House Price Prediction

Andreas Maos

06 September, 2019

Contents

1 Introduction **1**
1.1 The project . . . 1
1.2 The dataset . . . 2

2 Analysis and data wrangling **2**
2.1 Check for and fix any NA values . . . 2
2.2 Check and fix column names . . . 3
2.3 Exploratory data analysis . . . 3
2.4 Feature correlations . . . 9
2.5 Principal component analysis (PCA) . . . 10

3 Machine learning **12**

4 Results **13**
4.1 XGBoost . . . 14

5 Conclusion **14**

6 Appendix **16**
6.1 RMSE function . . . 16
6.2 Modify the test set to match the training set . . . 16
6.3 Rank deficiency test . . . 21
6.4 Gradient Boosting Machines (gbm_h2o) . . . 22
6.5 Operating system . . . 23
6.6 R session information . . . 23

1 Introduction

1.1 The project

Owning or renting a property is a big part of people's lives. Besides the comfort and warmth it offers, it also comes with significant expenses. House prices have been increasing for the past 50 years. The OECD provides data and interactive visualizations that prove this. Major cities around the world, such as Hong Kong, New York, London and San Francisco have seen their housing prices skyrocket. A property's price depends on several factors such as size, area, age and many more. With so much housing data now available, would it be possible to predict house prices? This project constitues the final part of the Professional Certificate in Data Science by HarvardX. The project aims to exploit available information about different houses and leverage the power of data science and machine learning (ML) to predict their prices. Economic and political data were not considered and the analysis focused on house-specific data, although the former can also impact house prices. The data was retrieved, explored, cleaned and then used to predict house prices using several ML models. The methodology and techniques used are descibed in the following sections.

1.2 The dataset

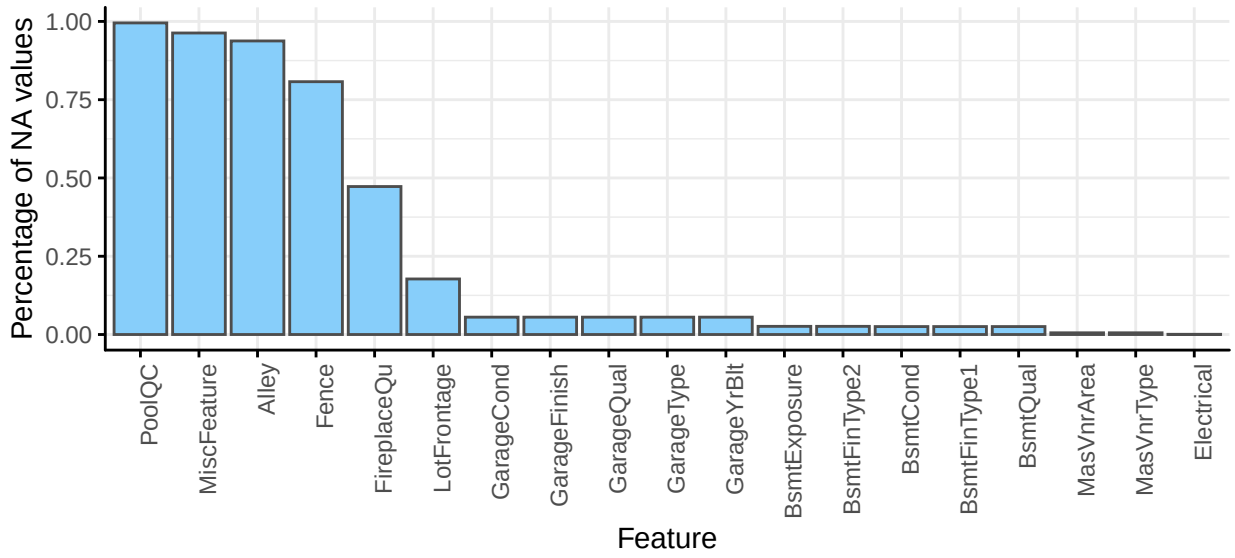
The data used in this project originated from one of Kaggle's competitions, **House Prices: Advanced Regression Techniques**. A **training set**, a **test set** and a **description file** were available on Kaggle. The description file contained useful information about each feature. The training set was used to explore the data, fit ML models and then make predictions using those models. The models were then evaluated by calculating the **Root Mean Square Error (RMSE)** between the **logarithm** of the predicted value and the logarithm of the observed sales price (see Appendix). Taking logs means that errors in predicting expensive houses and cheap houses will affect the result equally. After fitting ML models, they were used to predict house prices on the test set. Finally, the predictions were submitted to Kaggle to compare how well this analysis did against other people's attempts to predict house prices.

The training set contained 1460 rows and 81 columns corresponding to individual houses and to the different house features respectively. Some of these features included *SalePrice*, *Neighborhood*, *RoofStyle*, *YearBuilt* and many more, with the explanation for each one found in the description file. Different features were present in different formats. Some of the columns contained **numeric variables** and others had **character variables**. From the numeric variables, some were **continuous** and some were **ordinal/categorical** features. Also, several columns contained rows with missing values or *NA*. The first step in this analysis was to identify the *NA* values and either fix or remove them. Then, the features were separated into continuous and categorical in order to produce appropriate plots to explore each one.

2 Analysis and data wrangling

The downstream sections describe the methods used to explore and analyse the data in order to build accurate ML models. Anything described below has been performed using only the traning set. Data transformations and modifications that were applied on the training set were later applied on the test set as well (code in Appendix) in order to be able to make predictions using the test set.

2.1 Check for and fix any NA values



As seen in the plot above, there are 19 features with *NA* values in their columns. Some features like *PoolQC* contain may *NAs* whereas others like *Electrical* contain just a few. These features could contain useful information that could assist in accurately predicting house prices. As such, each one was further explored

to decide how to best deal with the *NAs*. To understand what each of these features described and how to appropriately fill in any *NA* values, the description file was consulted.

The features with *NA* values included: *LotFrontage, Alley, MasVnrType, MasVnrArea, BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1, BsmtFinType2, Electrical, FireplaceQu, GarageType, GarageYrBlt, GarageFinish, GarageQual, GarageCond, PoolQC, Fence, MiscFeature*. In all of these, with the exception of *MasVnrArea* and *Electrical*, the *NAs* were replaced with the string *None*. This was because either an *NA* meant that the feature was not present, according the description file, or because the actual value was missing and *None* was the most common value in the column, as in the case of *MasVnrType*. For *MasVnrArea*, the *NAs* were replaced with the numeric value 0 as the corresponding *MasVnrType* rows had *None*. In the case of *Electrical*, the *NAs* were replaced with the most common value in the column, *SBrkr*. Initially, in features such as *PoolQC* and *FireplaceQu*, the *NAs* were to be replace with characters such as *NoPool* and *NoFireplace*. However, using *None* instead made things easier downstream.

The features *LotFrontage* and *GarageYrBlt* were originally removed as it was not clear how to best replace the *NAs* in these ones. Later, it was decided to replace the *NAs* in *LotFrontage* with the mean of the remaining values in the column which was 70.05. *GarageYrBlt* was still kept out as there were other features related to the house's garage that already provided enough information.

