Capstone Project - Choose Your Own : House Prices Prediction

Arnaud RAULET

10/09/2020

Before proceeding, please install and load the packages below :

```
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(naniar)) install.packages("naniar", repos = "http://cran.us.r-project.org")
if(!require(missMDA)) install.packages("missMDA", repos = "http://cran.us.r-project.org")
if(!require(ggcorrplot)) install.packages("ggcorrplot", repos = "http://cran.us.r-project.org")
if(!require(factoextra)) install.packages("factoextra", repos = "http://cran.us.r-project.org")
if(!require(paran)) install.packages("paran", repos = "http://cran.us.r-project.org")
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
if(!require(randomForest)) install.packages("randomForest", repos = "http://cran.us.r-project.org")
if(!require(gam)) install.packages("gam", repos = "http://cran.us.r-project.org")
if(!require(mgcv)) install.packages("mgcv", repos = "http://cran.us.r-project.org")
if(!require(nlme)) install.packages("nlme", repos = "http://cran.us.r-project.org")
if(!require(xgboost)) install.packages("xgboost", repos = "http://cran.us.r-project.org")
if(!require(vtreat)) install.packages("vtreat", repos = "http://cran.us.r-project.org")
if(!require(cowplot)) install.packages("cowplot", repos = "http://cran.us.r-project.org")
if(!require(glmnet)) install.packages("glmnet", repos = "http://cran.us.r-project.org")
if(!require(magrittr)) install.packages("magrittr", repos = "http://cran.us.r-project.org")
```

Introduction

For this last project, I've chosen the House Prices dataset from kaggle.com(link : kaggle.com), which requires advanced regression techniques. Our goal is to use at least two models or algorithms that will be the most appropriated for this dataset, in order to predict house prices. As you can see on kaggle.com, this dataset describes several features of houses located in Ames, Iowa, and is also part of a contest, in which the submission with the lowest Root Mean Squared Error wins. The RMSE must be calculated between the logarithm of the predicted value and the logarithm of the observed sale price. As of today (September 10th 2020), the best RMSE submitted is 0.00044, while the one of the 50th place is equal to 0.07483. The best RMSE is really low, but I accept the challenge and we'll try to get a lower RMSE.

Before diving into the data, we will first download my GitHub repository containing the files and datasets, and save them as objects in the RStudio environment. There are two files : `train` and `test`. For clarity of purpose, we will rename the test file as `validation`, because we will use it at the end as "new" data to get the predictions with our model, and we will create another train set and a test set with the original `train` set.

```r
# Downloading the GitHub repository with the train and validation sets from GitHub
# to Rstudio environment

setwd("~") # To make sure we use a relative path

url_zip_gitrepo <- "https://github.com/a-raulet/house-prices-prediction/archive/master.zip"

download.file(url_zip_gitrepo, "house.zip")

unzip("house.zip", exdir = "~") # We choose again a relative path
```

All files will be unzipped in the following folder. We make sure R will find the data by setting the working directory with the following relative path.

```r
# Setting working directory

setwd("~/house-prices-prediction-master")
```

```r
# Train

train <- read.csv("train.csv", stringsAsFactors = FALSE)

# Validation

validation <- read.csv("test.csv", stringsAsFactors = FALSE)
```

There is a third file on the website called `data_description`, which gives details on every variable, and can also be found in the GitHub repository we have just downloaded or here : https://github.com/a-raulet/house-prices-prediction/blob/master/data_description.txt.

All files for this project (including .R, .Rmd and .pdf files) can be found at my GitHub repository here :

https://github.com/a-raulet/house-prices-prediction

Here are the key steps that were performed :

1. Exploratory Data Analysis

- Cleaning the data
- Analysing the outcome `SalePrice` and the different predictors
- Creating a correlation matrix and doing a PCA to find the most important predictors
- Parallel analysis to find the most relevant number of principal components

2. Training five different models

- Linear model
- Generalized Linear Model with elasticnet regularization (GLMnet)
- randomForest model
- Generalized Additive Model (GAM)
- eXtreme Gradient Boosting model (XGBoost)
- Sum up
- Final model

3. Predicting prices on the `validation` set with final model
4. Conclusion about the results

Now, let's look at the data.

Data Exploration

Let's take a look at the structure of the data.

```
# Structure of train set
str(train)
```

```
## 'data.frame':    1460 obs. of  81 variables:
##  $ Id           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ MSSubClass   : int  60 20 60 70 60 50 20 60 50 190 ...
##  $ MSZoning     : chr  "RL" "RL" "RL" "RL" ...
##  $ LotFrontage  : int  65 80 68 60 84 85 75 NA 51 50 ...
##  $ LotArea      : int  8450 9600 11250 9550 14260 14115 10084 10382 6120 7420 ...
##  $ Street       : chr  "Pave" "Pave" "Pave" "Pave" ...
##  $ Alley        : chr  NA NA NA NA ...
##  $ LotShape     : chr  "Reg" "Reg" "IR1" "IR1" ...
##  $ LandContour  : chr  "Lvl" "Lvl" "Lvl" "Lvl" ...
##  $ Utilities    : chr  "AllPub" "AllPub" "AllPub" "AllPub" ...
##  $ LotConfig    : chr  "Inside" "FR2" "Inside" "Corner" ...
##  $ LandSlope    : chr  "Gtl" "Gtl" "Gtl" "Gtl" ...
##  $ Neighborhood : chr  "CollgCr" "Veenker" "CollgCr" "Crawfor" ...
##  $ Condition1   : chr  "Norm" "Feedr" "Norm" "Norm" ...
##  $ Condition2   : chr  "Norm" "Norm" "Norm" "Norm" ...
##  $ BldgType     : chr  "1Fam" "1Fam" "1Fam" "1Fam" ...
##  $ HouseStyle   : chr  "2Story" "1Story" "2Story" "2Story" ...
##  $ OverallQual  : int  7 6 7 7 8 5 8 7 7 5 ...
##  $ OverallCond  : int  5 8 5 5 5 5 5 6 5 6 ...
##  $ YearBuilt    : int  2003 1976 2001 1915 2000 1993 2004 1973 1931 1939 ...
##  $ YearRemodAdd : int  2003 1976 2002 1970 2000 1995 2005 1973 1950 1950 ...
##  $ RoofStyle    : chr  "Gable" "Gable" "Gable" "Gable" ...
##  $ RoofMatl     : chr  "CompShg" "CompShg" "CompShg" "CompShg" ...
##  $ Exterior1st  : chr  "VinylSd" "MetalSd" "VinylSd" "Wd Sdng" ...
##  $ Exterior2nd  : chr  "VinylSd" "MetalSd" "VinylSd" "Wd Shng" ...
##  $ MasVnrType   : chr  "BrkFace" "None" "BrkFace" "None" ...
##  $ MasVnrArea   : int  196 0 162 0 350 0 186 240 0 0 ...
##  $ ExterQual    : chr  "Gd" "TA" "Gd" "TA" ...
##  $ ExterCond    : chr  "TA" "TA" "TA" "TA" ...
##  $ Foundation   : chr  "PConc" "CBlock" "PConc" "BrkTil" ...
##  $ BsmtQual     : chr  "Gd" "Gd" "Gd" "TA" ...
##  $ BsmtCond     : chr  "TA" "TA" "TA" "Gd" ...
##  $ BsmtExposure : chr  "No" "Gd" "Mn" "No" ...
##  $ BsmtFinType1 : chr  "GLQ" "ALQ" "GLQ" "ALQ" ...
##  $ BsmtFinSF1   : int  706 978 486 216 655 732 1369 859 0 851 ...
##  $ BsmtFinType2 : chr  "Unf" "Unf" "Unf" "Unf" ...
##  $ BsmtFinSF2   : int  0 0 0 0 0 0 0 32 0 0 ...
##  $ BsmtUnfSF    : int  150 284 434 540 490 64 317 216 952 140 ...
##  $ TotalBsmtSF  : int  856 1262 920 756 1145 796 1686 1107 952 991 ...
```

```
##  $ Heating      : chr  "GasA" "GasA" "GasA" "GasA" ...
##  $ HeatingQC    : chr  "Ex" "Ex" "Ex" "Gd" ...
##  $ CentralAir   : chr  "Y" "Y" "Y" "Y" ...
##  $ Electrical   : chr  "SBrkr" "SBrkr" "SBrkr" "SBrkr" ...
##  $ X1stFlrSF    : int  856 1262 920 961 1145 796 1694 1107 1022 1077 ...
##  $ X2ndFlrSF    : int  854 0 866 756 1053 566 0 983 752 0 ...
##  $ LowQualFinSF : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ GrLivArea    : int  1710 1262 1786 1717 2198 1362 1694 2090 1774 1077 ...
##  $ BsmtFullBath : int  1 0 1 1 1 1 1 1 0 1 ...
##  $ BsmtHalfBath : int  0 1 0 0 0 0 0 0 0 0 ...
##  $ FullBath     : int  2 2 2 1 2 1 2 2 2 1 ...
##  $ HalfBath     : int  1 0 1 0 1 1 0 1 0 0 ...
##  $ BedroomAbvGr : int  3 3 3 3 4 1 3 3 2 2 ...
##  $ KitchenAbvGr : int  1 1 1 1 1 1 1 1 2 2 ...
##  $ KitchenQual  : chr  "Gd" "TA" "Gd" "Gd" ...
##  $ TotRmsAbvGrd : int  8 6 6 7 9 5 7 7 8 5 ...
##  $ Functional   : chr  "Typ" "Typ" "Typ" "Typ" ...
##  $ Fireplaces   : int  0 1 1 1 1 0 1 2 2 2 ...
##  $ FireplaceQu  : chr  NA "TA" "TA" "Gd" ...
##  $ GarageType   : chr  "Attchd" "Attchd" "Attchd" "Detchd" ...
##  $ GarageYrBlt  : int  2003 1976 2001 1998 2000 1993 2004 1973 1931 1939 ...
##  $ GarageFinish : chr  "RFn" "RFn" "RFn" "Unf" ...
##  $ GarageCars   : int  2 2 2 3 3 2 2 2 2 1 ...
##  $ GarageArea   : int  548 460 608 642 836 480 636 484 468 205 ...
##  $ GarageQual   : chr  "TA" "TA" "TA" "TA" ...
##  $ GarageCond   : chr  "TA" "TA" "TA" "TA" ...
##  $ PavedDrive   : chr  "Y" "Y" "Y" "Y" ...
##  $ WoodDeckSF   : int  0 298 0 0 192 40 255 235 90 0 ...
##  $ OpenPorchSF  : int  61 0 42 35 84 30 57 204 0 4 ...
##  $ EnclosedPorch: int  0 0 0 272 0 0 0 228 205 0 ...
##  $ X3SsnPorch   : int  0 0 0 0 0 320 0 0 0 0 ...
##  $ ScreenPorch  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ PoolArea     : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ PoolQC       : chr  NA NA NA NA ...
##  $ Fence        : chr  NA NA NA NA ...
##  $ MiscFeature  : chr  NA NA NA NA ...
##  $ MiscVal      : int  0 0 0 0 0 700 0 350 0 0 ...
##  $ MoSold       : int  2 5 9 2 12 10 8 11 4 1 ...
##  $ YrSold       : int  2008 2007 2008 2006 2008 2009 2007 2009 2008 2008 ...
##  $ SaleType     : chr  "WD" "WD" "WD" "WD" ...
##  $ SaleCondition: chr  "Normal" "Normal" "Normal" "Abnorml" ...
##  $ SalePrice    : int  208500 181500 223500 140000 250000 143000 307000 200000 129900 118000 ...
```

```r
# Structure of validation set
str(validation)
```

```
## 'data.frame':    1459 obs. of  80 variables:
##  $ Id           : int  1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 ...
##  $ MSSubClass   : int  20 20 60 60 120 60 20 60 20 20 ...
##  $ MSZoning     : chr  "RH" "RL" "RL" "RL" ...
##  $ LotFrontage  : int  80 81 74 78 43 75 NA 63 85 70 ...
##  $ LotArea      : int  11622 14267 13830 9978 5005 10000 7980 8402 10176 8400 ...
##  $ Street       : chr  "Pave" "Pave" "Pave" "Pave" ...
##  $ Alley        : chr  NA NA NA NA ...
```

```
##  $ LotShape     : chr  "Reg" "IR1" "IR1" "IR1" ...
##  $ LandContour  : chr  "Lvl" "Lvl" "Lvl" "Lvl" ...
##  $ Utilities    : chr  "AllPub" "AllPub" "AllPub" "AllPub" ...
##  $ LotConfig    : chr  "Inside" "Corner" "Inside" "Inside" ...
##  $ LandSlope    : chr  "Gtl" "Gtl" "Gtl" "Gtl" ...
##  $ Neighborhood : chr  "NAmes" "NAmes" "Gilbert" "Gilbert" ...
##  $ Condition1   : chr  "Feedr" "Norm" "Norm" "Norm" ...
##  $ Condition2   : chr  "Norm" "Norm" "Norm" "Norm" ...
##  $ BldgType     : chr  "1Fam" "1Fam" "1Fam" "1Fam" ...
##  $ HouseStyle   : chr  "1Story" "1Story" "2Story" "2Story" ...
##  $ OverallQual  : int  5 6 5 6 8 6 6 6 7 4 ...
##  $ OverallCond  : int  6 6 5 6 5 5 7 5 5 5 ...
##  $ YearBuilt    : int  1961 1958 1997 1998 1992 1993 1992 1998 1990 1970 ...
##  $ YearRemodAdd : int  1961 1958 1998 1998 1992 1994 2007 1998 1990 1970 ...
##  $ RoofStyle    : chr  "Gable" "Hip" "Gable" "Gable" ...
##  $ RoofMatl     : chr  "CompShg" "CompShg" "CompShg" "CompShg" ...
##  $ Exterior1st  : chr  "VinylSd" "Wd Sdng" "VinylSd" "VinylSd" ...
##  $ Exterior2nd  : chr  "VinylSd" "Wd Sdng" "VinylSd" "VinylSd" ...
##  $ MasVnrType   : chr  "None" "BrkFace" "None" "BrkFace" ...
##  $ MasVnrArea   : int  0 108 0 20 0 0 0 0 0 0 ...
##  $ ExterQual    : chr  "TA" "TA" "TA" "TA" ...
##  $ ExterCond    : chr  "TA" "TA" "TA" "TA" ...
##  $ Foundation   : chr  "CBlock" "CBlock" "PConc" "PConc" ...
##  $ BsmtQual     : chr  "TA" "TA" "Gd" "TA" ...
##  $ BsmtCond     : chr  "TA" "TA" "TA" "TA" ...
##  $ BsmtExposure : chr  "No" "No" "No" "No" ...
##  $ BsmtFinType1 : chr  "Rec" "ALQ" "GLQ" "GLQ" ...
##  $ BsmtFinSF1   : int  468 923 791 602 263 0 935 0 637 804 ...
##  $ BsmtFinType2 : chr  "LwQ" "Unf" "Unf" "Unf" ...
##  $ BsmtFinSF2   : int  144 0 0 0 0 0 0 0 0 78 ...
##  $ BsmtUnfSF    : int  270 406 137 324 1017 763 233 789 663 0 ...
##  $ TotalBsmtSF  : int  882 1329 928 926 1280 763 1168 789 1300 882 ...
##  $ Heating      : chr  "GasA" "GasA" "GasA" "GasA" ...
##  $ HeatingQC    : chr  "TA" "TA" "Gd" "Ex" ...
##  $ CentralAir   : chr  "Y" "Y" "Y" "Y" ...
##  $ Electrical   : chr  "SBrkr" "SBrkr" "SBrkr" "SBrkr" ...
##  $ X1stFlrSF    : int  896 1329 928 926 1280 763 1187 789 1341 882 ...
##  $ X2ndFlrSF    : int  0 0 701 678 0 892 0 676 0 0 ...
##  $ LowQualFinSF : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ GrLivArea    : int  896 1329 1629 1604 1280 1655 1187 1465 1341 882 ...
##  $ BsmtFullBath : int  0 0 0 0 0 0 1 0 1 1 ...
##  $ BsmtHalfBath : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ FullBath     : int  1 1 2 2 2 2 2 2 1 1 ...
##  $ HalfBath     : int  0 1 1 1 0 1 0 1 1 0 ...
##  $ BedroomAbvGr : int  2 3 3 3 2 3 3 3 2 2 ...
##  $ KitchenAbvGr : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ KitchenQual  : chr  "TA" "Gd" "TA" "Gd" ...
##  $ TotRmsAbvGrd : int  5 6 6 7 5 7 6 7 5 4 ...
##  $ Functional   : chr  "Typ" "Typ" "Typ" "Typ" ...
##  $ Fireplaces   : int  0 0 1 1 0 1 0 1 1 0 ...
##  $ FireplaceQu  : chr  NA NA "TA" "Gd" ...
##  $ GarageType   : chr  "Attchd" "Attchd" "Attchd" "Attchd" ...
##  $ GarageYrBlt  : int  1961 1958 1997 1998 1992 1993 1992 1998 1990 1970 ...
##  $ GarageFinish : chr  "Unf" "Unf" "Fin" "Fin" ...
```

```
##  $ GarageCars   : int  1 1 2 2 2 2 2 2 2 2 ...
##  $ GarageArea   : int  730 312 482 470 506 440 420 393 506 525 ...
##  $ GarageQual   : chr  "TA" "TA" "TA" "TA" ...
##  $ GarageCond   : chr  "TA" "TA" "TA" "TA" ...
##  $ PavedDrive   : chr  "Y" "Y" "Y" "Y" ...
##  $ WoodDeckSF   : int  140 393 212 360 0 157 483 0 192 240 ...
##  $ OpenPorchSF  : int  0 36 34 36 82 84 21 75 0 0 ...
##  $ EnclosedPorch: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ X3SsnPorch   : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ ScreenPorch  : int  120 0 0 0 144 0 0 0 0 0 ...
##  $ PoolArea     : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ PoolQC       : chr  NA NA NA NA ...
##  $ Fence        : chr  "MnPrv" NA "MnPrv" NA ...
##  $ MiscFeature  : chr  NA "Gar2" NA NA ...
##  $ MiscVal      : int  0 12500 0 0 0 0 500 0 0 0 ...
##  $ MoSold       : int  6 6 3 6 1 4 3 5 2 4 ...
##  $ YrSold       : int  2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 ...
##  $ SaleType     : chr  "WD" "WD" "WD" "WD" ...
##  $ SaleCondition: chr  "Normal" "Normal" "Normal" "Normal" ...
```

There are 1460 observations and 81 variables for the `train` set, while the `validation` set has 1459 observations and 80 variables. The difference in the number of variables is the `SalePrice` column we will try to predict. Among the variables in the `train` set, 43 of them are character variables, while the other 38 are numerical as shown below.

```r
# Number of categorical variables

categ_data <- select_if(train, is.character)
length(categ_data)
```

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

```r
# Number of numerical variables

num_data <- select_if(train, is.numeric)
length(num_data)
```

```
## [1] 38
```

What about NAs ? Do we have missing data ? Let's check that with the `naniar` package.

