# Apartment Rental Prediction System

Dr. Ivan S. Zapreev

2020-01-09

## Contents

# Introduction

In this report we shall address building of an apartment rental price prediction system. The latter will be based on the 'Apartment rental offers in Germany' dataset available from:

https://www.kaggle.com/corrieaar/apartment-rental-offers-in-germany.

To reach our goal, we will use supervised machine learning techniques studied within the *"PH125.8x Data Science: Machine Learning"* course (a part of the broader HarvardX *Data Science Professional* certification program). The prediction system will be therefore based on (linear) statistical models trained on the subset of the 'Apartment rental offers in Germany' dataset.

The remainder of this section first present the data set, then defines the goal of this project more concretely, and finally identifies the main steps to be taken to reach the goal.

## Dataset overview

As stated on the webpage of the 'Apartment rental offers in Germany' dataset, it contains `198,379` rental offers scraped from the Germany's biggest real estate online platform ß ImmobilienScout24.

The data set consists of a single CSV file: *immo_data.csv* which only contains offers for rental properties. The data features important rental property attributes, such as the living area size, the rent (both base rent as well as total rent), the location, type of energy, and etc. The `date` column present in the data set defines the time of scraping, which was done on three distinct dates: *2018-09-22*, *2019-05-10* and *2019-10-08*.

The complete list of data set columns is extensive[1] and thus in this study we will use the following subset:

```
##  [1] "hasKitchen"            "heatingType"           "balcony"
##  [4] "lift"                  "garden"                "cellar"
##  [7] "noParkSpaces"          "livingSpace"           "typeOfFlat"
## [10] "noRooms"               "floor"                 "numberOfFloors"
## [13] "condition"             "newlyConst"            "interiorQual"
## [16] "yearConstructed"       "energyEfficiencyClass" "regio1"
## [19] "regio2"                "regio3"                "baseRent"
## [22] "electricityBasePrice"  "heatingCosts"          "serviceCharge"
## [25] "totalRent"             "date"
```

This sub-selection reduces the number of considered data set columns[2] from 48 to 26 and is motivated by the personal preferences of the report's author and has no scientifically proven motivation. On the contrary, this column selection shall be seen as a part of problem statement. In other words, the task is to build an accurate[3] rental price prediction model based on the predictors from this set of columns.

The additional data preparation steps will be described in the *"Data wrangling"* section of this document.

[1] Please consider reading *"Appendix A"* for the complete list of the data set columns.
[2] Please consider reading *"Appendix B"* for the column descriptions.
[3] Please consider reading the *"Project goal"* section for an exact goal formulation.

## Project goal

The goal of this project is to build a apartment rental-price prediction system for German cities. The model is to be trained and validated using the data from 'Apartment rental offers in Germany' dataset. The data is therefore to be split into a `modeling` and `validation` sets which will be defined in the *"Data wrangling"* section.

The system evaluation will be done using the Residual Mean Squared error (`RMSE`) which, similarly to a standard deviation, can be interpreted as: *the typical error we make when predicting a rent price.* In other words, `RMSE` will indicate an approximate amount of Euros we are on average off in our rental price predictions.

**Definition:** (`RMSE`)

Let $r_o$ be the true rental price for an offer $o$, $N_o$ be the number of offers, and $\hat{r}_o$ be our prediction, then:

$$RMSE = \sqrt{\sum_{m=1}^{N_o} (r_o - \hat{r}_o)^2}$$

The definition above trivially translates into the following `R` function:

```
RMSE <- function(true_ratings, predicted_ratings){
    sqrt(mean((true_ratings - predicted_ratings)^2))
  }
```

The ultimate goal of the project is to provide a statistical model, solely based on the modeling set, that on the `validation` set will be able to predict apartment rental prices with `RMSE` $\leq$ `TARGET_RMSE` $= 50$. The way achieve that goal will be explained in the *"Modeling approach"* section of this document.

## Execution plan

Let us now briefly outline the main steps to be performed to reach the previously formalized project goal:

1. **Prepare the data** – see the *"Data wrangling"* section:
   - Select, clean, and reshape relevant data; split it into training and validation sets; and etc.
2. **Analyze the dataset** – see the *"Dataset analysis"* section:
   - Perform data exploration and visualization; summarize insights on the data.
3. **Describe the modeling approach** – see the *"Modeling approach"* section:
   - Consider the insights of the data analysis; suggest the way obtain a prediction model.
4. **Present modeling results** – see the *"Results"* section:
   - Train the model(s); evaluate the model(s); analyze the results.
5. **Provide concluding remarks** – see the *"Conclusions"* section:
   - Summarize the results; mention any approach limitations; outline possible future improvements.

# Data wrangling

In this section we present cleaning, enriching, and restructuring the raw data taken from the 'Apartment rental offers in Germany' dataset.

This section will be organized as follows: First we explain how we cleaned the data and solved some of its inconsistencies, by enriching the data. Then we identify some structural changes done to the data. Further, we provide a summary of the wrangled data set. In the end, we explain how we split the entire data set into the `validation` and `modeling` sub-sets[4].

[4]The latter will also be split into the `training` and `testing` set for the sake of model cross-validation.

## Data cleaning & enriching

Let us note that the number of data entries in the original data set is equal to 198332. This data is however not ready to be worked with as it is very dirty. It contains multiple `N/A` values; is inconsistent – has mismatching row values, e.g. `floor = 10` and `typeOfFlat = "roof_storey"`; and has multiple outliers in numerical/integer columns.

The rest of the section is organized as follows: First, we explain cleaning of `N/A` values. Second, we discuss stripping of the data from the outliers. Third, we outline filtering out and correcting some of the data inconsistencies.

### Removing `N/A` values

Consider for example the next table summarizing the number of `N/A` values per data set column:

```
## # A tibble: 26 x 3
##    `Column name`         `N/A count` `N/A percent`
##    <chr>                       <int>         <dbl>
##  1 electricityBasePrice       151158         76.2
##  2 energyEfficiencyClass      143316         72.3
##  3 heatingCosts               135154         68.2
##  4 noParkSpaces               130405         65.8
##  5 interiorQual                83002         41.8
##  6 numberOfFloors              71792         36.2
##  7 condition                   50318         25.4
##  8 yearConstructed             42293         21.3
##  9 floor                       37612         19.0
## 10 heatingType                 32605         16.4
## 11 totalRent                   29762         15.0
## 12 typeOfFlat                  27572         13.9
## 13 serviceCharge                5110          2.58
## 14 hasKitchen                      1          0
## 15 lift                            1          0
## 16 garden                          1          0
## 17 cellar                          1          0
## 18 livingSpace                     1          0
## 19 noRooms                         1          0
## 20 regio2                          1          0
## 21 regio3                          1          0
## 22 baseRent                        1          0
## 23 date                            1          0
## 24 balcony                         0          0
## 25 newlyConst                      0          0
## 26 regio1                          0          0
```

As one can see, about $\frac{1}{2}$ of the columns has 2.5–80% `N/A`$^s$, whereas the other half has (almost) no `N/A`$^s$.

Cleaning the data from `N/A` values will be explained in the next steps:

1. We begin with the `totalRent` column as this is the value that we want to predict;
2. We proceed with the columns with the marginal (< 1%) of `N/A` values;
3. We cover the remaining columns in the descending order of the number of `N/A` values.

#### The first steps

The `totalRent` column contains data that we want to predict. Therefore, the rows with `totalRent` == `N/A` are useless to us and shall be removed. Unfortunately, this will reduce the data set by 15.01%. There are also 13 columns with a marginal (0 to 1) number of `N/A` values. The latter can be seamlessly removed as even if all of these `N/A`$^s$ appear in different rows, we will remove at most 13 entries which is just 0.0066% of data.

### The main columns

Let us consider the columns one by one. Note that, some modifications we will do to the data to remove the `N/A` values may introduce bias. To for test that we would need a clean data set with no `N/A` values initially present and then to use such a data set for the trained model(s) validation. Due to the lack of time this will not be done in the case study.

#### Column: `electricityBasePrice` - 76.2% `N/A` values

We will set the electricity base price for the `N/A` values to zero. The motivation is that, since the number of `N/A` values is almost 80% and no other zero values are present:

```
x <- arog_data$selected_data %>% filter(!is.na(electricityBasePrice))
sum(x$electricityBasePrice == 0)
```

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

it is likely that the `N/A` values were used to determine the fact that there is no electricity base price.

#### Column: `energyEfficiencyClass` - 72.3% `N/A` values

The energy efficiency factor levels are:

```
levels(arog_data$selected_data$energyEfficiencyClass)
```

```
##  [1] "A"              "A_PLUS"         "B"              "C"
##  [5] "D"              "E"              "F"              "G"
##  [9] "H"              "NO_INFORMATION"
```

So we shall naturally set all the `N/A` energy efficiency levels to `"NO_INFORMATION"`.

#### Column: `heatingCosts` - 68.2% `N/A` values

We will set the heating costs for the `N/A` values to zero as there are already 1989 zero-valued heating cost entries. It is unlikely that there are non-heated accommodations in Germany so *we assume that the* 0 *values, the same as `N/A`$^s$ mean - "unknown"*.

#### Column: `noParkSpaces` - 65.8% `N/A` values

We will set the number of parking places for the `N/A` values to zero as there is already 2850 zero-valued entries. By this step we assume that, `N/A` is interpreted as *"not applicable"* or *"no are available"*.

#### Column: `interiorQual` - 38.8% `N/A` values

The interior quality factor levels are:

```
levels(arog_data$selected_data$interiorQual)
```

```
## [1] "luxury"        "normal"        "simple"        "sophisticated"
```

So we shall introduce a new level for the `N/A` values, called `"unknown"`.

