Assignment: Predictive analytics of product return

S.M. Safiul Alom, alomsafi@student.hu-berlin.de

28/02/2019

1 Introduction

In the recent years the importance of data science in online retailing has increased vastly. One of the main reason for that is the huge data collection of such retailer. These data are also collected in a systematic way. that's why the application of machine learning methods has been lot easier than before. Now-a-days there are some open source software packages, such as R and Phthon, which are very useful to analysis big data, e.g. big data in e-commerce.

Now-a-days product return is one of the most common word in online retailing. Since return shipping is not free of cost, it has some impact on profitability of online retailers. For certain companies and products return costs are huge. Product can be returned for various reasons, such as impluse purchase or fraudulent behaviour of customers. Since free or inexpensive return shipping is a standard expectation of customers, online retailers are often bound to take the shipping cost of products return to stay competitive in the market. But they can influence customer purchase behaviorby developing some possible intervention strategies, if the probability of products return are too high, such as restricting payment options, reducing the time of products return or rejecting the transaction(in extrem cases).

To take the intervetions mentioned above, one need to know the likelihood of a product being returned. The objective of this homework assignment is to develop some models using machine learning methodes, which can predict the likelihood of products return accurately. For this special task, we are provided with real-world data by an online retailer. The whole assignment has been done in R and consists mainly of six parts:

- **Exploratory data analysis:** In this part of the assignment the main characteristics of the provided data will be summarized by e.g. some measures of location, dispersion measures and visualizations. For modeling, it is sometimes helpful to know the distribution of target variable.
- **Data preparation:** Data cleaning is one of the most important part in data science. To get a good predictive model a properly cleaned dataset is unavoidable. Data preperation includes, e.g. removing unwanted observations, filteing unwanted outliers, handling missing data and so on.
- **Model tuning and selection:** In this part, models will be developed, tuned by different values of hyperparameters and the the best model will be selected by estimating of its performance.
- **Model evaluation:** After developing a model it is necessary to compute its prediction accuracy with some new data, which is called model evaluation.
- **Special modeling challenge:** In this part of the assignment we will examine whether we can improve the standard logit model by picking the coefficients to optimize the profit directly.
- **Conclusion:** At the end, we will summarize the whole assignment and discuss about some limitations of the resulting predictive models.

2 load packages and data

In the first step, the requiered package, training and test dataset are uploded. after that both datasets are combined to get a single dataset.

```r
#function for installing required packages
install.load.packages = function(pack, packagePath){
  all.packages = pack[!(pack %in% installed.packages(lib.loc = packagePath)[, 1])]
  if (length(all.packages))
    install.packages(all.packages, lib = packagePath, dependencies = TRUE)
  sapply(pack, require, lib.loc = packagePath, character.only = TRUE)
}


libraryPath = "C:\\Users\\Himel\\OneDrive\\Studium\\R\\Packages"
packages =  c('caret', 'rpart', 'ggplot2', 'mlr',
              'xgboost', 'parallelMap', 'randomForest', 'dplyr', 'klaR',
              'lubridate','knitr', 'GA', 'papeR', 'knitcitations')
#install and load packages
install.load.packages(packages, libraryPath)

#set working-directory
setwd('C:\\Users\\Himel\\OneDrive\\Studium\\M.Sc. Statistics
      \\3_Business Analytics&Data Science\\Assignment\\Data')

#load known and unknown set
Prod.Returns.train = read.csv('BADS_WS1819_known.csv',
                              sep = ',', na.strings = "", stringsAsFactors = FALSE)
Prod.Returns.test = read.csv('BADS_WS1819_unknown.csv',
                             sep = ',', na.strings = "", stringsAsFactors = FALSE)
#combine test and training set
Prod.Returns.train = Prod.Returns.train %>% dplyr::mutate(dataset = "train")
Prod.Returns.test = Prod.Returns.test %>% dplyr::mutate(dataset = "test")
full.Prod.Returns = dplyr::bind_rows(Prod.Returns.train, Prod.Returns.test)
```

