New York City Property Sales - Harvard Data Science Project

Dr Daniel Mayenberger

May 2020

1 Executive Summary

There are thousands of real estate property transactions in New York City every year. Based on the transactions of one year from 01 Sep 2016 to 31 Aug 2017, we investigate algorithms that predict sale prices based on attributes of these properties.

The best-performing algorithm based on the average relative difference of prices with an average difference of 23% (-10% if excluding low prices, below $100k) is:

$$Y_{b,c} = \mu_{b,c}^{(p)} \times a + \varepsilon_{b,c} \quad ,$$

where $Y_{b,c}$ is the property sale prices in borough b for a building of class c, $\mu_{b,c}^{(p)}$ is the estimated square-foot price and a stands for the gross square footage.

2 Introduction

The dataset of New York City property sales is sourced from the Kaggle Website of NYC Property Sales, as downloaded on 25 May 2020. The goal is to predict the price of a property transaction based on the 20 other attributes of the property that are available from the data set. The performance of the model is measured by the mean of relative price differences, defined as

$$RMSE(m) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{\hat{v}_i(m)}{v_i} - 1 \right),$$

where N is the total number of observations, $\hat{v}_i(m)$ is the predicted sale price of property i under model m and v_i the actual sale price.

To do so, the data are examined and modelled in section 3 which is further broken down into:

- subsection 3.1 to elaborate on the techniques used, in particular those coded in the later modelling subsection 3.5.
- subsection 3.2 to provide an overview of the basic structure of the rating data.
- subsection 3.3 to perform data cleaning.
- subsection 3.4 to visualise the most important properties. These properties then inspire the modelling methods in the subseqeuent section.
- subsection 3.5 presents the models based on the most salient data properties and calibrates any free modelling parameters.

The results of all models are summarised in section 4 and the conclusions are drawn in section 5.

3 Methods and Analysis

3.1 Process and Techniques Used

3.1.1 Calculation of Means

Any means of rates should be calculated using the *harmonic mean* which is defined for values $x_i, i = 1, \ldots, N$ as

$$\mu_{harm} = \frac{1}{\frac{1}{N}\sum_{i=1}^{N}\frac{1}{x_i}}.$$

For example a price per square foot is a rate.

Also, any values whose distribution is log-normal rather than normal should be averaged using the *geometric mean* which is defined for positive values $Y_i, i = 1, \ldots, N$ as

$$\mu_{geom} = \left(\prod_{i=1}^{N} Y_i\right)^{\frac{1}{N}}$$

In R, these two functions are implemented as shown below:

```
#' Harmonic mean
#'
#' @param x Vector to calculate harmonic mean of
#' @return Harmonic mean of x
harm.mean <- function(x) 1/mean(1/x)

#' Geometric mean
#'
#' @param x Vector to calculate harmonic mean of
#' @return Harmonic mean of x
log10mean <- function(x) 10^mean(log10(x))
```

3.1.2 Local Estimated Scatterplot Smoothing (LOESS)

Unlike standard local regression that fits a line to the whole data set, the locally estimated scatterplot smoothing (LOESS) estimates a regression line in a local window that is progressively moved through the data. The parameter of LOESS is the width of this window.

Further details of the LOESS can be found on this Wikipedia website.

3.2 Data Structure and Loading

3.2.1 Data Download

The raw data is loaded from the Github location at this location with the code shown in the Appendix here. The only minimal transformation applied is the change of the column names to remove hyphens ("-"), replace white spaces with underscores ("_") and convert them to lower case. There are overall 84548 data points and 21 features as well as the sale price for each property.

3.2.2 Split into Training Set and Test Set

A first look at the raw data shows that the following columns can be removed:

- The variable `x1` is a counter.
- The `easement` variable only holds `NA` values.
- Apartment numbers are too granular as an attribute.
- Single address information is too granular as well.

The related code for these facts is:

```
# number of different entries in x1
length(unique(nycproperty_raw$x1))
```

```
## [1] 26736
```

```
# number of enries of `easement` that are not NA
sum(!is.na(nycproperty_raw$easement))
```

```
## [1] 0
```

```
# different apartment numbers
length(unique(nycproperty_raw$apartment_number))
```

```
## [1] 3989
```

```
# different addresses
length(unique(nycproperty_raw$address))
```

```
## [1] 67563
```

Further data will not be removed as that would constitute modelling hindsight, wherease discarding of the above columns does not require any modelling operations. With that, the data are split into a training and a test set with a *requested* ratio of 90/10:

```
nycproperty_raw <- nycproperty_raw %>%
    select(-x1, -easement, -apartment_number, -address)

# Test set will be 10% of overall data
# if using R 3.5 or earlier, use `set.seed(1)` instead
set.seed(1, sample.kind="Rounding")
test_index <- createDataPartition(y = nycproperty_raw$sale_price,
                                  times = 1, p = 0.1, list = FALSE)
train_nycp <- nycproperty_raw[-test_index,]
test_nycp <- nycproperty_raw[test_index,]

# save to file for report
save(train_nycp, test_nycp, file = "data/nycp_split_raw.Rdata")
```

For the purposes of this report, the data are loaded from the data file `nycp_split_raw.Rdata`:

```
load(file = "data/nycp_split_raw.Rdata")
```

After this first split, there are 68069 rows in the training set and 16479 rows in the test set. The resulting ratio of 4.13 is different from the desired ratio of 9:1, since the `createDataPartition` function attempts to have the values of all features equally represented in the training and the test set. As a result of this attempt, the actual split ratio is different from 9:1 and, as will become apparent, the values for some features that are in the test set are not occurring the training set.

So before the training and test data are used for further analysis, it is ensured that each of the features of the test set are contained in the training set as well. If that is not the case, the following function is applied to transfer the respective columns from the test set to the training set. This is the same type of procedure that was applied to the movielens data for the first project.

```
#' Function to ensure that all values of a certain column that appear
#' in the test set are also in the training set by transferring missing
#' values from the (temporary) test set to the training set
#'
#' @param train Training set
```

```r
#' @param temp_test Temporary test set
#' @return List of data frames named `train` and `test` containing the
#'         training set augmented by values previous in (temporary) test set
#'         but not in training set.
augment_train_by_test <- function(train, temp_test, column) {
    # checking condition
    nrows_before <- nrow(train) + nrow(temp_test)

    # Only transfer values from temporary set into test that are
    # also in the training set
    test <- temp_test %>% semi_join(train, by = column)

    # add rows removed from temporary test set into training set
    removed <- anti_join(temp_test, test, by = column)
    train <- rbind(train, removed)

    # check condition to ensure data integrity
    assert("Total number of data rows remains the same",
           nrows_before == nrow(train) + nrow(test))

    # return as list
    list(train = train, test = test)
    }
```

The features for which rows were transferred in this way are:

- Neighbourhood - variable `neighborhood`.
- Building class at present - variable `building_class_at_present`.
- ZIP code - variable `zip_code`.
- Number of residential units - variable `residential_units`.
- Number of commercial units - variable `commercial_units`.
- Number of total units - variable `total_units`.
- Year built - variable `year_built`, after transformation into a decade, removed erroneous entry with value of 1,110 from the test set, see also subsection 3.3.
- Building class at time of sale - variable `building_class_at_time_of_sale`, in combination with borough (`borough`)

To transfer the rows multiple steps are required (as functions in R by default do not allow to change parameters, aka "call by reference"), for example for the `neighborhood` feature:

```r
# invoke function to transfer any rows required from test set to
# training set with respect to the feature `neighborhood`
df_temp <- augment_train_by_test(train_nycp, test_nycp,
                                 column = "neighborhood")
# copy new data frames into training and test set
train_nycp <- df_temp$train
test_nycp <- df_temp$test
# double-check condition, should return zero
test_nycp %>% anti_join(train_nycp, by = "neighborhood") %>% nrow()
# discard temporary set
rm(df_temp)
```

The full code is in the accompanying R script `nycproperty.R`.

For these features no row transfer from training to test was necessary as all values assumed in the test set were contained in the training set, too:

- Building class category - variable `building_class_category`.
- Tax class at present - variable `tax_class_at_present`.
- Tax class at time of sale - variable `tax_class_at_time_of_sale`.

Finally, for the following features no transfer was conducted, although there are features in the test set that are not in the training set:

- Block - variable `block` - there are too many different values, over 9,000, to be useful as predictor.
- Lot - variable `lot` - also too many different values to make a useful predictor, about 2,000.
- Land square feet - `land_square_feet` is a continuous feature, so checking equality of values is not meaningful.
- Gross square feet - `gross_square_feet` is also a continuous feature.

Once all these data transfers are completed, the training and test set are stored in the data file `nycp_split_clean.Rdata` for later retrieval and loaded for this report:

```
load(file = "data/nycp_split_clean.Rdata")
```

Thereafter, there are 68150 rows in the training set and 16397 rows in the test set.

3.2.3 Basic Data Structure

Firstly, basic economics dictate that

- property prices be denominated in *prices per square footage* rather that in absolute dollar values.
- Ideally, this area should be *net* square footage, i.e. only the usable area of the building without area occupied by staircases, lift shafts, boiler rooms or other utilities.

However, net square footage is not given in the data set and, as we will find out in this section further below, the difference of gross square footage and land square footage is not usable as an approximation, although by definition the latter number should always be smaller than the former number. Moreover, even gross square footage is only provided in terms of a usable value (at least 10 ft^2) for about half of the data. Nonetheless, unless explicitly stated otherwise, any prices henceforth mentioned are to be understood as *prices per gross square footage*.

In the modelling subsection 3.5, methods will be defined to approximate the gross square footage where values are unavailable or not meaningful.

Data for boroughs are only provided as number. The different download sections on the The NYC Department of Finance Website reveal that these numbers signify:

1. Manhattan
2. Bronx
3. Brooklyn
4. Queens
5. Staten Island

From the overall 17 data attributes and one result column, the following basic data properties are elicited for each column.

No.	Variable	Basic Properties
1	borough	There are five boroughs overall, given by their numbers as above.
2	neighborhood	The overall number of neighbourhoods is 253, represented by character strings.
3	building_class_category	There are 46 different building class categories, of which 1-3 family dwellings, coops and condos account for over 90 % of transactions. The data is given as character strings starting with 2-digit numbers.
4	tax_class_at_present	There are overall nine tax classes, of which class 1, 2, and 4 cover over 90 % of sales. Provided as a string of 1-2 characters
5	block	There are about 10,000 different blocks, too granular for prediction.
6	lot	There are about 2,000 different lots, again too granular for modelling
7	building_class_at_present	There are around 150 different building classes, given as 2-character codes.
8	zip_code	ZIP codes are 5-digit numbers, if which there are about 180 different ones. There are invalid ZIP codes (zeroes), but these account for less than 1%.
9	residential_units	The number of residential units is an integer number.
10	commercial_units	The number of commercial units is an integer number.
11	total_units	The number of total units is an integer number. There are a few hundred entries for which the number of residential and commercial units do not add to, and is strictly lower than, the total number of units, so the total number of units will be used as more reliable.
12	land_square_feet	The land square feet is provided as character string with "-" or "0" denoting zeroes.
13	gross_square_feet	The land square feet is provided as character string with "-" or "0" denoting zeroes.
14	year_built	Provided as an integer number.
15	tax_class_at_time_of_sale	This is of the same format as the tax class at present (position #4).
16	building_class_at_time_of_sale	Given in the same format as the building class at present (position #7).
17	sale_date	Already given as date.
18	sale_price	Given as a character string with "-" or "0" denoting zeroes.