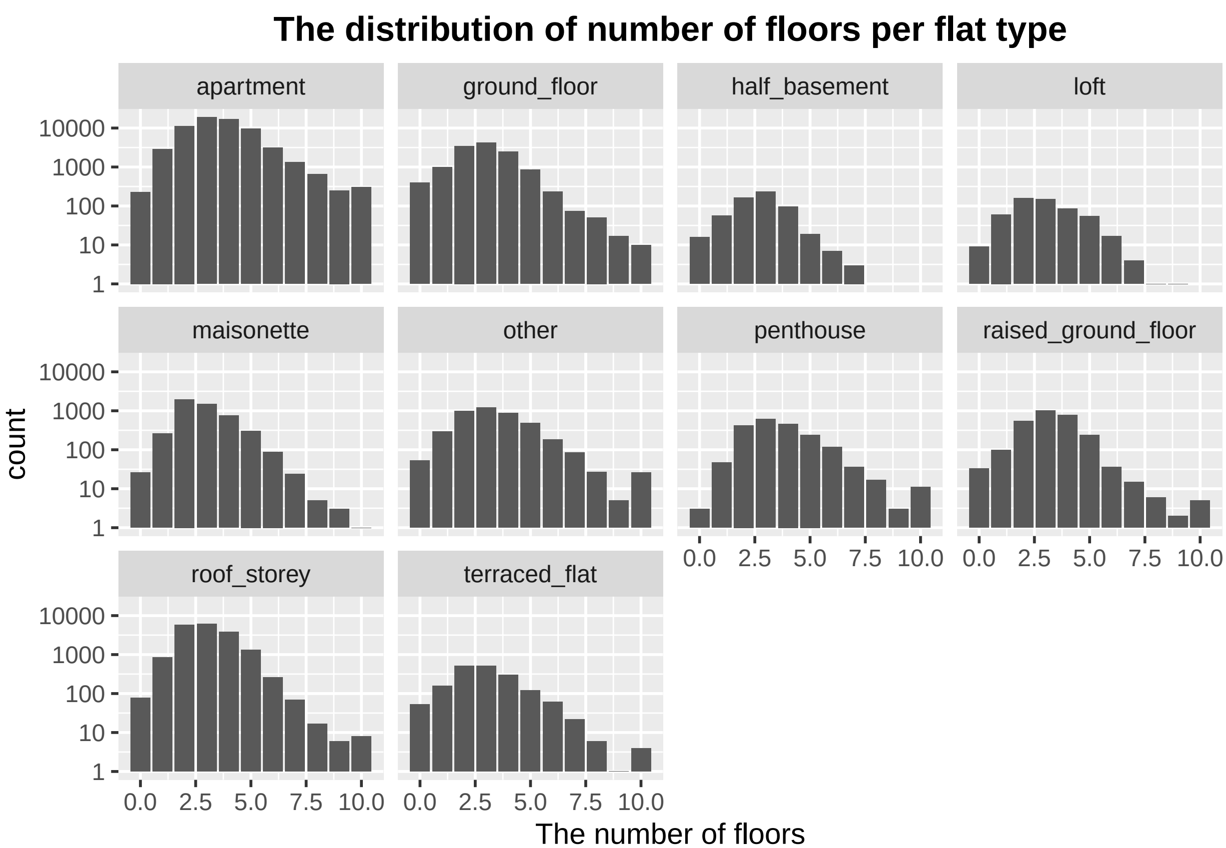
**Column: `numberOfFloors` - 36.2% `N/A` values**

Setting the `N/A` values for the floors shall be agreed with the apartment type, if we gather some number of floors statistics for each available apartment type we get the following:

Table 1: Type of flat vs. number of floors statistics

| | numberOfFloors | | | | |
|---|---|---|---|---|---|
| typeOfFlat | Count | Average | Standard error | Minimum | Maximum |
| apartment | 66844 | 3.9 | 6.6 | 0 | 999 |
| roof_storey | 18450 | 3.1 | 6.7 | 0 | 800 |
| ground_floor | 12802 | 3 | 8.9 | 0 | 999 |
| maisonette | 4979 | 2.9 | 1.5 | 0 | 43 |
| other | 4284 | 3.6 | 5 | 0 | 301 |
| raised_ground_floor | 2779 | 3.4 | 7.3 | 0 | 370 |
| penthouse | 2011 | 3.7 | 2.2 | 0 | 33 |
| terraced_flat | 1760 | 3 | 1.5 | 0 | 14 |
| half_basement | 598 | 2.7 | 1.1 | 0 | 7 |
| loft | 542 | 3 | 1.5 | 0 | 15 |

From where we conclude that the data we have is very polluted. Clearly, one can not expect apartments with 99 floors and alike. See also on the large average (all `+/-` around 3 floors) and the huge standard error values. If we visualize the results (filtering out 1662 flats with more than 10 floors), we see that:




the data seems to be approximately normally distributed (except for the `apartment` type) with the mean values within 2.5 - 4.0 range. This makes us believe that this data is too much biased and polluted. So we will not rely on this column in our analysis.

#### Column: `condition` - 25.4% `N/A` values

The condition factor levels are:

```
##  [1] "first_time_use"                     "first_time_use_after_refurbishment"
##  [3] "fully_renovated"                    "mint_condition"
##  [5] "modernized"                         "need_of_renovation"
##  [7] "negotiable"                         "refurbished"
##  [9] "ripe_for_demolition"                "well_kept"
```

So we shall introduce a new level for the `N/A` values, called `"unknown"`.

#### Column: `yearConstructed` - 21.3% `N/A` values

There is no good default to replace the `N/A` values here. Yet, it is a significant amount of data which we do not want to exclude. Therefore drop this column from the analysis and just use the `newlyConst` flag column.

#### Column: `floor` - 19.0% `N/A` values

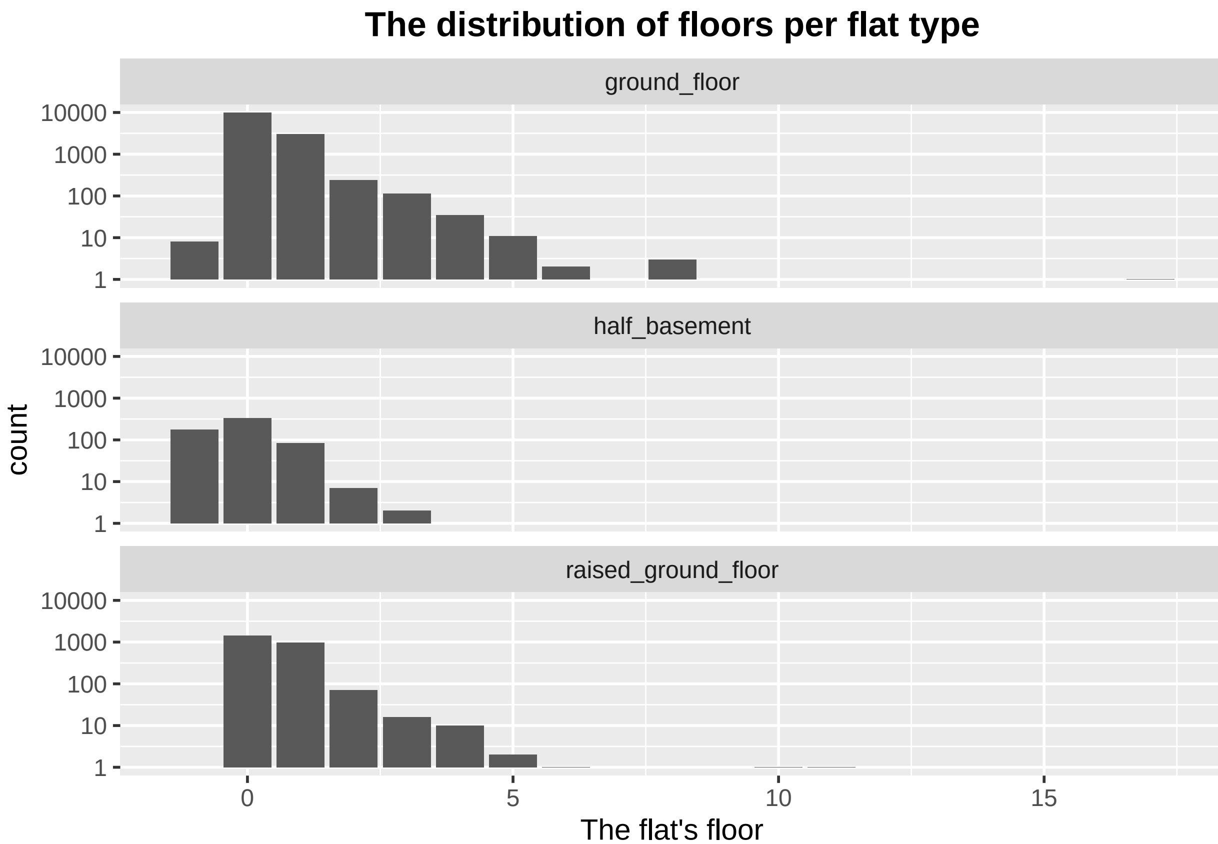
We could assign some `floor` values based on the flat types:

```
##  [1] "apartment"           "ground_floor"        "half_basement"
##  [4] "loft"                "maisonette"          "other"
##  [7] "penthouse"           "raised_ground_floor" "roof_storey"
## [10] "terraced_flat"
```

For example, we could consider assigning:

- `half_basement` – set `floor` to be the average half-basement floor value
- `ground_floor` – set `floor = 0`
- `raised_ground_floor` – set `floor` to be the average raised ground floor value

but, let us look at the floor values (filtering out 10 flats with `floor` > 100), for these flat types:




From the data above we see that we shall not only correct the `N/A` values but set all of the floor values for the considered flat types as follows:

- `half_basement` – set `floor = -1`
- `ground_floor` – set `floor = 0`
- `raised_ground_floor` – set `floor = 0`

If we do that then there will still be 20049 (10.1% of data) `N/A` floor values for the flat types for which we can not give any exact value. So we will just assign those to the mean floor value in the category.

#### Column: `heatingType` - 16.4% `N/A` values

The heating type factor levels are:

```
##  [1] "central_heating"                "combined_heat_and_power_plant"
##  [3] "district_heating"               "electric_heating"
##  [5] "floor_heating"                  "gas_heating"
##  [7] "heat_pump"                      "night_storage_heater"
##  [9] "oil_heating"                    "self_contained_central_heating"
## [11] "solar_heating"                  "stove_heating"
## [13] "wood_pellet_heating"
```

So we shall introduce a new level for the `N/A`, and `"H"` values, called `"unknown"`.

#### Column: `typeOfFlat` - 13.9% `N/A` values

The type of flat factor levels are:

```
##  [1] "apartment"           "ground_floor"        "half_basement"
```

```
##  [4] "loft"                  "maisonette"          "other"
##  [7] "penthouse"             "raised_ground_floor" "roof_storey"
## [10] "terraced_flat"
```

So we shall introduce a new level for the `N/A` values, called `"unknown"`. Note that, we do not use the pre-defined level `"other"` here as we interpret it as known flat type which is just not on the list of available choices.

### Column `serviceCharge` - 2.58% `N/A` values

We will set the service charges for the `N/A` values to zero. The motivation is that, there are:

```
x <- arog_data$selected_data %>% filter(!is.na(serviceCharge))
sum(x$serviceCharge == 0)
```

```
## [1] 2496
```

zero values present, so we interpret the `N/A` values as defining the fact of no additional service charges.

### Filtering outliers

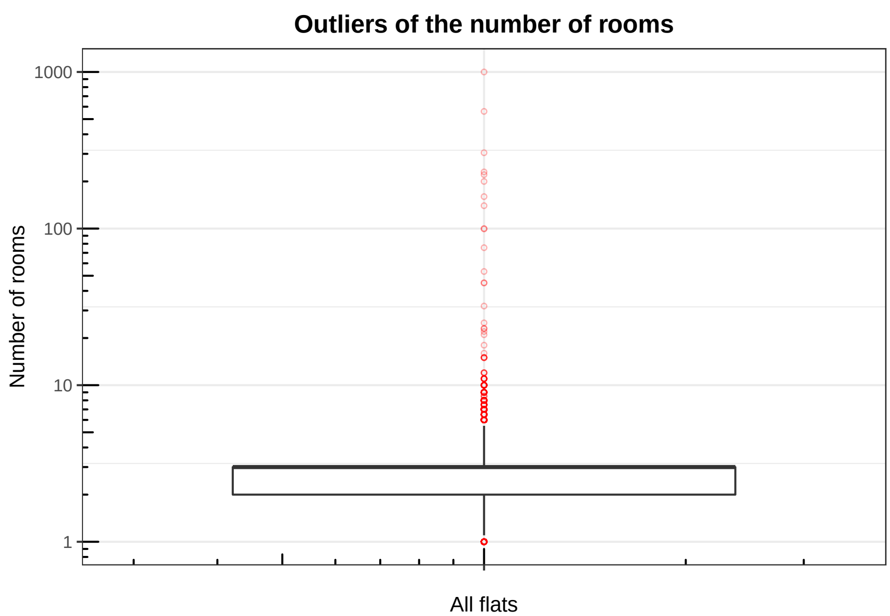
In this section we only considered the numeric/integer columns of the data set. The initial set of outliers per column is obtained using:

```
boxplot.stats(.)$out
```

However, not all of the obtained outlier values are the true outliers. It may be that some flats do stand out as examples of extraordinary property, and not due to owner input errors. This is why, for each of the column, the identified outliers are analyzed and it is then decided on how much of them is to be removed.

