Predicting Apartment Prices in Queens

Final project for MATH 342W Data Science at Queens College

March 25th, 2025

By Sergio E. Garcia Tapia
In collaboration with:
Paidi Mather
Ye Htut Maung
Allen Singleton
Esther Yee

Abstract

Prediction is both an art and a science with important ramifications and applications. It can be used to get ahead of natural disasters by predicting aspects of natural disasters, or to earn money by exploiting information about the economy. Yet another modest application, and the one explored in this paper, is to predict the price of apartments. This matters to many, particularly to those who are attempting to attain the American dream by purchasing a home rather than a house. In this paper, we explore scientific and artistic aspects of data science used to make predictions in the context of predicting apartment prices.

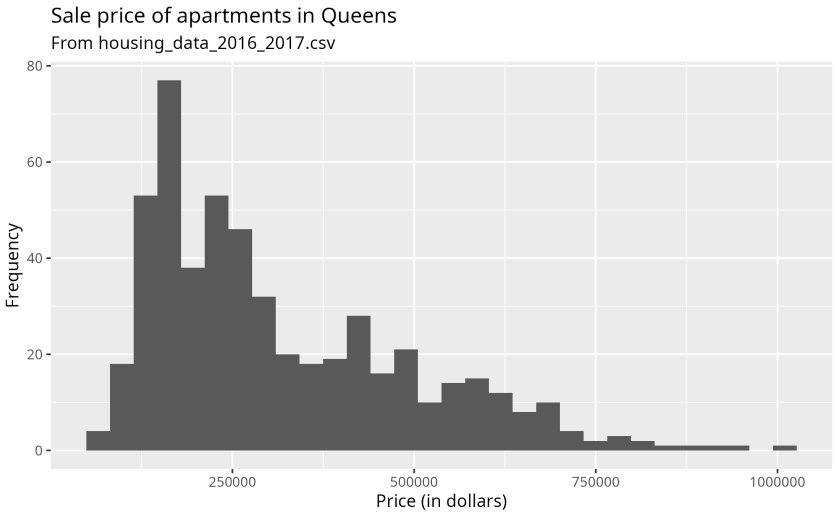

Figure 1: Histogram of sale prices for Queens apartments in the MLSLI data set.

1 Introduction

When residents in Queens, NY turn to home ownership, many consider buying a house instead of an apartment. Though apartments offer less opportunities for renovation and privacy, they can be a more reasonable investment for many. At first glance, predicting apartment prices may seem unnecessary because apartments that are available for sale tend to list a price. However, sellers may withhold the price in hopes that people bid on their property, or a buyer may be curious about an apartment with specific characteristics that is not currently available for sale. Given historical information about apartment listings, we can build a predictive model that can be of use in these situations.

A predictive model is a function used to make approximations about a phenomenon of interest using some given data. In this context, we are approximating the sale price of apartments in Queens, NY. More concretely, a predictive model is designed to answer the question: given a unit of observation represents an apartment, with a set of features including but not limited to number of bedrooms and bathrooms, location, and whether cats are allowed, what is its sale price? Using the same historical data, we will apply a variety of modeling approaches, namely a regression tree model, a linear model, and a random forests model. We will explore the trade-offs made by these models, mainly in account of interpretation and predictive performance.

2 The Data

The data set used in this project consists of 2230 observations obtained from MLSLI via Amazon Mturk. Each observation has 55 columns, of which about 28 include only metadata information related to Mturk, but is otherwise unrelated to the entries. Each observation corresponds to an apartment listing that was available for sale between February 2016 and 2017, with information such as: whether pets are allowed, the year the apartment was built, the number of rooms, and the kitchen type, among others. Unfortunately, 2230 is not a large enough sample to build a strong model for predicting apartment prices in Queens, considering that there are millions of units in NYC. Thus, the models we build are not expected to extrapolate well. Figure 2 shows a histogram of the sale prices for the observations that list it. From this right-skewed data set, we may have potential outliers on the right-end with apartments priced at $999,999. From this we expect the model predictions may be more accurate for apartments in the $175,000-$275,000 range.

2.1 Featurization

In my model I am using 21 features. Table 2.1 shows a summary of the nominal features. Table 2 shows a summary of the continuous features. From the nominal features, the `region` was not initially available. To create it, I parsed the ZIP codes from the `URL`, `full_address_or_zip_code`, and `url` columns in the raw data set. Then, I grouped the ZIP codes according to the region they belong to, dropping the three aforementioned columns thereafter. I did this with the intent of obtaining better interpretability, possibly at the expense of predictive power. Among the continuous features, I featurized `monthly_charges` from the `common_charges` and `maintenance_cost` columns in the raw data set. The rationale is that these correspond to condos and co-ops, respectively, rather than accepting the missingness and imputing, it made sense to unify them into a single feature. The remaining columns were provided in the raw data, and some that do not appear, such as `listing_price_to_nearest_1000` and `date_of_sale`, were selected for dropping after judging them to be inappropriate predictors of the `sale_price`.

2.2 Errors and Missingness

When I first loaded the data set into R, I looked at each column in turn to familiarize of myself with the values and get a sense for how much data was missing. Of the available data, there were many text input errors. The following are some examples:

- In the `dogs_allowed` column, there were entries such as `yes89` which were likely meant to be `yes`.
- In the `garage_exists` column, there were misspellings such as `eys` and abbreviations such as `UG` for `underground`.
- In the `kitchen_type` column, there were entries such as `efficiemcy`, `efficiency kitchene`, and `efficiency kitchen`. All of these were subsumed under `efficiency`.

For each such error, I either corrected to the value I presumed was correct, placed them into a pre-existing `other` category, or set them as missing. I used regular expressions to streamline the detections and changes.

The data set exhibited a lot of missingness as shown in Table 2.1 and Table 2. In some cases, missingness resulted from certain patterns in the data. For example, `common_charges` and `maintenance_cost` were condo or co-op specific, respectively. Since there was no overlap between the columns (i.e., no row had a present value in both columns), I combined them according to the value of `coop_condo` into a single column that retained their quantities. This condensing reduced missingness without needing to drop or manually alter the data. For `garage_exists`, there were in fact 1826 `NA` entries, while all present entries had the value `yes`. In this case, I decided to change all `NA` entries to `no`, making a possibly unjustified assumption about the respondent's original intent.

The data set also exhibited missingness in the response column, `sale_price`, with only 528 rows having a present value out of the total 2230. This meant that the model construction and validation could only rely on those 528 rows. However, rather than immediately discarding the rows for which `sale_price` was missing, I retained them imputation. Before imputing, I split the 528 rows into $\mathbb{D}_{\text{train}}$, $\mathbb{D}_{\text{select}}$, and $\mathbb{D}_{\text{test}}$. I did this in anticipation of the need to perform model validation and hyperparameter selection. I set aside the response values in $\boldsymbol{y}_{\text{select}}$ and $\boldsymbol{y}_{\text{test}}$ into a copy, and then for the purposes of imputing, I temporarily set the responses in $\mathbb{D}_{\text{select}}$ and $\mathbb{D}_{\text{test}}$ to missing (i.e. `NA`), to ensure that they did not influence the imputed values, lest we risk having a dishonest model. For simplicity I did not include a matrix M in the feature set that describes the missingness in the data set, which admittedly may have negatively affected performance of the trained models. To actually impute the values, I used the *MissForest* algorithm as implemented by the `missForest()` function in the `missForest` package in R.