The details of the related code can be found in the accompanying R script `nycproperty.R`.

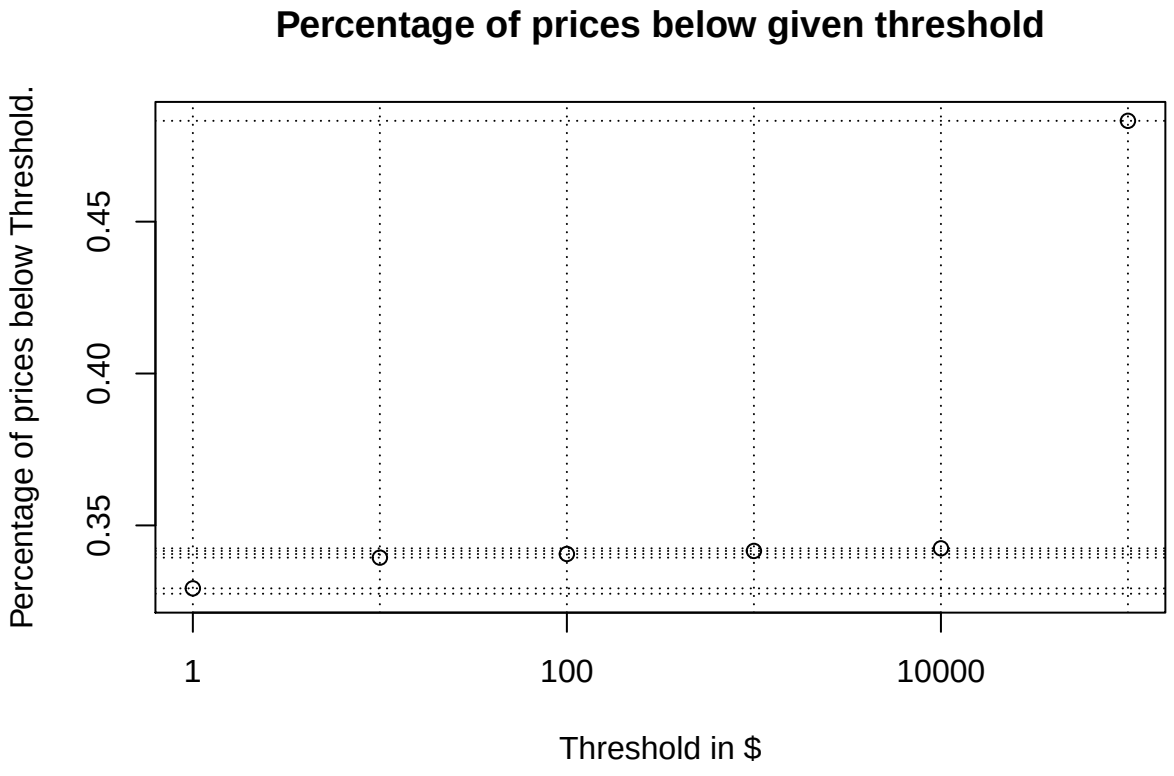
For the net square footage, the following simple evaluation shows that even for the half of the data (~24k records) for which the net square footage can be calculated as the simple difference of gross square footage and land square footage, only one third of this half (~7k records) results in a positive net square footage.

```
train_nycp %>%
    mutate(gross_square_feet = str_replace(gross_square_feet,"-","0"),
           land_square_feet = str_replace(land_square_feet,"-","0")) %>%
    mutate(gross_square_feet = parse_number(gross_square_feet),
           land_square_feet = parse_number(land_square_feet)) %>%
    mutate(net_square_feet = gross_square_feet -
               land_square_feet) %>%
    mutate(net_sqf_avail = (net_square_feet > 0),
           gross_sqf_avail = (gross_square_feet >= 10),
           land_sqf_avail = (land_square_feet >= 10),
```

```
          sale_price_avail = (sale_price > 0)) %>%
    group_by(sale_price_avail, net_sqf_avail,
             gross_sqf_avail, land_sqf_avail) %>%
    summarise(n = n())
```

```
## # A tibble: 10 x 5
## # Groups:   sale_price_avail, net_sqf_avail, gross_sqf_avail [6]
##    sale_price_avail net_sqf_avail gross_sqf_avail land_sqf_avail     n
##    <lgl>            <lgl>         <lgl>           <lgl>          <int>
##  1 FALSE            FALSE         FALSE           FALSE           6473
##  2 FALSE            FALSE         FALSE           TRUE            1295
##  3 FALSE            FALSE         TRUE            TRUE            8718
##  4 FALSE            TRUE          TRUE            FALSE              7
##  5 FALSE            TRUE          TRUE            TRUE            5827
##  6 TRUE             FALSE         FALSE           FALSE          21462
##  7 TRUE             FALSE         FALSE           TRUE             772
##  8 TRUE             FALSE         TRUE            TRUE           16685
##  9 TRUE             TRUE          TRUE            FALSE              2
## 10 TRUE             TRUE          TRUE            TRUE            6909
```

For the price, it should be noted that many prices are very low. The Kaggle website already highlights the fact and notes that these are from deed transfers. The chart below illustrates that these records account for a very small percentage, prices of $10,000 and below making up considerably less than 0.5%.



This will be addressed in the following Data Cleaning section.

3.3 Data Cleaning

Often data given as character strings will be transformed into a factor to facilitate processing by algorithms. For efficiency and to ensure that the factor

- is already ordered by the outcome variable `sale_price`, and
- levels are the same for training and test set,

the following function is defined:

```
#' Auxiliary function to convert data column in to factor consistently
#' for training and test set
#'
#' @param train Training set
#' @param test Test set
#' @param column_factor Column to convert into factor
#' @param  column_order = NA Optional column used to reorder the factor
#' column_factor
#' @param order_fct = NA Optional column specifying the function applied
#' for reordering the column_factor
#' @return List of data frames named `train` and `test` containing the
#'         training and test sets with factors having the same levels
#'         and order
apply_factor_trainandtest <- function(train, test, column_factor,
                                      column_order = NA, order_fct = NA) {
    # First for training data:
    # 1) convert to factor in-place
    res_train <- train %>%
        mutate(!!quo_name(column_factor) := factor(!!sym(column_factor)))

    # 2) Reorder factor if requested
    if(!is.na(column_order))
        res_train <- res_train %>%
            mutate(!!quo_name(column_factor) :=
                       reorder(!!sym(column_factor),
                               !!sym(column_order), order_fct))

    # retrive levels from training data
    f_levels <- res_train %>% pull(!!sym(column_factor)) %>% levels()

    # Then for test data
    # 1) Convert to factor; NB: Take levels from training set
    res_test <- test %>%
        mutate(!!quo_name(column_factor) :=
                   factor(!!sym(column_factor), levels = f_levels))


    return(list(train = res_train, test = res_test))
    }
```

The data cleaning is summarised in the table below:

No.	Variable	Data Cleaning Activities
1	`borough`	Converted to factor.
2	`neighborhood`	Converted to factor, ordered by sales price.
3	`building_class_category`	Converted to factor, ordered by sales price.
4	`tax_class_at_present`	Converted to factor, ordered by sales price.
5	`block`	No cleaning required.
6	`lot`	No cleaning required.
7	`building_class_at_present`	Converted to factor, ordered by sales price.
8	`zip_code`	Converted to factor, ordered by sales price.
9	`residential_units`	No cleaning required.
10	`commercial_units`	No cleaning required.
11	`total_units`	No cleaning required.
12	`land_square_feet`	Replaced `"-"` with `"0"`, apply `parse_number` to convert to number.
13	`gross_square_feet`	Replaced `"-"` with `"0"`, apply `parse_number` to convert to number.
14	`year_built`	Rounded down to decade and renamed to `decade_built`. Removed single data row from test set for which the decade is 1,110, clearly a data error.
15	`tax_class_at_time_of_sale`	Converted to factor, ordered by sales price.
16	`building_class_at_time_of_sale`	Converted to factor, ordered by sales price. Also transferred combinations of values of borough and building class at time of sale from the test set to the training set.
17	`sale_date`	Extracted month and stored in new column `sale_month`.
18	`sale_price`	Replaced `"-"` with `"0"`, apply `parse_number` to convert to number. Removed any values below 10,000.

Full details can be found in the accompanying R script `nycproperty.R`.

After all cleaning activities, the training and test set are stored in the file `nycp_split_trans.Rdata` and loaded for this report:

```
load(file = "data/nycp_split_trans.Rdata")
```

There are now 44693 rows in the training set and 13706 rows in the test set.

3.4 Data Exploration and Visualisation

We define thresholds to define when a square footage is *available*, namely, when it is 10 ft^2 or larger, as any value below is not meaningful.

