

NYILATKOZAT

Név: Pham Viet Hung

ELTE Természettudományi Kar, szak: BSc Matematika

PGRVWP'azonosító: YSOUT0

Szakdolgozat címe: A machine learning project: New York City Airbnb price prediction

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2020.12.07.



a hallgató aláírása



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

BACHELOR OF SCIENCE THESIS

A machine learning project: New York City Airbnb price prediction

Author:
Viet Hung Pham

Supervisor:
Tamás Prőhle

*A thesis submitted in fulfilment of the requirements
for the degree of Bachelor of Science in Mathematics*

in the

Department of Probability Theory and Statistics
Institute of Mathematics

December 6, 2020

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

*Abstract*Faculty of Natural Science
Institute of Mathematics

Bachelor of Science in Mathematics

A machine learning project: New York City Airbnb price prediction

by Viet Hung Pham

Machine learning gives computers the ability to learn without being explicitly programmed. It uses algorithms to find patterns in massive amounts of data to make accurate predictions when new data fed or discover new relationships between features. Nowadays, it is a hugely expanding field which is used in all segments of life, ranging from predicting patient medical conditions to face-recognition technologies. In this thesis, the object is to show an example of an end-to-end ML project. I would like to present some mathematics behind those machine learning algorithms and findings of predicting the prices of New York Airbnb's for one night. The data set comes from Kaggle:

<https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data>.

Firstly, I analysed and visualised the data to get a better understanding of the Airbnb market in New York. Then I did some data preprocessing, which includes data cleaning and imputations, and feature engineering to adjust our data set to better perform on those algorithms. As it is about predicting prices, it is considered as a regression task, which is a subclass of supervised learning. After separating data set into training and test set, I trained the data on Linear Regressions, Support Vector Machine, and various Ensemble learning algorithms while hyperparameter tuning them. The next step is to evaluate those models by using cross-validation. Finally, the best model is chosen to test on the unseen test set data. It turns out that Ensemble and Support Vector Machine learning algorithms are the best-performing ones. In the end, I would like to contemplate about how my project could be further developed and how it may be implemented in a real-world scenario.

Acknowledgements

First and foremost, I would like to pronounce my greatest thanks and gratitude to Tamás Prőhle for his guidance and continuous help throughout the thesis process. I am also grateful to Olivér Kiss, my colleague at Central European University MicroData Group, for his invaluable insights. I also owe tremendous gratitude to my parents. My late father was the one who introduced me to the world of mathematics at a very early age and encouraged me to pursue it. I cannot be more grateful to my mother, who always supports me in both good and bad times. I am also thankful for working together in the previous semester with Dóra Kinorányi and Krisztina Hegyi on a machine learning project for the 'Data Mining' course from which my thesis is inspired. Finally, I would like to thank my friends for their constant support throughout my bachelor degree.

Contents

| | |
|--|------------|
| Abstract | iii |
| Acknowledgements | v |
| 1 Machine Learning Project Checklist | 1 |
| 1.1 Overview and defining the problem | 1 |
| 1.2 Getting the data | 1 |
| 1.3 Exploratory Data Analysis | 2 |
| 1.4 Preparing the data for ML algorithms to uncover patterns | 2 |
| 1.5 Discovering and selecting the best performing models | 3 |
| 1.6 Fine-tuning the selected models | 3 |
| 1.7 Further improvement opportunities | 3 |
| 2 Overview of the Project | 5 |
| 2.1 Framework | 5 |
| 2.2 Performance Measurement | 6 |
| 2.3 ML Questions | 6 |
| 2.4 ML Hypothesis | 7 |
| 3 The Data | 9 |
| 3.1 Overview of the data | 9 |
| 3.2 Observations from histograms | 10 |
| 4 Exploratory Data Analysis | 13 |
| 4.1 Training, dev, and test set | 13 |
| 4.2 The analysis of the target variable | 13 |
| 4.3 The analysis of feature variables | 14 |
| 4.4 Outliers | 16 |
| 4.5 Tinkering with features | 18 |
| 4.6 Correlation | 18 |
| 4.7 Some miscellaneous analysis | 19 |
| 5 Data Preprocessing | 21 |
| 5.1 Data Cleaning | 21 |
| 5.2 Transformation Pipeline and Feature Scaling | 22 |
| 5.3 Yeo-Johnson transformation | 22 |
| 5.4 Analysis of the transformed data | 23 |
| 6 Model Training and Testing | 25 |
| 6.1 Dimensionality Reduction | 25 |
| 6.1.1 Feature Selection | 25 |
| 6.1.2 Factor Analysis | 26 |
| Principal Component Analysis | 26 |
| Multiple Correspondance Analysis | 27 |

| | | |
|----------|---|-----------|
| | Factor Analysis of Mixed Data | 27 |
| 6.2 | Linear Regression | 29 |
| 6.2.1 | Vanilla Linear Regression | 29 |
| 6.3 | Regularised Linear Regression | 30 |
| 6.4 | Support Vector Machine | 31 |
| 6.5 | Gradient Boosting – XGBoost | 31 |
| 6.6 | Bagging and Pasting | 31 |
| 6.7 | Voting Regression | 31 |
| 6.8 | Results of the Model Training | 32 |
| 6.9 | Final Model and Testing | 33 |
| 7 | Further Development Opportunities | 35 |
| 7.1 | Other Learning Algorithms | 35 |
| 7.2 | More features | 35 |
| 7.3 | Implementation in real life | 35 |
| A | Technical Details | 37 |
| A.1 | Linear Regression | 37 |
| A.1.1 | Gradient Descent | 37 |
| | Batch Gradient Descent | 37 |
| | Stochastic Gradient Descent | 38 |
| A.1.2 | Normal Equation | 38 |
| A.1.3 | Probabilistic interpretation | 40 |
| A.1.4 | Shrinkage Methods | 41 |
| A.2 | Kernel Methods | 43 |
| A.2.1 | Feature Maps | 43 |
| A.2.2 | Kernel Trick | 43 |
| | Some well-known kernels | 44 |
| | Necessary and sufficient conditions for kernel validity | 45 |
| A.3 | Support Vector Regression | 45 |
| B | Dimensionality Reduction | 47 |
| B.1 | Feature Selection | 47 |
| B.1.1 | Filter Methods | 47 |
| | Pearson’s Correlation | 47 |
| B.1.2 | Wrapper methods | 47 |
| | The algorithm of the SBFS | 48 |
| B.1.3 | Embedded methods | 48 |
| B.2 | Factor Analysis | 49 |
| B.2.1 | PCA | 49 |
| B.2.2 | Direction with Maximal Variance | 50 |
| B.2.3 | M-dimensional Subspace with Maximal Variance | 51 |
| C | Python Codes | 55 |
| C.1 | Codes from Section 5.2 | 55 |
| C.2 | Codes from Section 6.1 | 57 |
| C.3 | Codes from Section 6.8 | 57 |
| C.4 | Codes from Section 6.9 | 58 |

List of Figures

| | | |
|-----|--|----|
| 1.1 | Machine Learning Project Roadmap | 4 |
| 2.1 | Supervised Learning | 5 |
| 2.2 | Online Learning | 6 |
| 3.1 | Missing Values | 10 |
| 3.2 | Yeo-Johnson transformed <i>price</i> | 10 |
| 3.3 | Histograms | 11 |
| 4.1 | Y-J prices and neighbourhood groups' distribution | 14 |
| 4.2 | Histograms of Neighbourhood Group and Room Types | 15 |
| 4.3 | Room Type based on Neighbourhood Group | 15 |
| 4.4 | Room Types' distribution over NYC | 16 |
| 4.5 | Neighbourhood group and its availability | 16 |
| 4.6 | Boxplots of the numerical variables | 17 |
| 4.7 | Correlation Matrix Heatmap | 19 |
| 4.8 | Top 5 neighbourhoods | 20 |
| 5.1 | Correlation Matrix Heatmap on the Y-J transformed data | 24 |
| 6.1 | FAMD illustration | 28 |
| A.1 | Ridge Regression | 42 |
| A.2 | Lasso | 42 |
| B.1 | PCA illustration | 50 |

List of Tables

| | | |
|-----|--|----|
| 3.1 | Features, their contents and types | 9 |
| 4.1 | Prices | 14 |
| 4.2 | Neighbourhood group | 14 |
| 4.3 | Room types | 14 |
| 4.4 | Feature variables' correlations with <i>price</i> | 18 |
| 5.1 | Data Cleaning (code) | 21 |
| 5.2 | Data transformation and feature scaling (code) | 22 |
| 6.1 | Filter vs. Wrapper vs. Embedded methods | 26 |
| 6.2 | Training and CV <i>RMSE</i> errors of the learning models | 33 |
| B.1 | Filter methods | 47 |
| C.1 | Numerical Variable Transformer (code) | 55 |
| C.2 | Target Variable Transformer (code) | 56 |
| C.3 | Concatenation of numerical and categorical features (code) | 56 |
| C.4 | Factor Analysis of Mixed Data implementation (code) | 57 |
| C.5 | SVR with Gaussian Kernel Training (code) | 57 |
| C.6 | SVR with Gaussian Kernel Cross-Validation (code) | 58 |
| C.7 | SVR with Gaussian Kernel Training (code) | 58 |

List of Abbreviations

| | |
|-----------|--|
| Dev Set | Development set |
| Train Set | Training set |
| ML | Machine Learning |
| MSE | Mean Square Error |
| RMSE | Root Mean Square Error |
| SVC | Support Vector Classification |
| SVM | Support Vector Machine |
| SVR | Support Vector Regression |
| RBF | Radial Basis Function |
| Y-J | Yeo-Johnson |
| PCA | Principal Component Analysis |
| MCA | Multiple Correspondance Analysis |
| FAMD | Factor Analysis of Mixed Data |
| i.i.d | Independent Identically Distributed |
| LMS | Least Mean Squares |
| OLS | Ordinary Least Squares |
| SVD | Singular Value Decomposition |
| GD | Gradient Descent |
| BGD | Batch Gradient Descent |
| SGD | Stochastic Gradient Descent |
| Lasso | Least Absolute Shrinkage and Selection Operator |
| XGBoost | EXtreme Gradient B oosting |
| GBRT | Gradient B oosted R egression T ree |
| CV | Cross-Validation |

Dedicated to my late father...

Chapter 1

Machine Learning Project Checklist

My project is following mostly the ML project checklist proposed by Geron (2019) that I wish to present here. The eight main themes of an ML project are:

1. Overviewing and defining the problem.
2. Getting the data.
3. Gaining insights from the data set.
4. Preparing the data for ML algorithms to uncover patterns.
5. Discovering and selecting the best performing models.
6. Fine-tuning the selected models.
7. Further improvement opportunities.

1.1 Overview and defining the problem

1. Defining the (business) objective.
2. How should the problem be framed?
3. How should the performance be measured?
4. Listing (all) the questions regarding the projects.
5. Listing (all) the assumptions that have been made so far.

1.2 Getting the data

1. Listing all the data sources where they are downloaded.
2. How much space will the data take?
3. Putting the data into a format with which it can be easily worked without changing the data itself.

1.3 Exploratory Data Analysis

1. Sampling a test set, putting it aside for the later purpose of evaluating our best model(s).
2. Creating a copy of the data for exploration.
3. Creating a Jupyter/Jupyterlab/Google Colab notebook to keep track of data exploration.
4. Scrutinising characteristics of variables:
 - ▷ Name
 - ▷ Type (integer/float, categorical, text, boolean, etc.)
 - ▷ Percentage of missing values
 - ▷ Noisiness and type of noise such as outliers
 - ▷ Usefulness for the task
 - ▷ Type of distributions
5. Identifying the target attribute(s).
6. Visualising the data.
7. Examining correlations between attributes.
8. Identifying the most promising transformations that could be used on the data.

1.4 Preparing the data for ML algorithms to uncover patterns

1. Data cleaning:
 - ▷ Fixing or removing outliers (optional).
 - ▷ Filling in missing values (e.g., with zero, mean, median...) or drop their rows (or columns).
2. Feature selection (optional):
 - ▷ Dropping the feature(s) which do not provide useful information for the task by using algorithms such as PCA, Sequential Backward Selection, etc...
3. Feature engineering (if needed):
 - ▷ Discretising continuous features.
 - ▷ Decomposing features (e.g., categorical, date/time, etc.).
 - ▷ Adding promising transformations for features.
 - ▷ Aggregating variables into promising new features.
4. Feature scaling:
 - ▷ Using transformation(s) for the ML algorithm to better work on the data such as standardisation or normalisation.

It is highly recommended to keep the original data set intact and only work on the copied version. It is also advised to modularise all the data transformation into functions because of the following reasons:

- More comfortable to develop and debug new program features introduced to our program.
- When getting a new or updated data set, it is easier to prepare them and apply those transformations in the future.
- Cleaning and preparing the test set.
- Cleaning and preparing new data instances once a solution is live.

1.5 Discovering and selecting the best performing models

1. Training many quick-and-dirty ML models from different categories such as Linear Regression, Support Vector Machine, Ensemble models, and so on.
2. Measuring and comparing their performance:
 - ▷ Training the ML models on the training set (train set) and either evaluating on the **development set** (dev set) or using **k-fold cross-validation** and compute the mean and standard deviation of the performance measure on the k folds.
3. Analysing the types of errors the models make.
4. Shortlist the top-performing models, preferring ones that make different types of errors.

1.6 Fine-tuning the selected models

1. Fine-tune the hyperparameters using cross-validation:
 - ▷ Using Grid Search Cross-Validation when there are very few hyperparameter values.
 - ▷ Otherwise, using Random Search Cross-Validation makes sense.
2. After finishing with carrying out the steps above, select the final model based on the performance and run it on the test set to estimate the generalisation error.

1.7 Further improvement opportunities

- Further comments on the project.
- What other ML algorithms and feature could be used and collected?
- Improving performances and alternatives.
- Real-world application opportunities.

The following flowchart sums up the whole ML project roadmap.

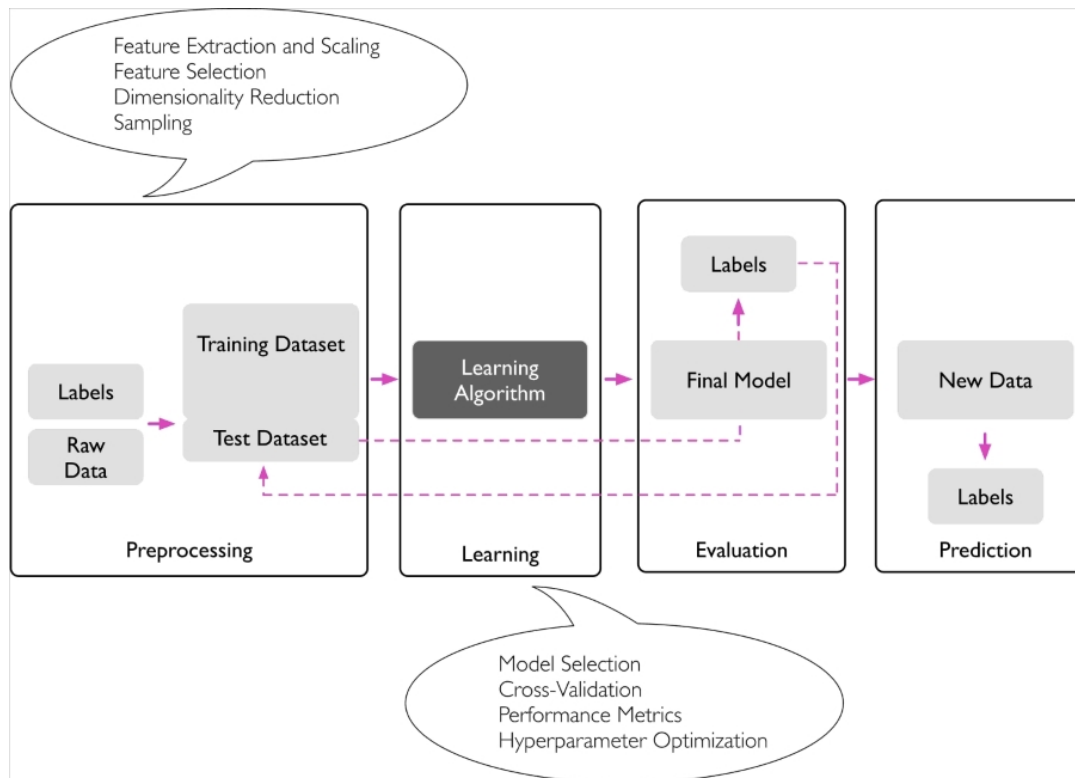


FIGURE 1.1: ML Project Roadmap (Raschka, Mirjalili, 2019)

Chapter 2

Overview of the Project

Our purpose with this project is to predict price/night (from now on merely price) of New York Airbnb accommodations based on its 2019 data. We aim to develop a model that can effectively and efficiently do this job.