Feature	Description	Missing Count	Percents
cats_allowed	Whether cats are allowed in the apartment.	0	No: 62.87% Yes: 37.13%
community_district_num	The number of the district in which an apartment is located.	19	24: 8.59% 25: 27.86% 26: 16.10% 27: 5.02% 28: 26.37% 29: 3.08% 30: 10.22% *other*: 2.76%
coop_condo	Whether an apartment is a co-op or a condo.	0	co-op: 74.48% condo: 25.52%
dining_room_type	The type of dining room in the apartment.	448	combo: 53.70% formal: 34.79% none: 0.11% other: 11.39%
dogs_allowed	Whether dogs are allowed in the apartment.	0	No: 75.52% Yes: 24.48%
fuel_type	The type of fuel using for heating, including cooking.	112	electric: 2.93% gas: 63.64% none: 0.01% oil: 31.35% other: 1.94%
garage_exists	Whether a garage is available at the premises.	0	No: 81.88% Yes: 18.12%
kitchen_type	The type of kitchen in the apartment.	17	combo: 18.03% eatin: 42.57% efficiency: 38.36% none: 1.04%
region	The region in Queens that an apartment belongs to, according ot its ZIP code.	0	central_queens: 5.43% jamaica: 6.41% north_queens: 24.93% northeast_queens: 8.03% northwest_queens: 3.50% southeast_queens: 6.77% southwest_queens: 9.19% west_central_queens: 20.49% west_queens: 15.25%

Table 1: Summary of nominal features in data set.

Feature	Description	Missing	Average	Std. Dev	Min	Max
approx_year_built	Approximate year when apartment was built.	39	1963	21.1	1893	2017
num_bedrooms	Number of bedrooms in apartment.	115	1.65	0.744	0	6
num_floors_in_building	Number of floors in the building of the apartment.	650	7.79	7.52	1	34
num_full_bathrooms	Number of full bathrooms in apartment.	0	1.23	0.445	1	3
num_half_bathrooms	Number of half bathrooms in apartment (no tub).	2058	0.953	0.302	0	2
num_total_rooms	Total number of rooms in apartment.	2	4.14	1.35	0	14
parking_charges	Charge for parking a vehicle in apartment premises.	1671	$107.57	$70.88	$6.00	$837.00
pct_tax_deductibl	Expense that can be subtracted from taxable income.	1754	45.40%	6.95%	20.00%	75.00%
sq_footage	Amount of area (space) in apartment in square feet.	1210	955	381	100	6215
total_taxes	Total dollar amount paid in taxes.	1646	$2,226.09	$1,850.09	$11.00	$9300.00
walk_score	A measure of access to amenities by walking.	0	83.9	14.8	7.00	99
monthly_charges	Co-op or condo specific charges.	193	$761.62	$395.87	$100.00	$4659.00

Table 2: Summary of continuous features.

3 Modeling

3.1 Regression Tree Modeling

The first candidate for a model I explored was a regression tree model. To compute the tree model, I used the `YARFCART` function in the `YARF` package. Since regression trees have a hyperparameter N_0 for the threshold node size, I employed a hyperparameter selection procedure to select an optimal value for N_0. I tried 75 different node sizes, each an integer between 1 and 75, each time building a model on $\mathbb{D}_{\text{train}}$ and computing the associated out-of-sample $RMSE$ by predicting on $\mathbb{D}_{\text{select}}$. I chose the node size corresponding to the smallest out-of-sample error $RMSE$, which resulted in $N_0 = 23$. Figure 2 illustrates the top 4 layers of the regression tree model. From the tree's top 5 layers (not depicted in that figure due to space), I determined the 10 most important features are:

- `approx_year_built`
- `sq_footage`
- `parking_charges`
- `monthly_charges`
- `total_taxes`
- `pct_tax_deductible`
- `num_floors_in_building`
- `coop_condo`
- `cats_allowed`
- `community_district_number`

The root split is according to whether the value of `approx_year_built` is at most 1963, possibly suggesting that buyers care about how old a building is. The next most significant split appears to be whether `sq_footage`, is around 800. It stands to reason that larger apartments tend to be more expensive. Though we can tell at a glance which are the most important features by inspecting the layers in the free, it's harder to discern their relative importance.

3.2 Linear Modeling

A line of best fit is a natural model to consider because of its simplicity. The advantage of a linear model is that its coefficients lend themselves to easy interpretations.

Coefficient	Value
Intercept	-233675.88743
approx_year_built	99.20360
cats_allowed1	2170.99809
community_district_num4	-107222.73430
community_district_num8	-109907.92626
community_district_num11	-49814.59426
community_district_num16	-98381.86284
community_district_num19	-45336.91857
community_district_num20	-150148.56494
community_district_num23	-75037.36398
community_district_num24	-92355.11922
community_district_num25	-16002.71146
community_district_num26	-9545.76940
community_district_num27	-44494.12820
community_district_num28	-58809.27350
community_district_num29	-45284.99205
community_district_num30	28255.00285
coop_condocondo	238345.93272
dining_room_typeformal	11296.81763
dining_room_typeother	17194.02611
dogs_allowed1	6835.33546
fuel_typegas	-8132.65876
fuel_typenone	16192.06722
fuel_typeoil	2664.79414
fuel_typeother	-4278.77459
garage_exists1	16921.39471
kitchen_typeeatin	-19453.39311
kitchen_typeefficiency	-38481.47712
num_bedrooms	37770.53423
num_floors_in_building	2338.81569
num_full_bathrooms	60791.15973
num_half_bathrooms	-9769.33225
num_total_rooms	7516.83982
parking_charges	444.93697
pct_tax_deductibl	-2476.48258
sq_footage	14.85828
total_taxes	14.09168
walk_score	305.14194
regionjamaica	-18065.91379
regionnorth_queens	45578.83425
regionnortheast_queens	42123.89355
regionnorthwest_queens	58988.52334
regionsoutheast_queens	46595.00855
regionsouthwest_queens	893.08745
regionwest_central_queens	91744.35725
regionwest_queens	55875.92791
monthly_charges	167.15893

Table 3: Coefficients for linear model.