### Column: `noRooms`

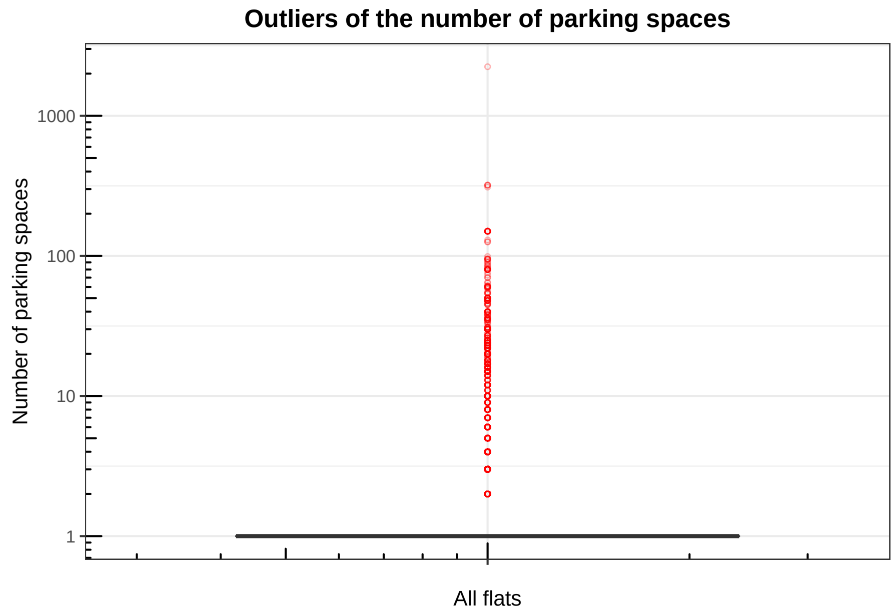
The `noRooms` column has 6038 outliers, see the plot:




We shall remove all the rows with `noRooms` outside the interval [1, 20].

#### Column: `noParkSpaces`

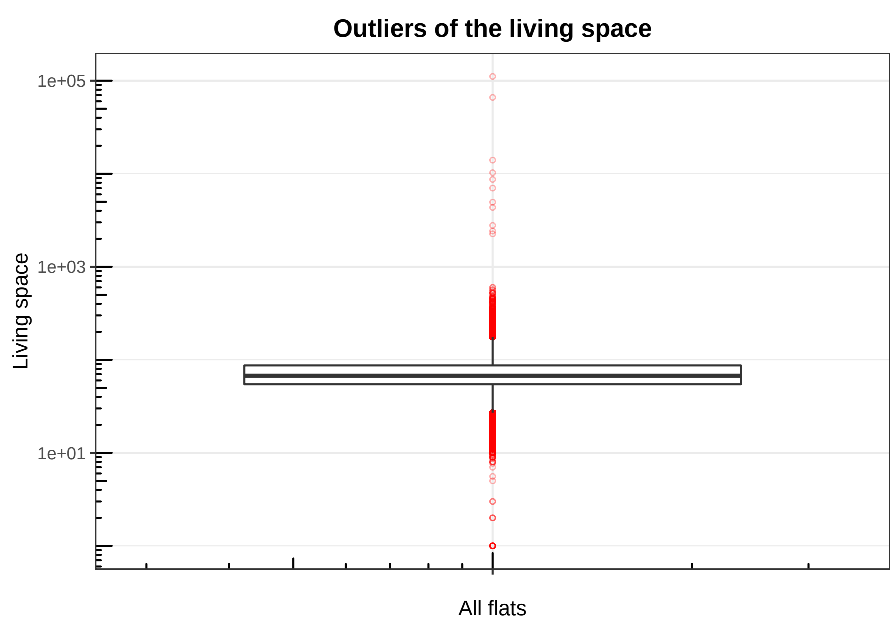
The `noParkSpaces` column has 10416 outliers, see the plot:




We shall remove all the rows with `noParkSpaces` outside the interval [0, 200].

#### Column: `livingSpace`

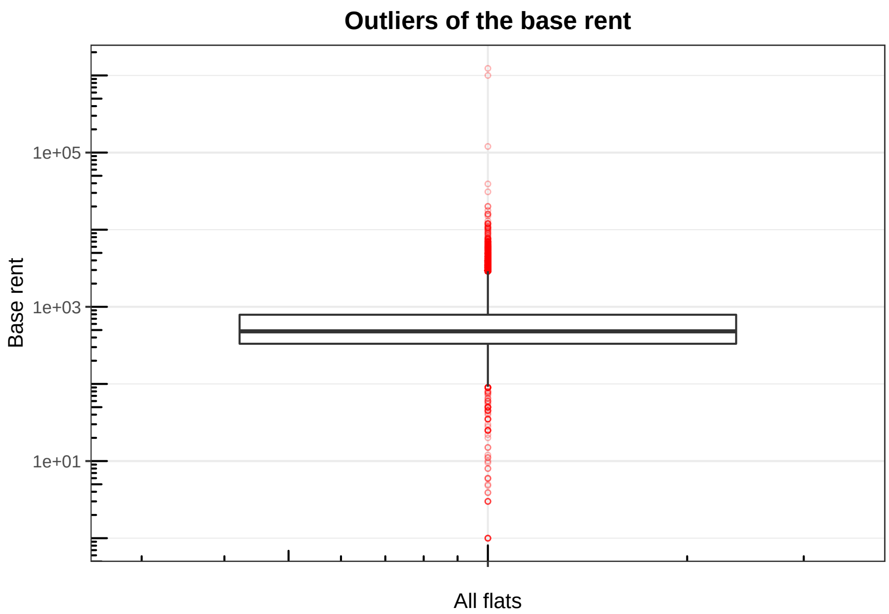
The `livingSpace` has 8830 outliers, see the plot:




We shall remove all the rows with `livingSpace` outside the interval [1, 1000].

#### Column: `baseRent`

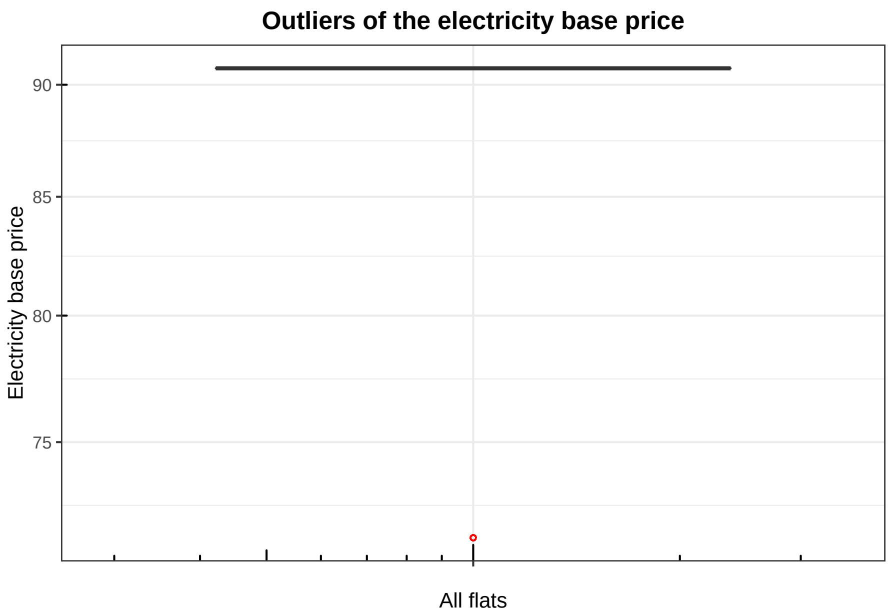
The `baseRent` has 10953 outliers, see the plot:




We shall remove all the rows with `baseRent` outside the interval $[100,\ 3 \times 10^4]$.

### Column: `electricityBasePrice`

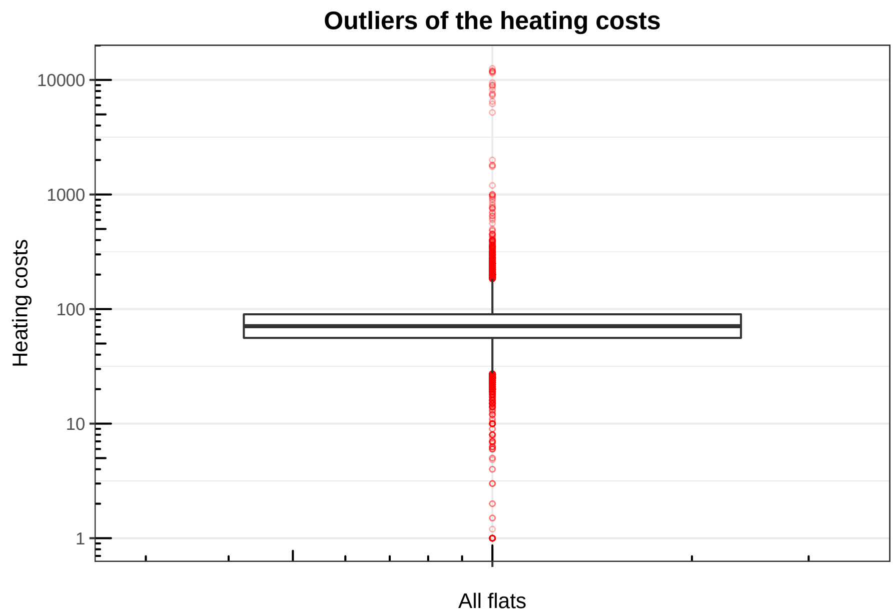
The `electricityBasePrice` has 4048 outliers, see the plot:




We shall remove all the rows with `electricityBasePrice` outside the interval $[0,\ 100]$.