2.2 Check and fix column names

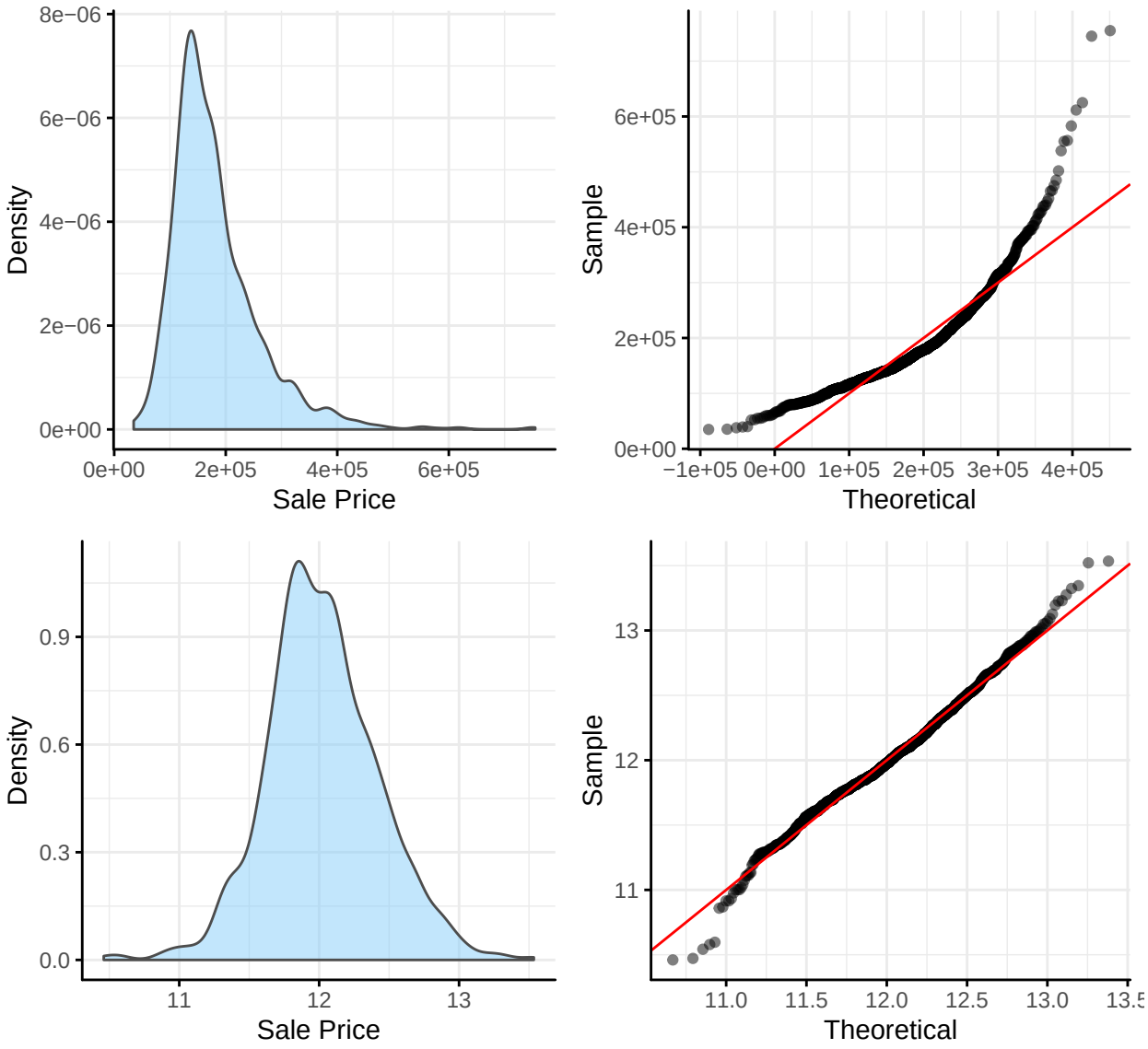
Some ML model would fail to fit if any of the feature names (i.e. column names) start with a digit. Hence, the column names of the dataset were checked to detect if any started with a digit. The `str_which()` function with the regex pattern `^\\d` were used to detect these columns. It turned out that 3 features, *1stFlrSF, 2ndFlrSF, 3SsnPorch*, had their name starting with a digit. These were changed to *FirstFlrSF*, *SecondFlrSF* and *ThreeSsnPorch* respecitvely.

2.3 Exploratory data analysis

In the following sections, the distribution of the *SalePrice* and the relationship of other features with *SalePrice* were studied.

2.3.1 Sale price distribution

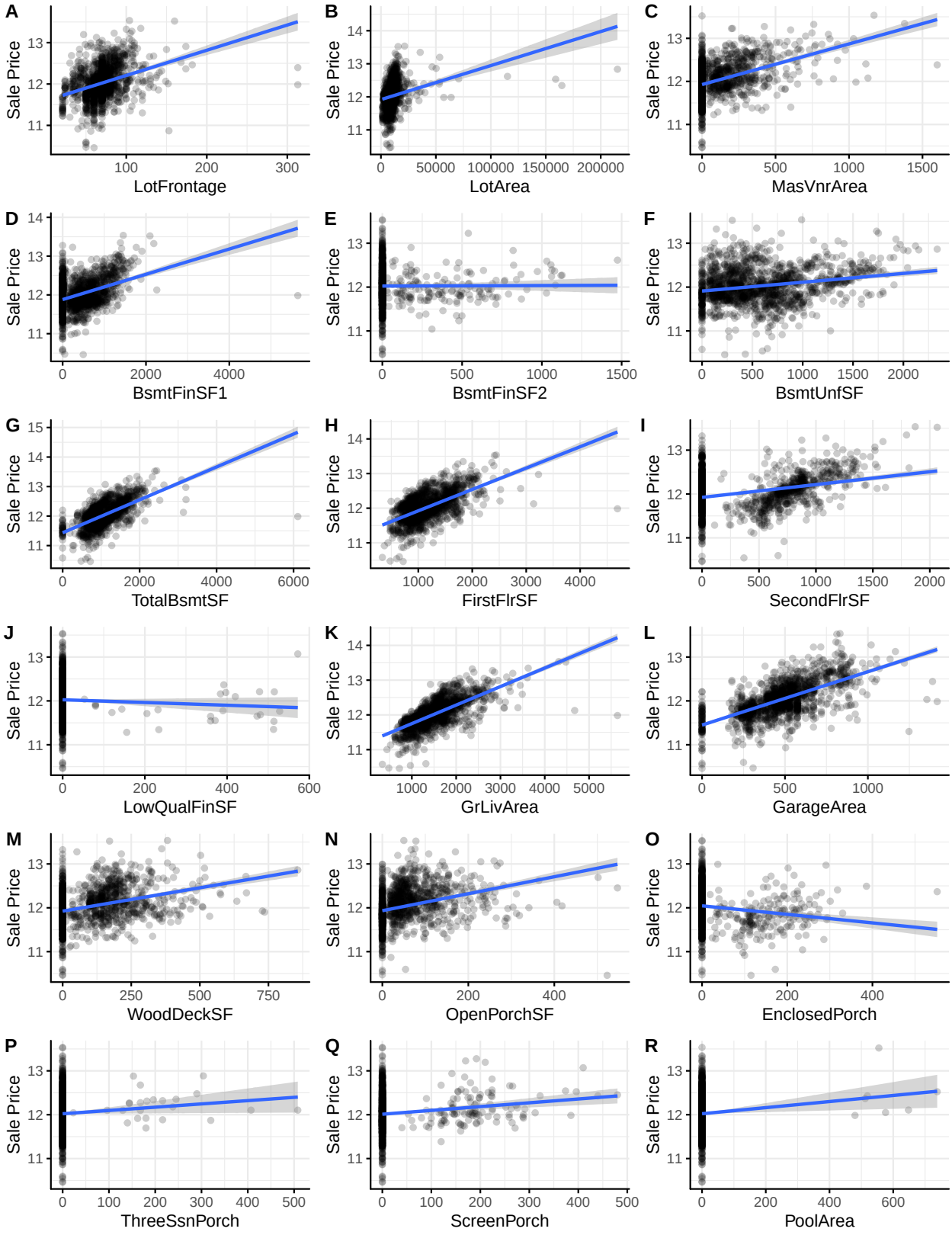
As shown below, in the plots in the first row, the *SalePrice* feature was not normally distributed. It had a right (or positive) skew. Hence, the *SalePrice* column was transformed using the `log1p()` function and was brought closer to the normal distribution as shown by the plots in the second row.