3 Exploratory data analysis

3.1 Characteristics of data

We are given two sets of data, namely Known and unknown Dataset. Unknown means that the target variable (here product return) is not known. Known dataset is used to train the model, i.e. known dataset is our training set and unknown dataset is used to predict product returns. The training set consists of 100000 observations and 14 variables including target variable. Most of the features have data type character and some of them have data type numeric. The target variable is categorial and takes on values 0 and 1(1 for return and 0 for keep)

```r
#str(Prod.Returns.train)
library(magrittr)
data.frame(variable = names(Prod.Returns.train),
           classe = sapply(Prod.Returns.train, typeof),
           first_values = sapply(Prod.Returns.train,
                                 function(x) paste0(head(x),  collapse = ", ")),
           row.names = NULL) %>%
  kable('latex')
```

Table 1: Characteristics of features and target variable

variable	classe	first_values
order_item_id	character	ID1, ID2, ID3, ID4, ID5, ID6
order_date	character	2016-05-13, 2016-09-26, 2017-02-28, 2017-01-23, 2016-05-13, 2016-06-27
delivery_date	character	2016-05-16, NA, 2017-05-09, 2017-01-24, 2016-05-16, 2016-06-29
item_id	integer	1040, 4490, 4792, 4686, 648, 826
item_size	character	38, m, m, L, 39, 46
item_color	character	red, grey, ecru, terracotta, petrol, purple
brand_id	integer	138, 133, 128, 105, 170, 111
item_price	double	69.9, 59.9, 99.9, 59.9, 139.9, 44.9
user_id	integer	19065, 32349, 74743, 32935, 2842, 10005
user_title	character	Mrs, Mrs, Mrs, Mrs, Mrs, Mrs
user_dob	character	NA, 1967-02-21, NA, 1961-01-08, 1966-05-01, 1958-03-05
user_state	character	Lower Saxony, North Rhine-Westphalia, North Rhine-Westphalia
user_reg_date	character	2016-05-14, 2015-02-17, 2017-02-14, 2015-02-17, 2015-04-06, 2015-02-17
return	integer	1, 0, 0, 0, 1, 1

3.2 Summary

The following table summarizes the numeric variables in the training set. It shows that, mean of the target variable(return) is 0.48, i.e. 48% of the Products has been returned by customers. It follows that there is no class imbalance in the training set.

```r
knitr::kable(papeR::summarize(Prod.Returns.train, type = 'numeric'),
             caption = "Summary of the training dataset")
```

```
## Factors are dropped from the summary
```

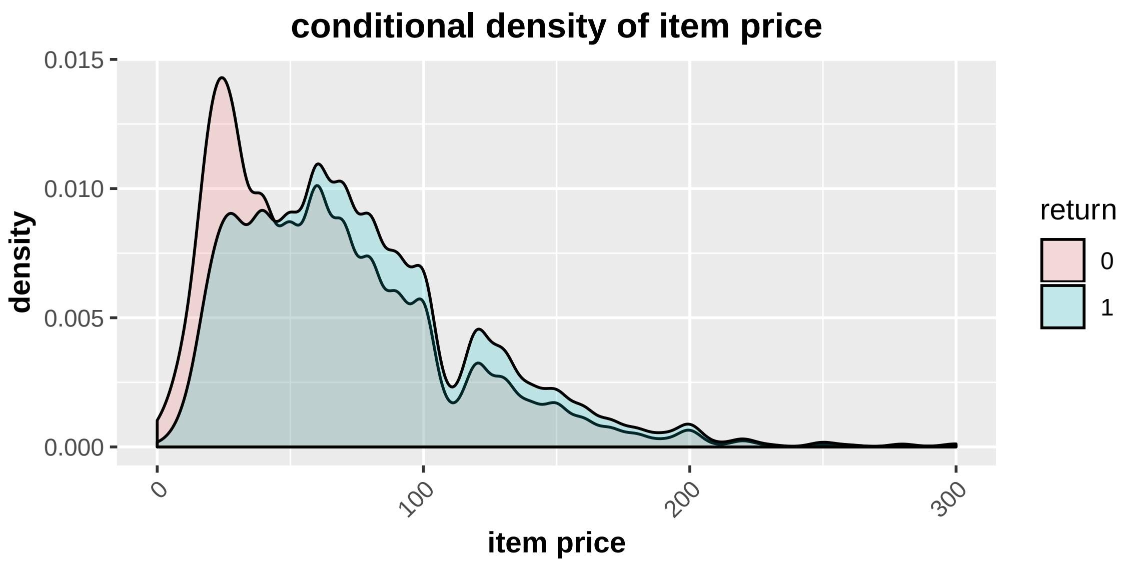
Table 2: Summary of the training dataset