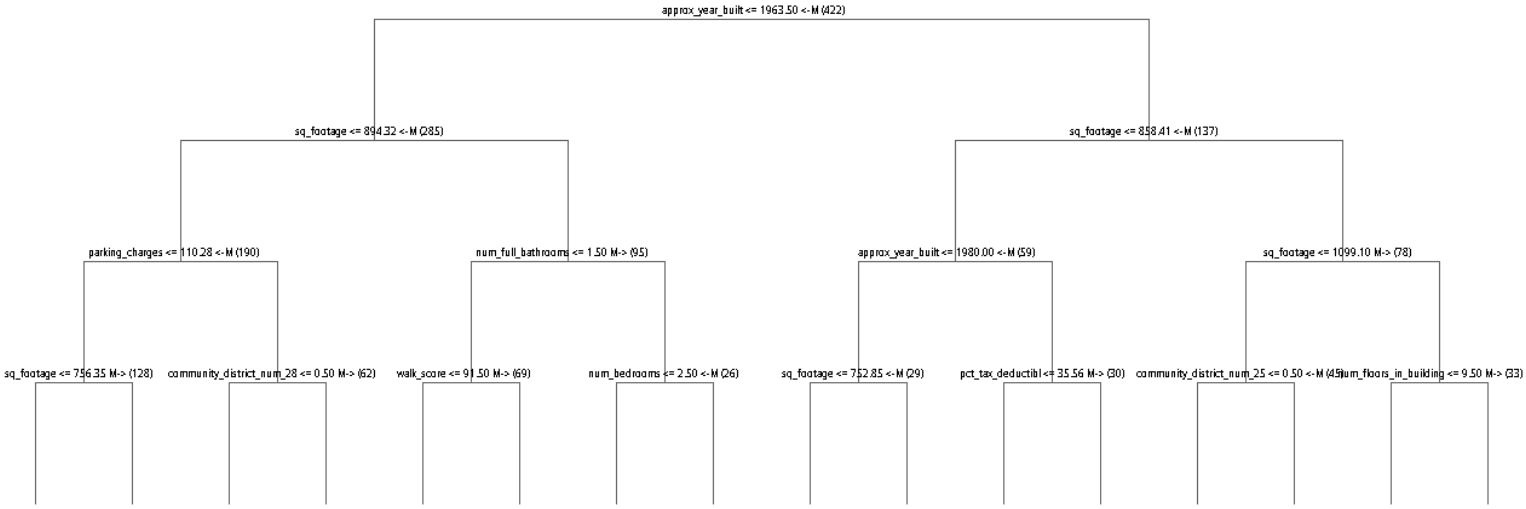

Figure 2: Top 4 layers in the regression tree model.

Table 3 shows the coefficients for the linear model computed. Consider, for example, the coefficient of `approx_year_built`, which was deemed most important by the regression tree model. Its corresponding coefficient in the linear model is 99.20360, which means that an increase by 1 year results in around \$99.20 change in `sale_price`. Meanwhile, a change in `sq_footage` appears to only imply an increase by \$14.86 in `sale_price`. The largest coefficients in absolute value appear to be `coop_condocondo`, where a flip implies a value change of about \$238,345.93, which may be because condos tend to be more expensive than co-ops. The coefficient `community_district_num20` has the largest absolute negative value, implying that deciding to purchase an apartment in this neighborhood implies the `sale_price` reduces by \$-150,148.56. The reduction may suggest that the district is generally undesirable as a place to live. These sorts of questions mirror how people normally think, weighing the consequence of a change in a particular characteristic.

How well does the linear model really do? Let's consider that by looking at the in-sample performance metrics:

$$\text{in-sample-}RMSE = 68649.9$$
$$\text{in-sample-}R^2 = 0.8697797$$

Let's also consider the corresponding out-of-sample metrics:

$$\text{out-of-sample-}RMSE = 80312.78$$
$$\text{out-of-sample-}R^2 = 0.801584$$

The increase in $RMSE$ and decrease in R^2 suggests the linear model is fitting some of the noise in the data. In spite of its seemingly impressive error metrics, the linear model fails to account for nonlinearities and interactions that surely exist in the data. Another issue is that linear models tend to extrapolate poorly. Indeed, this is the classic trade-off of interpretability and performance that we alluded to. These reasons make the linear model unsuitable for prediction.

3.3 Random Forest Modeling

In contrast with the linear model, the random forest model introduces the robustness necessary for complex interactions. Recall that the $RMSE$ metric is defined as $\sqrt{MSE}$, where MSE is the Mean-Squared Error. In turn, the error accrued in the MSE can be decomposed into bias and variance:

$$MSE := \text{Irreducible ignorance error} + \text{Bias} + \text{Variance}$$

The random forest model is effective because it simultaneously reduces the bias and variance. It reduces the bias by using regression trees that are strong learners, i.e., trees with low bias yet high variance. To reduce variance, it uses two techniques: bootstrap aggregation and parameter subsetting.

- In bootstrap aggregation, we compute M bootstrap samples, where for each time we sample with replacement out of the n observations in the data set $\mathbb{D}$. Then, on these M bootstrap samples $\mathbb{D}_1, \ldots, \mathbb{D}_M$, we compute corresponding tree models $g_1, \ldots, g_M$, and we compute their average $g_{\text{avg}} := \frac{1}{M}\sum_{m=1}^{M} g_m$. Each bootstrap has about two-thirds of the original data, and they all miss a slightly different one-third of the data. This results in the M models $g_1, \ldots, g_M$ having loosely coupled, hence they have low covariance, and hence g_{avg} has low variance.
- To take this a step further, random forest uses only a random subset of the features to build each tree model, and since the feature set used in each tree is slightly different, covariance further reduces.

Put together, these ensure the MSE and hence the $RMSE$ is reduced. From the description, however, we see that there are a few hyperpameters that need to be set: the threshold node size N_0 as in regression trees, the number of trees (effectively M as in the notation above), and the number m_{try} of features for subsetting. To select appropriate values of these hyperparameters, I employed a hyperparameter selection procedure by creating a three-dimensional grid of values, where each value was a triplet (m_{try}, M, N_0). I leveraged my $\mathbb{D}_{\text{select}}$ and computed a random forest model for each triplet, in the end using the one that gave the lowest $RMSE$. The optimal hyperparameter values according to this procedure were:

$$\begin{aligned} m_{\text{try}} &= 19 \\ M &= 20 \\ N_0 &= 2 \end{aligned}$$

The elaborate scheme described succeeds in improving performance, as can be seen from the out-of-sample performance metrics:

$$\begin{aligned} \text{out-of-sample-}RMSE &= 74097.86 \\ \text{out-of-sample-}R^2 &= 0.8311043 \end{aligned}$$

This is an improvement over the both the linear model and the single tree regression model. There is, however, some overfitting occurring, apparently more so than in the linear case:

$$\begin{aligned} \text{in-sample-}RMSE &= 34702.89 \\ \text{in-sample-}R^2 &= 0.9626422 \end{aligned}$$

The overfitting may due to the relatively large number of features compared to the size of the data set. Moreover, although there was an improvement out-of-sample, we gained this at the expense of interpretability. Indeed, it is much harder to look at 20 different trees simultaneously, each using different features, and somehow understand how the model uses them to compute its prediction. Nevertheless, this model is the most flexible, is capable of accounting for complex interactions between the features, and generalizes well.

Model	in-sample-RMSE	oosRMSE	in-sample-R^2	oosR^2
Regression Tree Model	$78,124.35	$101,784.00	0.8106687	0.6813115
Linear Model	$68,649.90	$80,312.78	0.8697797	0.801584
Random Forest Model	$34,702.89	$74,097.86	0.9626422	0.8311043

Table 4: Out-of-sample metrics for the three models we have considered.

4 Performance Results

Table 4 summarizes the performance of the three models we considered. We see that the random forest beats the linear model by a small margin. The R^2 metric for all models is relatively close to 1, implying that the proportion of variance explained is high. We can also say based on the out-of-sample $RMSE$ of the random forest model that the predictions will fall within $148,195.72 (twice the $oosRMSE$) of the mean `sale_price` $314,956.60. This still quite a big range, and in fact it may not be useful enough for someone using the model to make an actionable decision related to an apartment purchase.