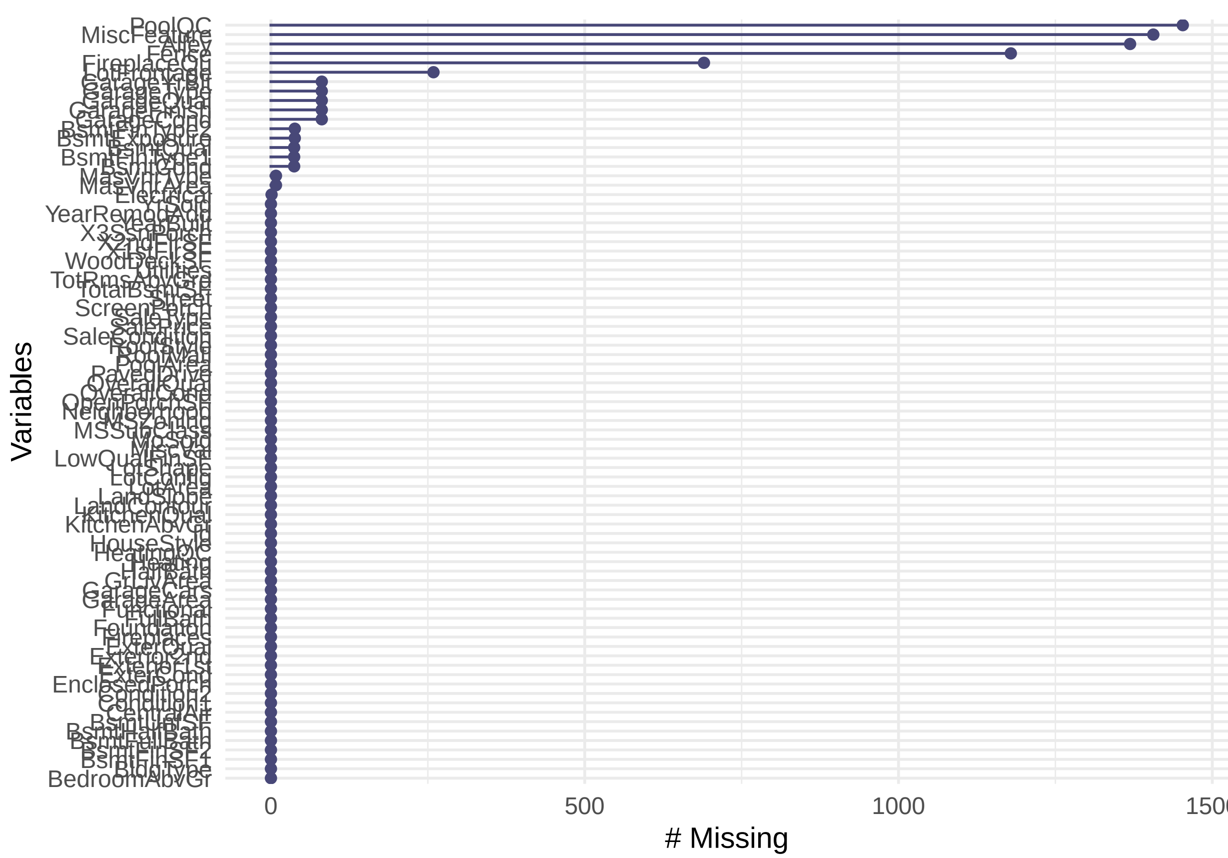
```r
library(naniar)

# Checking missing value
sum(is.na(train))
```

```
## [1] 6965
```

```r
miss_var_summary(train) %>% filter(n_miss > 0)
```

```
## # A tibble: 19 x 3
##    variable     n_miss pct_miss
##    <chr>         <int>    <dbl>
##  1 PoolQC         1453   99.5
##  2 MiscFeature    1406   96.3
##  3 Alley          1369   93.8
##  4 Fence          1179   80.8
##  5 FireplaceQu     690   47.3
##  6 LotFrontage     259   17.7
##  7 GarageType       81    5.55
##  8 GarageYrBlt      81    5.55
##  9 GarageFinish     81    5.55
## 10 GarageQual       81    5.55
## 11 GarageCond       81    5.55
## 12 BsmtExposure     38    2.60
## 13 BsmtFinType2     38    2.60
## 14 BsmtQual         37    2.53
## 15 BsmtCond         37    2.53
## 16 BsmtFinType1     37    2.53
## 17 MasVnrType        8    0.548
## 18 MasVnrArea        8    0.548
## 19 Electrical        1    0.0685
```

```r
# Barplot of variables with missing values
gg_miss_var(train)
```



It seems there are 19 variables with missing values, the sum of which equals to 6965. But if we look closely at the documentation, most of the NAs mean that the house does not have the specified feature. For example, as we saw in the plot above, 99.5% of the observations are NAs in the `PoolQC` column. This simply means that 99.5% of the houses in the `train` set does not have any pool, and only 7 houses out of 1460 does have one whose quality is fair, average, good or excellent. This is the same thing for most of the other variables like `Alley` (96.3% of houses have no alley access), `Fence` (93.8% have no fence) or the `Garage` variables.

It looks like we must clean and impute missing data before proceeding our analysis. We can wonder whether some of these variables have statistical significance since there are few information, but we will make a decision after cleaning the whole dataset.

Cleaning Data and Missing Data Imputation

We will first impute missing data for the 19 variables shown above. We see that there are mainly ordinal variables that describe quality from "Poor" to "Excellent" Quality, and NA means "None". For these variables, we will impute "None" instead of "NA". There are 15 variables in which we will impute "None" with the following code. All fifteen variables are : PoolQC, Fence, BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1, BsmtFinType2, FireplaceQu, GarageFinish, GarageQual, GarageCond, Alley, MiscFeature, GarageType, and MasVnrType.

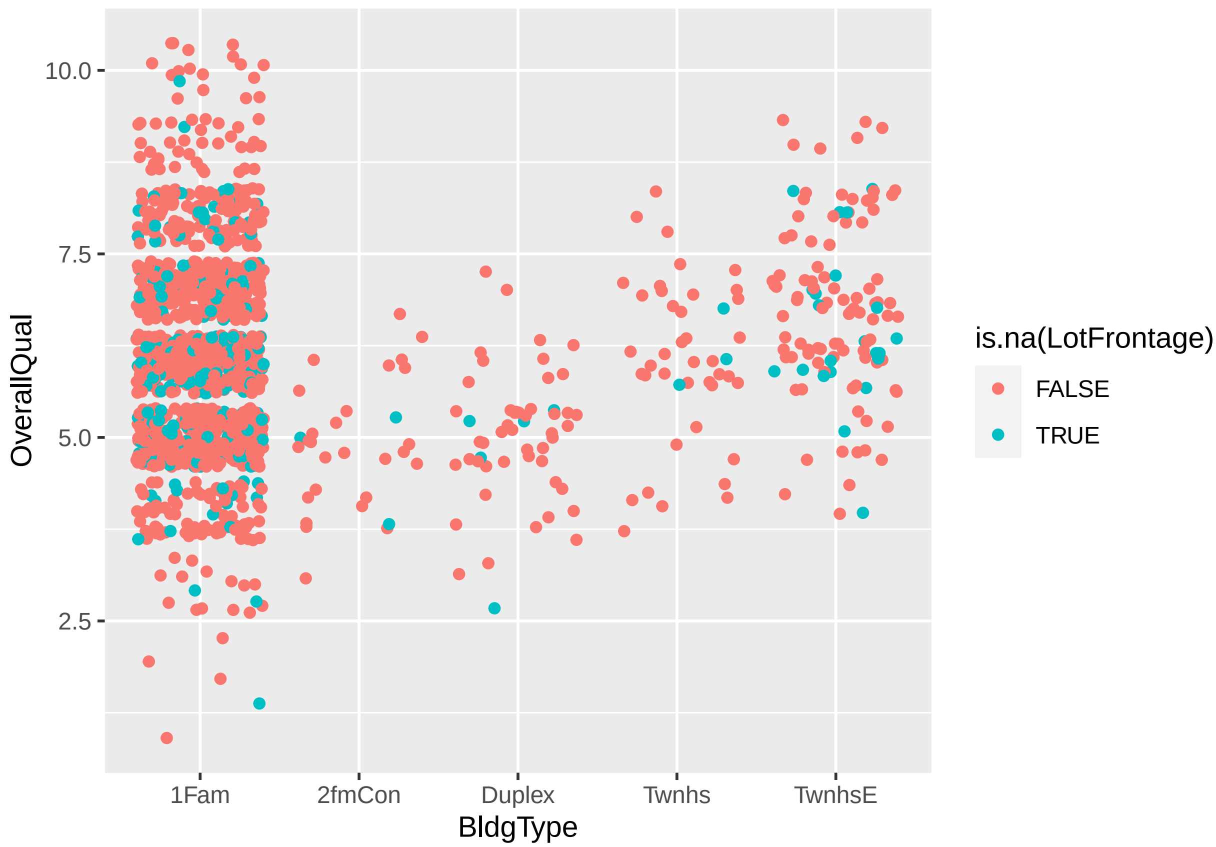
```r
# According to data description most NAs mean actually "None"
# Imputing "None" for 15 variables.

# For ordinal variables :
train$PoolQC[is.na(train$PoolQC)] <- "None"
train$Fence[is.na(train$Fence)] <- "None"
train$BsmtQual[is.na(train$BsmtQual)] <- "None"
train$BsmtCond[is.na(train$BsmtCond)] <- "None"
train$BsmtExposure[is.na(train$BsmtExposure)] <- "None"
train$BsmtFinType1[is.na(train$BsmtFinType2)] <- "None"
train$BsmtFinType2[is.na(train$BsmtFinType2)] <- "None"
train$FireplaceQu[is.na(train$FireplaceQu)] <- "None"
train$GarageFinish[is.na(train$GarageFinish)] <- "None"
train$GarageQual[is.na(train$GarageQual)] <- "None"
train$GarageCond[is.na(train$GarageCond)] <- "None"

# For categorical variables :
train$MiscFeature[is.na(train$MiscFeature)] <- "None"
train$Alley[is.na(train$Alley)] <- "None"
train$GarageType[is.na(train$GarageType)] <- "None"
train$MasVnrType[is.na(train$MasVnrType)] <- "None"
```

We now have some numeric and date variables with NAs. As we have just replaced NAs in the `MasVnrType` variable with "None", the `MasVnrArea` variable must be zero. And for the `GarageYrBlt` variable, we will replace NAs by the values in the `YearBuilt` variable, which is the year when the house was built.

```r
# For numerical variables
# If MasVnrType missing data are "None", then area must be 0

train$MasVnrArea[is.na(train$MasVnrArea)] <- 0

# We replace GarageYrBlt by the YearBuilt value
train$GarageYrBlt[is.na(train$GarageYrBlt)] <- train$YearBuilt[is.na(train$GarageYrBlt)]
```

The values in the LotFrontage variable (which is “linear feet of street connected to property”) are the most difficult to find. I first thought it was linked to the type of building, but it doesn’t seem to be the case, as shown in the plot below. NAs are among every type of building for all kind of quality.

```
# LotFrontage variable : Where are the NAs ?

train %>% ggplot(aes(BldgType, OverallQual, color = is.na(LotFrontage))) +
  geom_jitter()
```



Therefore, we will replace NAs with a PCA algorithm from the missMDA package. For convenience, we will take only numerical variables and create a matrix with them. We then estimate the optimal number of dimensions, and we impute the missing values with the number of components we’ve estimated. We extract the complete matrix and change it as a data frame as shown with the following piece of code.

```
library(missMDA)

# Creating matrix with numerical variables
num_matrix <- select_if(train, is.numeric) %>% as.matrix()

# Estimation of the optimal number of dimensions
number_cp <- estim_ncpPCA(num_matrix)

# Imputing values instead of NAs
complete_matrix <- imputePCA(num_matrix, ncp = number_cp$ncp, scale = TRUE)

# Extracting the complete matrix
```

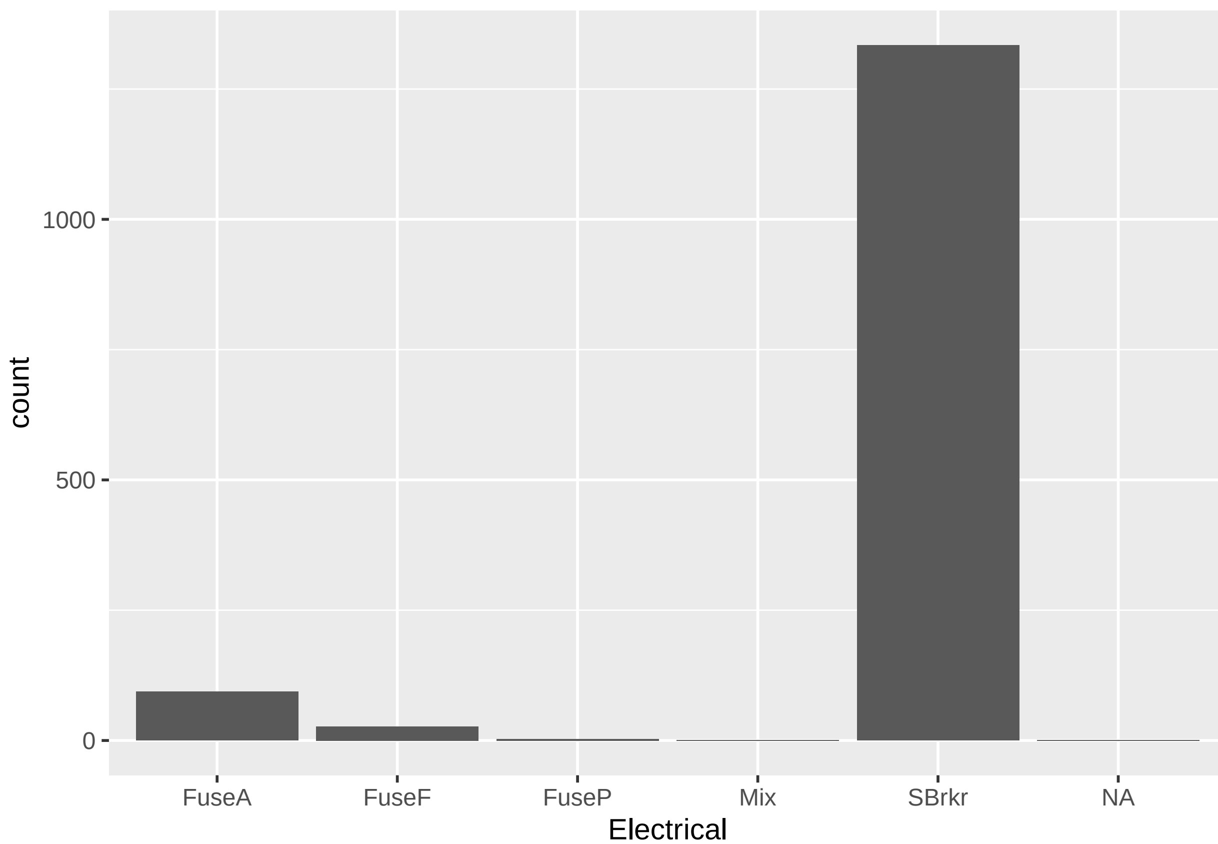
```r
clean_matrix <- complete_matrix$completeObs

# Changing the complete matrix as a data frame
clean_num <- as.data.frame(clean_matrix)
```

We then replace NAs in the `train` set by the values found in the `clean_num` data frame.

```r
# Replacing NAs in the train set for 'LotFrontage' variable :

train$LotFrontage[is.na(train$LotFrontage)] <- clean_num$LotFrontage[is.na(train$LotFrontage)]
```

Finally there is just 1 NA in the `Electrical` variable. According to its distribution, it doesn't seem too risky to replace the NA by the most common value "SBrkr".

```r
# We replace the only NA in Electrical variable with the most common value : SBrkr
train %>% ggplot(aes(Electrical)) +
  geom_bar()
```




```r
train$Electrical[is.na(train$Electrical)] <- "SBrkr"
```

To finish cleaning the data, we will convert some character columns into factors or numerical variables, some numerical columns containing dates into date variables, a numerical variable into factor variable, and one character column into a logical variable. Concerning the latter, it is the `CentralAir` variable that has just two inputs : "Y" for "Yes" and "N" for "No". As we prefer numerical values, we will convert them with "0" and "1".

```
# Changing 'CentralAir' variable with '0' and '1' :

train <- train %>% mutate(CentralAir = ifelse(CentralAir == "Y", 1, 0))
```

We must notice that there is a numerical variable that is actually a categorical variable : I mean the `MSSubClass` variable, which describes the type of dwelling. We will therefore convert it into a factor variable.

```
# Converting numerical variable 'MSSubClass' to factor variable :

train$MSSubClass <- factor(train$MSSubClass)
```

Among the character variables, we can notice a lot of them are ordinal variables explaining different degrees of quality of the features. As there are common labels to express quality among several variables, for example `c("Po", "Fa", "TA", "Gd")` to describe quality from "Poor" to "Excellent", changing them as factors will cause errors due to duplication. We will instead replace those degrees by numerical values. For instance, we will replace "Po", "Fa", "TA", "Gd" by 1, 2, 3 and 4. If there is "None", we will replace it by 0. There is a total of 19 ordinal variables. We will be careful to keep the order with the numerical values.

```
# Changing character ordinal variables to numerical ordinal variables

train <- train %>%
  mutate(
    LotShape = as.integer(as.character(factor(LotShape,
               levels = c("IR3", "IR2", "IR1", "Reg"), ordered = TRUE, labels = c(1, 2, 3, 4)))),
    LandContour = as.integer(as.character(factor(LandContour,
                  levels = c("Low", "HLS", "Bnk", "Lvl"), ordered = TRUE, labels = c(1, 2, 3, 4)))),
    LandSlope = as.integer(as.character(factor(LandSlope,
                levels = c("Gtl", "Mod", "Sev"), ordered = TRUE, labels = c(1, 2, 3)))),
    ExterQual = as.integer(as.character(factor(ExterQual,
                levels = c("Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(1, 2, 3, 4, 5)))),
    ExterCond = as.integer(as.character(factor(ExterCond,
                levels = c("Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(1, 2, 3, 4, 5)))),
    BsmtQual = as.integer(as.character(factor(BsmtQual,
               levels = c("None", "Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(0, 1, 2, 3,
    BsmtCond = as.integer(as.character(factor(BsmtCond,
               levels = c("None", "Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(0, 1, 2, 3,
    BsmtExposure = as.integer(as.character(factor(BsmtExposure,
                   levels = c("None", "No", "Mn", "Av", "Gd"), ordered = TRUE, labels = c(0, 1, 2, 3, 4
    BsmtFinType1 = as.integer(as.character(factor(BsmtFinType1,
                   levels = c("None", "Unf", "LwQ", "Rec", "BLQ", "BLQ", "ALQ", "GLQ"),
                   ordered = TRUE, labels = c(0, 1, 2, 3, 4, 5, 6, 7)))),
    BsmtFinType2 = as.integer(as.character(factor(BsmtFinType2,
                   levels = c("None", "Unf", "LwQ", "Rec", "BLQ", "BLQ", "ALQ", "GLQ"),
                   ordered = TRUE, labels = c(0, 1, 2, 3, 4, 5, 6, 7)))),
    HeatingQC = as.integer(as.character(factor(HeatingQC,
                levels = c("Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(1, 2, 3, 4, 5)))),
    KitchenQual = as.integer(as.character(factor(KitchenQual,
                  levels = c("Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(1, 2, 3, 4, 5)))
    Functional = as.integer(as.character(factor(Functional,
                 levels = c("Sal", "Sev", "Maj2", "Maj1", "Mod", "Min2", "Min1", "Typ"),
                 ordered = TRUE, labels = c(1, 2, 3, 4, 5, 6, 7, 8)))),
    FireplaceQu = as.integer(as.character(factor(FireplaceQu,
                  levels = c("None", "Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(0, 1, 2,
```

```
    GarageFinish = as.integer(as.character(factor(GarageFinish,
                   levels = c("None", "Unf", "RFn", "Fin"), ordered = TRUE, labels = c(0, 1, 2, 3)))),
    GarageQual = as.integer(as.character(factor(GarageQual,
                 levels = c("None", "Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(0, 1, 2, 3
    GarageCond = as.integer(as.character(factor(GarageCond,
                 levels = c("None", "Po", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(0, 1, 2, 3
    PoolQC = as.integer(as.character(factor(PoolQC,
             levels = c("None", "Fa", "TA", "Gd", "Ex"), ordered = TRUE, labels = c(0, 1, 2, 3, 4)))),
    Fence = as.integer(as.character(factor(Fence,
            levels = c("None", "MnWw", "GdWo", "MnPrv", "GdPrv"), ordered = TRUE, labels = c(0, 1, 2, 3
```

We can now convert the remaining character variables to factors with the following piece of code.

```
# Number of remaining character variables :
train %>% select_if(is.character) %>% length()
```

```
## [1] 23
```

```
# Changing remaining character variables to factor variables

train <- train %>% mutate_if(is.character, as.factor)
```

We will finish this cleaning process with the date variables. There are 5 date variables : `YearBuilt` is the original construction date, `YearRemodAdd` is the remodel date (but is equal to `YearBuilt` if there is none), `GarageYrBlt` and `MoSold` and `YrSold`. We will use the `lubridate` package to convert those variables into "date" class variables. For `MoSold` and `YrSold`, we will create a new variable called `DateSold` as we prefer to have both month and year in the same column to calculate duration. As we have almost only years available among the variables, the `parse_date_time` function will replace day and month by "01" when converting. We will also add two more variables : `durationBltSold` that calculates the difference between the sale date and the construction date, and `durationRemodSold`, which is the difference between the sale date and the remodel date as we may be interested in correlations between the sale price and newer or older house. We do all this with the following code.

```
# Converting "Year" variables to date variables and merging 'Sold Month' and 'Sold Year' :

train <- train %>% mutate(YearBuilt = parse_date_time(YearBuilt, orders = "Y"),
                          YearRemodAdd = parse_date_time(YearRemodAdd, orders = "Y"),
                          GarageYrBlt = parse_date_time(GarageYrBlt, orders = "Y"),
                          DateSold = make_date(year = YrSold, month = MoSold))

# Adding duration variables :
train <- train %>% mutate(durationBltSold = difftime(DateSold, YearBuilt, units = "weeks"),
                          durationRemodSold = difftime(DateSold, YearRemodAdd, units = "weeks"))
```

We clean the validation set the same way we did for the train set. Most of the NAs are the same in both sets. So we won't get in the details as we can find the full code in the .R file from the GitHub repository we have downloaded (or the following link containing the .R code for this project : github.com).

So, to sum up what we have just done with the train set and what we will do with the validation set is as follows :

- Imputing "None" for variables instead of NAs when the documentation explicitly indicates that NA means "none".

- Imputing "0" if it's a consequence of imputing "None" (for an area variable for example).
- Converting ordinal character variables into ordinal numerical variables.
- Converting those numerical ordinal variables into factors, adding "0" for "None".
- Converting numerical "Year" variables into date variables.
- When there are several other NAs, imputing missing data with a PCA algorithm.
- When there is 1 or 2 other NAs, imputing most common value or median.
- Converting remaining character variables into factors.
- Converting one character variable into binary variable (CentralAir).
- Converting one numerical variable into factor (MSSubClass).

