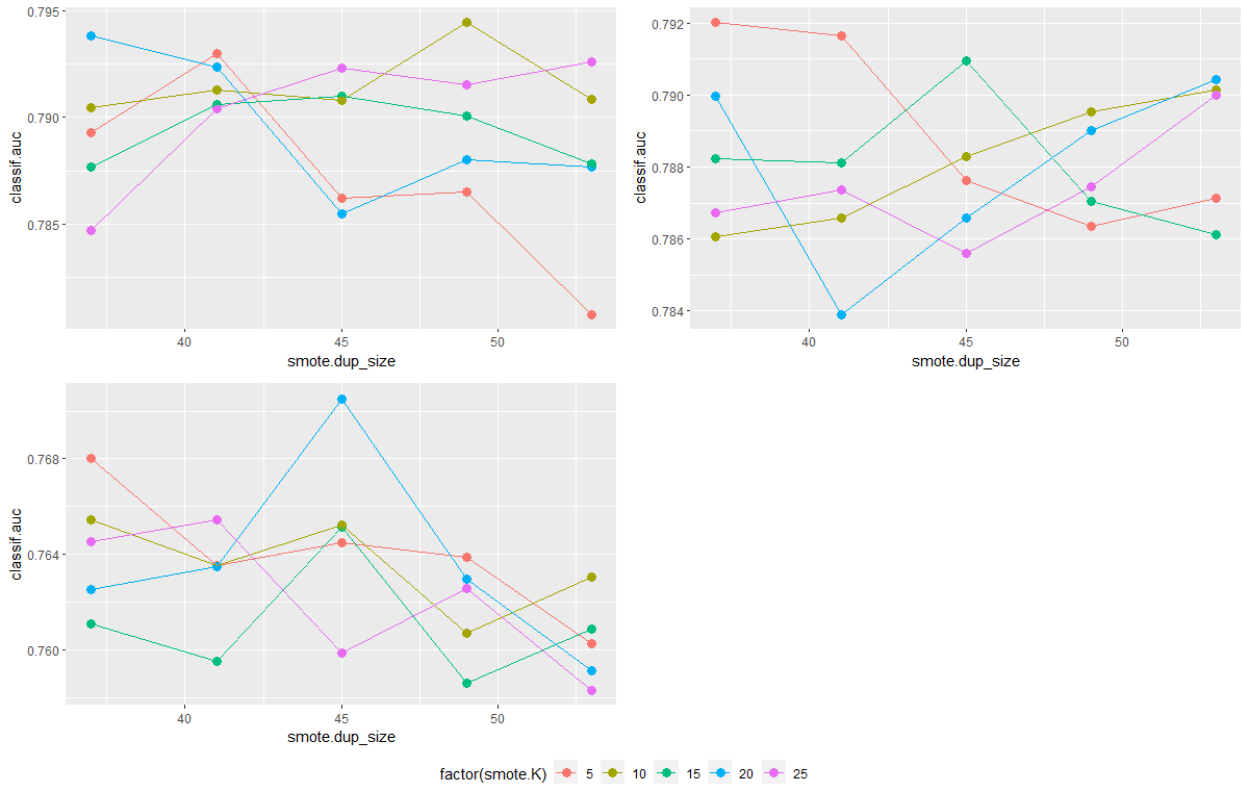
```
library(ranger)
# step 1: test for best matches with smote function's parameter
# learner with fixed number of trees: 1000
rf_lrn <- lrn("classif.ranger", predict_type = "prob", num.trees = 1000)
# train with smote function's 2 parameters
param_smote <- ParamSet$new(params = list(
  ParamInt$new("smote.dup_size", lower = 30, upper = 60),
  ParamInt$new("smote.K", lower = 10, upper = 20)
))
# inner resampling set
inner_rsmp <- rsmp("cv", folds = 3)
po_smote <- po("smote", dup_size = 50)
# create smote learner fixed in rf_trees_fixed_lrn
rf_smote_lrn <- GraphLearner$new(po_smote %>>% rf_lrn, predict_type = "prob")
# create Autolearner for step 1
rf_auto_smote <- AutoTuner$new(
  learner = rf_smote_lrn, resampling = inner_rsmp,
  measures = msr("classif.auc"), tune_ps = param_smote,
  terminator = term("none"), tuner = tnr("grid_search", resolution = 5)
)
# outer resampling set
outer_rsmp <- rsmp("cv", folds = 3L)
# design for smote
design_smote <- benchmark_grid(
  tasks = task,
  learners = rf_auto_smote,
  resampling = outer_rsmp
)
# create benchmark
run_benchmark <- function(design) {
  set.seed(2020)
```

```
  tic()
  bmr <- benchmark(design, store_models = TRUE)
  toc()
  run_benchmark <- bmr
}
# run benchmark
bmr_smote <- ren_benchmark(design_smote, store_models = TRUE)
# plot path
smote_path1 <- bmr_smote$data$learner[[1]]$archive("params")
rf_ggp1 <- ggplot(smote_path1, aes(
  x = smote.dup_size,
  y = classif.auc, col = factor(smote.K)
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")
smote_path2 <- bmr_smote$data$learner[[2]]$archive("params")
rf_ggp2 <- ggplot(smote_path2, aes(
  x = smote.dup_size,
  y = classif.auc, col = factor(smote.K)
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")
smote_path3 <- bmr_smote$data$learner[[2]]$archive("params")
rf_ggp3 <- ggplot(smote_path3, aes(
  x = smote.dup_size,
  y = classif.auc, col = factor(smote.K)
)) +
  geom_point(size = 3) +
  geom_line() #+ theme(legend.position = "none")
ggarrange(rf_ggp1, rf_ggp2, rf_ggp3, common.legend = TRUE, legend = "bottom")
```