	N	Mean	SD	Min	Q1	Median	Q3	Max
item_id	1e+05	2799.94	1862.59	4	666.0	3140.0	4686.0	6144.00
brand_id	1e+05	129.87	30.15	101	105.0	122.0	143.0	266.00
item_price	1e+05	70.28	44.97	0	34.9	59.9	89.9	399.95
user_id	1e+05	33423.09	23962.10	6	12944.0	28472.0	51219.0	86609.00
return	1e+05	0.48	0.50	0	0.0	0.0	1.0	1.00

3.3 Graphical visualization

Product returns are often positively correlated to the price of the product. As we can see from the following graphic, the distribution of the product-price differs by return and non return. Non-returned products with lower price have higher probability than returned products with lower price and vice versa, i.e. product price and product return aren't independent.

```r
p2=ggplot(Prod.Returns.train)+
  geom_density(aes(x = item_price, fill = as.factor(return)),alpha=0.2)+
  theme(axis.text.x = element_text(angle = 45, hjust = 1),
        axis.title.x = element_text( face="bold"),
        axis.title.y = element_text( face="bold"))+
  xlab('item price')+
  ylab('density')+
  ggtitle('conditional density of item price')+ xlim(c(0, 300))+
  theme(plot.title = element_text(hjust = 0.5, color = "black", face="bold"))
p2 + guides(fill=guide_legend(title="return"))
```



4 Data preparation

In this part of the assignment we will transform and modify the training set so that our desired machine learning methodes can be applied to it. First we will transform all the character variables(except the date variables and order_item_id) into factor variables. To do that, first we will reduce the levels of the variables in a systematic way and after that the factor transformations will be done.

```r
#reducing factor levels of item_size
#sort(table(full.Prod.Returns$item_size))
name.item_size = full.Prod.Returns%>% dplyr::select(item_size)%>%table()%>%
  sort()%>%names()
name.item_size = name.item_size[(length(name.item_size)-15): length(name.item_size)]
full.Prod.Returns$item_size = ifelse(!(full.Prod.Returns$item_size %in% name.item_size),
                                     'other', full.Prod.Returns$item_size)
#reducing factor levels of item_color
#sort(table(full.Prod.Returns$item_color))
name.item_color = full.Prod.Returns%>% dplyr::select(item_color)%>%table()%>%
  sort()%>%names()
name.item_color = name.item_color[(length(name.item_color)-14): length(name.item_color)]
full.Prod.Returns$item_color = ifelse(!(full.Prod.Returns$item_color %in% name.item_color),
                                      'other', full.Prod.Returns$item_color)
#reducing factor levels of user_title
full.Prod.Returns$user_title = ifelse(!(full.Prod.Returns$user_title %in%
                    c('Mr', 'Mrs')), 'other', full.Prod.Returns$user_title)
#reducing factor levels of user_state
#sort(table(full.Prod.Returns$user_state))
name.user_state = full.Prod.Returns%>% dplyr::select(user_state)%>%table()%>%
  sort()%>%names()
name.user_state = name.user_state[(length(name.user_state)-4): length(name.user_state)]
full.Prod.Returns$user_state = ifelse(!(full.Prod.Returns$user_state %in% name.user_state),
                                      'other', full.Prod.Returns$user_state)
#character variable are transformed to factor
```

```r
factor_vars = c("item_size", "item_color", "user_title", "user_state")
full.Prod.Returns[factor_vars] <- lapply(full.Prod.Returns[factor_vars],
                                         function(x) as.factor(x))
```