We make also a lot of plots depending on the variables to find the best value to impute or the best way to proceed.

We won't show the cleaning process of the validation set with the plots and results, but you can check them with .R code file.

As we changed a lot of character variables into factors, we must be sure we have the same levels in both sets. This can be done with the following code.

```
# We must be sure we have the same number of levels of in both sets.
# Checking the levels of factors in train and validation sets :
a <- train %>% summarise_if(is.factor, nlevels)

b <- validation %>% summarise_if(is.factor, nlevels)

a == b
```

```
##      MSSubClass MSZoning Street Alley Utilities LotConfig Neighborhood
## [1,]      FALSE     TRUE   TRUE  TRUE     FALSE      TRUE         TRUE
##      Condition1 Condition2 BldgType HouseStyle RoofStyle RoofMatl Exterior1st
## [1,]       TRUE      FALSE     TRUE      FALSE      TRUE    FALSE       FALSE
##      Exterior2nd MasVnrType Foundation Heating Electrical GarageType PavedDrive
## [1,]       FALSE       TRUE       TRUE   FALSE      FALSE       TRUE       TRUE
##      MiscFeature SaleType SaleCondition
## [1,]       FALSE     TRUE          TRUE
```

```
# To be sure we have the same number of levels in both sets, we bind
# the sets by row and reordering the levels.
temp_df <- rbind(train[,-81], validation)

temp_df <- temp_df %>% mutate_if(is.factor, as.factor)

# Getting the train set reordered
train_reordered <- temp_df[1:1460, ]

# We add the 'SalePrice' column from the previous train set :
train_reordered$SalePrice <- train$SalePrice

validation <- temp_df[1461:nrow(temp_df), ]


# Now we can see that all levels are the same :
c <- train_reordered %>% summarise_if(is.factor, nlevels)
```

```r
d <- validation %>% summarise_if(is.factor, nlevels)

c == d
```

```
##      MSSubClass MSZoning Street Alley Utilities LotConfig Neighborhood
## [1,]       TRUE     TRUE   TRUE  TRUE      TRUE      TRUE         TRUE
##      Condition1 Condition2 BldgType HouseStyle RoofStyle RoofMatl Exterior1st
## [1,]       TRUE       TRUE     TRUE       TRUE      TRUE     TRUE        TRUE
##      Exterior2nd MasVnrType Foundation Heating Electrical GarageType PavedDrive
## [1,]        TRUE       TRUE       TRUE    TRUE       TRUE       TRUE       TRUE
##      MiscFeature SaleType SaleCondition
## [1,]        TRUE     TRUE          TRUE
```

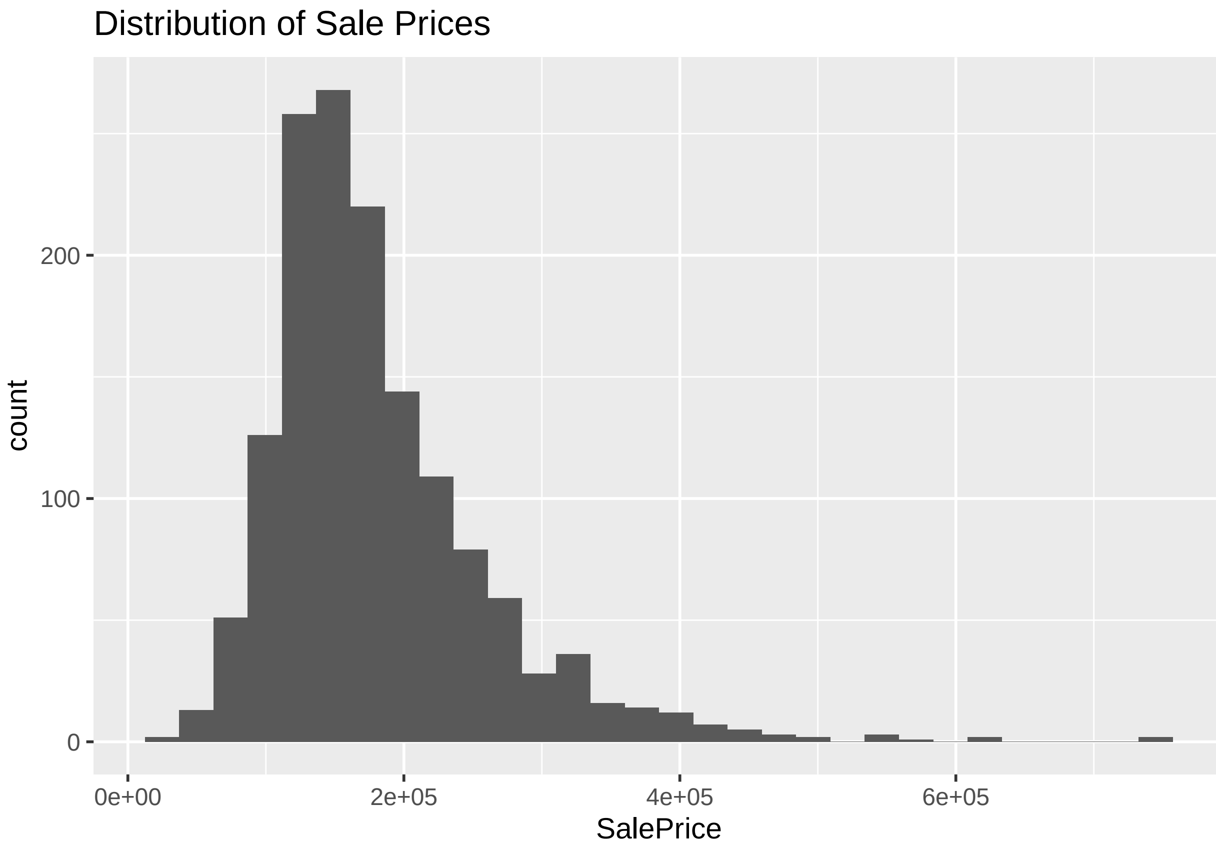
```r
# Everything is clean. We rename 'train_reordered' with a shorter name and reuse the name 'train' :

train <- train_reordered
```

Our dataset is finally clean. We can proceed our analysis. From now, we won't touch the `validation` set until we're done with our final model.

Analysis of the sale price variable

Now, let's go back to the `SalePrice` column, the outcome we want to predict, and see its distribution.

```r
# Distribution of Sale Prices

ggplot(train, aes(SalePrice)) +
  geom_histogram() +
  ggtitle("Distribution of Sale Prices")
```



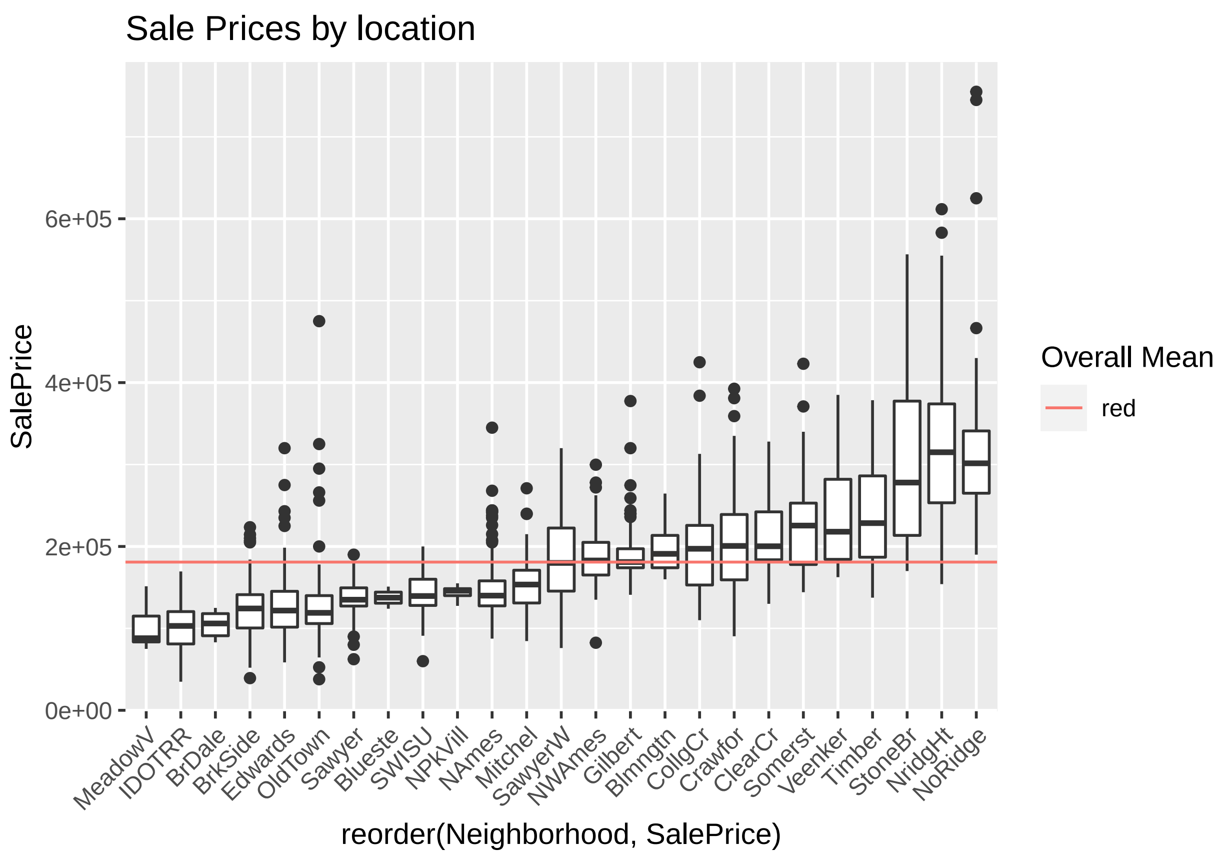
The distribution is right-skewed and most of the houses seem to be sold under $200,000. We can summarize the data in this plot with the following piece of code that confirms what we saw on the histogram, with a median at $163,000 and the mean under $181,000.

```
# Summary of Sale Prices Distribution

summary(train$SalePrice)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   34900  129975  163000  180921  214000  755000
```

One of the first things that comes to mind when we talk about the price of a house is its area and its location. The bigger the house, the higher the price. Prices tend also to get higher in the most attractive places. Let's see if we find the same tendencies in the dataset.

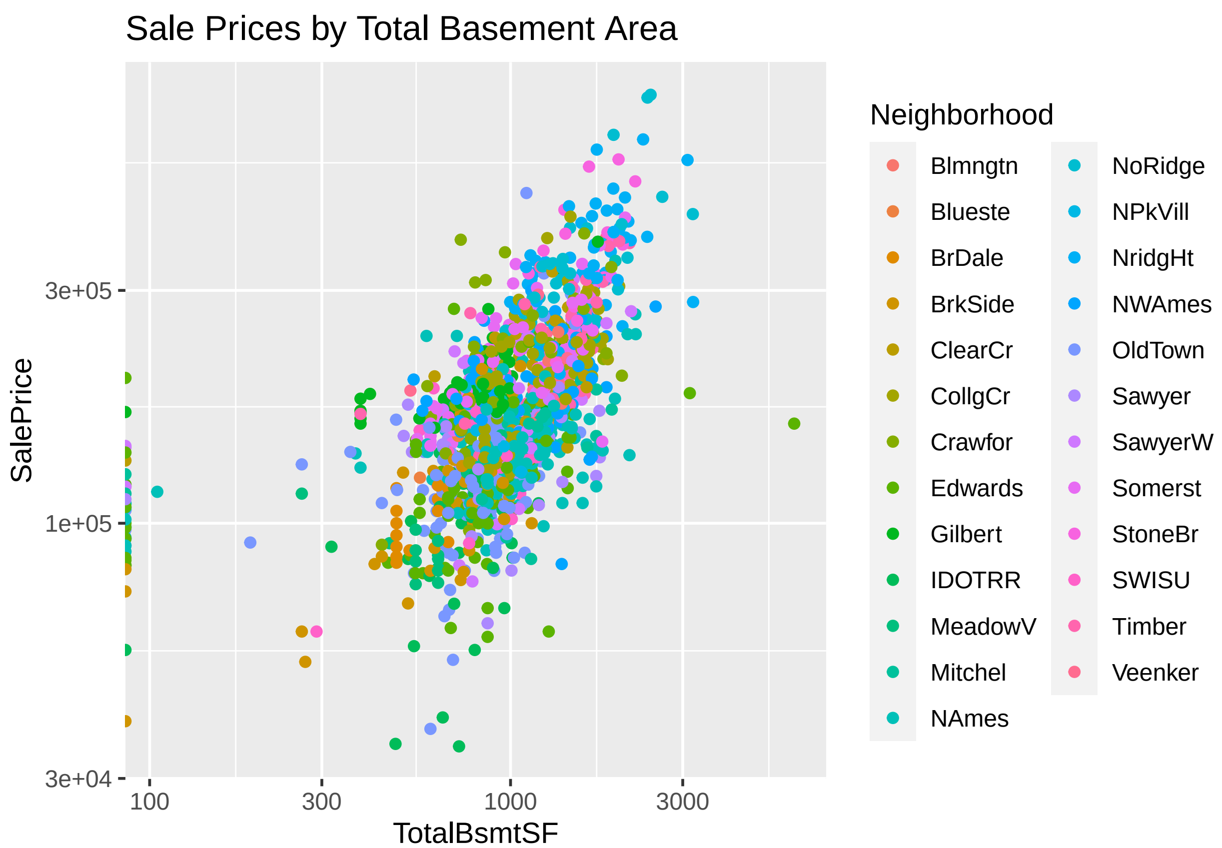
```
# Boxplot of Sale Prices by location

ggplot(train, aes(reorder(Neighborhood, SalePrice), SalePrice)) +
  geom_boxplot() +
  geom_hline(aes(yintercept = mean(SalePrice), color = "red")) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(title = "Sale Prices by location",
       col = "Overall Mean")
```



As we can see, there are three main locations where prices are well above the mean and having the most expensive houses (i.e., Northridge, Northridge Heights and Stone Brook). On the other hand, some locations are almost half the mean, that is to say around $100,000 or below (for example, Meadow Village or Iowa DOT and Rail Road).

What about the area ? There are several variables describing area, but we will use the total basement area as we want the whole area of the house. We add a log scale for a better visualization and colors with the `Neighborhood` variable. As we could expect, the biggest houses seem to be located in the most attractive places.

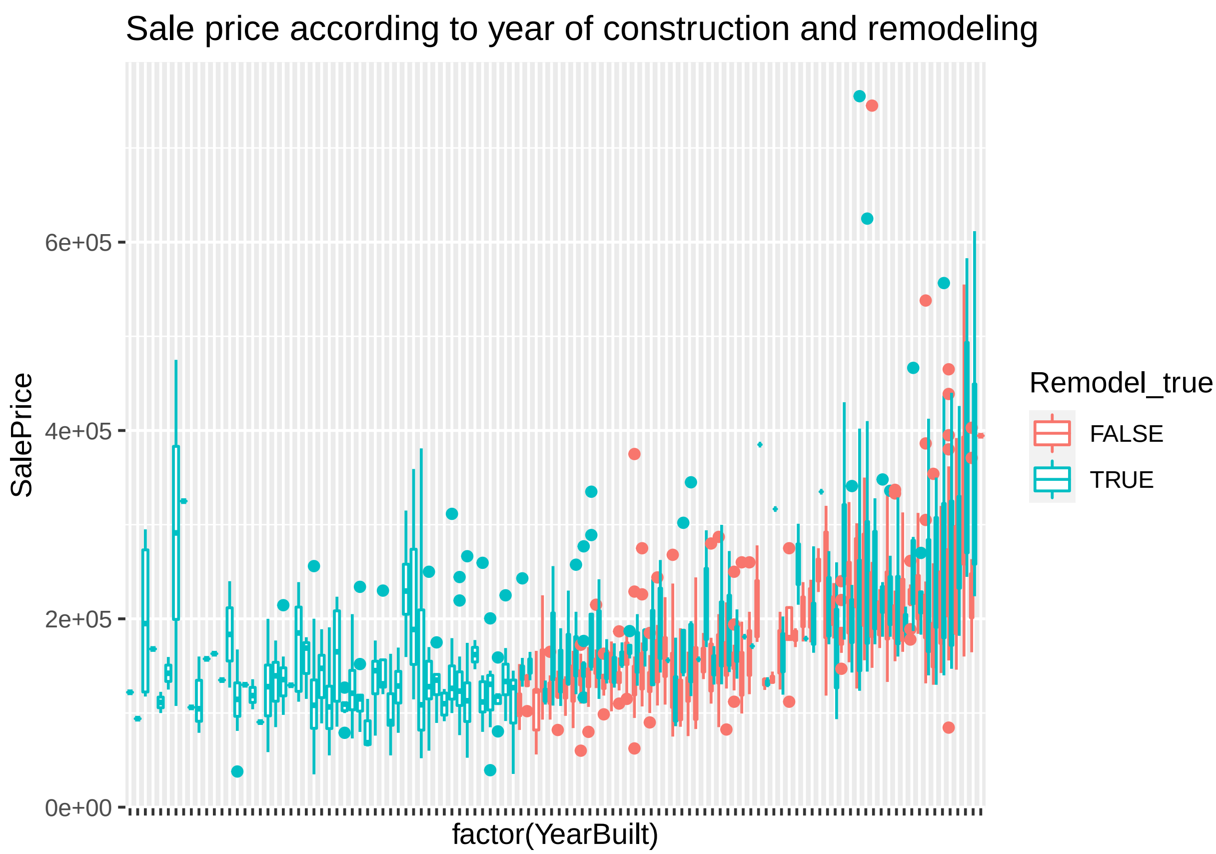
```
# Plot of Sale Prices by Total Basement Area

ggplot(train, aes(TotalBsmtSF, SalePrice, color = Neighborhood)) +
  geom_point(position = "jitter") +
  scale_x_log10() +
  scale_y_log10() +
  ggtitle("Sale Prices by Total Basement Area")
```



This scatterplot seem to follow a bivariate normal distribution. Therefore, it seems there is a correlation between the sale price and `TotalBsmtSF` area variable. However, we can expect the same kind of correlations between the sale price and other "area" variables. This will cause some problems of collinearity and could add noise to our model. We'll talk more about this in the next part.

One more thing we can think of is whether there is a correlation between the sale price and old or new houses. Let' check that with the next plot. We will also check if there is a link with remodeled houses. Remember that the remodeling year `YearRemodAdd` can be the same as the year of construction when there was no remodeling. So we will use an `ifelse` function to know whether the house was remodeled or not with the following code.

```
# Plot of Sale Prices according to year of construction

train %>% mutate(Remodel_diff = YearRemodAdd - YearBuilt,
                 Remodel_true = ifelse(Remodel_diff > 0, TRUE, FALSE)) %>%
ggplot(aes(factor(YearBuilt), SalePrice, color = Remodel_true)) +
  geom_boxplot() +
  theme(axis.text.x = element_blank()) +
  ggtitle("Sale price according to year of construction and remodeling")
```



As we can see, the most recent houses tend to be more expensive than older houses. Most of the houses were remodeled and they seem to be a bit more expensive compared to houses that were not, but the correlation seem quite slight.

As there are still a lot of variables, we must find a quicker way to get insights from the dataset. Let's start by creating a grid plot with all the numerical variables. We will plot them against the `SalePrice` to check the distribution between the predictors and the outcome. We can do this with the following piece of code.

