Product sales prediction

Beatriz Estrella

26/10/2020

Contents

1. INTRODUCTION

1.1. Project goal

The motivation of this project is to complete the task of the course in Data Science: Capstone from HarvardX, course PH125.9x, under the *Choose your own!* project submission.

The project aims to predict the units sold for each specific product. The database was downloaded from - Kaggle and consists of data from platform wish. *Wish* is a retailer that sells millions of product to customers in a e-commerce marketplace. The data consists of the products under the tag "summer" showed during August 2020.

We want to see which features can predict the units that will be sold in a specific product, and the accuracy that we get under the different machine learning algorithms.

This is a multinomial classification problem as units sold will be treated as a categorical value, due to the low number of different values we have for this column in the dataset. That way, we will use different Machine Learning algorithms that can predict under this assumption.

Apart from the main dataset (stored as data frame in main variable), we have a support dataset that contains the main tag categories and the count on how many times they are used in the platform, thus we can deduce that the higher the count, the more relevant they are (stored as data frame in cat variable).

Datasets, .R code and all relevant documentation regarding this project can be found online at github.com/beatrizeg/Wish-Units-Solds.

1.2. Inspecting the dataset

Now we proceed to inspect the main dataset that is directly downloaded from Kaggle. Using the dim and summary functions we see:

```
dim(main)
```

```
## [1] 1573   43
```

```
summary(main)
```

```
##     title            title_orig            price          retail_price
##  Length:1573        Length:1573        Min.   : 1.000   Min.   :  1.00
##  Class :character   Class :character   1st Qu.: 5.810   1st Qu.:  7.00
##  Mode  :character   Mode  :character   Median : 8.000   Median : 10.00
##                                        Mean   : 8.325   Mean   : 23.29
##                                        3rd Qu.:11.000   3rd Qu.: 26.00
##                                        Max.   :49.000   Max.   :252.00
##
##  currency_buyer       units_sold     uses_ad_boosts       rating
##  Length:1573        Min.   :     1   Min.   :0.0000   Min.   :1.000
##  Class :character   1st Qu.:   100   1st Qu.:0.0000   1st Qu.:3.550
##  Mode  :character   Median :  1000   Median :0.0000   Median :3.850
##                     Mean   :  4339   Mean   :0.4329   Mean   :3.821
##                     3rd Qu.:  5000   3rd Qu.:1.0000   3rd Qu.:4.110
##                     Max.   :100000   Max.   :1.0000   Max.   :5.000
##
##   rating_count     rating_five_count rating_four_count rating_three_count
##  Min.   :    0.0   Min.   :    0.0   Min.   :   0.0    Min.   :   0.0  
##  1st Qu.:   24.0   1st Qu.:   12.0   1st Qu.:   5.0    1st Qu.:   4.0  
##  Median :  150.0   Median :   79.0   Median :  31.5    Median :  24.0  
##  Mean   :  889.7   Mean   :  442.3   Mean   : 179.6    Mean   : 134.6  
##  3rd Qu.:  855.0   3rd Qu.:  413.5   3rd Qu.: 168.2    3rd Qu.: 129.2  
##  Max.   :20744.0   Max.   :11548.0   Max.   :4152.0    Max.   :3658.0  
##                    NA's   :45        NA's   :45        NA's   :45      
##  rating_two_count  rating_one_count   badges_count    badge_local_product
##  Min.   :   0.00   Min.   :   0.00   Min.   :0.0000   Min.   :0.00000    
##  1st Qu.:   2.00   1st Qu.:   4.00   1st Qu.:0.0000   1st Qu.:0.00000    
##  Median :  11.00   Median :  20.00   Median :0.0000   Median :0.00000    
##  Mean   :  63.71   Mean   :  95.74   Mean   :0.1055   Mean   :0.01844    
##  3rd Qu.:  62.00   3rd Qu.:  94.00   3rd Qu.:0.0000   3rd Qu.:0.00000    
##  Max.   :2003.00   Max.   :2789.00   Max.   :3.0000   Max.   :1.00000    
##  NA's   :45        NA's   :45                                            
##  badge_product_quality badge_fast_shipping     tags          
##  Min.   :0.00000       Min.   :0.00000     Length:1573       
##  1st Qu.:0.00000       1st Qu.:0.00000     Class :character  
##  Median :0.00000       Median :0.00000     Mode  :character  
##  Mean   :0.07438       Mean   :0.01271                       
##  3rd Qu.:0.00000       3rd Qu.:0.00000                       
##  Max.   :1.00000       Max.   :1.00000                       
##                                                              
##  product_color      product_variation_size_id product_variation_inventory
##  Length:1573        Length:1573               Min.   : 1.00              
##  Class :character   Class :character          1st Qu.: 6.00              
##  Mode  :character   Mode  :character          Median :50.00              
##                                               Mean   :33.08              
##                                               3rd Qu.:50.00              
##                                               Max.   :50.00              
##                                                                          
##  shipping_option_name shipping_option_price shipping_is_express
##  Length:1573          Min.   : 1.000        Min.   :0.000000   
##  Class :character     1st Qu.: 2.000        1st Qu.:0.000000   
##  Mode  :character     Median : 2.000        Median :0.000000   
##                       Mean   : 2.345        Mean   :0.002543   
##                       3rd Qu.: 3.000        3rd Qu.:0.000000   
##                       Max.   :12.000        Max.   :1.000000   
##                                                                
##  countries_shipped_to inventory_total has_urgency_banner urgency_text      
##  Min.   :  6.00       Min.   : 1.00   Min.   :1          Length:1573       
##  1st Qu.: 31.00       1st Qu.:50.00   1st Qu.:1          Class :character  
##  Median : 40.00       Median :50.00   Median :1          Mode  :character  
##  Mean   : 40.46       Mean   :49.82   Mean   :1                            
##  3rd Qu.: 43.00       3rd Qu.:50.00   3rd Qu.:1                            
##  Max.   :140.00       Max.   :50.00   Max.   :1                            
##                                       NA's   :1100                         
##  origin_country     merchant_title     merchant_name     
##  Length:1573        Length:1573        Length:1573       
##  Class :character   Class :character   Class :character  
##  Mode  :character   Mode  :character   Mode  :character  
##                                                          
##                                                          
## 
```

```
##
##  merchant_info_subtitle merchant_rating_count merchant_rating
##  Length:1573            Min.   :      0       Min.   :2.333
##  Class :character       1st Qu.:   1987       1st Qu.:3.917
##  Mode  :character       Median :   7936       Median :4.041
##                         Mean   :  26496       Mean   :4.032
##                         3rd Qu.:  24564       3rd Qu.:4.162
##                         Max.   :2174765       Max.   :5.000
##
##  merchant_id        merchant_has_profile_picture merchant_profile_picture
##  Length:1573        Min.   :0.0000               Length:1573
##  Class :character   1st Qu.:0.0000               Class :character
##  Mode  :character   Median :0.0000               Mode  :character
##                     Mean   :0.1437
##                     3rd Qu.:0.0000
##                     Max.   :1.0000
##
##  product_url        product_picture     product_id           theme
##  Length:1573        Length:1573        Length:1573        Length:1573
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##  crawl_month
##  Length:1573
##  Class :character
##  Mode  :character
##
##
##
##
```

The dataset consists of 1573 rows and 43 columns. We see as well that we have character and numeric values and that we also have some NAs in the dataset.

First thing we have to take into account now, is that having only ~1k of data points is not a lot. This will probably lead to the accuracy of the prediction not being too high. We will be able to check on this later on.

To check which columns gives us NAs we use.

```r
nas <- apply(main, 2, function(x) any(is.na(x)))
knitr::kable(nas[which(nas)], caption = "Check columns with NAs")
```

Table 1: Check columns with NAs

	x
rating_five_count	TRUE
rating_four_count	TRUE
rating_three_count	TRUE
rating_two_count	TRUE
rating_one_count	TRUE

	x
product_color	TRUE
product_variation_size_id	TRUE
has_urgency_banner	TRUE
urgency_text	TRUE
origin_country	TRUE
merchant_name	TRUE
merchant_info_subtitle	TRUE
merchant_profile_picture	TRUE

So we proceed to substitute the NAs in *rating___count* columns to 0 and also the ones in the *has_urgency_banner* to 0 as 0 represents *FALSE*.

```r
main <- main %>% mutate(rating_five_count=ifelse(is.na(rating_five_count),0,rating_five_count),
                        rating_four_count=ifelse(is.na(rating_four_count),0,rating_four_count),
                        rating_three_count=ifelse(is.na(rating_three_count),0,rating_three_count),
                        rating_two_count=ifelse(is.na(rating_two_count),0,rating_two_count),
                        rating_one_count=ifelse(is.na(rating_one_count),0,rating_one_count),
                        has_urgency_banner=ifelse(is.na(has_urgency_banner),0,has_urgency_banner))
```

We see that we still get NAs in columns *product_color*, *product_variation_size_id*, *urgency_text*, *origin_country*, *merchant_name*, *merchant_info_subtitle* and *merchant_profile_picture*, but we will deal with these later as we will see, because these features will not be included in the model, except for *product_color*, *product_variation_size_id* and *merchant_profile_picture* that will be tidied up later on.

Lastly, it is important to note that the *product_id* is the unique differentiator of each product. We will see later the reason for a few of them being duplicated and solve the issue.

1.3. Libraries

All of these these libraries will be loaded:

```r
library(stringr)
library(purrr)
library(caret)
library(ggplot2)
library(corrplot)
library(forcats)
library(rattle)
library(xgboost)
library(klaR)
library(h2o)
library(knitr)
library(tictoc)
```

2. METHOD AND ANALYSIS

2.1. Exploration of the dataset

Now we proceed to inspect the different features/variables in the dataset, taking into account that the value we want to be able to predict is *units_sold*.

2.1.1. Checking features variability and adjusting

product_color First of all we are going to study the different categories under the predictor of *product_color*. We can see all the different colors we get by showing the table and histogram below.

```
knitr::kable(table(main$product_color) %>% sort(decreasing = TRUE), caption = "Product color values")
```

Table 2: Product color values

Var1	Freq
black	302
white	254
yellow	105
blue	99
pink	99
red	93
green	90
grey	71
purple	53
armygreen	31
navyblue	28
winered	28
orange	27
multicolor	20
beige	14
khaki	12
lightblue	12
gray	11
white & green	10
rosered	8
skyblue	8
brown	7
coffee	7
darkblue	6
floral	5
rose	5
black & green	4
fluorescentgreen	4
leopard	4
lightpink	4
navy	4
Black	3
black & white	3
camouflage	3
lightgreen	3
orange-red	3
White	3
applegreen	2
apricot	2
Army green	2
black & blue	2
black & yellow	2
burgundy	2

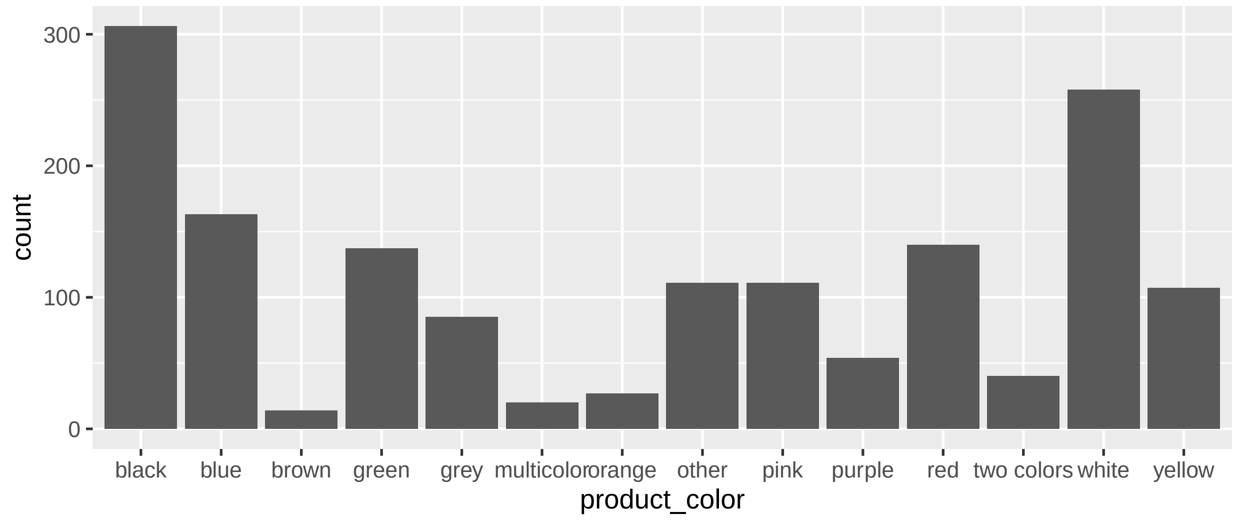
Var1	Freq
camel	2
coolblack	2
coralred	2
dustypink	2
lakeblue	2
lightred	2
lightyellow	2
mintgreen	2
navy blue	2
Pink	2
pink & black	2
pink & blue	2
pink & grey	2
pink & white	2
silver	2
watermelonred	2
white & black	2
whitefloral	2
wine	2
army	1
army green	1
black & stripe	1
blackwhite	1
Blue	1
blue & pink	1
brown & yellow	1
claret	1
darkgreen	1
denimblue	1
gold	1
gray & white	1
greysnakeskinprint	1
ivory	1
jasper	1
leopardprint	1
light green	1
lightgray	1
lightgrey	1
lightkhaki	1
lightpurple	1
navyblue & white	1
nude	1
offblack	1
offwhite	1
orange & camouflage	1
prussianblue	1
rainbow	1
RED	1
red & blue	1
Rose red	1
rosegold	1
star	1

Var1	Freq
tan	1
violet	1
white & red	1
whitestripe	1
wine red	1
winered & yellow	1

We see that most of the categories only apply for 2 or less *products_id*, so we will group into main color categories this feature in order to provide the algorithm with more valuable data.

```r
main <- main %>% mutate(product_color=
                          as.factor(case_when(
                            str_detect(product_color, "&") ~ "two colors",
                            str_detect(product_color, "blue") ~ "blue",
                            str_detect(product_color, "navy") ~ "blue",
                            str_detect(product_color, "green") ~ "green",
                            str_detect(product_color, "red") ~ "red",
                            str_detect(product_color, "gray") ~ "grey",
                            str_detect(product_color, "grey") ~ "grey",
                            str_detect(product_color, "coffee") ~ "brown",
                            str_detect(product_color, "brown") ~ "brown",
                            str_detect(product_color, "pink") ~ "pink",
                            str_detect(product_color, "rose") ~ "pink",
                            str_detect(product_color, "black") ~ "black",
                            str_detect(product_color, "white") ~ "white",
                            str_detect(product_color, "purple") ~ "purple",
                            str_detect(product_color, "orange") ~ "orange",
                            str_detect(product_color, "multicolor") ~ "multicolor",
                            str_detect(product_color, "yellow") ~ "yellow",
                            TRUE ~ "other")))

main %>% ggplot(aes(product_color))+geom_bar()
```



Now we get just a few main categories which allows as to treat this feature as a categorical value.

product_variation_size_id We are going to do the same exercise we did with the *product_color* variable with the *product_variation_size_id*. We can check the high variability and low information it provides as it is given in the dataset.

```
knitr::kable(table(main$product_variation_size_id) %>% sort(decreasing = TRUE),
             caption = "Product size values")
```

Table 3: Product size values

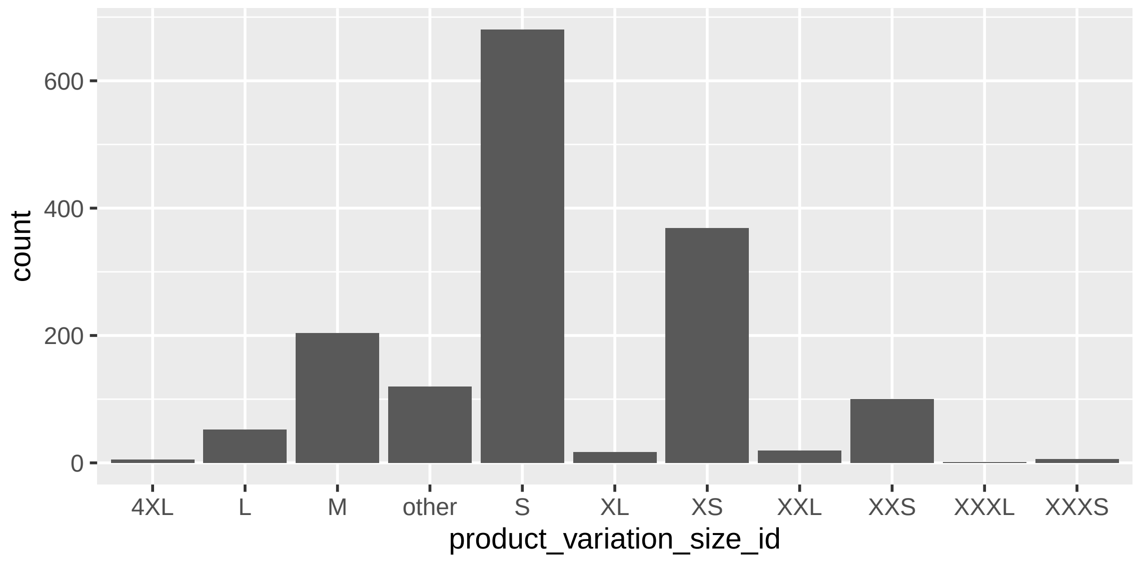
Var1	Freq
S	641
XS	356
M	200
XXS	100
L	49
S.	18
XL	17
XXL	15
XXXS	6
4XL	5
s	5
Size S	5
XS.	5
2XL	4
M.	4
SIZE XS	4
Size-XS	4
10 ml	3
2pcs	3
33	3
34	3
Size-S	3
SizeL	3
1	2
1 pc.	2
25	2
29	2
35	2
3XL	2
5XL	2
EU 35	2
One Size	2
S Pink	2
S(bust 88cm)	2
Size -XXS	2
Size M	2
size S	2
Size S.	2
SIZE-XXS	2
Size4XL	2
Suit-S	2
XXXXL	2
XXXXXL	2