2.1 Framework

Our task is a supervised (Figure 2.1), regression learning task because having a price tag label for each instance. Price (our target variable) is continuous.

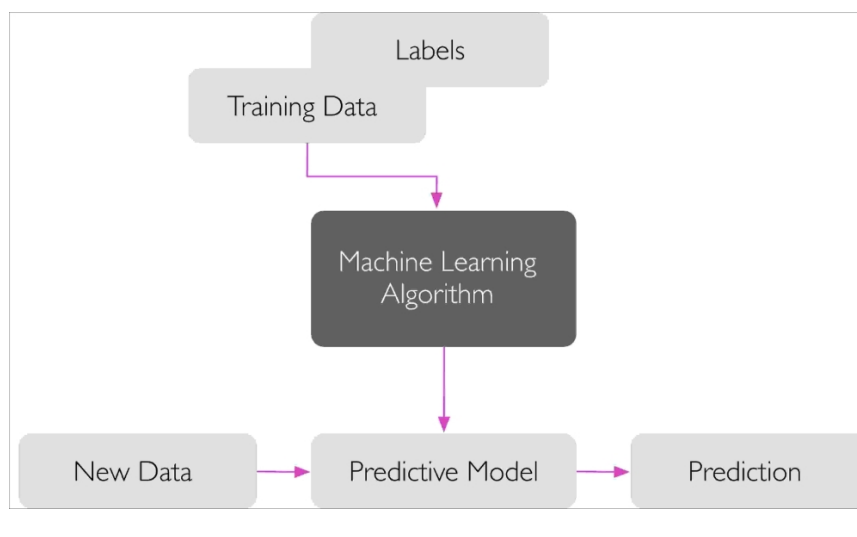


FIGURE 2.1: Supervised Learning (Raschka, Mirjalili, 2019)

Since it is small data (approximately 7MB), batch learning is going to be applied. Batch learning means that the whole data set will be fed into system without learning anymore in contrast to online learning which a technique used to situations when data set are massive and cannot fit in a computer's main memory. Online learning algorithms operate in a way that it loads part of the data, runs training on the 'partial data', and does this again until all data is trained. When new data comes in, it will keep learning.

Here is the figure summing up the online-learning algorithm.

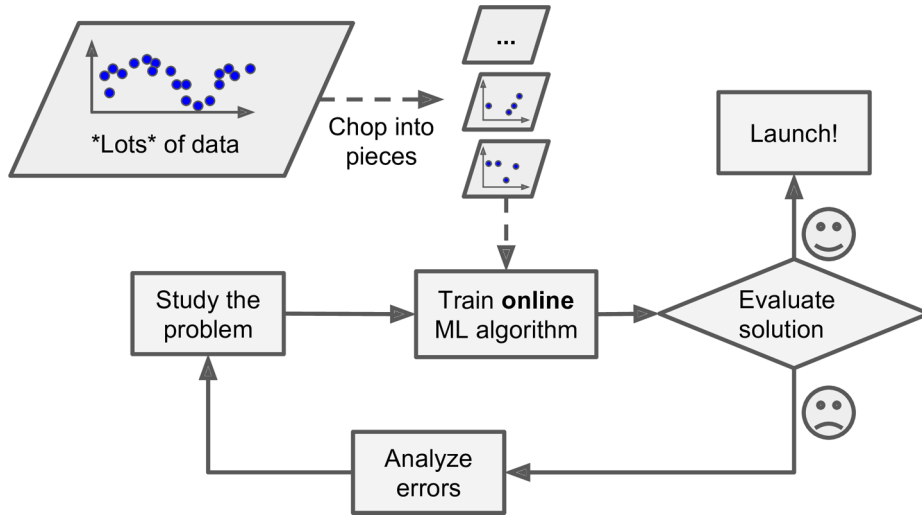


FIGURE 2.2: Big data set handled by online learning (Geron, 2019)

Technical Details

In this ML project, codes were implemented in Python 3 was used heavily with the Sklearn (Pedregosa et al., 2011), Pandas (McKinney et al., 2010), NumPy (Harris et al., 2020), and Matplotlib/Seaborn (Hunter, 2017) packages.

2.2 Performance Measurement

Measuring the performance of a regression learning task is often done by either of the following cost functions:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \quad (2.1)$$

$$RMSE(\mathbf{X}, h_{\theta}) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2} \quad , \quad (2.2)$$

where $(x^{(i)}, y^{(i)})$ is the i^{th} instance of a training set and the corresponding i^{th} target variable. The θ is the parameter of either of the cost functions $J(\theta)$ or $RMSE(\mathbf{X}, h_{\theta})$. h_{θ} is the hypothesis function. From now on, instead of h_{θ} , the short-handed h will be used. $J(\theta)$ and $RMSE(\mathbf{X}, h_{\theta})$ can be regarded as equivalent cost functions because the '1/2' used in the former formula are just there for the sake of taking a derivative easier. Later on, $J(\theta)$ is going to be used in the derivation of gradient descent algorithms, but the $RMSE$ is applied for model evaluation in our Python program.

2.3 ML Questions

- What kind of learning algorithms will be used?

- Minimalise or maximise our chosen cost function?
- How should we dissect our data set for the subsets being more representative?
- How should we clean or transform the data so that the ML algorithms work well on it?
- How should we avoid overfitting/underfitting?

These question will be answered shortly.

2.4 ML Hypothesis

- *price* variable does not follow the normal distribution.
- Multicollinearity occurs among some feature variables.
- There are a considerable amount of outliers.

Chapter 3

The Data

In this part, the general overview and observations of the data are going to be discussed. The aim is to get a sound grasp of our data set before delving into a more in-depth analysis.

3.1 Overview of the data

The New York City Airbnb data is from the Kaggle:

<https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data>.

- 48895 instances with 16 variables, including the target variables.
- Two attributes are text-based, one is date, three are categorical, remaining 10 are numbers.
- The following table sums up the general knowledge about the variables:

| Features | Contents of the features | Types |
|--------------------------------|---------------------------|-------------|
| id | a unique id | integer |
| name | name of the accommodation | text |
| host_id | unique id of the host | integer |
| host_name | name of the host | text |
| neighbourhood_group | neighbourhood group | categorical |
| neighbourhood | neighbourhood | categorical |
| latitude | latitude | float |
| longitude | longitude | float |
| room_type | room type | categorical |
| price | price in dollar | float |
| minimum_nights | min # of nights spent | integer |
| number_of_reviews | # of reviews | integer |
| last_review | date of the last review | date |
| reviews_per_month | reviews per month | integer |
| calculated_host_listings_count | # of listings of a host | integer |
| availability_365 | # of days available | integer |

TABLE 3.1: Features, their contents and types

- There are some missing values in our data set: 16 from *name*, 21 from *neighbourhood_group*, 10052 from *last_review* and *reviews_per_month*. Assume that the data are *missing completely at random* (MCAR). The following table summarises this in ascending order:

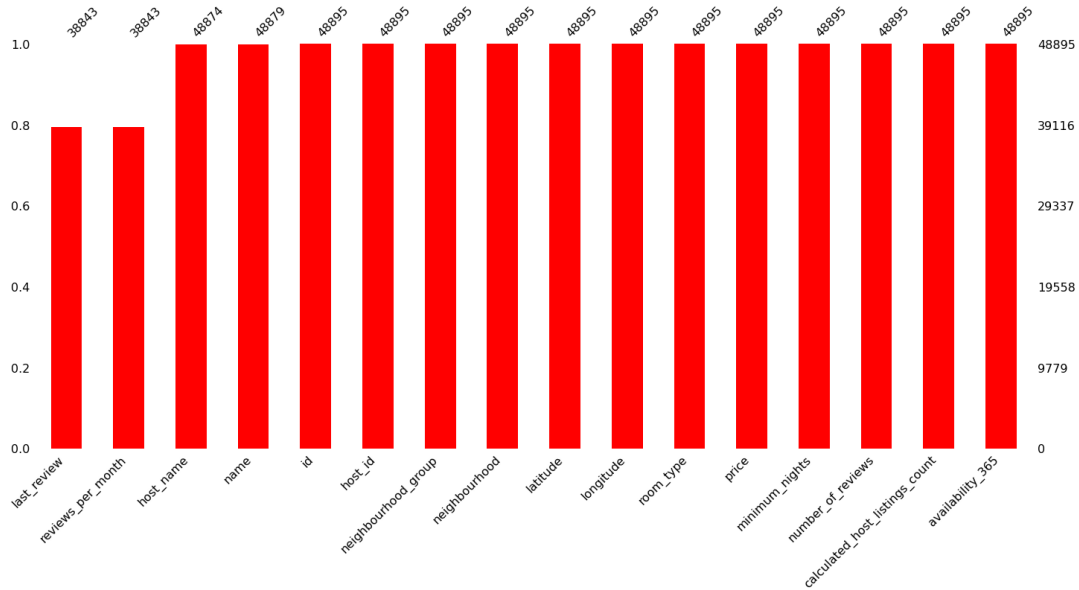
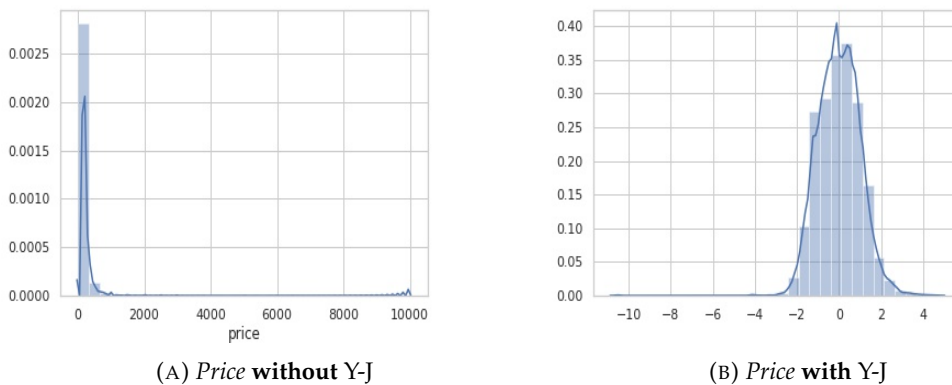


FIGURE 3.1: Missing values of the variables

3.2 Observations from histograms

Histograms used for explaining observations are from Figure 3.2 and Figure 3.3:

- Most histograms tend to skew to the left. That is they present heavy-tailed distributions.
- The range of values of some of the features is wide.
- To this end, we need to transform our data set so that those variables will be standardised and resemble something like the normal distribution. It is an essential step for the ML algorithms to better work on the data.
- *Latitude* and *Longitude* follow the normal distribution. It means that most of the accommodations concentrate in central New York City.
- The transformation used will be Yeo-Johnson (Y-J) and be discussed later on. Look at the example of the power of the Y-J transformation through the *price* variable.

FIGURE 3.2: Y-J transformation on the *Price* variable

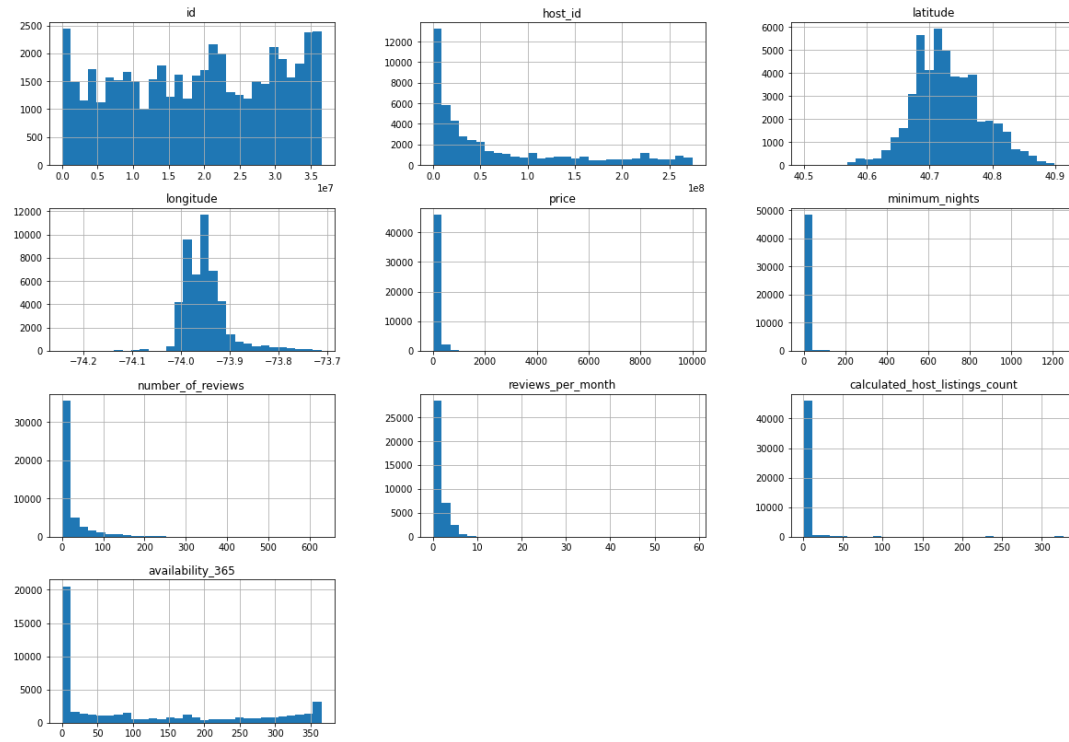


FIGURE 3.3: Histograms of numerical variables

Chapter 4

Exploratory Data Analysis

In this chapter, splitting of the original data set is going to occur based on stratified sampling. From Section 4.2 - 4.7 details of variables is uncovered.

4.1 Training, dev, and test set

- 64% of the data set will be used for training, 16% is distributed to the dev set and 20% for running the best performing model on the test set. It is achieved by dissecting first 80% - 20% in favour of train + dev set, and from that 80%, we subdivide it to 80%-20% in favour of the train set.
- The reason why the dev set is needed is that we can evaluate our trained model on the dev set. If further adjustment and optimisation are necessary, we do it on the dev set rather than the test set. For our purpose, it is mainly useful when training quick-and-dirty ML models.
- The alternative, which is used in the project, is to perform k-fold cross-validation, which will be discussed in Section 6.8. Generally, it is preferred over using the dev set for validation. However, cross-validation is computationally a much more expensive process.
- The test set has the purpose of running the best final fine-tuned model on it to see the generalisation error of our model on the unseen data.
- It is vital that all of the subsets are representative of the whole data set, so that is why stratified sampling is used when dissecting the data set. In other words, data points are allocated to their respective homogeneous subgroups called strata, and the appropriate number of data entries are sampled from each stratum. In the end, representativeness can be ensured.
- Stratified sampling is carried out according to *neighbourhood_group* because the location of a property greatly influence the price people can rent it out. These five strata are Brooklyn, Manhattan, Queens, Staten Island, Bronx.

After the split, there are 39116 instances in the train + dev set with 13, 14, 8062, 8062 missing values from *name*, *host_name*, *last_review*, *reviews_per_month*, respectively. Creating a copy of this set is essential because, in Section 4.5, there will be some tinkering with the variables.

4.2 The analysis of the target variable

The following table shows the average, minimum and maximum prices for a night:

| | |
|---------|----------|
| Average | \$152.72 |
| Median | \$107.00 |
| Minimum | \$0 |
| Maximum | \$10000 |

TABLE 4.1: Prices

An interesting thing could be observed from the table is that there are \$0 Airbnb properties. Analysing together with Figure 3.2 and 4.6, it could be noticed that the range where prices fluctuate is comprehensive. The average, median, maximum values are \$152.72, \$107.00, \$10000, respectively. The two figures below show the distribution of Y-J prices and neighbourhood groups over the map of New York City. It can be clearly seen that in general, the most expensive accommodations can be found in Manhattan and the cheapest ones are in the Bronx.