The degradation in the error metrics is sharper in the tree and random forest models, suggesting that they are more prone to overfitting in this setting. The random forest model beats the linear model on the basis of its out-of-sample metrics. Moreover as we discussed earlier, it is likely to generalize better because it can handle interactions and nonlinearities in the data. However, due to the small sample size, and the fact that I did not do cross-validation, the out-of-sample metrics are not completely reliable.

5 Discussion

In this project, I compared the effectiveness of three different models in predicting apartment prices in Queens from 2016 to 2017. I familiarized myself with the data through data wrangling, cleaning, and employing missing data mechanisms. Working with the data enabled me to apply three predictive models that ranged in their simplicity, interpretability, and performance: a regression tree model, a linear model, and a random forest model. In the end, the random forest model was determined to be the most effective.

In spite of these results, my approach was lacking in many areas. I made assumptions about features with a lot of missingness such as `garage_exists` which may have had an unpredictable effect on the performance of my models. I also certain features such as the ZIP code into regions and possibly lost performance from it. I also noticed that some of my models suffered from not having all categories of certain features represented, and when going to predict out-of-sample, it resulted in these features ultimately being ignored, thereby biasing results (for example, community district number).

Overall, I could have been more thoughtful in regards to feature engineering. To do a better job, I would work harder to understand the housing space and how the features interact. I would also search for ways to supplement the data set, with more observations and more features. To address the issue of representation, I would perform stratified sampling.

Yet another area of improvement is in regards to error metrics. Since I did not perform cross-validation, the out-of-sample metrics are not very reliable in spite of being honest. As it stands, my random forest model is not production-ready, and it is not at the stage where it can beat a model curated by Zillow. However, by implementing some of the improvements suggested earlier, it may be able to improve enough to at least be useful.

Code Appendix

```
---
title: "Final Modeling Project for MATH 342W"
author: "Sergio E. Garcia Tapia"
output: pdf_document
---

# Final MATH 342W Project: Predicting Apartment Prices

We want to predict apartment prices in Queens, NY using the housing data
provided.

## Loading and Cleaning the Data

We can begin by loading the data into a data frame and viewing a summary of its
contents.

```{r}
# data.table provides fread(), and skimr provides skim()
pacman::p_load(skimr, data.table)
raw_df = fread("housing_data_2016_2017.csv")

# Provide a summary of the values and columns.
skim(raw_df)

# Display a list of column names.
colnames(raw_df)

raw_df
```

In what follows, we will "clean" the data, thereby modifying the original data
frame. In case we may need the original data, we will perform this cleanup on a
copy:

```{r}
apt_df = copy(raw_df)
```

Since the data was collected using Amazon MTurk, the first 27 columns have
metadata information that we can remove:

```{r}
# Remove all metadata columns.
metadata_columns = c(1:27)
apt_df = apt_df[, (metadata_columns) := NULL]
rm(metadata_columns)

cat("Remaining columns:\n\n")
colnames(apt_df)
```
```

```
cat("\nTotal rows:", nrow(apt_df), "\n")
```

In what follows, I will look at each column in turn, viewing some sample values, looking for oddities that may need to be fixed, and deciding whether the information is valuable enough to be kept.

`URL`

```{r}
cat("First few URLs:\n\n")
head(apt_df$URL)

cat("\nNumber of missing URLs:\n")
sum(is.na(apt_df$URL))
```

The `URL` column lists the URL for each apartment listing. About a third of the rows do not have an associated `URL`. The first few entries displayed above show that address information about a given apartment is available in the URL. Incidentally, the column listing shows that there are other columns, `url` and `full_address_or_zip_code`, which may also contain address information.

The simple approach I will take is to extract the ZIP code from each `URL` into a new column called `zip_code`. When we look at `full_address_or_zip_code` and `url` later, we can do the same for any `zip_code` values that may still not be present due to their absence in `URL`.

```{r}
# Obtain the str_match() function to extract ZIP code with a regular expression.
pacman::p_load(tidyverse)

# In URL, zip codes consist of 5 digits surrounded by dashes
apt_df[, zip_code := as.factor(str_match(apt_df$URL, "-(\\d{5})-")[, 2])]
cat("Extracted", nrow(apt_df[!is.na(zip_code)]), "zip codes out of",
    nrow(apt_df[!is.na(URL)]), "URLs:\n")
apt_df[, URL := NULL]

table(apt_df$zip_code)
cat("\n", length(table(apt_df$zip_code)), "different zip codes.\n")
```

`approx_year_built`

```{r}
cat("First few approx_year_built values:\n")
head(apt_df$approx_year_built)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$approx_year_built))
```

```
cat("\nCounts for approx_year_built values:\n")
table(apt_df$approx_year_built)
```

The `approx_year_built` column stores the approximate year in which a given apartment was built. I will keep this as an integer build.

`cats_allowed`

```{r}
cat("First few cats_allowed values:\n")
head(apt_df$cats_allowed)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$cats_allowed))

cat("\nCounts for cat_allowed values:")
table(apt_df$cats_allowed)
```

The `cats_allowed` column encodes whether cats are allowed by the tenants on the premises. We see that there are no missing values here. However, there are categories for `y` and `yes`, which are probably both meant to be `yes`. We convert and `y` entry to `yes` and then treat the column as a categorical variable:

```{r}
apt_df[cats_allowed == 'y', cats_allowed := 'yes']
apt_df[, cats_allowed := as.factor(as.numeric(cats_allowed == 'yes'))]
table(apt_df$cats_allowed)
```

`common_charges`

```{r}
cat("First few common_charges values:\n")
head(apt_df$common_charges)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$common_charges))
```

The `common_charges` column lists monthly payments that tenants would need to make while living in the apartment. Most of the rows do not list such a value, which may mean that either the agent did not disclose the amount, or that there are no such charges, but we cannot be sure which. We can convert the non-NA values to integers. In anticipation of needing to do this for other columns with dollar values, such as `sale_price`, I'll package it as a function that can be reused.

```{r}
```

```r
#' Strips commas, spaces, and $ symbols from the given string.
#' @param dollar_str  A string representing an integer dollar amount.
#' @return            A string without the symbols £, commas, or spaces.
dollars_string_to_int = function(dollar_str) {
  str_replace_all(dollar_str, "[$,\\s]", "")
}
```

The function does not do the conversion to integer since it needs to done for the entire column anyway.