Var1	Freq
04-3XL	1
1 PC - XL	1
100 cm	1
100 x 100cm(39.3 x 39.3inch)	1
100pcs	1
10pcs	1
17	1
1m by 3m	1
1pc	1
2	1
20pcs	1
20PCS-10PAIRS	1
25-S	1
26(Waist 72cm 28inch)	1
3 layered anklet	1
30 cm	1
32/L	1
36	1
4	1
4-5 Years	1
40 cm	1
5	1
5PAIRS	1
60	1
6XL	1
80 X 200 CM	1
AU plug Low quality	1
B	1
Baby Float Boat	1
Base & Top & Matte Top Coat	1
Base Coat	1
choose a size	1
daughter 24M	1
EU39(US8)	1
first generation	1
Floating Chair for Kid	1
H01	1
L.	1
Pack of 1	1
pants-S	1
Round	1
S (waist58-62cm)	1
S Diameter 30cm	1
S..	1
S(Pink & Black)	1
S/M(child)	1
SIZE S	1
Size XXS	1
SIZE XXS	1
Size–S	1
SIZE-4XL	1
Size-5XL	1

Var1	Freq
Size-L	1
Size-XXS	1
Size/S	1
US 6.5 (EU 37)	1
US-S	1
US5.5-EU35	1
White	1
Women Size 36	1
Women Size 37	1
X L	1
XXXL	1

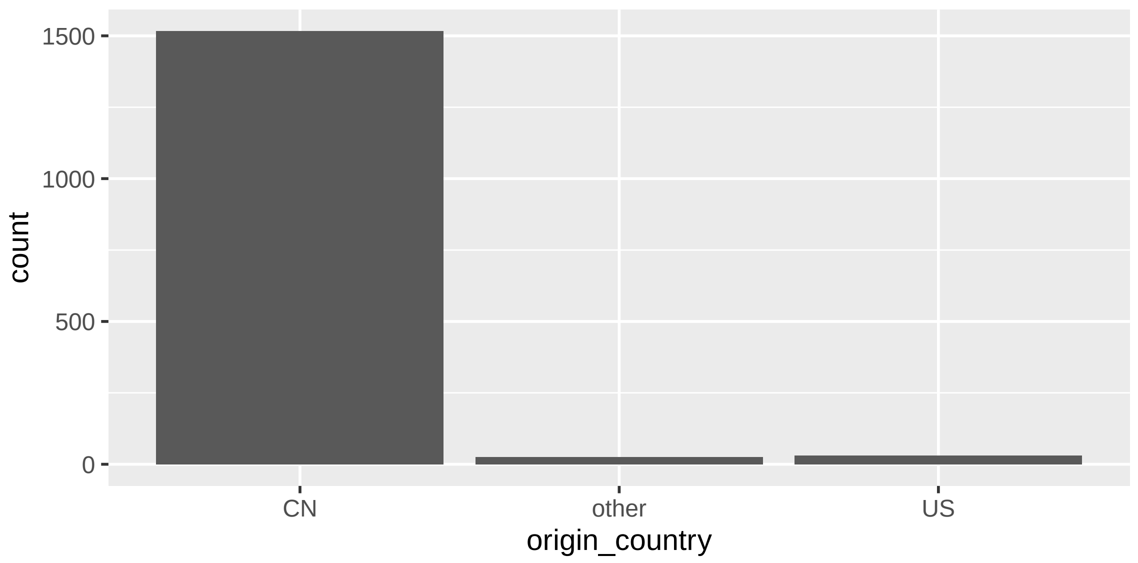
We again reassign the values to the main categories and again, we treat it as a categorical value, which provides much more information to the algorithms.

```
main %>% ggplot(aes(product_variation_size_id))+geom_bar()
```



origin_country We again see variability for *origin_country*, and reassign converting the variable to factor.

```
##
##   CN   US   VE   SG   AT   GB
## 1516   31    5    2    1    1
```



currency_buyer We check that there is only one currency in the dataset and that no unification of units is needed.

```
n_distinct(main$currency_buyer)
```

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

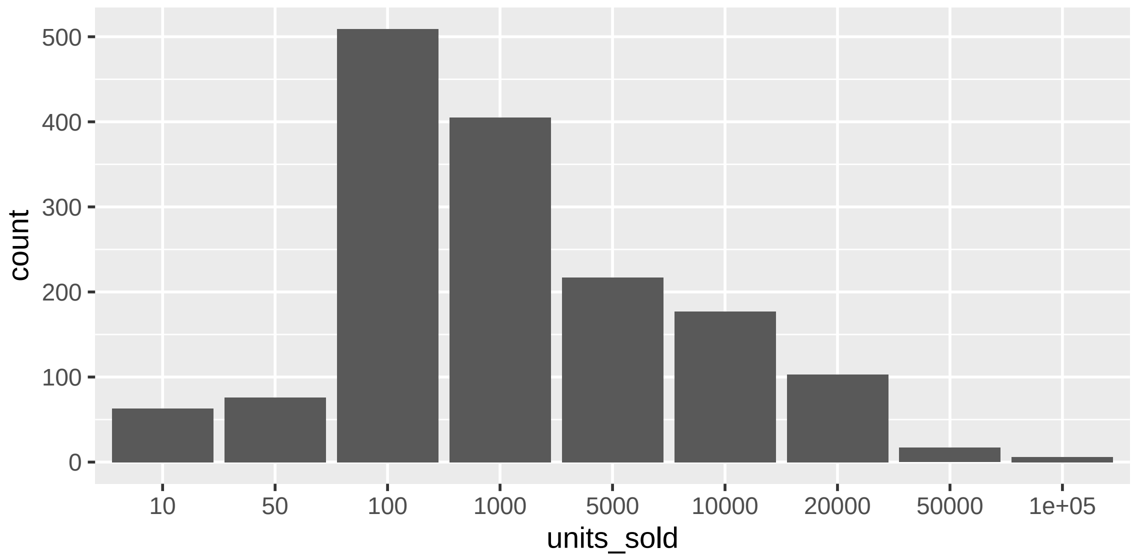
units_sold Now we study the characteristics of the variable we want to predict.

```
knitr::kable(table(main$units_sold) %>% sort(decreasing = TRUE), caption = "Units sold values and freque
```

Table 4: Units sold values and frequency

Var1	Freq
100	509
1000	405
5000	217
10000	177
20000	103
50	76
10	49
50000	17
1e+05	6
8	4
1	3
2	2
3	2
7	2
6	1

We see that there are only 15 different values, and in fact six of them are below 10 so we could group this into 9 different categories and treat the project as a categorical problem. This is what we will do.



product_id As we commented in the introduction, we can easily check that there are less unique *product_id* values than rows in the dataset. In fact, there are 1341 different *product_id* and 1573 rows.

```
n_distinct(main$product_id)
```

```
## [1] 1341
```

So if I examine a duplicated *product_id* (ex. "5577faf03cef83230c39d0c3") and see the differences in the rows:

```
knitr::kable(main %>% group_by(product_id) %>% summarize(n=n()) %>% arrange(desc(n))
             %>% head(10), caption = "Examining duplicates")
```

Table 5: Examining duplicates

product_id	n
5c80e8a150c63d28c67b8f14	3
5cde56ea6bbbd86b1cbab4a8	3
5cedf93ac0baab7389f4ccd7	3
5dea1d9cec016f062ce8aab1	3
5e142dee04c3e579e89576a3	3
5e16cb87e6dd7c03be24b28a	3
5e93d60ebc5446aedde50c50	3
5e9932cab3eafb25c00ba79f	3
5e9a74e447f7d92c8db8d14b	3
5e9dad8cbc19c300417e1733	3

We can see that the almost every column acquires the same value, except for *has_urgency_banner*. Thus we can easily deduce that during the month, this feature was changed for the *product_id* and a new row was created. As the impact is minimal and we do not know which of the rows was active during most of the month, we will just simply delete the duplicated rows as:

```
main <- distinct(main, product_id, .keep_all = TRUE)
```

2.1.2. Assigning classes to features and calculating % stars rating instead of total count

Now we have studied the variability of the features that had a class "character" and reassigned them the class "factor" for the algorithms to work better, a few more predictors that acquire few different values are assigned to factor as well. We also saw that some features acquired either a 0 or a 1, so we will change this to logical class.

For the columns of rating_X_count, instead of keeping the total count, we will keep the percentage by dividing for each * by the total counts.

```
main <- main %>% mutate(currency_buyer=as.factor(currency_buyer),
                        badges_count=as.factor(badges_count),
                        uses_ad_boosts=as.logical(uses_ad_boosts),
                        badge_local_product=as.logical(badge_local_product),
                        badge_product_quality=as.logical(badge_product_quality),
                        badge_fast_shipping=as.logical(badge_fast_shipping),
                        shipping_option_price=as.factor(shipping_option_price),
                        shipping_is_express=as.logical(shipping_is_express),
                        has_urgency_banner=as.logical(has_urgency_banner),
                        merchant_has_profile_picture=as.logical(merchant_has_profile_picture),
                        inventory_total=as.factor(inventory_total))

main <- main %>% mutate(rating_five_count=rating_five_count/rating_count,
                        rating_four_count=rating_four_count/rating_count,
                        rating_three_count=rating_three_count/rating_count,
                        rating_two_count=rating_two_count/rating_count,
                        rating_one_count=rating_one_count/rating_count)

main <- main %>% mutate(rating_five_count=ifelse(is.na(rating_five_count),0,rating_five_count),
                        rating_four_count=ifelse(is.na(rating_four_count),0,rating_four_count),
                        rating_three_count=ifelse(is.na(rating_three_count),0,rating_three_count),
                        rating_two_count=ifelse(is.na(rating_two_count),0,rating_two_count),
                        rating_one_count=ifelse(is.na(rating_one_count),0,rating_one_count))
```

2.1.3. Introducing tags model

In the main dataset, we can see a column named *tags* that includes all tags that were given to each *product_id*. In the cat dataset, there are two columns, the column *counts* gives the number of times each tag, which is in column *keyword*, appears in the main dataset. We can assume then, that those keywords that appear the most, are more relevant than those than appear the less. Thus we create a new column in the cat dataset, named *cat_n*, with a number that goes from 1 to 4, with the idea of weighting more those most popular, as in code below. What we are trying to do here is to have a column that provides a relevance number for the tags that were used in the *product_id*.

```
cat <- cat %>% mutate(cat_n =
                        case_when(count>=1000 ~ 4,
                                  count<1000 & count>=500 ~ 3,
                                  count<500 & count>=200 ~ 2,
                                  count < 200 ~ 1,
                                  TRUE ~ 0))
```

Once we have this, we want to transfer this information to the main dataset. First, the *tags* column contains all tags separated by a comma. We split all the tags into different columns, getting 41 columns. Then, we substitute each tag for its value *cat_n* that we assigned in the cat dataset. Once we have the values in *numeric* class, we sum all tag values for each row or *product_id* and bind this new column, named *n_tags* to the main dataset.

```
main_tags <- str_split(main$tags, ",", simplify = TRUE)

for (i in 1:41){
main_tags[,i] <- with(cat, cat_n[match(main_tags[,i], keyword)])
} #next step change to numeric values

main_tags <- as.data.frame(main_tags)
main_tags[] <- lapply(main_tags, function(x) as.numeric(as.character(x)))
main_tags <- main_tags %>% mutate(n_tags = rowSums(main_tags, na.rm=TRUE)) %>% dplyr::select(n_tags)
main_m <- bind_cols(main, main_tags)
```

Lastly, we disregards all 41 columns that were created and keep only those that can provide relevant information *(price, retail_price, units_sold, uses_ad_boosts, rating, rating_count, rating_five_count, rating_four_count, rating_three_count, rating_two_count, rating_one_count, badges_count, badge_local_product, badge_product_quality, badge_fast_shipping, product_color, product_variation_size_id, product_variation_inventory, shipping_option_price, shipping_is_express, countries_shipped_to, inventory_total, has_urgency_banner, origin_country, merchant_rating_count, merchant_rating, merchant_has_profile_picture, product_id, n_tags)*, disregarding also those such as *title*, *merchant_name*, *product_url*, etc.

2.1.4. Predictors that do not vary across sample

We are going to do a quick check on predictors that maintain close to the same value across all the sample to evaluate if we want to disregard them. We will use:

```
no_var <- nearZeroVar(main_m, saveMetrics = TRUE)
knitr::kable(no_var[no_var[,"zeroVar"] + no_var[,"nzv"] > 0, ],
             caption = "Predictors with near zero variability")
```

Table 6: Predictors with near zero variability

	freqRatio	percentUnique	zeroVar	nzv
badge_local_product	46.89286	0.1491424	FALSE	TRUE
badge_fast_shipping	69.57895	0.1491424	FALSE	TRUE
shipping_is_express	334.25000	0.1491424	FALSE	TRUE
inventory_total	665.50000	0.7457122	FALSE	TRUE
origin_country	47.92593	0.2237136	FALSE	TRUE

As we can see, there are no predictors with zero variance, but there are 5 predictors with near zero variance. For those five, only *inventory_total* is unique for the 74.5% of the *product_id*, the rest have higher variance. As we are not sure how having a low value in this column might impact the units sold, we decide to still keep all of them.

2.1.5. Adding *perc_price* column

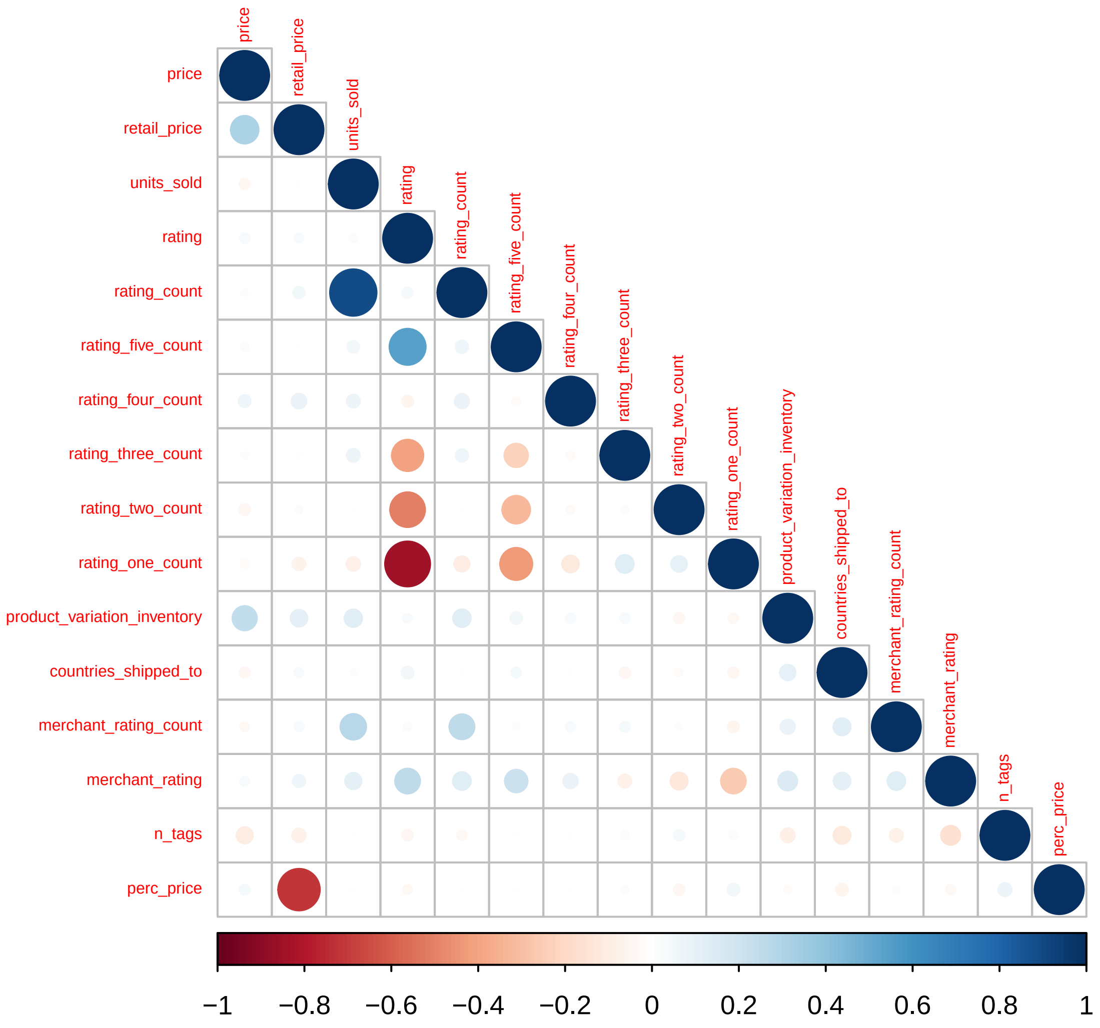
From the dataset we see that we have a *price* column, and a *retail_price* column. *Price* is the price for which the product is being sold in the platform, and *retail_price* is the price that similar products have in the same or in other platforms, we can say, a benchmark price. To include any possible sales effect, we add a new column that measures if the price is higher or lower than average.

```r
main_m <- main_m %>% mutate(perc_price=(price-retail_price)/retail_price)
```

2.2. Studying correlation between variables

To study correlation between variables we are going to do two different plots. First, we will do the correlation matrix for the numeric variables.

```r
main_m.cor <- main_m %>% mutate(units_sold=as.numeric(units_sold)) %>%
  dplyr::select_if(is.numeric) %>%
  cor(.)
corrplot(main_m.cor, type="lower", tl.cex = 0.5)
```



We can see that the units sold are highly affected by the total number of ratings, thus by the count of ratings for every star, but we cannot see a high relation for more units sold with a higher rating or higher % of five star ratings. As well, there is a correlation with the number of ratings for the merchant and its rating. There is also a slight positive correlation with the *product_variation_inventory* variable.