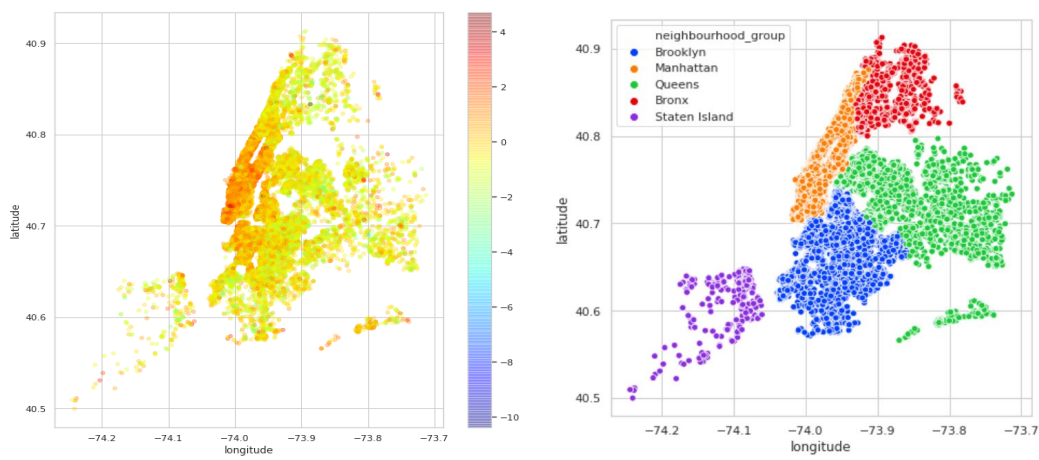


FIGURE 4.1: Y-J transformed prices and neighbourhood groups' distribution in NYC

4.3 The analysis of feature variables

The following tables and figures represent the possible values for *neighbourhood_group*, *room_type* and their number of occurrences. The most number of properties can be found in Manhattan followed by Brooklyn, Queens, Bronx and Staten Island. The last three together have significantly lower proportions of all properties in New York City. Regarding the room type, entire home and private home dominate the New York City Airbnb offer with a tiny number of shared rooms can be rented.

| Neighbourhood group | Quantity |
|---------------------|----------|
| Manhattan | 21661 |
| Brooklyn | 20104 |
| Queens | 5666 |
| Bronx | 1091 |
| Staten Island | 373 |

TABLE 4.2: Neighbourhood group

| Room Types | Quantity |
|-----------------|----------|
| Entire home/apt | 25409 |
| Private room | 22326 |
| Shared room | 1160 |

TABLE 4.3: Room types

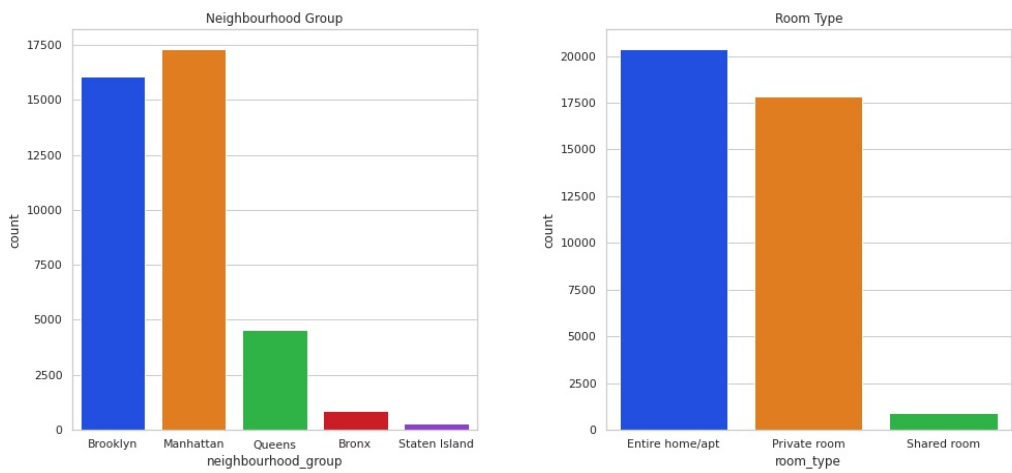


FIGURE 4.2: Histograms of Neighbourhood Group and Room Types

In the next figure, it could be observed that what room types are popular in which neighbourhood groups. It could be seen that renting entire homes are popular in Manhattan and Brooklyn, but in the latter, private rooms are slightly ahead of entire homes. Shared room renting is significantly lower in all of the room types.

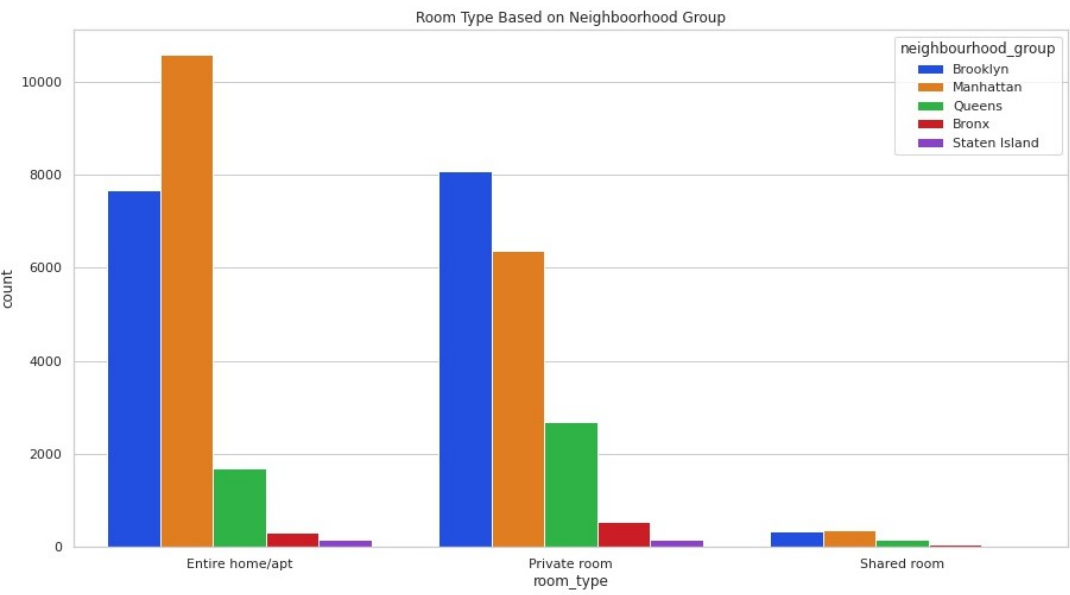


FIGURE 4.3: The distribution of room types among the neighbourhood group

The same conclusion could be drawn from the map of New York City.

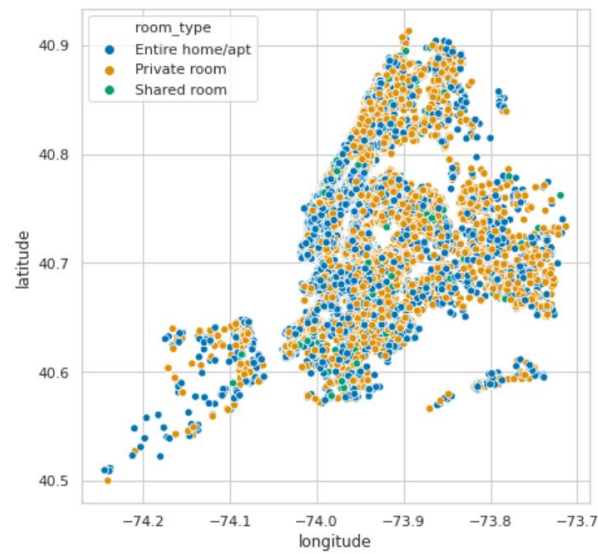


FIGURE 4.4: Room Types' distribution over NYC map

An interesting boxplot worth showing is the availability of neighbourhood groups during the year. What is striking, the median value of Staten Island regarding the availability is just under 250 days. Properties in Manhattan are high in demand.

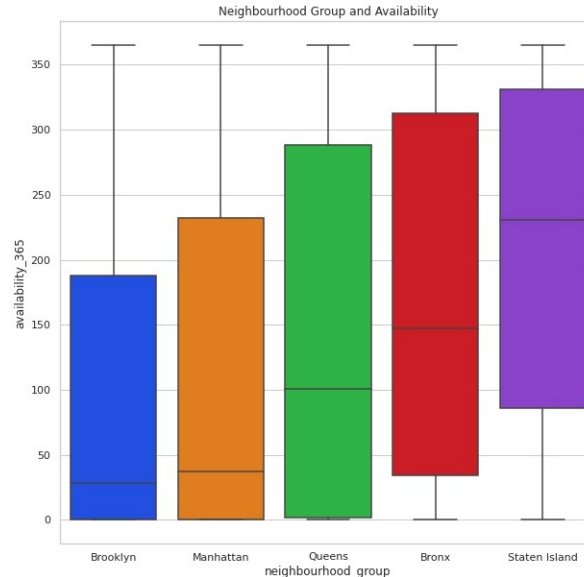


FIGURE 4.5: Availability of NYC Airbnb according to each neighbourhood group

4.4 Outliers

We could see the boxplots of all variables from Figure 4.6:

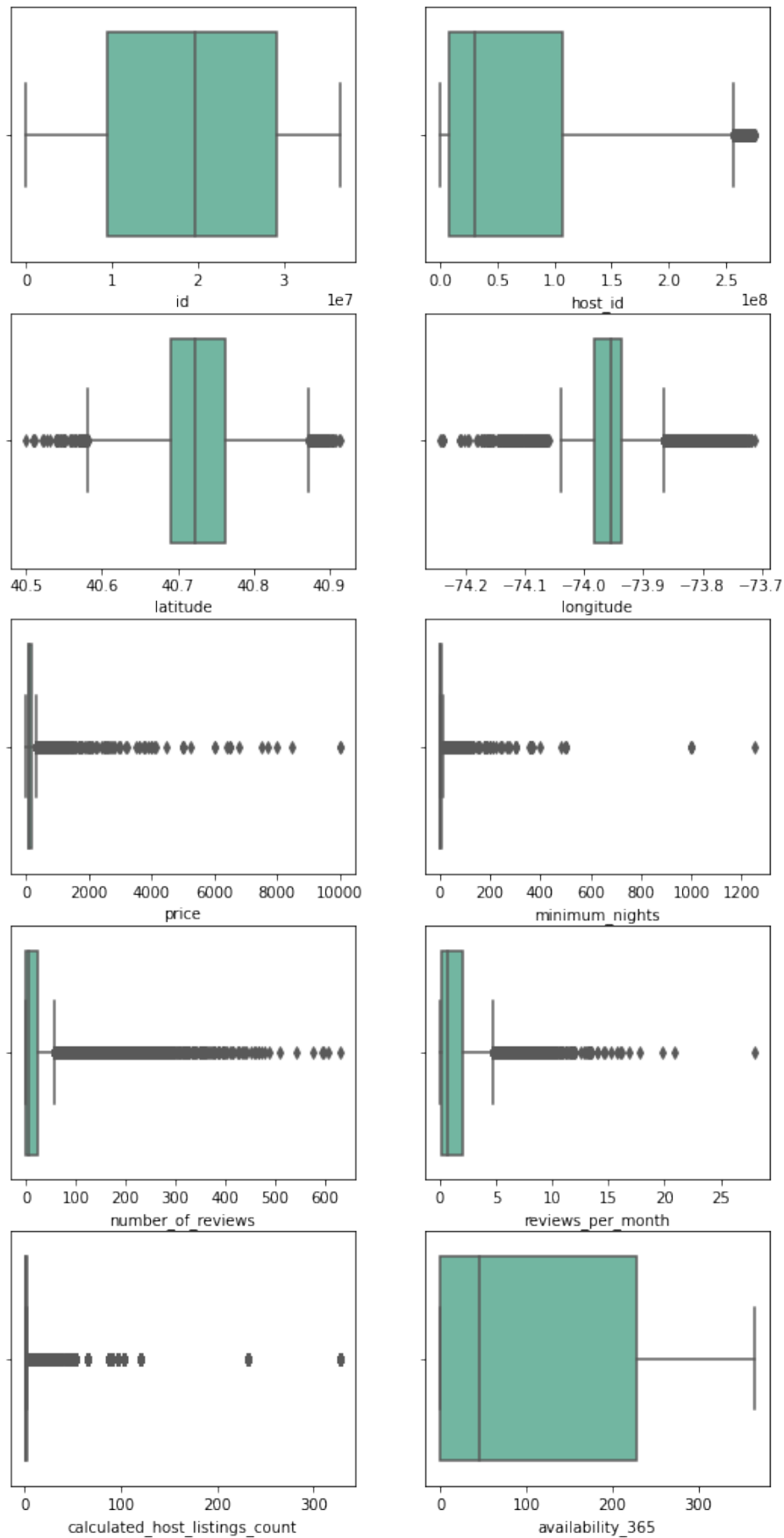


FIGURE 4.6: Boxplots of the numerical variables

What can be observed from these host of boxplots is that it just confirmed our hypothesis from Figure 3.3 that lots of features are skewed and follow tail-heavy distributions. In addition, there are a considerable amount of outliers in some attributes. There are several techniques to handle outliers, but I opt for the **Y-J** transformation, which will be discussed in the next chapter. As mentioned earlier, it is essential to manage outliers due to the fact that it can negatively influence the effectiveness of the ML algorithms on our data.

4.5 Tinkering with features

In this subsection, we are going to do some simple cleaning, imputation, feature extraction on the copied data set. The appropriate data preprocessing will be carried out in Chapter 5. Some attributes should be removed due to being irrelevant not only for the correlation analysis but for the whole prediction. These variables are *id*, *host_id*, *name*, *host_name*, *last_review*. As it is known, there are 8062 missing values from the *reviews_per_month* that should be somehow treated. There are several methods to deal with this problem, but in this case, the most appropriate one is to fill missing values by the median value of the *reviews_per_month*. Using median is more robust to the outliers in this variable than applying mean value to fill missing values. After that, one attribute can be established and added to the data set, namely the *months*. It is created by dividing *number_of_reviews* with *reviews_per_month*, and its purpose is to measure the number of months people writing reviews about a specific Airbnb property.

4.6 Correlation

We wish to see how the *price* target variable correlates with (continuous) numerical feature variables. The table below shows all these relationships:

| Features | Correlation Coefficient with <i>price</i> |
|--------------------------------|---|
| availability_365 | 0.086939 |
| calculated_host_listings_count | 0.060829 |
| latitude | 0.035996 |
| minimum_nights | 0.028761 |
| reviews_per_month | -0.038919 |
| months | -0.040933 |
| number_of_reviews | -0.049548 |
| longitude | -0.161296 |

TABLE 4.4: Feature variables' correlations with *price*

It can be recognised that none of these variables has strong predictive power for the *price* in a linear space as they are relatively uncorrelated to the price. Even the attribute *longitude* with the highest correlation coefficient in absolute value is fairly weakly correlated with the *price*. However, it does not mean that none of those features has a weak relationship with our target variable in a learning algorithm operating in higher dimensions.

Another interesting question is how feature variables are correlated with each other. Are there any signs of multicollinearity in our data set which could weaken ML algorithms? Let's find out. Here is the correlation matrix heatmap:



FIGURE 4.7: Correlation Matrix Heatmap showing linear relationships between the features

It can be noticed that *reviews_per_month* and *number_of_reviews* fairly correlate with each other with the correlation coefficient of 0.57. It may be an option to remove the *number_of_reviews* because *reviews_per_month* and *months* do not correlate too much with each other in contrast to *number_of_reviews* with *months*. It can be assumed that even in a higher dimension, *reviews_per_month* and *number_of_reviews* are quite strongly related due to the fact that they are very similar measures. In this case, **Occam's razor** principle is more or less applicable later on ML models meaning that given all other things being equal, a shorter explanation for observed data should be favoured over a lengthier explanation. As we can see from Section 5.4, after appropriate transformations those correlations become more robust, so it is suggested to remove *number_of_reviews*.

4.7 Some miscellaneous analysis

There are 221 unique neighbourhoods in the train + dev set. Here are the top 5 neighbourhoods in this data set:

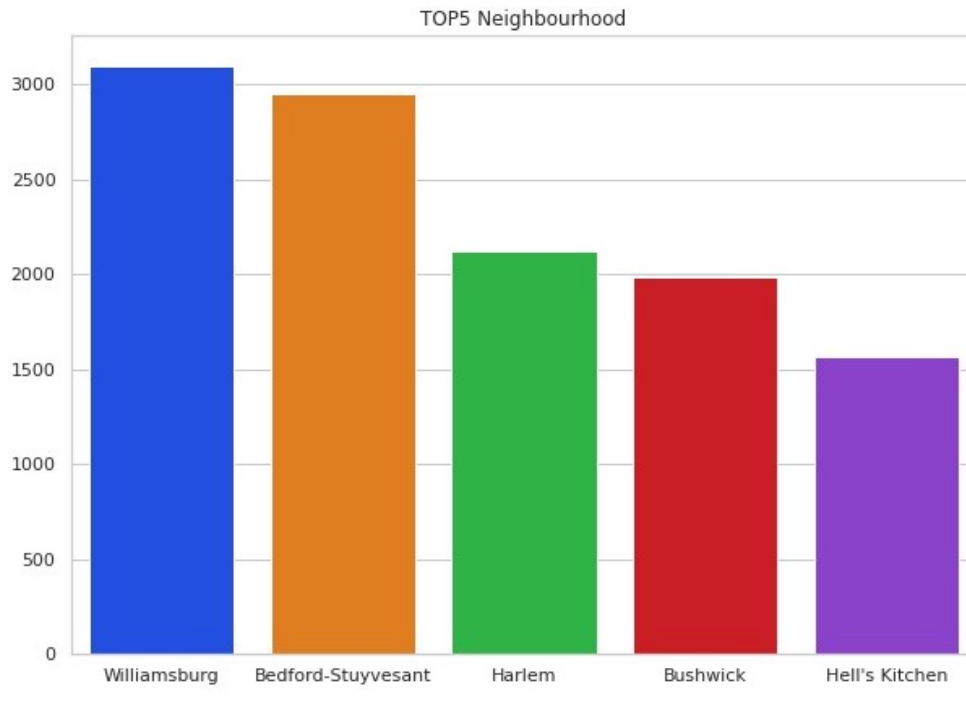


FIGURE 4.8: Top 5 neighbourhoods in New York City

There are 3098 properties in Williamsburg, 2948 in Bedford-Stuyvesant, 2117 in Harlem, 1986 in Bushwick, 1568 in Hell's Kitchen. In the 4.1, *neighbourhood_group* was chosen to be the basis for the stratified sampling. This *neighbourhood* variable could have been a better option if there had been enough accommodations in the data set. It means that for example, there are neighbourhoods with only one instance such as Fort Wadsworth or New Drop. The reason why *neighbourhood* would have been more suitable for our purpose is that even inside a *neighbourhood_group* there are a considerable amount of variabilities in terms of prices. Being more specific in terms of territory could have given more leverage to predict prices. For this reason, we should drop *neighbourhood* variables before applying a learning algorithm.