2.3.2 Numeric value columns

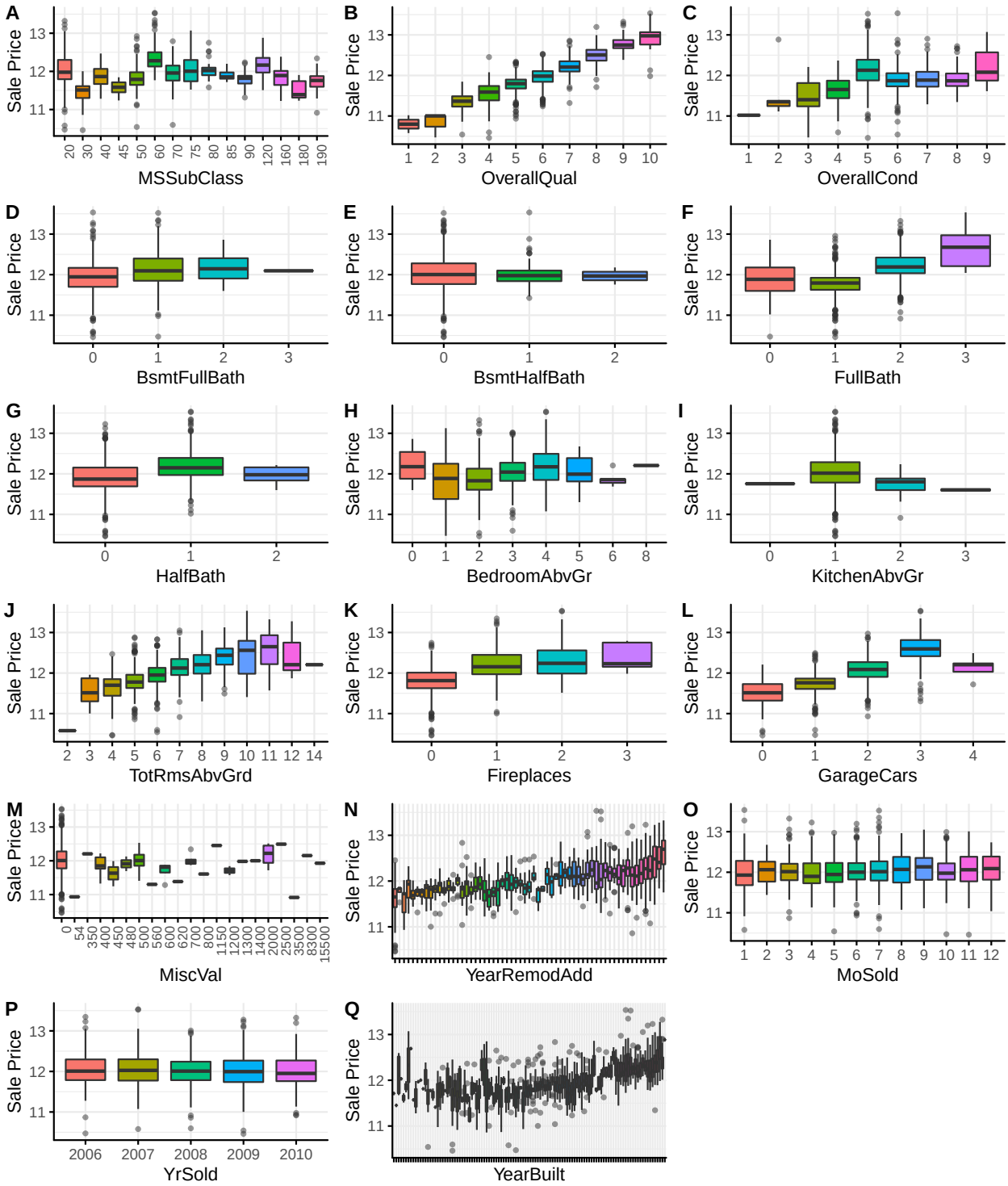
As it was mentioned before, some columns had numeric and some had character variables. First, the numeric ones were isolated and their relationship with *SalePrice* was studied. Of the training set's 80 columns, 37 contained numeric variables. These numeric features could be further split into continuous and ordinal/categorical features. The separation was done manually with the help of the description file. First, the continuous features were isolated and plotted against the *SalePrice*. A scatter plot with a line of best fit was plotted for each continuous feature.

2.3.2.1 Continuous features



2.3.2.2 Ordinal features

Then, the categorical/ordnial features were isolated and plotted against the *SalePrice* using boxplots. An ordinal variable is similar to a categorical variable. The difference between the two is that for ordinal features there is a clear ordering of the variables.



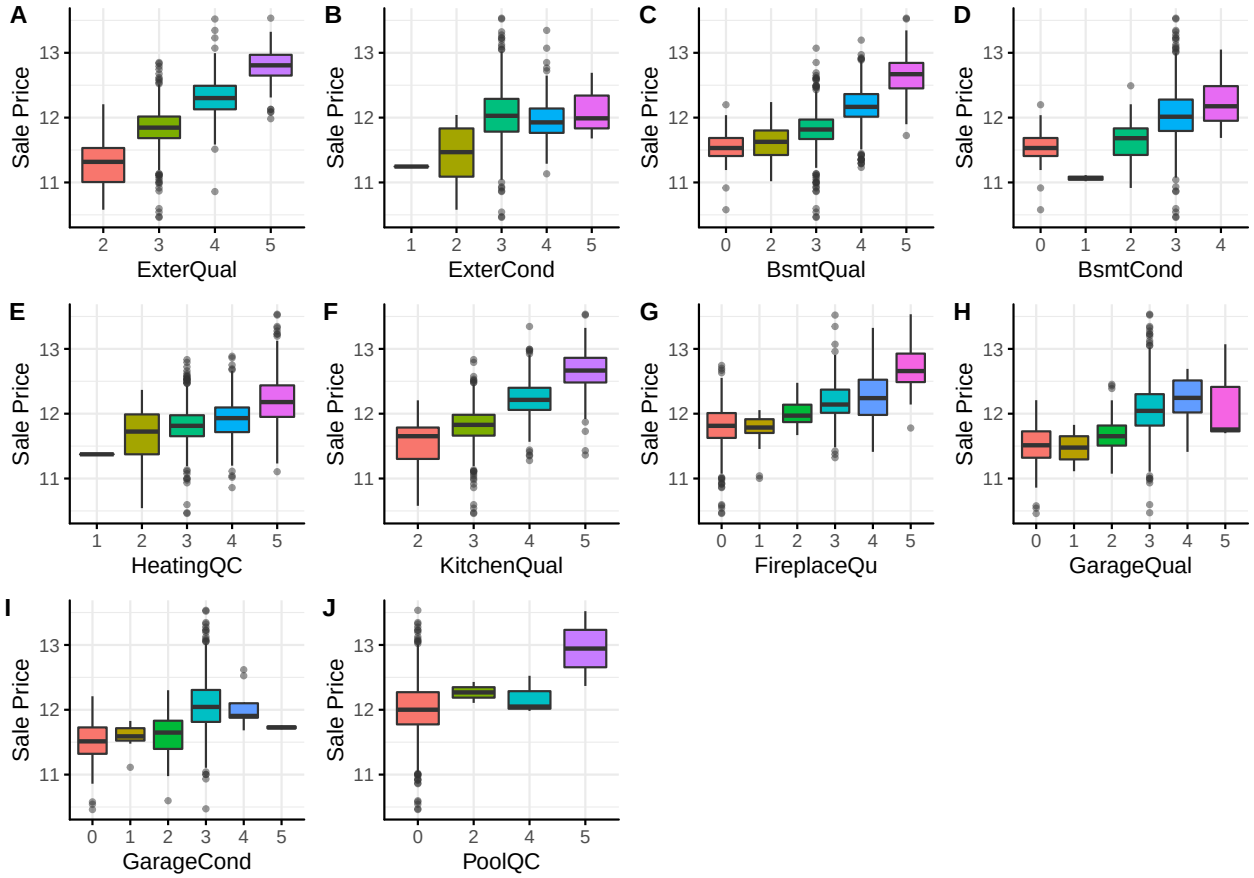
2.3.3 Character value columns

After studying the columns with numeric features, the focus shifted on features with character variables.