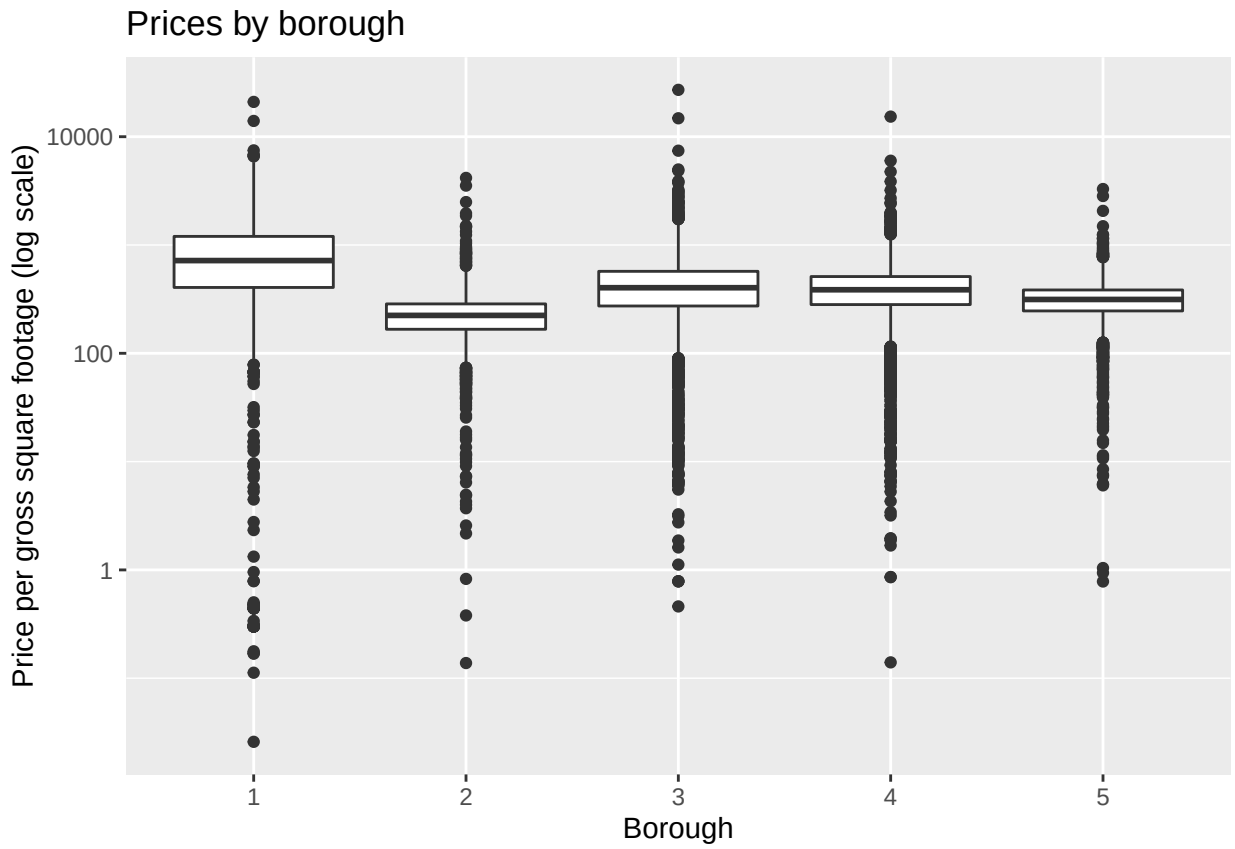
```
# Define constants for square footage - below 10 sqft area is not meaningful
FOOTAGE_MIN <- 10
```

3.4.1 Borough

Plotting the distribution of square-foot prices by borough shows:

```
train_nycp %>%
    filter(gross_square_feet >= FOOTAGE_MIN) %>%
    mutate(price_per_gsf = sale_price / gross_square_feet) %>%
    ggplot(aes(x = borough, y = price_per_gsf)) +
    scale_y_log10() +
    geom_boxplot() +
    labs(title = "Prices by borough",
         x = "Borough",
```

```
        y = "Price per gross square footage (log scale)")
```

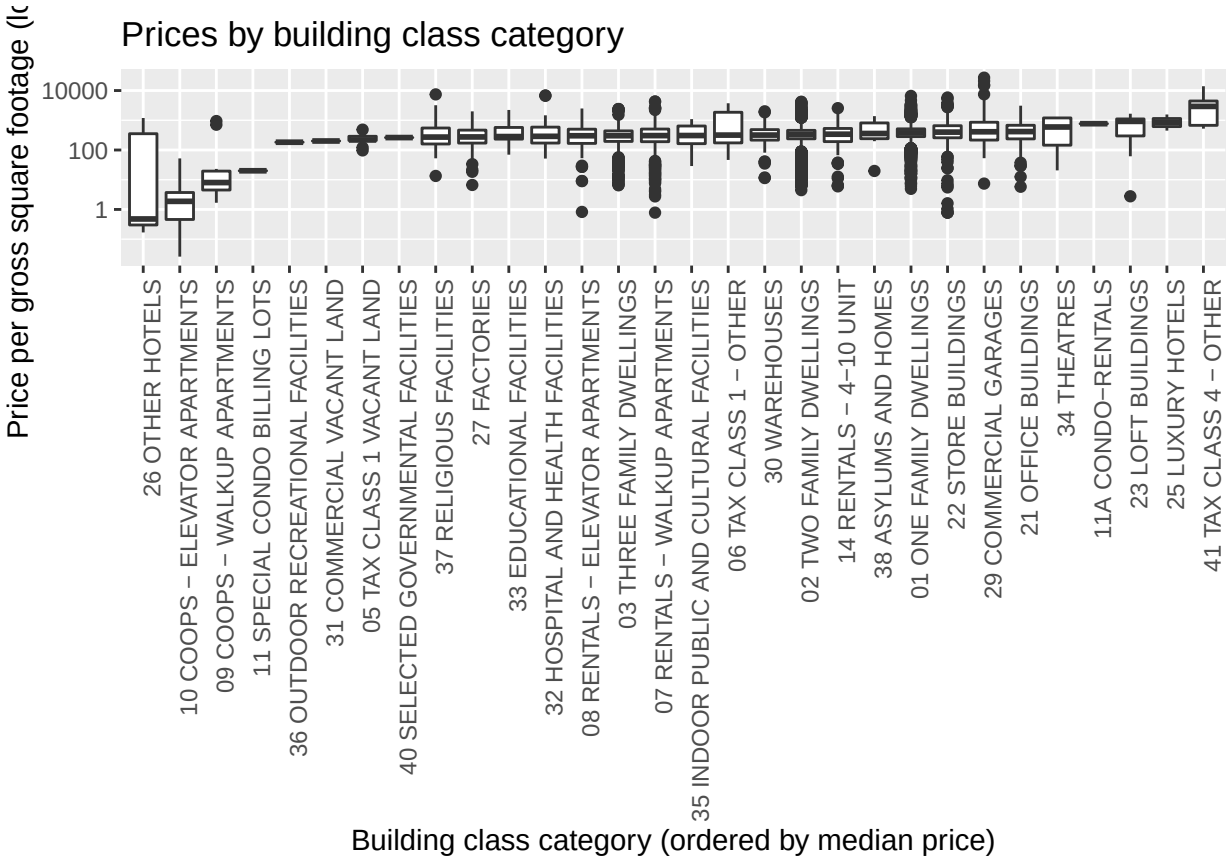


So the prices are lower in the Bronx than in the other four boroughs.

3.4.2 Building Class Category

Next, we examine the prices by building class category:

```
train_nycp %>%
    filter(gross_square_feet >= FOOTAGE_MIN) %>%
    mutate(price_per_gsf = sale_price / gross_square_feet) %>%
    mutate(building_class_category =
               reorder(building_class_category, price_per_gsf, median)) %>%
    ggplot(aes(x = building_class_category,
               y = price_per_gsf)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
    scale_y_log10() +
    geom_boxplot() +
    labs(title = "Prices by building class category",
         x = "Building class category (ordered by median price)",
         y = "Price per gross square footage (log scale)")
```

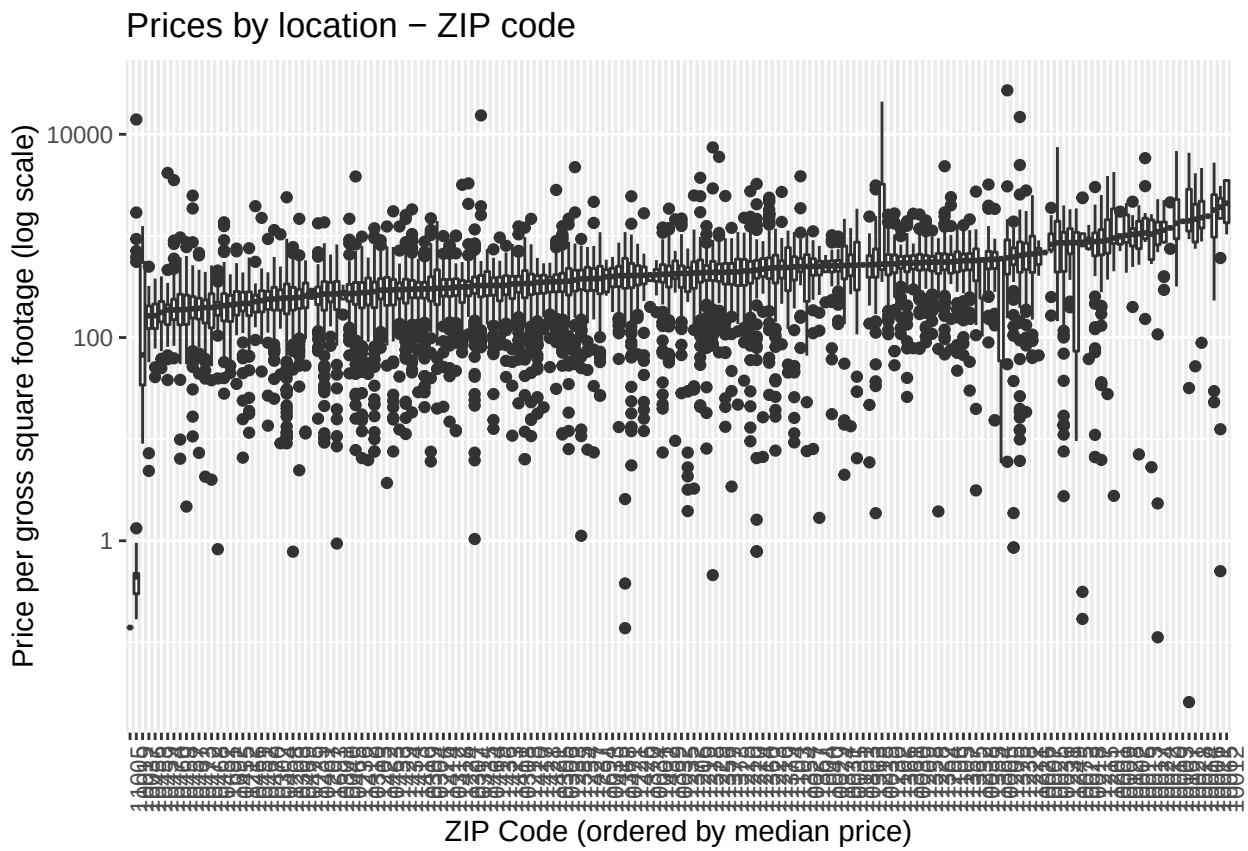


So categories 10 (coops elevator apartments) and 9 (coops walkup apartments) are distinguished from the other classes by lower prices, but all the other classes have large overlaps