After transforming the character variables, we need to check our combined dataset to find out the columns with NA values. delivery_date and user_dob both columns contain NA values. Since some machine learning algorithms can not deal with NA values, we replace those values by some dummy values. In the next step we convert order_item_id and date variables into integer by a one-to-one transformation.

```r
#column with NA's
#full.Prod.Returns %>% lapply(is.na) %>%sapply(any)
#delivery_date and user_dob contain NA's
#filling all Rows of NA's with dummy value
#missing user date of birth are replaced by mode
full.Prod.Returns$user_dob[is.na(full.Prod.Returns$user_dob)] =
  names(table(full.Prod.Returns$user_dob))[table(full.Prod.Returns$user_dob)
                                           == max(table(full.Prod.Returns$user_dob))]
#missing delivery_date are replaced by a dummy date
full.Prod.Returns$delivery_date[is.na(full.Prod.Returns$delivery_date)] = "2099-31-12"
#transform character order_item_id to integer order_item_id
ID = substr(full.Prod.Returns$order_item_id, 3,length(full.Prod.Returns))
full.Prod.Returns$order_item_id = as.integer(ID)
#str(full.Prod.Returns)
#character date variable are transformed to data type "Date"
date_vars = c("order_date", "delivery_date", "user_dob", "user_reg_date")
full.Prod.Returns[date_vars] <- lapply(full.Prod.Returns[date_vars],
                                       function(x) as.Date(x))
#create dummy date
date_vars = unlist(lapply(full.Prod.Returns,  lubridate::is.Date))
date_vars.dummy = paste(names(date_vars[date_vars]), ".dummy", sep = "")
full.Prod.Returns[date_vars.dummy] = lapply(full.Prod.Returns[date_vars],
                                    function(x) as.integer(as.Date("2019-01-13") - x))
full.Prod.Returns[names(date_vars[date_vars])] = NULL
```

Some features are not relevant, i.e. they don't have any influence or very weak influence on target variable. Such features should be removed to avoid overfitting and complexity of model. In our case order_item_id cannot be a relevant feature since the rows of this feature are unique. Item_color and user_state are also considered as irrelevant features because they do not improve the prediction accuracy and the model overfits. Since some methodes can not deal with factor variables, they have to be transformed into dummy variables.

```r
#return as factor
full.Prod.Returns$return = factor(full.Prod.Returns$return,
                              levels = c(0,1), labels = c('NO', 'YES'))
#item_color and user_state aren't relevant features
full.Prod.Returns$item_color = NULL
full.Prod.Returns$user_state = NULL
#str(full.Prod.Returns)
#dummy feature
full.Prod.Returns.dummy = mlr::createDummyFeatures(full.Prod.Returns, target = 'return')
```

```
#chack data
#str(full.Prod.Returns.dummy)
#devide data into original known- and unknown set
known = full.Prod.Returns.dummy %>% dplyr::filter(dataset == "train") %>%
  dplyr::select(-dataset)
unknown <- full.Prod.Returns.dummy %>% dplyr::filter(dataset == "test") %>%
  dplyr::select(-c(dataset, return))
#order_item_id is also not a relevant feature
known$order_item_id =NULL
unknown$order_item_id = NULL
```

5 Building the model

5.1 Methodical foundation

To develop product return models, we use a homogenous ensamble approach, namely gardient boosting. Like other ensemble methods boosting aggregates many models to get a better prediction and can be applied to many statistical learning methodes, e.g. decision tree. A very special characteristics of boosting is that the models do not depend on boostrap resampling. In this approach every sub-model are developed successively and can be improved using information from previously developed model. This kind of approach tells us that boosting sub-models are not independent(see details at Gareth James, Hastie, and Tibshirani 2013, 321–24).

5.2 parameter tuning and model selection

Gardient boosting is implemented in R-Package **Xgboost**. We also use **mlr** package in R to simplify the implementation of **Xgboost**. In the first step, we need to create a classification task using the function **makeClassifTask**, which takes training set, target variable and positive value as arguments. A Task can also be interpreted as a training set.

After that we define a xgboost learner for classification with the type of prediction(here probability). This learner function takes also some other parameter values, e.g. early stopping when no improvement for k iterations.

```
#make task
task = mlr::makeClassifTask(data = known, target = 'return', positive  = 'YES')
#make learner
xgb.makeLearner <- mlr::makeLearner("classif.xgboost", predict.type = "prob",
                                    par.vals = list("verbose" = 0,
                                                    "early_stopping_rounds"=20))
```

The performance of a model can be improved by tuning the parameters of this model. We need additionally a performance measure(here AUC) to compare models with different parameters. AUC is a mesure which calculates area under the ROC curve and lies between 0 and 1. The larger the value of AUC is, the better the model is.

In the mlr-package implemented function **makeParamSet** can be used for tuning a wide range of hyper-paraters of Xgboost model**(see discription of the parameters at *Package Mlr* 2018)**. We input different types of hyper-parameters in this function, namely numeric, integer and discrete parameters. For numeric and integer parameters we need to give an interval and for discrete parameter some values. Obviously, we cannot try every combination of parameter values. That's why we take some random combinations of parameter values from the given sets of parameters(here maxit = 100). To compare models with different parameters, we use AUC and to avoid overfitting we use 2-fold cross-validation.