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df[!is.na(common_charges), common_charges := dollars_string_to_int(common_charges)]
# Convert dollar string to integer.
apt_df[, common_charges := as.numeric(common_charges)]
```

`community_district_num`

```{r}
cat("First few community_district_num values:\n")
head(apt_df$community_district_num)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$community_district_num))

cat("\nCounts for community_district_num:\n")
table(apt_df$community_district_num)
```

The `community_district_num` refers to the community district number of the neighborhood that apartment is in. Though the entries are numbers, they are really labels that should be treated as categorical data. Hence we factorize:

```{r}
apt_df[, community_district_num := factor(community_district_num)]
```

`coop_condo`

```{r}
cat("First few coop_condo values:\n")
head(apt_df$coop_condo)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$coop_condo))

cat("\nCounts for coop_condo:\n")
table(apt_df$coop_condo)
```

```

`coop_condo` classifies an apartment as a co-operative or a condominium.
Assuming these are the only valid categories, we can create a factor variable
from it:

```{r}
apt_df[, coop_condo := factor(coop_condo)]
```

`data_of_sale`

```{r}
cat("First few date_of_sale values:\n")
head(apt_df$date_of_sale)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$date_of_sale))
```

The `date_of_sale` contains the date when the apartment in the listing was sold.
Many entries have missing values, likely because not all apartments were sold at
the time. Whether the apartment was sold or not does not necessarily influence
its sales price, so we can drop it:

```{r}
apt_df[, date_of_sale := NULL]
```

`dining_room_type`

```{r}
cat("First few dining_room_type values:\n")
head(apt_df$dining_room_type)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$dining_room_type))

cat("\nCounts for dining_room_type:\n")
table(apt_df$dining_room_type)
```

The `dining_room_type` field lists the type of dining room in the apartment.
The values are:

- `combo`: A combined dining room and living room area.
- `formal`: A dedicated space for sit-down meals.
- `other`: A category that does not fall in one of the former two.
- `none`: No dining room.

There is a `dining area` entry which is likely an input error corresponding to
```

````
one of the other two, and since we don't know which, we will take the safe route
and put it into `other`.

```{r}
apt_df[dining_room_type == 'dining area', dining_room_type := 'other']
table(apt_df$dining_room_type)
apt_df[, dining_room_type := factor(dining_room_type)]
```

### `dogs_allowed`

```{r}
cat("First few dogs_allowed values:\n")
head(apt_df$dogs_allowed)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$dogs_allowed))

cat("\nCounts for dogs_allowed:\n")
table(apt_df$dogs_allowed)
```

This is similar to the `cats_allowed` field. We ought to fix the entries that
say `yes89`, which are likely input errors, by converting them into `yes`.

```{r}
apt_df[dogs_allowed == 'yes89', dogs_allowed := 'yes']
apt_df[, dogs_allowed := as.factor(as.numeric(dogs_allowed == 'yes'))]
table(apt_df$dogs_allowed)
```

### `fuel_type`

```{r}
cat("First few fuel_type values:\n")
head(apt_df$fuel_type)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$fuel_type))

cat("\nCounts for fuel_type:\n")
table(apt_df$fuel_type)
```

The `fuel_type` refers to what is used for heating, water heating, and cooking.
In this case we see the most prevalent fuel types are `electric`, `gas`,
and `oil`. We will need to combine combine `Other` and `other`.

```{r}
apt_df[fuel_type == 'Other', fuel_type := 'other']
apt_df[, fuel_type := factor(fuel_type)]
````
````

```r
table(apt_df$fuel_type)
```

`full_address_or_zip_code`

```{r}
cat("First few full_address_or_zip_code values:\n")
head(apt_df$full_address_or_zip_code)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$full_address_or_zip_code))
```

The `full_address_or_zip_code` gives the location of the apartment for sale. As pointed out earlier, we will simply extract the ZIP code, and use it to populate any missing entries in the `zip_code` column we created earlier:

```{r}
apt_df[
  is.na(zip_code),
  zip_code := str_match(full_address_or_zip_code, "[,\\s](\\d{5})")[, 2]
]
nrow(apt_df[is.na(zip_code)])
apt_df[is.na(zip_code) & !is.na(full_address_or_zip_code), .(full_address_or_zip_code)]
apt_df[, full_address_or_zip_code := NULL]
```

Note that some ZIP codes did not match the regular expression, and we can see why from the data frame displayed above: they have mistakes. For example, the the address entry `35-25 77 St, Jackson Heights NY, 1137` is cleraly missing a digit at the end. I will return to this if later when extracting ZIP codes if there is still missingness.

`garage_exists`

```{r}
cat("First few garage_exists values:\n")
head(apt_df$garage_exists)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$garage_exists))

cat("\nCounts for garage_exists:\n")
table(apt_df$garage_exists)
```

All of these seem to imply "yes" to the question of whether there is a garage. Moreover I will assume that `NA` means `"no"`.

```{r}
apt_df[!is.na(garage_exists) & garage_exists != 'yes', garage_exists := 'yes']
```

```
apt_df[is.na(garage_exists), garage_exists := 'no']
apt_df[, garage_exists := as.factor(as.numeric(garage_exists == 'yes'))]
table(apt_df$garage_exists)
```

`kitchen_type`

```{r}
cat("First few kitchen_type values:\n")
head(apt_df$kitchen_type)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$kitchen_type))

cat("\nCounts for kitchen_type:\n")
table(apt_df$kitchen_type)
```

The `kitchen_type` field describes the type of kitchen available. It makes sense to use a categorical variable in this scenario. Also, there is a `1955` entry that my in fact have to do with the year the apartment was built:

```{r}
apt_df[str_detect(kitchen_type, "^[Ee]a"), kitchen_type := 'eatin']
apt_df[str_detect(kitchen_type, "^[Ee]ff"), kitchen_type := 'efficiency']
apt_df[str_detect(kitchen_type, "^Combo"), kitchen_type := 'combo']
apt_df[kitchen_type == '1955']
apt_df[kitchen_type == '1955', approx_year_built := 1955]
apt_df[kitchen_type == '1955', kitchen_type := NA]
apt_df[, kitchen_type := factor(kitchen_type)]
table(apt_df$kitchen_type)
```

`maintenance_cost`

```{r}
cat("First few maintenance_cost values:\n")
head(apt_df$maintenance_cost)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$maintenance_cost))
```

`maintenance_cost` likely describes services such as HVAC, plumbing, painting, and so on. In any case, We can apply a similar transformation that we applied to `common_charges`:

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df[!is.na(maintenance_cost),
```

```
       maintenance_cost := dollars_string_to_int(maintenance_cost)]
# Convert dollar string to integer.
apt_df[, maintenance_cost := as.numeric(maintenance_cost)]
```

`model_type`

```{r}
cat("First few model_type values:\n")
head(apt_df$model_type)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$model_type))

#cat("\nCounts for model_type:\n")
#table(apt_df$model_type)
```

It's unclear what `model_type` is supposed to represent, since the values do not have uniformity. Bluntly, the data is garbage, so it makes sense to drop it:

```{r}
apt_df[, model_type := NULL]
```

`num_bedrooms`

```{r}
cat("First few num_bedrooms values:\n")
head(apt_df$num_bedrooms)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$num_bedrooms))

cat("\nCounts for num_bedrooms:\n")
table(apt_df$num_bedrooms)
```

`number_of_bedrooms` can be kept as an integer field in case a new apartment becomes available with more than 6 bedrooms (as opposed to using a factor that would not allow for this).

`num_floors_in_build`

```{r}
cat("First few num_floors_in_build values:\n")
head(apt_df$num_floors_in_build)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$num_floors_in_build))
```

```
cat("\nCounts for num_floors_in_build:\n")
table(apt_df$num_floors_in_build)
```

We leave this as an integer (and not a factor) for a similar reason as in `num_bedrooms`.