2.3.3.1 Convert quality/condition related columns to numeric

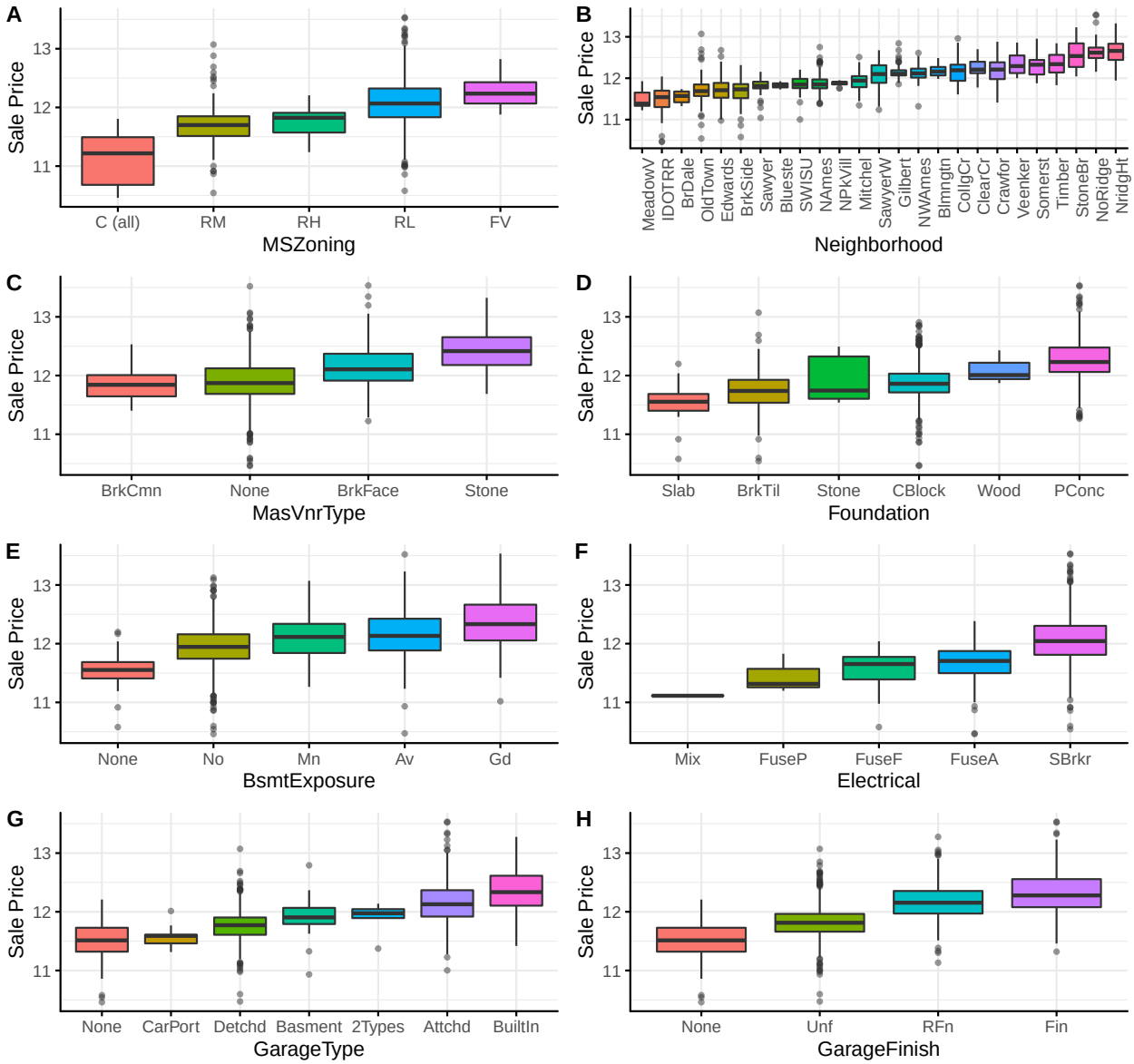
Of the 43 columns with character variables, it was noticed that some of them encoded features referring to the quality or condition of a house attribute (regex pattern `.Q.|.Cond$`). These contained character values that could easily be replaced with numeric ones as shown in the table below. The character value meaning was extracted from the description file while the value *None* came from the *NA* replacements that occured above.

Value	Meaning	Replacement
None	Not applicable	0
Po	Poor	1
Fa	Fair	2
TA	Typical/Average	3
Gd	Good	4
Ex	Excellent	5



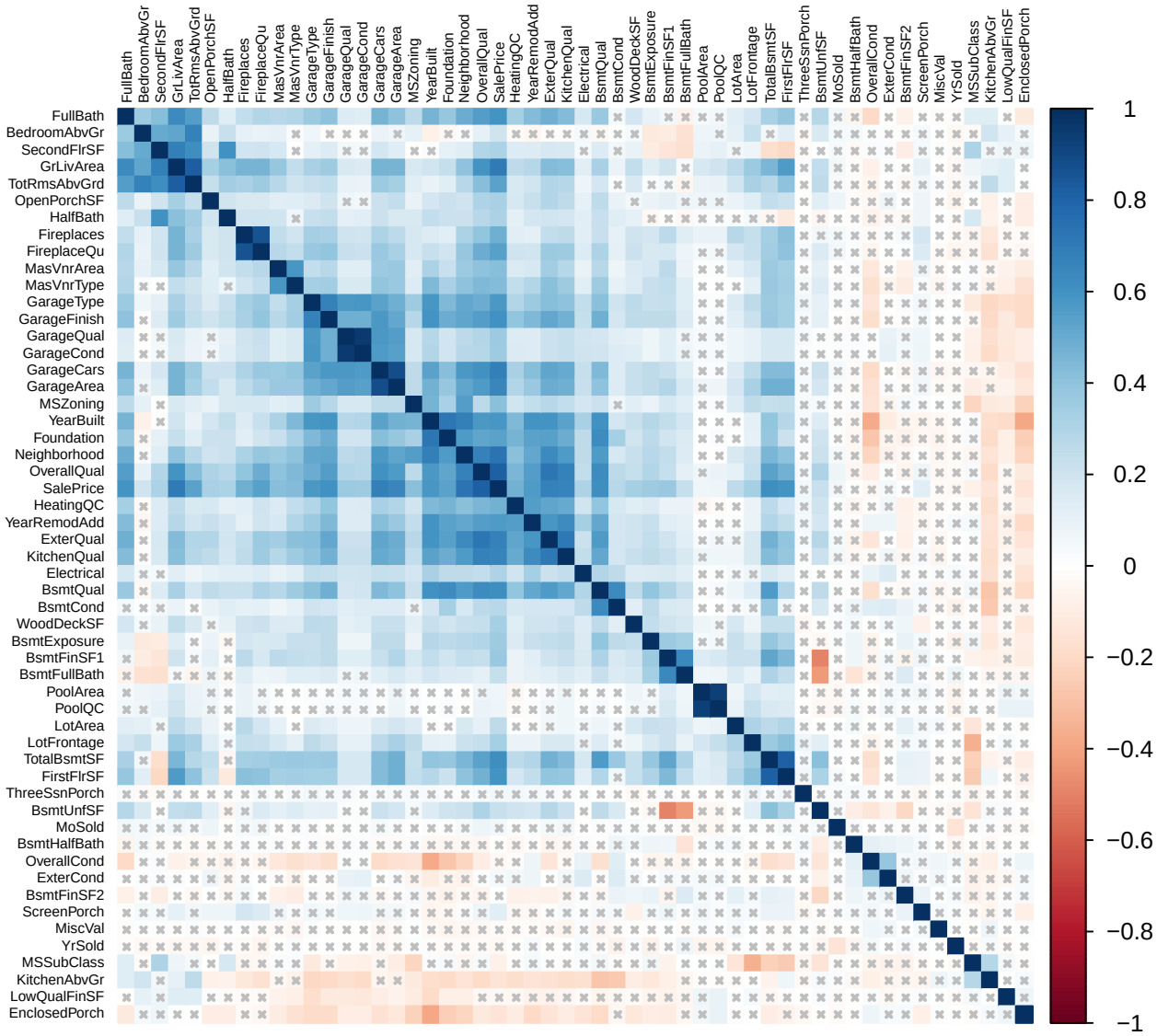
2.3.3.2 Explore the remaining columns

The remaining 33 features contained character variables but replacing these with numbers was not straightforward. As these represented categorical features, they were plotted against *SalePrice* using boxplots. Before plotting, the column entries were converted to factors and their levels were ordered in order of asceding median *SalePrice*. Here, not all 33 plots are shown, but rather only those that were observed to have an effect on *SalePrice*. These columns were converted to numeric just like the quality/condition related columns above. Features that could be converted to numbers include: *MSZoning*, *Neighborhood*, *MasVnrType*, *Foundation*, *BsmtExposure*, *Electrical*, *GarageType* and *GarageFinish*.