3.4.3 ZIP Code

Performing the same analysis for the ZIP code:

```
train_nycp %>%
    filter(gross_square_feet >= FOOTAGE_MIN) %>%
    filter(zip_code != 0) %>%
    mutate(price_per_gsf = sale_price / gross_square_feet) %>%
    mutate(zip_code = reorder(zip_code, price_per_gsf, median)) %>%
    ggplot(aes(x = zip_code, y = price_per_gsf)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
    scale_y_log10() +
    geom_boxplot() +
    labs(title = "Prices by location - ZIP code",
         x = "ZIP Code (ordered by median price)",
         y = "Price per gross square footage (log scale)")
```

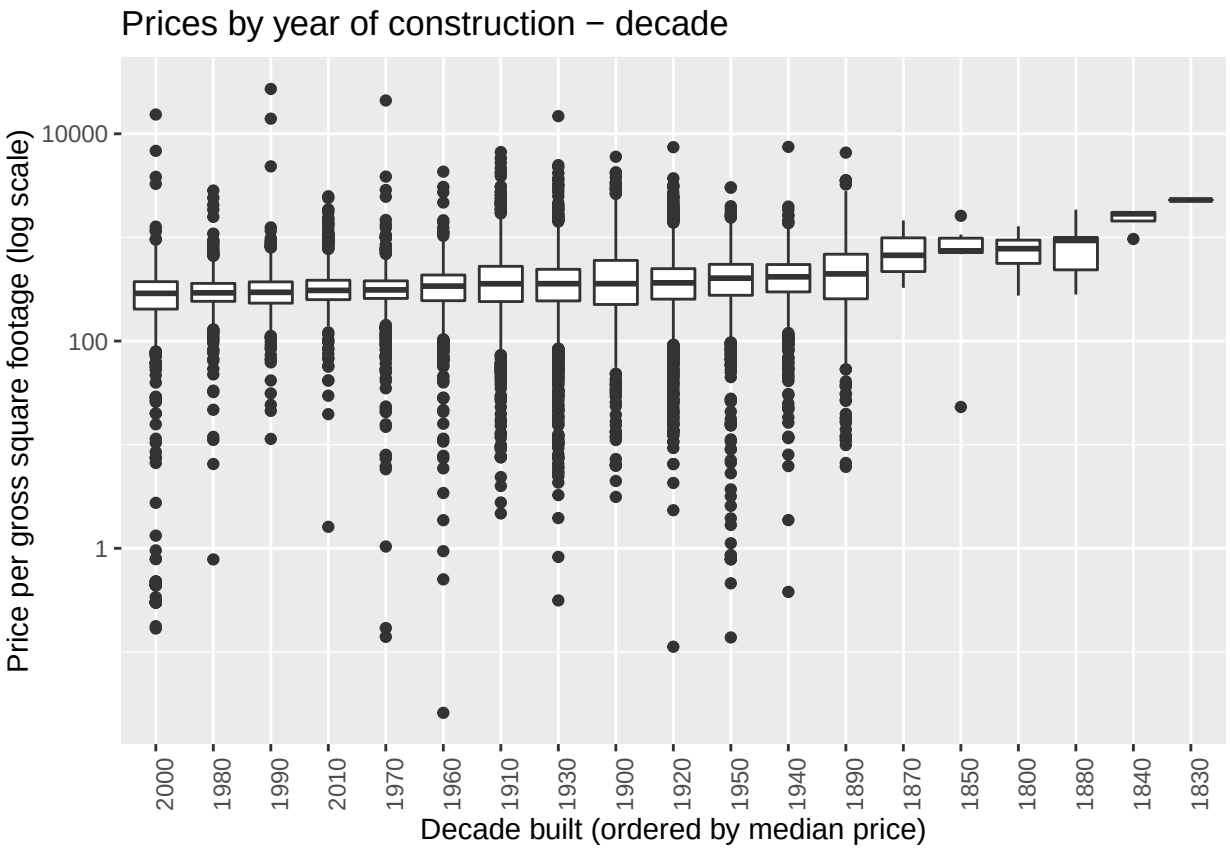


It becomes apparent that there is some effect of the location on the price per square foot. This is to be expected, as some neighbourhoods have better amenities and offer higher standards of housing on average.

3.4.4 Year Built

Plotting the distribution of prices against the decade in which the property was built:

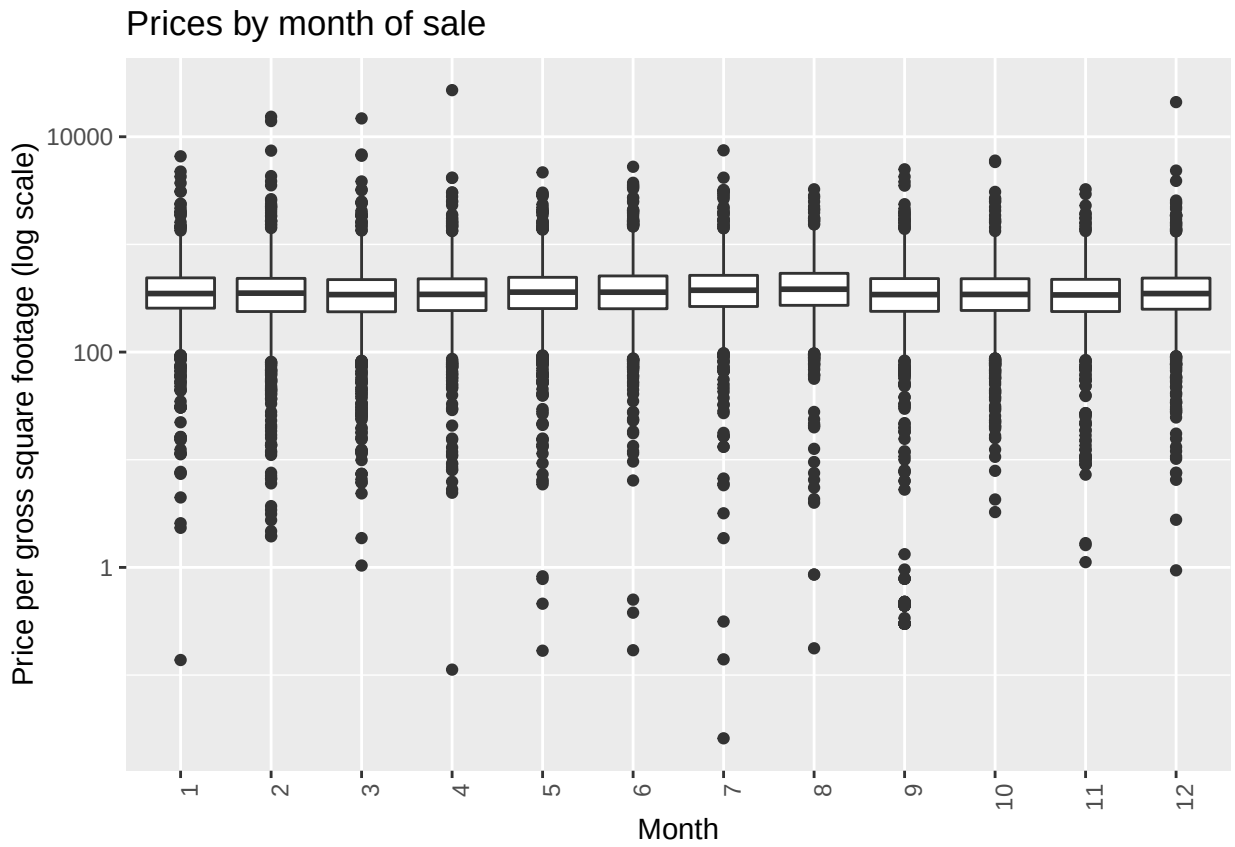
```
train_nycp %>%
    filter(gross_square_feet >= FOOTAGE_MIN) %>%
    filter(decade_built > 0) %>%
    mutate(price_per_gsf = sale_price / gross_square_feet) %>%
    mutate(decade_built = reorder(decade_built, price_per_gsf, median)) %>%
    ggplot(aes(x = decade_built, y = price_per_gsf)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
    scale_y_log10() +
    geom_boxplot() +
    labs(title = "Prices by year of construction - decade",
         x = "Decade built (ordered by median price)",
         y = "Price per gross square footage (log scale)")
```



3.4.5 Sale Date

One would expect a certain seasonality of the price, given that in particular for residential properties people tend to relocate in Spring and Summer and avoid in particular the Winter months.

```
train_nycp %>%
    filter(gross_square_feet >= FOOTAGE_MIN) %>%
    mutate(price_per_gsf = sale_price / gross_square_feet) %>%
    ggplot(aes(x = factor(sale_month), y = price_per_gsf)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
    scale_y_log10() +
    geom_boxplot() +
    labs(title = "Prices by month of sale",
         x = "Month",
         y = "Price per gross square footage (log scale)")
```

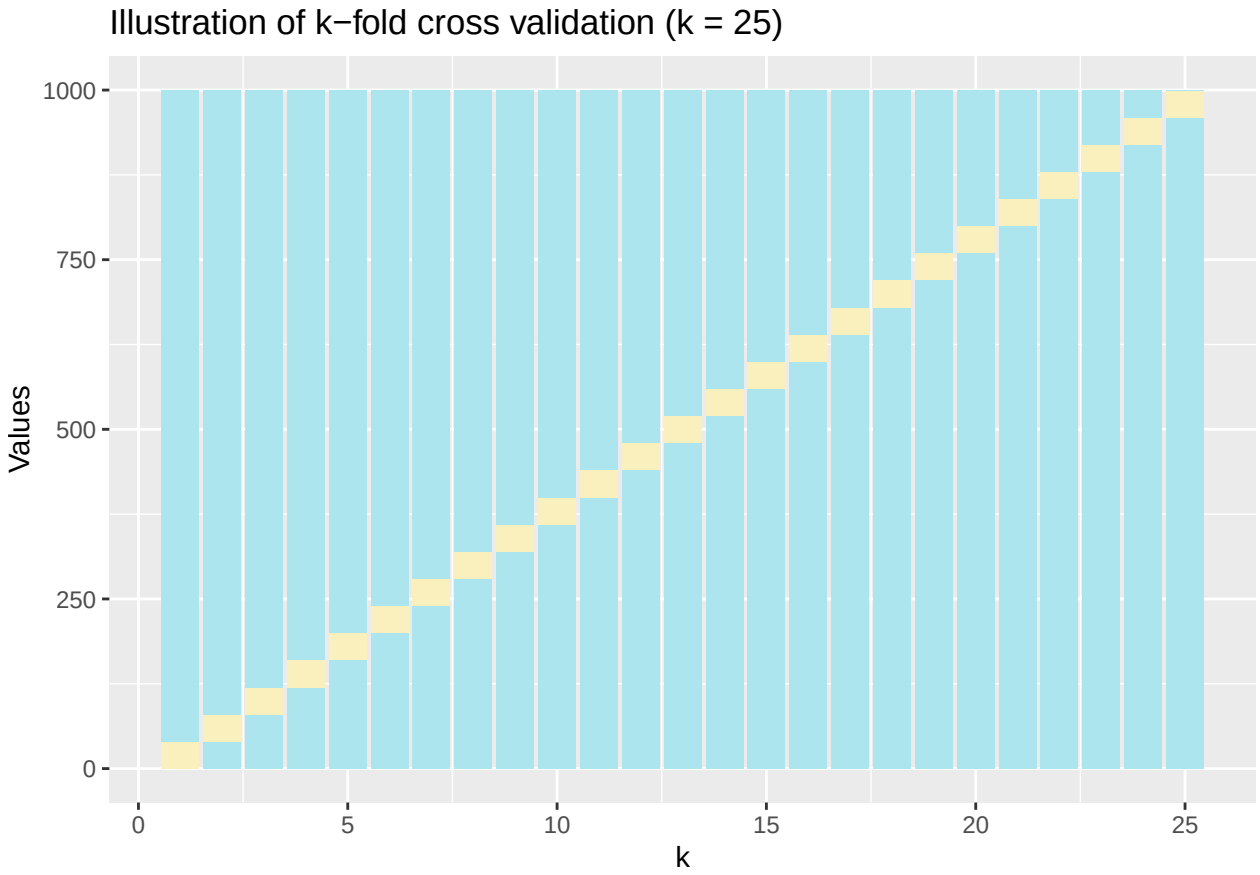


However, there is no seasonality whatsoever in this year. The sale prices per square foot have virtually the same distribution for every month.