`num_full_bathrooms`

```{r}
cat("First few num_full_bathrooms values:\n")
head(apt_df$num_full_bathrooms)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$num_full_bathrooms))

cat("\nCounts for num_full_bathrooms:\n")
table(apt_df$num_full_bathrooms)
```

We leave this as an integer (and not a factor) for a similar reason as in `num_bedrooms`.

`num_half_bathrooms`

```{r}
cat("First few num_half_bathrooms values:\n")
head(apt_df$num_half_bathrooms)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$num_half_bathrooms))

cat("\nCounts for num_half_bathrooms:\n")
table(apt_df$num_half_bathrooms)
```

We leave this as an integer (and not a factor) for a similar reason as in `num_bedrooms`.

`num_total_rooms`

```{r}
cat("First few num_total_rooms values:\n")
head(apt_df$num_total_rooms)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$num_total_rooms))

cat("\nCounts for num_total_rooms:\n")
table(apt_df$num_total_rooms)
```

````
We leave this as an integer (and not a factor) for a similar reason
as in `num_bedrooms`.

### `parking_charges`

```{r}
cat("First few parking_charges values:\n")
head(apt_df$parking_charges)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$parking_charges))

cat("\nCounts for parking_charges:\n")
table(apt_df$parking_charges)
```

We apply the same transformation as in `common_charges` to convert dollar
strings to integers:

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df[!is.na(parking_charges),
       parking_charges := dollars_string_to_int(parking_charges)]
# Convert dollar string to integer.
apt_df[, parking_charges := as.numeric(parking_charges)]
apt_df[!is.na(parking_charges)]$parking_charges
```

### `pct_tax_deductibl`

```{r}
cat("First few pct_tax_deductibl values:\n")
head(apt_df$pct_tax_deductibl)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$pct_tax_deductibl))

cat("\nCounts for pct_tax_deductibl:\n")
table(apt_df$pct_tax_deductibl)
```

This percentage values can be kept as an integer.

### `sale_price`

```{r}
cat("First few sale_price values:\n")
head(apt_df$sale_price)

cat("\nNumber of missing values:\n")
````
````

```
sum(is.na(apt_df$sale_price))
```

`sale_price` is the sale price of an apartment, and it is the response column that we are interested in predicting. Many rows have missing values, which means we will not be able to use all rows of the data set as training data. However, rather than dropping units with missingness in this column, we can use those units to help impute missing values for other entries in the data. Beforehand, we will convert the dollar strings to integers again:

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df[!is.na(sale_price), sale_price := dollars_string_to_int(sale_price)]
# Convert dollar string to integer.
apt_df[, sale_price := as.numeric(sale_price)]
```

`sq_footage`

```{r}
cat("First few sq_footage values:\n")
head(apt_df$sq_footage)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$sq_footage))
```

`sq_footage` describes the amount of space available in the apartment. We can leave this field as an integer.

`total_taxes`

```{r}
cat("First few total_taxes values:\n")
head(apt_df$total_taxes)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$total_taxes))
```

We can apply the dollar string to integer conversion from earlier again:

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df[!is.na(total_taxes), total_taxes := dollars_string_to_int(total_taxes)]
# Convert dollar string to integer.
apt_df[, total_taxes := as.numeric(total_taxes)]
```

`walk_score`

```{r}
cat("First few walk_score values:\n")
head(apt_df$walk_score)

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$walk_score))

cat("\nCounts for walk_score:\n")
table(apt_df$walk_score)
```

We can keep the `walk_score` as an integer.

`listing_price_to_nearest_1000`

```{r}
cat("A few listing_price_to_nearest_1000 values:\n")
head(apt_df[!is.na(listing_price_to_nearest_1000), listing_price_to_nearest_1000])

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$listing_price_to_nearest_1000))
```

This quantity is the amount the seller is an initial asking price when the seller placed the apartment for sale. This quantity is like a prediction itself, which may skew our predictions. Put another way, not all apartments for which we will be predicting prices will list such a quantity because they were not all necessary for sale. In fact let's inspect the `sale_price` and the `listing_price_to_nearest_1000` columns side by side:

```{r}
apt_df[, .(sale_price, listing_price_to_nearest_1000)]
```

The listing prices are missing observations where the `sale_price` is present. Therefore if we kept `listing_price_to_nearest_1000`, all entries would be missing for that column in the training set. I believe it's best to drop this column altogether, but we will do that after imputing. Before that, though, we need to change the values to be numeric:

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df[!is.na(listing_price_to_nearest_1000),
       listing_price_to_nearest_1000 := dollars_string_to_int(listing_price_to_nearest_1000
# Convert dollar string to integer.
apt_df[, listing_price_to_nearest_1000 := as.numeric(listing_price_to_nearest_1000)]
```

`url`

```{r}


```

```
cat("First few non-missing values:\n")
head(apt_df[!is.na(url), url])

cat("\nNumber of missing values:\n")
sum(is.na(apt_df$url))
```

As mentioned earlier, we can simply extract the ZIP code and populate any missing `zip_code` entries:

```{r}
apt_df[
  is.na(zip_code),
  zip_code := str_match(url, "-(\\d{5})-")[, 2]
]
apt_df[, url := NULL]
sum(is.na(apt_df$zip_code))
table(apt_df$zip_code)
```

`zip_code`

Below, I ensure that whatever ZIP codes I matched do indeed fall in the regions of Queens that are predicting for:

```{r}
extracted_zips = as.numeric(names(unlist(table(apt_df$zip_code))))

northeast_queens = c(11361, 11362, 11363, 11364)
north_queens = c(11354, 11355, 11356, 11357, 11358, 11359, 11360)
central_queens = c(11365, 11366, 11367)
jamaica = c(11412, 11423, 11432, 11433, 11434, 11435, 11436)
northwest_queens = c(11101, 11102, 11103, 11104, 11105, 11106)
west_central_queens = c(11374, 11375, 11379, 11385)
southeast_queens = c(11004, 11005, 11411, 11413, 11422, 11426, 11427, 11428, 11429)
southwest_queens = c(11414, 11415, 11416, 11417, 11418, 11419, 11420, 11421)
west_queens = c(11368, 11369, 11370, 11372, 11373, 11377, 11378)

queen_zips = c(northeast_queens, north_queens, central_queens, jamaica,
                   northwest_queens, west_central_queens, southeast_queens,
                   southwest_queens, west_queens)
zips_outside_domain_of_interest = setdiff(extracted_zips, queen_zips)
if (length(zips_outside_domain_of_interest) > 0) {
  cat("Warning: ZIP codes not in the areas in Queens of interest:\n")
  cat(zips_outside_domain_of_interest, "\n")
} else {
  cat("All ZIP codes fall within the regions of interest.")
}
```

In an attempt to make the upcoming models easier to interpret while risking a

loss of predictive power, I will group the zip codes into the regions implied earlier, and then drop the `zip_code` column.

```{r}
apt_df[zip_code %in% northeast_queens, region := "northeast_queens"]
apt_df[zip_code %in% north_queens, region := "north_queens"]
apt_df[zip_code %in% central_queens, region := "central_queens"]
apt_df[zip_code %in% jamaica, region := "jamaica"]
apt_df[zip_code %in% northwest_queens, region := "northwest_queens"]
apt_df[zip_code %in% west_central_queens, region := "west_central_queens"]
apt_df[zip_code %in% southeast_queens, region := "southeast_queens"]
apt_df[zip_code %in% southwest_queens, region := "southwest_queens"]
apt_df[zip_code %in% west_queens, region := "west_queens"]
apt_df[, zip_code := NULL]
apt_df[, region := as.factor(region)]
apt_df
```

Remaining Feature Set

During the clean-up, we created a `zip_code` column and eliminated `URL`, `full_address_or_zip_code`, and `url`. We also eliminated `model_type` because it seemed most of its data lacked uniformity. We further dropped the `listing_price_to_nearest_1000` column.