### Column: `heatingCosts`

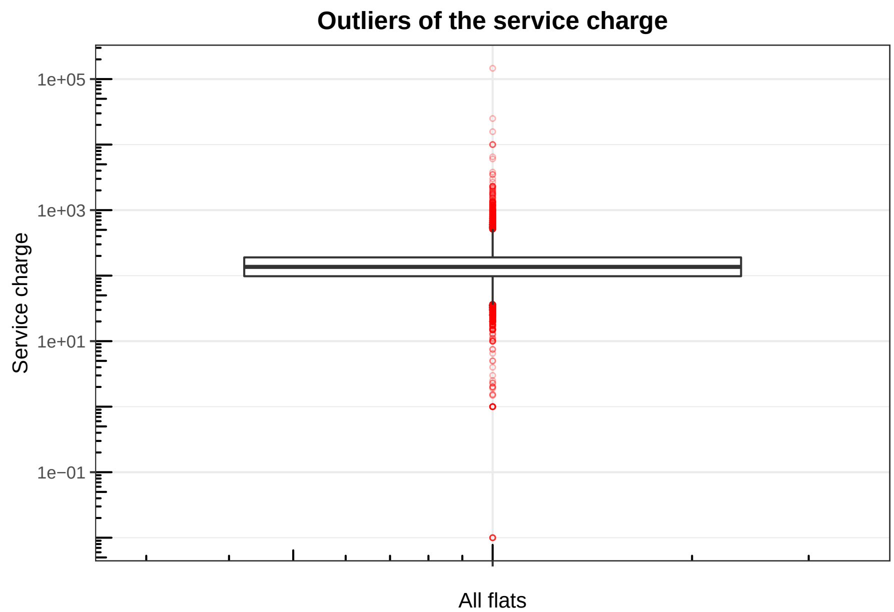
The `heatingCosts` has 4765 outliers, see the plot:




We shall remove all the rows with `heatingCosts` outside the interval [0, 3000].

### Column: `serviceCharge`

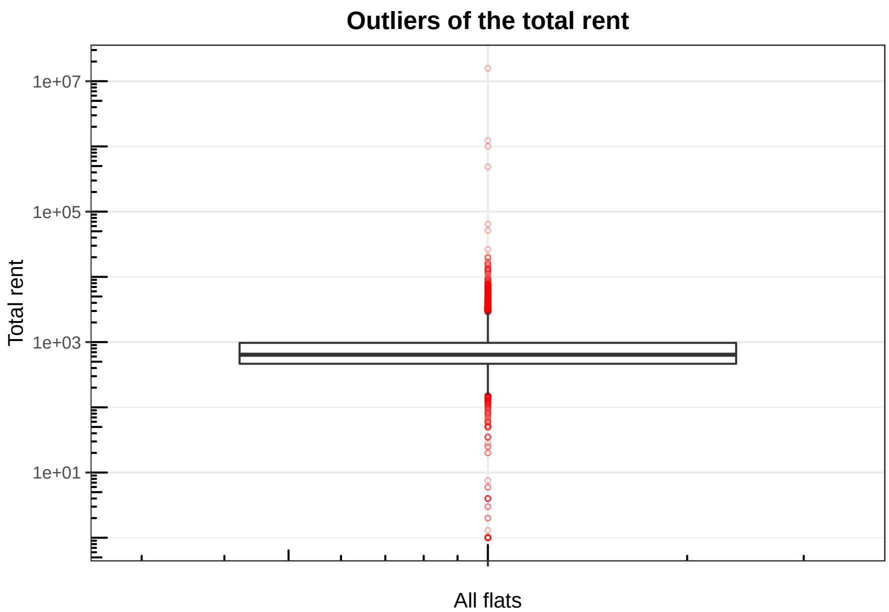
The `serviceCharge` has 6557 outliers, see the plot:




We shall remove all the rows with `serviceCharge` outside the interval $[10, 10^4]$.

### Column: `totalRent`

The `totalRent` has 9366 outliers, see the plot:




We shall remove all the rows with `totalRent` outside the interval $[10, 10^5]$.

### Fixing inconsistencies

In addition to the data alternations done above we have also done the following:

- Re-setting the number of floors:
  - `half_basement` – `floor` $= -1$
  - `ground_floor` – `floor` $= 0$
  - `raised_ground_floor` – `floor` $= 0$
- Filter out flats:
  - Other than `"half_basement"`, `"other"`, and `"unknown"`; but with `floor` $< 0$
  - With zero `totalRent` values (207 entries)
- Fix wrong and convert scraping dates:
  - `"Sep18"` changes into `ymd("2018-09-22")`[5]
  - `"May19"` changes into `ymd("2019-05-10")`
  - `"Oct19"` changes into `ymd("2019-10-08")`
- Made sure that the integer-valued ("floor", "noParkSpaces", "noRooms") columns are rounded.

There may be more inconsistencies present in the data, for example we expect that:

```
totalRent = baseRent + electricityBasePrice + heatingCosts + serviceCharge
```

which may not be the case. Unfortunately due to the lack of time these were not analyzed further and thus may potentially influence the "accuracy" of the trained statistical model.

## Restructuring data

There data set at hand does not have any complex structure. However, because we want to be able to do predictions per city and avoid cities with the same names within different lands and regions we shall combine the `regio` columns into a new single one, as follows:

[5] The `ymd(.)` function is provided by the `lubridate` package.

```r
clean_arog_data <- clean_arog_data %>%
  unite("location", c("regio1", "regio2", "regio3"), remove=FALSE) %>%
  select(-regio1, -regio2, -regio3)
```

The resulting columns have values constructed according to the following pattern:

`location = regio1 + "_" + regio2 + "_" + regio3`

For example:

```r
arog_data$wrangled_data$location[1:5]
```

```
## [1] Nordrhein_Westfalen_Essen_Karnap
## [2] Nordrhein_Westfalen_Steinfurt_Kreis_Emsdetten
## [3] Nordrhein_Westfalen_Bottrop_Lehmkuhle
## [4] Sachsen_Anhalt_Salzlandkreis_Schönebeck_Elbe
## [5] Sachsen_Chemnitz_Bernsdorf
## 8087 Levels: Baden_Württemberg_Alb_Donau_Kreis_Allmendingen ...
```

In addition we have rounded and turned into integer columns all the integer-valued columns of the data set:

```r
c("floor", "noParkSpaces", "noRooms")
```

```
## [1] "floor"        "noParkSpaces" "noRooms"
```

### Wrangled data set

Let us now summarize the resulting clean data:

```
## # A tibble: 22 x 3
##    `Column name`         `N/A count` `N/A percent`
##    <chr>                       <int>         <dbl>
##  1 hasKitchen                      0             0
##  2 heatingType                     0             0
##  3 balcony                         0             0
##  4 lift                            0             0
##  5 garden                          0             0
##  6 cellar                          0             0
##  7 noParkSpaces                    0             0
##  8 livingSpace                     0             0
##  9 typeOfFlat                      0             0
## 10 noRooms                         0             0
## 11 floor                           0             0
## 12 condition                       0             0
## 13 newlyConst                      0             0
## 14 interiorQual                    0             0
## 15 energyEfficiencyClass           0             0
## 16 location                        0             0
## 17 baseRent                        0             0
## 18 electricityBasePrice            0             0
## 19 heatingCosts                    0             0
## 20 serviceCharge                   0             0
## 21 totalRent                       0             0
## 22 date                            0             0
```

As one can notice, the dat set size has been reduced from 198332 to 162742. The major reason for that is excluding the rows with the `N/A` values of the `totalRent` column. Let us recall that the number of such raws was 15.01% of the data set, e.g. 29770 rows. It now remains to notice that $198332 - 29770 = 168562 > 162742$. The remaining delta of 5820 rows ($\approx$ 2.9% of data) is explained by cleaning the `floor`/`typeOfFlat` columns, filtering-out outliers, and etc.

The data has been cleaned but we can expect that there is some noise in the data which we have not addressed. We might get more data-quality insights when we perform data analysis in the subsequent sections.

## Splitting data

To facilitate supervised learning, the wrangled data is split into the `modeling`, 90% of data, and `validation`, 10% of data, sets. The former will be used for training statistical model(s) and the latter for the model(s) validation. Note that, for the sake of subsequent cross validation during modeling part, we further split the `modeling` set into the `training`, 80% thereof, and `testing`, 20% thereof, sets.

We split the data in the following steps:

1. The data is randomly split into to parts according to the specified ratio:

```
test_index <- createDataPartition(y = data_set$totalRent, times = 1,
                                  p = ratio, list = FALSE)
```

2. The `test_index` rows are the candidates for the `testing`/`validation` set rows
3. The factorized column values of the `testing`/`validation` set are considered:

```
str_data <- capture.output(str(arog_data$wrangled_data))
str_replace_all(str_subset(str_data, "Factor"), "levels.*","levels ...")
```

```
## [1] " $ heatingType            : Factor w/ 14 levels ..."
## [2] " $ typeOfFlat             : Factor w/ 11 levels ..."
## [3] " $ condition              : Factor w/ 11 levels ..."
## [4] " $ interiorQual           : Factor w/ 5 levels ..."
## [5] " $ energyEfficiencyClass: Factor w/ 10 levels ..."
## [6] " $ location               : Factor w/ 8087 levels ..."
```

4. The rows with the values not present in the `testing`/`modeling` set are dropped
5. The `testing`/`modeling` set consists of rows absent in the `testing`/`validation` set

The procedure above ensures that the `testing`/`validation` set can always be evaluated on a model trained on the `testing`/`modeling` set. For more details, see the `create_arog_data` and `split_train_test_sets` functions located in the `apartment_rental_project.R` script.

The resulting set sizes are as follows:

- `modeling` – 146660 rows, 90.1% of data
  - `training` – 117820 rows, 80.3% of `modeling` set
  - `testing` – 28840 rows, 19.7% of `modeling` set
- `validation` – 16082 rows, 9.9% of data

As expected, due to returning `testing`/`validation` set rows to the `testing`/`modeling` set for consistency, the desired set ratios are biased. The `validation` set size is almost as prescribed (10% of data), but the `testing` set size is affected more significantly[6]. Yet, we see no issue as the `testing` set is still > 10% of the `modeling` set, which should be enough for performing cross validation.

[6] The requested `testing` set size was 20% of the `modeling` set.

# Data analysis

In this section we shall use data visualization and other techniques to investigate the properties of the data we could use to build the prediction model. In the remainder of the section we shall address:

1. Possible `timing` effects
2. Possible `location` effects
3. Predictor's correlation
4. Principle component analysis

Eventually, we shall conclude the section with a short summary of our findings.