After defining all necessary parameters, the **mlr** function **tuneParams** can be run for model tuning. This function computes **AUC** for the model with random combinations of parameters and chooses hyper-paramters with largest **AUC**. Since the function **tuneparams** takes too much time to tune the parameters, we save the best hyperparameters, which was found by using parameter tuning, as a **.rds** file so that we do not need to run the function **tuneparams** multiple times, when we need those parameters.

```r
# Set tuning parameters
xgb.makeParamSet <- makeParamSet(
  makeNumericParam("eta", lower = 0.01, upper = 0.5),
  makeIntegerParam("nrounds", lower=80, upper=500),
  makeIntegerParam("max_depth", lower=2, upper=8),
  makeDiscreteParam("gamma", values = 0),
  makeDiscreteParam("colsample_bytree", values = 1),
  makeDiscreteParam("min_child_weight", values = 1),
  makeDiscreteParam("subsample", values = 1),
  makeNumericParam("lambda", lower = -1, upper = 0, trafo = function(x) 10^x)
)
# random parameter-combination
xgb.makeTuneControlRandom <- mlr::makeTuneControlRandom(maxit=100,
                                                        tune.threshold = FALSE)
# 2-fold cross-validation
xgb.makeResampleDesc <- makeResampleDesc(method = "RepCV", rep = 3,
                                         folds=2, stratify = TRUE)
# set.seed(123) # Set seed for the local random number generator
# # Tune parameters
# xgb.tuneParams <- mlr::tuneParams(xgb.makeLearner, task = task,
#             resampling = xgb.makeResampleDesc,par.set = xgb.makeParamSet,
#             control = xgb.makeTuneControlRandom, measures = mlr::auc)

#save the optimal hyperparameters to run the model several times
#saveRDS(xgb.tuneParams, file = "C:\\Users\\Himel\\OneDrive\\
#Studium\\M.Sc. Statistics\\3_Business #Analytics&Data Science
#\\Assignment\\Objects\\xgb.tuneParams.rds")
```

Now we describe those hyperparameter, which are used for parameter tuning.

- **eta:** a conservative factor to prevent overfitting of a boosting model. After every boosting step, we get a new featue weights and eta shrinks these weights. smaller value of eta helps to prevent overfitting but it makes the computation slower. eta can take a value between 0 and 1. Parameter tuning provides eta = 0.05644954.
- **gamma:** a non negative real value giving the minimum loss reduction, which is needed to make a further partition on a leaf node of the tree. a smaller value of gamma means the algorithm is more conservative. We choose here gamma = 0.
- **nrounds:** number of boosting round in the current training. Parameter tuning provides nround = 367.
- **max_depth:** a integer value giving the maximum depth of a tree. A larger value of max_depth makes the model more complicated(here Tuned max_depth = 7)
- **colsample_bytree:** A real value giving the subsample ration of columns by constructing each tree.

colsample_bytree has a range of (0,1].

- **min_child_weight:** minimum sum of instance weight(hessian) needed in a child(here min_child_weight = 1)
- **subsample:** subsample ratio of training set, e.g. subsample is equal to 0.6 means 60% of the training data are randomly selected to grow a tree. We use the whole training dataset to growing tree, i.e. subsample = 1.
- **lambda:** L2 regularization parameter. Parameter tuning process provides lambda = 0.108.

```r
xgb.tuneParams = readRDS(file="C:\\Users\\Himel\\OneDrive\\Studium\\M.Sc. Statistics
        \\3_Business Analytics&Data Science\\Assignment\\Objects\\xgb.tuneParams.rds")
kable(as.data.frame(xgb.tuneParams$x), format = 'latex',
      booktabs = T, digits = 4,
      caption = 'Identified hyperparameter after parameter tuning')
```

Table 3: Identified hyperparameter after parameter tuning

eta	nrounds	max_depth	gamma	colsample_bytree	min_child_weight	subsample	lambda
0.0564	367	7	0	1	1	1	0.1077