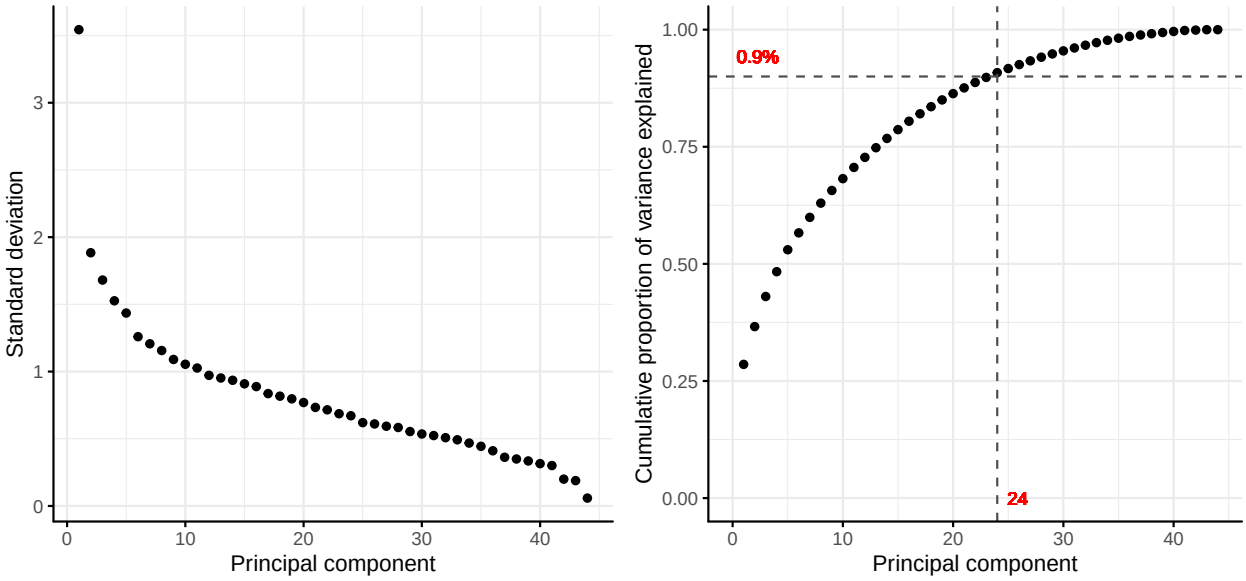


2.4 Feature correlations

By this point, all the numeric features as well as the character ones converted to numeric, had been merged into a single data frame. All the entries of the data frame were in numeric format. The `rcorr()` function from the `Hmisc` package was used to calculate the **correlation coefficient** along with its corresponding **p-value** for each feature-feature pair. The **correlogram** is shown below with blue corresponding to positive correlation, red to negative correlation and grey crosses to feature-feature pairs with a non-significant p-value for their correlation coefficient. This was used to get a more quantitative representation of the correlation of *SalePrice* with the other variables, as opposed to the more visual scatter and box plots shown in the previous sections.



2.5 Principal component analysis (PCA)



PCA was performed to dissect any hidden structure in the data. The **standard deviation** explained by each principal component (PC) along with the **cumulative proportion of variance** per PC were calculated and plotted as shown above. PCA revealed that by using just 24 PCs one could explain 0.9% of the dataset's variance. Since the dimensionality of the dataset was not so high and the computing resources were enough to handle the dataset, the data was not subsetted based on any PCs and as such, all dimensions were used.

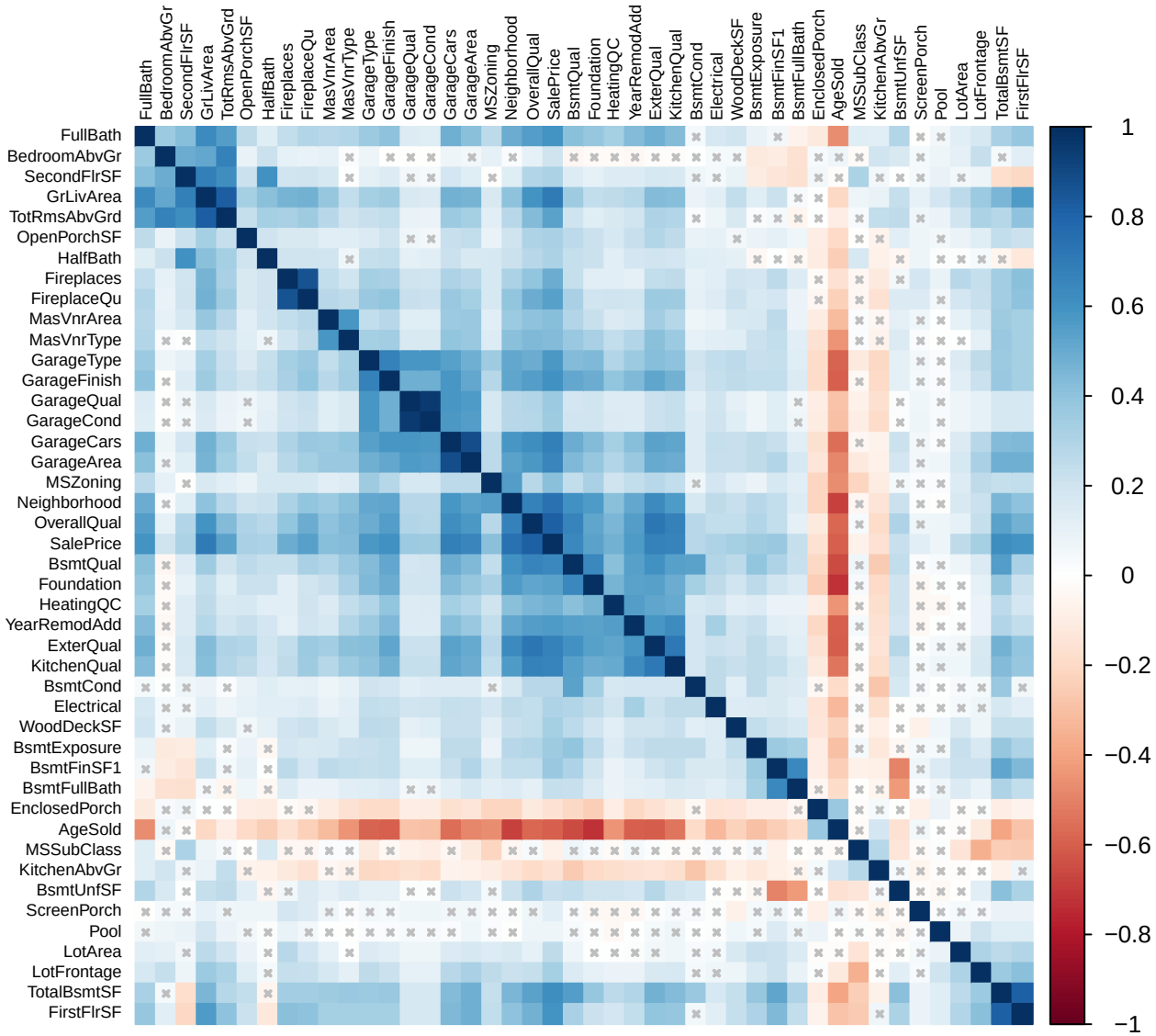
2.5.1 Feature insights and decisions

After exploring the relationship of each feature with *SalePrice*, certain decisions were made:

- *MoSold* and *YrSold* had no effect on *SalePrice*. *MoSold* was removed and *YrSold* was used to create a new, more useful feature encoding the age of a house when it was sold: `AgeSold = YrSold - YearBuilt`.
- Very few houses Had a pool and hence *PoolArea* and *PoolQC* did not have a very strong correlation with *SalePrice*. However, as they had some significant correlation with *SalePrice*, it was decided to include them by combining them in a single feature as `Pool = PoolArea * PoolQC`.
- *ThreeSsnPorch*, *MiscVal*, *BsmtHalfBath*, *OverallCond*, *ExterCond*, *BsmtFinSF2* and *LowQualFinSF* seemed to have no or very low correlation with *SalePrice* and hence a decision to remove them was taken.
- There were 5 houses with `GarageCars == 4` that had a median *SalePrice* less than those with `GarageCars == 3` and almost equal to those with `GarageCars == 2`. These were probably outliers and were removed.