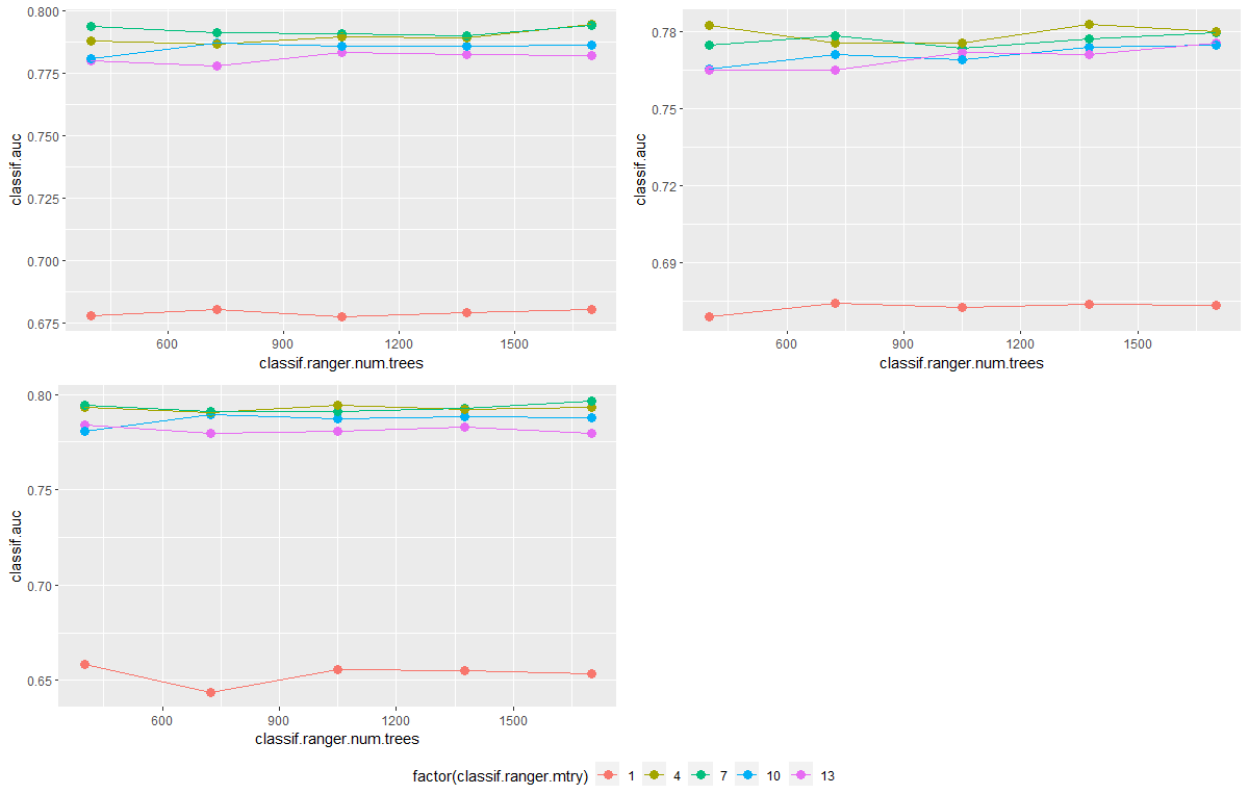
We can see from the plots above, **smote.dup_size** ranging from 37 to 53 together with **smote.k** from 15 to 25 tends to have better AUC values. Then we need to reset the **smote.dup_size** to 37 to 53 and **smote.k** 5 to 25, repeating the **grid_search** again to explore the two parameters further.

```
# learner with fixed number of trees: 1000
rf_lrn <- lrn("classif.ranger", predict_type = "prob", num.trees = 1000)
# train with smote function's 2 parameters
param_smote <- ParamSet$new(params = list(
  ParamInt$new("smote.dup_size", lower = 37, upper = 53),
  ParamInt$new("smote.K", lower = 5, upper = 25)
))
# create smote learner fixed in rf_trees_fixed_lrn
rf_smote_lrn <- GraphLearner$new(po_smote %>>% rf_lrn, predict_type = "prob")
```