To study correlation between non numerical variables, we will perform a Chi-squared test of independence, looking at the p-values. We can say that two different variables are independent if the probability distribution of one is not affected by the presence of the other.

In this case, the null hypotheses is that the variables are independent from the units sold. With a significance level of 0.05, we will test the hypotheses of them being independent.

```
main_m.chisq <- main_m %>%
  dplyr::select_if(function(col) is.character(col) |
              is.factor(col) | is.logical(col) |
              all(col == .$units_sold)) %>% dplyr::select(-product_id)

columns <- 1:ncol(main_m.chisq)
```

```r
vars <- names(main_m.chisq)[columns]
out <-  apply( combn(columns,2),2,function(x){
  chisq.test(table(main_m.chisq[,x[1]],main_m.chisq[,x[2]]),correct=F)$p.value
})

out <- cbind(as.data.frame(t(combn(vars,2))),out)
out_dep <- out %>% filter(V1=="units_sold") %>% filter(out<0.05) %>%arrange(out)
knitr::kable(out_dep, caption = "Dependent categorical variables from units_sold")
```

Table 7: Dependent categorical variables from units_sold

V1	V2	out
units_sold	merchant_has_profile_picture	0.0000007
units_sold	product_variation_size_id	0.0000049
units_sold	badge_product_quality	0.0008471
units_sold	shipping_option_price	0.0066469
units_sold	uses_ad_boosts	0.0086633
units_sold	badges_count	0.0187025

We can see that we get p-values below 0.05 for 6 variables, thus, we can reject in these cases the null hypothesis and assume that these are not independent to units_sold.

Also, we can assume that these 7 other predictors are independent and do not affect the output of the units sold. We will be able to check on this when we study the variable importance of the machine learning algorithms.

```r
out_ind <- out %>% filter(V1=="units_sold") %>% filter(out>=0.05) %>% arrange(out)
knitr::kable(out_ind, caption = "Independent categorical variables from units sold")
```

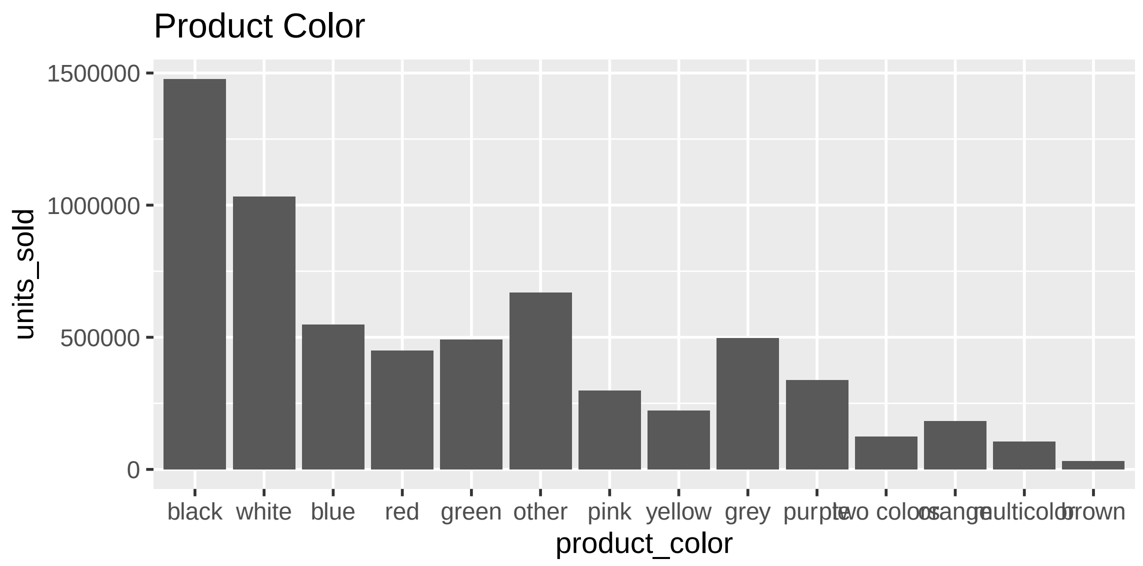
Table 8: Independent categorical variables from units sold

V1	V2	out
units_sold	badge_fast_shipping	0.0535864
units_sold	inventory_total	0.0770737
units_sold	origin_country	0.4893234
units_sold	product_color	0.6832928
units_sold	badge_local_product	0.8983125
units_sold	shipping_is_express	0.9672496
units_sold	has_urgency_banner	0.9904042

2.3. Checking predictors effect - graphs

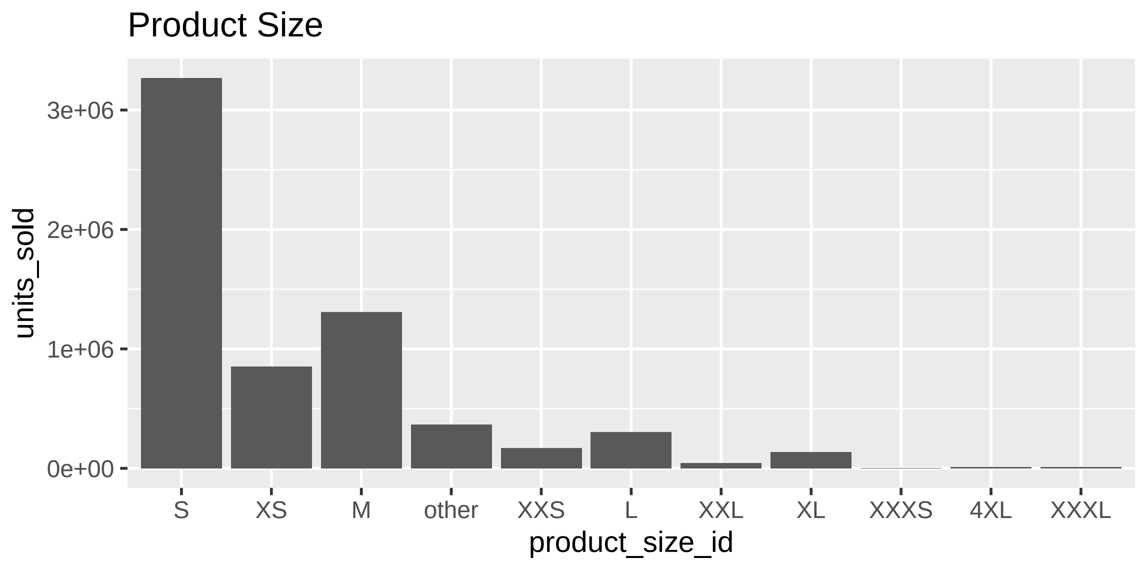
In this section we will show different graphs with the relationship between units sold and other predictors.

product_color



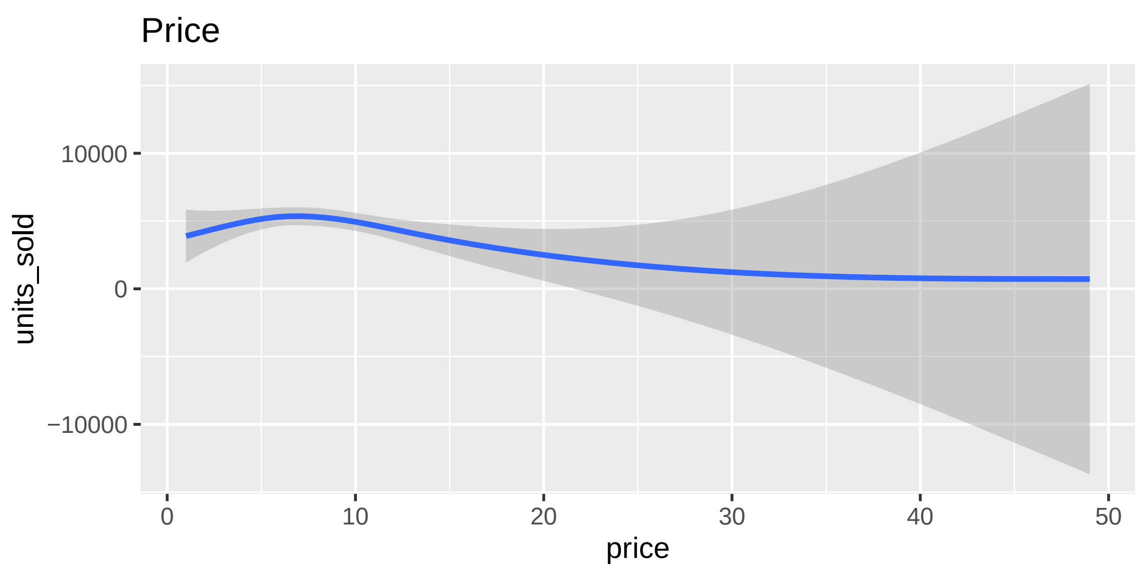
Black and white are the colors most sold in the platform.

product_size_id



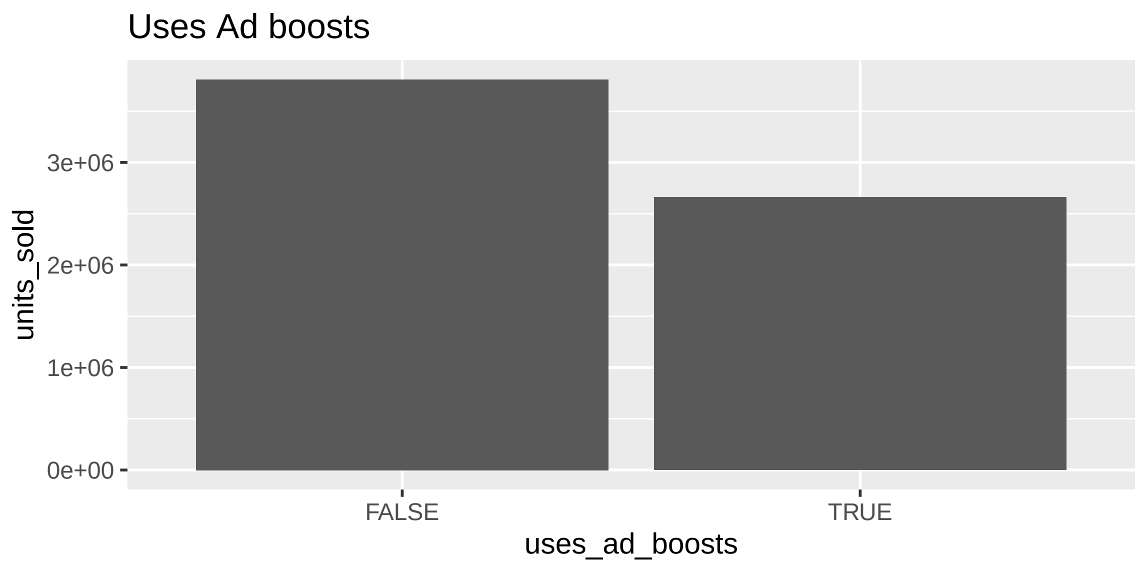
S is the most common size within units sold.

price



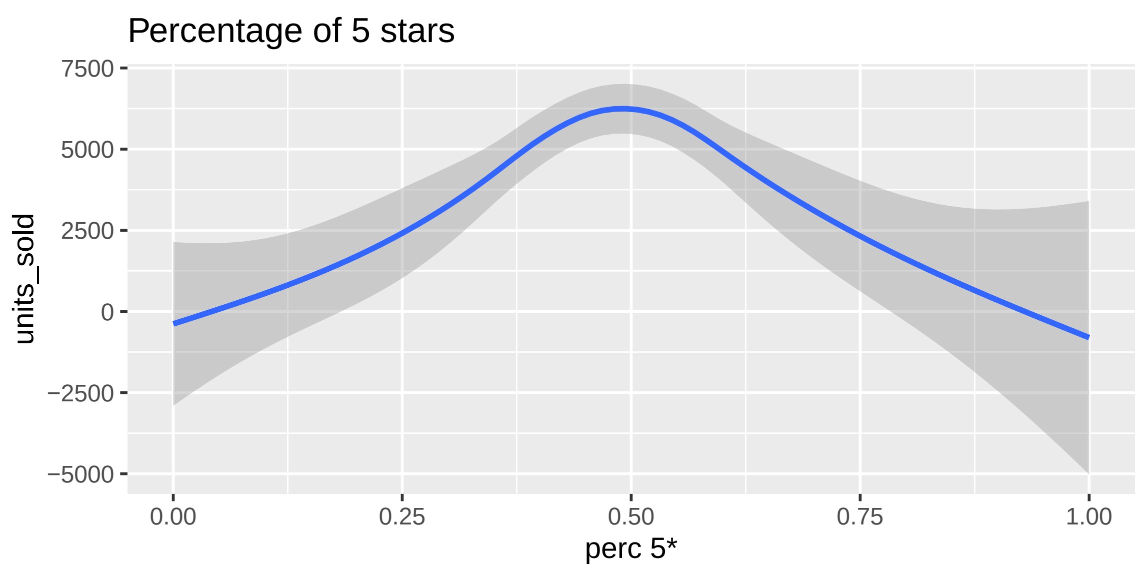
Prices within 5 to 10 EUR are the most common for units sold, and high prices are not common, reason for the smooth function to cover a big area for those prices.

uses_ad_boosts



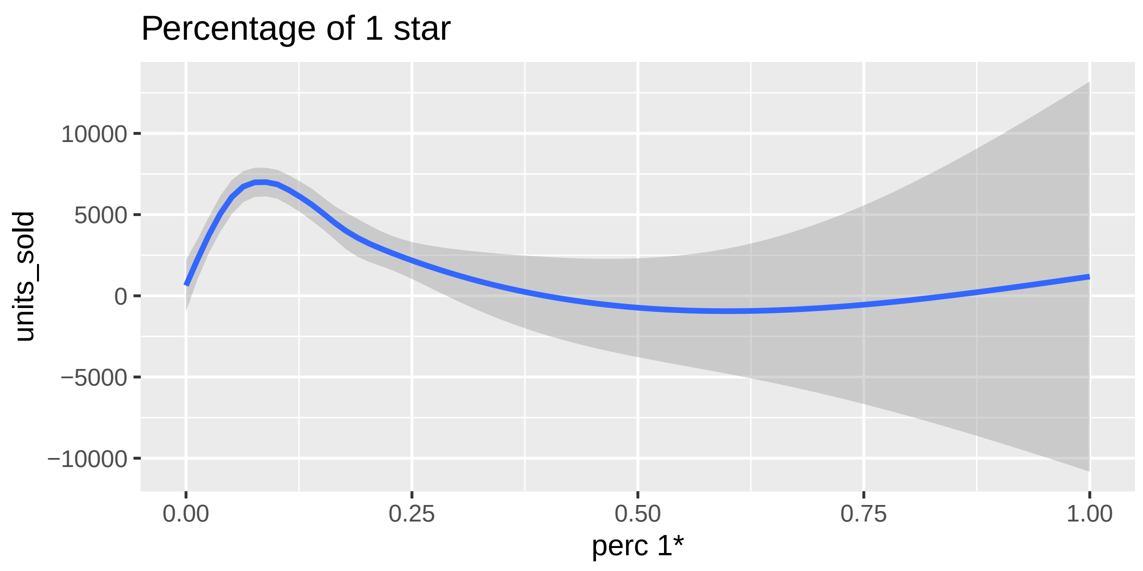
We can see that it is more common in units sold to not use ad_boosts.