After performing some **data filtering** and **feature engineering** based on the exploratory data analysis, feature correlations and PCA results, ML models were ready to be fitted and used to make predictions. Feature correlations were once again checked to confirm that there are no insignificant correlations between features (see correlogram below). In addition, before moving on to fitting ML models, the data frame to be used for model fitting was checked for **matrix rank deficiency**. The code to check for rank deficiency is based on **QR decomposition** and can be found in the Appendix.

The following features **were not** used for ML: *Id, Street, Alley, LotShape, LandContour, Utilities, LotConfig, LandSlope, Condition1, Condition2, BldgType, HouseStyle, OverallCond, YearBuilt, RoofStyle, RoofMatl, Exterior1st, Exterior2nd, ExterCond, BsmtFinType1, BsmtFinType2, BsmtFinSF2, Heating, CentralAir, LowQualFinSF, BsmtHalfBath, Functional, PavedDrive, ThreeSsnPorch, PoolArea, PoolQC, Fence, MiscFeature, MiscVal, MoSold, YrSold, SaleType, SaleCondition.* This was because there was no evidence that they had an effect on *SalePrice* or because it was hard to encode the character variables with numeric ones.



3 Machine learning

A range of different ML regression models were fitted and then used to make predictions. The ML models were fit both on the **standardised and non-standardised** training set. When standardising, the values were replaced by their *z score* resulting in them being redistributed with mean $\mu = 0$ and standard deviation $\sigma = 1$. For each ML model, the RMSE and duration to fit the model and make predictions were recorded. Different ML algorithms use different parameters and some of these are tunable, for example the number of neighbours in KNN. Properly tuning these parameters results in more accurate predictions. In order to optimize any tunable parameters, **k-fold cross validation** was used.

In k-fold cross validation, the observations are randomly split into k **non-overlapping sets**, with each set consisting of a training set and a test set. In this project, 10-fold cross-validation was utilized meaning that the original training set was split into 10 folds, with each one consisting of 90% training data and 10% test data. For each fold, the model was fitted on the 90% training set, evaluated on the 10% test set and the **mean square error (MSE) for each fold** was calculated. This was repeated using a range of different parameters and at the end the parameters that minimized the MSE were selected. The model with the optimum parameters was returned and used to make predictions of *SalePrice*. The `caret` package enables one to specify how cross-validation will be performed, using the `trainControl()` function, and also to allow repetition of the calculations until the optimum parameters are identified, using the `train()` function. The range of values for any tunable parameter can be expanded beyond the defaults using the `tuneGrid` argument of `train()`.

The following ML models were fitted and used to make predictions of *SalePrice*:

- Support vector machine with linear kernel (*svmLinear*)
- Boosted generalized linear model (*glmboost*)
- Linear regression (*lm*)
- Penalized linear regression (*penalized*)
- Generalized additive model using LOESS (*gamLoess*)
- Stochastic gradient boosting (*gbm*)
- K-Nearest neighbour (*kknn*)
- eXtreme gradient boosting (*xgbLinear*)
- Random forest (*rf*)
- eXtreme gradient boosting (*xgbTree*)
- Boosted tree (*bstTree*)

During the analysis, a **Gradient boosting machines (*gbm_h2o*)** model was also fitted and evaluated with a resulting RMSE of 0.11 (see Appendix for code). However, given that this model did not perform significantly better than any of the other models and due to the *gbm_h2o* model running for a lot longer than any of the others, it was decided not to include it in the final results.

4 Results

Model	RMSE	Time (minutes)
svmLinear	**0.14602**	**0.22**
glmboost	0.14095	0.03
penalized	0.13698	0.64
lm	0.13682	0.01
kknn	0.13393	0.20
gamLoess	0.12276	0.88
gbm	0.10703	0.14
xgbTree	0.08552	2.24
bstTree	0.07957	2.57
ensemble	**0.05795**	**10.86**
rf	0.05671	3.51
xgbLinear	**0.02978**	**2.53**

Note:
WITHOUT STANDARDISATION/SCALING

Model	RMSE	Time (minutes)
svmLinear	**0.36524**	**0.22**
glmboost	0.35256	0.03
penalized	0.34254	0.65
lm	0.34221	0.02
kknn	0.32921	0.23
gamLoess	0.30704	1.03
gbm	0.26734	0.16
bstTree	0.24605	2.45
xgbTree	0.23565	1.98
ensemble	**0.15018**	**10.93**
rf	0.14530	3.68
xgbLinear	**0.01131**	**2.82**

Note:
WITH STANDARDISATION/SCALING

The tables above show the results for each ML model for both the standardised and non-standardised training set. As the results with standardisation do not differ significantly from those without standardisation, something that was later confirmed by the Kaggle submissions, only the results from the non-standardised training set were considered.

For each model the RMSE and time taken to fit the model and make predictions is shown. The models have been ordered by RMSE with the worst performing model (red) shown at the top and the best performing one (green) shown at the bottom of the table. Overall, it is evident that tree-based models such as *xgbTree*, *bstTree* and *rf* performed better than the rest. The *lm* model had a very similar performance to the *kknn* one, although it being 16 times faster. What is surprising is the fact that *lm* performed better than its regularized counterpart, *penalized*, which utilized both L1 and L2 regularization.

The best performing model, the one the the minimum RMSE, was *xgbLinear*. This had an RMSE that was about 2 times smaller than the next best performing one, *rf*. Was this due to the fact that *xgbLinear* actually is a better model or was it because of overfitting?

Ensembles combine multiple machine learning algorithms into one model to improve predictions, and in this case the ensemble performed really well. The *ensemble* model (grey) combined the predictions obtained by all the models with an RMSE of less than or equal to 0.1, *xgbLinear, rf, xgbTree, bstTree*. Then for each observation, the mean of the predictions across all 4 models was calculated and those means were used to

calculate the RMSE. The time taken was calculated by taking the sum of the 4 constituent models's durations. In practice, calculating the ensemble predictions, after the constituent models have been fitted, takes minimal time. Would this be a better method of making predictions by being less prone to overfitting?

Several of these models were used to make predictions on the test set, and those predisctions were submitted to the Kaggle *House Prices: Advanced Regression Techniques* competition. The models and their RMSEs as reported by Kaggle are shown in the table below.

Model	RMSE
xgbLinear	**0.15347**
rf (standardised)	0.14649
rf	0.14580
xgbTree	0.14488
xgbLinear and rf ensemble	0.14373
bstTree	0.14360
ensemble (standardised)	0.14272
ensemble	**0.13960**

Note:
The observed SalePrice values are knly known by Kaggle

4.1 XGBoost

Two of the best performing models, *xgbLinear, xgbTree*, are based on the **XGBoost algorithm**. It is hence worth briefly explaining how this works. XGBoost is an **ensemble learning method**. This means that it aggregates the results of multiple leearners, called **weak (or base) learners**. These are models (classifiers, predictors etc.) that perform poorly, meaning that their accuracy is barely higher than that of chance/guessing. Any base learner can be used in the boosting framework such as linear models (*xgbLinear*) and decision trees (*xgbTree*). By combining many weak learners, a strong ensemble model can be built, and this is what XGBoost does. The **boosting approach** employed by this algorithm allows subsequent models to reduce the errors of the previous ones. Essentially, each model learns from its predecessors and updates the residual errors. The result is a **strong learner** that reduces both the bias and the variance. A more detailed explanation about how XGBoost works can be found in the *XGBoost: A Scalable Tree Boosting System* paper.