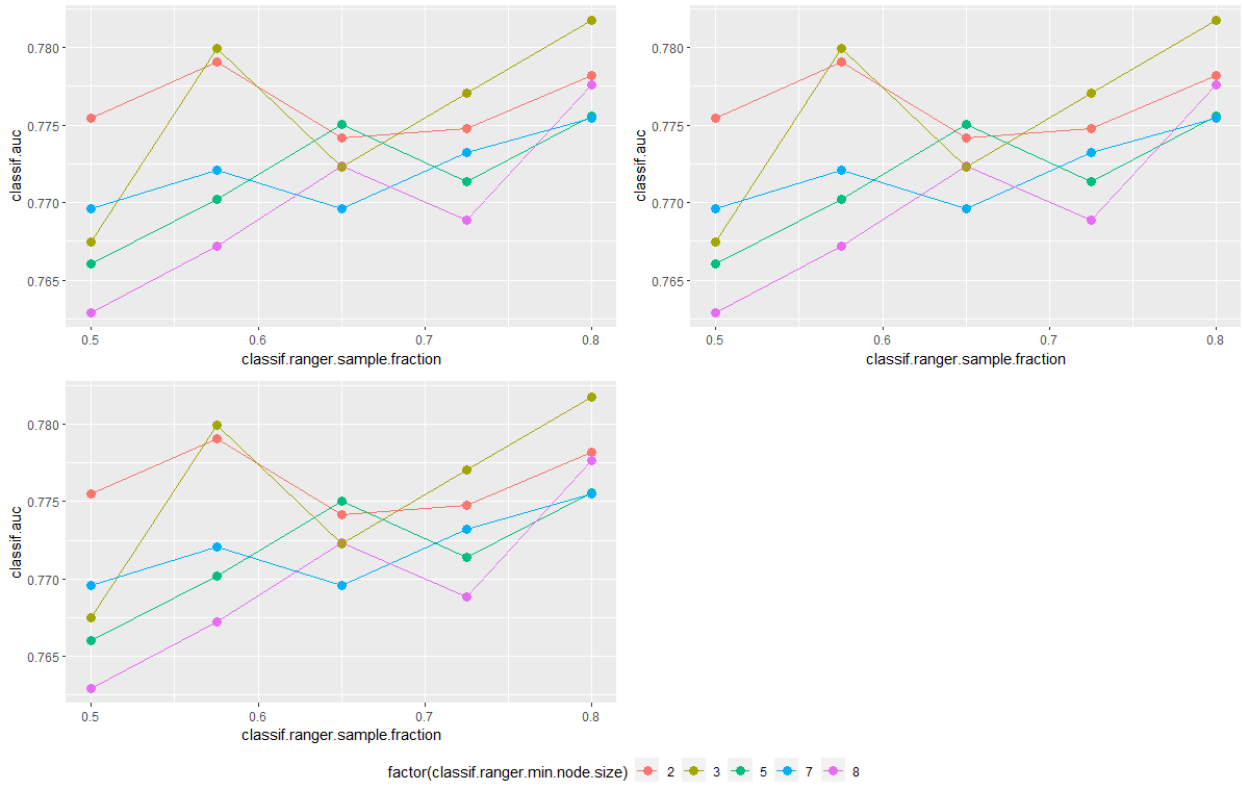
Shown on the plots above, we have discovered that the two parameters **smote.dup_size** and **smote.K** should be set to 45 and 15 seperately where the AUC distributions always tend to have a summit around dup_size value 45 and K with value 15 is much more stable with the prediction performance here at the same time. Then the next step we shall need to stay with the optimal smote parameter settings (**smote.dup_size** is set to 45 and **smote.K** to 15) and take the mtry and the **num.trees** parameters into consideration with parameters' tuning ranges set to 1-13 and 400-1700 in order to get the best pair settings with the highest AUC value.

```
# RF with 4 different paramSets, to analysis how to do further tuning
# tuning for best num.trees and mtry pairs
rf_lrn <- lrn("classif.ranger", predict_type = "prob")
# train with ranger function's 2 parameters
param_nt_mtry <- ParamSet$new(params = list(
  ParamInt$new("classif.ranger.num.trees", lower = 400, upper = 1700),
  ParamInt$new("classif.ranger.mtry", lower = 1, upper = 13)
))
po_smote <- po("smote", dup_size = 45, K = 15)
# create smote learner fixed in rf_lrn
rf_smote_lrn <- GraphLearner$new(po_smote %>>% rf_lrn, predict_type = "prob")
```