5 star %



Here, the plot shows how most units are sold with an average of 50% of 5 star ratings.

1 star %



Opposite to the perc 5* graph, we can see that the most units are sold with a low percentage of 1 star ratings.

Lastly, as we commented before, we change the class of *units_sold* to factor, so that it is categorical.

```r
levels <- c("10", "50", "100", "1000", "5000", "10000", "20000", "50000", "1e+05")
main_p <- main_m %>% mutate(units_sold = factor(units_sold, levels=levels))
```

2.4. Creation of train and test set

We split our data into a train set where we run the algorithms, and a test set to test the results and see how good our algorithm did. Test set usually consists of 10%-20% of the data. In our case, test set will have 15%

of the data and train set 85%. As we do not have too much data, we do not want to compromise test set to have very few rows, or train set to not be big enough to produce good models.

Our train set consists of 1138 rows and test set of 203 rows.

```
set.seed(1, sample.kind = "Rounding")
test_index <- createDataPartition(main_m$units_sold, times=1, p=0.15, list=FALSE)
train_set <- main_p[-test_index,] %>% dplyr::select(-product_id)
test_set <- main_p[test_index,] %>% dplyr::select(-product_id)
```

2.5. Method

The method will be to train different Machine Learning algorithms, that are included in `caret` package. We will train the models using only the train_set, while repeated cross-validation or simple cross-validation will be used to optimize the hyperparameters of each model. Once the model is optimized using only the train_set, results will be tested in test_set.

The methods to use are those appropriate to study a multinomial categorical problem such as this one. Thus, those like *Logistic Regression* or *Linear Models* will be disregarded.

As well, `h2o` package will be used at the end, to check which is the best performer model and compare results.

3. RESULTS

3.1. GAM Loess

Gam Loess is a Generalized Additive Model that keeps the inspiration of Linear Models but incorporating non-linear forms. The model uses smooth functions of the predictors, and the loess function is used to fit. The loess function acts as a local regression as fitting at point x is weighted toward the data closest to x. The data considered from x is controlled by the `span` argument, and represents the percentage of data that will be taken into account.

To optimize the `span` argument, we will use repeated cross validation.

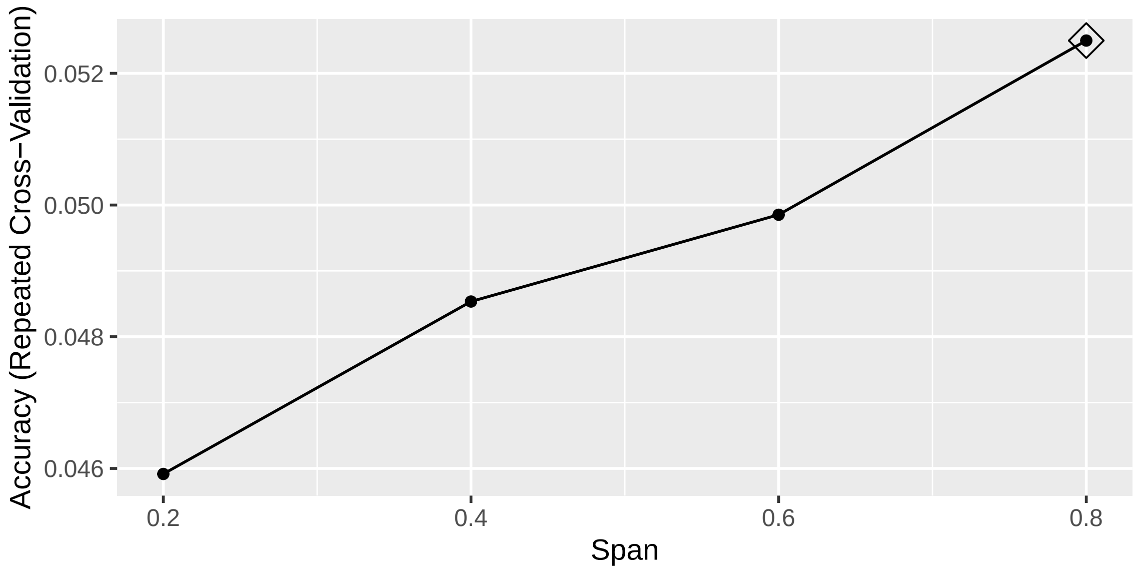
Repeated Cross-Validation has two different values, `number` and `repeats`. `Number` means the times we ramdonly divide our data into, so in this case, we will created 3 different sets to test results while training the model in the other 2. `Repeats` means that we will repeat 4 times the process for each partition, and it will calculate the average.

We only divide into 3 different sets due to the small amount of data we have.

```
tic("GAM Loess")
set.seed(1, sample.kind = "Rounding")
control <- trainControl(method = "repeatedcv", number = 3, repeats = 4,
                        savePredictions = "all")
grid_loess <- expand.grid(span=seq(0.2,0.9,0.2), degree=1)
train_loess <- caret::train(units_sold ~ ., data=train_set, method="gamLoess",
                            trControl=control, tuneGrid=grid_loess)
gam_toc <- toc()
```

```
## GAM Loess: 195.022 sec elapsed
```

```
ggplot(train_loess, highlight = TRUE)
```



So best `span` in this case is 0.8, and we only get an accuracy of 0.052. If we tried guessing, the probability of being correct would be 1/9 = 0.11. So we would do better just by guessing! In fact, if we use the predict function, we see that it is just predicting the same category every time.

```
y_loess <- predict(train_loess, test_set, type="raw")
acc_loess <- confusionMatrix(y_loess, test_set$units_sold)$overall[['Accuracy']]
acc_results <- tibble(method = "Gam Loess",
                      Accuracy_Train = max(train_loess$results$Accuracy),
                      Accuracy_Test = acc_loess,
                      Time = gam_toc$toc - gam_toc$tic)
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 9: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022

This model clearly does not work.

3.2. K nearest neighbors

Now we will use the method of k-nearest neighbors. This method relies on the assumption that similar features will provide the same output. The model equals “similar features” to closeness in distance.

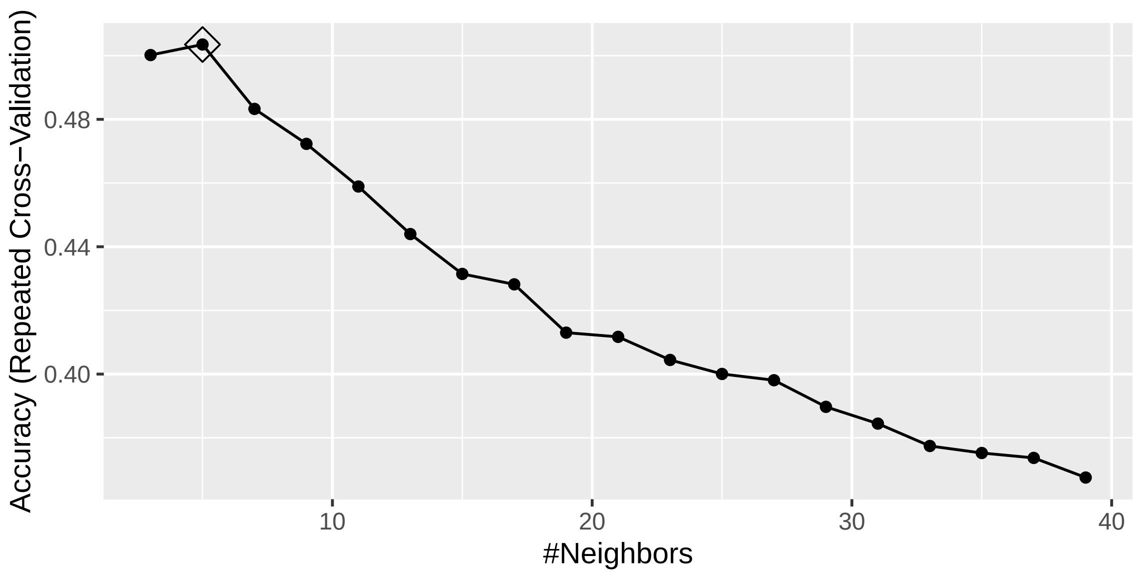
In this case, we need to optimize the k argument, which indicates the number of neighbors to include in every calculation. We use as well repeated cross-validation to find the optimal k.

```
tic("KNN")
set.seed(2007, sample.kind = "Rounding")
control <- trainControl(method = "repeatedcv", number=3, repeats=4)
```

```
train_knn <- train(units_sold ~ ., data=train_set, method="knn",
                   tuneGrid = data.frame(k=seq(3, 40, 2)), trControl=control)
knn_toc <- toc()
```

```
## KNN: 9.884 sec elapsed
```

```
ggplot(train_knn, highlight = TRUE)
```



In this case, we get the higher accuracy with a k=5. In that case, the accuracy in the train_set is 0.503. Now we are doing better than guessing. Let's check our results in the test_set.

```
y_knn <- predict(train_knn, test_set, type="raw")
acc_knn <- confusionMatrix(y_knn, test_set$units_sold)$overall[['Accuracy']]
acc_results <- bind_rows(acc_results,
                         data_frame(method="KNN",
                                    Accuracy_Train = max(train_knn$results$Accuracy),
                                    Accuracy_Test = acc_knn,
                                    Time = knn_toc$toc - knn_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

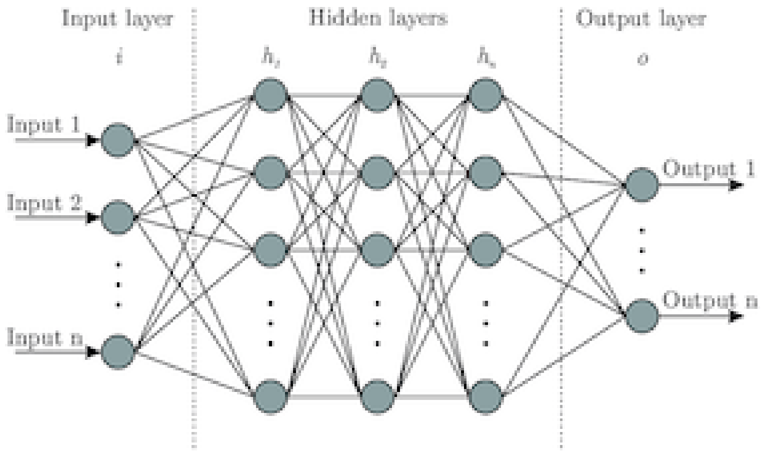
Table 10: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884

It gets even better, now we predict correctly more than half of the units sold.

3.3. Neural networks

In this section, we introduce a Machine Learning algorithm that we did not study during the course, but that is very powerful and easy to understand. It is inspired in replicating the way of working of the neurons in the brain.



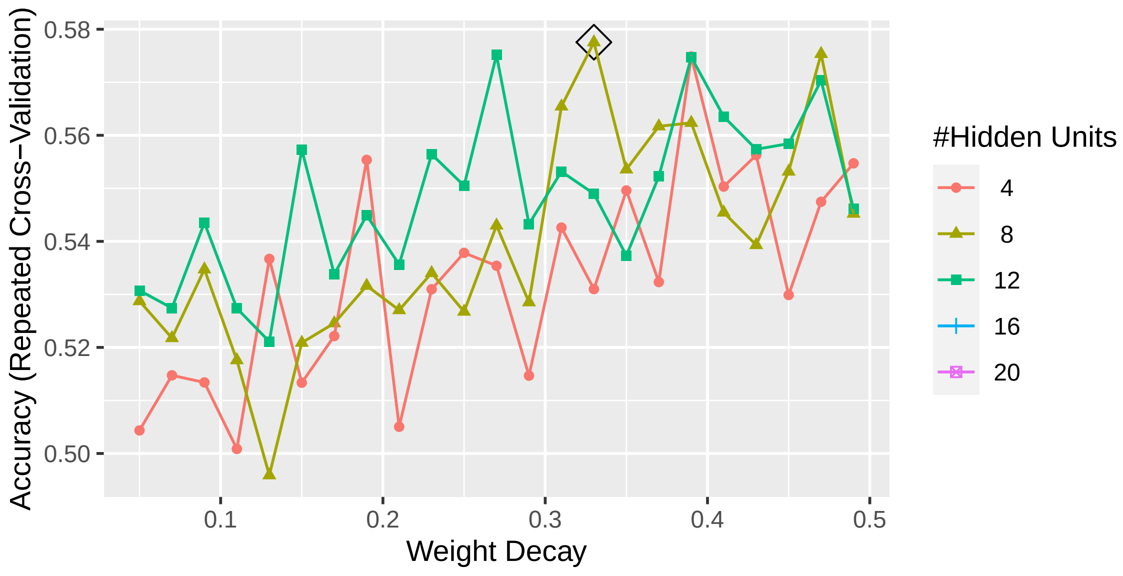
neural network:

Using the `caret` package, we can optimize now two different parameters: *size* and *decay*. *Size* is the number of units in hidden layer and *decay* is a parameter of regularization to avoid over-fitting that can vary between 0 (default) and 1.

We will try with sizes going from 4 to 20 and decays from 0.05 to 0.5, and a few minutes later we get the results.

```
tic("NN1")
set.seed(2007, sample.kind = "Rounding")
control <- trainControl(method = "repeatedcv", number=3, repeats=4)
grid_nnet1 <- expand.grid(size=seq(4,20,4), decay=seq(0.05, 0.5, 0.02))
train_nnet1 <- train(units_sold ~ ., data=train_set, method="nnet", trControl=control,
                     tuneGrid=grid_nnet1)
nn1_toc <- toc()
```

```
ggplot(train_nnet1, highlight = TRUE)
```

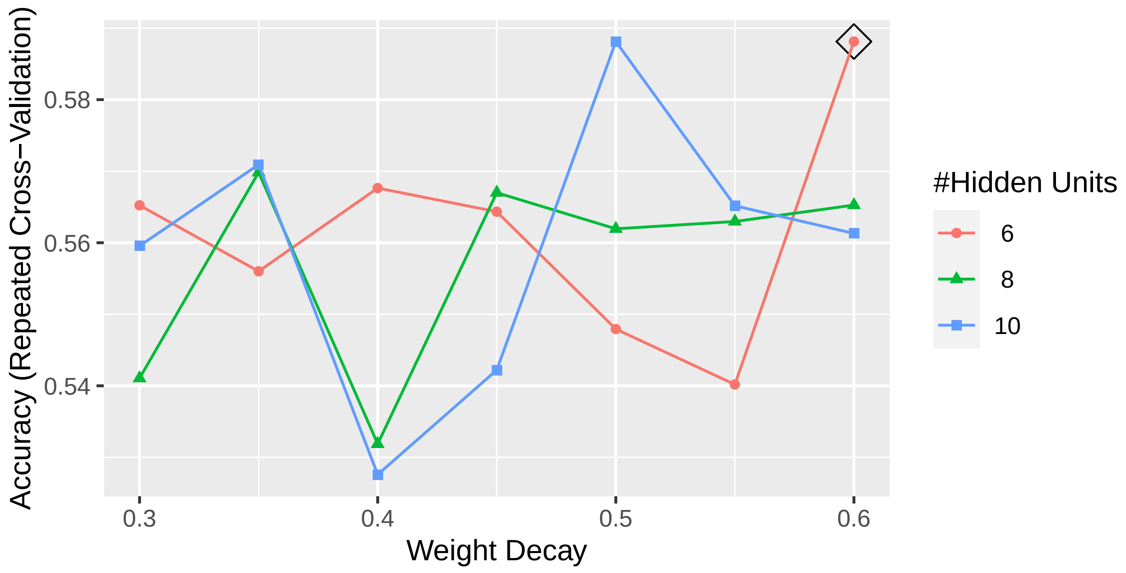


We see that the best results are for size=8 and decay=0.33, which provides an accuracy on train_set of 0.577. Looks like we are improving!