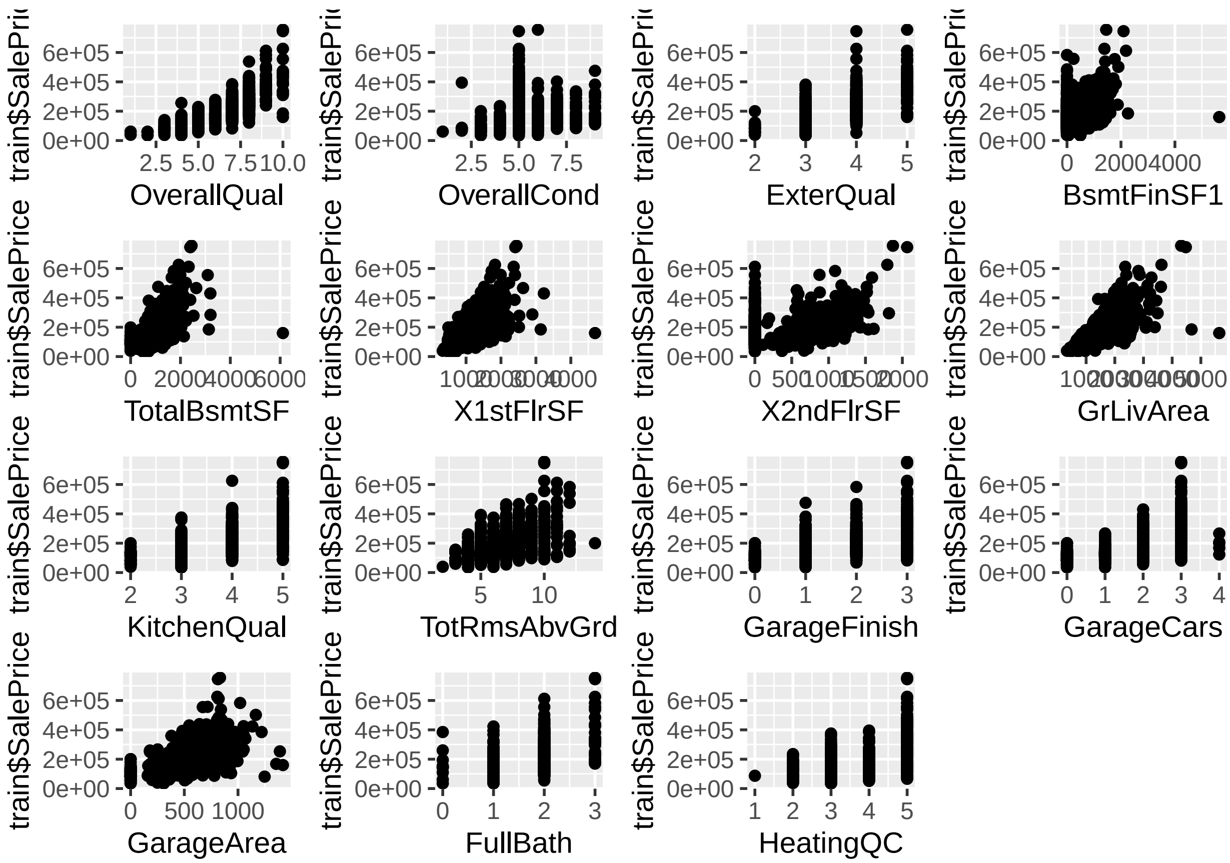
```
# Selecting only numerical variables

num_data <- select_if(train, is.numeric)
length(num_data)
```

```
## [1] 54
```

```
# Creating a grid plot with all numerical variables against the 'SalePrice' :
grid_num <- lapply(2:ncol(num_data[, -54]),
                   function(col) ggplot2::qplot(x = num_data[[col]],
                                                y = train$SalePrice,
                                                geom = "point",
                                                xlab = names(num_data)[[col]],
                                                ylab = ""))

cowplot::plot_grid(plotlist = grid_num)
```

This plot needs to be seen on a full screen to get insights from it. For the purpose of this report and clarity of visualization for the reader, I will plot only variables that seem to follow a bivariate normal distribution with the outcome according to the grid plot above.

```
# Variables with bivariate normal distribution

selected_num <- num_data %>% dplyr::select(OverallQual, OverallCond, ExterQual, BsmtFinSF1, TotalBsmtSF
                    GrLivArea, KitchenQual, TotRmsAbvGrd, GarageFinish, GarageCars, GarageArea,
                    FullBath, HeatingQC)

grid_num_selected <-lapply(1:ncol(selected_num),
                    function(col) ggplot2::qplot(x = selected_num[[col]],
                                                 y = train$SalePrice,
                                                 geom = "point",
                                                 xlab = names(selected_num)[[col]]))

cowplot::plot_grid(plotlist = grid_num_selected)
```



These variables and the outcome `SalePrice` seem to follow a bivariate normal distribution. Therefore they should be correlated. We can notice that most of these variables are either measuring areas, or measuring quality.

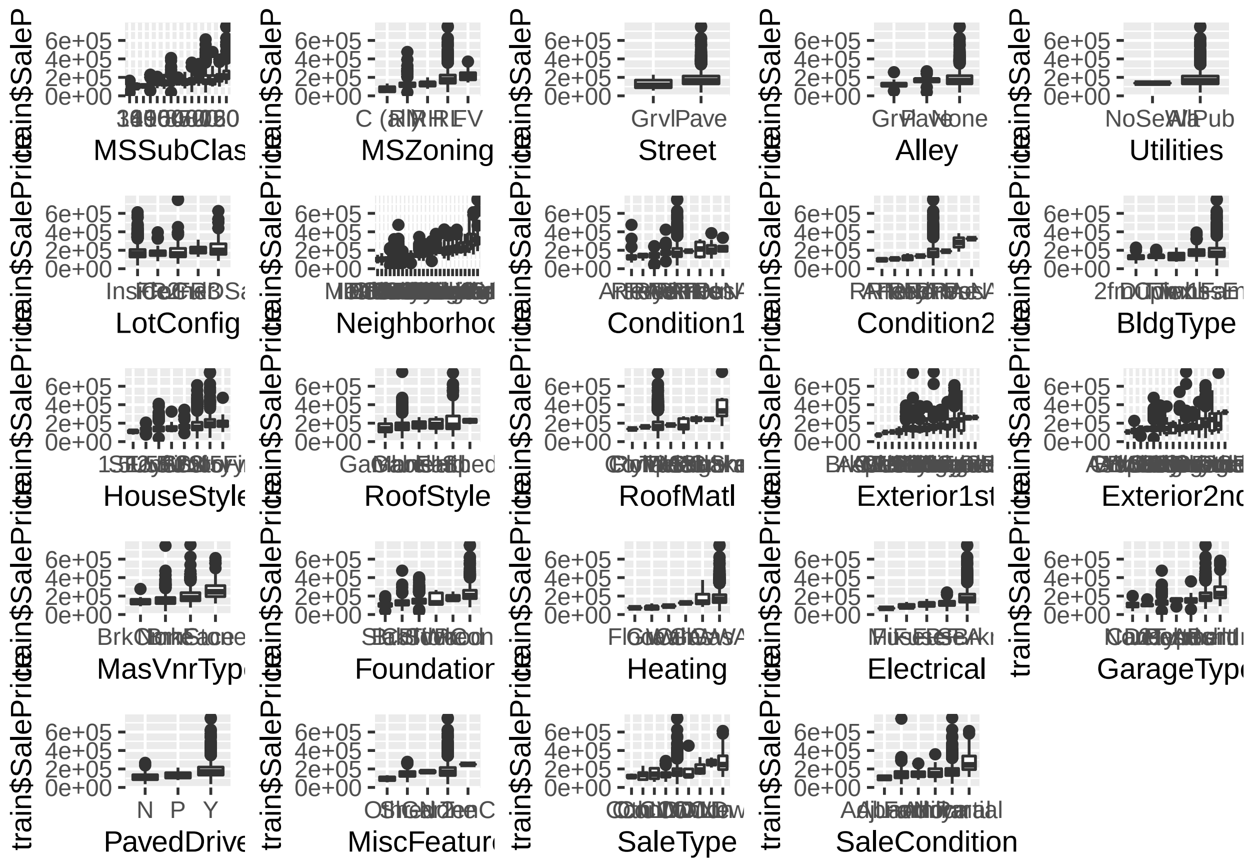
Let's do the same thing with the categorical variables. We make sure to reorder them by the `SalePrice`, like we did earlier with the `Neighborhood` variable.

```
# Selecting only categorical variables
```

```
categ_data <- select_if(train, is.factor)
length(categ_data)
```

```
## [1] 24
```

```
# Creating a grid plot with all categorical variables against the 'SalePrice' :
list_factors <-lapply(1:ncol(categ_data),
              function(col) ggplot2::qplot(reorder(categ_data[[col]], train$SalePrice), train$SalePrice
                                            geom = "boxplot",
                                            xlab = names(categ_data)[[col]]))

cowplot::plot_grid(plotlist = list_factors)
```



The labels on the x-axis are a bit hard to read for some variables, but we're not interested in that for the moment. Our main focus is to find which variables describe differences in the sale prices. We can see the `Neighborhood` variable on the 7th plot we've already plotted earlier. We've already noticed its importance but we can see that plots `MSSubClass`, `MSZoning`, `Condition1` and `Condition2`, `RoofStyle`, `RoofMatl`, `Exterior1st`, `Exterior2nd` and `MasVnrType` look also important while other variables like `GarageType`, `SaleType` and `SaleCondition` seem a bit less correlated. `MSSubClass` and `MSZoning` describe the type of dwelling and the general zoning, `Condition1` and `Condition2` both describe the proximity to various conditions (if there is more than one for the latter), `RoofStyle` describes the type of roof and `RoofMatl` the material used for the roof, `MasVnrType` is the type of masonry veneer, `Exterior1st` and `Exterior2nd` both describe exterior covering (if more than one material for the latter). For these last two variables, we can see that the plots are almost the same, so we should only use one for modeling. `GarageType` describes the type of garage, and `SaleType` and `SaleCondition` describe the type of sale and the condition of sale.

These categorical variables seem more important than the others. Some other variables seem to be slighter correlated, but we will mainly consider the ones we've described above in our model.

Up to now, we have briefly studied only variables based on common sense and experience, that is to say area, location and year of construction. We have also checked categorical variables against the `SalePrice` variable with a grid boxplot. We have also found that there are 10 numerical variables that follow a bivariate normal distribution with the sale price. Some of them are expected to have collinearity with others, and some others will explain more or less of the variability. We will therefore build a correlation matrix and do a Principal Component Analysis to better understand correlations between variables.

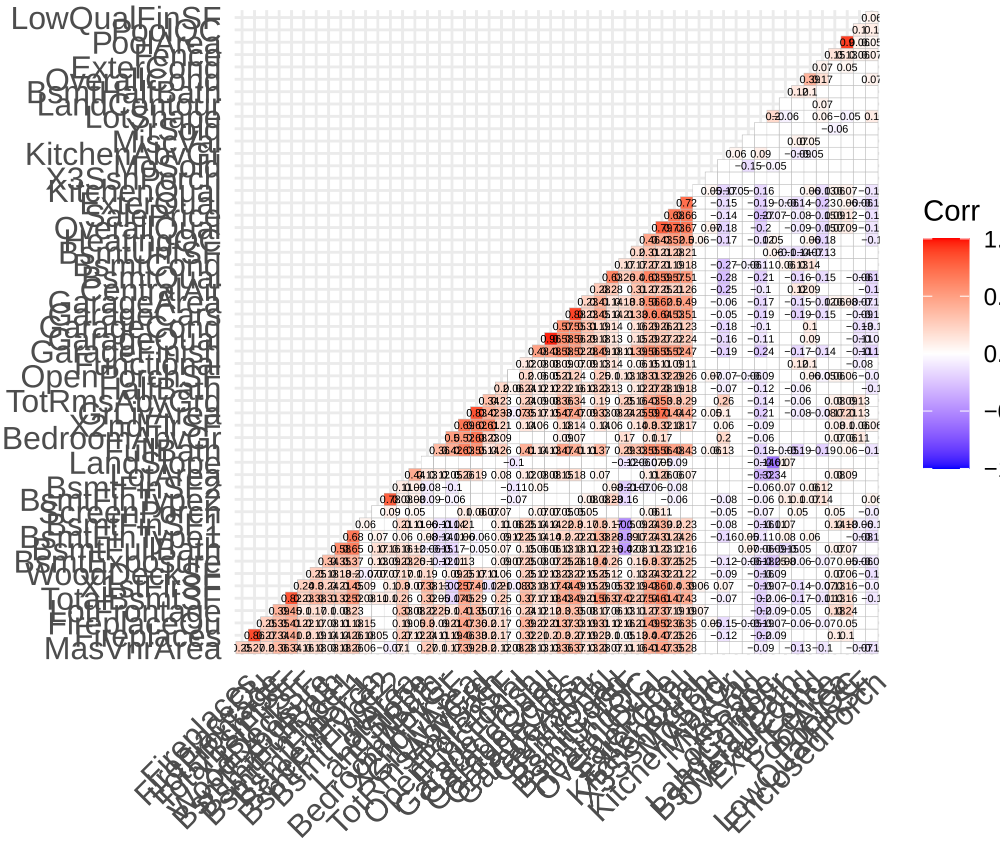
Correlation matrix and PCA

Let's first check which variables are the most correlated with the `SalePrice` variable. We will use the `cor` function and the `ggcorrplot` package to plot the matrix of correlations. As there are a lot of numerical variables, we will also create a matrix of the p-values of the correlations in order to get rid of insignificant correlation values and get a better visualization. The `Id` variable is not useful here to study correlations with the the sale price. So we won't input it neither. We do all of this with the following code.

```
# Getting correlation matrix for all numerical variables

train_num <- dplyr::select_if(train, is.numeric)
length(train_num) # We have now 54 numerical variables
```

```
## [1] 54
```

```
train_num_correl <- cor(train_num[, -1])# The 'Id' variable (the 1st column) will not be useful

# Creating a matrix of p-value to get rid of insignificant values on the correlation matrix
library(ggcorrplot)

p_mat <- cor_pmat(train_num[, -1])

# Plot of the correlations

ggcorrplot(as.matrix(train_num_correl),
           hc.order = TRUE,
           type = "lower",
           lab = TRUE,
           lab_size = 1.5,
           p.mat = p_mat,
           insig = "blank")
```



According to the plot, the `OverallQual` variable, an ordinal variable describing the overall quality of the houses, is the most correlated to the sale price with a correlation score of 0.79. The second one is another quality variable, `ExterQual` which evaluates the quality of the material on the exterior, with a score of 0.73. The third one is the `GrLivArea` variable, which is the above ground living area in square feet, with a score of 0.71. We can notice that the other good correlation scores are either linked to area (`GarageArea`, `GarageCars` (the greater the area of the garage, the greater the number of cars we can park), `X1stFlrSF`, `TotalBsmtSF`), or linked to quality (`KitchenQual`, `BsmtQual`, `FireplaceQual`). We may also add the `FullBath` variable, that counts the number of full bathrooms above grade, and has a correlation score of 0.56.

As we can see on the plot, and as expected, some variables are strongly correlated which will cause a problem of collinearity. As described above, variables measuring areas are strongly correlated to each other and the same thing occurs for variables measuring quality. Therefore, we will only select the most important variables for our model, like `OverallQual` and `GrLivArea`. Another solution is to do a principal component analysis that will enable us to extract principal components that express most of the variability of the dataset.

To do the PCA, we prefer continuous numerical variables. We could transform the categorical variables into dummy variables, but we will get too many binary variables compared to the continuous variables, and we won't get useful results. We will be careful to center and scale the data as we have different kinds of variables. We do not need the `Id` column (1st column), and we don't want the outcome `SalePrice` (54th column in `train_num`) in the PCA.

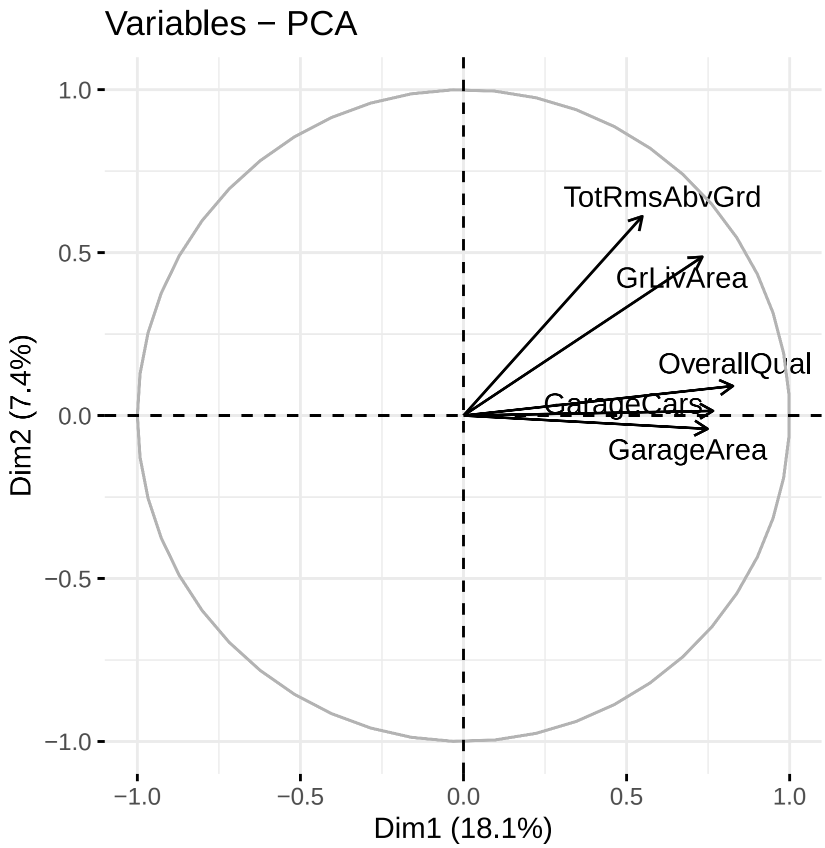
```r
# PCA

pca_num <- prcomp(train_num[, -c(1, 54)], center = TRUE, scale. = TRUE)

summary(pca_num)
```

```
## Importance of components:
```

```
##                           PC1     PC2     PC3     PC4     PC5     PC6     PC7
## Standard deviation     3.0663 1.95660 1.71434 1.53210 1.43716 1.41703 1.38452
## Proportion of Variance 0.1808 0.07362 0.05652 0.04514 0.03972 0.03861 0.03686
## Cumulative Proportion  0.1808 0.25444 0.31096 0.35610 0.39582 0.43443 0.47129
##                            PC8     PC9    PC10    PC11   PC12    PC13    PC14
## Standard deviation     1.29837 1.21125 1.18748 1.10447 1.0793 1.04589 1.04034
## Proportion of Variance 0.03242 0.02821 0.02712 0.02346 0.0224 0.02104 0.02081
## Cumulative Proportion  0.50371 0.53193 0.55904 0.58250 0.6049 0.62594 0.64675
##                           PC15    PC16    PC17    PC18    PC19    PC20    PC21
## Standard deviation     1.02553 1.01526 1.00599 0.99566 0.95788 0.94653 0.93628
## Proportion of Variance 0.02023 0.01982 0.01946 0.01906 0.01764 0.01723 0.01686
## Cumulative Proportion  0.66698 0.68680 0.70626 0.72533 0.74297 0.76020 0.77706
##                           PC22    PC23    PC24    PC25    PC26    PC27    PC28
## Standard deviation     0.91155 0.90591 0.87277 0.85896 0.81228 0.79833 0.78750
## Proportion of Variance 0.01598 0.01578 0.01465 0.01419 0.01269 0.01226 0.01193
## Cumulative Proportion  0.79304 0.80882 0.82347 0.83766 0.85035 0.86260 0.87453
##                           PC29    PC30    PC31   PC32    PC33    PC34    PC35
## Standard deviation     0.76041 0.74818 0.73279 0.7104 0.70007 0.67783 0.62605
## Proportion of Variance 0.01112 0.01076 0.01033 0.0097 0.00943 0.00884 0.00754
## Cumulative Proportion  0.88565 0.89641 0.90674 0.9164 0.92587 0.93471 0.94224
##                           PC36    PC37    PC38    PC39    PC40    PC41    PC42
## Standard deviation     0.61002 0.59057 0.55334 0.52653 0.51222 0.49222 0.46523
## Proportion of Variance 0.00716 0.00671 0.00589 0.00533 0.00505 0.00466 0.00416
## Cumulative Proportion  0.94940 0.95611 0.96200 0.96733 0.97237 0.97703 0.98119
##                           PC43    PC44   PC45    PC46    PC47    PC48    PC49
## Standard deviation     0.44382 0.41904 0.4015 0.33534 0.33230 0.31008 0.29305
## Proportion of Variance 0.00379 0.00338 0.0031 0.00216 0.00212 0.00185 0.00165
## Cumulative Proportion  0.98498 0.98836 0.9915 0.99362 0.99574 0.99759 0.99925
##                           PC50      PC51      PC52
## Standard deviation     0.19810 1.397e-15 1.362e-15
## Proportion of Variance 0.00075 0.000e+00 0.000e+00
## Cumulative Proportion  1.00000 1.000e+00 1.000e+00
```

When we check the summary, we can see that the first two components explain only 25% of the variance, and that we need more components to encapsulate more information. We will use the `factoextra` package to visualize the results.