5 Conclusion

The aim of this project was to use different data science and ML techniques to predict house prices using the Kaggle *House Prices: Advanced Regression Techniques* dataset. To achieve this there were several steps. Firstly, *NA* values were detected and replaced with apropriate values. Secondly, continuous and ordinal/categorical fetures were identified and appropriately plotted against *SalePrice* to examine the relationship between each pair.Thirdly, features that were identified to have an effect on the target feature, were isolated. Some of these contained character values and hence dummy coding was needed to convert these into numeric values. Fourthly, a correlogram was consulted to identify features that were correlated with *SalePrice* and had a significant p-value for their correlation coefficient. Additionally, this enabled some features to be removed of combined into one new feature. Then, different regression ML models were fitted on the selected features and *SalePrice* predictions were made. the best performing models were selected and used to make *SalePrice* predictions on the test set, which were then submitted to the Kaggle competition. As shown above, an **ensemble** approach, using several ML algorithms, was the best at predicting *SalePrice*.

There were several challeges associated with this project. How to best fill in any *NAs* was not straightforward. This could have been done in a different way and would result in different predictions at the end. Also,

the dummy coding used can be more specifically called **integer encoding**. One could also use **one-hot encoding**. This is where the integer encoded variable is removed and a new binary variable is added for each unique integer value. This could improve the predictions and should be used in further work. In addition, some features could be combined to create new ones. This was also challenging and could be done in several different ways, leaving many unexplored possibilities. Also, there are many ML regression models that could be fitted and each one uses different tunable parameters. The ML models used and the range for each tunable parameter could be further explored to develop more accurate models.

Overall, this project was very interesting and exciting and carried with it the extra enthusiasm of participating in a Kaggle competition. It provided a great opportunity to learn about the different features that affect a house's price and to test and explore different ML algorithms. Having the chance to be introduced to and use the XGBoost algorithm was very helpfull and has placed another tool in my data science arsenal. Looking forward to completing more ML projects and participating in Kaggle competitions.

6 Appendix

6.1 RMSE function

```
RMSE <- function(true_ratings, predicted_ratings) {

  sqrt(mean((true_ratings - predicted_ratings)^2))

}
```

6.2 Modify the test set to match the training set

```
# Data transformation function

# This combines all the data wrangling and modification methods/techniques
# used on the training set throughout the project i.e. replacing NAs,
# modifying columns etc.
# It is used to convert the test set in the same format as the
# training set in order to be able to make predictions using different ML models.
# These ML models were fit on the training set after it had been modified and hence
# they could not be used with the original test set to make predictions
# as the test set would
# be in a different format to the modified training set that the ML models were fit onto.

# NOTE: This function should be run only from this point onwards. This is because
# at some points, the function compares variables in num_train
# to variables in test and uses the differences to properly rename/modify feature entries.
# Hence, for this comparison and modifications to work,
# num_train must be in its final format.

modify_data <- function(to_convert, original, final) {

  # to_convert: the data frame that is to be converted
  # original: the original working template
  # final: the final/modified working template
  # original and final will be used to make appropriate modification to to_convert

  # Remove GarageYrBlt column

  to_convert <- to_convert %>% select(-GarageYrBlt)

  # Detect columns with NAs

  na_values <- map_df(.x = colnames(to_convert), .f = function(c) {

    num_na <- sum(is.na(to_convert[[c]]))

    if (num_na != 0) {

      data.frame(col = c,
                 na_n = num_na,
```

```
                na_pct = mean(is.na(to_convert[[c]])),
                stringsAsFactors = FALSE)

  } # if

}) # map_df

# Replace NAs as in the training set

for (na_col in na_values$col) {

  if (na_col == "PoolQC") {

    to_convert[which(is.na(to_convert$PoolQC)), "PoolQC"] <- "None"

  } else if (na_col == "MiscFeature") {

    to_convert[which(is.na(to_convert$MiscFeature)), "MiscFeature"] <- "None"

  } else if (na_col == "Alley") {

    to_convert[which(is.na(to_convert$Alley)), "Alley"] <- "None"

  } else if (na_col == "Fence") {

    to_convert[which(is.na(to_convert$Fence)), "Fence"] <- "None"

  } else if (na_col == "FireplaceQu") {

    to_convert[which(is.na(to_convert$FireplaceQu)), "FireplaceQu"] <- "None"

  } else if (na_col == "LotFrontage") {

    to_convert[which(is.na(to_convert$LotFrontage)), "LotFrontage"] <-
      mean(to_convert$LotFrontage[!is.na(to_convert$LotFrontage)])

  } else if (na_col == "MasVnrType") {

    to_convert[which(is.na(to_convert$MasVnrType)), "MasVnrType"] <-
      names(which.max(table(to_convert$MasVnrType)))

  } else if (na_col == "MasVnrArea") {

    to_convert[which(is.na(to_convert$MasVnrArea)), "MasVnrArea"] <- 0

  } else if (na_col == "Electrical") {

    to_convert[which(is.na(to_convert$Electrical)), "Electrical"] <-
      names(which.max(table(to_convert$Electrical)))

  } else if (na_col %in%
             na_values$col[str_detect(string = na_values$col,
                                      pattern = "^Garage(?!Y|A|.*Cars$)")]) {
```

```
    to_convert[which(is.na(to_convert[[na_col]])), na_col] <- "None"

  } else if (na_col %in%
             na_values$col[str_detect(string = na_values$col,
                                      pattern = "^Bsmt(?!FinSF\\d|.*SF$|.*Bath$)")]) {

    to_convert[which(is.na(to_convert[[na_col]])), na_col] <- "None"

  } else{

    if (class(to_convert[[na_col]]) == "character") {

      to_convert[which(is.na(to_convert[[na_col]])), na_col] <-
        names(which.max(table(to_convert[[na_col]])))

    } else if (class(to_convert[[na_col]]) == "numeric") {

      to_convert[which(is.na(to_convert[[na_col]])), na_col] <-
        as.numeric(which.max(table(to_convert[[na_col]])))

    } # nested else if

  } # else

} # loop

# Rename colnames to avoid any that start with a digit

colnames(to_convert)[str_which(string = colnames(to_convert),
                               pattern = "^\\d")][1] <- "FirstFlrSF"

colnames(to_convert)[str_which(string = colnames(to_convert),
                               pattern = "^\\d")][1] <- "SecondFlrSF"

colnames(to_convert)[str_which(string = colnames(to_convert),
                               pattern = "^\\d")][1] <- "ThreeSsnPorch"

if (any(grepl(pattern = "^\\d", colnames(to_convert)))) {
  warning("ERROR! Check that there are no column names that start with a digit!")
}

# Get columns with numeric values

numeric_cols <- sapply(X = colnames(to_convert), FUN = function(c) {

  ifelse(test = class(to_convert[[c]]) == "numeric", yes = TRUE, no = FALSE)

})

# Isolate columns with numeric values

num_test <- to_convert[, numeric_cols]
```

```r
# Isolate columns with character values

chr_test <- to_convert[, !numeric_cols]

# Get columns with quality/condition information

qu_cols <- colnames(chr_test)[str_detect(string = colnames(chr_test),
                                         pattern = ".Q.|.Cond$")]

# Convert character ratings to numeric ones - these will still be categorical features

for (c in qu_cols) {

  chr_test[[c]] <- replace(x = chr_test[[c]], list = chr_test[[c]] == "Ex", values = 5)

  chr_test[[c]] <- replace(x = chr_test[[c]], list = chr_test[[c]] == "Gd", values = 4)

  chr_test[[c]] <- replace(x = chr_test[[c]],  list = chr_test[[c]] == "TA", values = 3)

  chr_test[[c]] <- replace(x = chr_test[[c]], list = chr_test[[c]] == "Fa", values = 2)

  chr_test[[c]] <- replace(x = chr_test[[c]], list = chr_test[[c]] == "Po", values = 1)

  chr_test[[c]] <- replace(x = chr_test[[c]],
                           list = !chr_test[[c]] %in% c(1, 2, 3, 4, 5), values = 0)

  chr_test[[c]] <- factor(x = as.numeric(chr_test[[c]]))

} # loop

# Add the (now) numeric columns of chr_test to num_test

num_test <- to_convert[, numeric_cols] %>%
  bind_cols(chr_test[, qu_cols])

# Modify some of the character value columns as they were modified in the training set

updated_cols <- c("MSZoning", "Neighborhood", "MasVnrType", "Foundation",
                  "BsmtExposure", "Electrical", "GarageType", "GarageFinish")

for (col in updated_cols) {

  replacement <- data.frame(filter(.data = original, GarageCars < 4) %>% select(col),
                            new = final[[col]],
                            stringsAsFactors = FALSE) %>%
    distinct() %>%
    mutate_at(col, .funs = factor)

  for (i in 1:nrow(replacement)) {

    chr_test[[col]] <- replace(x = chr_test[[col]],
                               list = chr_test[[col]] == replacement[i, col],
                               values = replacement[i, "new"])
```

```
    } # nested loop

  } # loop

  # Add the new numeric columns of chr_test to num_test

  num_test <- to_convert[, numeric_cols] %>%
    bind_cols(chr_test[, qu_cols]) %>%
    bind_cols(chr_test[, updated_cols]) %>%
    mutate_all(.funs = as.numeric) %>%
    select(-c(Id))

  # Filtering and feature engineering based on the correlogram

  # Combine columns to create new features

  # Then remove the old columns

  num_test <- num_test %>%
    mutate(AgeSold = YrSold - YearBuilt) %>%
    select(-c(YrSold, YearBuilt, MoSold)) %>%
    mutate(Pool = PoolArea * PoolQC) %>%
    select(-c(PoolArea, PoolQC))

  # Filter out some data

  num_test <- num_test %>%
    # filter(GarageCars < 4) %>% # no need to filter out these in the test set
    select(-c(ThreeSsnPorch, MiscVal, BsmtHalfBath,
              OverallCond, ExterCond, BsmtFinSF2,
              LowQualFinSF))

  if (colnames(final)[which(!colnames(final) %in% colnames(num_test))] == "SalePrice") {

    return(num_test)

  } else {

    warning("ERROR! Data modification was not successful!")

  }

} # function
```