A further change we can make is to combine the information in the columns named `common_charges` and `maintenance_cost` into a single column. This is because `common_charges` typically correspond to condos, and `maintenance_cost` typically corresponds to co-ops (thanks to Allen for pointing this out). We will create a new column named `monthly_charges` and drop the previous ones:

```{r}
apt_df[coop_condo == 'condo', monthly_charges := common_charges]
apt_df[coop_condo == 'co-op', monthly_charges := maintenance_cost]
apt_df[, common_charges := NULL]
apt_df[, maintenance_cost := NULL]
```

After all of these changes, the following columns remain as our feature set:

```{r}
colnames(apt_df)
ncol(apt_df)
```

The following summarizes the missingness and statistics across the remaining feature set:

```{r}
skim(apt_df)
```

```
proportions(table(apt_df$cats_allowed))
proportions(table(apt_df$community_district_num))
proportions(table(apt_df$coop_condo))
proportions(table(apt_df$dining_room_type))
proportions(table(apt_df$dogs_allowed))
proportions(table(apt_df$fuel_type))
proportions(table(apt_df$garage_exists))
proportions(table(apt_df$kitchen_type))
proportions(table(apt_df$region))
```

Missingness and Imputation

Having cleaned up the data, we can use an imputation library to fill missing values. We ought to be careful, though, since there are rows that do not contain a `sale_price`. Let's re-arrange our data frame so that all present values appear first:

```{r}
# Move sale_price to last column
setcolorder(apt_df, c("sale_price"), after = ncol(apt_df))
apt_df_with_responses = apt_df[!is.na(sale_price)]
apt_df_without_responses = apt_df[is.na(sale_price)]
```

We cannot use the rows that have a missing response for building our data set, but we can use them for imputation. Before imputing, we will create a train-select-test split from the rows with present responses:

```{r}
set.seed(16)
K = 5
n = nrow(apt_df_with_responses)

n_select = ceiling(n / K)
n_test = n_select
n_train = n - n_select - n_test

test_idx = sample(1 : n, n_test, replace = FALSE)
select_idx = sample(setdiff(1 : n, test_idx), n_select, replace = FALSE)
train_idx = setdiff(1 : n, c(test_idx, select_idx))
```

To be careful about the test set influencing the imputation, we will set aside the responses from the test set and `NA` them for the purpose of imputation:

```{r}
y_test = apt_df_with_responses$sale_price[test_idx]
y_select = apt_df_with_responses$sale_price[select_idx]
apt_df_with_responses[test_idx, sale_price := NA]
apt_df_with_responses[select_idx, sale_price := NA]
```

```

Now we can impute:

```{r}
apt_df_responses_at_start = rbind(apt_df_with_responses, apt_df_without_responses)
pacman::p_load(missForest)
apt_df_imp = missForest(apt_df_responses_at_start)$ximp
```

As mentioned earlier, we will just drop the listing price now, which we only kept for the purpose of imputing:

```{r}
# Remove dollar symbols, commas, and spaces.
apt_df_imp[, listing_price_to_nearest_1000 := NULL]
skim(apt_df_imp)
```

The imputation may have produced continuous values for integer data, such as `num_bedrooms`, so let's fix values first by rounding up:

```{r}
apt_df_imp[, num_bedrooms := ceiling(num_bedrooms)]
apt_df_imp[, num_floors_in_building := ceiling(num_floors_in_building)]
apt_df_imp[, num_full_bathrooms := ceiling(num_full_bathrooms)]
apt_df_imp[, num_half_bathrooms := ceiling(num_half_bathrooms)]
apt_df_imp[, num_total_rooms := ceiling(num_total_rooms)]
apt_df_imp[, approx_year_built := ceiling(approx_year_built)]
apt_df_imp[, walk_score := ceiling(walk_score)]
```

Having populated missing values, we can extract our data set, and reintroduce the test responses that we set aside earlier:

```{r}
X = apt_df_imp[1 : n, !c("sale_price")]
y = apt_df_imp[1 : n]$sale_price
y[test_idx] = y_test
y[select_idx] = y_select
```

Finally, we separate our train and test data (note the response for the test set was already set aside earlier):

```{r}
X_test = X[test_idx]
X_select = X[select_idx]

X_train = X[train_idx]
y_train = y[train_idx]
```
```

```

Let's take a quick look at the sale prices graphically:

```{r}
pacman::p_load(ggplot2)
ggplot(data.frame(y = y)) +
  aes(y) +
  geom_histogram() +
  ggtitle("Sale price of apartments in Queens",
          subtitle = "From housing_data_2016_2017.csv") +
  xlab("Price (in dollars)") +
  ylab("Frequency")
ggsave('./report/images/apartment_sale_price_histogram.png')
```

We can supplement this graphic with a 5-number summary of the sale prices:

```{r}
summary(y)
```

We see a right-skewed distribution with a potential outlier priced at 999,999 dollars.

```{r}
skim(apt_df_imp)
```

Regression Tree Modeling

For our first approach, we will fit a single regression tree. We will use the `YARF` package:

```{r}
if (!pacman::p_isinstalled(YARF)){
  pacman::p_install_gh("kapelner/YARF/YARFJARs", ref = "dev")
  pacman::p_install_gh("kapelner/YARF/YARF", ref = "dev", force = TRUE)
}
options(java.parameters = "-Xmx4000m")
pacman::p_load(YARF)
```

Fitting a single tree model is easy, but we ought to find the right hyperparameter `N_0` for the tree size:

```{r}
# Find optimal node size
nodesizes = seq(from = 1, to = 75)
oos_error_by_nodesize = array(data = NA, dim = length(nodesizes))
for (m in 1 : length(nodesizes)) {
```
```

```
  tree_mod = YARFCART(X_train, y_train, nodesize = nodesizes[m],
                      calculate_oob_error = FALSE, verbose = FALSE)
  y_hat_select = predict(tree_mod, X_select)
  oos_error_by_nodesize[m] = sqrt(mean((y_select - y_hat_select)^2))
}

# Visualize OOS error by node size
ggplot(data.frame(nodesize = nodesizes, oos_error = oos_error_by_nodesize)) +
  aes(x = nodesize, y = oos_error) +
  geom_point()

optimal_nodesize = nodesizes[which.min(oos_error_by_nodesize)]
cat("Optimal node size:", optimal_nodesize)
```

We can now build a tree model using this node size:

```{r}
tree_mod = YARFCART(
  rbind(X_train, X_select),
  c(y_train, y_select),
  nodesize = optimal_nodesize,
  calculate_oob_error = FALSE,
  verbose = FALSE
)
```