## Possible `timing` effects

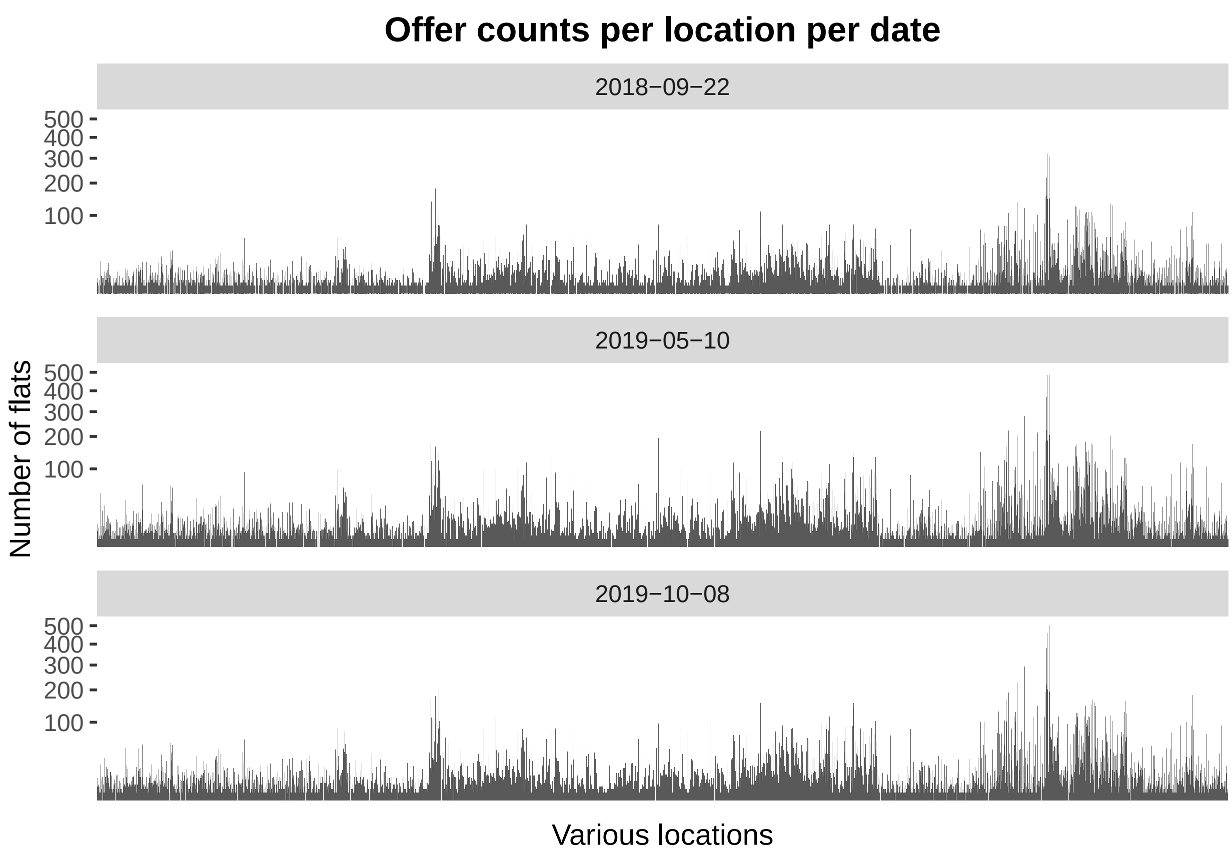
Let us consider any possible effects introduced by the scraping date. In essence, the goal of this section is to understand whether:

1. We shall keep the data scraped on different date distinct, and have the `date` column as a predictor, or
2. We could consider the combined statistics for all the data ignoring the `date` field as a predictor

We answer this question based on the analysis of "Flat offers per `date`", "Flat counts per `location` per `date`", and "Average `totalRent` per `date`"

### Flat offers per `date`

Let us consider the number of flat offers per location per data scraping date:

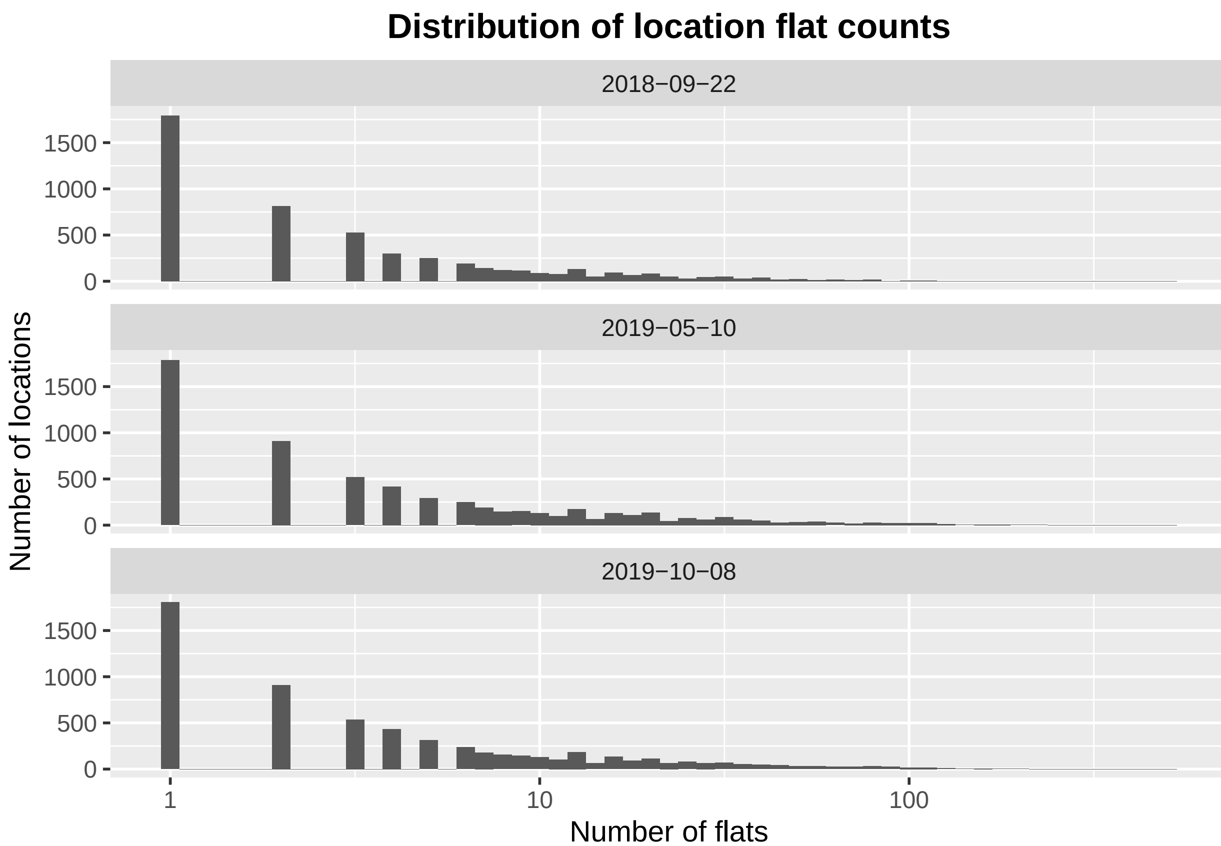


The number of flats listed by location seems to follow the same pattern on all of the three data scraping dates.

It seems like we may ignore the `date` column and consider the joint statistics. To have more solid grounds for that, let us investigate the flat counts per `location` per `date`.

#### Flat counts per `location` per `date`

The distribution of flat counts per `location` indicates that there are a lot of locations with just one flat. However, there are also a few locations with a "large" ($> 100$) number of flat offers.



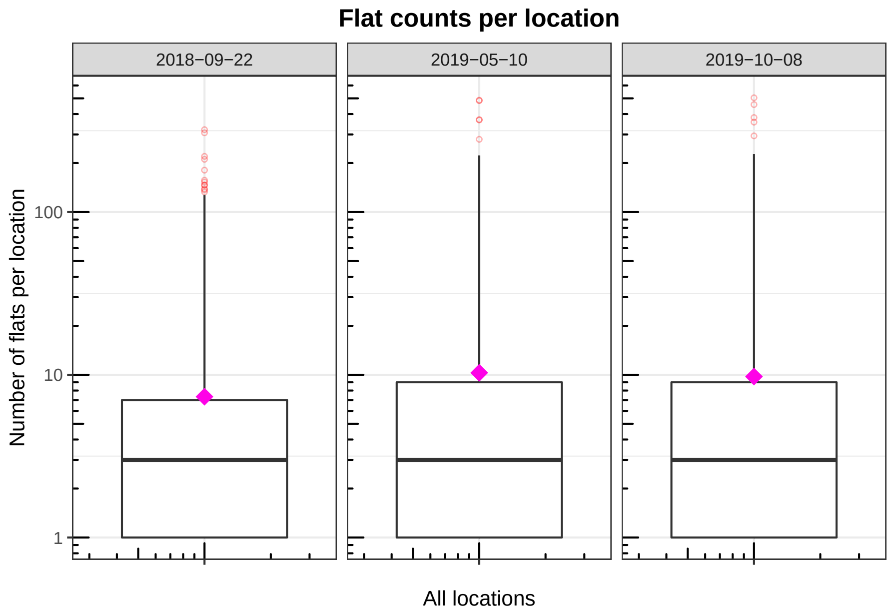

The median numbers of flats per location are equal for all the dates:

```
## # A tibble: 3 x 2
##   date       median
##   <date>      <dbl>
## 1 2018-09-22      3
## 2 2019-05-10      3
## 3 2019-10-08      3
```

The average numbers of flats per location for the last two scrapes are equal, and for the first one is somewhat smaller:

```
## # A tibble: 3 x 2
##   date       average
##   <date>       <dbl>
## 1 2018-09-22       7
## 2 2019-05-10      10
## 3 2019-10-08      10
```

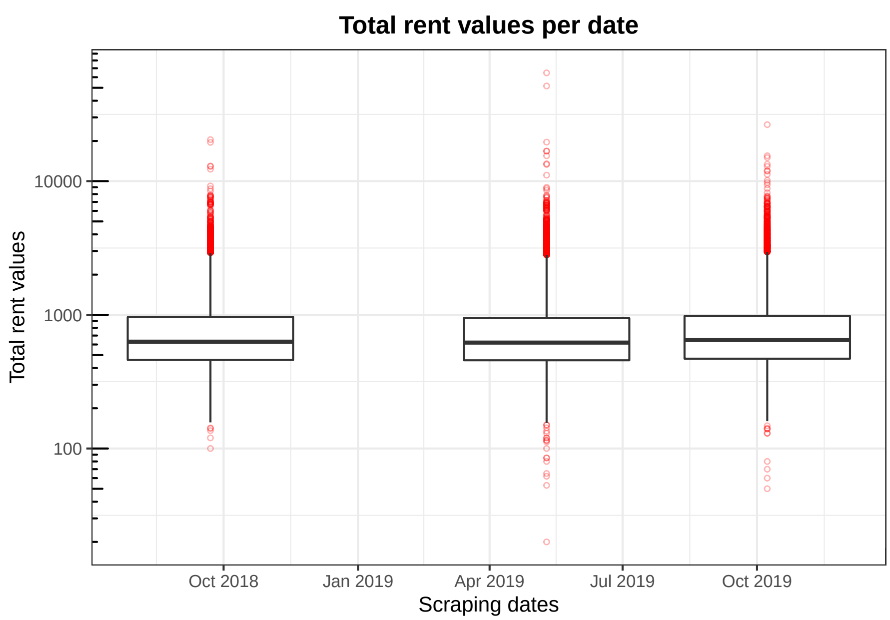
Overall, the number of flats per location seems to be stable from one date to another:



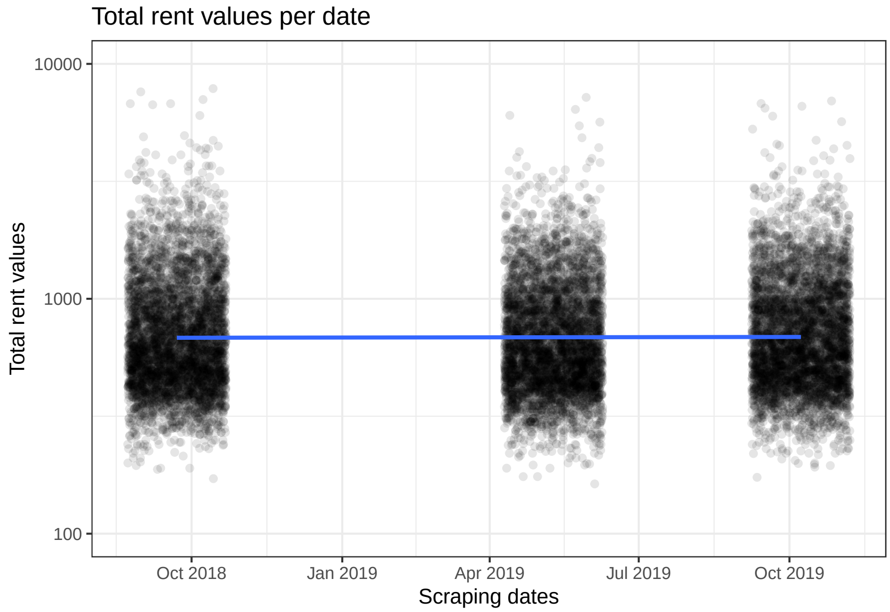

It seems like we may ignore the `date` column and consider the joint statistics. To have more solid grounds for that, let us investigate the average `totalRent` per `date`.