Chapter 5

Data Preprocessing

In this chapter, I am going to walk through the data processing steps, namely, data cleaning, handling categorical variables, imputation, feature selection and extraction with Python code examples that I created for this purpose. Before doing any data preprocessing, it is essential at this point in this project to separate the target variables from the train and dev sets.

5.1 Data Cleaning

In the following code snippet, we could see some initial data cleaning steps before delving into further data preprocessing:

```
def dropUnnecessary(df, df_target, dropList):
    df_new = df.drop(dropList, axis=1)
    return df_new, df_target

def dropCategorical(df):
    dropList1 = df.select_dtypes(include=['object']).columns
    df_num = df.drop(dropList1, axis=1)
    return df_num

def pureVersion(df, df_target, dropList):
    imputed, imputed_target = dropUnnecessary(df, df_target, dropList)
    imputed_num = dropCategorical(imputed)

    return imputed, imputed_num, imputed_target
```

TABLE 5.1: Data cleaning before any transformations on the data set

In the *pureVersion* function, we aim to create 3 data sets. **imputed** data frame includes only the relevant features parametrised by *dropList* list containing user-defined useless features. **imputed_num** only contains numerical variables from **imputed**. **imputed_target** accommodate our target variable, the *price*. In our case, *dropList* contains the following features:

```
dropList = ['name', 'host_name', 'last_review', 'neighbourhood', 'host_id',
            'id']
```

These attributes, as discussed in the previous chapter, are not particularly useful for the regression task for different reasons, so it is advisable to dump them.

5.2 Transformation Pipeline and Feature Scaling

```
def df_processed(df, df_target, dropList, desired_cat_attr, full_pipeline1
                = None, full_pipeline2 = None, pT = None):
    imputed, imputed_num, imputed_target = puredVersion(df, df_target,
                                                         dropList)

    imputed_target, targetTransform = target_var_transformer(imputed_target,
                                                             pT)

    strategies = ['imputer', 'y-j']
    for strategy in strategies:
        if strategy == 'imputer':
            imputed_num, fittedColumnTransform1 =
            num_var_transformer(imputed_num, full_pipeline1, strategy)
        else:
            imputed_num, fittedColumnTransform2 =
            num_var_transformer(imputed_num, full_pipeline2, strategy)

    imputed_concat = concat_num_cat_var(imputed, imputed_num,
                                         desired_cat_attr)
    imputed_concat.drop(['latitude'], axis=1, inplace=True)

    return imputed_concat, imputed_target, fittedColumnTransform1,
           fittedColumnTransform2, targetTransform
```

TABLE 5.2: Data transformation and feature scaling according to Yeo-Johnson

df_processed is our data transformation function which encapsulates *puredVersion* (Table 5.1), *target_var_transformer* (Table C.2), *num_var_transformer* (Table C.1), and *concat_num_cat_var* (Table C.3) functions. Details about their Python codes can be found in Appendix C. In short, what *df_processed* function does is that firstly, it imputes the missing values in an attribute by using median values of that feature. The reason for preferring median to mean values is using median values for imputation would make that attribute more robust to outliers compared to applying mean values. Then in case of the train set, it **fits then transforms** numerical and target variables using **Yeo-Johnson transformation**. In case of the dev and test set, it **only transforms** those two types of variables using the transformation pipelines defined by *fittedColumnTransform1*, *fittedColumnTransform2*, and *targetTransform* when fitting data on the train set. At the same time, when carrying out Y-J transformation, standardisation is implemented, that is applying zero-mean, unit-variance normalisation to the transformed data. *df_processed* also one-hot encodes categorical variables controlled by the *desired_cat_attr* list parameter and concatenate it with the Y-J transformed numerical variables. In the end, *latitude* feature is dropped for a reason discussed in Section 5.4

5.3 Yeo-Johnson transformation

Data transformation, which is defined on \mathbb{R} for reducing skewness and making a distribution normal-like, has a significant role in regression tasks (Weisberg, 2001;

Yeo & Johnson, 2000). Brief detail about the **Box-Cox** transformation should be outlined, because the Yeo-Johnson transformation is just the extension of the Box-Cox transformation for non-positive variables.

The transformation made by Box & Cox (1964) is the ψ^{BC} given by:

$$\psi^{\text{BC}}(\lambda, x_i) = \begin{cases} (x_i^\lambda - 1)/\lambda & \text{if } \lambda \neq 0, \\ \log(x_i) & \text{if } \lambda = 0, \end{cases} \quad (5.1)$$

for positive x_i . Those two are the data point in our data set for all i .

The transformation proposed by Yeo & Johnson (2000):

$$\psi^{\text{YJ}}(\lambda, x_i) = \begin{cases} ((x_i + 1)^\lambda - 1)/\lambda & \text{if } \lambda \neq 0, x_i \geq 0, \\ \log(x_i + 1) & \text{if } \lambda = 0, x_i \geq 0, \\ -[(-x_i + 1)^{2-\lambda} - 1]/(2 - \lambda) & \text{if } \lambda \neq 2, x_i < 0, \\ -\log(-x_i + 1) & \text{if } \lambda = 2, x_i < 0, \end{cases} \quad (5.2)$$

where x_i is the data point in our data set for all i , and λ is the parameter approximated by the *Maximum Likelihood Estimation*. The Yeo-Johnson transformation is the same as the Box-Cox transformation of $(x_i + 1)$ when x_i is positive. However, when x_i is negative, the Yeo-Johnson transformation is the same as the Box-Cox transformation of $(-x_i + 1)$ with the power of $(2 - \lambda)$. When λ equals either 0 or 2, a similar method applies to them but now with the *log* transformation. It is also important to notice that $+1$ at both x_i and $-x_i$. Therefore, the case when x_i is equal to 0 can be handled. The power of the Yeo-Johnson transformation is that it enables us to extend the notion of normality approximation to variables containing 0 and negative values.

5.4 Analysis of the transformed data

After transforming the data frame to the desired form, the following could be noticed using:

- All the values of *latitude* variable are transformed into 0. The reason is that in the train + dev set the untransformed maximum and minimum values in that feature are 40.499790 and 40.913059, respectively. So after the Yeo-Johnson transformation, those values are translated into 0 across all instances. To this end, *latitude* can be dropped from the transformed data set as it has no predictive power for this regression task.
- Figure 5.1¹ shows that *number_of_reviews* has strong correlations to both months and *reviews_per_month* after the transformation. The decision is to remove *number_of_reviews* from our data set.

¹The Figure 5.1 contains the *number_of_reviews* and *latitude* but the implementation Python code C.1 and 5.2 do not.

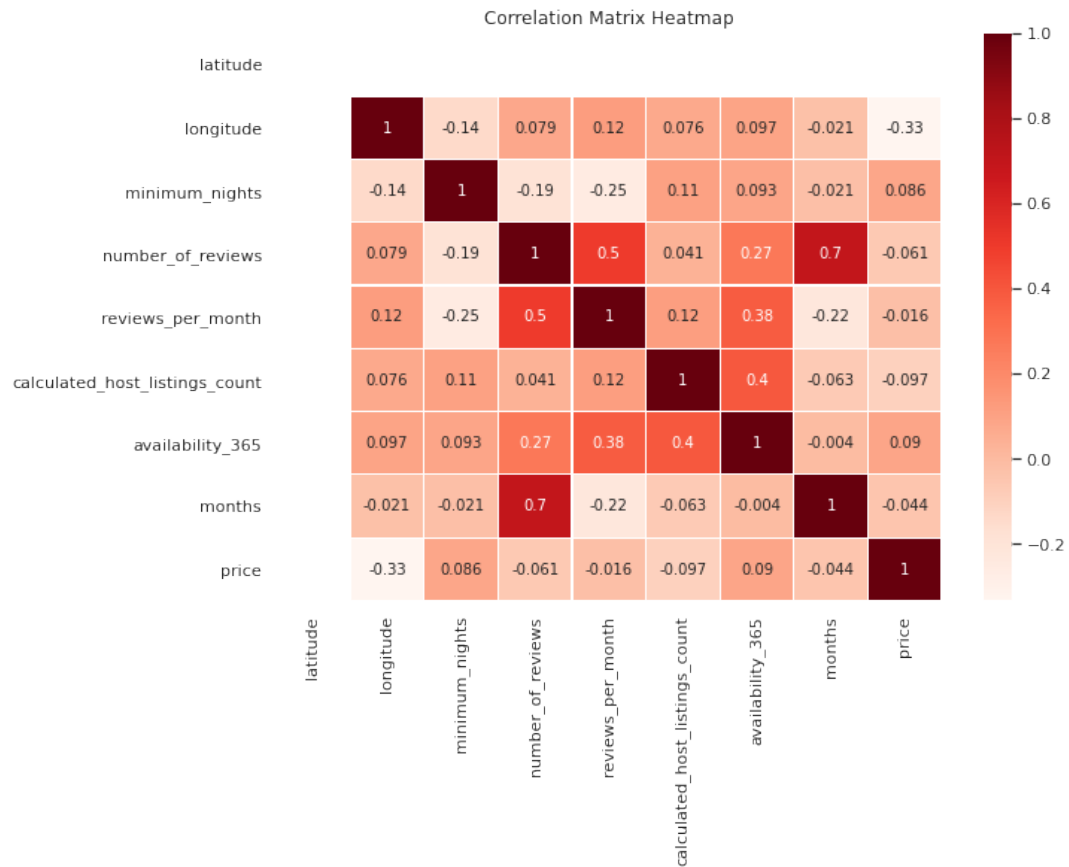


FIGURE 5.1: Correlation Matrix Heatmap on the Y-J transformed data

Chapter 6

Model Training and Testing

In this chapter, a brief introduction of dimensionality reduction along with feature selection and factor analysis is presented. It is also shown how key steps of Principal Component Analysis (PCA) are carried out in practice, and what technique should be used when dealing with data containing both numerical and categorical variables. After that, we will delve into the details of training models and their evaluation. Some examples of models that are used in this chapter: Linear Regression, Ridge/Lasso Regression, SVR with different kernels, Ensemble models. Mathematical backgrounds of some of those methods are contained in Appendix A. Our main goal is to find the most promising models after doing evaluation either on the dev set or through k-fold cross-validation for further hyperparameter optimisation. In the end, the performance of our models is displayed, and among them, the best performing one is going to be run on the unseen test set.

6.1 Dimensionality Reduction

Working with high-dimensional data often comes with challenges as it is hard to analyse, interpret, and visualise. Still, it usually has properties that can be harnessed. In most real-world problems, training instances are not spread out uniformly across all dimensions. For example, many dimensions/features are almost constant, while others can be expressed by a combination of the remaining ones. As a consequence, all training instances may lie within (or close to) a much lower-dimensional subspace. (Geron, 2019; Deisenroth et al., 2020). We deal with two types of dimensionality reduction techniques: *feature selection* and *feature extraction*.

6.1.1 Feature Selection

In Iuhaniwal's article (2019) states that in order to carry out ML task in high dimensional data, feature selection becomes essential. Some attributes may be irrelevant or less important to the target variable, so their inclusion to the model would lead to:

- Increase in complexity of a model and makes it harder to interpret.
- Increase in time complexity for a model to get trained.
- May result in an unreliable model with less accurate predictions.

These reasons give rise to the use of feature selection in ML. For a dataset with d -dimensional feature space, the feature selection process reduces to a k feature space such that $k < d$, where k is the number of a smaller set of significant features.

The benefits of feature selections are:

- Training an ML algorithm is much faster.
- Complexity reduction of a model.
- More interpretable models.
- Better prediction power of a more sensible model.
- Reduction in overfitting.

There are three main types of feature selection methods: Filter, Wrapper and Embedded methods.

In Table 6.1, the comparison of the three feature selection methods' properties is highlighted.

| Filter methods | Wrapper methods | Embedded methods |
|---|---|--|
| A generic set of methods not including a specific ML algorithm. | Evaluating on a specific ML algorithm to find optimal features. | Embedding (fix) features during model building process. Feature selection is done by observing each iteration of a model training phase. |
| A lot faster than Wrapper methods. | High computation time for a dataset with many features. | Sits between Filter and Wrapper methods in computation time. |
| Less susceptible to overfitting. | High chances of overfitting due to involving training of ML models with different combinations of attributes. | Reducing overfitting by penalising coefficients of a model being too large. |
| Pearson's Corr., χ^2 , LDA, etc.. | Sequential Forward/Backward Selection, etc... | Lasso/Ridge Regression, Elastic Net, etc.. |

TABLE 6.1: Filter vs. Wrapper vs. Embedded methods

In the project, only some of the Embedded methods will be used. For more detailed information about these three feature selections methods, please refer to Appendix B.1.

6.1.2 Factor Analysis

Another method for avoiding *the curse of dimensionality* is the factor analysis. The *Principal Component Analysis (PCA)*, *Multiple Correspondence Analysis (MCA)*, and *Factor Analysis of Mixed Data (FAMD)* are going to be discussed briefly. In the project, FAMD is applied, which is the combination of PCA and MCA.

Principal Component Analysis

The mathematical details behind this concept are discussed in Appendix B.2. Here, only the algorithm of the PCA is outlined (Raschka et al., 2019; Deisenroth et al., 2020):

1. Computing the mean μ of the dataset and subtracting it from every data point. It leads to the dataset having $\mathbf{0}$ as mean. This process may help in avoiding the risk of numerical problems. (Optional step)
2. Dividing the data points by the **standard deviation** σ_d of the dataset for every dimension $d = 1, \dots, D$. The data is unit-free, and has variance 1 along each axis.
3. Computing then decomposing the data symmetric covariance matrix into its eigenvalues and corresponding eigenvectors.
4. Sorting those eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. Selecting the *explained variance ratio* that is applied to the data. It will result in k eigenvectors, where $k(< d)$ is the dimension of the new feature subspace.
6. Constructing \mathbf{B} , the projection matrix from the best k eigenvectors.

Multiple Correspondance Analysis

MCA is often considered as a categorical analogy of Principal Component Analysis. According to Greenacre (2007), MCA is concerned with the relationships amongst (or within) a set of variables which can be recoded as dummy variables. As a result, it is going to form an *indicator matrix* a.k.a a one-hot encoded matrix. It has as many rows as cases and as many columns as categories of response. MCA is obtained by applying a standard correspondence analysis (CA) on this indicator matrix. The result is a linear combination of rows (also referred to as factors) that best describe the data. Similarly, as in the Principal Component Analysis, the first axis is the most important dimension, the second axis the second most important, and so on, in terms of the amount of variance they capture. The number of axes to be retained for analysis is determined by calculating modified eigenvalues.

Factor Analysis of Mixed Data

FAMD is the extension of MCA by including quantitative variables. It makes it possible to analyse the similarity between individuals by taking into account mixed types of variables. Additionally, one can explore the association between all variables, both quantitative and qualitative variables.

The criterion for FAMD (Pages, 2014):

Let's define \mathbb{R}^I as the *variables' space* or the *space of functions on I* (a function on I attributes a numerical/categorical value to each individual i). Let a centred unit vector be v of \mathbb{R}^I (of which the i th coordinate v_i is the value function v for individual i). As a result, v is a quantitative function. It turns out that within the same space of functions on I , it is highly useful to involve both numerical and categorical variables as the measurement of (the intensity of) the relationship between two variables of different types is expressed simply in this space. It is vital for analysing these two types of variables concurrently in FAMD.

Let the data include K_1 ($k_1 = 1, \dots, K_1$) standardised quantitative variables and K_2 ($k_2 = 1, \dots, K_2$) indicators for qualitative variables. Therefore, $K = K_1 + K_2$ is the total number of quantitative and indicator variables. It is known that:

- $r(k_1, v)$ is the correlation coefficient between variables k_1 and v .
- $\eta^2(k_2, v)$ is the squared correlation ratio between variables k_2 and v .