Now we will run again the code but focusing on sizes from 6 to 10 and decays from 0.26 to 0.42 to see if we can get a higher accuracy.

```
tic("NN2")
set.seed(2007, sample.kind = "Rounding")
control <- trainControl(method = "repeatedcv", number=3, repeats=4)
grid_nnet2 <- expand.grid(size=seq(6,10,2), decay=seq(0.3, 0.6, 0.05))
train_nnet2 <- train(units_sold ~ ., data=train_set, method="nnet", trControl=control,
                     tuneGrid=grid_nnet2)
nn2_toc <- toc()
```

```
ggplot(train_nnet2, highlight = TRUE)
```



And the best results are for size=6 and decay=0.6, providing an accuracy of 0.588. So now, we will use this train_nnet2 model to predict the results in the test set.

```
y_nnet <- predict(train_nnet2, test_set, type="raw")
acc_nnet <- confusionMatrix(y_nnet, test_set$units_sold)$overall[['Accuracy']]
acc_results <- bind_rows(acc_results,
                         data_frame(method="Neural Network",
                                    Accuracy_Train = max(train_nnet2$results$Accuracy),
                                    Accuracy_Test = acc_nnet,
                                    Time = nn2_toc$toc - nn2_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 11: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645

And we see an improvement as well! Now we are over 0.59 of accuracy on the test set. It is important to note how slow this method is, mainly due to the repeated cross-validation being performed.

3.4. Classification Trees

In this section we will use the classification trees model. This is a very intuitive model that is highly used for categorical problems.

The idea is to create a decision tree, with cut points so that, when applied in a hierarchical way, lead to the final output.

`Caret` package allows us to optimize the *cp* parameter via tuneGrid. *cp* is the complexity parameter of the tree and it defaults to 0. It helps decide the depth of the tree by not continuing building the tree if the improvement does not meet the criteria.

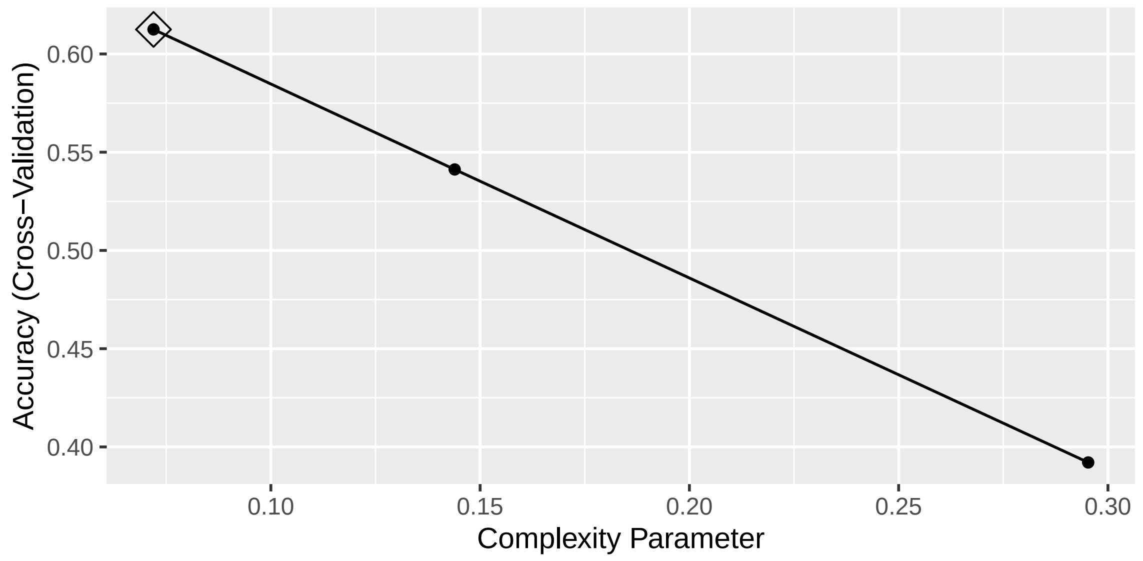
Firstly, we will use default values to train the model, with regular cross-validation for computer optimization purposes. Also we need to change the levels name of the *units_sold* parameter for the function to not throw and error message.

```
levels(train_set$units_sold) <- c("X10", "X50", "X100", "X1000", "X5000", "X10000",
                                   "X20000", "X50000", "X05")
levels(test_set$units_sold) <- c("X10", "X50", "X100", "X1000", "X5000", "X10000",
                                  "X20000", "X50000", "X05")

tic()
set.seed(2007, sample.kind = "Rounding")
control <- trainControl(method = "cv", number=4, classProbs = TRUE)
train_rpart0 <- train(units_sold ~ ., data=train_set, method="rpart", trControl=control)
rp0_toc <- toc()
```

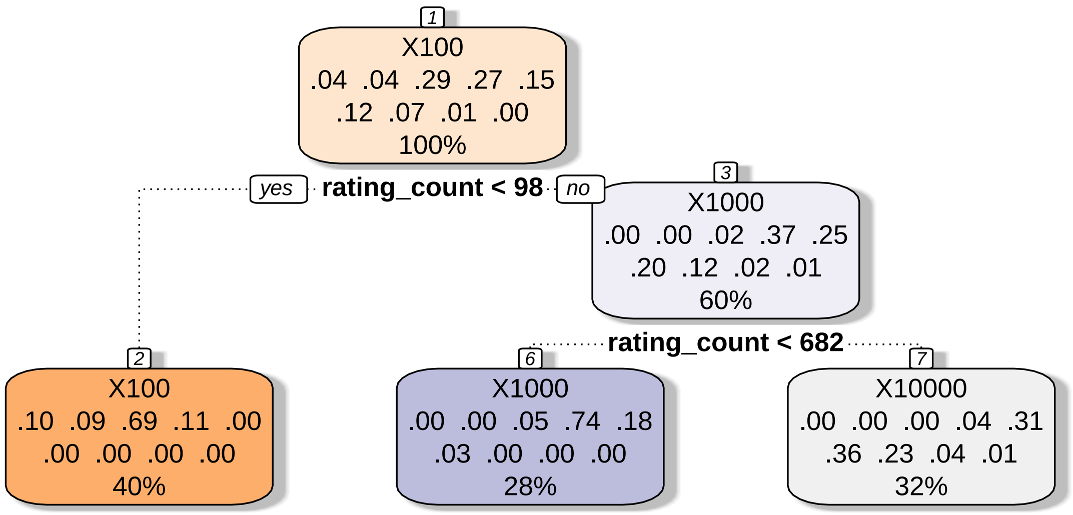
```
## 1.439 sec elapsed
```

```
ggplot(train_rpart0, highlight = TRUE)
```

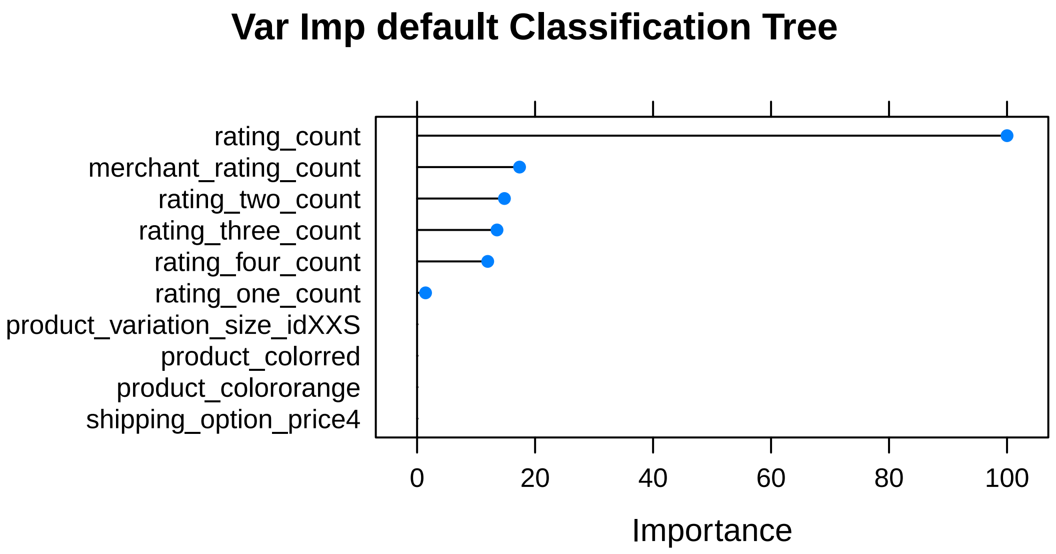


We see that the optimal value for cp in this case is for cp=0.07196, which provides an accuracy of 0.612. To visualize the tree, we will use the fancyRpartPlot from `rattle` library. We can see that it is actually quite simple, and that only one feature, *rating_count*, is used to make the predictions! In fact, it does not even predict all the possible levels. Also, we can see the variable importance.

```r
fancyRpartPlot(train_rpart0$finalModel, sub = NULL)
```



```r
rpart0_imp <- varImp(train_rpart0)
plot(rpart0_imp, top = 10, main="Var Imp default Classification Tree")
```



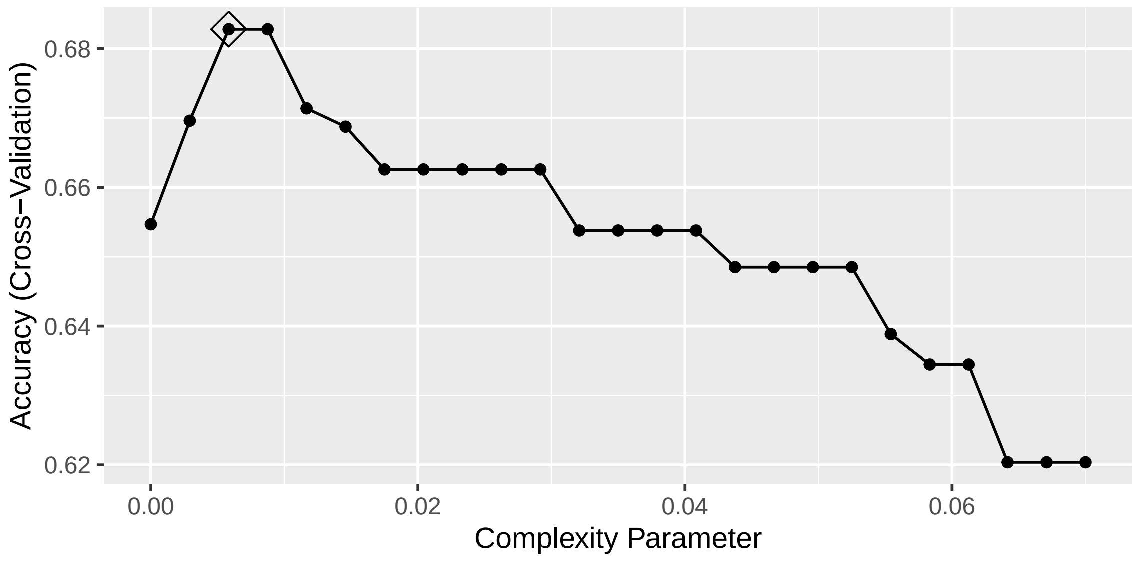
For the purpose of predicting using the model and be able to compare results, we will use it to predict and get accuracy on test set.

Table 12: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645
Classification Trees not optimised	0.6124815	0.6206897	1.439

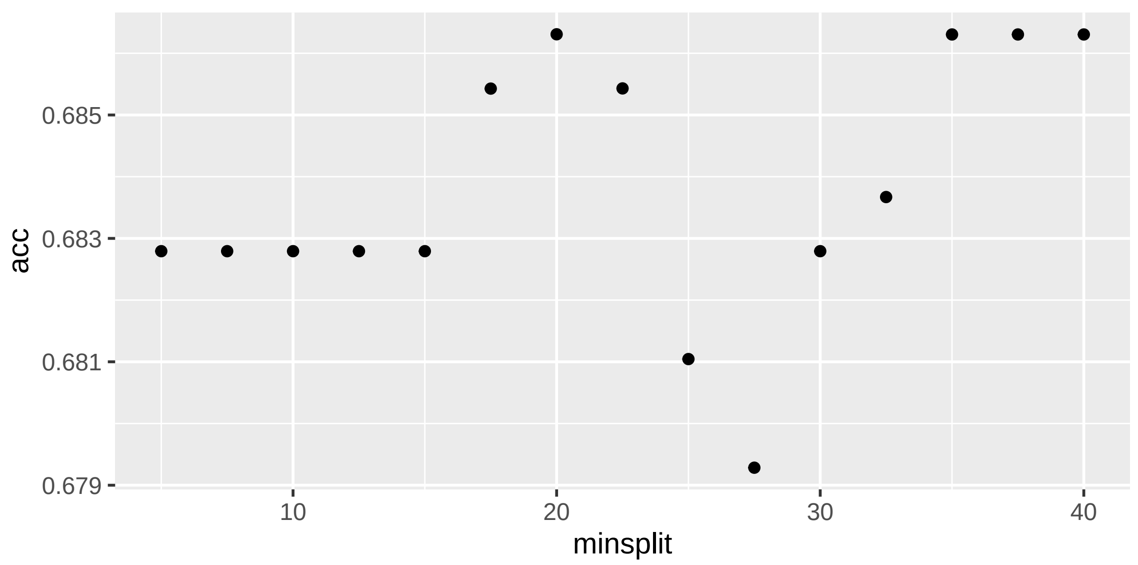
Now we will use tuneGrid to try different values of cp that range from 0 to 0.07, as it seems to be the area with the higher accuracy values. *Minsplit* will be held constant at 15.

```r
set.seed(2007, sample.kind = "Rounding")
control1 <- trainControl(method = "cv", number=4, classProbs = TRUE)
train_rpart1 <- train(units_sold ~ ., data=train_set, method="rpart",
                      tuneGrid = data.frame(cp = seq(0, 0.07, len = 25)),
                      control=rpart::rpart.control(minsplit=15), trControl=control1)
ggplot(train_rpart1, highlight = TRUE)
```



We now get a higher accuracy of 0.682 with a cp=0.0058, mainly due to the variation of the *minsplit* parameter. Now that we have optimized cp, we will try different minsplit numbers to optimize the model.

```r
cp <- train_rpart1$bestTune$cp
minsplit <- seq(5, 40, len=15)
acc <- sapply(minsplit, function(ms){
  set.seed(2007, sample.kind = "Rounding")
  control1 <- trainControl(method = "cv", number=4, classProbs = TRUE)
  train(units_sold ~ ., method = "rpart", data = train_set, tuneGrid = data.frame(cp=cp),
        control=rpart::rpart.control(minsplit=ms), trControl=control1)$results$Accuracy })
qplot(minsplit, acc)
```



```r
minsplit[which.max(acc)]
```

```
## [1] 20
```

```r
max(acc)
```

```
## [1] 0.6863078
```

```r
minsplit <- minsplit[which.max(acc)]
```

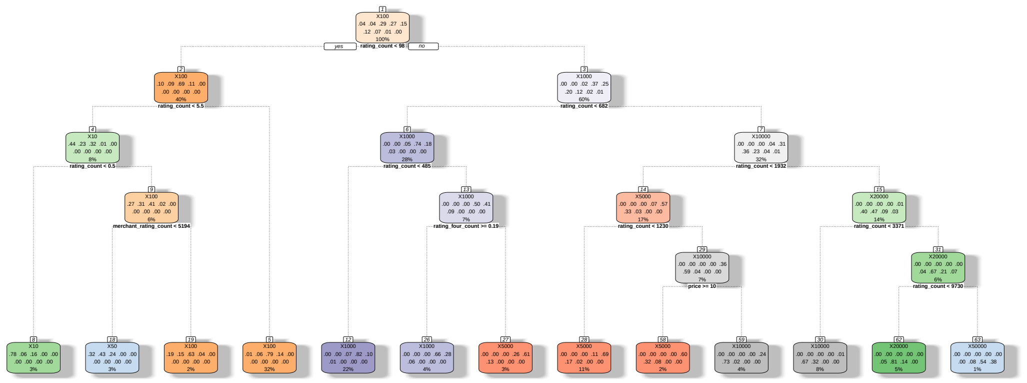
Great, for a minsplit=20 of we get an accuracy of 0.686. Let's now train a new model with the optimized cp and minsplit values and plot the new tree.

We see now that the tree gets more decision points, and also that more predictors are taken into consideration, such as *price*, *merchant_rating_count* and *rating_four_count*.

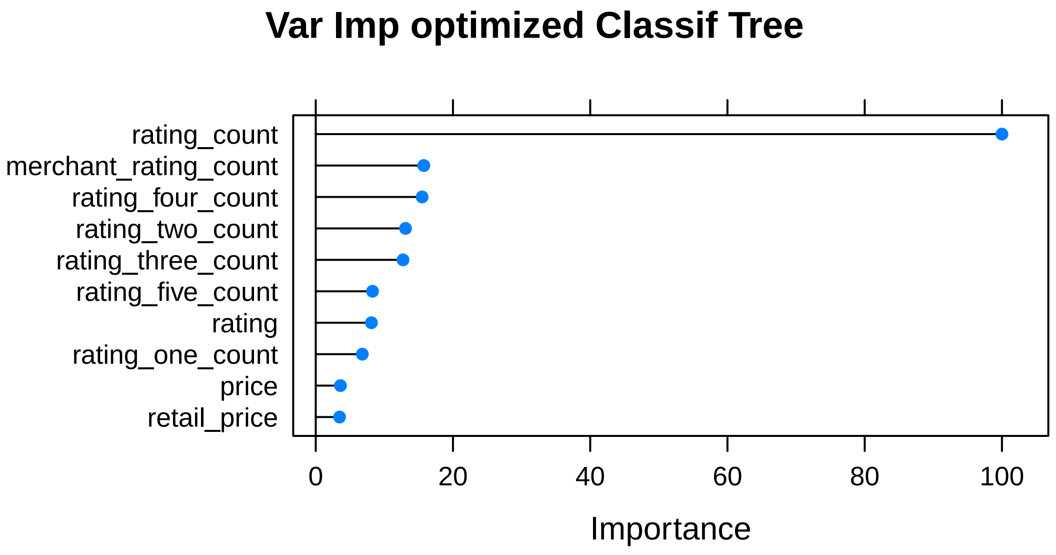
```r
tic("rpart")
set.seed(2007, sample.kind = "Rounding")
control1 <- trainControl(method = "cv", number=4, classProbs = TRUE)
train_rpart2 <- train(units_sold ~ ., data=train_set, method="rpart",
                      tuneGrid = data.frame(cp = cp),
                      control=rpart::rpart.control(minsplit=minsplit),
                      trControl=control1)
rp2_toc <- toc()
```

```
## rpart: 1.097 sec elapsed
```

```r
fancyRpartPlot(train_rpart2$finalModel, sub = NULL)
```



```r
rpart2_imp <- varImp(train_rpart2)
plot(rpart2_imp, top = 10, main="Var Imp optimized Classif Tree")
```



The accuracy we get with this model in the train set is 0.686, similar to the one in the previous tree model, so this is the one we will use for predictions.