### Average `totalRent` per `date`

Let us plot the `totalRent` per `date`:




As we see the data seems to be distributed similarly, moreover the does not seem to be any trend in average `totalRent` per `date`, as computed with `geom_smooth(method="lm")`:




There does not seem to be any global[7] `timing` effect in `totalRent` related to the scraping date. Please note that in the plot above, for better visualization, we took a 5000 sub-sample of the data and limited the $y$ axis range by $[100, 10^4]$. The latter did not influence the `"lm"` computed trend.

**Conclusion:** We can ignore the `date` column and consider the joint statistics for all the dates.

## Possible `location` effects

Let us consider any possible effects introduced by the location. In essence, the goal of this section is to understand whether:

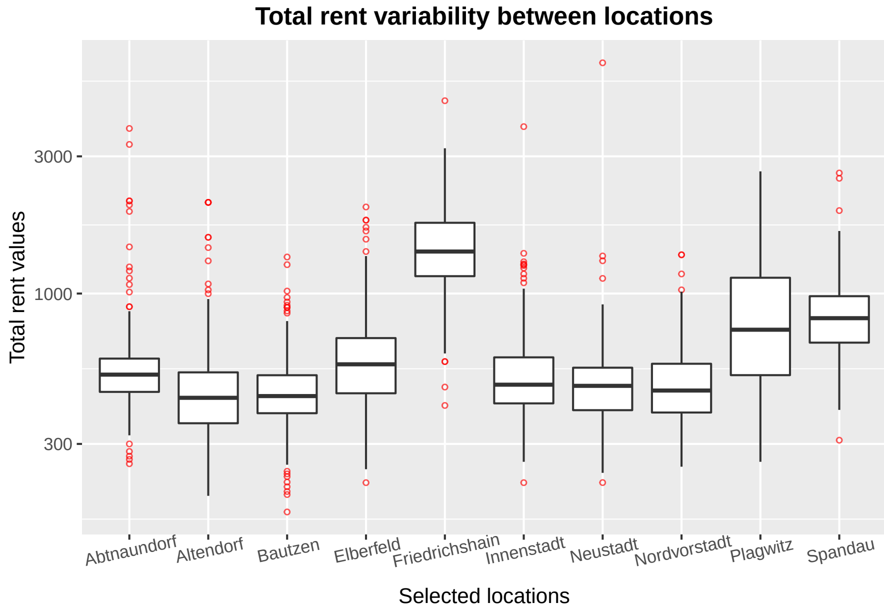
1. The `totalRent` values are `location` dependent
   - Suggests using the `location` column as a predictor
2. The offered flat counts per `location`:
   1. Have influence on the number of `totalRent` value outliers
      - Suggests using Regularization in statistical modeling
   2. Are correlated with the `totalRent` values
      - Facilitates using the `location` column as a predictor

We answer this question based on the analysis of "Average `totalRent` per `location`", "Min/Max `totalRent` per `location` flat count", and "Average `totalRent` per `location` flat count".

### Average `totalRent` per `location`

Let us consider the variability of the `totalRent`, for several randomly chosen locations, with $> 100$ offers:

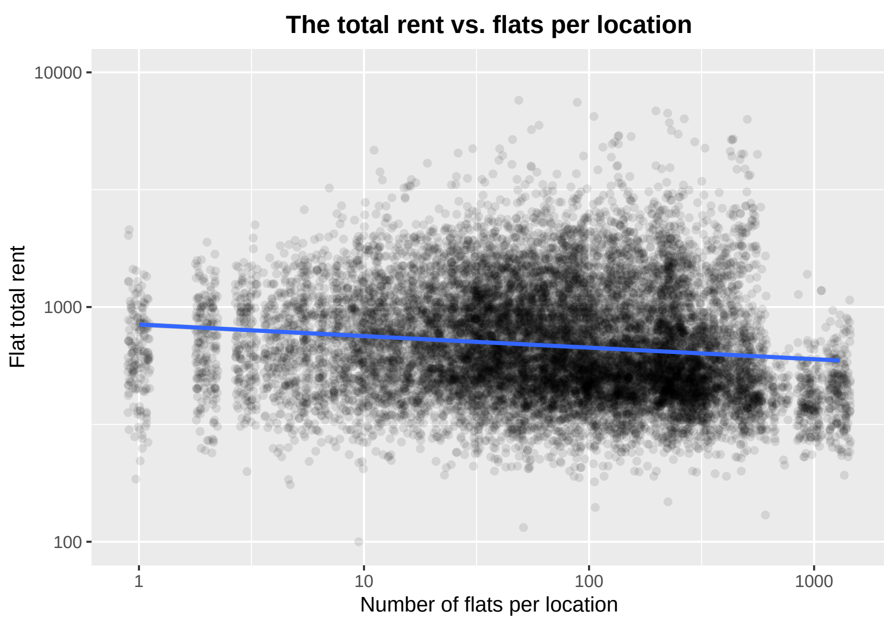
[7] There may be some location specific timing effects, but we do not consider them due to the lack of time.




**Conclusion:** Locations differ quite significantly in their `totalRent` – this is a `location` effect.

### Average `totalRent` per `location` flat count

Let us plot the `totalRent` per location flat count, ordered by the count. Also to show the trend we will use `ggplot` with `geom_smooth(method = 'gam')`.



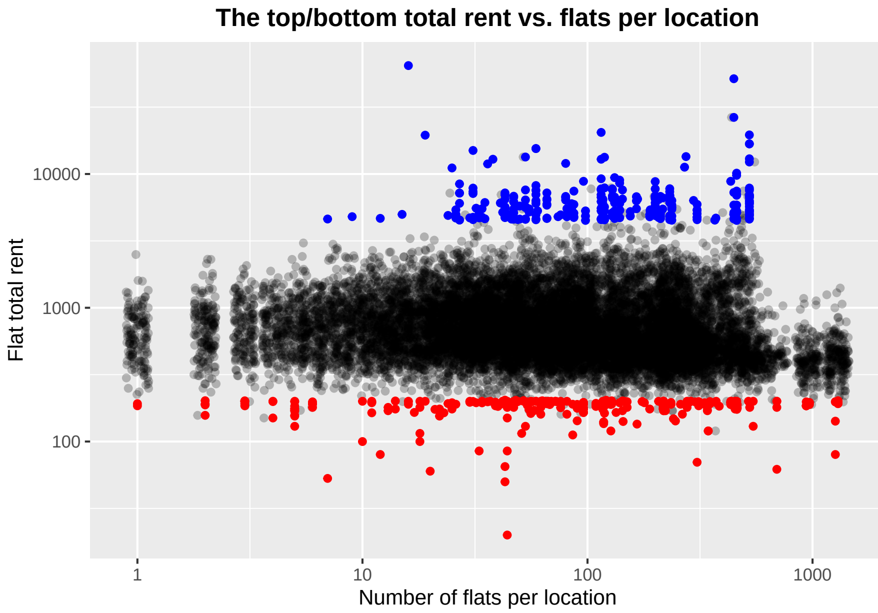

Please note that in the plot above, for better visualization, we took a $1.5 \times 10^4$ sub-sample of the data and limited the $y$ axis range by $[100, 10^4]$. The latter did not influence the `"gam"` computed trend.

**Conclusion:** There is a trend in that the places with more flat offers have on-average lower `totalRent` values - this is a `number of offers` effect.

### Min/Max `totalRent` per `location` flat count

Let us the maximum and minimum `totalRent` values relative to the flat count per `location`. In other words, each `location` has a number of flats offered. If we now combine the flat offers per location into a single totally ordered *flat-offers range*, we can plot how many locations there are with that many orders. Or we can plot the `totalRent` values offered in locations with the same number of flats. The latter will help us to see if the extreme `totalRent` values, like 300 `top` and 300 `bottom` priced flats are more likely to be encountered in locations with lower or larger number of flat offers.

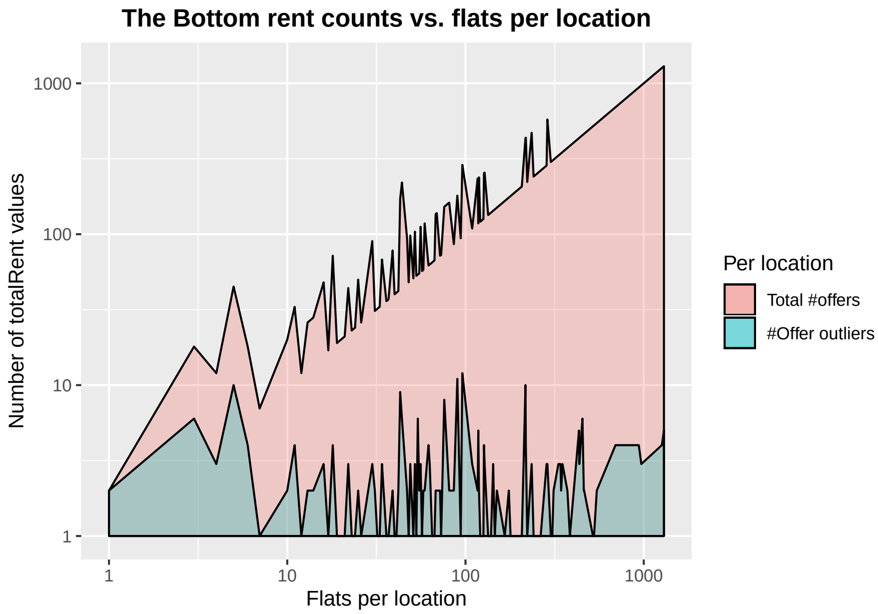
If we indicate `top` and `bottom` priced flats on the plot then we can observe:




that it seems that[12^] the `top/bottom` priced rentals are, almost uniformly, spread all over the *flat-offers range*. [^12]: Please note that in this plot, for better visualization, we took a $1.5 \times 10^4$ sub-sample of the data.