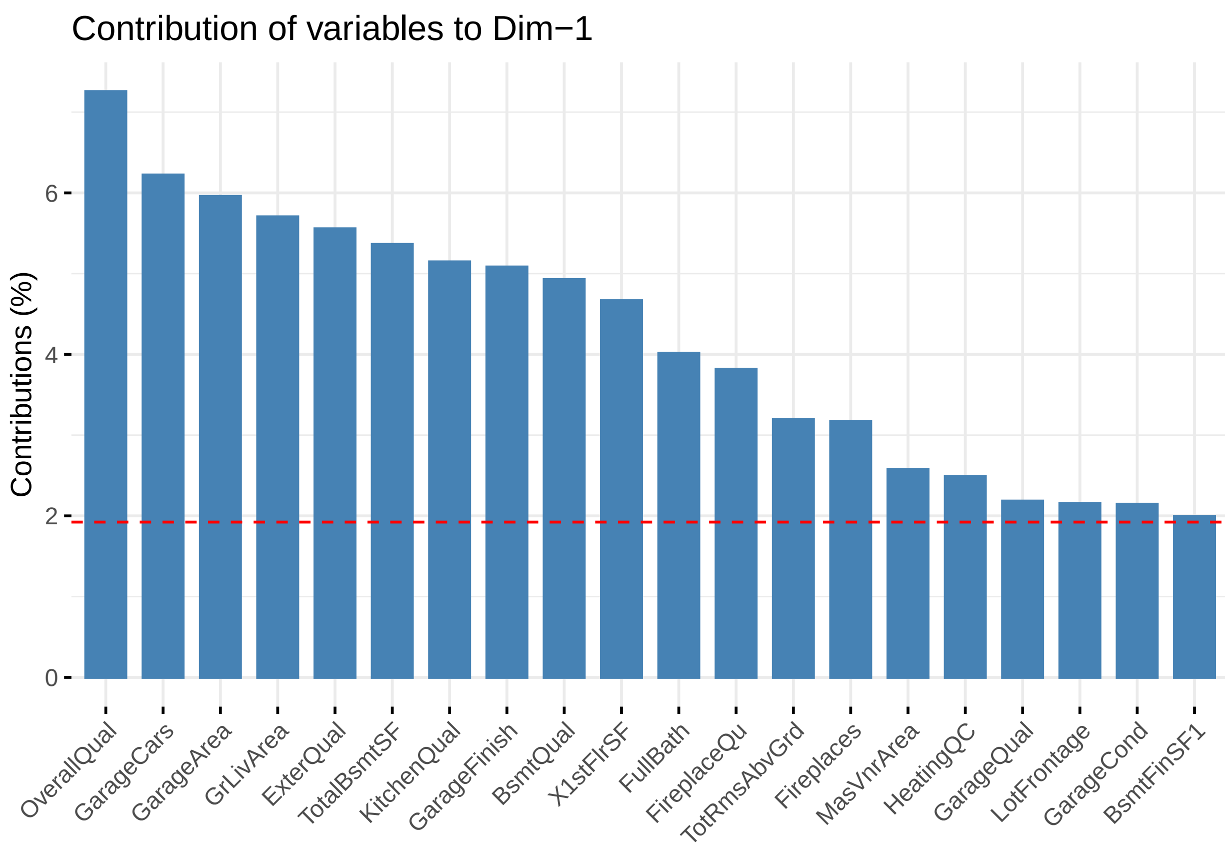
```r
# Plotting PCA results
library(factoextra)

fviz_pca_var(pca_num, select.var = list(contrib = 5), repel = TRUE)
```



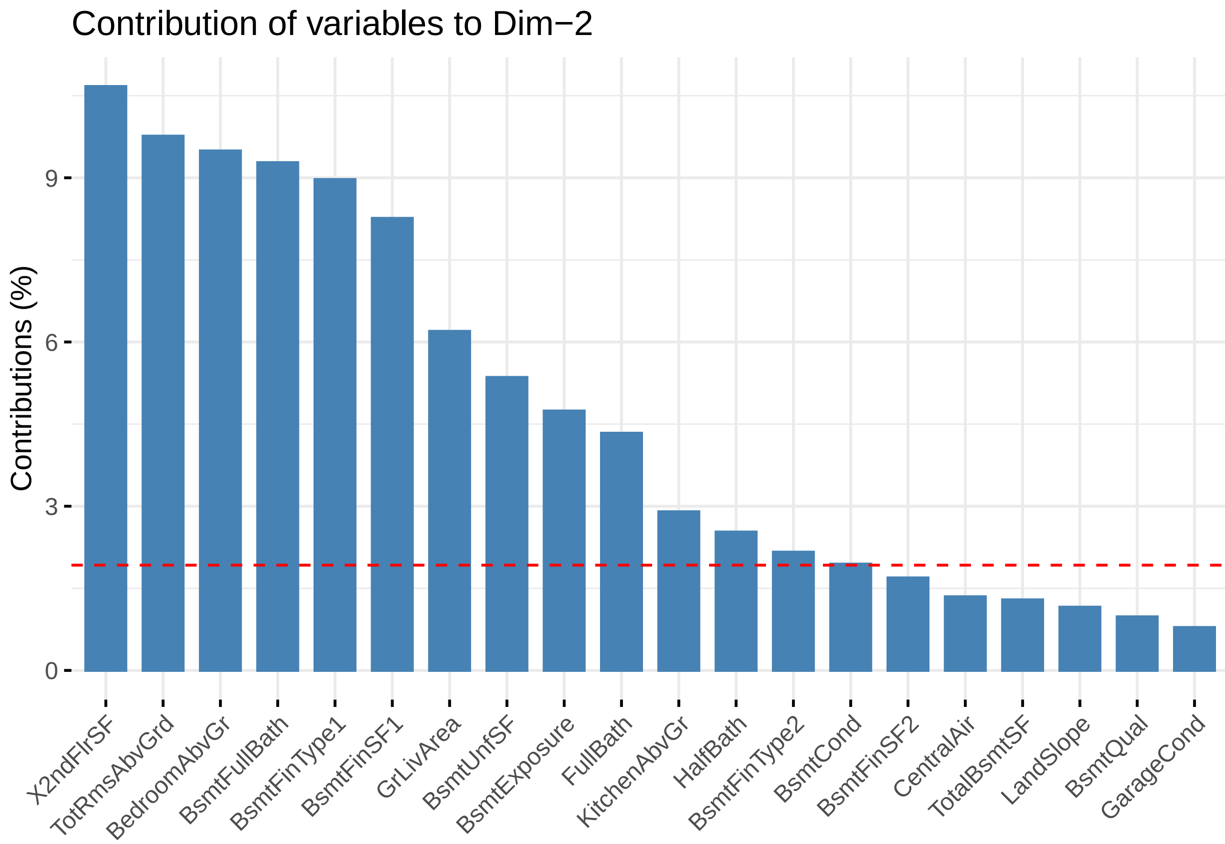
Here we can see the five most contributing variables to the first two components : `GrLivArea`, `OverallQual`, `TotRmsAbvGrd`, `GarageCars` and `GarageArea`. This confirms what we saw in the previous plots. It is also not a surprise to see that the two "garage" variables are almost on the same axis. We've already discussed the importance of `GrLivArea` and `OverallQual`.

We can also check which variables contribute the most to each component. Here we will check contributions of 20 variables for PC1 in the first plot, and PC2 in the second one with the following code.

```
# Checking contribution of variables for PC1 and PC2 :
# PC1

fviz_contrib(pca_num, choice = "var", axes = 1, top = 20)
```

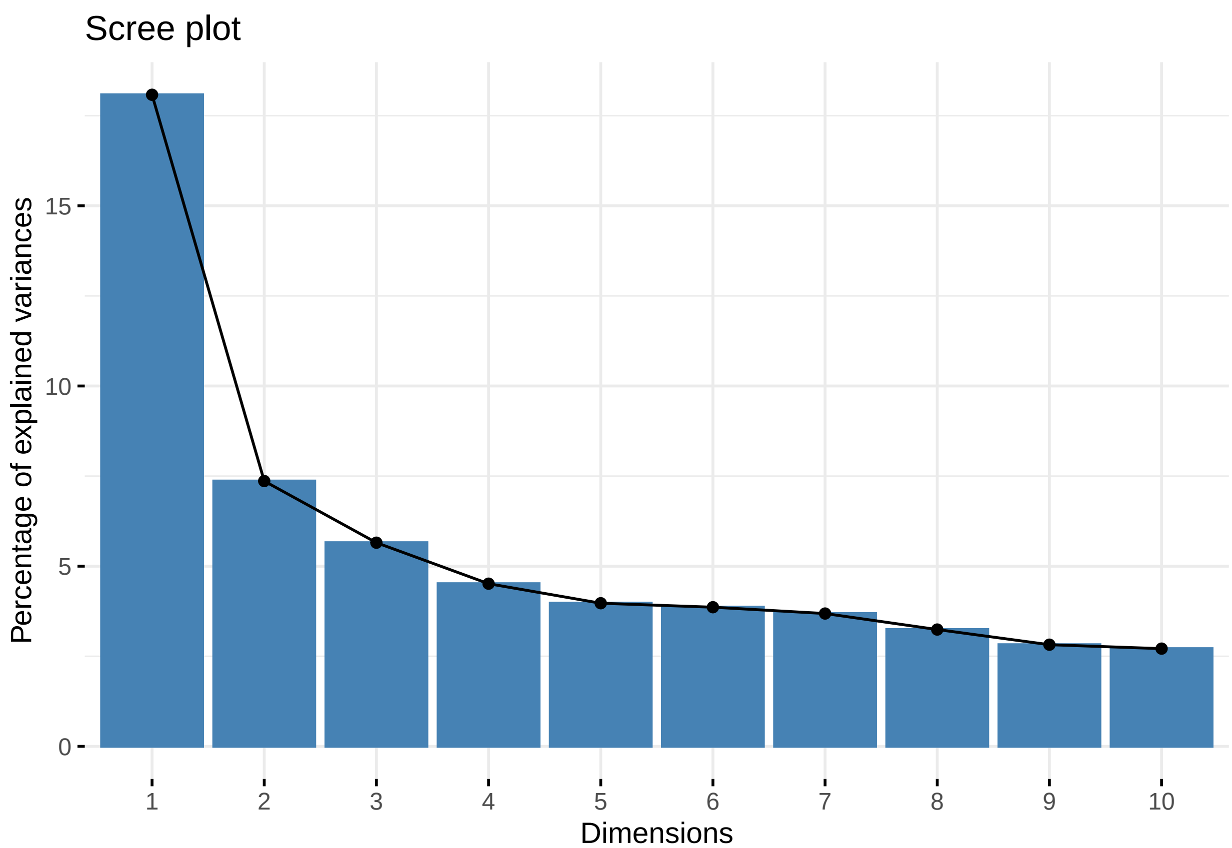



```r
# PC2

fviz_contrib(pca_num, choice = "var", axes = 2, top = 20)
```



Again, we find "quality" and "area" variables we've talked about earlier. But as we said, PC1 and PC2 explain only 25% of the variance. We need more components, but how many ? The scree test and its "elbow" criterion with the following screeplot cannot help us.

```
# Screeplot to visualize the "elbow" to determine the number of components

fviz_screeplot(pca_num)
```



As we can see, the "elbow" seems to level off at the third dimensions, so if we were to follow this criterion, we would retain only the first two components (and only 25% of the explained variance !).

The Kaiser-Guttman rule tells us we must retain only components whose eigenvalues are above 1. We can get them with the following code.

```
# Applying Kaiser-Guttman rule to determine the number of components (eigenvalue > 1) :

get_eigenvalue(pca_num) %>% filter(eigenvalue > 1)
```

```
##    eigenvalue variance.percent cumulative.variance.percent
## 1    9.402419        18.081576                    18.08158
## 2    3.828301         7.362118                    25.44369
## 3    2.938970         5.651866                    31.09556
## 4    2.347320         4.514076                    35.60964
## 5    2.065424         3.971969                    39.58160
## 6    2.007962         3.861465                    43.44307
## 7    1.916887         3.686321                    47.12939
## 8    1.685761         3.241848                    50.37124
## 9    1.467126         2.821396                    53.19263
## 10   1.410108         2.711747                    55.90438
## 11   1.219862         2.345888                    58.25027
## 12   1.164886         2.240166                    60.49043
## 13   1.093876         2.103608                    62.59404
## 14   1.082315         2.081374                    64.67542
## 15   1.051711         2.022521                    66.69794
```

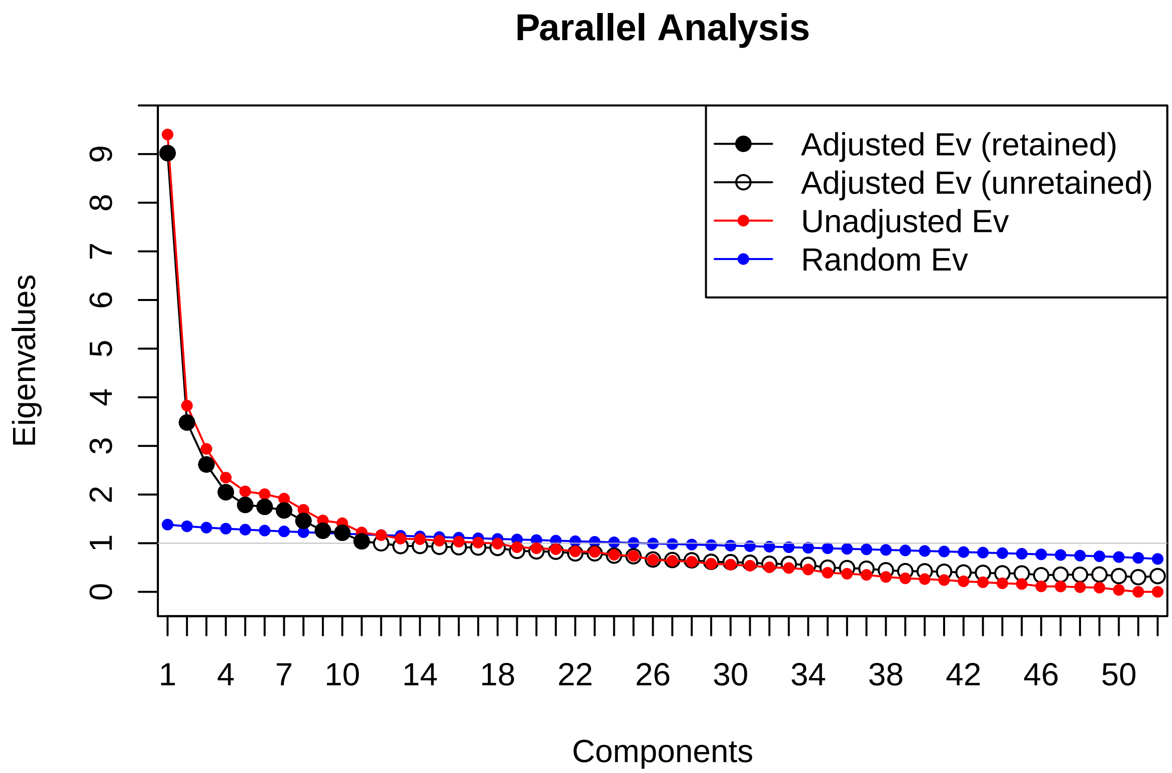
```
## 16   1.030758         1.982227                    68.68017
## 17   1.012025         1.946203                    70.62637
```

Each row represents one dimension, so according to these results, we should retain 17 components. The cumulative variance percentage is now 70%.

Another method to determine the number of components is the parallel analysis, which compares the eigenvalues generated from the original data to the eigenvalues of a Monte-Carlo simulated matrix created from a random data of the same size. As it generates random data, we must set the seed.

```
# Parallel analysis to determine the number of components to retain
# Parallel analysis
paran_output <- paran(train_num[,-c(1, 54)], seed = 69, graph = TRUE)
```

```
##
## Using eigendecomposition of correlation matrix.
## Computing: 10%  20%  30%  40%  50%  60%  70%  80%  90%  100%
##
##
## Results of Horn's Parallel Analysis for component retention
## 1560 iterations, using the mean estimate
##
## --------------------------------------------------
## Component   Adjusted    Unadjusted    Estimated
##             Eigenvalue  Eigenvalue    Bias
## --------------------------------------------------
## 1           9.020369    9.402419      0.382049
## 2           3.481147    3.828301      0.347154
## 3           2.618555    2.938970      0.320414
## 4           2.048460    2.347319      0.298858
## 5           1.787021    2.065423      0.278402
## 6           1.748022    2.007961      0.259939
## 7           1.674288    1.916886      0.242597
## 8           1.459607    1.685761      0.226154
## 9           1.256516    1.467125      0.210609
## 10          1.214454    1.410108      0.195654
## 11          1.038943    1.219861      0.180918
## --------------------------------------------------
##
## Adjusted eigenvalues > 1 indicate dimensions to retain.
## (11 components retained)
```




```
# Number of components to retain
paran_output$Retained
```

```
## [1] 11
```

According to this parallel analysis, we should retain 11 components (which explain 58% of the variance according to PCA). So how many components should we retain ? 11 or 17 ? To answer this question, we will train two more datasets : one containing the first 11 components, and another one containing 17 components. We will then compare the RMSE of each train set.

Along this Exploratory Data Analysis, we have used different plots and have found some variables seem to be more important than others. That is to say, the `SalePrice` outcome and some of the variables follow a bivariate normal distribution which suggests they are correlated and therefore, the conditional expectation is given by the regression line. Thus, we will need linear models to train the dataset, or models that can solve regression problems.

The PCA and the parallel analysis has just confirmed what we have seen on the plots, and we know that we can greatly reduce the number of dimensions of the dataset. The `caret` package has a `preProcess` argument that enables PCA. It will be useful when training our model with linear regression. I also want to know which number of components predict the best. So I will extract the first 17 principal components and create two more train sets in the next part : one with 11 components, as suggested by the parallel analysis, and another one with 17 components, as suggested by the Kaiser-Guttman rule. I'll also add the `SalePrice` column.

```
# Extracting principal components :

train_pc <- pca_num$x[, 1:17] %>% as.data.frame()

# Adding the outcome :

train_pc <- train_pc %>% cbind(train$SalePrice) %>% rename(SalePrice = `train$SalePrice` )
```

In the next part, I will have 3 train sets : one with the original values, and the two others with principal components (11 and 17). I will compare the RMSE of the 3 sets before building the final model. For better readability, the training part will only focus on the train set with the original variables. I will only show the results of the different sets at the end of the training part.

It is now time to create and train our algorithm.

Training the models

We have found during EDA that we are facing a regression problem. We will therefore use linear models. We will also use tree-based models. First, we need to create a train set and a test set.

Data partition and cross-validation plan

We will now create a new train set and a test set with the `CreateDataPartition` function from the `caret` package. It will randomly order the dataset. This ensures that the training set and test set are both random samples and that any biases in the ordering of the dataset are not retained in the samples. We need to set the seed to get constant results. As we saw at the beginning, the `train` and `validation` has almost the same number of observations, i.e. around 1460. We will split the `train` set with 80% for the new `train` set and 20% for the `test` set. It should be enough to build a model for the `validation` set. Going beyond 80% may give better results in the `test` set, but could lead to overfitting. On the other hand, if we split below 70%, the model may not have enough data.

```
# Splitting in train and test sets

set.seed(69, sample.kind = "Rounding")
```

```
## Warning in set.seed(69, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
test_index <- createDataPartition(train$SalePrice, times = 1, p = 0.8, list = FALSE)

train_set <- train[test_index,]
test_set <- train[-test_index,]
```

As explained earlier, I want to check the predictions of the principal components. So I create two more train sets : one with 11 principal components, the second one with 17 components. The RMSE for both of these sets will be shown at the end of the training. But for each model shown below, we will focus on the main train set with the original variables.

```
# Train and test sets with 11 components :

train_11 <- train_pc[, 1:11] %>% cbind(train$SalePrice) %>% rename(SalePrice = `train$SalePrice` )
```

```r
test_11 <- train_pc[, 1:11] %>% cbind(train$SalePrice) %>% rename(SalePrice = `train$SalePrice` )

train_11 <- train_11[test_index, ]
test_11 <- test_11[-test_index, ]

# Train and test sets with 17 components :

train_17 <- train_pc[test_index,]
test_17 <- train_pc[-test_index,]
```

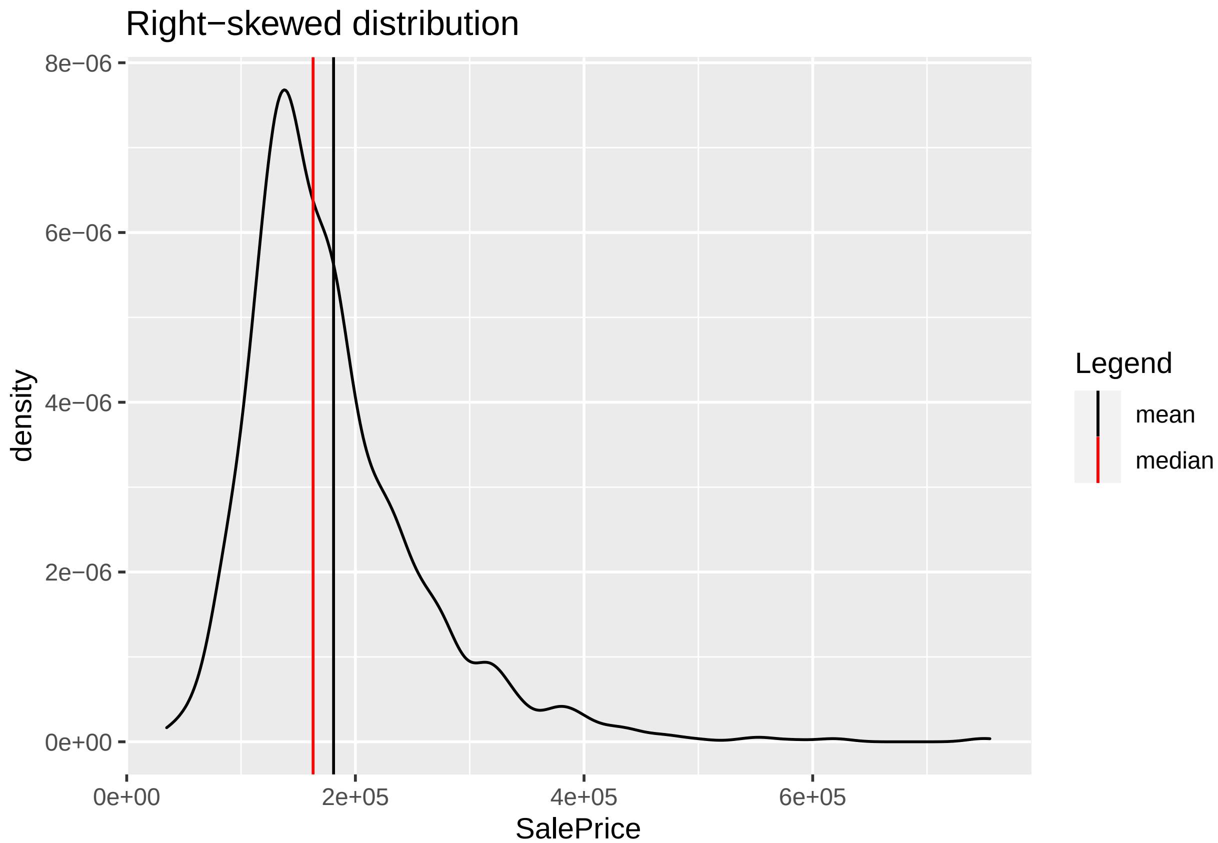
We will then create a cross-validation plan to train our models. As there is not a lot of observations, it is indeed necessary. We still use the `caret` package and its `trainControl` function. We will use a 10-fold cross-validation.

```r
# Creating a cross-validation plan

cv_plan <- trainControl(method = "cv", number = 10)
```