3.5 Modelling Approach

The training of model parameters is generally done using k-fold cross-validation. To do this, the `train_nycp` data is split into a training and a validation set k times.

Graphically, k-fold validation can be illustrated by k subsequent splits of the overall `train_nycp` set into training sets (shown in blue) and validation sets (shown in yellow).



For example, the different averaging methods for prices-per-square-foot are applied to the training data and the RMSE that results from that particular choice of averaging method is evaluated on the validation data. This is done $k = 25$ times to produce an estimate of the RMSE for the different averaging methods.

To do that, a list of 25 index vectors is created using the function `createFolds`:

```
# Generate list of training and test sets for k-fold cross-validation
# Use function createFolds with k = 25
set.seed(342, sample.kind = "Rounding")
index_list <- createFolds(train_nycp$sale_price, k = 25)
```

The **performance** of each algorithm is evaluated using the RMSE which is implemented in the function `RMSE_price`:

```
RMSE_price <- function(pred_price, actual_price)
    mean(pred_price/actual_price - 1)
```

3.5.1 Constant Values

As baseline methods, the simplest model assumes that the price is a constant:

$$Y_{b,c} = \mu^{(Y)} + \varepsilon_{b,c} \quad ,$$

where $Y_{b,c}$ is the sale price of a building in borough b of building class at time of sale c, $\mu^{(Y)}$ is the average price across all properties and $varepsilon_{b,c}$ is the error term.

The parameter μ_Y may be estimated in the following way:

- As arithmetic mean: $\hat{\mu}^{(Y)} = \frac{1}{N} \sum_{i=1}^{N} Y_i$

- As geometric mean: $\hat{\mu}^{(Y)} = \left(\prod_{i=1}^{N} Y_i\right)^{\frac{1}{N}}$
- As median.

We note that the geometric mean can equivalently be calculated as

$$\left(\prod_{i=1}^{N} Y_i\right)^{\frac{1}{N}} = 10^{\frac{1}{N}\sum_{i=1}^{N} log_{10}(Y_i)}.$$

The related models for arithmetic average and median price are simply implemented:

```
# model to predict arithmetic average
predict_const_avg <- function(newdata) mean(train_nycp$sale_price)

# model to predict geometric average
predict_const_geom <- function(newdata) log10mean(train_nycp$sale_price)

# model to predict median
predict_const_med <- function(newdata) median(train_nycp$sale_price)
```

3.5.2 Constant square-foot prices

Still a baseline model, but somewhat instructive in terms of the different ways to calculate averages over rates, is the model that assumes that the price per square foot is a constant:

$$Y_{b,c} = \mu^{(p)} \times a + \varepsilon_{b,c} \quad ,$$

where $\mu^{(p)}$ is the average *price per gross square footage* and a the gross square footage of the **individual** property.

Again, the implementation is straightforward. The only addition that must be considered is the fallback solution for records which do not have any gross square footage available. For the first model, the fallback price is the arithmetic average of prices and for the second one, the geometric average.

```
# Average of all prices per square foot - arithmetic
predict_sqf_const_avg <- function(newdata) {
    # fallback price: average price
    mu = mean(train_nycp$sale_price)

    # calculate average price per square foot
    price_per_sqf <- train_nycp %>%
        filter(gross_square_feet >= FOOTAGE_MIN) %>%
        summarise(price_per_sqf = sum(sale_price)/
                      sum(gross_square_feet)) %>%
        .$price_per_sqf

    # calculate sqf x price/sqf if sqf available, fall back to
    # mean price otherwise
    prices_hat <- newdata %>%
        mutate(price_hat = ifelse(gross_square_feet >= FOOTAGE_MIN,
                                  gross_square_feet * price_per_sqf,
                                  mu)) %>% .$price_hat

    return(prices_hat)
    }
```

```r
# Average of all prices per square foot - harmonic average
# for square-foot price and geometric average of abs prices as fallback
predict_sqf_const_harm <- function(newdata) {
    # fallback price: gemetric mean price
    mu = log10mean(train_nycp$sale_price)

    # calculate harmonic average price per square foot
    price_per_sqf <- train_nycp %>%
        filter(gross_square_feet >= FOOTAGE_MIN) %>%
        summarise(price_per_sqf = harm.mean(sale_price/gross_square_feet)) %>%
        .$price_per_sqf

    # calculate sqf x price/sqf if sqf available, fall back to
    # mean price otherwise
    prices_hat <- newdata %>%
        mutate(price_hat = ifelse(gross_square_feet >= FOOTAGE_MIN,
                                  gross_square_feet * price_per_sqf,
                                  mu)) %>% .$price_hat

    return(prices_hat)
    }
```

3.5.3 Square-foot prices by borough and building class

From subsubsection 3.4.1 and **??** it is known that there is some dependency of the price on borough and building class at time of sale. So the model is

$$Y_{b,c} = \mu_{b,c}^{(p)} \times a + \varepsilon_{b,c} \quad ,$$

where $\mu_{b,c}^{(p)}$ is the average price for borough b and building class c.

As we have learnt in subsection 3.3, the gross square footage is only available for half of all property transactions, so we employ a cascading model:

- Is gross square footage a available?
  1. Yes → Calculate $Y_{b,c} = \hat{\mu}_{b,c}^{(p)} \times a$, where $\hat{\mu}_{b,c}^{(p)}$ is the estimated average price for borough b and building class c. and a the property area as gross square footage.
  2. No →
     - Is the number of total units u available?
       1. Yes → Approximate the gross square footage a of the property as $\hat{a} = f(u)$ with a smooth function f. Then calculate $Y_{b,c} = \hat{\mu}_{b,c}^{(p)} \times \hat{a}$.
       2. No → Calculate $Y_{b,c} = \hat{\mu}_{b,c}^{(Y)}$, where $\hat{\mu}_{b,c}^{(Y)}$ is the average absolute price of a property in borough b of building class c.

Within this model, there is one more cascade: The average price $\hat{\mu}_{b,c}^{(p)}$ is not available for all combinations of b and c because the gross square footage is not available for **any** property for certain combinations of b and c. So the following nested cascade is implemented:

- Is gross square footage a available for any property in borough b of building class c?
  1. Yes → Calculate $\hat{\mu}_{b,c}^{(p)}$ as average price $\hat{\mu}_{b,c}^{(p)} = m\left(y_{b,c,i}, a_i\right)$ using all sale prices $y_{b,c,i}$ and areass a_i of all properties with an appropriate averaging function m.
  2. No → * Is $\hat{\mu}_{b^*,c}^{(p)}$ available for the same class c in any other borough b^*?
     1. Yes → Approximate $\hat{\mu}_{b,c}^{(p)}$ as average of all such $\hat{\mu}_{b^*,c}^{(p)}$ over all other boroughs b^*, weighted by the number of prices using an averaging function m.
     2. No →

– Is $\hat{\mu}^{(p)}_{b,c^*}$ available for any other building class $c^*$of the same general class, e.g. $c^* = C1$ for $c = C8$.
  1. Yes → Approximate $\hat{\mu}^{(p)}_{b,c}$ as average of all such $\hat{\mu}^{(p)}_{b,c^*}$ over all building classes c^* with the same general class, weighted by the number of prices using an averaging function m.

These cascades are implemented subsequently from paragraph 3.5.3.1 to paragraph 3.5.3.5.

3.5.3.1 Step 1: Square footage directly available

The only decision that must be made is on the type of average to use to calculate $\hat{\mu}^{(p)}_{b,c} = m\left(y_{b,c,i}, a_i\right)$ which comes down to the choice of the function m. The following choices are explored. For brevity of notation the indices b and c are omitted, replaced by one single index i.

1. Arithmetic overall mean: $\hat{\mu} = \frac{\frac{1}{N}\sum_i y_i}{\frac{1}{N}\sum_i a_i}$.
2. Geometric overall mean $\hat{\mu} = \frac{\left(\prod_i y_i\right)^{\frac{1}{N}}}{\left(\prod_i a_i\right)^{\frac{1}{N}}}$.
3. Median of prices $\frac{y_i}{a_i}$.
4. Harmonic mean of prices $\hat{\mu} = \frac{1}{\frac{1}{N}\sum_i \frac{a_i}{y_i}}$.