In the PCA of K_1 , the objective is to find the function on I that is the most correlated to all K_1 variables:

$$\max \sum_{k_1} r^2(k_1, v) \quad (6.1)$$

In case of the MCA of K_2 , the aim is to look for the function on I that is the most related to all K_2 variables:

$$\max \sum_{k_2} \eta^2(k_2, v) \quad (6.2)$$

For FAMD $\{K_1, K_2\}$, the objective is to find the function on I that is the most related to all $K_1 + K_2$ variables:

$$\max \left(\sum_{k_1} r^2(k_1, v) + \sum_{k_2} \eta^2(k_2, v) \right) \quad (6.3)$$

The following figure summarises the PCA, the MCA, and the FAMD:

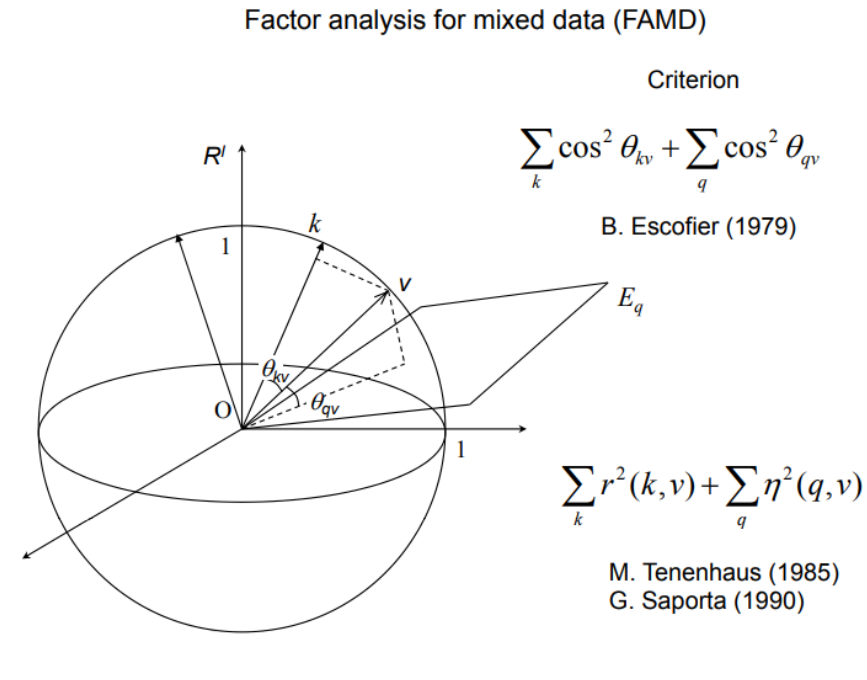


FIGURE 6.1: Illustration of the factor analysis (Pages, 2011)

The implementation of FAMD can be found in [C.4](#). It turns out from the code implementation that ten components are retained, and they account for around 95% of the explained variance. In the next several sections, some learning models are introduced with their background information. Those learning models are used in

this project. The decision to include only those learning models comes after training many quick-and-dirty ML models and evaluate them without hyperparameter optimisation.

6.2 Linear Regression

6.2.1 Vanilla Linear Regression

Section 6.2, 6.3 and the corresponding Appendix A.1 are based on Andrew Ng's CS229 lecture note about supervised learning (2020), and Hastie, Tibshirani and Friedman's book about statistical learning (2009). One of the most broadly used supervised learning models for regression tasks is Linear Regression. It assumes that the target variable y is a function of inputs with ϵ residual error that captures either unmodeled effect or random noise.

$$y(x) = \theta^T x + \epsilon = \sum_{i=0}^d \theta_i x_i + \epsilon \quad (6.4)$$

Let's introduce some notations for future uses (not just for Linear Regression):

- n : number of training instances
- d : number of features
- θ : Parameters of a model
- θ_j : the parameter of the j^{th} feature variable
- $J(\theta)$: (Least-squares) cost function
- $x_j^{(i)}$: The j^{th} feature variable of the i^{th} instance
- $y^{(i)}$: Corresponding target variable of the i^{th} in the training set
- $(x^{(i)}, y^{(i)})$: i th training example
- $\{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$: training set
- \mathcal{X} : Input Space
- \mathcal{Y} : Output Space
- $h_\theta(x)$ or $h(x)$: Hypothesis function

Given the training set, our goal is to learn the function $h : \mathcal{X} \mapsto \mathcal{Y}$. It means that $h(x)$ approximates the corresponding value of y as closely as possible. So the corresponding equation of the y in Equation (6.4) is almost the same except the lack of the presence of the error term

$$h(x) = \theta^T x = \sum_{i=0}^d \theta_i x_i, \quad (6.5)$$

where θ_j 's are the parameters parameterising the above mentioned linear mappings, and we let $x_0 = 1$ (the intercept term).

The measurement of the difference between the $y^{(i)}$ and $h(x^{(i)})$ is done by the cost function:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^n (h_\theta(x^{(i)}) - y^{(i)})^2 \quad (6.6)$$

or equivalently in norm form

$$J(\theta) = \frac{1}{2} \sum_{i=1}^n \|y^{(i)} - \theta^T x^{(i)}\|^2. \quad (6.7)$$

Our aim is to

$$\min_{\theta} J(\theta), \quad (6.8)$$

that is minimising the cost function when approximating the $y^{(i)}$ with $h(x^{(i)})$ by choosing the appropriate θ parameter, which can be found in many ways. The first method for finding the optimal θ when training a model is by using the *normal equation*, the closed-form solution, which does the trick in one step.

$$\hat{\theta} = \arg \min_{\theta} J(\theta) = (X^T X)^{-1} X^T y, \quad (6.9)$$

which is the **OLS** estimator for θ . $X \in \mathbb{R}^{n \times (d+1)}$ design matrix given a training set. This approach may not work if $X^T X$ is not invertible. For those cases, calculating $\hat{\theta} = X^+ y$ is the way forward, where X^+ is the pseudoinverse of X (Moore-Penrose inverse). This method harnesses the power of the *Singular Value Decomposition* (SVD). Using pseudoinverse is more efficient than the normal equation, and it also handles non-invertible X cases well (Geron, 2019). The drawback of these two one-step solutions is that they are computationally expensive, especially when the number of features grows fast. In the normal equation, inverting the $X^T X$ matrix takes about $O(n^4)$ to $O(n^3)$, while using SVD costs "only" $O(n^2)$.

Other methods for minimising $J(\theta)$ is using the gradient descent algorithm, which is more suitable for cases when there are lots of features or too many training examples to fit in memory. More details can be found in Appendix A.1.

6.3 Regularised Linear Regression

We specifically mention three shrinkage methods: Ridge Regression, Lasso Regression (Least Absolute Shrinkage and Selection Operator Regression), and Elastic Net (a mixture of the Ridge and the Lasso). The common feature of all these techniques is that they add regularisation term to their respective cost functions to avoid overfitting. They are also a part of the embedded feature selection methods because the regularisation term penalises the parameter θ being too big. As a result, they influence the feature subset used for model training. Using Equation (6.7), the regularised cost functions are

$$J_{\text{Ridge}}(\theta) = J(\theta) + \frac{\lambda}{2} \|\theta\|_2^2 \quad (6.10)$$

$$J_{\text{Lasso}}(\theta) = J(\theta) + \lambda \|\theta\|_1 \quad (6.11)$$

$$J_{\text{Elastic}}(\theta) = J(\theta) + r \lambda \|\theta\|_1 + (1-r) \frac{\lambda}{2} \|\theta\|_2^2, \quad (6.12)$$

where $\|\theta\|_2^2 = \sum_{i=1}^n \theta_i^2$, $\|\theta\|_1 = \sum_{i=1}^n |\theta_i|$, λ is the hyperparameter controlling the regularisation of a model, and r is the mix ratio between Ridge and Lasso Regression (Geron, 2019).

6.4 Support Vector Machine

The aim of the Support Vector Classification (SVC) is to insert the widest possible road between two classes. At the same, it wishes to minimise the margin violation. In other words, the number of data points to include on that road or on the wrong side of the road. However, the objective of the Support Vector Regression (SVR) is to try to fit as many instances as possible on the street while limiting margin violations, that is instances off the road. Therefore, the objectives of SVR and SVC are opposite. In this project, we only tackle the SVR. In the case of SVR, the width of the road is controlled by a parameter, ϵ . Including more training examples within the margin does not affect the model's predictions. That is why SVR is considered to be an ϵ -sensitive model. To overcome nonlinear regression tasks, the way to go is to use a kernelised SVM model (Geron, 2019). The mathematical details of kernel methods, SVR and their relationship can be found in Appendix A.2 and A.3.

6.5 Gradient Boosting – XGBoost

The general idea of boosting is to train all the predictors one-by-one consequently, so the next predictor tries to correct the drawbacks of the preceding ones. XGBoost (Extreme Gradient Boosting) is an optimised implementation of the Gradient Boosting. In contrast to Adaboost, Gradient Boosting does not tinker with the weight of new instances. It tries to fit the new predictor on the *residual error* caused by the previous predictor(s) (Geron, 2019). The base regressor for the XGBoost is a decision tree, which is also called in this context as Gradient Boosted Regression Trees (GBRT).

6.6 Bagging and Pasting

Bagging and Pasting is an Ensemble learning method which uses the same basic learning algorithm such as a decision tree for every predictor and trains them on different subsets of the training set. When sampling **with** replacement, it is called *Bagging*. If it is done **without** replacement, it is called *Pasting*. That is, *Bagging and Pasting* allow training instances to be sampled several times across multiple predictors. Still, only Bagging enables training instances to be sampled several times for the same predictor. Once all predictors are trained, the ensemble can predict for a new instance by simply aggregating the predictions of all predictors. The aggregation function for the regression task is the *average*. Each predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance. In the end, typically the result of Bagging and Pasting has an effect of having a lower variance than a single predictor trained on the original training set with a similar bias. (Geron, 2019)

6.7 Voting Regression

Voting Regression is an Ensemble learning method that fits several base regressors, each on the whole data set. Then it averages the individual predictions to form a final prediction. Well-chosen weakly dependent (ideally independent) diverse regressors could uncover and smooth each other's disadvantages, therefore boost the overall performance of the Voting Regression.

6.8 Results of the Model Training

In this section, the above-mentioned learning models are hyperparameter tuned and cross-validated on the training set. From now on, the meaning of the training set is the train + dev set together. Grid Search Cross-Validation technique is an exhaustive hyperparameter searching and optimising method coupled with cross-validation. Grid Search CV experiments with all the hyperparameter values provided by the user, then it uses cross-validation to evaluate all the possible combinations of hyperparameter values. In this project, Grid Search CV is only used on Regularised Linear Regression and XGBoost Regression because of the time and memory constraints. All the others are hyperparameter tuned by hand.

Hyperparameters of all those models are:

- *Regularised Linear Regression* (with SGD):
`{ 'alpha': 0.005, 'epsilon': 0.01, 'eta0': 0.01, 'learning_rate': 'optimal', 'loss': 'squared_loss', 'penalty': 'l2', 'random_state': 69 }`

It turns out that after doing Grid Search CV, the Regularised Linear Regression becomes Ridge Regression because of the 'l2' penalty among the hyperparameter values.

- *SVR with Gaussian Kernel*:
`{ 'kernel': 'rbf', 'C': 2.0, 'gamma': 0.2, 'epsilon': 0.5 }`
- *Bagging*:
`{ 'base_estimator': 'DecisionTreeRegressor()', 'n_estimators': 200, 'max_samples': 1000, 'bootstrap': 'True' }`
- *Voting Regression*:
`{ 'estimators': [('xgb', xgb2_reg), ('svmGauss', svmGauss1), ('bagging', bagging1_reg)],`
 where *svmGauss1* is the SVR regressor with the hyperparameters above, *bagging1_reg* is the Bagging regressor with the hyperparameters above, and *xgb2_reg* is an XGBoost regressor (not the final one!) defined by the hyperparameters as follows:
`{ 'max_depth': 5, 'learning_rate': 0.05, 'objective': 'reg:squarederror', 'n_estimators': 200, 'booster': 'gbtree', 'gamma': 10, 'subsample': 0.9 }`
- *XGBoost Regression*:
`{ 'booster': 'gbtree', 'gamma': 5, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 500, 'objective': 'reg:squarederror', 'subsample': 0.9 }`

As it is mentioned in Section 2.2, the performance measurement unit is the *RMSE*. 6-fold cross-validation is used to evaluate the performance of the learning models. It means that the training set is divided into six subsets. Five of them are subsets on which a learning model is trained, and the evaluation happens on the sixth one. This process is done on all the six subsets being an evaluation subset after training on the other five subsets. Then the system averages all the six *RMSE* values given by the standalone evaluation subsets.

The result of the training and cross-validation *RMSE* of the learning models are summarised in the following table:

| Model | Training <i>RMSE</i> | CV <i>RMSE</i> |
|--------------------------|----------------------|-------------------|
| Ridge | 0.71539861 | 0.71389930 |
| SVR w/ RBF kernel | 0.64375725 | 0.65615193 |
| Bagging | 0.66090620 | 0.67705554 |
| Voting | 0.64708188 | 0.66379841 |
| XGBoost | 0.62054892 | 0.66705217 |

TABLE 6.2: Training and CV *RMSE* errors of the learning models

The best performing model, according to the CV *RMSE*, is the SVR with Gaussian (RBF) kernel after hyperparameter tuning. Therefore, the final model is the SVR with Gaussian kernel with the appropriate hyperparameter values, which is going to be tested on the unseen test set data. The Python code of the winning model can be found in [C.5](#) and [C.6](#).

6.9 Final Model and Testing

Running the SVR with Gaussian (RBF) kernel learning model on the unseen test data yields us *RMSE* of **0.66589135**, which is slightly worse than the 6-fold CV *RMSE*. Still, it is kind of expected because the hyperparameters are tuned on the validation sets and not on the unseen data. The Python code can be found in [C.7](#).

Chapter 7

Further Development Opportunities

There are a lot of things that could have been implemented in the project to make predictions better or to extend it at a greater scale. In the following sections, these issues are discussed.

7.1 Other Learning Algorithms

In *Keras* and *Tensorflow* libraries, Artificial Neural Network (ANN) and Deep Learning (DL) algorithms could have been carried out on the NYC Airbnb data set. However, more computational power may be needed to implement those more complex learning algorithms than what I had on Google Colab.

7.2 More features

Generally speaking, it would have been useful if we had more data. However, better additional features would have been much more appreciated for our project. Some suggestions for those attributes are:

- The median income in a *neighbourhood_group*.
- Some crime statistics.
- Public transport availability around accommodations on the scale of 5, where 1 is the worst and 5 is exceptionally good.
- Property prices around accommodations in terms of either m^2 or ft^2 .
- Distribution of types of shops around accommodations.
- Distances to some prominent sightseeing places.
- Time Series statistics for the changes in prices for accommodations.

7.3 Implementation in real life

Hypothetically, if we are ready with our models, it could be developed into the production environment as a part of a wide-scale system. It could be done in a web application where a user gives some inputs such as price category or the name of a neighbourhood. Then it sends this query to a web server which would call the model's *predict()* method and direct the answer back to the user. Furthermore, by

having advanced technologies in cloud computing, it could be implemented in either on the Google Cloud AI Platform, AWS (Amazon Web Service) or Microsoft Azure. However, it is not the end of the story by implementing our model in a production environment. For the model's performance to be consistent and adaptive, an appropriate program should be written to check models' real live performance as more and more data coming into the system. (Geron, 2019) The system should send an alert if the performance dropped significantly for some reasons such as outdated data or matters which do not relate to the model itself. Continuous data collection, training and hyperparameter tuning are needed for this kind of system. Some human raters may be required for tagging labels in case of classification models where the models could be unstable due to lack of data. A platform for such raters is, for example, the Amazon Mechanical Turk.

Appendix A

Technical Details

A.1 Linear Regression

A.1.1 Gradient Descent

Our purpose is to minimise $J(\theta)$. Now, an iterative method is shown, namely the gradient descent algorithm, which takes a step in the steepest descent direction of the objective function. It starts with some initial θ , and iteratively performs the update for all values of $j = 0, \dots, d$ at the same time:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), \quad (\text{A.1})$$

where α is the *learning rate*. Before going into deeper in the algorithm, the partial derivative part has to be discussed in the previous update rule for the gradient descent. For a single training example (x, y) , the cost function is $J(\theta) = 1/2 (h(x) - y)^2$.

$$\begin{aligned} \frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h(x) - y)^2 \\ &= \frac{1}{2} \cdot 2(h(x) - y)^2 \frac{\partial}{\partial \theta_j} (h(x) - y)^2 \\ &= (h(x) - y) \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^d \theta_i x_i - y \right) \\ &= (h(x) - y) x_j \end{aligned} \quad (\text{A.2})$$

Therefore the update rule for only one training example

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h(x^{(i)}) \right) x_j^{(i)} \quad (\text{A.3})$$

This update rule is known as the least mean squares (*LMS*) update rule (also called as *Widrow-Hoff* learning rule). Tweaking this LMS rule for considering the whole data set would give us **batch gradient descent** (BGD) and **stochastic gradient descent** (SGD).