Transformation of the outcome 'SalePrice' with log function

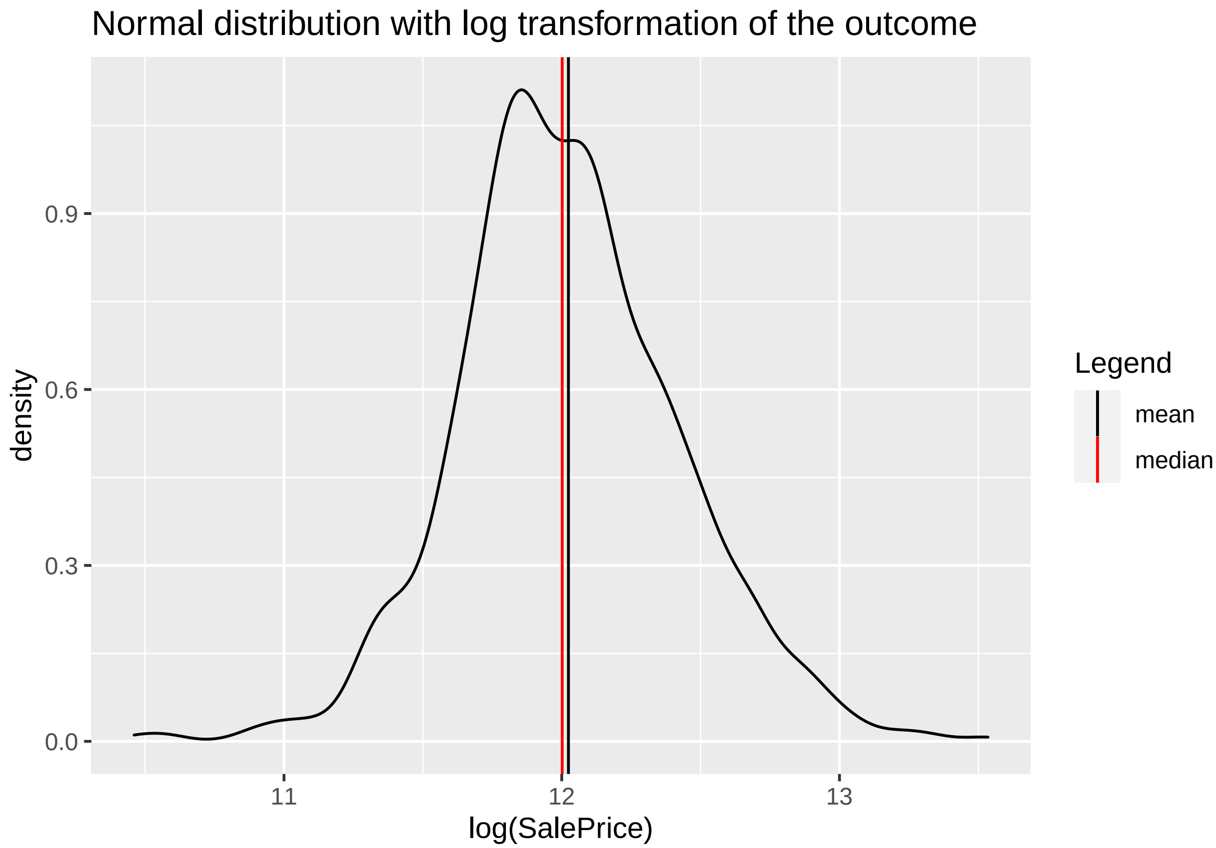
When we first checked the distribution of the `SalePrice` outcome, we could see that its distribution was right-skewed. We plot it one more time with the median and the mean.

```r
# Distribution of 'SalePrice' with mean and median

train %>% ggplot(aes(SalePrice)) +
  geom_density() +
  geom_vline(aes(xintercept = mean(SalePrice), color = "mean")) +
  geom_vline(aes(xintercept = median(SalePrice), color = "median")) +
  scale_color_manual(name = "Legend", values = c(mean = "black", median = "red")) +
  ggtitle("Right-skewed distribution")
```



If we perform a regression directly, the model could overpredict typical values. If we transform the outcome with a log function, we get a classic normal distribution : we can see that the mean and median are closer, and the dynamic of the predictions will be more reasonable, as plotted below.

```
# Log transformation of 'SalePrice'

train %>% ggplot(aes(log(SalePrice))) +
  geom_density() +
  geom_vline(aes(xintercept = mean(log(SalePrice)), color = "mean")) +
  geom_vline(aes(xintercept = median(log(SalePrice)), color = "median")) +
  scale_color_manual(name = "Legend", values = c(mean = "black", median = "red")) +
  ggtitle("Normal distribution with log transformation of the outcome")
```



We will therefore use `log(SalePrice)` as outcome to predict in our models. As an example, we can check the best RMSE between `SalePrice` and `log(SalePrice)` using a linear model with just two of the main variables, i.e. `GrLivArea` and `OverallQual`.

```
# Linear model with 'SalePrice' as outcome to predict :

model_lm <- lm(SalePrice ~ GrLivArea * OverallQual,
                         train_set)

pred_lm <- predict(model_lm, test_set)

RMSE(log(pred_lm), log(test_set$SalePrice))
```

```
## [1] 0.2221613
```

```
# Linear model with 'log(SalePrice)' as outcome to predict :

model_lm_log <- lm(log(SalePrice) ~ GrLivArea * OverallQual,
                             train_set)

pred_lm_log <- predict(model_lm_log, test_set)

RMSE(pred_lm_log, log(test_set$SalePrice))
```

```
## [1] 0.2104293
```

The RMSE is better when we transform the outcome in our model with the `log` function rather than transforming our predictions afterwards.

First model : Linear regression

Let's first train a linear regression using `train` of the caret package. We will add the cross-validation, and also a preprocess step that will :

- identify predictors with near zero variance that won't be useful ;
- center predictors ;
- scale predictors ;
- and finally perform a PCA.

During Exploratory Data Analysis, we saw that PCA could reduce the number of predictors between 11 and 17 components. So this preprocess will reduce the number of dimensions and get rid of collinearity. We do this with the following piece of code.

```
# Training a linear model

model_lm <- train(log(SalePrice) ~ .,
                  method = "lm",
                  trControl = cv_plan,
                  preProcess = c("nzv", "center", "scale", "pca"),
                  data = train_set)
```

We now make predictions and calculate the RMSE of the logs. As `log(SalePrice)` is already in our model, the predictions will also be logs. So we must add the `log` function to the `SalePrice` of the test set in the `RMSE` function from the `caret` package.

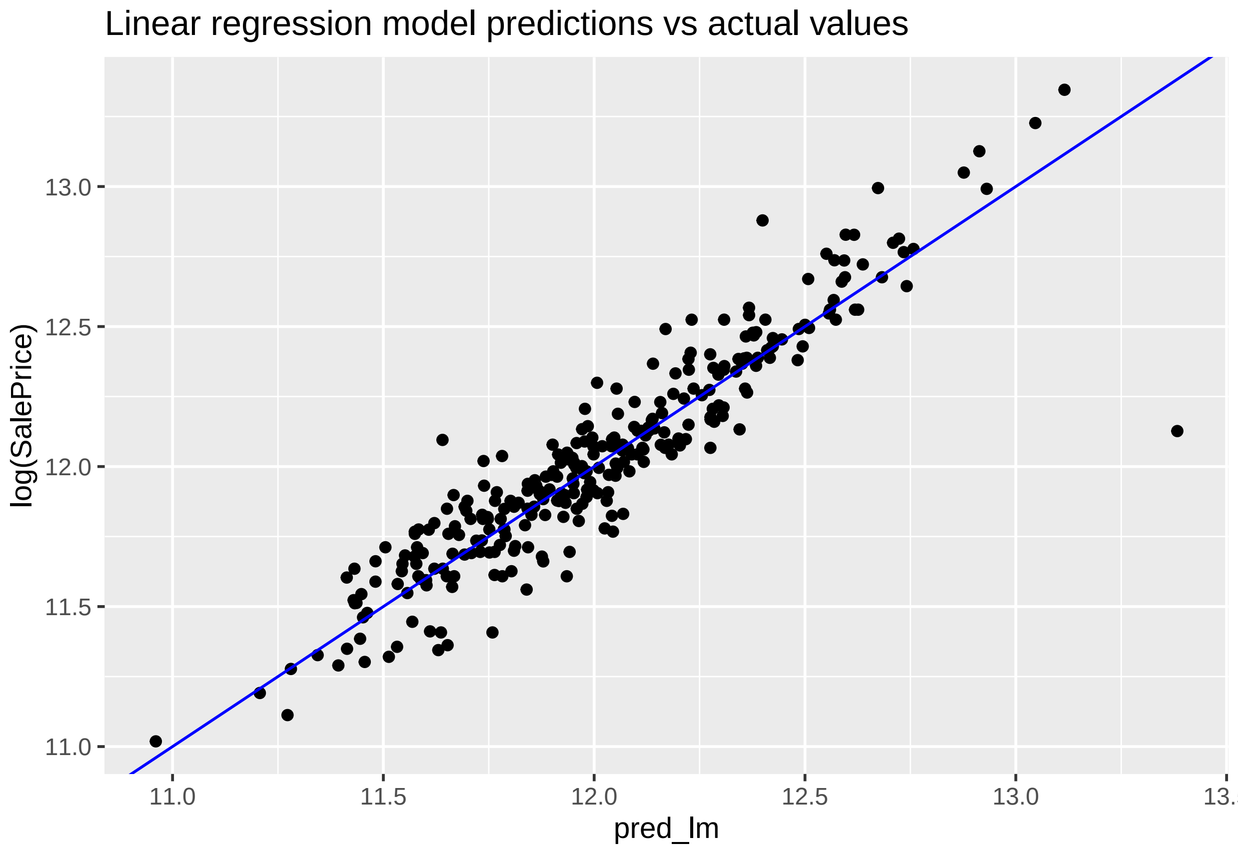
```
# Predicting results

pred_lm <- predict(model_lm, test_set)

(rmse_lm <- RMSE(log(test_set$SalePrice), pred_lm))
```

```
## [1] 0.1468603
```

The RMSE is 0.146. There is a lot to do to reach the best RMSE of the Kaggle contest.

We can visualize our predictions with the following plot.

```
# Plot of predictions

test_set %>% cbind(pred_lm) %>%
  ggplot(aes(pred_lm, log(SalePrice))) +
  geom_point() +
  geom_abline(color = "blue") +
  ggtitle("Linear regression model predictions vs actual values")
```



Second model : GLMnet

GLMnet is an extension of the linear regression, but it helps dealing with collinearity and small datasets, and penalizes number of non-zero-coefficients (also known as "lasso regression") and penalizes absolute magnitude of coefficients (also known as "ridge regression") in order to find a simpler model. There are two main parameters that we must tune for this model :

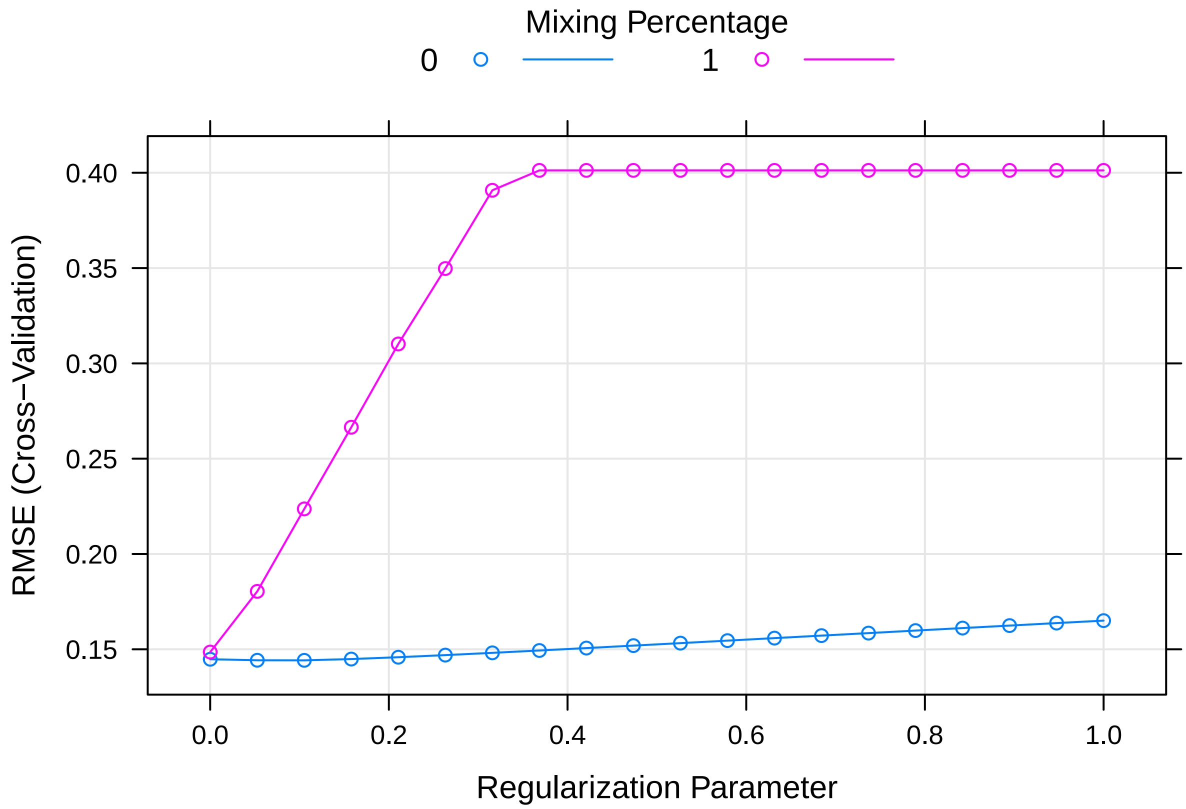
- alpha : "0" for a pure ridge regression, and "1" for a pure lasso regression ;
- lambda : this is where we tune the size of the penalty.

We tune this two parameters inside the `tuneGrid` argument. We must set seed to get the same results. Tuning parameters are almost the same as the linear regression model, except that we don't run PCA this time.

```
# Training a GLMnet model
set.seed(69, sample.kind = "Rounding")

model_glmnet <- train(log(SalePrice) ~ .,
                      train_set,
                      tuneGrid = expand.grid(alpha = 0:1,
                                             lambda = seq(0.0001, 1, length = 20)),
                      method = "glmnet",
                      trControl = cv_plan,
                      preProcess = c("nzv", "center", "scale"))
```

Let's see now which is the best tune for this model.

```r
# Plot of ridge and lasso parameters

plot(model_glmnet)
```



It seems that a low parameter for ridge regression (the "0" blue line on the plot) gives the lowest RMSE. Let's see the predictions of this model with the test set.

```r
# Predictions of GLMnet model :

pred_glmnet <- predict(model_glmnet, test_set)

(rmse_glmnet <- RMSE(log(test_set$SalePrice), pred_glmnet))
```

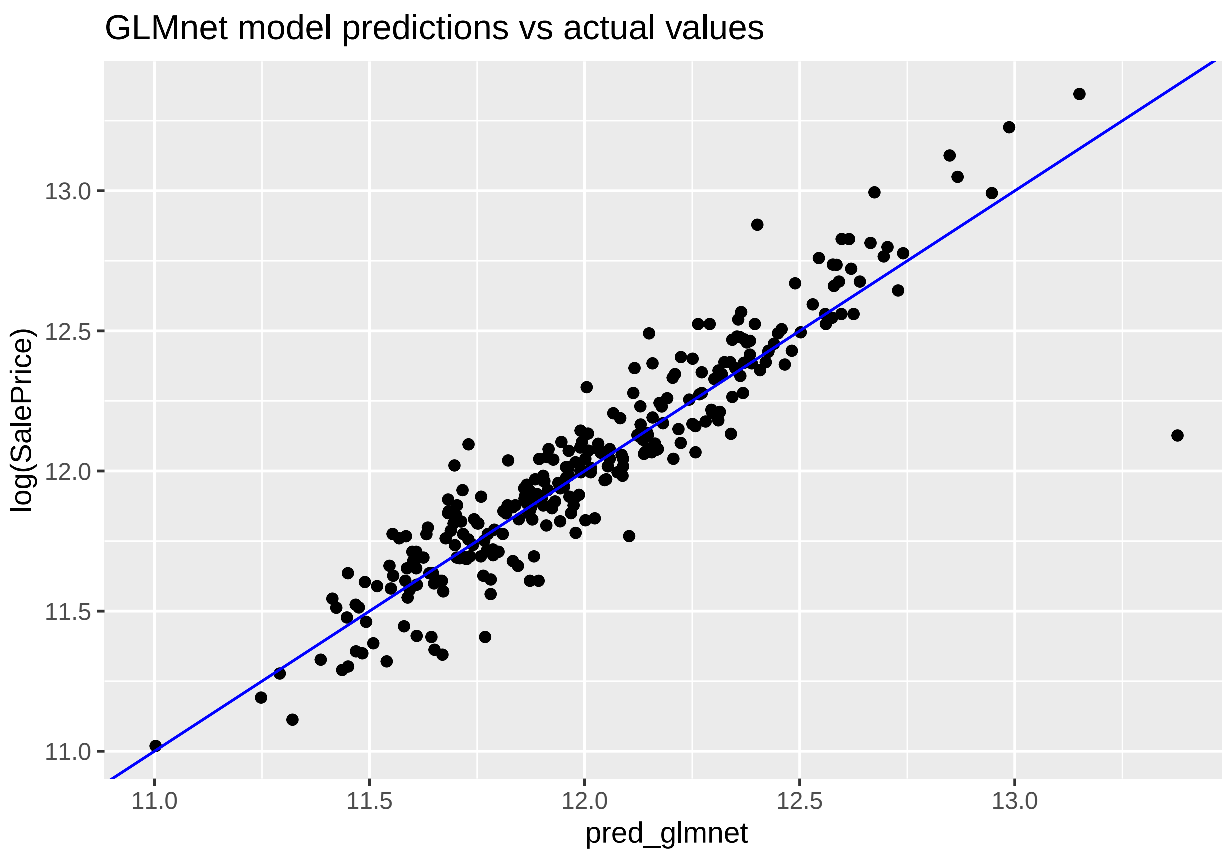
```
## [1] 0.1434452
```

This time the RMSE is 0.143. It's slightly better than the first model, but we're still far from the 0.00044 goal.

Let's check again the predictions against actual values.

```r
# Plot of predictions

test_set %>% cbind(pred_glmnet) %>%
```

```r
ggplot(aes(pred_glmnet, log(SalePrice))) +
  geom_point() +
  geom_abline(color = "blue") +
  ggtitle("GLMnet model predictions vs actual values")
```



Like the first model, the GLMnet model seems to be a good fit. But it is still not enough. Let's keep going.