```r
y_rpart2 <- predict(train_rpart2, test_set, type="raw")
acc_rpart2 <- confusionMatrix(y_rpart2, test_set$units_sold)$overall[['Accuracy']]
acc_results <- bind_rows(acc_results,
                         data_frame(method="Classification Trees Optimized",
                                    Accuracy_Train = max(train_rpart2$results$Accuracy),
                                    Accuracy_Test = acc_rpart2,
                                    Time = rp2_toc$toc-rp2_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 13: Accuracy Results

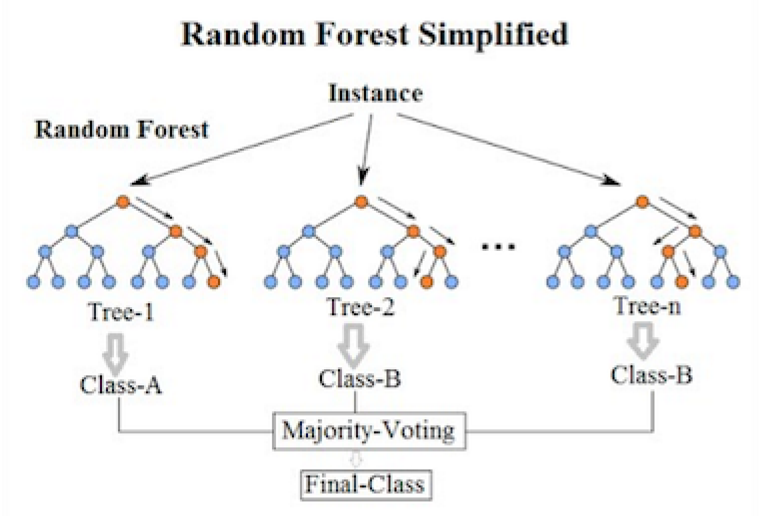
method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645
Classification Trees not optimised	0.6124815	0.6206897	1.439
Classification Trees Optimized	0.6863078	0.7487685	1.097

Great! We are almost on a 75% of correct predictions.

3.5. Random Forest

Random Forest is a widely used algorithm, either for Classification problems (as this one), but also for continuous problems, in that case, it is called Regression Forest. Random Forest uses the technique from Decision Trees that we studied in the section above, but using a large number of different decision trees. Each decision tree in the forest provides an output, the class that was outputed the most of the times by the decision trees, is the one that the algorithm chooses.

So this model usually outperforms the decision tree model, because taking into account different trees to make the decision, disregards errors that single trees may produce.



By Venkata Jagannath - : https://community.tibco.com/wiki/random-forest-template-tibco-spotfirer-wiki-page, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=68995764*

`Caret` package allows us to optimize the *mtry* parameter via tuneGrid. *Mtry* parameter refers to the number of variables available for splitting at each tree node. A common rule to choose *mtry* in classification problems is to $mtry = \sqrt(number of predictors)$.

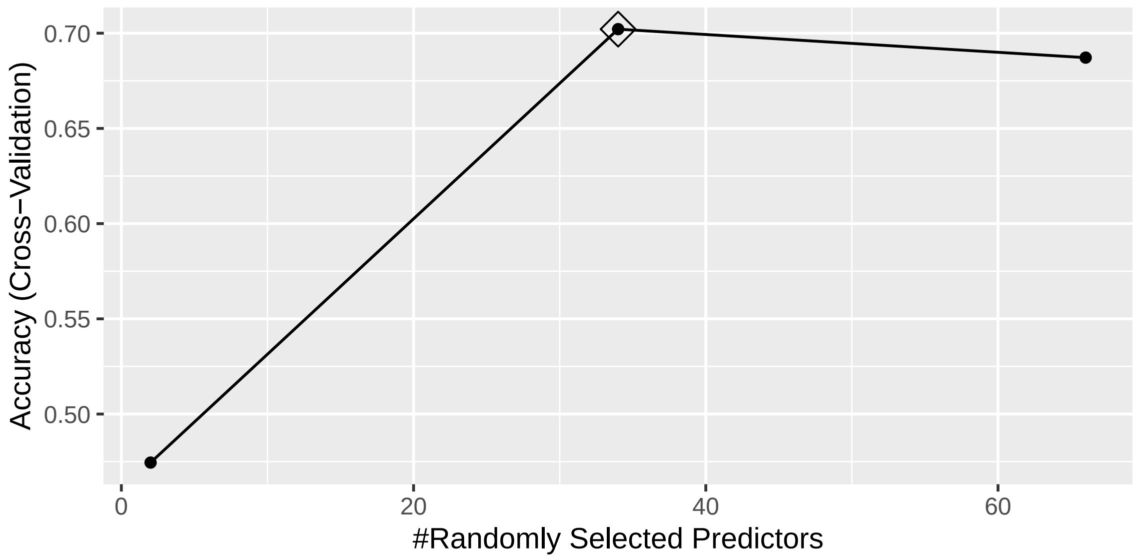
We will start to run a model with the default values from `caret` package. As in Random Forest we already use independent trees in the model, we will use simple cross-validation in this case.

```
tic("default RF")
set.seed(1234, sample.kind = "Rounding")
control_rf <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                           verboseIter = FALSE)
```

```r
train_rf0 <- train(units_sold ~ ., data=train_set, method="rf", trControl=control_rf)
rf0_toc <- toc()
```

```
## default RF: 22.806 sec elapsed
```

```r
ggplot(train_rf0, highlight = TRUE)
```



We see that the best accuracy of 0.702 we get is for a mtry=34 . As we said before, in the case best mtry>number of predictors in our dataset. This is because `caret::train` function converts the database into a matrix, and considers, as an independent predictor, each level of the categorical variables.

```r
y_rf0 <- predict(train_rf0, test_set, type="raw")
acc_rf0 <- confusionMatrix(y_rf0, test_set$units_sold)$overall[['Accuracy']]
acc_results <- bind_rows(acc_results,
                         data_frame(method="Random Forest not optimized",
                                    Accuracy_Train = max(train_rf0$results$Accuracy),
                                    Accuracy_Test = acc_rf0,
                                    Time = rf0_toc$toc-rf0_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 14: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645
Classification Trees not optimised	0.6124815	0.6206897	1.439
Classification Trees Optimized	0.6863078	0.7487685	1.097
Random Forest not optimized	0.7021243	0.7684729	22.806

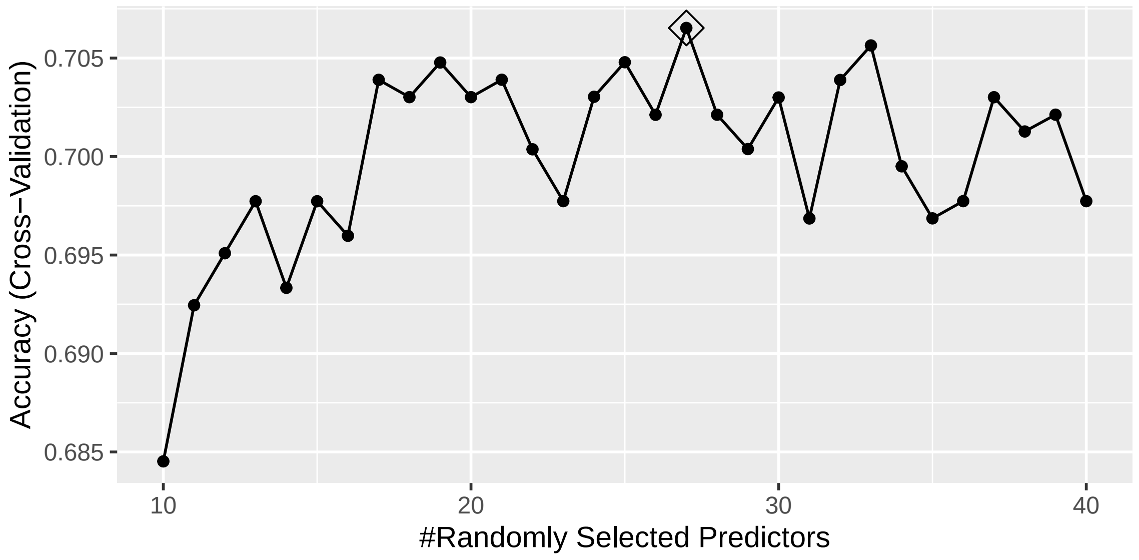
And we get an accuracy on test set of 0.76! Great!.

Now we will try to optimize the model. As we have 28 predictors in our database, we will try *mtry* that range from 20 to 40.

```
tic("mtry optimized RF")
set.seed(1234, sample.kind = "Rounding")
control_rf <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                           verboseIter = FALSE)
grid_rf <- expand.grid(mtry=seq(10,40,1))
train_rf1 <- train(units_sold ~ ., data=train_set, method="rf", tuneGrid=grid_rf,
                   trControl=control_rf)
rf1_toc <- toc()
```

```
## mtry optimized RF: 199.394 sec elapsed
```

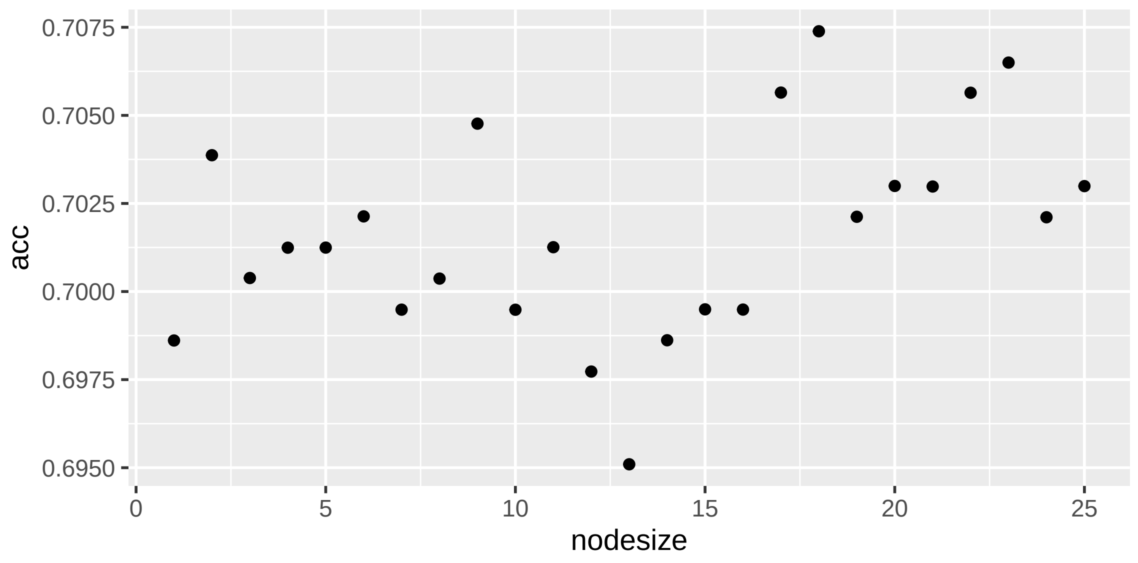
```
ggplot(train_rf1, highlight = TRUE)
```



```
mtry <- train_rf1$bestTune$mtry
```

And now we get a higher accuracy of 0.706 for a mtry=27. Once that we have optimized the *mtry* parameter, we will try with different nodesizes and see if we can get any improvement.

```
grid_mtry <- expand.grid(mtry=mtry)
nodesize <- seq(1, 25, 1)
acc <- sapply(nodesize, function(ns){
  set.seed(1234, sample.kind = "Rounding")
  control_rf <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                             verboseIter = FALSE)
  train(units_sold ~ ., method = "rf", data = train_set, tuneGrid = grid_mtry,
        trControl=control_rf,
        nodesize = ns)$results$Accuracy })
qplot(nodesize, acc)
```



```
nodesize <- nodesize[which.max(acc)]
max(acc)
```

```
## [1] 0.7073875
```

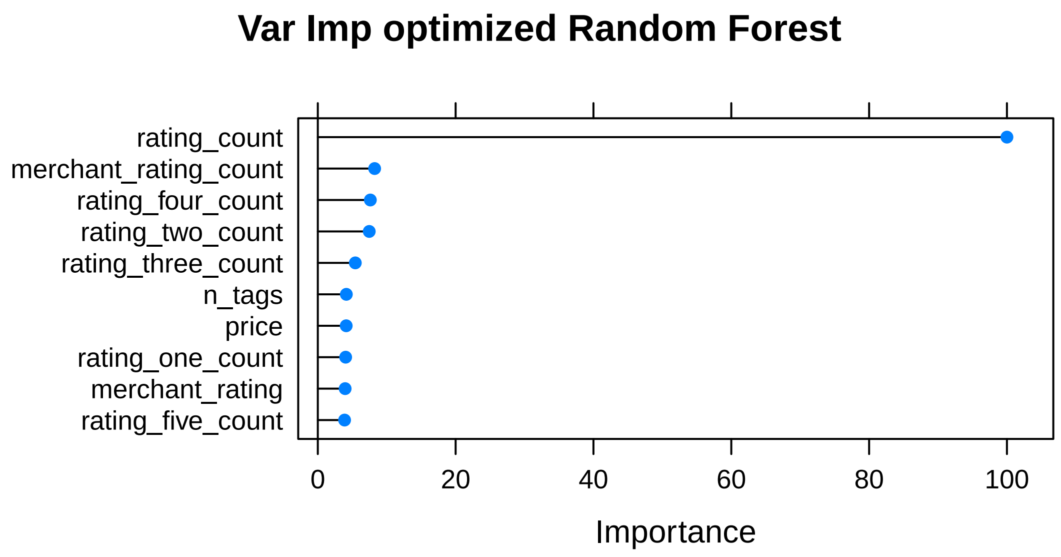
Great! So the optimal nodesize=18 which gives an accuracy of 0.707. It is important to note now that we are not able to beat the previous accuracy of 0.707. Now, keeping our optimal nodesize=18 and mtry=27, we will train the model.

```
tic("Optimized RF")
set.seed(1234, sample.kind = "Rounding")
control_rf <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                           verboseIter = FALSE)
train_rf2 <- train(units_sold ~ ., method = "rf", data = train_set,
                   tuneGrid = grid_mtry, nodesize = nodesize, trControl=control_rf)
rf2_toc <- toc()
```

```
## Optimized RF: 8.919 sec elapsed
```

As we said the accuracy is higher than in model *train_rf1*, so we will use *train_rf2* to test results in test set. First, we plot the variable importance of the model.

```
rf2_imp <- varImp(train_rf2)
plot(rf2_imp, top = 10, main="Var Imp optimized Random Forest")
```



```
y_rf <- predict(train_rf2, test_set, type="raw")
acc_rf <- confusionMatrix(y_rf, test_set$units_sold)$overall[['Accuracy']]
acc_results <- bind_rows(acc_results,
                         data_frame(method="Random Forest optimized",
                                    Accuracy_Train = max(train_rf2$results$Accuracy),
                                    Accuracy_Test = acc_rf,
                                    Time = rf2_toc$toc-rf2_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 15: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645
Classification Trees not optimised	0.6124815	0.6206897	1.439
Classification Trees Optimized	0.6863078	0.7487685	1.097
Random Forest not optimized	0.7021243	0.7684729	22.806
Random Forest optimized	0.7073875	0.7536946	8.919

Looks like we did not improve the results compared to the default method. Although on train_set it looked like we did, when applying the model on test set, we got a lower accuracy than with mtry=34. Thus, in this case, the best model is the default.