Batch Gradient Descent

Repeat until convergence {
 For $j = 0, 1, \dots, d$ (iterating over the number of features)

$$\theta_j := \theta_j + \alpha \sum_{i=1}^n \left(y^{(i)} - h(x^{(i)}) \right) x_j^{(i)} \quad (\text{A.4})$$

}

By putting all θ_j updates into a vector θ , Equation (A.4) can be rewritten in the following way:

$$\theta := \theta + \alpha \sum_{i=1}^n \left(y^{(i)} - h(x^{(i)}) \right) x^{(i)} \quad (\text{A.5})$$

It is the *batch gradient descent* (BGD) on the cost function $J(\theta)$. The reason why this is *batch* is that algorithm above consider every example in the entire training set on every step. In general, gradient descent (GD) can be susceptible to a local minimum, but fortunately, the cost function for the Linear Regression happens to be convex. It means that it has only one global minimum point, so the GD will always converge to that minimum point (unless the learning rate α is not too large).

Stochastic Gradient Descent

Repeat until convergence {

For $i = 1, \dots, n$ (iterating over the number of training instances)

For $j = 0, 1 \dots, d$ (iterating over the number of features)

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h(x^{(i)}) \right) x_j^{(i)} \quad (\text{A.6})$$

}

Similarly, Equation (A.6) can be expressed concisely:

$$\theta := \theta + \alpha \left(y^{(i)} - h(x^{(i)}) \right) x^{(i)} \quad (\text{A.7})$$

It is the *stochastic gradient descent* (SGD) algorithm. The advantage of the SGD over the BGD for a more extensive training set is that BGD has to take into account the entire training set before taking any single step. It is costly for a massive number of training instances. Whereas SGD at any single step, it just takes an instance and carries on with it for that step. Usually, SGD gets θ close to the optimum significantly faster than BGD, but unlike BGD, it may never converge to that optimum. θ might just jump around (oscillate) the minimum of $J(\theta)$. Despite this fact, the SGD algorithm will approximate θ to the true minimum surprisingly well.

A.1.2 Normal Equation

Let $X \in \mathbb{R}^{n \times (d+1)}$ design matrix that include training instances' input values in its rows:

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(n)})^T \end{bmatrix} \quad (\text{A.8})$$

Let $y \in \mathbb{R}^{n \times 1}$ include all the target values from the training set:

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \quad (\text{A.9})$$

Because of $h(x^{(i)}) = (x^{(i)})^T \theta$, it can be recognised that

$$\begin{aligned} X\theta - y &= \begin{bmatrix} (x^{(1)})^T \theta \\ (x^{(2)})^T \theta \\ \vdots \\ (x^{(n)})^T \theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \\ &= \begin{bmatrix} h(x^{(1)}) - y^{(1)} \\ \vdots \\ h(x^{(n)}) - y^{(n)} \end{bmatrix} \end{aligned} \quad (\text{A.10})$$

By using the dot product property of $a^T a = \sum_i a_i^2$ for a vector a :

$$\begin{aligned} \frac{1}{2}(X\theta - y)^T(X\theta - y) &= \frac{1}{2} \sum_{i=1}^n (h(x^{(i)}) - y^{(i)})^2 \\ &= J(\theta) \end{aligned} \quad (\text{A.11})$$

To minimise $J(\theta)$, differentiate $J(\theta)$ with respect to θ and set this derivative to zero:

$$\begin{aligned} 0 &= \nabla_{\theta} J(\theta) = \nabla_{\theta} \frac{1}{2} (X\theta - y)^T (X\theta - y) \\ &= \frac{1}{2} \nabla_{\theta} \left((X\theta)^T X\theta - (X\theta)^T y - y^T (X\theta) + y^T y \right) \\ &= \frac{1}{2} \nabla_{\theta} \left(\theta^T (X^T X) \theta - 2 y^T (X\theta) \right) \\ &= \frac{1}{2} \nabla_{\theta} \left(\theta^T (X^T X) \theta - 2 (X^T y)^T \theta \right) \\ &= \frac{1}{2} \left(2 X^T X \theta - 2 X^T y \right) \\ &= X^T X \theta - X^T y \end{aligned} \quad (\text{A.12})$$

So after reconstructing Equation (A.12), we get:

$$X^T X \theta = X^T y \quad (\text{A.13})$$

$$\hat{\theta} = \arg \min_{\theta} J(\theta) = (X^T X)^{-1} X^T y \quad (\text{A.14})$$

So the closed-form normal equation gives us $\hat{\theta}$, the value that minimises the cost function $J(\theta)$.

A.1.3 Probabilistic interpretation

In this section, we give another interpretation of why $J(\theta)$ makes sense, which is the probabilistic interpretation. For a single training instance, rewriting Equation (6.4) gives us

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}. \quad (\text{A.15})$$

Suppose that $\epsilon^{(i)}$ are i.i.d and $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$. So the density function of $\epsilon^{(i)}$ is

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}\right). \quad (\text{A.16})$$

Plugging Equation (A.15) (rearranged for $\epsilon^{(i)}$) into (A.16)

$$p(y^{(i)}|x^{(i)}; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right), \quad (\text{A.17})$$

which equivalently means that $y^{(i)}|x^{(i)}; \theta \sim \mathcal{N}(\theta^T x^{(i)}, \sigma^2)$. For the whole training set X given by (A.8), the density function of y given by (A.9) is $p(y|X; \theta)$. If we look at this function as a function of the unknown parameter of θ , it is called the *likelihood function*. Owing to our assumption about the $\epsilon^{(i)}$, we can express it as

$$\begin{aligned} L(\theta) &= L(\theta; X, y) = p(y|X; \theta) \\ &= \prod_{i=1}^n p(y^{(i)}|x^{(i)}) \\ &= \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right). \end{aligned} \quad (\text{A.18})$$

We would like to find the best $\hat{\theta}$ that maximises this $L(\theta)$ function. This process is called *Maximum Likelihood Estimation*. However, instead of using this strictly increasing $L(\theta)$, let's use the *log-likelihood* $l(\theta)$

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^n \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= n \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2. \end{aligned} \quad (\text{A.19})$$

It can be seen that maximising $l(\theta)$ is the same as

$$\min_{\theta} \frac{1}{2} \sum_{i=1}^n (h_{\theta}(x^{(i)}) - y^{(i)})^2 = \min_{\theta} J(\theta). \quad (\text{A.20})$$

A.1.4 Shrinkage Methods

Here we discuss only *Lasso* and *Ridge*. The optimum $\hat{\theta}$ s that minimise the cost functions of $J_{Ridge}(\theta)$ (6.10) and $J_{Lasso}(\theta)$ (6.11) (separately) are

$$\begin{aligned}\hat{\theta}^{Ridge} &= \arg \min_{\theta} \{J(\theta) + \lambda \|\theta\|_2^2\} \\ &= \arg \min_{\theta} \left\{ \frac{1}{2} \sum_{i=1}^n \|y^{(i)} - \theta^T x^{(i)}\|^2 + \frac{\lambda}{2} \|\theta\|_2^2 \right\} \\ &= \arg \min_{\theta} \left\{ \sum_{i=1}^n \|y^{(i)} - \theta^T x^{(i)}\|^2 + \lambda \|\theta\|_2^2 \right\}\end{aligned}\tag{A.21}$$

$$\begin{aligned}\hat{\theta}^{Lasso} &= \arg \min_{\theta} \{J(\theta) + \lambda \|\theta\|_1\} \\ &= \arg \min_{\theta} \left\{ \frac{1}{2} \sum_{i=1}^n \|y^{(i)} - \theta^T x^{(i)}\|^2 + \lambda \|\theta\|_1 \right\}.\end{aligned}\tag{A.22}$$

$\lambda \geq 0$ is the parameter controlling the amount of shrinkage. The larger the value of λ , the greater the amount of shrinkage. The coefficients are shrunk toward zero (and each other).

The equivalent way to express (A.21) and (A.22) are

$$\begin{aligned}\hat{\theta}^{Ridge} &= \arg \min_{\theta} \left\{ \sum_{i=1}^n \|y^{(i)} - \theta^T x^{(i)}\|^2 \right\} \\ &\text{s.t. } \|\theta\|_2^2 \leq t\end{aligned}\tag{A.23}$$

$$\begin{aligned}\hat{\theta}^{Lasso} &= \arg \min_{\theta} \left\{ \frac{1}{2} \sum_{i=1}^n \|y^{(i)} - \theta^T x^{(i)}\|^2 \right\} \\ &\text{s.t. } \|\theta\|_1 \leq t.\end{aligned}\tag{A.24}$$

There is a one-to-one correspondence between the parameters λ (in (A.21) and (A.22)) and t (in (A.23) and (A.24)). When there is a lot of high correlation between variables in the Linear Regression model, their coefficients can have significantly less predictive power. Therefore, the model is going to overfit. A similarly large negative coefficient can cancel a wildly large positive coefficient on one variable on its correlated cousin. By imposing a size constraint on the coefficients, this problem can be alleviated.

As we can see the constraint/penalty terms of Ridge and Lasso are in L_2 and L_1 respectively. Because the penalty term of Lasso makes the solutions nonlinear in the $y^{(i)}$, there is no closed-form expression as in the case of Ridge. So using the matrix form as in Section A.1.2 for the Ridge provides us

$$J_{Ridge}(\theta) = (X\theta - y)^T (X\theta - y) + \lambda \theta^T \theta\tag{A.25}$$

So the solution for the equation above is

$$\hat{\theta}^{Ridge} = \arg \min_{\theta} J_{Ridge}(\theta) = (X^T X + \lambda I)^{-1} X^T y. \quad (A.26)$$

With the choice of quadratic penalty $\theta^T \theta$, the Ridge solution is a linear function of y .

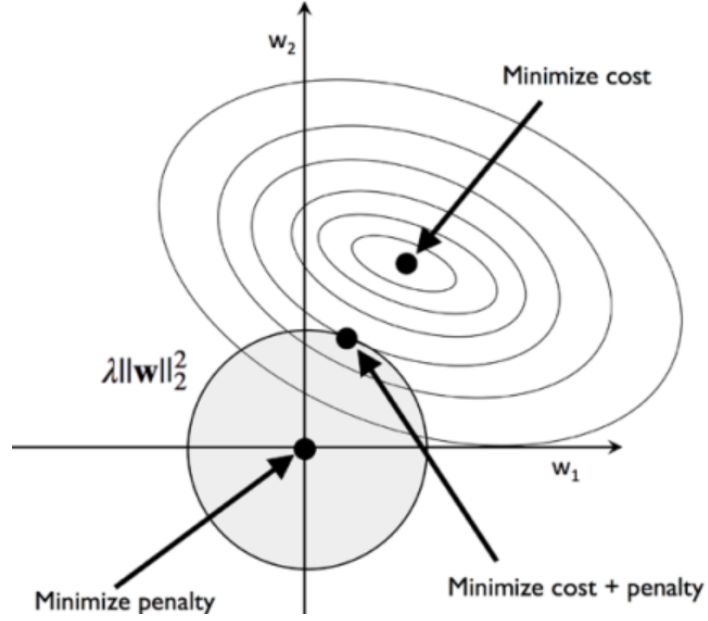


FIGURE A.1: Geometric Interpretation of the Ridge Regression (Raschka et al., 2019)

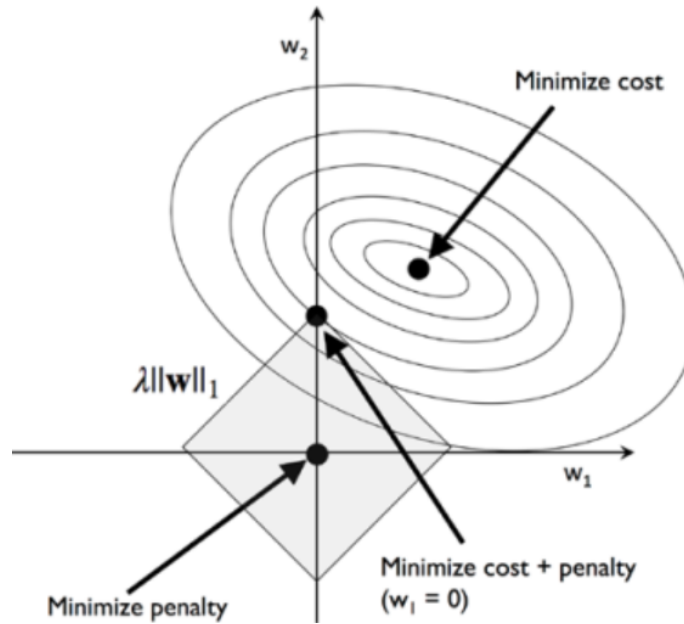


FIGURE A.2: Geometric Interpretation of the Lasso (Raschka et al., 2019)

A.2 Kernel Methods

This section is based on Andrew Ng's lecture note about kernel methods and SVM (2020), Bishop's book about machine learning (2007). It is a basis for some techniques in machine learning, such as Support Vector Machine (SVM) or Kernelised Principal Component Analysis (KPCA).

A.2.1 Feature Maps

Let's define $\phi : \mathbb{R}^d \mapsto \mathbb{R}^p$ be a feature space map that maps attribute $x \in \mathbb{R}^d$ to the features $\phi(x) \in \mathbb{R}^p$. Let's demonstrate on the BGD and SGD, which are all based on LMS. Gradient descent algorithm is constructed for fitting the model $\theta^T \phi(x)$. The equivalent forms of the BGD (A.5) and SGD (A.7) respectively are

$$\theta := \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)}) x^{(i)} \quad (\text{A.27})$$

$$\theta := \theta + \alpha (y^{(i)} - \theta^T x^{(i)}) x^{(i)}. \quad (\text{A.28})$$

Replacing $x^{(i)}$ in the algorithms above with $\phi(x^{(i)})$ provides us

$$\theta := \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)}) \quad (\text{A.29})$$

$$\theta := \theta + \alpha (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)}). \quad (\text{A.30})$$

A.2.2 Kernel Trick

Let's define the *kernel function* $K : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ corresponding to the feature map ϕ as

$$k(x, z) = \phi(x)^T \phi(z) = \langle \phi(x), \phi(z) \rangle, \quad (\text{A.31})$$

where $k(x, z) \geq 0$ and symmetric.

The kernel trick is now demonstrated on the cost function of the Ridge Regression. Equation (6.10) could be written as follows using the feature map ϕ :

$$J_{\text{Ridge}}(\theta) = \frac{1}{2} \sum_{i=1}^n (\theta^T \phi(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2} \theta^T \theta, \quad (\text{A.32})$$

where $\lambda \geq 0$. Setting $\nabla_{\theta} J(\theta) = 0$ gives us

$$\hat{\theta} = -\frac{1}{\lambda} \sum_{i=1}^n (\theta^T \phi(x^{(i)}) - y^{(i)}) \phi(x^{(i)}) = \sum_{i=1}^n \beta_i \phi(x^{(i)}) = \Phi^T \beta, \quad (\text{A.33})$$

where Φ is the design matrix, whose n^{th} row is given by $\phi(x^{(i)})^T$. It can be easily recognised that $\hat{\theta}$ is a linear combination of the vectors $\phi(x^{(i)})$ with coefficients β that are functions of θ . The vector $\beta = (\beta_1, \dots, \beta_n)$ is defined as

$$\beta_i = -\frac{1}{\lambda} (\theta^T \phi(x^{(i)}) - y^{(i)}). \quad (\text{A.34})$$

Instead of using the parameter θ in the cost function, it can be expressed in terms of the vector β , which gives space for dual representation. Plugging $\theta = \Phi^T \beta$ into $J(\theta)$ provides us

$$J_{Ridge}(\beta) = \frac{1}{2} \beta^T \Phi \Phi^T \Phi \Phi^T \beta - \beta^T \Phi \Phi^T y + \frac{1}{2} y^T y + \frac{\lambda}{2} \beta^T \Phi \Phi^T \beta, \quad (\text{A.35})$$

where $y = (y^{(1)}, \dots, y^{(n)})$. Let's define the *Gram matrix* $K = \Phi \Phi^T$, which is an $d \times d$ symmetric matrix with elements

$$K_{ij} = \phi(x^{(i)})^T \phi(x^{(j)}) = k(x^{(i)}, x^{(j)}), \quad (\text{A.36})$$

where the right-hand side of the equality is the kernel function defined at (A.31). The (A.35) cost function can be rewritten using the Gram matrix in the following way:

$$J_{Ridge}(\beta) = \frac{1}{2} \beta^T K K \beta - \beta^T K y + \frac{1}{2} y^T y + \frac{\lambda}{2} \beta^T K \beta. \quad (\text{A.37})$$

Setting $\nabla_{\beta} J(\beta) = 0$ gives us

$$\hat{\beta} = (K + \lambda I)^{-1} y. \quad (\text{A.38})$$

Substituting this back to the hypothesis function of the Ridge Regression model we get the following prediction for a new input x

$$h(x) = \theta^T \phi(x) = \beta^T \Phi \phi(x) = k(x)^T (K + \lambda I)^{-1} y, \quad (\text{A.39})$$

where the vector $k(x)$ is defined with elements $k_n(x) = k(x_n, x)$. Or equivalently

$$h(x) = \theta^T \phi(x) = \sum_{i=1}^n \beta_i \phi(x^{(i)})^T \phi(x) = \sum_{i=1}^n \beta_i k(x^{(i)}, x). \quad (\text{A.40})$$

The idea is if having an algorithm devised in a way that the input vector x enters only in the form of inner product, that inner product can be substituted for some kernel function. So instead of calculating with the multidimensional ϕ , it is enough to know its existence, because k encapsulates the corresponding ϕ .