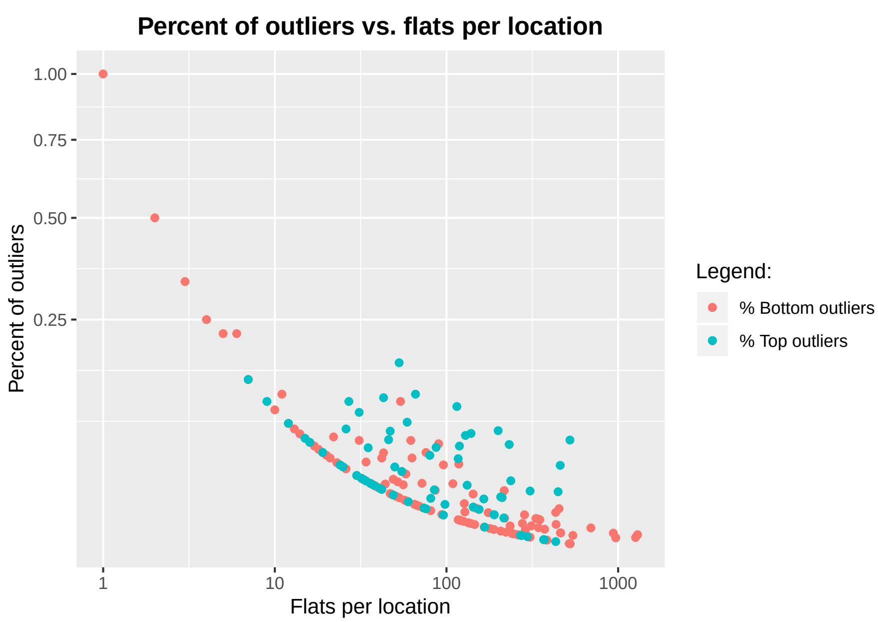
To get a better view on data, considering `top` and total `totalRent` counts versus the *flat-offers range*:

Considering `bottom` and total `totalRent` counts versus the *flat-offers range*:




The total count is computed as the sum of offers in the locations with the same number of offers. Effectively it is equal to `"number of offers in location"` $\times$ `"number of locations with this number of offers"`. This explains almost line-like behavior of the `"Total #offers"` plots. The irregularities are caused by having more than one location with the given number of flat offers (mind the `log10` scale of the $x$ and $y$ axis).

The number of `bottom` outliers is $\approx$ uniform across the *flat-offers range*. The number of `top` outliers looks somewhat biased to be larger for locations with more offers. Let us plot the outliers to total ratio:




The plot above has, even though contains quite some "noise" for 50-200 flats per location, indicates the common trend of having proportionally less outliers in locations with more rental offers.

**Conclusion 1:** It is less likely to have a cheap rentals in locations with large number of offers. **Conclusion 2:** It is less likely to have an expensive rentals in locations with a larger number of offers.

### Predictor's correlation

In this case-study we have selected 21 potential predictors:

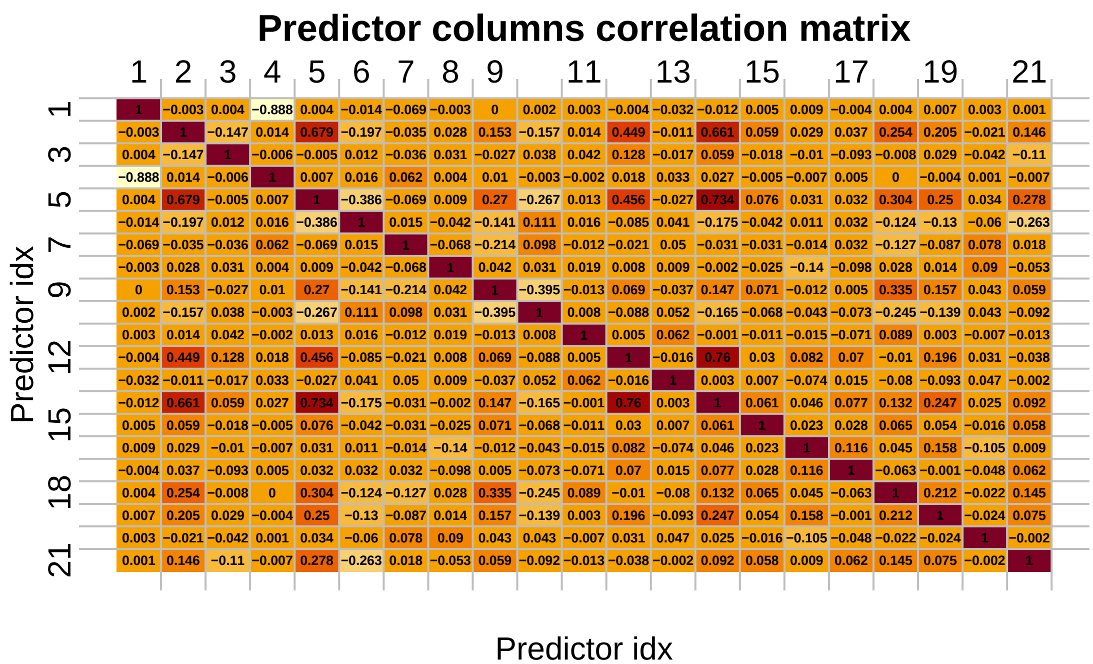
```
##  [1] "hasKitchen"            "heatingType"           "balcony"
##  [4] "lift"                  "garden"                "cellar"
##  [7] "noParkSpaces"          "livingSpace"           "typeOfFlat"
## [10] "noRooms"               "floor"                 "condition"
## [13] "newlyConst"            "interiorQual"          "energyEfficiencyClass"
## [16] "location"              "baseRent"              "electricityBasePrice"
## [19] "heatingCosts"          "serviceCharge"         "date"
```

As we have discussed in the `timing` effects section the `date` column seems to be eligible for exclusion. However, to give this a more formal ground and also to reduce the number of predictors even further let us consider the correlation matrix of the training data columns:

```r
#First create the predictors matrix, from the wrangled data
predictors_mtx <- arog_data$wrangled_data %>%
  select(-totalRent) %>%
  data.matrix(.)

#Next compute the correlations matrix and round off the values
corr_mtx <- predictors_mtx %>%
  cor(.) %>% round(., 3)
```

When visualized, we see that there is quite a few predictors with strong correlation:




For instance, `heatingType` (idx: 2) is strongly correlated with `garden` (idx: 5), `typeOfFlat` (idx: 9), `condition` (idx: 12), `interiorQual` (idx: 14), `electricityBasePrice` (idx: 18), and `date` (idx: 21).

Having highly correlated columns implies that we could use less predictors to build a statistical model without loosing much accuracy.

## Principle Component Analysis

As motivated by the correlation analysis above, here we will use the Principle Component Analysis (PCA) to see if we have a distance preserving transformation of our data that gives us a new feature-space basis in which most of the data variability is explained by fewer predictors:

```r
#Run the PCA analysis
pca_result <- prcomp(predictors_mtx)
#Report the summary
summary(pca_result)
```

```
## Importance of components:
##                              PC1       PC2       PC3      PC4      PC5      PC6
## Standard deviation     2101.8878 459.05602 148.70605 68.76676 41.45316 19.46240
## Proportion of Variance    0.9485   0.04524   0.00475  0.00102  0.00037  0.00008
## Cumulative Proportion     0.9485   0.99371   0.99846  0.99947  0.99984  0.99992
##                             PC7  PC8   PC9  PC10  PC11  PC12  PC13  PC14   PC15
## Standard deviation     16.96046 4.49 3.968 3.475 3.463 2.446 2.242 1.343 0.6259
## Proportion of Variance  0.00006 0.00 0.000 0.000 0.000 0.000 0.000 0.000 0.0000
## Cumulative Proportion   0.99998 1.00 1.000 1.000 1.000 1.000 1.000 1.000 1.0000
##                         PC16   PC17   PC18   PC19   PC20   PC21
## Standard deviation     0.502 0.4409 0.4353 0.3871 0.3665 0.2265
## Proportion of Variance 0.000 0.0000 0.0000 0.0000 0.0000 0.0000
## Cumulative Proportion  1.000 1.0000 1.0000 1.0000 1.0000 1.0000
```

As one can see from the PCA summary above, according to Cumulative Proportion of Variance, it shall suffice to use the first two principle components (`PC1`, and `PC2`) to explain $\approx 99.3\%$ of the data.

## Data analysis summary

In the *"Data analysis"* section we have analyzed our data with respect to possible `timing`, and `location` effects. In addition, to reduce the number of predictors we've performed predictor's correlation and principle component analysis. The findings of this section can be summarized as follows.

### Data effects:

In case we are to build our own statistical model, as opposed using one of the already available via the `cared` package of `R`, we should:

1. The `timing` effect is not confirmed:
    - We can ignore the `date` column and consider the joint statistics for all the dates.
2. The `location` effects are confirmed:
    - The strong correlation of `location` with `totalRent` has to be taken into account[8]

### Dimension reduction:

Due to a high correlation of multiple predictors we can significantly reduce the dimensionality of the feature space. It suffices to use the first two/six principle components to explain $\approx 99.3/99.99\%$ of the data.

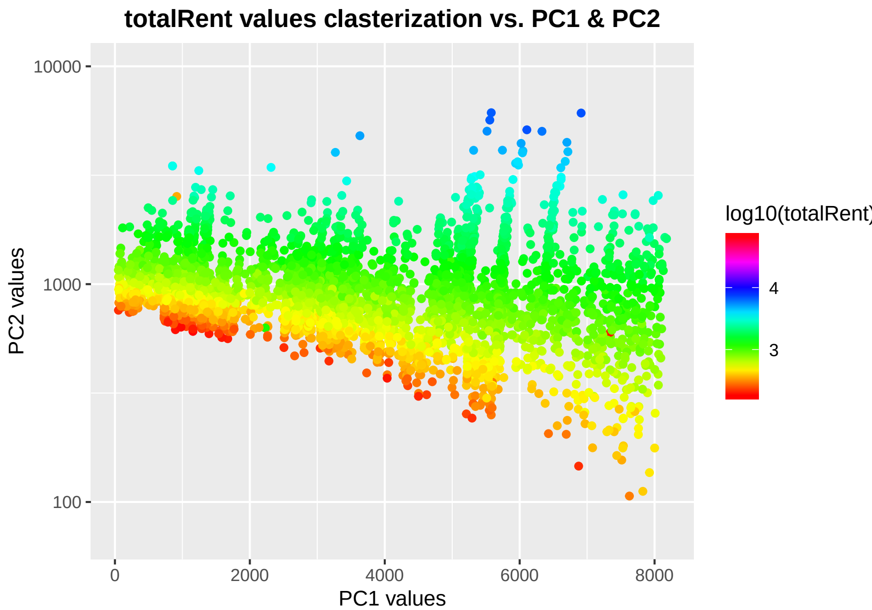
[8] This includes the discovered correlation from the `flat counts` per location as the flat offers per location do not very much with time.