6.3 Rank deficiency test

```
# Get the matrix's rank - check for rank deficiency

# If rank deficient, check which columns maximize the rank when removed and then
# check how these correlate with SalePrice. Remove those with low/no correlation

if (qr(num_train)$rank != ncol(num_train)) {

  # Check which columns, if removed, maximize the rank

  rankifremoved <- sapply(X = 1:ncol(num_train), function (x) {
    qr(num_train[, -x])$rank
  })

  which(rankifremoved == max(rankifremoved))

  # Check which columns, from the linearly dependent ones, can be removed

  # Check feature correlation with SalePrice

  rank_test <- bind_cols(num_train["SalePrice"],
                         num_train[, which(rankifremoved == max(rankifremoved))])

  cor_rank_test <- rcorr(x = as.matrix(rank_test),
                         type = c("pearson","spearman"))

  corrplot(corr = cor_rank_test$r,
           p.mat = cor_rank_test$P,
           order = "hclust",
           sig.level = 0.01,
           type = "upper",
           insig = "blank",
           tl.cex = 0.9, tl.col = "black", tl.srt = 90)

} else {

  print("Rank deficiency test: OK")

}

# Remove any problematic columns with low correlation with SalePrice
```

6.4 Gradient Boosting Machines (gbm_h2o)

```
# Not sure if it is worth running this model - might include it later

# RMSE: 0.11

model <- "gbm_h2o"

# modelLookup(model)

if(!require(h2o)) {install.packages("h2o")}

set.seed(1)

h2o.init()

h2o.no_progress()

before <- Sys.time()

set.seed(1)

fits[[model]] <- train(SalePrice ~ ., data = num_train, method = model, trControl = control)

preds[[model]] <- predict(object = fits[[model]], newdata = num_train)

rmses[[model]] <- RMSE(preds[[model]], num_train$SalePrice)

rmses[[model]]

after <- Sys.time()

durations[[model]] <- difftime(after, before, units = 'mins')

durations[[model]]

h2o.removeAll()

h2o.shutdown(prompt = FALSE)

detach("package:h2o", unload = TRUE)

rmse_results <- bind_rows(rmse_results,
                          data.frame(model = model,
                                     RMSE = rmses[[model]],
                                     time = durations[[model]],
                                     stringsAsFactors = FALSE))
```

6.5 Operating system

```
##                _
## platform       x86_64-apple-darwin15.6.0
## arch           x86_64
## os             darwin15.6.0
## system         x86_64, darwin15.6.0
## status
## major          3
## minor          5.3
## year           2019
## month          03
## day            11
## svn rev        76217
## language       R
## version.string R version 3.5.3 (2019-03-11)
## nickname       Great Truth
```

6.6 R session information

```
## R version 3.5.3 (2019-03-11)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS Mojave 10.14.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8
##
## attached base packages:
## [1] splines   parallel  stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] randomForest_4.6-14 xgboost_0.90.0.2    bst_0.3-17
##  [4] gbm_2.1.5           kknn_1.3.1          gam_1.16.1
##  [7] foreach_1.4.7       mboost_2.9-1        stabs_0.6-3
## [10] kernlab_0.9-27      penalized_0.9-51    plyr_1.8.4
## [13] tinytex_0.15        kableExtra_1.1.0    knitr_1.24
## [16] RColorBrewer_1.1-2  corrplot_0.84       ggrepel_0.8.1
## [19] cowplot_1.0.0       Hmisc_4.2-0         Formula_1.2-3
## [22] survival_2.44-1.1   caret_6.0-84        lattice_0.20-38
## [25] data.table_1.12.2   lubridate_1.7.4     forcats_0.4.0
## [28] stringr_1.4.0       dplyr_0.8.3         purrr_0.3.2
## [31] readr_1.3.1         tidyr_0.8.3         tibble_2.1.3
## [34] ggplot2_3.2.1       tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
##  [1] nlme_3.1-141        doParallel_1.0.15   webshot_0.5.1
##  [4] httr_1.4.1          tools_3.5.3         backports_1.1.4
##  [7] R6_2.4.0            rpart_4.1-15        lazyeval_0.2.2
```

```
## [10] colorspace_1.4-1    nnet_7.3-12          withr_2.1.2
## [13] tidyselect_0.2.5    gridExtra_2.3        compiler_3.5.3
## [16] cli_1.1.0           rvest_0.3.4          htmlTable_1.13.1
## [19] xml2_1.2.2          labeling_0.3         scales_1.0.0
## [22] checkmate_1.9.4     mvtnorm_1.0-11       nnls_1.4
## [25] quadprog_1.5-7      digest_0.6.20        foreign_0.8-72
## [28] rmarkdown_1.15      base64enc_0.1-3      pkgconfig_2.0.2
## [31] htmltools_0.3.6     htmlwidgets_1.3      rlang_0.4.0
## [34] readxl_1.3.1        rstudioapi_0.10      generics_0.0.2
## [37] jsonlite_1.6        acepack_1.4.1        ModelMetrics_1.2.2
## [40] magrittr_1.5        Matrix_1.2-17        Rcpp_1.0.2
## [43] munsell_0.5.0       partykit_1.2-5       stringi_1.4.3
## [46] yaml_2.2.0          inum_1.0-1           MASS_7.3-51.4
## [49] recipes_0.1.6       grid_3.5.3           crayon_1.3.4
## [52] haven_2.1.1         hms_0.5.1            zeallot_0.1.0
## [55] pillar_1.4.2        igraph_1.2.4.1       reshape2_1.4.3
## [58] codetools_0.2-16    stats4_3.5.3         glue_1.3.1
## [61] evaluate_0.14       latticeExtra_0.6-28  modelr_0.1.5
## [64] vctrs_0.2.0         cellranger_1.1.0     gtable_0.3.0
## [67] assertthat_0.2.1    xfun_0.9             gower_0.2.1
## [70] prodlim_2018.04.18  libcoin_1.0-5        broom_0.5.2
## [73] class_7.3-15        viridisLite_0.3.0    timeDate_3043.102
## [76] iterators_1.0.12    cluster_2.1.0        lava_1.6.6
## [79] ipred_0.9-9
```