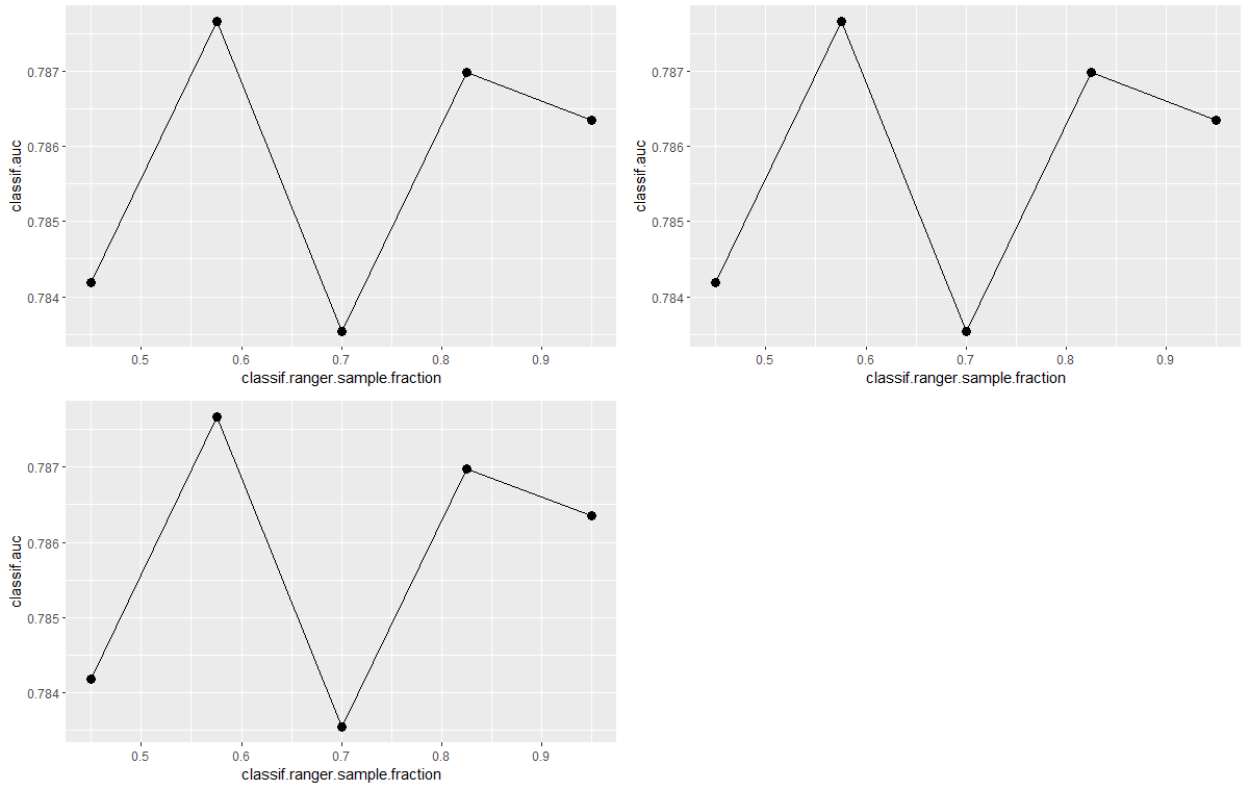
So, we would go with the combination of **num.trees** = 1000 and **mtry** = 7. And the next step would be to tune **min.node.size** and **sample.fraction** with the previous selected parameters settings. According to rangerTune algorithm, we shall try setting tuning range for **min.node.size** from 2 to 8 and **sample.fraction** from 0.5 to 0.8 and see the corresponding AUC performance.

```
# RF with 4 different paramSets, to analysis how to do further tuning
## tuning for best min.node.size and sample.fraction pairs
rf_lrn <- lrn("classif.ranger", predict_type = "prob", num.trees = 1000, mtry = 7)
# train with smote function's 2 parameters
param_mns_sf <- ParamSet$new(params = list(
  ParamInt$new("classif.ranger.min.node.size", lower = 2, upper = 8),
  ParamDbl$new("classif.ranger.sample.fraction", lower = 0.5, upper = 0.8)
))

po_smote <- po("smote", dup_size = 45, K = 15)
# create smote learner fixed in rf_lrn
rf_smote_lrn <- GraphLearner$new(po_smote %>>% rf_lrn, predict_type = "prob")
```