5.3 Prediction of product return

After identifying the optimal hyperparameters, we need to update the learner to these hyperparameters. By using this learner and task defined above, we will train our optimal **xgboost model**. One of the most important part of Modeling is to assess the created model. There are several types of Measures, which can be used for the model assessment. Here we are given **AUC** measure to evaluate the performance of identified model.The larger the **AUC**, the better a model is. For prediction of product return, we use the new dataset **unknown.csv**. To compute the model-AUC using the unknown data, we need to upload order_item_id and prediction of the unknown dataset to online kaggle compition page. It provides an AUC-score of 0.68403 for our identified **xgboost model**. Based on this result, one can interpret the identified model as a good model. Further improvement of **AUC** is also possible by doing more refined imputation, feature creation, better tuning, etc.

```r
# Update the learner to the optimal hyperparameters
xgb.makeLearner <- setHyperPars(xgb.makeLearner,
                      par.vals = c(xgb.tuneParams$x, "verbose" = 0))
model.xgb = mlr::train(xgb.makeLearner, task = task)
yhat <- predict(model.xgb, newdata = unknown)
#create a prediction file
prediction = Prod.Returns.test %>% dplyr::select("order_item_id" = order_item_id) %>%
  dplyr::mutate("return" = yhat$data$prob.YES)
# write.csv(prediction, file =
#             "C:\\Users\\Himel\\OneDrive\\Studium\\M.Sc. Statistics\\
#3_Business Analytics&Data Science\\Assignment\\result3\\prediction4.csv",
#row.names = FALSE)
kable(head(prediction, n = 6),
    caption = 'Prediction of product return with the identified model')
```

Table 4: Prediction of product return with the identified model

order_item_id	return
ID100001	0.4820875
ID100002	0.5183406
ID100003	0.3212535
ID100004	0.4735467
ID100005	0.3826177
ID100006	0.4843530

As we have already noticed, choice of parameters have a great influence on model performence. Some models are bad not just because of noisy data or weak learner, but also due to selection of bad parameter values. So far, we selected parameters by using random parameter tuning, which is always not a optimal way. In the next section we will see, how we can choose parameters in a clever way

6 Parameter optimization by using Genetic Alogrithm

Genetic Algorithm(GA) is a evolutionary algorithm, which tries to find the best input values(here: parameter) to optimize a given fitness function. To work with GA, first we need to define a initial population consisting of some solutions. Each solution is called individual and each individual is defined by a set of parameters. We also need a fitness function. Each individual has a fitness value(fitness function evaluated at the individual). The larger the fitness value, the higher the quality of the individual(solution). We will select individuals with high quality to produce their off-springs. Finally, the individuals with lower quality are replaced by these off-springs. This process repeats until some stop criteria are fulfilled(see details at *Introduction to Optimization with Genetic Algorithm* 2018, and @GA1).

Now we will apply this idea to our given product return dataset. By predicting a binary target variable, we can make two types of errors, e.g. classifying a customer as returner who is actually non returner and classifying a customer as non returner who is actually returner. In practice, these errors do not necessarily have the same consequences and it follows that cost of errors are not symmetric, e.g. in credit risk, the cost of the misclassification of a bad customer is much higher than the cost of the misclassification of a good customer. Asymmetric cost behavior is also a typical case in product return modeling. For this reason, we need to assess the model prediction in terms of a given cost matrix.

The objective of this section is to develop another product return model by using logit methods. The coefficients of the logit model are selected by using GA, which optimizes a given fitness function. This function is build by using the information of a given cost matrix.

Since we can not estimate the coefficients, if the features are highly correlated. That's why we need to remove those features which are correlated with another features. For example, Variable **item_id** and **order_date.dummy** are highly correlated. So one of these two variables has to be removed. Another important reason for correlation of the features is dummy variable trap, i.g. if we make for every level of a feature a dummy variable, then those dummy variables are linearly dependent with the intercept variable. In our case, to solve this problem, we remove one dummy column of item_size and one dummy column of user_title. In the next step, we devide our known dataset into training and test set. After that, we standardize all features to make the coefficients comparable across features.

```r
#----------------------Logit + GA-----------------------------------#
#item_size.XXL, user_title.Mr 1-0 dummy => one column of every dummy coding are removed
known.clean = known %>% dplyr::select(-c(item_size.XXL, user_title.Mr ))
known.clean.x = known.clean%>%dplyr::select(-return)
known.clean.x = melt(cor(known.clean.x))
known.clean.x %>% dplyr::filter((value > 0.7 | value < - 0.7) & X1 != X2 )
```

```
##                  X1                X2      value
## 1 order_date.dummy           item_id -0.8353374
## 2           item_id order_date.dummy -0.8353374
```

```r
#=> item_id and order_date.dummy are highly correlated, one of them has to be removed
known.clean = known.clean %>% dplyr::select(-c(item_id))

#create test and training set
set.seed(12345)
idx.train <- caret::createDataPartition(y = known.clean$return, p = 0.6, list = FALSE)
tr <- known.clean[idx.train, ]
ts <-  known.clean[-idx.train, ]

standard.model <- caret::preProcess(tr, method = c("center", "scale"))
tr <- predict(standard.model, newdata = tr)
ts <- predict(standard.model, newdata = ts)
```