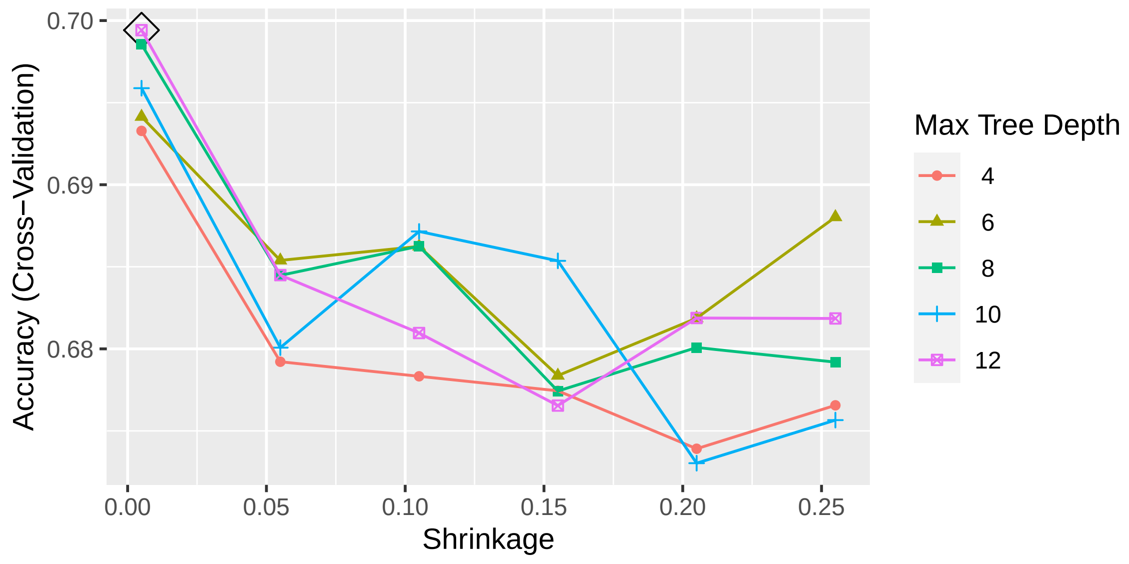
3.6. XGBoost

XGBoost is another model we have not studied during the course, but that is becoming very popular due to its execution speed and model performance. Also, it dominates structured datasets on classification problems. It uses the gradient boosting decision tree algorithm, which is an ensemble technique where new models are added to correct the errors made by the existing models. It keeps adding models until no improvement is achieved.

We will start by optimizing the tuneGrid hyperparameters of *eta* and *maxdepth*. To start the first model, we will keep the *nodesize*=5, which is a standard value for multinomial categorical problems.

We will vary *eta* from 0.005 to 0.3, which controls the learning rate. And *max_depth* from 4 to 12, which controls the maximum depth of the trees. If we make *max_depth* too large, we have higher chances of over-fitting the model.

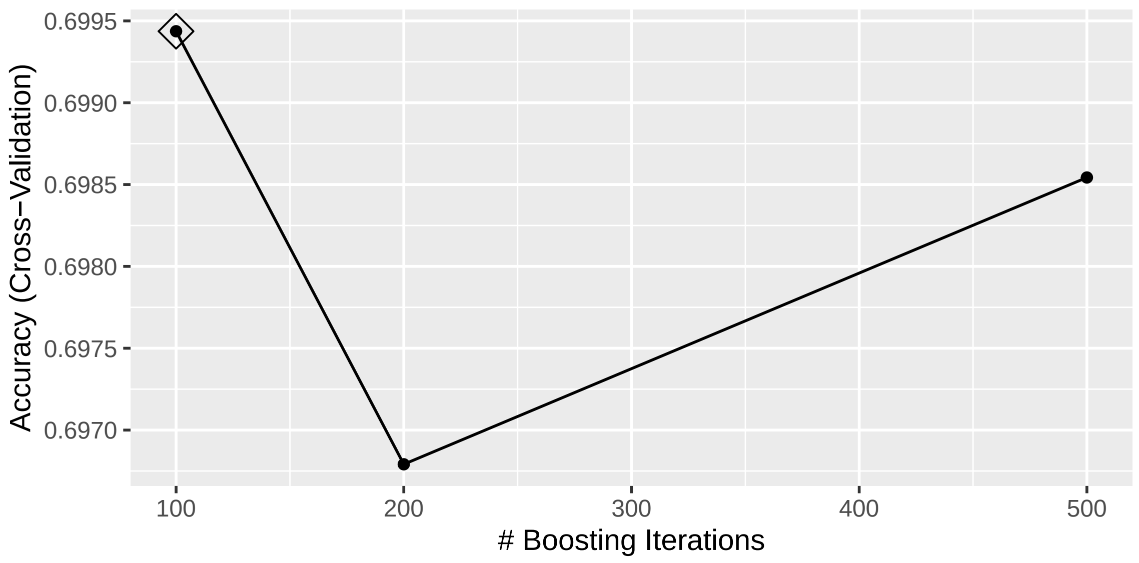
```
grid_xgbm1 <- expand.grid(min_child_weight=c(5), eta=seq(0.005, 0.3, 0.05),
                          nrounds=c(500), max_depth=seq(4,12,2), gamma=0,
                         colsample_bytree=c(0.8), subsample=1)
set.seed(62, sample.kind = "Rounding")
control_xgbm <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                             verboseIter = FALSE)
train_xgbm1 <- train(units_sold ~ ., method="xgbTree", data=train_set,
                     trControl=control_xgbm, tuneGrid=grid_xgbm1, verbose=TRUE)
ggplot(train_xgbm1, highlight = TRUE)
```



```
eta <- train_xgbm1$bestTune$eta
max_depth <- train_xgbm1$bestTune$max_depth
```

Once we have our optimized values of *eta*=0.005 and *max_depth*=12, giving an accuracy of 0.699, we will fix those values and try to optimize the *nrounds* hyperparameter. *nrounds* controls the maximum number of iterations.

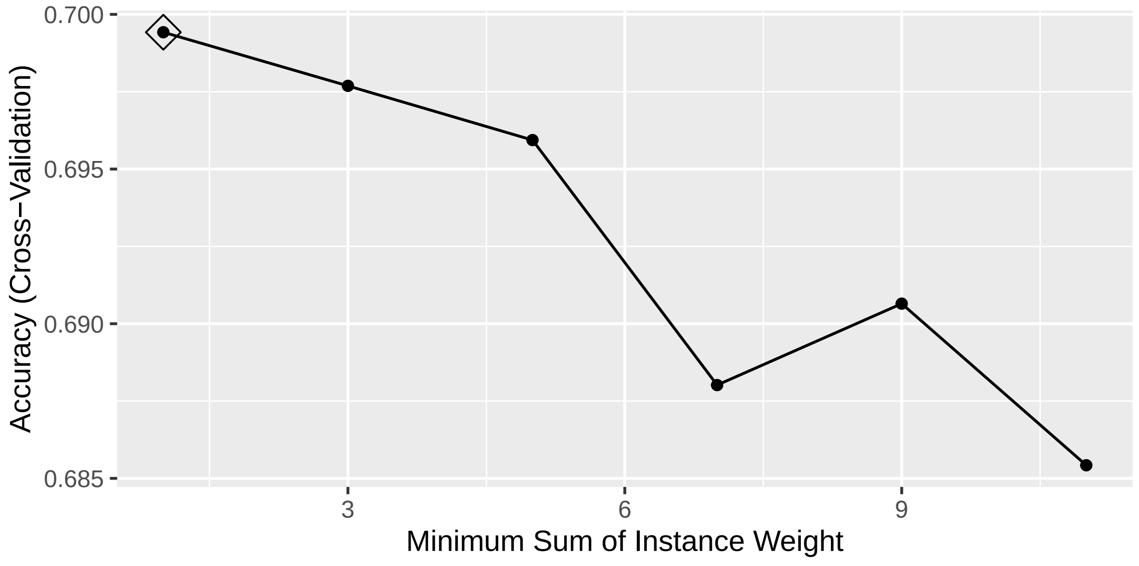
```
grid_xgbm2 <- expand.grid(min_child_weight=c(5), eta=c(eta),
                          nrounds=c(100,200,500), max_depth=c(max_depth),
                          gamma=0, colsample_bytree=c(0.8), subsample=1)
set.seed(62, sample.kind = "Rounding")
control_xgbm <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                             verboseIter = FALSE)
train_xgbm2 <- train(units_sold ~ ., method="xgbTree", data=train_set,
                     trControl=control_xgbm, tuneGrid=grid_xgbm2, verbose=TRUE)
ggplot(train_xgbm2, highlight = TRUE)
```



```
nrounds <- train_xgbm2$bestTune$nrounds
```

We need to keep in mind that the higher the *nrounds* number, longer the time the model needs to run. Now that we have also optimized the *nrounds* parameter to equal 100, which provides an accuracy of 0.699, we will try to improve the model by looking at different *min_child_weight* values. It means the same as *nodesize* in Random Forest.

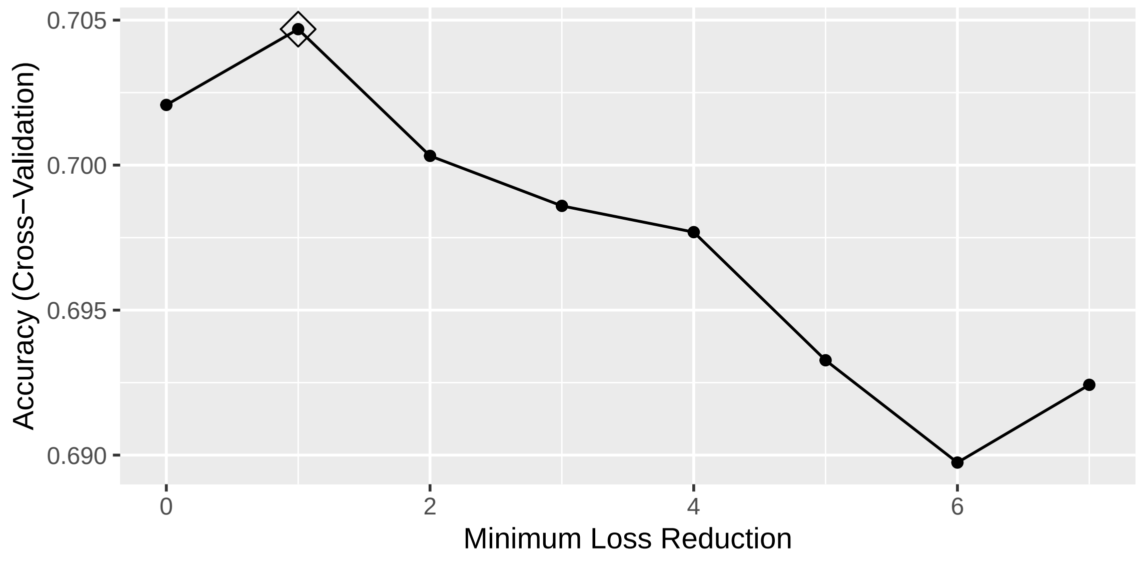
```
grid_xgbm3 <- expand.grid(min_child_weight=c(1,3,5,7,9,11), eta=c(eta), nrounds=c(nrounds),
                          max_depth=c(max_depth), gamma=0,
                         colsample_bytree=c(0.8), subsample=1)
set.seed(62, sample.kind = "Rounding")
control_xgbm <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                             verboseIter = FALSE)
train_xgbm3 <- train(units_sold ~ ., method="xgbTree", data=train_set,
                     trControl=control_xgbm, tuneGrid=grid_xgbm3, verbose=TRUE)
ggplot(train_xgbm3, highlight = TRUE)
```



```
nodesize <- train_xgbm3$bestTune$min_child_weight
```

Great, we now have a new parameter optimized. With *min_child_weight*=1 we get an accuracy of 0.699. Lastly, we will see if trying a different *gamma* can improve our accuracy. *Gamma* is the regularization parameter that prevents over-fitting and allows weighting. It defaults to 0.

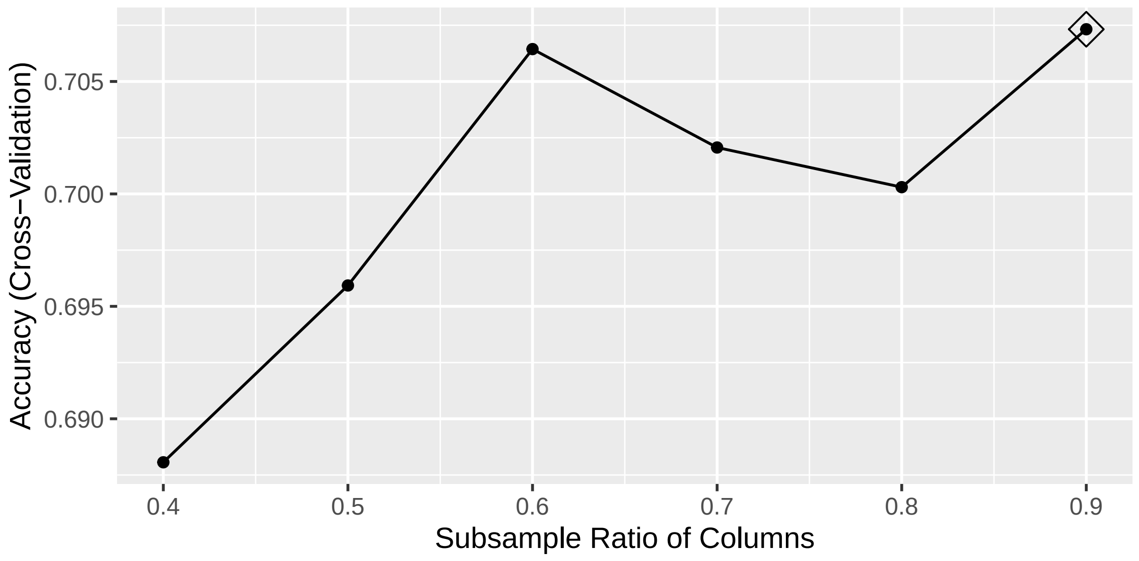
```
grid_xgbm4 <- expand.grid(min_child_weight=c(nodesize), eta=c(eta), nrounds=c(nrounds),
                          max_depth=c(max_depth), gamma=seq(0,7,1),
                          colsample_bytree=c(0.8), subsample=1)
set.seed(62, sample.kind = "Rounding")
control_xgbm <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                             verboseIter = FALSE)
train_xgbm4 <- train(units_sold ~ ., method="xgbTree", data=train_set,
                     trControl=control_xgbm, tuneGrid=grid_xgbm4, verbose=TRUE)
ggplot(train_xgbm4, highlight = TRUE)
```



```
gamma <- train_xgbm4$bestTune$gamma
```

We can see that we get the higher accuracy of 0.704 for a gamma=1. Once that we have optimized all these parameters, we try to optimized the *colsample_bytree* parameter. It is the fraction of features that will be used to train each tree, similar to *mtry* in Random Forest.

```
grid_xgbm5 <- expand.grid(min_child_weight=c(nodesize), eta=c(eta),
                          nrounds=c(nrounds), max_depth=c(max_depth), gamma=gamma,
                          colsample_bytree=seq(0.4, 0.9, 0.1), subsample=1)
set.seed(62, sample.kind = "Rounding")
control_xgbm <- trainControl(method = "cv", number=3, savePredictions = FALSE, verboseIter = FALSE)
train_xgbm5 <- train(units_sold ~ ., method="xgbTree", data=train_set,
                     trControl=control_xgbm, tuneGrid=grid_xgbm5, verbose=TRUE)
ggplot(train_xgbm5, highlight = TRUE)
```



```
colsample_bytree <- train_xgbm5$bestTune$colsample_bytree
```

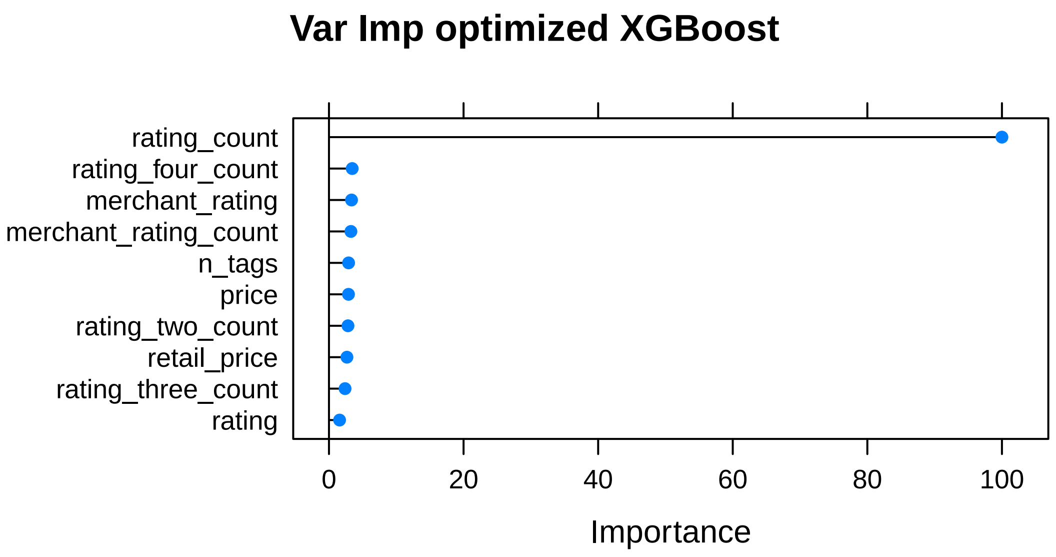
We see that we get the optimal *colsample_bytree* at 0.9 with an accuracy of 0.707.