Third model : randomForest

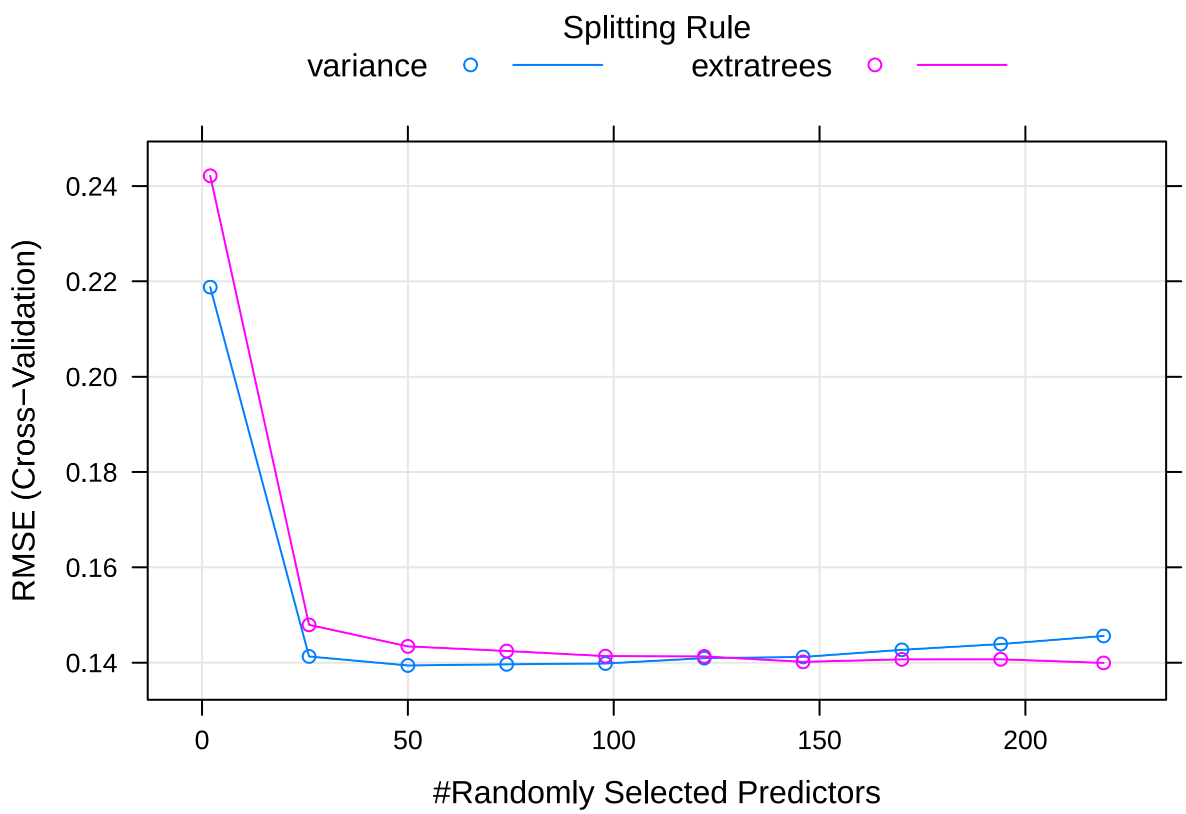
Using randomForest seems natural and intuitive for this dataset. As said in the EDA, when we look for a house, we think about area and location at first. We know that if the area of the house exceeds a certain value, the price will be higher. In the same way, if a house is located in an attractive place, it will be more expensive. RandomForest "thinks" the same way. Instead of using the `randomForest` package, we will use the `method = "ranger"` of the `caret` package. It is said to be faster than the original randomForest. The main tuning parameters this time are `mtry`, which is the number of randomly selected variables at each split, and `tuneLength`. The higher `mtry`, the less random it is, and better the results. However, it is hard to know in advance which value of `mtry` is best. The `tuneLength` parameter is here to help us. Its default value is 3, which means that it tries 3 different models, but we will set it to 10. It is however a bit longer to run.

```r
# Training a randomForest model

model_rf <- train(log(SalePrice) ~ .,
                  tuneLength = 10,
                  data = train_set,
```

```
                  method = "ranger",
                  trControl = cv_plan)
```

Now, let's see the results.

```
# Plot of the tuning of 10 randomForest models

plot(model_rf)
```




```
# Best tune for the randomForest model :

model_rf$bestTune
```

```
##   mtry splitrule min.node.size
## 5   50  variance             5
```

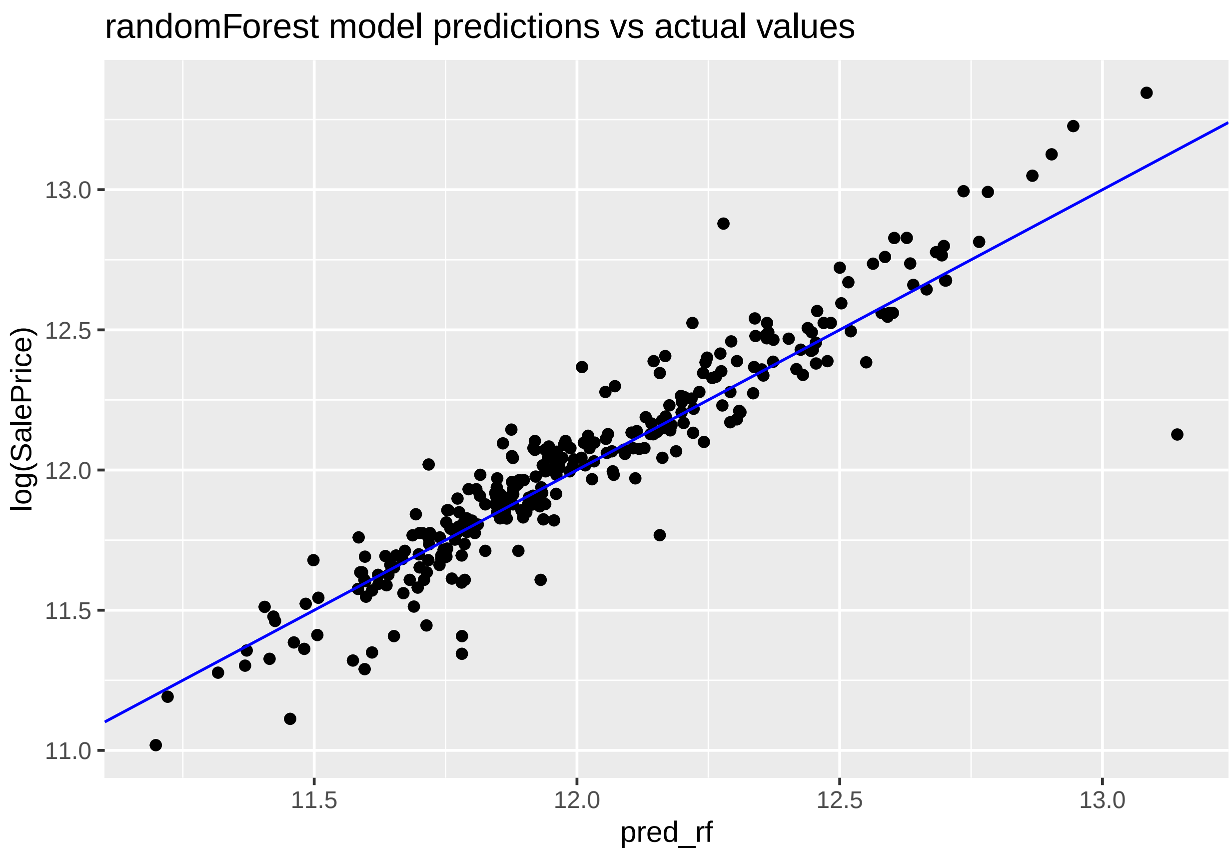
According to the plot and the best tune of the model, the lowest RMSE is reached with 50 randomly selected predictors (`mtry`) using variance as splitting rule. Let's check the predictions.

```
# Predictions of the randomForest model :

pred_rf <- predict(model_rf, test_set)

(rmse_rf <- RMSE(log(test_set$SalePrice), pred_rf))
```

```
## [1] 0.1352571
```

The RMSE is 0.135. This is better than our two previous models. Let's check again our predictions against actual values.

```
# Plot of the randomForest predictions

test_set %>% cbind(pred_rf) %>%
  ggplot(aes(pred_rf, log(SalePrice))) +
  geom_point() +
  geom_abline(color = "blue") +
  ggtitle("randomForest model predictions vs actual values")
```

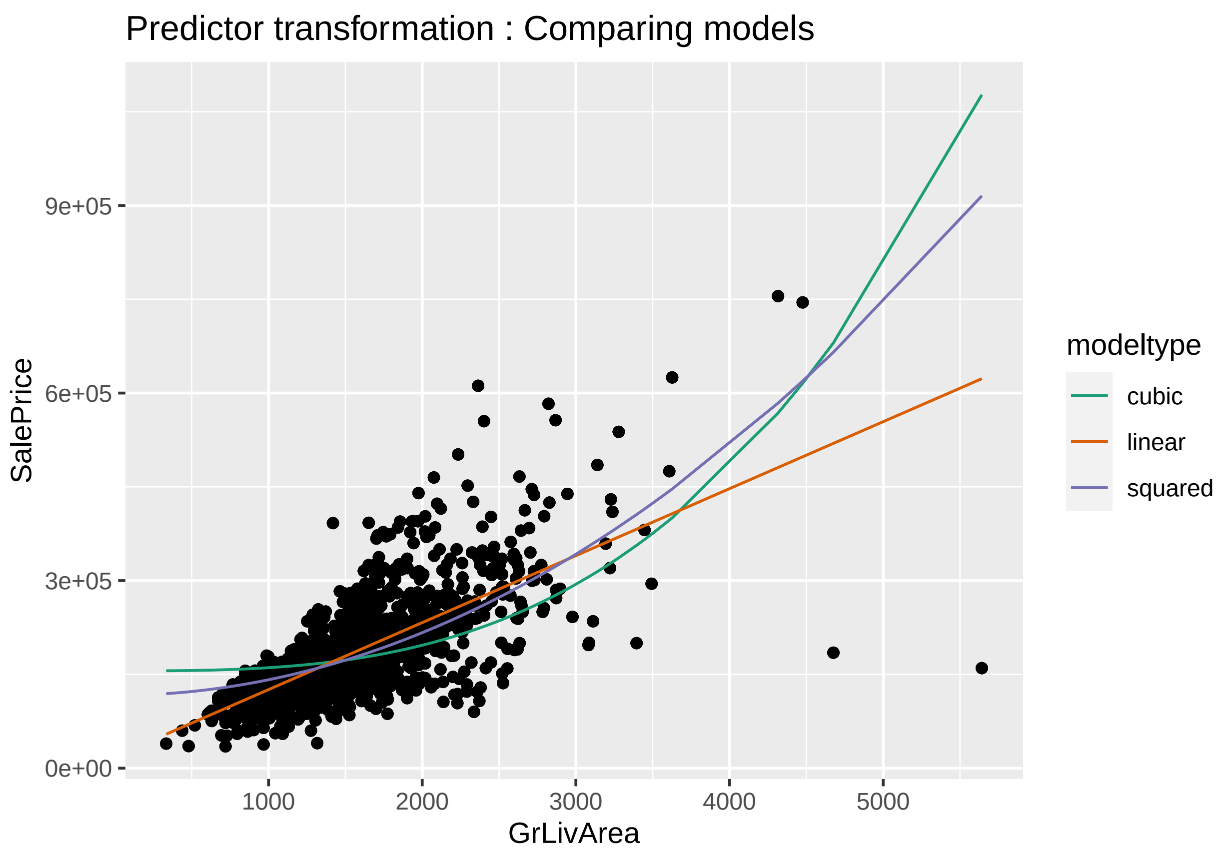


It seems the biggest errors are for the cheapest houses and the most expensive houses. It seems there is also the same outlier we have been seeing on the two previous models.

Fourth model : Generalized Additive Model (GAM)

The GAM is another extension of the linear model. The advantage of GAM is that it can deal with non-linear predictors. Sometimes, predictors that measure areas (like `GrLivArea`, `TotalBsmtSF`, or `LotArea` in the dataset) must be transformed, i.e. squared or cubic, to fit a model that wants to predict a price (like in this dataset). In that kind of cases, observations follow a line along low values but sharply increases as area and prices get higher. We will better understand this with the next plot. We will take the `GrLivArea` as an example and plot it against `SalePrice`. We calculate three linear models : one without transformation of `GrLivArea`, one with `GrLivArea` squared, and the last one with cubic transformation.

```r
# Transformation of predictor : an example with 'GrLivArea' against 'SalePrice'

# Transforming predictor : squared and cubic
fmla_sqr <- SalePrice ~ I(GrLivArea^2)

fmla_cub <- SalePrice ~ I(GrLivArea^3)

# Fitting a model of price as a function of squared area and cubic area
model_sqr <- lm(fmla_sqr, train)

model_cub <- lm(fmla_cub, train)

# Fitting a model of price as a linear function of 'GrLivArea'
model_lin <- lm(SalePrice ~ GrLivArea, train)


# Making predictions and comparing
train %>% mutate(linear = predict(model_lin), # predictions from linear model
         squared = predict(model_sqr),        # predictions from quadratic model
         cubic = predict(model_cub)) %>%      # predictions from cubic model
  gather(key = modeltype, value = pred, linear, squared, cubic) %>% # gather the predictions
  ggplot(aes(x = GrLivArea)) +
  geom_point(aes(y = SalePrice)) +  # actual prices
  geom_line(aes(y = pred, color = modeltype)) + # the predictions
  scale_color_brewer(palette = "Dark2") +
  ggtitle("Predictor transformation : Comparing models")
```



As we can see, the linear and cubic models seem to better follow higher values. We can calculate the RMSE with the following code.

```r
# Comparing RMSE of the three models :

train %>%
  mutate(linear = predict(model_lin),    # predictions from linear model
         squared = predict(model_sqr),   # predictions from quadratic model
         cubic = predict(model_cub)) %>%     # predictions from cubic model
  gather(key = modeltype, value = pred, linear, squared, cubic) %>%
  group_by(modeltype) %>%
  summarize(rmse = RMSE(log(SalePrice), log(pred)))
```

```
## # A tibble: 3 x 2
##   modeltype  rmse
##   <chr>     <dbl>
## 1 cubic     0.340
## 2 linear    0.276
## 3 squared   0.297
```

Unexpectedly, the linear model still gives the best RMSE here, but the quadratic model is close. This difference is maybe due to the two outliers we can see at the bottom on the right of the plot, and also due to the lack of observations with high prices.

This kind of predictors are called additive predictors, and can be specified in a generalized additive model with the `s()` function to specify that we want to use a "spline" to model the non-linear relationship between the outcome and the predictor. This is used for continuous variables as we've seen above.

I will still train a GAM, but this time I'll use the original package `gam`, instead of the `method = "gam"` of the `train` function in the `caret` package, because I got better results when I trained different models of GAM. I will also use the most relevant predictors we found during Exploratory Data Analysis (it returns an error with all predictors), and I will add a spline (`s()` function) to continuous variables, except `GrLivArea`, as we've just seen that a linear relationship gives a better RMSE. It is advised to use the spline with a continuous variable containing at least 10 continuous values. We could have used the spline for the `OverallQual` and `OverallCond` as there are 10 levels, but unique values are below 10, so we won't use it.

```
# Training a GAM

model_gam <- gam(log(SalePrice) ~ Neighborhood + OverallQual + OverallCond +
                   GrLivArea + GarageCars + s(GarageArea) + ExterQual + s(TotalBsmtSF) +
                   KitchenQual + FullBath + s(X1stFlrSF) + MSSubClass + MSZoning +
                   TotRmsAbvGrd + RoofStyle + SaleType + SaleCondition + Condition1,
                 family = gaussian,
                 train_set)
```

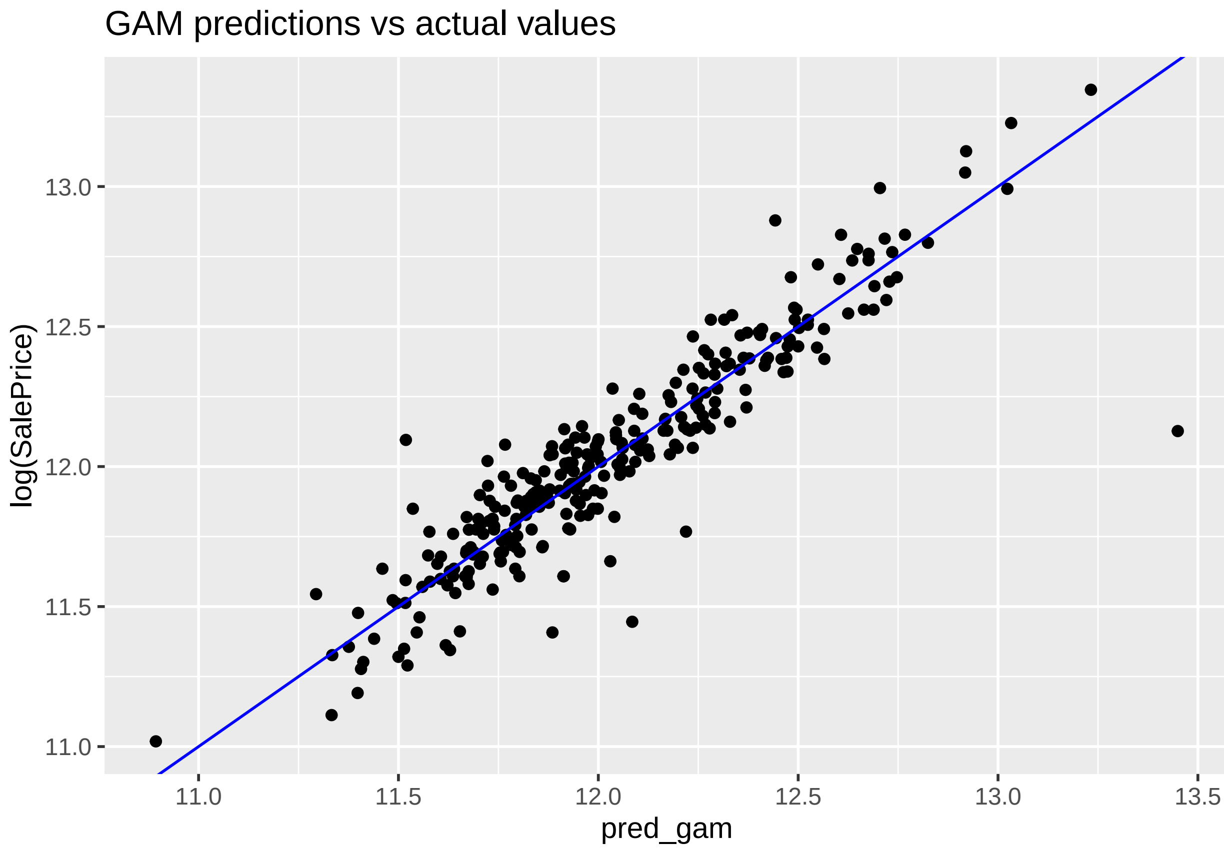
Let's check the RMSE of the predictions.

```
# Predictions of GAM :

pred_gam <- predict(model_gam, test_set)

# RMSE for GAM :

(rmse_gam <- RMSE(log(test_set$SalePrice), pred_gam))
```

```
## [1] 0.1536102
```

We get 0.153. The results of the four models are close to each other. Let's plot the predictions of this model.

```
# GAM predictions VS actual values plot :

test_set %>% cbind(pred_gam) %>%
  ggplot(aes(pred_gam, log(SalePrice))) +
  geom_point() +
  geom_abline(color = "blue") +
  ggtitle("GAM predictions vs actual values")
```



According to the visualization, the model seems somehow well fit, but there is still too much dispersion and the same outlier. We can notice that we get almost the same results than the other models with just a few well-selected predictors, those we found during Exploratory Data Analysis. When we look at the RMSE, it seems it will be difficult to reach the best RMSE of the Kaggle contest. But I don't give up and I will train a fifth model : XGBoost.

Fifth model : XGBoost

Except the fact that it sounds "cool", I will use an eXtreme Gradient Boosting model because it is a tree-based model that builds several models and incrementally improves its model from the residuals with the previous one and thus, learns from its errors contrary to randomForest which builds independent models each time. Up to now, randomForest has the best RMSE among our models, but it is still high, especially for extreme values (lowest and highest prices), so I want to try this model to see if it can learn what randomForest couldn't predict previously. If so, it could improve our performance.

The problem with XGBoost is that it can easily overfit. So we must be careful when tuning hyperparameters.