To make this decision, we employ k-cross validation as described at the beginning of subsection 3.5:

```
rmse_v_L1 <- map_df(1:length(index_list), function (listIdx) {
    # split data into training and test set
    train_set <- train_nycp[-index_list[[listIdx]], ]
    test_set <- train_nycp[index_list[[listIdx]], ]

    # compile table of average sqf prices by borough and building class
    ppgsf_L1_tbl <- train_set %>%
        select(borough, building_class_at_time_of_sale,
           gross_square_feet, sale_price) %>%
        filter(gross_square_feet >= FOOTAGE_MIN) %>%
        group_by(borough, building_class_at_time_of_sale) %>%
        summarise(nprices = n(),
                  price_per_gsf = sum(sale_price) / sum(gross_square_feet),
                  price_per_gsf_w = log10mean(sale_price) /
                      log10mean(gross_square_feet),
                  price_per_gsf_med = median(sale_price/gross_square_feet),
                  price_per_gsf_harm = harm.mean(sale_price/gross_square_feet))

    # predict prices on test set using available (square footage) x (sqf price)
    predictions <- test_set %>%
        filter(gross_square_feet >= FOOTAGE_MIN) %>%
        inner_join(ppgsf_L1_tbl,
                   by = c("borough", "building_class_at_time_of_sale")) %>%
        mutate(sale_price_hat = gross_square_feet * price_per_gsf,
               sale_price_hat_w = gross_square_feet * price_per_gsf_w,
               sale_price_hat_med = gross_square_feet * price_per_gsf_med,
               sale_price_hat_harm = gross_square_feet * price_per_gsf_harm) %>%
        mutate(perdiff = (sale_price_hat/sale_price - 1),
               perdiff_w = (sale_price_hat_w/sale_price - 1),
               perdiff_med = (sale_price_hat_med/sale_price - 1),
               perdiff_harm = (sale_price_hat_harm/sale_price - 1))
    predictions %>%
        summarise(RMSE_avg = mean(perdiff),
```

```
                  RMSE_avg_w = mean(perdiff_w),
                  RMSE_avg_med = mean(perdiff_med),
                  RMSE_avg_harm = mean(perdiff_harm))
    })

# summarise error for different average methods
rmse_v_L1 %>% summarise_all(mean)
```

```
## # A tibble: 1 x 4
##   RMSE_avg RMSE_avg_w RMSE_avg_med RMSE_avg_harm
##      <dbl>      <dbl>        <dbl>         <dbl>
## 1    0.652      0.393        0.486        0.0612
```

The harmonic mean clearly produces the lowest average percentage difference, so it is implemented in this first table:

```
price_gsf_L1_tbl <- train_nycp %>%
    select(borough, building_class_at_time_of_sale,
           gross_square_feet, sale_price) %>%
    filter(gross_square_feet >= FOOTAGE_MIN) %>%
    group_by(borough, building_class_at_time_of_sale) %>%
    summarise(nprices = n(),
              price_per_gsf_harm = harm.mean(sale_price/gross_square_feet))

# RMSE vector no longer needed
rm(rmse_v_L1)
```

3.5.3.2 Step 2: Square footage approximated by number of units

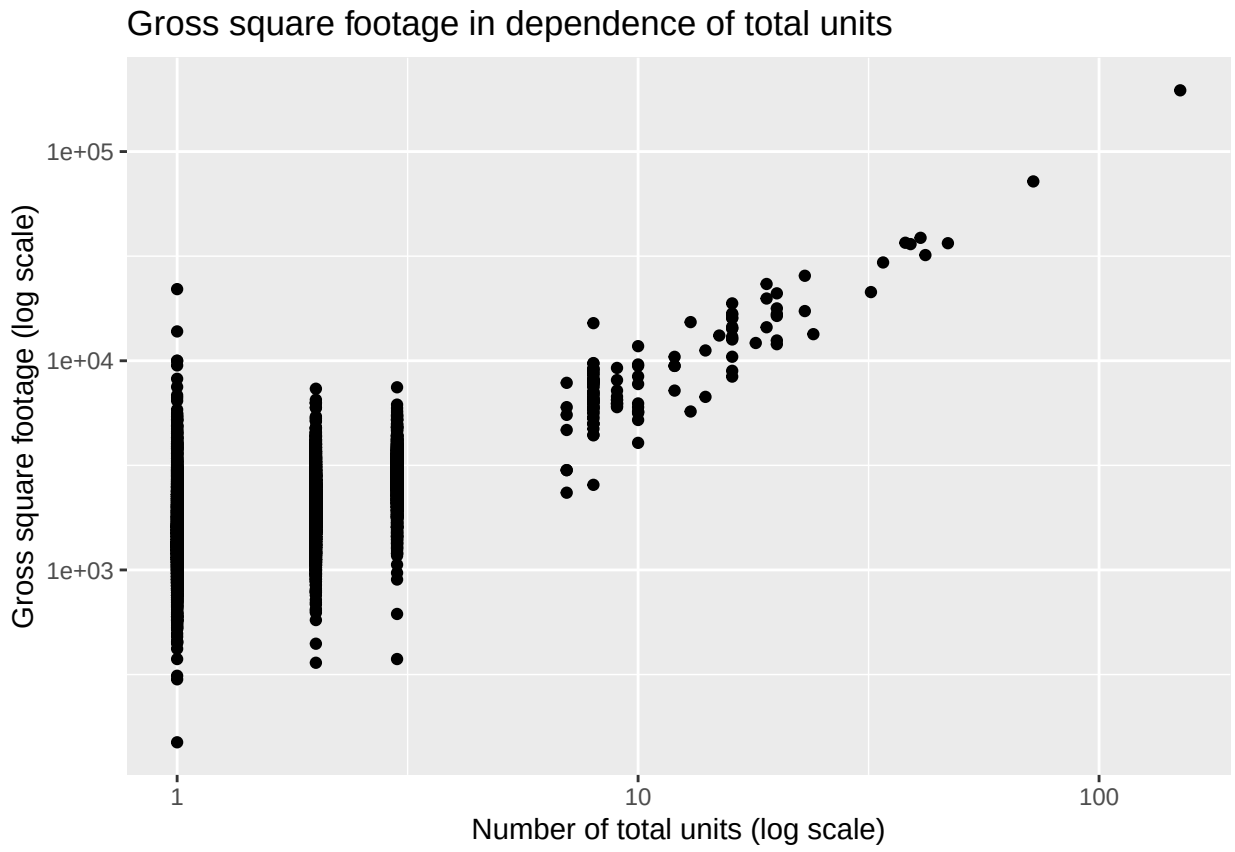
For the borough / building class combinations for which no square foot prices are available, the gross square footage a is approximated by a smooth function of the number of units u, so

$$\hat{a} = f(u).$$

To determine the type of function f, we follow these steps:

a) Determine which borough / building class combinations required this approximation.
b) Extract all data for which both the square footage and the number of total units are available and examine a scatter plot to determine the type of function to fit
c) Train any parameters of the function using k-fold cross-validation.

For steps a) and b), the code below visualises the relationship between gross square footage and number of total units:

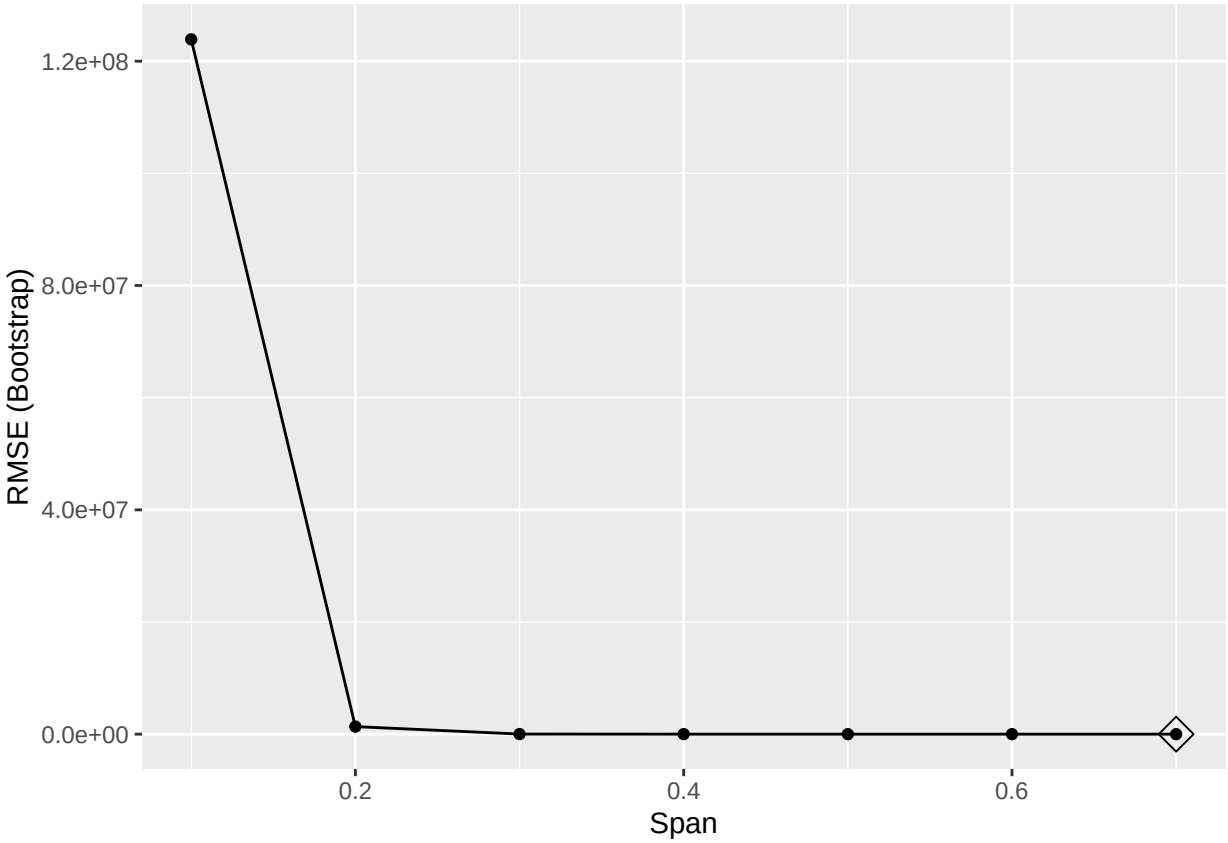


So the relationship is smooth but non-linear, but locally it can be linear. This is a good application for a local regression (LOESS). This is implemented in the `gamLoess` model in the `gam` package and comes with a `caret` interface for training the `span` parameter that determines the size of the window for the local regressions for step c):

```
set.seed(3532, sample.kind = "Rounding")
train_bbc_L1_inc <- train(gross_square_feet ~ total_units,
                          data = train_nycp_bbc_L1_inc,
                          method = "gamLoess",
                          tuneGrid = data.frame(degree = 1,
                                                span = seq(0.10, 0.70, 0.1)))
save(train_bbc_L1_inc, file = "data/model_sqf_LOESS.Rdata")
```

The model generated by the code above is saved to the file `model_sqf_LOESS.Rdata` and loaded from there for the report to evaluate the optimal `span` parameter:

