ML2 Project

Eric Jonas, Caroline Graebel, Naim Iskandar Zahari, Kseniia Holovchenko

2025-01-04

Introduction

Understanding rent prices in Berlin is essential for anyone looking to rent an apartment in the city, given its dynamic housing market and the growing demand for affordable living spaces. To address this, our project focuses on analyzing historical rental data from Berlin in 2020, aiming to uncover insights and develop predictive models that estimate rental prices based on key property attributes and evaluate them on current apartment offers.

The following dataset, available at Kaggle, contains information about rental properties listed on ImmoScout24. The data was scraped from ImmoScout24 between February and October 2020 and includes listings from all German federal states. For our project, we will focus exclusively on Berlin and train a model to predict either the `totalRent` or the `basePrice`.

Data Description

The data provided in the Kaggle dataset consists of approximately 268,000 rental listings in Germany. In the dataset, the information from the rental listings are structured in a CSV file with 49 columns. The columns include:

- Geographical information (state, city name and zip code)
- Property information (construction year, flat type, heating type and number of rooms)
- Total rent, base rent and service charge (in Euros)

Objectives

- **Data Understanding and Cleaning**:
  Before training a model, it is essential to thoroughly understand and clean the data. To achieve this, we will provide a concise visual overview of the dataset to highlight its key characteristics and identify potential issues.
- **Predict Missing Values**:
  Because many columns in our data contain missing values, one of our goals is to impute these values and use the resulting data instead of the raw data.
- **Model Selection and Comparison**:
  We aim to train a series of tree-based models on the dataset to predict the `baseRent` of an apartment, starting from simple decision trees that can provide explainable predictions and gradually progressing to more complex models such as Random Forests. Additionally, we will compare the performance of these tree-based methods with Support Vector Machine (SVM) regression to determine which approach yields the best results.

Data Overview and Cleaning

Historical rental listing data for Berlin is crucial for achieving the objectives of this project. The ImmoScout24 dataset provides rental listings from all of Germany. To begin, we need to extract data specific to Berlin from the full dataset. This extracted data contains numerous missing values and redundant entries, making it essential to first thoroughly understand and clean the dataset before proceeding.

Data Preparation

After filtering for Berlin listings, more than 10,000 entries remain. The next step is to remove irrelevant fields from the dataset, which will help with the development of machine learning models in subsequent stages. Identifying and removing these irrelevant fields requires a thorough understanding of the dataset. A brief explanation of each column can be found on Kaggle. Below is a list of the fields removed, along with the reasons for their exclusion:

Column Name	Reason for Removal
regio1	Redundant, as it contains only "Berlin."
geo_bln	Redundant, as it contains only "Berlin."
geo_krs	Redundant, as it contains only "Berlin."
regio2	Redundant, as it contains only "Berlin."
street	Redundant information; contains the same data as `street_plain`, but of lower quality.
telekomTvOffer	Irrelevant, as all listings can offer this service.
telekomHybridUploadSpeed	Unnecessary, with mostly missing (NA) values.
scoutId	Unique ID, not useful for analysis.
yearConstructedRange	Artificial column, not relevant to the analysis.
streetPlain	Would require significant cleaning or conversion into longitude and latitude.
houseNumber	Requires a corresponding street to provide meaningful context.
baseRentRange	Artificial column, not needed for analysis.
livingSpaceRange	Artificial column, not needed for analysis.
noRoomsRange	Artificial column, not needed for analysis.
electricityKwhPrice	Deprecated since 2020; irrelevant as the dataset lacks earlier data.
electricityBasePrice	Deprecated since 2020; irrelevant.
energyEfficiencyClass	Deprecated since 2020; irrelevant.
date	Redundant, as it duplicates `date_full`.
description	Redundant, as it only repeats information already provided in other columns.
facilities	Redundant, as it only repeats information already provided in other columns.
totalRent	Artificial column, a sum of baseRent, heatingCosts, and serviceCharge; unnecessary for analysis.
newlyConst	Artificial column, constructionYear >= 2017 ? TRUE : FALSE

Upon removing the irrelevant fields from the Berlin dataset, the remaining fields include:

Data Analysis

To understand the contents of the columns in our dataset and how they interact, we visualize them. We will focus on how the districts are distributed across our dataset and how attributes, such as the `baseRent`, are distributed within them. Additionally, we will examine the apartments in our dataset and investigate whether there are any correlations present.

District-Level Apartment Overview and Model Considerations

As seen in the data overview chapter, our dataset does not include the districts directly but only the postal codes (`geo_plz`). To derive the districts, we join our dataset with a reference dataset that maps Berlin's postal codes to districts. It is important to note that some postal codes belong to multiple districts. For simplicity, we associate each postal code with only one district. This simplification introduces a bias towards districts which appear earlier in our data, The following figure displays the frequency of each district represented in our dataset.