Some well-known kernels

- Linear Kernel

$$k(x, z) = x^T z \quad (\text{A.41})$$

- Polynomial Kernel

$$k(x, z) = (\delta x^T z + r)^m \quad (\text{A.42})$$

- Gaussian Kernel or Radial Basis Function (RBF) Kernel

$$k(x, z) = -\exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right) \quad (\text{A.43})$$

These kernels could be used in the context of Support Vector Regression (SVR). The question is how we know whether the kernel is valid or not. The Mercer theorem would ensure the validity of a kernel.

Necessary and sufficient conditions for kernel validity

Theorem A.2.1 (Mercer). *Let $K : \mathbb{R}^d \mapsto \mathbb{R}^d$ be given. For K to be a valid / Mercer kernel, it is necessary and sufficient that for any $\{x^{(1)}, \dots, x^{(n)}\}$ ($n < \infty$), the corresponding kernel matrix is symmetric and positive semi-definite.*

A.3 Support Vector Regression

This section is based on Murphy's (2012) book.

As mentioned in Section 6.4, SVR is an ϵ -sensitive model. To this end, the following *epsilon insensitive loss function* is defined as (Vapnik et al., 1997)

$$L_\epsilon(y, \hat{y}) = \begin{cases} 0 & \text{if } |y - \hat{y}| < \epsilon \\ |y - \hat{y}| - \epsilon & \text{otherwise,} \end{cases} \quad (\text{A.44})$$

where let $\hat{y} = h(x)$. In other words, this loss function represents the fact that if any point lying inside an ϵ -tube (street), the prediction is not penalised.

The corresponding objective function is

$$J(\theta) = C \sum_{i=1}^n L_\epsilon(y^{(i)}, \hat{y}^{(i)}) + \frac{1}{2} \|\theta\|^2, \quad (\text{A.45})$$

where $C = 1/\lambda$ is a regularisation constant. The objective function is convex and unconstrained, but not differentiable, owing to the absolute value function in the loss term. One way to unlock this problem is to redefine the objective as a constrained optimisation problem. Let's introduce *slack variables* to depict the degree to which each point lies outside the ϵ -tube :

$$y^{(i)} \leq \hat{y}^{(i)} + \epsilon + \xi_i^+ \quad (\text{A.46})$$

$$y^{(i)} \geq \hat{y}^{(i)} - \epsilon - \xi_i^- \quad (\text{A.47})$$

By using them, the objective can be rewritten as follows:

$$J(\theta) = C \sum_{i=1}^n (\xi_i^+ + \xi_i^-) + \frac{1}{2} \|\theta\|^2, \quad (\text{A.48})$$

where ξ_i^+ and ξ_i^- are non-negative constants. With this knowledge, the optimisation problem can be formulated as:

$$\begin{aligned} \min_{\theta} J(\theta) &= \min_{\theta} C \sum_{i=1}^n (\xi_i^+ + \xi_i^-) + \frac{1}{2} \|\theta\|^2 \\ \text{s.t. } & y^{(i)} \leq \hat{y}^{(i)} + \epsilon + \xi_i^+ \\ & y^{(i)} \geq \hat{y}^{(i)} - \epsilon - \xi_i^- \\ & \xi_i^+ \geq 0 \\ & \xi_i^- \geq 0 \end{aligned} \quad (\text{A.49})$$

It is a standard quadratic program in $2n + d + 1$ variables. Shoelkopf and Smola (2002) show that the optimal solution is in the form of

$$\hat{\theta} = \sum_{i=1}^n \beta_i x^{(i)}, \quad (\text{A.50})$$

where $\beta_i \geq 0$. It can be proved that β is a sparse vector (that is a vector with a lot of 0 elements) because errors smaller than ϵ are ignored. The $x^{(i)}$ vectors for which $\beta_i > 0$ are called the *support vectors*. These form the street for which errors lie on or outside the ϵ tube.

After a model is trained, the prediction made by the usual hypothesis function formula, that is:

$$h(x) = \hat{y}(x) = \hat{\theta}^T x \quad (\text{A.51})$$

Plugging $\hat{\theta}$ determined in the formula (A.50) into the hypothesis function gives us

$$h(x) = \hat{y}(x) = \sum_{i=1}^n \beta_i \langle x^{(i)}, x \rangle. \quad (\text{A.52})$$

Replacing $\langle x^{(i)}, x \rangle$ with $k(x^{(i)}, x)$ to get a kernelised solution

$$h(x) = \hat{y}(x) = \sum_{i=1}^n \beta_i k(x^{(i)}, x). \quad (\text{A.53})$$

Appendix B

Dimensionality Reduction

B.1 Feature Selection

B.1.1 Filter Methods

It is usually applied as a preprocessing step as it has been done in Section 5.4 with Pearson's correlation before any model building. Despite being less accurate, it is fast compared to the other two methods. Usually, various statistical tests are performed, and the features selected for model building are based on the resulting statistical scores. Some examples of filter methods include:

| Feature\Response | Continuous | Categorical |
|------------------|-----------------------|-------------|
| Continuous | Pearson's Correlation | LDA |
| Categorical | ANOVA | χ^2 |

TABLE B.1: Filter methods

These are the primary filter methods even though some other statistical tests related to this method exist.

Pearson's Correlation

This method is applied when quantifying linear dependence between two continuous variables is the question to answer. The possible range of values is from -1 to 1 , where -1 means perfect negative correlation, 0 stands for uncorrelatedness, and 1 represents a perfect positive correlation. It is defined by (Blitzstein et al., 2019):

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma_x \sigma_y} \quad (\text{B.1})$$

where X and Y are random variables, $\sigma_x \sigma_y$ are the variance of X and Y , respectively. $\text{Cov}(X, Y)$ is the covariance of X and Y defined by:

$$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])] \quad (\text{B.2})$$

B.1.2 Wrapper methods

In these methods, a specific ML algorithm fitting on a given dataset determines the feature selection process. Wrapper methods are a family of greedy search algorithms in which possible combinations of attributes are tested against the evaluation criterion. The evaluation criterion is the performance measure, which depends on the type of ML task. For regression tasks, the evaluation criterion is usually the RMSE,

but it could be R2, p-values, MAE, and so on. In the end, it selects the combination of attributes giving the optimal result for the appropriate ML algorithm (Iuhaniwal, 2019). Some well-known wrapper methods are sequential forward (floating) selection, sequential backward (floating) selection, exhaustive selection. Now, only sequential backward floating selection (SBFS) is discussed at some length. The floating version is just an extension to the SFS and SBS. The floating algorithms have an additional exclusion or inclusion step to remove features once they were included (or excluded) so that a larger number of feature subset combinations can be sampled. It is important to emphasize that this step is conditional and only occurs if the resulting feature subset is assessed as "better" by the criterion function after removal (or addition) of a particular feature. (Raschka, 2020)

The algorithm of the SBFS

The algorithm proposed below is based on Raschka's mlexend package about Sequential Feature Selection (2020)

Input: the set of all features, $Y = y_1, y_2, \dots, y_m$

Output: $X_k = x_j \mid j = 1, 2, \dots, k; x_j \in Y$, where $k = (0, 1, 2, \dots, m)$

Initialisation: $X_0 = Y, k = m$

Step 1 (Exclusion):

$$x^- = \operatorname{argmax} J(x_k - x), \text{ where } x \in X_k$$

$$X_{k-1} = X_k - x^-$$

$$k = k - 1$$

Go to Step 2

- x^- is the feature that maximizes our criterion function on removal, that is, the feature that is associated with the best regression performance it is removed from X_k .
- Repeat this procedure until the termination criterion is satisfied.

Step 2 (Conditional Inclusion): - (Extra step for the SBFS)

$$x^+ = \operatorname{argmax} J(x_k + x), \text{ where } x \in Y - X_k$$

If $J(X_k + x) > J(X_k)$:

$$X_{k+1} = X_k + x^+$$

$$k = k + 1$$

Go to Step 2

- In Step 2, we search for features that improve the classifier performance if they are added back to the feature subset. If such features exist, we add the feature x^+ for which the performance improvement is maximised. If $k = 2$ or an improvement cannot be made (i.e., such feature x^+ cannot be found), go back to step 1; else, repeat this step.

Termination: $k = p$

- We add features from the feature subset X_k until the feature subset of size k contains the number of desired features p that we specified a priori.

B.1.3 Embedded methods

Embedded methods integrate the feature selection process within the construction of the machine learning algorithm itself. That is, they perform feature selection during

the model training. They find the feature subset for the algorithm being trained. The advantage of this method is that it considers the relationships between features like in the case of wrapper methods but much faster than them and more accurate than filter methods. There are several examples which are shown in Chapter 6, such as regularisation based methods (Ridge/Lasso), or tree-based algorithms (Decision Tree, Random Forest, XGBoost).

B.2 Factor Analysis

There are several factor analysis techniques. In the project, **FAMD** is used, which is the combination of PCA and MCA. In the following section, only the mathematics behind the PCA is detailed.

B.2.1 PCA

This section is based on the *Mathematics for Machine Learning* by Deisenroth et al. (2020). There are several version of PCA such as Probabilistic PCA (PPCA), Kernel PCA (KPCA), but here only the vanilla PCA is discussed.

Principal Component Analysis (PCA) is a linear dimensionality reduction method proposed by Pearson and Hotelling and mainly used for features extractions and dimensionality reduction. It projects the data to the hyperplane that is closest to the data by identifying the axis that preserves the greatest amount of variance in the training set. Finding projections \tilde{x}_n of data points x_n that are as similar to the original data points as possible with a considerably lower intrinsic dimensionality is our chief goal.

Consider an i.i.d. dataset $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ ($\mathbf{x}_n \in \mathbb{R}^D$), with mean $\mathbf{0}$ and the corresponding covariance matrix (For more details, please consult the **Remark** section):

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^T. \quad (\text{B.3})$$

Now assume that there exists a low-dimensional compressed representation (code)

$$\mathbf{z}_n = \mathbf{B}^T \mathbf{x}_n \in \mathbb{R}^M \quad (\text{B.4})$$

of \mathbf{x}_n , where the projection matrix \mathbf{B} is defined as follow:

$$\mathbf{B} := [\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_M] \in \mathbb{R}^{D \times M} \quad (\text{B.5})$$

As we know the columns of \mathbf{B} form a basis of the M -dimensional subspace in which the projected data lie in $\tilde{\mathbf{x}} = \mathbf{B} \mathbf{B}^T \mathbf{x} \in \mathbb{R}^D$. Assume that these columns of \mathbf{B} are orthonormal. M -dimensional subspace $U \subseteq \mathbb{R}^D$ ($\dim(U) = M < D$) is looked for, where data is projected onto this U subspace. The projected data $\tilde{\mathbf{x}}_n \in U$ and their coordinates (with respect to the basis vector of $\mathbf{b}_1, \dots, \mathbf{b}_M$ of U) are \mathbf{z}_n . The goal is to find $\tilde{\mathbf{x}} \in \mathbb{R}^D$, or in other words, the codes \mathbf{z}_n and the basis vectors $\mathbf{b}_1, \dots, \mathbf{b}_M$. As a result, they would be similar to the original data \mathbf{x}_n and minimize the loss because of the compression. \mathbf{z} is the lower-dimensional representation of the compressed data $\tilde{\mathbf{x}}$, which is in the original data space but has an intrinsic lower-dimensional representation than \mathbf{x} . \mathbf{z} also directs the amount of information flowing between \mathbf{x} and $\tilde{\mathbf{x}}$. In PCA, for a suitable linear mapping \mathbf{B} , $\mathbf{z} = \mathbf{B}^T \mathbf{x}$ and $\tilde{\mathbf{x}} = \mathbf{B} \mathbf{z}$ are the linear relationship between \mathbf{z} and \mathbf{x} , and between $\tilde{\mathbf{x}}$ and \mathbf{z} , respectively. \mathbf{B} acts as a decoder

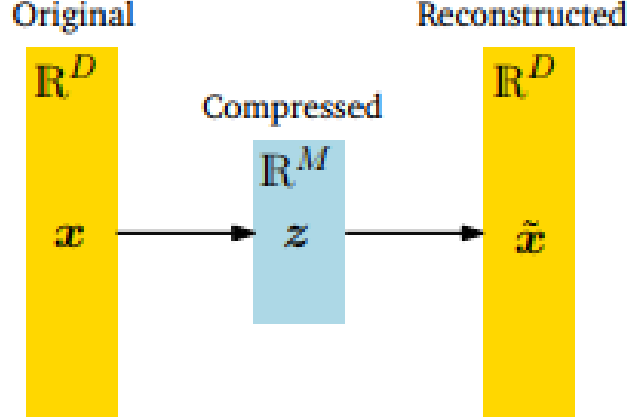


FIGURE B.1: PCA illustration

mapping the low-dimensional code $\mathbf{z} \in \mathbb{R}^M$ back into the original data space \mathbb{R}^D . \mathbf{B}^T encodes the original data \mathbf{x} as a low-dimensional, compressed code \mathbf{z} .

Remark

In Equation (B.3), we assumed centred data with μ mean:

$$\text{Var}[\mathbf{z}] = \text{Var}[\mathbf{B}^T(\mathbf{x} - \mu)] = \text{Var}[\mathbf{B}^T\mathbf{x} - \mathbf{B}^T\mu] = \text{Var}[\mathbf{B}^T\mathbf{x}] \quad (\text{B.6})$$

In other words, the variance of the low dimension code \mathbf{z} does not depend on the mean of the data. So without the loss of generality, data with $\mu = \mathbf{0}$ can be assumed for the rest of this section. Given this knowledge, the expectation value of \mathbf{z} is:

$$E[\mathbf{z}] = E[\mathbf{B}^T\mathbf{x}] = \mathbf{B}^T E[\mathbf{x}] = \mathbf{0} \quad (\text{B.7})$$

B.2.2 Direction with Maximal Variance

The aim is to maximise the variance of the low-dimensional code using a sequential approach. Let's start by searching for a vector $\mathbf{b}_1 \in \mathbb{R}^D$ maximising the variance of the projected data. That is, maximise the variance of the first coordinate z_1 of $\mathbf{z} \in \mathbb{R}^D$:

$$V_1 := \text{Var}[z_1] = \frac{1}{N} \sum_{n=1}^N z_{1n}^2 \quad (\text{B.8})$$

is maximised, where i.i.d assumption of the data and z_{1n} are harnessed (the first coordinate of the low-dimensional representation $\mathbf{z}_n \in \mathbb{R}^M$ of $\mathbf{x}_n \in \mathbb{R}^D$). The first component of \mathbf{z}_n is z_{1n} , and it is the coordinate of the orthogonal projection of \mathbf{x}_n onto the one-dimensional subspace spanned by \mathbf{b}_1 :

$$z_{1n} = \mathbf{b}_1^T \mathbf{x}_n \quad (\text{B.9})$$

Substituting Equation (B.9) into Equation (B.8) gives us

$$V_1 = \frac{1}{N} \sum_{n=1}^N (\mathbf{b}_1^T \mathbf{x}_n)^2 = \frac{1}{N} \sum_{n=1}^N \mathbf{b}_1^T \mathbf{x}_n \mathbf{x}_n^T \mathbf{b}_1 \quad (\text{B.10})$$

$$= \mathbf{b}_1^T \left(\frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^T \right) \mathbf{b}_1 = \mathbf{b}_1^T \mathbf{S} \mathbf{b}_1. \quad (\text{B.11})$$

It is vital to normalize \mathbf{b}_1 , that is $\|\mathbf{b}_1\| = 1$, otherwise increase in \mathbf{b}_1 would lead to increase V_1 in the factor of a square. With this restriction, the following constrained optimisation problem is proposed in terms of \mathbf{b}_1 , which points in the direction of maximum variance:

$$\begin{aligned} \max_{\mathbf{b}_1} V_1 &= \max_{\mathbf{b}_1} \mathbf{b}_1^T \mathbf{S} \mathbf{b}_1 \\ \text{s.t. } \|\mathbf{b}_1\|^2 &= 1 \end{aligned} \quad (\text{B.12})$$