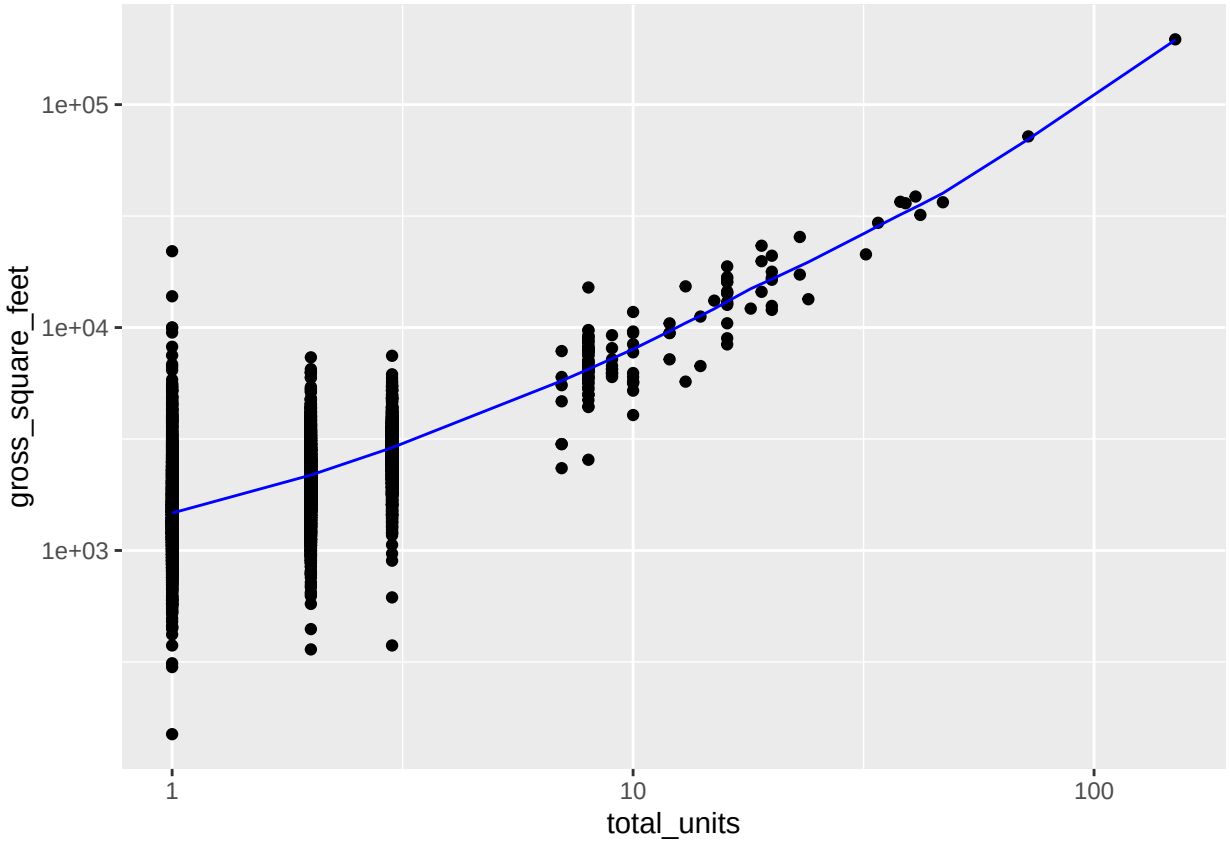
```
load(file = "data/model_sqf_LOESS.Rdata")
ggplot(train_bbc_L1_inc, highlight = TRUE)
```




```
train_bbc_L1_inc$bestTune
```

```
##   span degree
## 7  0.7      1
```

This model produces a good fit:



Note that during the evaluation on the test set, the model may produce warning messages for any number of units that are larger than used for the training set. Given the good-quality fit shown above, these messages may be safely ignored.

Before the model is used on test data, the table of square foot prices is extended using the training data:

```
# Take data with total units > 0 and gross_square_feet < FOOTAGE_MIN to
# a) put in proxy for gross square feet
# b) use that proxy to predict price/ft^2
price_gsf_L1_tbl_inc <- train_nycp %>%
    filter(gross_square_feet < FOOTAGE_MIN & total_units > 0) %>%
    mutate(gross_square_feet_hat = predict(train_bbc_L1_inc, newdata = .)) %>%
    group_by(borough, building_class_at_time_of_sale) %>%
    summarise(nprices = n(),
              price_per_gsf_harm = harm.mean(sale_price/gross_square_feet_hat))

# only keep (borough, bldg class at time of sale) combos
# that are NOT already in price_gsf_L1_tbl
price_gsf_L1_tbl_inc <- price_gsf_L1_tbl_inc %>%
    anti_join(price_gsf_L1_tbl, by = c("borough",
                                       "building_class_at_time_of_sale"))

# and get new table
price_gsf_L2_tbl <- rbind(price_gsf_L1_tbl,
                          price_gsf_L1_tbl_inc)
```

3.5.3.3 Step 3: Absolute prices for any borough / building class combinations

Although absolute prices are the final fallback, the table of these prices should be as complete as possible so is already computed for all borough / building combinations from the training and the test set in the table `bbc_L3_tbl`.

```
# compile table of borough / bldg class at TOS for Step 3
bbc_L2_inc_tbl <- train_nycp %>%
    count(borough, building_class_at_time_of_sale)

# ensure that combos from test set also covered
bbc_L2_inc_tbl_add <- test_nycp %>%
    count(borough, building_class_at_time_of_sale)

# merge the two tables
bbc_L3_tbl <- full_join(bbc_L2_inc_tbl, bbc_L2_inc_tbl_add,
                  by = c("borough", "building_class_at_time_of_sale"))
rm(bbc_L2_inc_tbl, bbc_L2_inc_tbl_add)
```

The k-fold cross-validation is used to determine the type of average that produces the best fit, arithmetic average, geometric average or median:

```
rmse_v_L3 <- map_df(1:length(index_list), function(listIdx) {
    # split data into training and test set
    train_set <- train_nycp[-index_list[[listIdx]], ]
    test_set <- train_nycp[index_list[[listIdx]], ]

    # calculate different average prices on training set
    price_L3_tbl <- train_set %>%
        left_join(bbc_L3_tbl,
                  by = c("borough","building_class_at_time_of_sale")) %>%
        group_by(borough, building_class_at_time_of_sale) %>%
        summarise(nprices = n(),
                  price_hat = mean(sale_price),
                  price_hat_w = log10mean(sale_price),
                  price_hat_med = median(sale_price))

    # calculate RMSE in test set, ONLY FOR THOSE ROWS for which
    # square footage is neither directly available (>= FOOTAGE_MIN) nor
    # can be approximated (total_units > 0)
    predictions <- test_set %>%
        filter(gross_square_feet < FOOTAGE_MIN & total_units == 0) %>%
        inner_join(price_L3_tbl,
                   by = c("borough","building_class_at_time_of_sale")) %>%
        mutate(perdiff = price_hat / sale_price - 1,
               perdiff_w = price_hat_w / sale_price - 1,
               perdiff_med = price_hat_med / sale_price - 1)
    predictions %>%
        summarise(RMSE_avg = mean(perdiff),
                  RMSE_avg_w = mean(perdiff_w),
                  RMSE_avg_med = mean(perdiff_med))
    })

# evaluate results
rmse_v_L3 %>% summarise_all(mean)
```

```
## # A tibble: 1 x 3
```

```
##   RMSE_avg RMSE_avg_w RMSE_avg_med
##      <dbl>      <dbl>        <dbl>
## 1    0.769      0.301        0.250
```

The median provides the best approximation, so the table of absolute prices `price_L3_tbl` is compiled using the median:

```
price_L3_tbl <- train_nycp %>%
    left_join(bbc_L3_tbl,
              by = c("borough","building_class_at_time_of_sale")) %>%
    group_by(borough, building_class_at_time_of_sale) %>%
    summarise(nprices = n(),
              price_hat_med = median(sale_price))

# remove temporary variables
rm(rmse_v_L3)
```

3.5.3.4 Step 4: Prices extrapolated from other boroughs

The extrapolation implementation involves a lot of data wrangling for which the code is not displayed here for better readability. The main function that is used for extrapolation from borough / building class combinations within the same building class to other boroughs uses the harmonic mean that has been shown to be the optimal type of average in paragraph 3.5.3.1:

```
#' Function to fill in $/sqft table,  again use HARMONIC mean,
#' in a weighted variant
#'
#' @param price_tbl Table of prices per square foot
#' @param building_class Building class to extend the table for, e.g. "A0"
#' @return Data frame of prices for all boroughs not yet covered
price_gsf_lookup <- function(price_tbl, building_class) {
    price_tbl %>%
        filter(building_class_at_time_of_sale == building_class &
                   !is.na(nprices)) %>%
        group_by(building_class_at_time_of_sale) %>%
        summarise(price_per_gsf_harm = sum(nprices)/
                      sum(nprices/price_per_gsf_harm))
    }
```

For the remainder of the code, please refer to the source file `nycproperty.R`.

3.5.3.5 Step 5: Prices extrapolated for building classes of same main category

Similar to the preceding section, this step involves many data operations which are not that interesting from a methodological standpoint. Again, the main extrapolation function that extends the table of prices for similar building classes uses the harmonic mean shown to be optimal in paragraph 3.5.3.1:

```
#' second helper function to fill in price/sqft using HARMONIC mean
#'
#' @param price_tbl Table of prices per square foot
#' @param bldg_class Building class to extend the table for, e.g. "A0"
#' @return Data frame of average price across major building class
price_gsf_fill <- function(price_tbl, bldg_class) {
    # extract main class
    main_class = str_sub(bldg_class, 1, 1)
    # take average price over all prices/sqft with that main class
    res <- price_tbl %>%
```

```
        mutate(mainclass = str_sub(building_class_at_time_of_sale,1,1)) %>%
        filter(mainclass == main_class) %>%
        group_by(mainclass) %>%
        summarise(price_per_gsf_harm = sum(nprices)/
                      sum(nprices/price_per_gsf_harm)) %>%
        select(-mainclass)
    res <- res %>%
        mutate(building_class_at_time_of_sale = bldg_class)
    return(res)
    }
```

For the remainder of the code, please see the R script file `nycproperty.R`. The only thing worth noting is that even with this second extrapolation not all borough / building class combinations are covered, but this only affects one record in the training set, and no records in the test set, so will have no effect on the final evaluation of the model. The final square-foot-price table by borough / building class is stored in `price_gsf_L2_tbl_final`.