Our benchmark model is standard logit model and our special task is to improve the standard logit model by picking the coefficients to optimize the profit directly. To run the logit model in R, one can use the function **glm()**, which is implemented in R-package **stats**. As argument, we have to give the target variable, features and the link function(here: "logit")

```r
#standard logit model
logit <- glm(return~., data = tr, family = binomial(link = "logit"))
```

To run the GA, first we need to define some auxiliary functions.

6.1 Estimating positive probability

```r
predict.Prob(x, beta)
```

Description:

Function *predict.Prob* estimetes positive probability $P(Y = 1|X = x)$ for a given matrix x and coefficient vector β.

Arguments:

- **X:** A design matrix or a matrix object
- **beta:** A numeric vector object, possibly a coefficient vector

Implementation

```r
predict.Prob <- function(x, beta){
  ## Calculate logit prediction
  yhat <- x %*% beta
  # Logit transformation
  prob <- exp(yhat) / (1 + exp(yhat))
  return(prob)
}
```

6.2 Calculating decision profit

```
decision.profit(y, x, beta, itemvalue)
```

Description: Function *decision.profit* computes decision profit for given vector y, matrix X, coefficient vector beta and a numeric vector itemvalue.

Arguments

- **y:** a numeric vector or a numeric target variable
- **X:** a design matrix or a matrix object
- **beta:** A numeric vector object, possibly a coefficient vector
- **itemvalue:** a numeric vector

Implementation

6.3 Computing fitness value

Description: Function **Fitness.Func** computes fitness value for given coefficient vector beta, model matrix X, numeric target variable y, a numeric vector itemvalue and a fitness function metric.

```
Fitness.Func(beta, x, y, metric, itemvalue)
```

Arguments

- **y:** a numeric vector or a numeric target variable
- **X:** a design matrix or a matrix object
- **beta:** A numeric vector object, possibly a coefficient vector
- **itemvalue:** a numeric vector
- **metric:** a fitness function

Implementation

```
Fitness.Func <- function(beta, x, y, metric, itemvalue){
  #prob <- predict.Prob(x, beta)
  fitnessScore <- do.call('metric', list(y = y, x, beta = beta, itemvalue = itemvalue ))
  return(fitnessScore)
}
```

6.4 GA

After implementing the fitness function, we also need to define the model matrix and target variable(here: return). To perform the GA, one can use the function **ga()**, which is implemented in R-package **GA**. Besides design matrix, target variable and fitness function, we have to give some other hyperparamter in this this function, e.g. lower and upper bound of the coefficients, population size(here: 70), the maximum number of iterations to run before the GA search is halted(here: 500) and so on. Since the fitness function is a profit function, **ga()** maximizes it to choose the model coefficients.

```
#model matrix
train.x <- model.matrix(return~., tr)
test.x <- model.matrix(return~., ts)
train.y.numeric = (as.numeric(tr$return)-1)
test.y.numeric = (as.numeric(ts$return)-1)
lower.bound = min(as.numeric(logit$coefficients)) - 10
upper.bound = max(as.numeric(logit$coefficients)) + 10
#GA Algorithm
```

```r
ga.logit =  ga(type = "real-valued",
                fitness = Fitness.Func,
                x = train.x, y = train.y.numeric , metric = decision.profit,
                itemvalue = tr$item_price,
                lower = rep(lower.bound, ncol(train.x)),
                upper = rep(upper.bound, ncol(train.x)),
                popSize = 70, pcrossover = 0.8, pmutation = 0.01, elitism = 0.01,
                maxiter = 500, run = 100,
                parallel = FALSE,
                monitor = FALSE
)
```