# Modeling approach

To begin, let us recall the project's goal and one important data observation gained:

- **Goal**: Build a prediction model for the `totalRent` values with an `RMSE` score $\leq 50$;
- **Data**: About 99.3% data variability is in the first two principle components;

Let us make a scatter plot of `PC1` vs `PC2` indicating the corresponding `totalRent` values with color:




Here, for better visualization, we:

- Take a 5000 sub-sample of data;
- Limit the `PC2` range by $[70,10^4]$;
- Linearly shift both `PC1` and `PC2` to the positive half-space;
- Use log10 scale for both `PC1` and `PC2`;

From the plot above, there is a clear clasterization of the `totalRent` values. However, its structure is intricate and is not likely to be captured, with desired accuracy, with a simple (linear) regression models. Thus, instead of creating a new Bayesian model, we shall attempt using the existing ones that can be trained via the `caret` package of `R`. Given that the `totalRent` column is numeric, we will consider the following models[9]:

- `"lm"` – Linear Regression
- `"glm"` – Generalized Linear Model
- `"knn"` – k-Nearest Neighbor Classification
- `"Rborist"` – Random Forest[10]
- `"svmLinear"` – Support Vector Machines with Linear Kernel
- `"gamLoess"` – Generalized Additive Model using LOESS

The algorithms above will be run with the default parameters, unless tuning grid parameters are required. Therefore we will consider the entire `modeling` set, as any cross-validation will be done within the `train(.)` method of the `caret` package. Last but not least, we will only use the first two principle components of the available predictors data. Any models that will fail to be trained within (`GLOBAL_METHOD_TIME_OUT_SECONDS` `=` 3600) seconds will be discarded from the further use. Any additional model training parameters, if required, will be presented in the *"Results"* section of this report.

[9]For more information see also the list of caret's Available Models.
[10]This is a high-performance implementation of a random forest model `"rf"`

## Modeling code snippets

To further clarify our approach, below we provide some of the code used for model training and evaluation. First, we present the main utility functions. Next, we show their use in a single *model-train-evaluate* sequence.

As for PCA analysis, we convert our data into a numerical-data matrix using the next function:

```
#-------------------------------------------------------------------
# This function prepares predictors data matrix from a given data set.
#    data_set -- the data set to prepare the  data matrix from
# The performed steps are:
#    1. Remove the totalRent column
#    2. Converting to numeric (data) matrix
# The resulting matrix is returned "as-is"
#-------------------------------------------------------------------
prepare_data_matrix <- function(data_set) {
  #Compute the full matrix from the data set
  data_mtx <- data_set %>% select(-totalRent) %>% data.matrix(.)

  #Return the predictors matrix
  return(data_mtx)
}
```

To get the selected PC predictors matrix from the numerical-data matrix we use:

```
#-------------------------------------------------------------------
# This function takes:
#    pca_result - the PCA analysis results
#    data_mtx - the data matrix
#    num_pc - the number of PC to consider, defaults to NUM_PC_TO_CONSIDER
# and transforms the data_mtx into the PC matrix by:
#    1. Zero-centering the data_mtx columns
#    2. Applying pca_result$rotation matrix
#    3. Selecting num_pc first columns
# The resulting matrix is returned "as-is"
#-------------------------------------------------------------------
prepare_pc_predictors <- function(pca_result, data_mtx, num_pc = NUM_PC_TO_CONSIDER) {
  #Zero-center the columns
  cent_pred_mtx <- sweep(data_mtx, 2, colMeans(data_mtx))

  #Rotate to move to the new basis
  rot_pred_mtx <- cent_pred_mtx %*% pca_result$rotation

  #Only return the required principle component columns
  return(rot_pred_mtx[,1:num_pc])
}
```

The model training is be with the next function, mind the time-out handling:

```
#-------------------------------------------------------------------
# This function allows to train a model specified by the method
#    data_mtx - the numeric-valued predictors data matrix
#    exp_res - the expected results vector for the predictors
#    method - the method to be used
# The training will be done with a time-out defined by the
#   GLOBAL_METHOD_TIME_OUT_SECONDS
# The result is the list with the following elements:
```

```
#    method - the method used
#    start_time - the time the training started
#    success - the success indicating flag
#    end_time - the time the training finished, if success == TRUE
#    fit_model - the fit model, if success == TRUE
#---------------------------------------------------------------------
train_model <- function(data_mtx, exp_res, method, ...) {
  #Remove the fit model global if it exists
  ifrm(fit_model)

  #Initialize new empty training results list
  train_res <- list(method = method)

  #Train the model, with a time-out
  tryCatch({
    train_res <- withTimeout({
      #Record the start time
      train_res <- append(train_res, list(start_time = Sys.time()))

      #Fit the model from data
      fit_model <- train(data_mtx, exp_res, method = method, ...)

      #Record the end time and the result
      train_res <- append(train_res, list(fit_model = fit_model))
    }, timeout = GLOBAL_METHOD_TIME_OUT_SECONDS)
  }, TimeoutException = function(ex) {
    message("Timeout (", GLOBAL_METHOD_TIME_OUT_SECONDS,
            " sec.) while training the '", method, "' model, skipping!")
  })

  #Mark the success flag
  train_res <- append(train_res,
                      list(end_time = Sys.time(),
                           success = !is.null(train_res$fit_model)))

  #Remove the fit model global if it exists
  ifrm(fit_model)

  #Return the result
  return(train_res)
}
```

The model validation is done by the next function:

```
#---------------------------------------------------------------------
# The model evaluation matrix takes the:
#     mdl_res - the modeling results with the fit model to make predictions
#     pc_mtx - the validation set selected pc predictors matrix
#     exp_res - the actual validation set values
# Once the model predicts the values are the RMSE score is computed.
# The result of the function is the list with the following elements:
#    mdl_res - the modeling results
#    pc_mtx  - the selected pc predictors matrix
#    exp_res - the expected (true) values to compare with
```

```r
#    act_res - the actually predicted values
#    rmse    - the RMSE score between exp_res and act_res
#---------------------------------------------------------------------
evaluate_model <- function(mdl_res, pc_mtx, exp_res) {
  if(mdl_res$success) {
    #Predict the raw data based in the fit model and predictor values
    act_res <- predict(mdl_res$fit_model, pc_mtx, type = "raw")

    #Compute the RMSE score
    rmse <- RMSE(act_res, exp_res)
  } else {
    #Training failed so return the NA results
    act_res <- NA
    rmse    <- NA
  }

  #Create the resulting list and return
  return(list(mdl_res = mdl_res, pc_mtx  = pc_mtx,
              exp_res = exp_res, act_res = act_res,
              rmse    = rmse))
```

With the functions above t complete *model-train-evaluate* sequence for a `KNN` look as follows:

```r
#Get the modeling and validation data
model_set <- arog_data$modeling_data
valid_set <- arog_data$validation_data

#Compute the data matrixes for the data sets
model_mtx <- prepare_data_matrix(model_set)
valid_mtx <- prepare_data_matrix(valid_set)

#Perform the PCA analysis on the modeling matrix
pca_result <- prcomp(model_mtx)

#Prepare the predictors for the modeling and validation sets
model_pc_mtx <- prepare_pc_predictors(pca_result, model_mtx)
valid_pc_mtx <- prepare_pc_predictors(pca_result, valid_mtx)

#Train and validate the KNN model
rbt_train_res <- train_model(model_pc_mtx, model_set$totalRent, "knn",
                             tuneGrid = data.frame(k = seq(13, 18, 1)))
rbt_mdl_res <- evaluate_model(rbt_train_res, valid_pc_mtx, valid_set$totalRent)
rbt_mdl_res$rmse
```

For more details, see the supplied `apartment_rental_project.R` modeling script.

# Results

## Used hardware

The experiments were run on the following hardware at hand:

```
Model Name: MacBook Pro
```

```
Model Identifier: MacBookPro11,4
Processor Name: Intel Core i7
Processor Speed: 2,2 GHz
Number of Processors: 1
Total Number of Cores: 4
L2 Cache (per Core): 256 KB
L3 Cache: 6 MB
Memory: 16 GB
Boot ROM Version: 194.0.0.0.0
SMC Version (system): 2.29f24
```

### Modeling results

#### Linear Regression – `lm`

#### Generalized Linear Model – `glm`

#### K-Nearest Neighbors – `knn`

#### Random Forest – `Rborist`

#### Support Vector Machines – `svmLinear`

#### Generalized Additive Model – `gamLoess`

## Results summary

## Conclusions

### Future work

Check for introducing any bias by data wrangling. Use model ensembles. Use more principle components. Factor the totalRent and use e.g. `knn3`.

## Appendix A: The complete list of data set columns

Hereby we present the complete list of columns from the original 'Apartment rental offers in Germany' dataset:

```
##  [1] "regio1"                    "serviceCharge"
##  [3] "heatingType"               "telekomTvOffer"
##  [5] "telekomHybridUploadSpeed" "newlyConst"
##  [7] "balcony"                   "electricityBasePrice"
##  [9] "picturecount"              "pricetrend"
## [11] "telekomUploadSpeed"        "totalRent"
## [13] "yearConstructed"           "electricityKwhPrice"
## [15] "scoutId"                   "noParkSpaces"
## [17] "firingTypes"               "hasKitchen"
## [19] "geo_bln"                   "cellar"
## [21] "yearConstructedRange"      "baseRent"
```

```
## [23] "houseNumber"              "livingSpace"
## [25] "geo_krs"                  "condition"
## [27] "interiorQual"             "petsAllowed"
## [29] "streetPlain"              "lift"
## [31] "baseRentRange"            "typeOfFlat"
## [33] "geo_plz"                  "noRooms"
## [35] "thermalChar"              "floor"
## [37] "numberOfFloors"           "noRoomsRange"
## [39] "garden"                   "livingSpaceRange"
## [41] "regio2"                   "regio3"
## [43] "description"              "facilities"
## [45] "heatingCosts"             "energyEfficiencyClass"
## [47] "lastRefurbish"            "date"
```

# Appendix B: Data set column descriptions

Here is the list of the initially considered data set columns with the descriptions thereof:

1. `hasKitchen` – has a kitchen
2. `balcony` – does the object have a balcony
3. `cellar` – has a cellar
4. `lift` – is elevator available
5. `floor` – which floor is the flat on
6. `garden` – has a garden
7. `noParkSpaces` – number of parking spaces
8. `livingSpace` – living space in sqm
9. `condition` – condition of the flat
10. `interiorQual` – interior quality
11. `regio1` – Bundesland
12. `regio2` - District or Kreis, same as geo krs
13. `regio3` – City/town
14. `noRooms` – number of rooms
15. `numberOfFloors` – number of floors in the building
16. `typeOfFlat` – type of flat
17. `yearConstructed` – construction year
18. `newlyConst` – is the building newly constructed
19. `heatingType` – Type of heating
20. `energyEfficiencyClass` – energy efficiency class
21. `heatingCosts` – monthly heating costs in €
22. `serviceCharge` – auxiliary costs such as electricity or Internet in €
23. `electricityBasePrice` – monthly base price for electricity in €

24. `baseRent` – base rent without electricity and heating
25. `totalRent` – total rent (usually a sum of base rent, service charge and heating cost)
26. `date` – time of scraping