We would go with **min.node.size** = 3 as we can see the AUC value tends to reach the best performance when we set it to 3. However, it is still not clear the approximated defined range for parameter **sample.fraction.** Next, we need to retrain the model with a wider range with parameter **sample.fraction** from 0.45 to 0.95.

```
# RF with 4 different paramSets, to analysis how to do further tuning
## tuning for best sample.fraction
rf_lrn <- lrn("classif.ranger",
  predict_type = "prob", num.trees = 1000,
  mtry = 7, min.node.size = 3
)
# train with smote function's 2 parameters
param_sf <- ParamSet$new(params = list(ParamDbl$new("classif.ranger.sample.fraction",
  lower = 0.45, upper = 0.95
)))
# inner resampling set
inner_rsmp <- rsmp("cv", folds = 3)

po_smote <- po("smote", dup_size = 45, K = 15)

# create smote learner fixed in rf_lrn
rf_smote_lrn <- GraphLearner$new(po_smote %>>% rf_lrn, predict_type = "prob")
```



When reaching the highest AUC value, **sample.fraction** is always taking good performance values around 0.55 and 0.6 and the distributions of **min.node.size** are showing better performance results (measured by AUC) with value 3. Therefore, we would choose the **sample.fraction** with 0.575 and have a better AUC performance value with 0.83.

After the step-by-step hyperparameter tuning, we have finally managed to reach our best prediction performance with AUC value fixed at 0.83 which shows the improvement of 0.07 in comparison with the performance of Random Forest's default model with value 0.76. In conclusion, when **smote.dup_size** being set to 45 and **smote.K** to 15 with the SMOTE operator, the final parameter settings combination would be **num.trees** = 1000, **mtry** = 7, **min.node.size** = 3 and **sample.fraction** = 0.575, reaching an AUC level at 0.83.

8 Conclusion and Discussion

8.1 Conclusion

In general, we have implemented four different training models (Logistic Regression, KNN, SVM and Random Forest) on the same training task (dl_iv_data). Besides the basic training process with default parameter settings with different models, we have also discovered the necessary methods (SMOTE algorithm and Oversampling Operator) to tackle with the imbalanced target data distribution and tune the models with corresponding parameters afterwards. By plotting the changing trend of different parameters separately, we shall be able to determine the best parameter setting combinations and achieve the highest AUC score with the ultimate chosen model. After determining for the related parameters which are used for data balancing and finishing the model tuning procedures with parameters accordingly, we have run into the conclusion that Random Forest has shown a considerably better prediction performance, with an average of 10% improvement among all other training models, in regard to the decision on whether or not to issue the credit card to the potential customer. As for our mission to predict the ultimate decision concerning credit card approval, we would definitely choose Random Forest algorithm and implement the prediction afterwards with various customers.