But first, we need to prepare the train set. The `XGBoost` model only accept matrix with numerical values. So we must dummify the categorical variables, i.e. we must convert each level of factors into binary variables. We will use the `vtreat` package to do that.

```
# Dummifying categorical variables of the train set (One-hot encoding)
library(vtreat)

# Defining categorical predictors to dummify
vars <- names(categ_data)
```

```r
# Creating the treatment plan
treatplan <- designTreatmentsZ(train_set, vars)
```

```
## [1] "vtreat 1.6.1 inspecting inputs Mon Oct 26 14:58:02 2020"
## [1] "designing treatments Mon Oct 26 14:58:02 2020"
## [1] " have initial level statistics Mon Oct 26 14:58:02 2020"
## [1] " scoring treatments Mon Oct 26 14:58:02 2020"
## [1] "have treatment plan Mon Oct 26 14:58:02 2020"
```

```r
# Checking the scoreFrame
scoreFrame <- treatplan %>%
  magrittr::use_series(scoreFrame) %>%
  dplyr::select(varName, origName, code)

# We only want the rows with code "lev"
newvars <- scoreFrame %>%
    filter(code == "lev") %>%
    magrittr::use_series(varName)

# Creating the treated training data
df_treat <- prepare(treatplan, train_set, varRestriction = newvars)

# Finally, we add the new binary variables with the numerical variables.
# We must not use the 'SalePrice' column.
train_treat <- train_set[, -c(1, 84)] %>% select_if(is.numeric) %>% cbind(df_treat)
```

We do the same thing for the test set.

```r
# Preparing categorical variables of the test set and converting them into binary variables
df_test_treat <- prepare(treatplan, test_set, varRestriction = newvars)

test_treat <- test_set[, -c(1, 84)] %>% select_if(is.numeric) %>% cbind(df_test_treat)
```

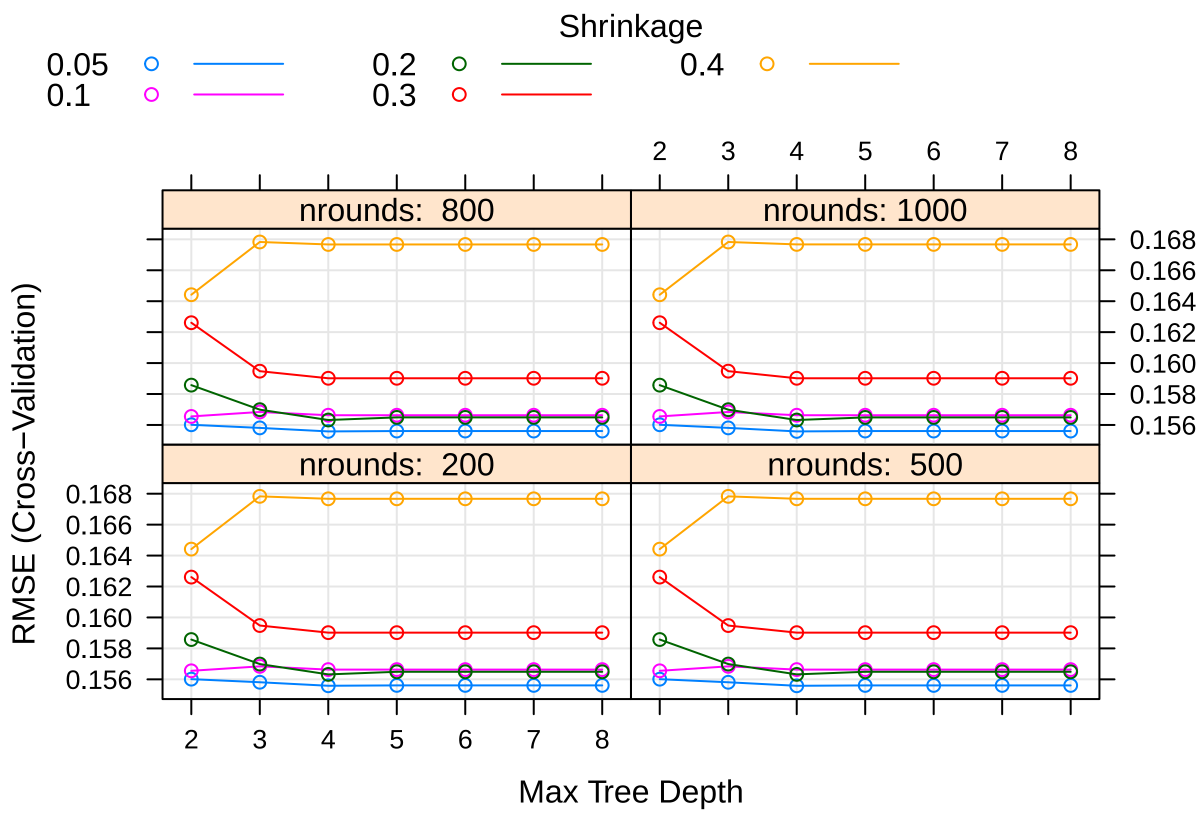
The `XGBoost` model has a lot of hyperparameters to tune. We will focus on the three main ones : the number of trees (`nrounds`), the depth of the trees (`max_depth`) and the learning rate (`eta`). To find the best tuning, we will use the `method = xgbTree` of the `caret` package. We will use the function `expand.grid()` in which we will enter the different parameters, as shown below.

```r
# Tuning hyperparmaters for XGBoost model :

xgb_grid <- expand.grid(nrounds = c(200, 500, 800, 1000),
                        max_depth = c(2,3,4,5,6,7,8), # default = 6
                        eta = c(0.05,0.1,0.2,0.3,0.4),# default = 0.3
                        gamma = 1,
                        colsample_bytree = 1, # default = 1
                        min_child_weight = 1, # default = 1
                        subsample = 1) # default = 1


set.seed(69, sample.kind = "Rounding") # We must set the seed to get the same results

# For tuning with the 'caret' package, we add the 'SalePrice' column to the 'train_treat' dataset.
```

```r
# Remember that 'SalePrice' outcome must not be part of the dataset
# when training with 'xgboost' function or it
# will give overoptimistic results.

train_treat_tune <- train_treat  %>% cbind(train_set$SalePrice) %>%
  rename(SalePrice = `train_set$SalePrice`)

# Be careful ! Running time for this code is around 40 minutes !
xgb_tune <-train(log(SalePrice) ~ .,
                 data = train_treat_tune,
                 method = "xgbTree",
                 trControl = cv_plan, # 10-fold cross-validation plan
                 tuneGrid = xgb_grid, # hyperparameters we wan to tune
                 verbose = FALSE,
                 metric = "RMSE",
                 nthread =3)
```

We can plot the results like this.

```r
# Plot of the XGBoost tuning

plot(xgb_tune)
```



We can find the best tuning with the code below.

```r
# XGBoost best tune

xgb_tune$bestTune
```

```
##    nrounds max_depth  eta gamma colsample_bytree min_child_weight subsample
## 11     800         4 0.05     1                1                1         1
```

Now let's train an XGBoost model with the best tuning found above.

```r
# Training an XGBoost model

model_xgb <- xgboost(data = as.matrix(train_treat), # training data as matrix without 'SalePrice'
                     label = log(train_set$SalePrice),  # we must use the original column of train_set
                     nrounds = 800,       # number of trees to build
                     objective = "reg:squarederror", # for regression
                     eta = 0.05,
                     max_depth = 4,
                     verbose = 0)  # silent
```

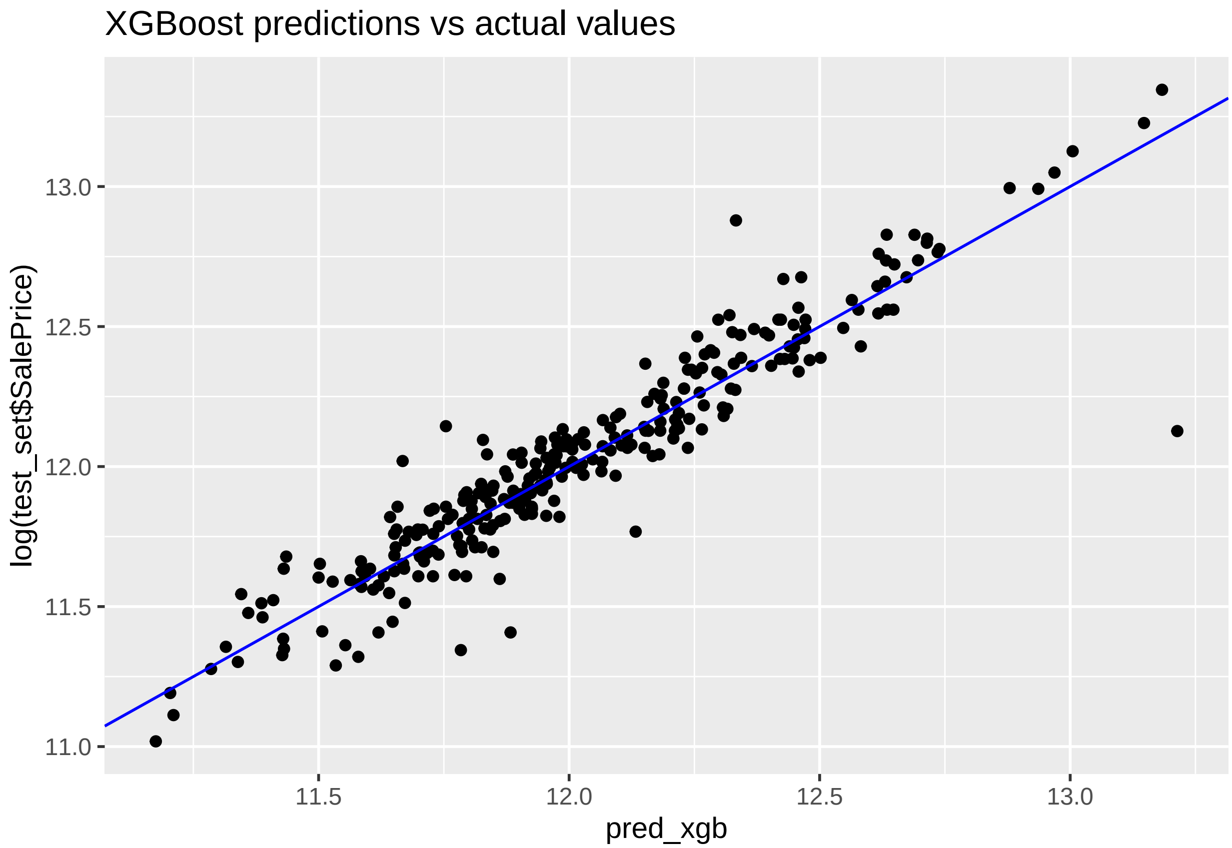
Let's see now the result.

```r
# Predictions of the XGBoost model

pred_xgb <- predict(model_xgb, as.matrix(test_treat))

(rmse_xgb <- RMSE(log(test_set$SalePrice), pred_xgb))
```

```
## [1] 0.1302853
```

This XGBoost model gives the best result. But the RMSE is still higher than the best one of the Kaggle contest. Let's visualize our predictions.

```r
# XGBoost predictions versus actual values

test_treat %>% cbind(pred_xgb) %>% ggplot(aes(pred_xgb, log(test_set$SalePrice))) +
  geom_point() +
  geom_abline(color = "blue") +
  ggtitle("XGBoost predictions vs actual values")
```



The outlier is still there, but the difference seems less important than the other models. Would an ensemble of the different models improve RMSE ? Before answering this question, Let's sum up and check the results of the different train sets. We will compare the results we found above and those of the train sets containing 11 and 17 principal components.

```r
# Sum up of results of the different train sets (original variables, 11 PCs, and 17 PCs)

data.frame(Model_type = c("Linear", "GLMnet", "randomForest", "GAM", "XGBoost"),
           RMSE_original_train = c(rmse_lm, rmse_glmnet, rmse_rf, rmse_gam, rmse_xgb),
           RMSE_11_components_train = c(rmse_lm_11, rmse_glmnet_11, rmse_rf_11, rmse_gam_11, rmse_xgb_1
           RMSE_17_components_train = c(rmse_lm_17, rmse_glmnet_17, rmse_rf_17, rmse_gam_17, rmse_xgb_1
```

```
##     Model_type RMSE_original_train RMSE_11_components_train
## 1       Linear           0.1468603                0.1647668
## 2       GLMnet           0.1434452                0.1639404
## 3 randomForest           0.1352571                0.1403977
## 4          GAM           0.1536102                0.1480081
## 5      XGBoost           0.1302853                0.1398979
##   RMSE_17_components_train
## 1                0.1672974
## 2                0.1662843
## 3                0.1374679
## 4                0.1531496
## 5                0.1340921
```

The two train sets containing 11 and 17 principal components has almost the same results than the train set containing the original predictors, except for the XGBoost. It is important to notice that when training XGBoost models with the principal components, the `SalePrice` outcome must not be in the `data` argument like we are used to do with the `train` function of the `caret` package or other models. The reason is that it will be considered as predictor, and it will give an overoptimistic result and the model will be overfit. That's why we use the `SalePrice` column of the original `train_set` while being carful to keep the same index of rows by using the same `test_index` on every train set.

It is also important to notice that we can get almost the same results than the train set with original variables with only 11 principal components. The parallel analysis was greatly useful to confirm the number of dimensions to retain.

Ensemble

The XGBoost model has slightly improved our results. Despite this improvement, our model seems still far from the best result of the Kaggle contest. Would an ensemble improve the results ? I will select the best 3 models so far : GLMnet, randomForest, and XGBoost models. We can do this with the following piece of code.

```
# Creating an ensemble of the 3 best models : GLMnet, randomForest and XGBoost

ensemble <- (pred_glmnet + pred_rf + pred_xgb) / 3
```

So, does the ensemble improve the overall predictions ?

```
# RMSE of the ensemble

(rmse_ensemble <- RMSE(log(test_set$SalePrice), ensemble))
```

```
## [1] 0.1292534
```

The RMSE of the ensemble model is a little bit better than the RMSE of the XGBoost model.

I could only use the XGBoost model for the validation set. However, my main concern is the risk of overfitting. Even though the RMSE of the XGBoost model is close to the RMSE of the other models, we cannot neglect the risk of overfitting. The ensemble we've just made gives a slightly better result than most of the models whose RMSE is between 0.13 and 0.15.

Therefore, as a final model I will choose the ensemble model as its overall result is better than most of the models (linear, GLMnet, randomForest, and GAM) and it should balance with the risk of overfitting of the XGBoost model. Kaggle allows us to send several submissions, so I will send some other models we've studied so far in order to compare with the new data, the validation set.

Validation

It is now time to make predictions on the validation set. As said earlier, we will create an ensemble with a GLMnet model, a randomForest model, and a XGBoost model. We create this ensemble with the following piece of code.

```
## Predictions of GLMnet

pred_val_glmnet <- predict(model_glmnet, validation)

## Predicting 'SalePrice' with the randomForest model
```

```r
pred_val_rf <- predict(model_rf, validation)

## Predictions of XGBoost model

# Creating the treated validation data
df_val_treat <- prepare(treatplan, validation, varRestriction = newvars)

# Finally, we add the new binary variables with the numerical variables
validation_treat <- validation[, -1] %>% select_if(is.numeric) %>%
  cbind(df_val_treat)

# Predictions of the XGBoost model
pred_val_xgb <- predict(model_xgb, as.matrix(validation_treat))



## Creating the ensemble

ensemble_val <- (pred_val_glmnet + pred_val_rf + pred_val_xgb) / 3
```

We must not forget that our predictions are a log transformation. So we must use the exponential function to get the real prices, as we are asked to send the real prices. Only the `Id` column and the predictions column are needed for the submission. So we will prepare a file with these two columns only as shown below.

```r
# Selecting Id and predictions for submission. We must not forget to use the exponential
# function to get the real values of sale prices.

submission_ensemble <- validation %>% mutate(SalePrice = exp(ensemble_val)) %>%
  dplyr::select(Id, SalePrice)

# Saving the submission
write.csv(submission_ensemble, "Submission Ensemble.csv", row.names = FALSE)
```

Final result

According to Kaggle, the final RMSE for the ensemble model we built above is 0.13224. As expected, we are very far from the best result of the contest, but our result is constant with what we have found during training.

For fun, and because we made several models, let's compare the results of other models for the validation set in the following data frame.

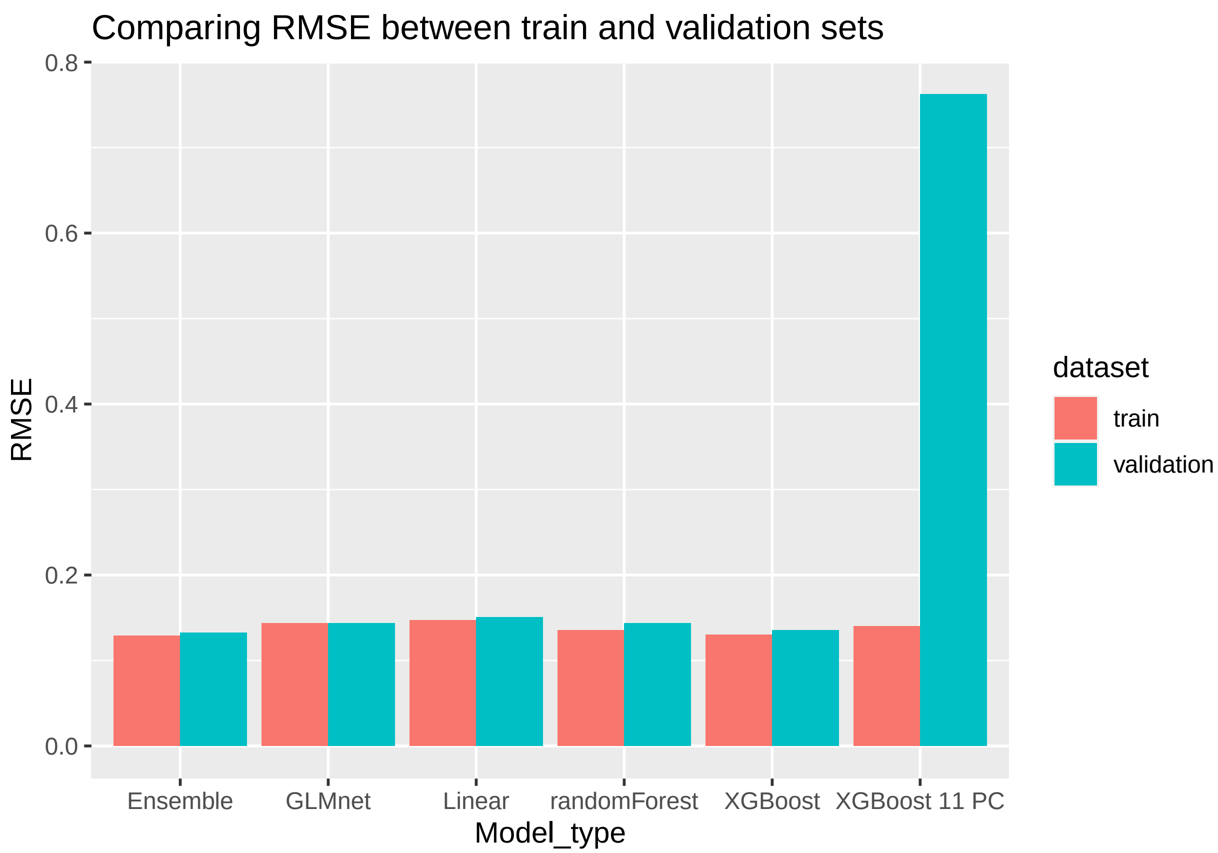
```r
# Comparison of RMSEs between train and validation sets

data.frame(Model_type = c("Linear", "GLMnet", "randomForest", "XGBoost", "Ensemble", "XGBoost 11 PC"),
           RMSE_original_train = c(rmse_lm, rmse_glmnet, rmse_rf, rmse_xgb, rmse_ensemble, rmse_xgb_11)
           RMSE_validation = c(0.15088, 0.14341, 0.14376, 0.13568, 0.13224, 0.76273))
```

```
##      Model_type RMSE_original_train RMSE_validation
## 1        Linear           0.1468603         0.15088
## 2        GLMnet           0.1434452         0.14341
## 3  randomForest           0.1352571         0.14376
```

```
## 4       XGBoost          0.1302853         0.13568
## 5      Ensemble          0.1292534         0.13224
## 6 XGBoost 11 PC          0.1398979         0.76273
```

For better visualization, we can make the following plot.

```r
comparison_rmse <- data.frame(Model_type = c("Linear", "GLMnet", "randomForest", "XGBoost", "Ensemble", 
                              RMSE = c(rmse_lm, rmse_glmnet, rmse_rf, rmse_xgb, rmse_ensemble, rmse_xgl
                              dataset = c("train", "train", "train", "train", "train", "train", "valida


# Bar plot of RMSEs between train and validation sets

comparison_rmse %>% ggplot(aes(Model_type, RMSE, fill = dataset)) +
  geom_col(position = "dodge") +
  ggtitle("Comparing RMSE between train and validation sets")
```



As we can see, the RMSE for all models between train and validation sets was almost the same, except for the XGBoost model containing 11 principal components. The RMSE on the validation set was 0.76273, which is six times higher than during training. This model was clearly overfit. But in general, ours models are well fit.

Conclusion

One of the goal of this capstone project was to use at least two models or algorithms. We used five different models during the training part, and used three of them to build an ensemble that is well balanced whose final RMSE is close to what we found during the training part. We also found that we could greatly reduce the number of dimensions with a PCA. The parallel analysis showed us that we could retain 11 principal components, and we got almost the same results than with models containing original predictors, except for the XGBoost model on the validation set.

However, I did not succeed to beat the best RMSE of the kaggle contest, which is 0.00044. My final model, which is an ensemble of three models (GLMnet, randomForest, and XGBoost), reached a Root Mean Squared Error of 0.13224 on the validation set. I think this model can be use as a base for further improvements. Here are the improvements that I'm thinking about :

- Improving tuning of hyperparameters : some models like GLMnet and XGBoost have a lot of parameters. We only used the main ones here, so it is possible to do better.
- Using cross-validation only without data partition : we split the data into train and test sets (80% vs 20% respectively), but maybe I should have used the whole dataset (the first `train` set) without splitting it to get more information as the dataset is quiet small. Training would have been done through cross-validation only, but we wouldn't have been able to test our model with a `test` set. However RMSE can still be calculated through cross-validation and can give a good indication whether a model is good or not before building the final model.
- Deleting outliers : During the training part, we noticed there was probably an outlier when we visualized our predictions against the actual values. Deleting it may improve the model, but I think it would not make such a big difference.
- The cleaning of the dataset, which was a big part before the analysis, may also have been done in a different way.

Despite it was "just" a regression problem, this dataset was a great challenge and going deeper in the models to get the most out of them was fascinating. I would greatly appreciate comments, advices or feedback on how you would improve this model, or what kind of models you would have used.

`Arnaud RAULET`