To get a sense of performance, we can compute the in-sample R^2 and $RMSE$ metrics:

```{r}
y_hat_train_select = predict(tree_mod, rbind(X_train, X_select))

SST_train_select = sum((c(y_train, y_select) - mean(c(y_train, y_select)))^2)
inSSE_tree_mod = sum((c(y_train, y_select) - y_hat_train_select) ^2)

inRMSE_tree_mod = sqrt(inSSE_tree_mod / ((n_train + n_select) - 1))
inRsq_tree_mod = 1 - inSSE_tree_mod / SST_train_select

cat("The in-sample RMSE metric for the tree model is:", inRMSE_tree_mod, "\n")
cat("The in-sample R^2 metric for the tree model is:", inRsq_tree_mod, "\n")
```

To get a sense of performance, we can compute the oos R^2 and $RMSE$ metrics:

```{r}
y_hat_test = predict(tree_mod, X_test)

SST = sum((y_test - mean(y_test))^2)
SSE_tree_mod = sum((y_test - y_hat_test) ^2)
```

```
RMSE_tree_mod = sqrt(SSE_tree_mod / (n_test - 1))
Rsq_tree_mod = 1 - SSE_tree_mod / SST

cat("The RMSE metric for the tree model is:", RMSE_tree_mod, "\n")
cat("The R^2 metric for the tree model is:", Rsq_tree_mod, "\n")
```

The R^2 metric is not bad; it's positive and moderately close to 1. We could compute the metrics for the null model to get a sense for how good well the tree model did. The R^2 metric of the null model is 0 by definition, so comparing against its R^2 metric is not useful. But we can consider the RMSE:

```{r}
RMSE_0 = sqrt(SST / (n_test - 1))
cat("The RMSE metric for the null model is:", RMSE_0, "\n")
cat("Proportional change in RMSE compared to null model: ",
    (RMSE_tree_mod - RMSE_0) / RMSE_0, "\n")
```

It seems that there is about a 43% reduction in the RMSE from using our tree model in comparison to the null model
Next, let's illustrate the first few layers to get a sense for what the most important features might be:

```{r}
?illustrate_trees
illustrate_trees(tree_mod, max_depth = 5, font_size = 8, open_file = TRUE,
                 title = "tree_mod_report", file_format = "png")
illustrate_trees(tree_mod, max_depth = 6, font_size = 8, open_file = TRUE,
                 title = "tree_mod_top_10", file_format = "png")
```

At the top we have `approx_year_built` as the most important feature. Other important features are `sq_footage`, `parking_charges`, and `monthly_charges`.

```{r}
get_tree_num_nodes_leaves_max_depths(tree_mod)
```

Linear Modeling

Next we try a linear model.

```{r}
X_train_and_select = rbind(X_train, X_select)
y_train_and_select = c(y_train, y_select)
Xy_train_and_select = cbind(X_train_and_select, y_train_and_select)

linear_mod = lm(y_train_and_select ~ ., Xy_train_and_select)
data.frame(linear_mod$coefficients)
```

First we see how OLS does in-sample:

```{r}
cat("The in-sample RMSE metric for the linear model is:",
    summary(linear_mod)$sigma, "\n")
cat("The in-sample R^2 metric for the linear model is:",
    summary(linear_mod)$r.sq, "\n")
```

Let's also see how it does out-of-sample:

```{r}
y_hat_test = predict(linear_mod, X_test)

SSE_linear_mod = sum((y_test - y_hat_test) ^2)

RMSE_linear_mod = sqrt(SSE_linear_mod / (n_test - 1))
Rsq_linear_mod = 1 - SSE_linear_mod / SST

cat("The RMSE metric for the linear model is:", RMSE_linear_mod, "\n")
cat("The R^2 metric for the linear model is:", Rsq_linear_mod, "\n")
```

Random Forest Modeling

I will perform model selection with a grid of triplets to find the best triplet of the hyperparameter N_0 (threshold number of nodes), number of trees, and m_{try} (size of feature subsets).

```{r}
# Create grid of triplets for hyperparameters to test for
mtry_values = c(1, seq(from = 1, to = ncol(X), by = 2))
numtree_values = c(1, seq(from = 10, to = 60, by = 10))
nodesize_values = c(1, seq(from = 2, to = 30, by = 2))

optimal_mtry = 1
optimal_numtree = 1
optimal_nodesize = 1
optimal_oos_RMSE = Inf

total_iterations =
  length(mtry_values) *
  length(numtree_values) *
  length(nodesize_values)

current_iteration = 1

# Find the optimal hyperparameter triplet
for (i in 1 : length(mtry_values)) {
  for (j in 1 : length(numtree_values)) {
```

```
    for (k in 1 : length(nodesize_values)) {
      cat("Iteration", current_iteration, "of", total_iterations, "\n")
      current_iteration = current_iteration + 1

      rf_mod = YARF(X_train, y_train,
                    mtry = mtry_values[i],
                    num_trees = numtree_values[j],
                    nodesize = nodesize_values[k],
                    calculate_oob_error = FALSE, verbose = FALSE)

      y_hat_select = predict(rf_mod, X_select)
      current_oos = sqrt(mean((y_select - y_hat_select)^2))

      if (current_oos < optimal_oos_RMSE) {
        optimal_oos_RMSE = current_oos
        optimal_mtry = mtry_values[i]
        optimal_numtree = numtree_values[j]
        optimal_nodesize = nodesize_values[k]
      }
    }
  }
}

cat("Optimal oos_RMSE", optimal_oos_RMSE, "\n")
cat("Optimal mtry:", optimal_mtry, "\n")
cat("Optimal node size:", optimal_nodesize, "\n")
cat("Optimal number of trees:", optimal_numtree, "\n")
```

Using these "optimal" values, we can build a random forest model:

```{r}
rf_mod = YARF(rbind(X_train, X_select), c(y_train, y_select),
                    mtry = optimal_mtry,
                    num_trees = optimal_numtree,
                    nodesize = optimal_nodesize,
                    calculate_oob_error = FALSE, verbose = FALSE, seed = 42)
```

```{r}
y_hat_train_select = predict(rf_mod, rbind(X_train, X_select))
inSSE_rf_mod = sum((c(y_train, y_select) - y_hat_train_select) ^2)
SST_train_select = sum((c(y_train, y_select) - mean(c(y_train, y_select)))^2)

inRMSE_rf_mod = sqrt(inSSE_rf_mod / ((n_train + n_select) - 1))
inRsq_rf_mod = 1 - inSSE_rf_mod / SST_train_select

cat("The in-sample RMSE metric for the random forest model is:", inRMSE_rf_mod, "\n")
cat("The in-sample R^2 metric for the random forest model is:", inRsq_rf_mod, "\n")
```

Finally, we compute the associated out-of-sample metrics metrics:

```{r}
y_hat_test = predict(rf_mod, X_test)

SSE_rf_mod = sum((y_test - y_hat_test) ^2)

RMSE_rf_mod = sqrt(SSE_rf_mod / (n_test - 1))
Rsq_rf_mod = 1 - SSE_rf_mod / SST

cat("The RMSE metric for the random forest model is:", RMSE_rf_mod, "\n")
cat("The R^2 metric for the random forest model is:", Rsq_rf_mod, "\n")
```