8.2 Discussion

Based on our previous work, there are still many aspects can be improved in our article. First of all, more powerful models that we can use to solve some problems that we met. For example, it's computationally expensive to train the SVM model, a 3*3 nested resampling of SVM model takes 5 hours. Secondly, we have tried xgboost and we found it extremly fast to excute but with the worst prediction performance of all (AUC=0.5 and not improved after any kinds of model tuning). On the other hand, there are still a lot of cutting-edge models we haven't used, e.g. lightGBM(not included in mlr3 yet), Deep learning method(require high computational power). Last but not least, there are still areas to explore in our feature engineering procedure. For example, dimension reduction methods such as PCA, MDS and Factor analysis (included in mlr3 filters) can also be used.

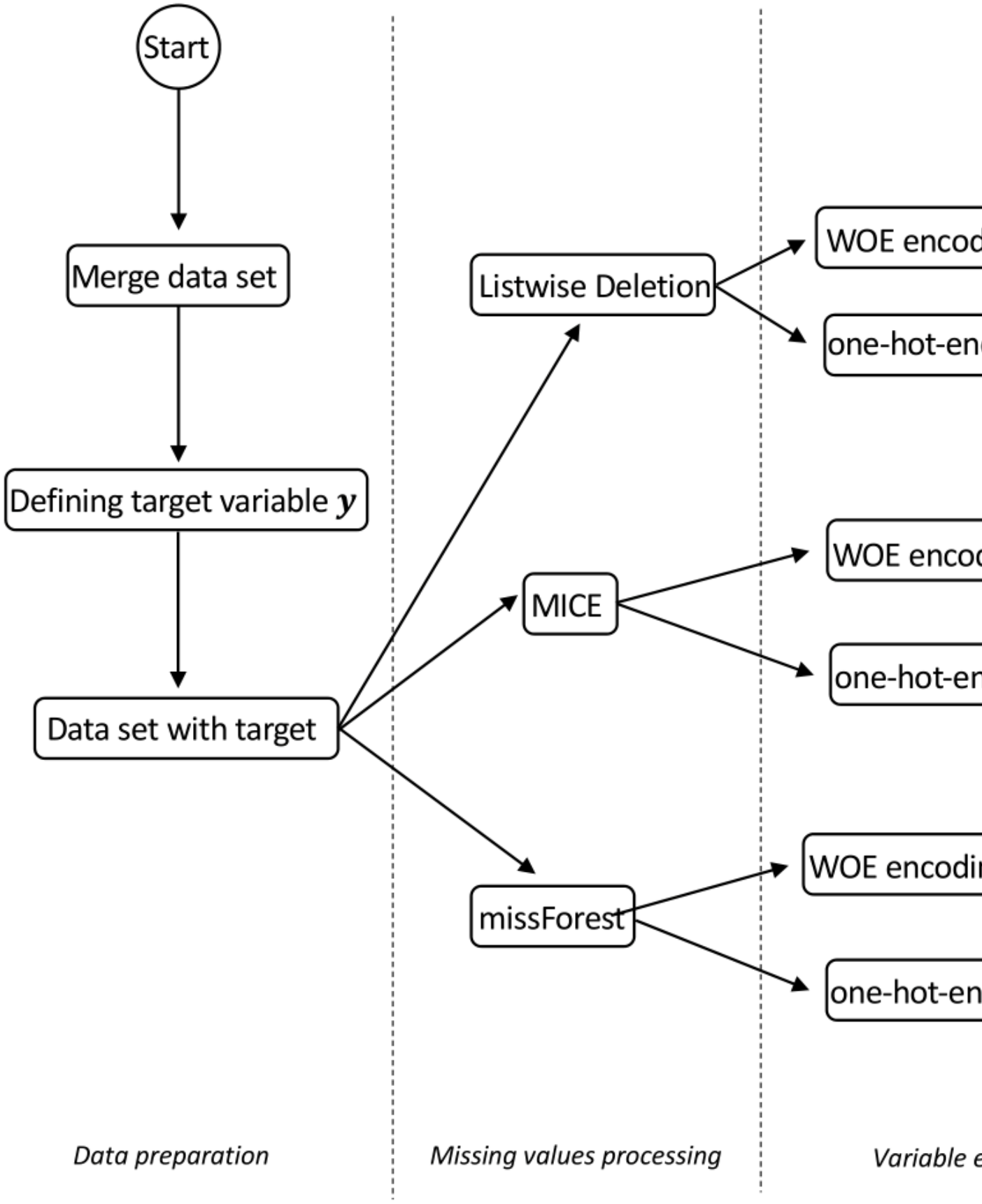


Figure 1: Flowchart: process of experiment