Once we have optimized all these parameters, we run the optimized model. By using the varImp function, we can plot as well the importance that the different predictors have in the model.

```
tic("Optimized XGBoost")
grid_xgbm_op <- expand.grid(min_child_weight=c(nodesize), eta=c(eta), nrounds=c(nrounds),
                            max_depth=c(max_depth), gamma=gamma,
                            colsample_bytree=c(colsample_bytree), subsample=1)
set.seed(62, sample.kind = "Rounding")
control_xgbm <- trainControl(method = "cv", number=3, savePredictions = FALSE,
                             verboseIter = FALSE)
train_xgbm_op <- train(units_sold ~ ., method="xgbTree", data=train_set,
                       tuneGrid=grid_xgbm_op, trControl=control_xgbm, verbose=TRUE)
xgbm_toc <- toc()
```

```
## Optimized XGBoost: 17.516 sec elapsed
```

```
xgbm_imp <- varImp(train_xgbm_op)
plot(xgbm_imp, top = 10, main="Var Imp optimized XGBoost")
```



We use our optimized model to test the results and compare with the previous models.

```
y_xgbm <- predict(train_xgbm_op, test_set, type="raw")
acc_xgbm <- confusionMatrix(y_xgbm, test_set$units_sold)$overall[['Accuracy']]
acc_results <- bind_rows(acc_results,
                         data_frame(method="XGBoost",
                                    Accuracy_Train = max(train_xgbm_op$results$Accuracy),
                                    Accuracy_Test = acc_xgbm,
                                    Time=xgbm_toc$toc-xgbm_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 16: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645
Classification Trees not optimised	0.6124815	0.6206897	1.439
Classification Trees Optimized	0.6863078	0.7487685	1.097
Random Forest not optimized	0.7021243	0.7684729	22.806
Random Forest optimized	0.7073875	0.7536946	8.919
XGBoost	0.7038250	0.7487685	17.516

With our final XGBoost model we were not able to outperform the default Random Forest model when running on test set and results were very similar to the ones we got with Random Forest optimized. Time for XGBoost was also a little bit lower than for the default Random Forest. In fact, XGBoost gets the same accuracy than the optimized Classification Tree model, but in almost x10 time!.

3.7. H2O AutoML

For our last model we will use the library "h2o" and its auto Machine Learning algorithm. This algorithms runs different models (Random Forest, XGBoost, Neural Networks, etc) and specially, it also does Stacked Ensembles. Stacked Ensemble method uses more than one ML algorithm to improve performance.

```r
library(h2o)

h2o.init()
data_h2o <- as.h2o(train_set)
test_h2o <- as.h2o(test_set)
tic("h2oautoml")
automl_all <- h2o.automl(y=3, training_frame=data_h2o, max_runtime_secs=500,
                         validation_frame = test_h2o,
                            seed=1, keep_cross_validation_predictions=TRUE)
h2o_toc <- toc()
```

```r
automl_all_lb <- head(automl_all@leaderboard)
knitr::kable(automl_all_lb, caption = "Best h2o AutoML performer model")
```

Table 17: Best h2o AutoML performer model

model_id	mean_per_class_error	logloss	rmse	mse
GBM_grid___1_AutoML_20201026_120242_model_2	0.4567085	0.8788940	0.5068906	0.2569381
GBM_3_AutoML_20201026_120242	0.4600545	0.8318316	0.5055218	0.2555523
XGBoost_grid___1_AutoML_20201026_120242_model_4	0.4654483	0.8880129	0.4975908	0.2475966
XGBoost_grid___1_AutoML_20201026_120242_model_1	0.4686344	0.8777661	0.4956094	0.2456287
XGBoost_grid___1_AutoML_20201026_120242_model_7	0.4692609	0.8144873	0.4919901	0.2420543
XGBoost_grid___1_AutoML_20201026_120242_model_10	0.4739027	0.8264232	0.4910372	0.2411175

In the table above, we can see the performance of the different models that were trained using h2o. We see that the best performer is a Gradient Boosting model that we did not study in the course neither was used in the previous sections.

If we study the best performer model, we can see the different features it used, which were optimized by using cross-validation.

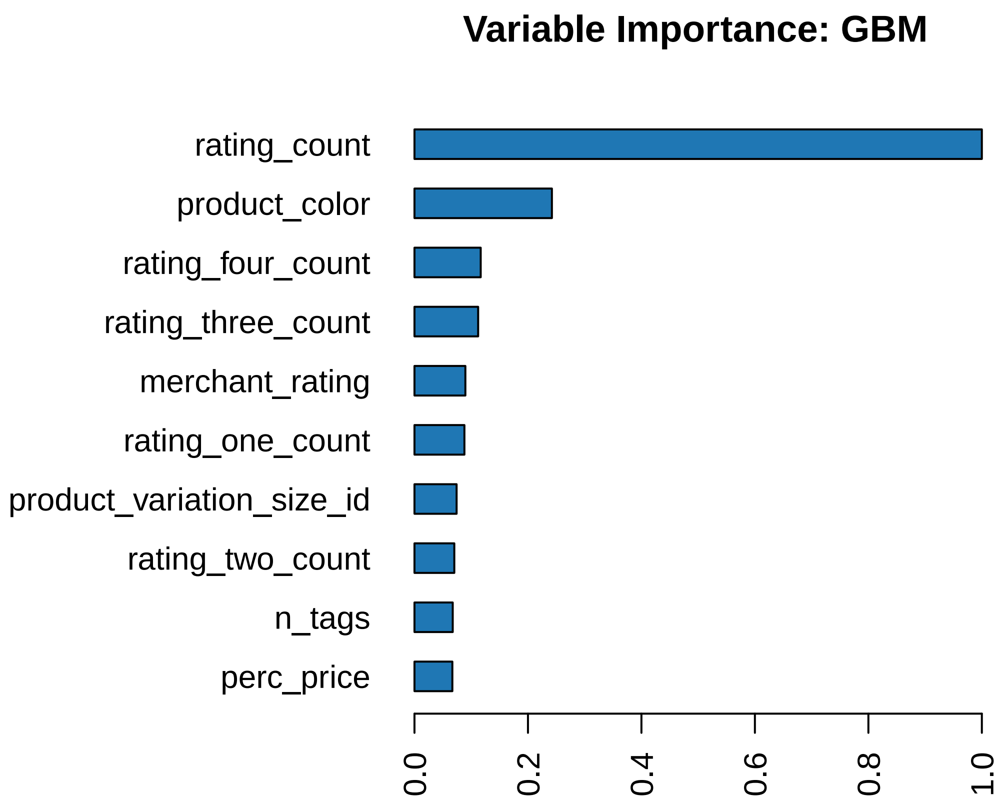
```r
automl_all@leader
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: gbm
## Model ID:  GBM_grid__1_AutoML_20201026_120242_model_2
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1              41                      369              242757         4
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1         8    7.97290          6         86    47.61247
##
##
## H2OMultinomialMetrics: gbm
## ** Reported on training data. **
##
## Training Set Metrics:
## =====================
##
```

```
## Extract training frame with `h2o.getFrame("automl_training_train_set_sid_a865_1")`
## MSE: (Extract with `h2o.mse`) 0.005945891
## RMSE: (Extract with `h2o.rmse`) 0.07710961
## Logloss: (Extract with `h2o.logloss`) 0.06022962
## Mean Per-Class Error: 0
## R^2: (Extract with `h2o.r2`) 0.9983111
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##        X05 X10 X100 X1000 X10000 X20000 X50 X5000 X50000  Error        Rate
## X05      5   0    0     0      0      0   0     0      0 0.0000 =     0 / 5
## X10      0  44    0     0      0      0   0     0      0 0.0000 =    0 / 44
## X100     0   0  332     0      0      0   0     0      0 0.0000 =   0 / 332
## X1000    0   0    0   307      0      0   0     0      0 0.0000 =   0 / 307
## X10000   0   0    0     0    139      0   0     0      0 0.0000 =   0 / 139
## X20000   0   0    0     0      0     83   0     0      0 0.0000 =    0 / 83
## X50      0   0    0     0      0      0  43     0      0 0.0000 =    0 / 43
## X5000    0   0    0     0      0      0   0   170      0 0.0000 =   0 / 170
## X50000   0   0    0     0      0      0   0     0     15 0.0000 =    0 / 15
## Totals   5  44  332   307    139     83  43   170     15 0.0000 = 0 / 1.138
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-9 Hit Ratios:
##   k hit_ratio
## 1 1  1.000000
## 2 2  1.000000
## 3 3  1.000000
## 4 4  1.000000
## 5 5  1.000000
## 6 6  1.000000
## 7 7  1.000000
## 8 8  1.000000
## 9 9  1.000000
##
##
## H2OMultinomialMetrics: gbm
## ** Reported on validation data. **
##
## Validation Set Metrics:
## =====================
##
## Extract validation frame with `h2o.getFrame("test_set_sid_a865_3")`
## MSE: (Extract with `h2o.mse`) 0.2227816
## RMSE: (Extract with `h2o.rmse`) 0.4719975
## Logloss: (Extract with `h2o.logloss`) 0.7451385
## Mean Per-Class Error: 0.4139084
## R^2: (Extract with `h2o.r2`) 0.9338983
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##        X05 X10 X100 X1000 X10000 X20000 X50 X5000 X50000  Error       Rate
## X05      1   0    0     0      0      0   0     0      0 0.0000 =    0 / 1
## X10      0   2    1     0      0      0   1     0      0 0.5000 =    2 / 4
## X100     0   1   57     5      0      0   1     0      0 0.1094 =   7 / 64
## X1000    0   0    7    43      0      0   0     5      0 0.2182 =  12 / 55
## X10000   0   0    0     1     12      4   0     7      0 0.5000 =  12 / 24
## X20000   0   0    0     0      2     12   0     0      2 0.2500 =   4 / 16
## X50      0   2    3     0      0      0   2     0      0 0.7143 =   5 / 7
## X5000    0   0    0     6      7      0   0    17      0 0.4333 =  13 / 30
## X50000   1   0    0     0      0      1   0     0      0 1.0000 =   2 / 2
## Totals   2   5   68    55     21     17   4    29      2 0.2808 = 57 / 203
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
## =======================================================================
## Top-9 Hit Ratios:
##   k hit_ratio
## 1 1  0.719212
## 2 2  0.945813
## 3 3  0.985222
## 4 4  1.000000
## 5 5  1.000000
## 6 6  1.000000
## 7 7  1.000000
## 8 8  1.000000
## 9 9  1.000000
##
##
## H2OMultinomialMetrics: gbm
## ** Reported on cross-validation data. **
## ** 5-fold cross-validation on training data (Metrics computed for combined holdout predictions) **
##
## Cross-Validation Set Metrics:
## =====================
##
## Extract cross-validation frame with `h2o.getFrame("automl_training_train_set_sid_a865_1")`
## MSE: (Extract with `h2o.mse`) 0.2569381
## RMSE: (Extract with `h2o.rmse`) 0.5068906
## Logloss: (Extract with `h2o.logloss`) 0.878894
## Mean Per-Class Error: 0.4567085
## R^2: (Extract with `h2o.r2`) 0.9270184
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,xval = TRUE)`
## =======================================================================
## Top-9 Hit Ratios:
##   k hit_ratio
## 1 1  0.694200
## 2 2  0.931459
## 3 3  0.972759
## 4 4  0.988576
## 5 5  0.993849
## 6 6  0.997364
## 7 7  0.999121
## 8 8  1.000000
## 9 9  1.000000
##
##
## Cross-Validation Metrics Summary:
##                               mean          sd cv_1_valid cv_2_valid cv_3_valid
## accuracy                0.69418424 0.047234684  0.6403509 0.76754385 0.69298244
## err                     0.30581576 0.047234684 0.35964912 0.23245615 0.30701753
## err_count                     69.6   10.737783       82.0       53.0       70.0
## logloss                 0.87887454  0.09953173  1.0045577 0.74186826  0.9123204
## max_per_class_error      0.9483333  0.07080882  0.8666667        1.0        1.0
## mean_per_class_accuracy  0.5914773 0.085180975 0.67937976  0.6485122  0.5003914
## mean_per_class_error    0.40852275 0.085180975 0.32062024 0.35148782 0.49960858
## mse                     0.25694156  0.03684781 0.30299324 0.20554134 0.25829542
## r2                       0.9266415 0.012255214  0.9075066  0.9415578   0.929718
## rmse                    0.50582755 0.036743194 0.55044824  0.4533667  0.5082277
##                         cv_4_valid cv_5_valid
## accuracy                 0.7004405  0.6696035
## err                     0.29955947 0.33039647
## err_count                     68.0       75.0
## logloss                  0.8248251  0.9108012
## max_per_class_error          0.875        1.0
## mean_per_class_accuracy 0.62895924 0.50014365
## mean_per_class_error    0.37104073 0.49985635
## mse                     0.24079373  0.2770841
## r2                       0.9282127 0.92621255
## rmse                     0.4907074  0.5263878
```

In the cross validation of the training data, we can see that it achieves an accuracy of 0.697. Also, we can study the variable importance of the model, and check if it matches the variable importance of the XGBoost and Random Forest model we performed before.

```
plot <- h2o.varimp_plot(automl_all@leader, num_of_features = 10)
```



It is curious to see the importance of the *product_color* predictor in this model, while it did not even appear for XGBoost and Random Forest. Still, by far, the most important feature in all these models is *rating_count*, while the rest of variables are usually the same, although they do change in position.

Now, running the model in our test_set, we get an accuracy of (203-57)/203=0.7192.

```
##   |                                                                              |
```

```
h2o.confusionMatrix(automl_all@leader, newdata = test_h2o)
```

```
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##         X05 X10 X100 X1000 X10000 X20000 X50 X5000 X50000  Error       Rate
## X05       1   0    0     0      0      0   0     0      0 0.0000 =    0 / 1
## X10       0   2    1     0      0      0   1     0      0 0.5000 =    2 / 4
## X100      0   1   57     5      0      0   1     0      0 0.1094 =   7 / 64
## X1000     0   0    7    43      0      0   0     5      0 0.2182 =  12 / 55
## X10000    0   0    0     1     12      4   0     7      0 0.5000 =  12 / 24
## X20000    0   0    0     0      2     12   0     0      2 0.2500 =   4 / 16
## X50       0   2    3     0      0      0   2     0      0 0.7143 =    5 / 7
## X5000     0   0    0     6      7      0   0    17      0 0.4333 =  13 / 30
## X50000    1   0    0     0      0      1   0     0      0 1.0000 =    2 / 2
## Totals    2   5   68    55     21     17   4    29      2 0.2808 = 57 / 203
```

```
acc_results <- bind_rows(acc_results,
                         data_frame(method="h2oAutoML", Accuracy_Train = 0.694,
                                    Accuracy_Test = 0.7192,
                                    Time=h2o_toc$toc-h2o_toc$tic))
knitr::kable(acc_results, caption = "Accuracy Results")
```

Table 18: Accuracy Results

method	Accuracy_Train	Accuracy_Test	Time
Gam Loess	0.0524974	0.0246305	195.022
KNN	0.5035013	0.5517241	9.884
Neural Network	0.5881207	0.5960591	220.645
Classification Trees not optimised	0.6124815	0.6206897	1.439
Classification Trees Optimized	0.6863078	0.7487685	1.097
Random Forest not optimized	0.7021243	0.7684729	22.806
Random Forest optimized	0.7073875	0.7536946	8.919
XGBoost	0.7038250	0.7487685	17.516
h2oAutoML	0.6940000	0.7192000	938.908

In fact, it is only capable of improving the not optimized Classification Tree model. Personally, I think that it is always nice to check that the autoML algorithm does not beat the work done by the data scientist, and that other methods get a higher accuracy, so it is great that we were able to run a few more models that improve the results.

4. CONCLUSION

For this multinomial problem, we have just checked how some methods work better than others. In this case, winners are Random Forest and XGBoost, as they get very similar results. If we were having a larger dataset, time would be an important factor when deciding for which method to use, and the fastest method would get chosen, so for this case, Random Forest is the chosen one, once optimized hyperparameters are fixed.

The reason why I would disregard Classification Tree even being faster than XGBoost, is because the Prediction Demand problem is usually one which keeps entering new data and the database is being continuously fed. The classification tree problem is more sensitive to new data changes thus the model would be continuously changing to be improved. In Random Forest and XGBoost, as they form a "forest" of several trees, and in XGBoost the errors are minimized continuously, these two would be more stable to new and change of the data in the real world.

Future work of this project would be running a Gradient Boost Model, which was the best performer for h2o, but optimizing its hyperparameters to see if we could improve the accuracy.

As well, it would be important to dive deep into the Random Forest from `caret` package to optimize other parameters such as *ntry*, etc, and see if we can beat the default model.

In addition, it is important to keep in mind that the dataset did not contain large amount of data, thus, we can easily see accuracy deviations if running model in new datasets, reason for which for future work, we should at least try two ML algorithms (RF and XGBoost, as winners) and see which perform better.

Also, all predictors have always been considered, but sometimes, it is better to disregard those that do not contribute much to the models, which actually yields into better results. As we saw, only a few predictors actually contribute to the final results, and they are quite similar among the different models, and taking this into account in our models could improve the results and also reduce running times.

Finally, in this case, *rating_count* and *rating_star_count* were considered as predictors, as the assumption was that we wanted to predict demand for this products in the future. In the case that we would be interested in predicting demand for new products to include in the platform, as we would not have any ratings yet, these predictors should be disregarded from the model.