3.5.3.6 Implementing the final function

With the five previous steps, the cascade can be implemented using these objects:

1. Table of square-foot prices (`price_gsf_L2_tbl_final`).
2. Model to predict square footage from numer of total units (`train_bbc_L1_inc`).
3. Table of absolute prices (`price_L3_tbl`).

```
#' Function to predict prices using cascade of methods
#'
#' @param newdata Property data for which to predict sale price
#' @param price_gsf_tbl Table of prices per gross sq ft per
#' borough / building class combination
#' @param model_gsqf_totalunits Model to predict gross square feet from number
#' of total units (total_units)
#' @param price_tbl Table of prices per borough / building class combination
predict_price_sqf <- function(newdata,
                              price_gsf_tbl,
                              model_gsqf_totalunits,
                              price_tbl) {
    newdata %>%
        left_join(price_gsf_tbl,
                  by = c("borough", "building_class_at_time_of_sale")) %>%
        mutate(method = ifelse(gross_square_feet >= FOOTAGE_MIN, "gsf",
                               ifelse(total_units > 0, "gsf_hat", "price"))) %>%
        mutate(gsf = ifelse(gross_square_feet >= FOOTAGE_MIN,
                            gross_square_feet,
                            ifelse(total_units > 0,
                                   predict(model_gsqf_totalunits, newdata = .),
                                   NA))) %>%
        left_join(price_tbl,
                  by = c("borough", "building_class_at_time_of_sale")) %>%
        mutate(sale_price_hat =
                   ifelse(method %in% c("gsf", "gsf_hat"),
                          gsf * price_per_gsf_harm,
                          price_hat_med),
               perdiff = sale_price_hat/sale_price - 1) %>%
        .$sale_price_hat
```

```
    }
```

4 Results

The model errors are estimated for each of the three main models (and five submodels) presented in subsection 3.5 and collected in a table for comparison. All values are computed to the full seven digits to show also marginal differences in performance. The code follows these steps for each method:

1. Calculate estimate according to the method on the `test_nycp` data set.
2. Compute average percentage error for this estimate on the `test_nycp` data set.
3. Store results in table `rmse_results`.

```
# 1 Constant absolute price
# predict prices
price_hat_avg <- predict_const_avg(test_nycp)
# calculate RMSE and store in results table
rmse_const_avg <- RMSE_price(price_hat_avg, test_nycp$sale_price)
rmse_results <- tibble(Method = "Simple arithmetic average price",
                       AvgPerDiff = rmse_const_avg)

# predict prices
price_hat_med <- predict_const_med(test_nycp)
# calculate RMSE and store in results table
rmse_const_med <- RMSE_price(price_hat_med, test_nycp$sale_price)
rmse_results <- rbind(rmse_results,
                      tibble(Method = "Simple median price",
                             AvgPerDiff = rmse_const_med))

# 2 Constant sqf price
# predict prices, with arithmetic avg for both sqf price and fallback
price_hat_sqf_const_avg <- predict_sqf_const_avg(test_nycp)
# calculate RMSE and store in results table
rmse_sqf_const_avg <- RMSE_price(price_hat_sqf_const_avg, test_nycp$sale_price)
rmse_results <- rbind(rmse_results,
                      tibble(Method = "Square-foot price arithmetic average",
                             AvgPerDiff = rmse_sqf_const_avg))

# predict prices, with harmonic avg for sqf price and geom mean as fallback
price_hat_sqf_const_harm <- predict_sqf_const_harm(test_nycp)
# calculate RMSE and store in results table
rmse_sqf_const_harm <- RMSE_price(price_hat_sqf_const_harm,
                                  test_nycp$sale_price)
rmse_results <- rbind(rmse_results,
                      tibble(Method = "Square-foot price harmonic average",
                             AvgPerDiff = rmse_sqf_const_harm))

# 3 Sqf price by borough and building class, cascading with LOESS
# model to approximate square footage

# predict prices, with class median
price_hat_sqf <- predict_price_sqf(test_nycp,
                                   price_gsf_tbl = price_gsf_L2_tbl_final,
                                   model_gsqf_totalunits = train_bbc_L1_inc,
                                   price_tbl = price_L3_tbl)
```

```r
# calculate RMSE and store in results table
rmse_sqf <- RMSE_price(price_hat_sqf, test_nycp$sale_price)
rmse_results <-
    rbind(rmse_results,
          tibble(Method = "Cascading sqft price by borough and building class",
                                AvgPerDiff = rmse_sqf))

# display table of results
rmse_results %>% knitr::kable()
```

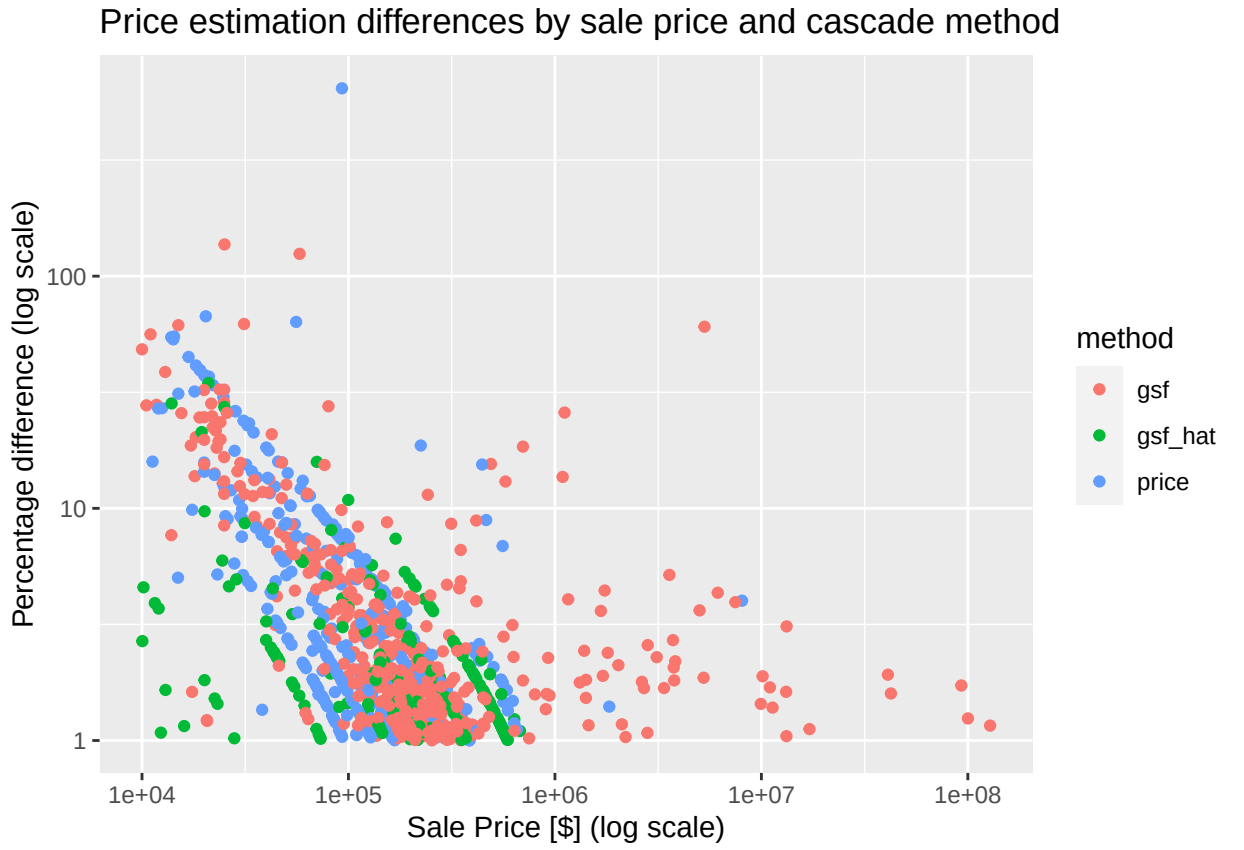
Method	AvgPerDiff
Simple arithmetic average price	2.0903066
Simple median price	0.5820559
Square-foot price arithmetic average	2.9084590
Square-foot price harmonic average	0.4686871
Cascading sqft price by borough and building class	0.2282295

The mathematical formulation of the model is shown in the following overview:

Method	Model Formula
Simple arithmetic average price	$Y_{b,c} = \mu^{(Y)} + \varepsilon_{b,c}$
Simple median price	$Y_{b,c} = median^{(Y)} + \varepsilon_{b,c}$
Square-foot price arithmetic average	$Y_{b,c} = \mu^{(p)} \times a + \varepsilon_{b,c}$
Square-foot price harmonic average	$Y_{b,c} = \mu_{harm}^{(p)} \times a + \varepsilon_{b,c}$
Cascading square-foot price by borough and building class	$Y_{b,c} = \mu_{b,c}^{(p)} \times a + \varepsilon_{b,c}$

The last most clearly produces the best performance with an average price difference of 23%.

We examine the largest (below -100% and above 100%) differences by sale price and method withing the cascade (`gsf` - square footage available, `gsf_hat` - square footage approximated by number of total units and `price` - price fallback used as neither square footage nor total units available).



It is evident from the plot that:

- All large percentage differences are positive.
- The largest percentage differences occur for low prices.
- There is an even distribution among the cascade methods.

5 Conclusion

The data set of property transactions is challenging because it:

- Contains sales with low prices that do not constitute real sales.
- Does not contain net square footage and calculating net square footage based on gross square footage and land square footage does not lead to plausible results, implying poor data quality.
- Often both the number of residential units are commercial units are zero while the number of total units is one, also implying data quality issues.

So the overall best method among those tested calculates square-foot prices as harmonic average by borough and building class (at time of sale) and approximates square footage where not otherwise given using a LOESS model based on the number of total units, falling back to harmonic average of prices per borough and building class as last resort. The obtained percentage difference of prices is 23%, going down to -10% when realistic property values above $100,000 are considered. The model formula is

$$Y_{b,c} = \mu_{b,c}^{(p)} \times a + \varepsilon_{b,c}.$$

The model could be further improved by

- Incorporating an additional effect for the building class category for the most frequent building codes.
- Adding location effects - for example using the ZIP code.

- Adding effects for the year in which the property was built.

Appendix A - Code of Data Import

To enhance readability of the report, the code for data download in subsubsection 3.2.1 is shown in this Appendix.

Data Download in Section 3.2 - Data Structure and Loading

The code for data download in subsubsection 3.2.1:

```
# link address
url <- "https://raw.githubusercontent.com/danmayen/nycproperty/master/
data/nyc-property-sales.zip"
# download file from url
tmp_filename <- tempfile()
csv_filename <- "nyc-rolling-sales.csv"
download.file(url, tmp_filename)
# unzip file
unzip(tmp_filename, csv_filename)
# remove temporary zipfile
file.remove(tmp_filename)

# read csv file into data frame
nycproperty_raw <- read_csv(csv_filename)
# remove csv file
file.remove(csv_filename)
# remove temporary variables
rm(url, tmp_filename, csv_filename)

# column names with underscores and lowercase
newcolnames <- colnames(nycproperty_raw) %>%
    str_to_lower() %>%
    str_replace_all(" ", "_") %>%
    str_replace_all("-", "")
colnames(nycproperty_raw) <- newcolnames

# save to one file for report
save(nycproperty_raw, file = "data/nycproperty_raw.Rdata")

# remove temporary variables
rm(url, tmp_filename, csv_filename, newcolnames)
```