We could write this problem in terms of Lagrange multipliers to solve the optimisation problem:

$$\mathcal{L}(\mathbf{b}_1, \lambda) = \mathbf{b}_1^T \mathbf{S} \mathbf{b}_1 + \lambda_1(1 - \mathbf{b}_1^T \mathbf{b}_1) \quad (\text{B.13})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} = 2\mathbf{b}_1^T \mathbf{S} - 2\lambda_1 \mathbf{b}_1^T, \quad \frac{\partial \mathcal{L}}{\partial \lambda_1} = 1 - \mathbf{b}_1^T \mathbf{b}_1 \quad (\text{B.14})$$

Setting these to 0 yields us:

$$\mathbf{S} \mathbf{b}_1 = \lambda_1 \mathbf{b}_1 \quad (\text{B.15})$$

$$\mathbf{b}_1^T \mathbf{b}_1 = 1 \quad (\text{B.16})$$

From Equation (B.15), \mathbf{b}_1 is an eigenvector of the data covariance matrix \mathbf{S} , and the Lagrange multiplier λ_1 is the corresponding eigenvalue. With the help of Equation (B.15) and the optimisation problem (B.12), Equation (B.8) could be written as follow:

$$V_1 = \mathbf{b}_1^T \mathbf{S} \mathbf{b}_1 = \lambda_1 \mathbf{b}_1^T \mathbf{b}_1 = \lambda_1 \quad (\text{B.17})$$

What this rewritten variance objective represents is the variance of the data projected onto a one-dimensional subspace equals the eigenvalue that is associated with the basis vector b_1 that spans this subspace. To maximise the variance of the low-dimensional code, the basis vector associated with the largest eigenvalue of the data covariance matrix should be chosen. This eigenvector \mathbf{b}_1 is called the first **principal component**. In order to decode z_{1n} coordinate back to the original data space, which provides us with the projected data point, the following process is necessary:

$$\tilde{\mathbf{x}}_n = \mathbf{b}_1 z_{1n} = \mathbf{b}_1 \mathbf{b}_1^T \mathbf{x}_n \in \mathbb{R}^D \quad (\text{B.18})$$

B.2.3 M-dimensional Subspace with Maximal Variance

Let's suppose that the first $m - 1$ principal components have found that is the $m - 1$ eigenvectors of \mathbf{S} that are associated with the largest $m - 1$ eigenvalues. Because of the symmetric matrix \mathbf{S} , the spectral theorem can be applied.

Theorem B.2.1 (Spectral Theorem). *If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is symmetric, there exists an orthonormal basis of the corresponding vector space V consisting of eigenvectors of \mathbf{A} , and each eigenvalue is real.*

For our case, it means that an orthonormal eigenbasis of an $(m - 1)$ -dimensional subspace of \mathbb{R}^D can be built by the eigenvectors of the symmetric matrix \mathbf{S} . By and large, the m th principal component can be calculated by subtracting the effect of the first $m - 1$ principal components $\mathbf{b}_1, \dots, \mathbf{b}_{m-1}$ from the data. As a result of that, the

aim is to find principal components which compress the rest of the information.

$$\tilde{\mathbf{X}} := \mathbf{X} - \sum_{i=1}^{m-1} \mathbf{b}_i \mathbf{b}_i^T \mathbf{X} = \mathbf{X} - \mathbf{B}_{m-1} \mathbf{X}, \quad (\text{B.19})$$

where $\tilde{\mathbf{X}} := [\tilde{x}_1, \dots, \tilde{x}_N] \in \mathbb{R}^{D \times N}$ contains the information in the data that has not yet been compressed, and \mathbf{B}_{m-1} matrix projects onto the subspace spanned by $\mathbf{b}_1, \dots, \mathbf{b}_{m-1}$.

To find the m th principal component the following corresponding variance has to be **maximised** in a similar way as in Equation B.12

$$V_m := \text{Var}[z_m] = \frac{1}{N} \sum_{n=1}^N (\mathbf{b}_m^T \tilde{\mathbf{x}}_n)^2 = \mathbf{b}_m^T \tilde{\mathbf{S}} \mathbf{b}_m \quad (\text{B.20})$$

s.t. $\|\mathbf{b}_m\|^2 = 1,$

where $\tilde{\mathbf{S}}$ is the data covariance matrix of the transformed dataset $\tilde{\chi} := \{\tilde{x}_1, \dots, \tilde{x}_N\}$. Analogously, after solving this constrained optimisation problem, it turns out that the optimal solution \mathbf{b}_m is the eigenvector of $\tilde{\mathbf{S}}$ that is associated with the largest eigenvalue of $\tilde{\mathbf{S}}$. Moreover, \mathbf{b}_m is also an eigenvector of \mathbf{S} . It can also be shown that the sets of eigenvectors of \mathbf{S} and $\tilde{\mathbf{S}}$ are identical. However, if the eigenvectors of \mathbf{S} are part of the $(m-1)$ dimensional principal subspace (the case of $i < m$), the associated eigenvalue of $\tilde{\mathbf{S}}$ is 0. In other words,

$$\tilde{\mathbf{S}} \mathbf{b}_i = 0 \mathbf{b}_i, \quad (\text{B.21})$$

that is, $\mathbf{b}_1, \dots, \mathbf{b}_{m-1}$, which are also eigenvectors of $\tilde{\mathbf{S}}$, span the null space of $\tilde{\mathbf{S}}$ because of being associated with the eigenvalue 0. If an eigenvector that is not among the first $m-1$ principal component (the case of $i \geq m$) it is orthogonal to the first $m-1$ principal components, and $\tilde{\mathbf{S}} \mathbf{b}_i = \lambda_i \mathbf{b}_i$. Suppose the m th is not among first $m-1$ principal component, then

$$\tilde{\mathbf{S}} \mathbf{b}_m = \mathbf{S} \mathbf{b}_m = \lambda_m \mathbf{b}_m. \quad (\text{B.22})$$

This \mathbf{b}_m is an eigenvector of both $\tilde{\mathbf{S}}$ and \mathbf{S} . Furthermore, λ_m is the largest eigenvalue of $\tilde{\mathbf{S}}$ and the m th largest eigenvalue of \mathbf{S} , and their corresponding eigenvector is \mathbf{b}_m . Using Equation B.22 and $\mathbf{b}_m^T \mathbf{b}_m = 1$, the variance of the data projected onto the m th principal component is

$$V_m = \mathbf{b}_m^T \mathbf{S} \mathbf{b}_m = \lambda_m \mathbf{b}_m^T \mathbf{b}_m = \lambda_m. \quad (\text{B.23})$$

To find an M -dimensional subspace of \mathbb{R}^D retaining as much information as possible, we need to add up the eigenvalues associated with the first M principal components (those eigenvalues are also the M largest eigenvalues at the same time) of the data covariance matrix \mathbf{S} to reach the maximum amount of variance the PCA can exploit.

$$V_M = \sum_{m=1}^M \lambda_m. \quad (\text{B.24})$$

Therefore, the variance lost by data compression via PCA is

$$J_M := \sum_{j=M+1}^D \lambda_j = V_D - V_M. \quad (\text{B.25})$$

The alternative variance lost measurement is the relative variance/explained variance ratio which is given in the form of

$$\text{explained variance ratio} = \frac{V_M}{V_D}. \quad (\text{B.26})$$

Appendix C

Python Codes

The most important code snippets are shown here. For the complete implementation, please consult my project implementation on [my GitHub page](#).

C.1 Codes from Section 5.2

```
def num_var_transformer(imputed_num, full_pipeline, strategy):
    if strategy == 'imputer':
        num_pipeline = Pipeline([('imputer', SimpleImputer(strategy="median"))])
    elif strategy == 'y-j':
        num_pipeline = Pipeline([('y-j',
                                   PowerTransformer(method='yeo-johnson'))])

    num_attr = list(imputed_num)

    if full_pipeline is None:
        full_pipeline = ColumnTransformer([
            ("num", num_pipeline, num_attr)])

    fittedColumnTransform = full_pipeline.fit(imputed_num)
    imputed_num_array = fittedColumnTransform.transform(imputed_num)
    imputed_num_ready = pd.DataFrame(imputed_num_array, columns =
                                     imputed_num.columns, index = imputed_num.index)

    if (strategy == 'imputer') and ('number_of_reviews' not in dropList):
        imputed_num_ready['months'] = imputed_num_ready['number_of_reviews']
        / imputed_num_ready['reviews_per_month']
        imputed_num_ready.drop(['number_of_reviews'], axis=1, inplace=True)

    else:
        fittedColumnTransform = full_pipeline
        imputed_num_array = fittedColumnTransform.transform(imputed_num)
        imputed_num_ready = pd.DataFrame(imputed_num_array, columns =
                                         imputed_num.columns, index = imputed_num.index)

        if (strategy == 'imputer') and ('number_of_reviews' not in dropList):
            imputed_num_ready['months'] = imputed_num_ready['number_of_reviews']
            / imputed_num_ready['reviews_per_month']
            imputed_num_ready.drop(['number_of_reviews'], axis=1, inplace=True)

    return imputed_num_ready, fittedColumnTransform
```

TABLE C.1: Numerical Variable Transformation


```

def target_var_transformer(imputed_target, pT):
    if pT is None:
        pT = preprocessing.PowerTransformer(method='yeo-johnson') #transforming
        the target variable
        targetTransform = pT.fit(imputed_target)
        imputed_target_ready_array = targetTransform.transform(imputed_target)
        imputed_target_ready = pd.DataFrame(imputed_target_ready_array, columns
        = imputed_target.columns, index = imputed_target.index)

    else:
        targetTransform = pT
        imputed_target_ready_array = targetTransform.transform(imputed_target)
        imputed_target_ready = pd.DataFrame(imputed_target_ready_array, columns
        = imputed_target.columns, index = imputed_target.index)

    return imputed_target_ready, targetTransform

```

TABLE C.2: Target Variable Transformation

```

def concat_num_cat_var(imputed, imputed_num, desired_cat_attr):
    temp_cat_df = imputed[desired_cat_attr]
    imputed_concat = copy.deepcopy(imputed_num)

    imputed_concat = pd.concat([imputed_concat, temp_cat_df], axis = 1)

    return imputed_concat

```

TABLE C.3: Concatenation of numerical and categorical features after one hot encoding categorical variables

C.2 Codes from Section 6.1

```
#####Train FAMD fit and transform
import prince

dropList = ['name', 'host_name', 'last_review', 'neighbourhood', 'host_id',
            'id']
desired_cat_attr = ['room_type', 'neighbourhood_group']
#desired_cat_attr = ['room_type']
nycAB_ready, nycAB_imputed_target, fittedColumnTransform1_tr_dev,
    fittedColumnTransform2_tr_dev, targetTransform_tr_dev =
    df_proccessed(nycAB, nycAB_target, dropList, desired_cat_attr,
        full_pipeline1 = None, full_pipeline2 = None, pT = None)

fitTransfromFAMD_tr_dev = prince.FAMD(n_components = 10, n_iter=10, engine
    = 'auto', random_state = randState).fit(nycAB_ready)
nycAB_ready = fitTransfromFAMD_tr_dev.transform(nycAB_ready)

sum(fitTransfromFAMD_tr_dev.explained_inertia_) ###explained variance is
    0.9511413328166206

#####Test FAMD fit transform
X_test_ready, Y_test_ready, _, _, _ = df_proccessed(X_test, Y_test,
    dropList, desired_cat_attr, full_pipeline1 =
    fittedColumnTransform1_tr_dev, full_pipeline2 =
    fittedColumnTransform2_tr_dev, pT = targetTransform_tr_dev)
X_test_ready = fitTransfromFAMD_tr_dev.transform(X_test_ready)
```

TABLE C.4: Factor Analysis of Mixed Data implementation

C.3 Codes from Section 6.8

```
svmGauss1 = SVR(kernel="rbf", C=2.0, gamma = 0.2, epsilon=0.5)
svmGauss1.fit(nycAB_ready, nycAB_imputed_target)

Y_trPred_svmGauss1 = svmGauss1.predict(nycAB_ready)
svmGauss1_trPred_mse = mean_squared_error(nycAB_imputed_target,
    Y_trPred_svmGauss1)
svmGauss1_trPred_rmse = np.sqrt(svmGauss1_trPred_mse)

print("SVM w/ Gaussian Kernel Train RMSE:", svmGauss1_trPred_rmse) ####
    0.6437572536668337
```

TABLE C.5: Support Vector Regression with Gaussian Kernel Training

```

svmGauss1_scores = cross_val_score(svmGauss1, nycAB_ready,
    nycAB_imputed_target, scoring="neg_mean_squared_error", cv=6)
svmGauss1_rmse_scores = np.sqrt(-svmGauss1_scores)

display_scores(svmGauss1_rmse_scores) ##### 0.65615193

```

TABLE C.6: Support Vector Regression with Gaussian Kernel Cross-Validation

C.4 Codes from Section 6.9

```

final_pred = svmGauss1.predict(X_test_ready)

final_mse = mean_squared_error(Y_test_ready, final_pred)
final_rmse = np.sqrt(final_mse)
print(final_rmse) ##### 0.6658913507277759

```

TABLE C.7: Support Vector Regression with Gaussian Kernel Training

Bibliography

Geron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (2nd Edition). O'Reilly Media, Incorporated.

Raschka, S., & Mirjalili, V. (2017). *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow* (2nd Edition). Packt Publishing.

Yeo, I., & Johnson, R. (2000). *A New Family of Power Transformations to Improve Normality or Symmetry*. *Biometrika*, 87(4), 954-959. <http://www.jstor.org/stable/2673623>

Box, G., & Cox, D. (1964). *An Analysis of Transformations*. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2), 211-252. <http://www.jstor.org/stable/2984418>

Weisberg, S. (2001). *Yeo-Johnson Power Transformations*. <https://www.stat.umn.edu/arc/yjpower.pdf>.

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (2nd Edition). O'Reilly Media

Deisenroth, M. P., Faisal, A. A., Ong, C. S. (2020). *Mathematics for Machine Learning*. Cambridge University Press. <https://mml-book.github.io/book/mml-book.pdf>

Pedregosa et al. (2011). *Scikit-learn: Machine Learning in Python*. *JMLR* 12, pp. 2825-2830.

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. (2020). *Array programming with NumPy*. *Nature* 585, 357–362

McKinney, W., & others. (2010). *Data structures for statistical computing in python*. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56).

Hunter, J.D. (2017). *Matplotlib: A 2D Graphics Environment*, *Computing in Science & Engineering*, 9, 90-95.

Raschka, S. (2020). *Sequential Feature Selector*. http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/

- Luhaniwal, V. (2019). *Feature selection using Wrapper methods in Python*. <https://towardsdatascience.com/feature-selection-using-wrapper-methods-in-python-f0d352b346f>
- K. Blitzstein, J., Hwang, J. (2019). *Introduction to Probability* (2nd Edition). <https://projects.iq.harvard.edu/stat110/home>
- Greenacre, M. (2007). *Correspondence Analysis in Practice*, (2nd Edition). Chapman & Hall/CRC Interdisciplinary Statistics
- Pages, J. (2014). *Multiple Factor Analysis by Example Using R*. Chapman & Hall/CRC The R Series
- Pages, J. (2011). *Factorial analysis of qualitative and quantitative data Factorial analysis of qualitative and quantitative data both mixed and structured according to a hierarchy*. Theory and Application of High-dimensional Complex and Symbolic Data Analysis in Economics and Management Science. Beijing, China October 27-29, 2011. <http://www.modulad.fr/sda11/HCSDA11-Pages.pdf>
- Ng, A. (2020). CS29 Lecture Notes 1: Supervised Learning. <http://cs229.stanford.edu/notes2020fall/notes2020fall/cs229-notes1.pdf>
- Ng, A. (2020). CS29 Lecture Notes 3: Kernel Methods and SVM. <http://cs229.stanford.edu/notes2020fall/notes2020fall/cs229-notes3.pdf>
- Hastie, T., Hastie, T., Tibshirani, R., & Friedman, J. H. (2001). *The elements of statistical learning: Data mining, inference, and prediction* (2nd Edition). New York: Springer. https://web.stanford.edu/~hastie/ElemStatLearn/printings/ESLII_print12_toc.pdf
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in R*. <http://faculty.marshall.usc.edu/gareth-james/ISL/ISLR%20Seventh%20Printing.pdf>
- Bishop, C. M. (2007). *Pattern Recognition and Machine Learning* (Information Science and Statistics). Springer.
- Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. Cambridge, MA: MIT Press.
- Vapnik, V., S. Golowich, and A. Smola (1997). *Support vector method for function approximation, regression estimation, and signal processing*. In NIPS
- Schoelkopf, B. and A. Smola (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.