The following table shows the estimates of first 6 coefficients, where the second column is the coeficients estimation of standard logit model and the third one is the coeficients estimation of GA.

```r
coef.ga.logit = ga.logit@solution[1,]
names(coef.ga.logit) = colnames(train.x)
coef.logit = logit$coefficients

coef.mat = data.frame(Coefficient = names(coef.ga.logit),
           Estimete_GA = as.numeric(coef.ga.logit),
           estimete_standard = as.numeric(logit$coefficients))
kable(head(coef.mat, n = 6), caption = "Table of Estimates of first 6 coefficients")
```

Table 5: Table of Estimates of first 6 coefficients

Coefficient	Estimete_GA	estimete_standard
(Intercept)	8.1498491	-0.0745201
brand_id	0.0161843	0.0659408
item_price	0.3359071	0.2406438
user_id	0.1815961	-0.0289504
order_date.dummy	-0.3685987	-0.0742470
delivery_date.dummy	2.5317200	0.1444404

In the following table, we represent the profit of standard logit model and GA model using test and training set. The first and second columns of the table show the profit of the training set using GA model and standard logit model, respectively and The third and fourth columns of the table show the profit of the test set using GA model and standard logit model, respectively. As we can extract from the table, for both types of datasets, the GA model provides higher profit than standard logit model, i.e. we can improve our cost sensetive logit model for product retun by using genetic algorithm.

```r
#decision cost of training set
 standard.profit = decision.profit(y = train.y.numeric , x = train.x,
                beta = logit$coefficients, itemvalue = tr$item_price)
train.ga.profit = decision.profit(y = train.y.numeric , x = train.x,
                beta = ga.logit@solution[1,], itemvalue = tr$item_price)
#decision cost of test set
test.standard.profit = decision.profit(y = test.y.numeric , x = test.x,
                beta = logit$coefficients, itemvalue = ts$item_price)
test.ga.profit= decision.profit(y = test.y.numeric , x = test.x,
                beta = ga.logit@solution[1,], itemvalue = ts$item_price)
profit = data.frame(train.ga.profit,
                    standard.profit, test.ga.profit, test.standard.profit)
kable(profit,
caption= 'profit of standard logit model and GA model using test and training set')
```

Table 6: profit of standard logit model and GA model using test and training set

train.ga.profit	standard.profit	test.ga.profit	test.standard.profit
0.0133151	-1.816389	0.0181775	-1.81614

7 Conclusion

As already described, the objective of this term paper is to develop a product return model, which can predict with a high accuracy if a new customer is returner or non returner. There are some machine learning methods, e.g. logistic regression, random forest and boosting, which are very pupular and useful for modeling binary target variable. For our analysis, We choose boosting method, which is a method of sequential improvement of a model by using some weak learners. The identified boosting model provides good prediction for product return. Since the identified model is found by using feature engineering and parameter tuning, one can get a better model by using a better parameter tuning and feature engineering.

We are also given a special model challenge, which is to improve the standard logitmodel by picking the coefficients to optimize the profit directly. For choosing the coefficients, we use **GA**, which is a algorithm for finding coefficients by optimizing a given fitness function. The result of **GA** on product return data shows us that the standard logit model can be improved(in the sense of profit maximization) by using **GA**.

Declaration of Authorship

I hereby confirm that I have authored this term paper independently and without use of others than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such.

Berlin, March 18, 2019, S.M. Safiul Alom ....

References

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. "An Introduction to Statistical Learning." In. Springer.

Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller and. 2019. *Package Dplyr*. https://cran.r-project.org/web/packages/dplyr/dplyr.pdf.

Hadley Wickham, Winston Chang, and Lionel Henry. 2018. *Package Ggplot2*. https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf.

Introduction to Genetic Algorithm & Their Application in Data Science. 2017. https://www.analyticsvidhya.com/blog/2017/07/introduction-to-genetic-algorithm/.

Introduction to Optimization with Genetic Algorithm. 2018. https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b.

Lessmann, Stefan. 2019a. *Ensemble Learning*.

———. 2019b. *Evaluation and Estimation of Marketing Decision Support Models*.

———. 2019c. *Imbalanced & Cost-Sensitive Learning*.

Package Caret. 2018. https://cran.r-project.org/web/packages/caret/caret.pdf.

Package Mlr. 2018. https://cran.r-project.org/web/packages/mlr/mlr.pdf.

Tianqi Chen, Tong He, Michael Benesty, and Vadim Khotilovich. 2019. *Package Xgboost*. https://cran.r-project.org/web/packages/xgboost/xgboost.pdf.