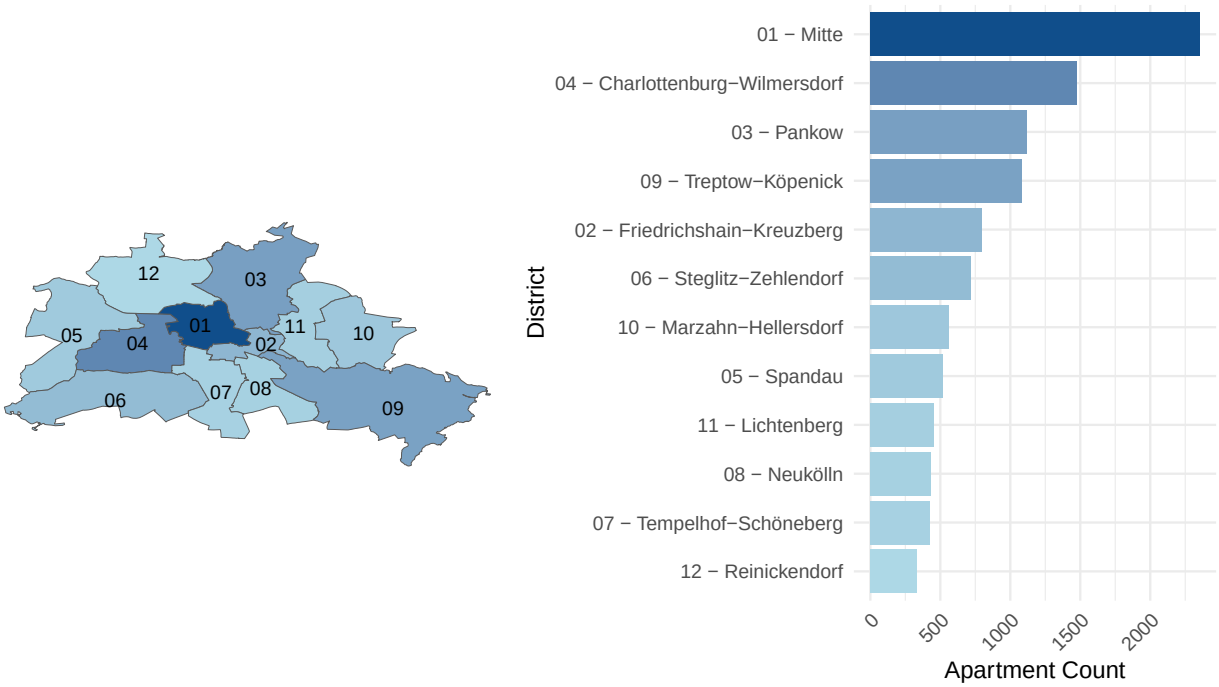

Figure 1: Apartment Count by District in Berlin

We observe that districts in central and eastern parts of Berlin are more frequently represented in the dataset compared to those in the southern or peripheral areas. Based on this we will evaluate later whether our model is bias towards districts that appear more frequently.

Furthermore, we analyzed the median prices across different districts. Our initial assumption was that districts closer to the city center have a higher median price compared to those in the outskirts. This assumption is based on the knowledge that central areas typically offer superior infrastructure and greater access to amenities and activity hubs.

Our initial assumption is confirmed: districts closer to the city center generally have higher rental prices, with Mitte standing out with several significant outliers. These outliers could pose challenges for our model if it lacks the robustness to accommodate them. Interestingly, when comparing this plot with the earlier analysis of apartment counts per district, the correlation between rental prices and the number of apartments appears weak. For example, Treptow-Köpenick, despite having a high number of apartments, shows relatively low rental prices.

Analysis of Rental Characteristics

The next step in understanding our dataset is to analyze characteristics of the apartments. This gives us an initial insight into the typical apartment types in our data and helps identify anomalies. First, we examine the distribution of the number of rooms across the apartments.

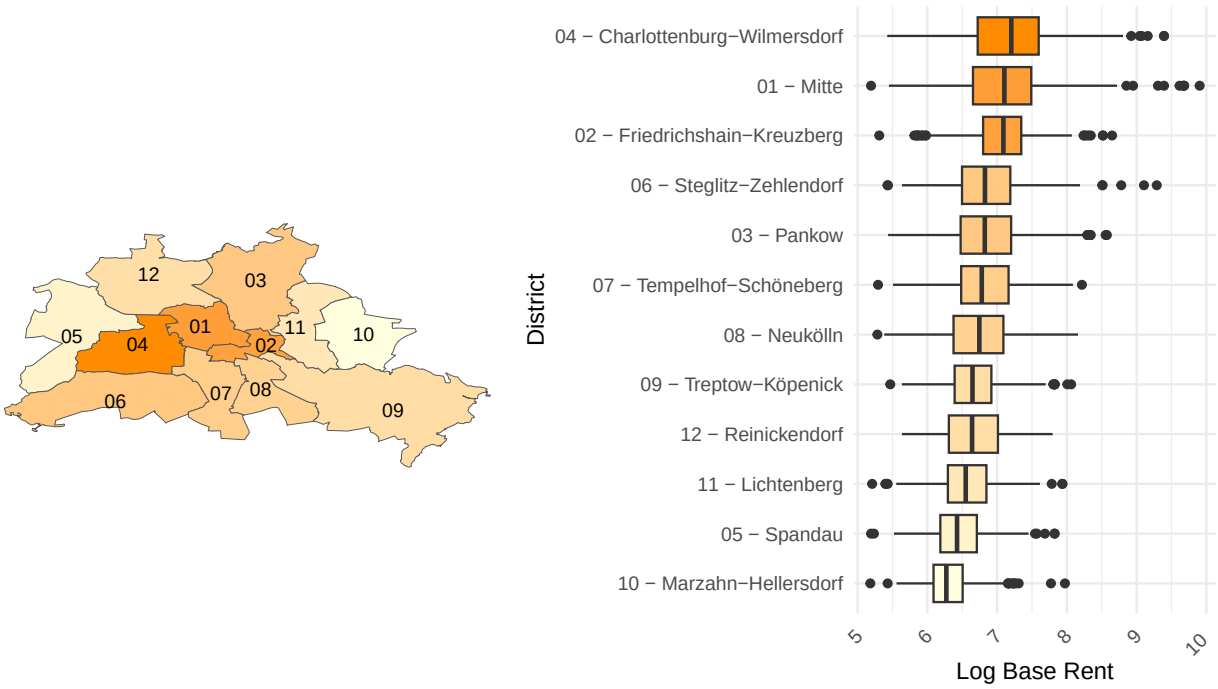

Figure 2: Median Price by District in Berlin

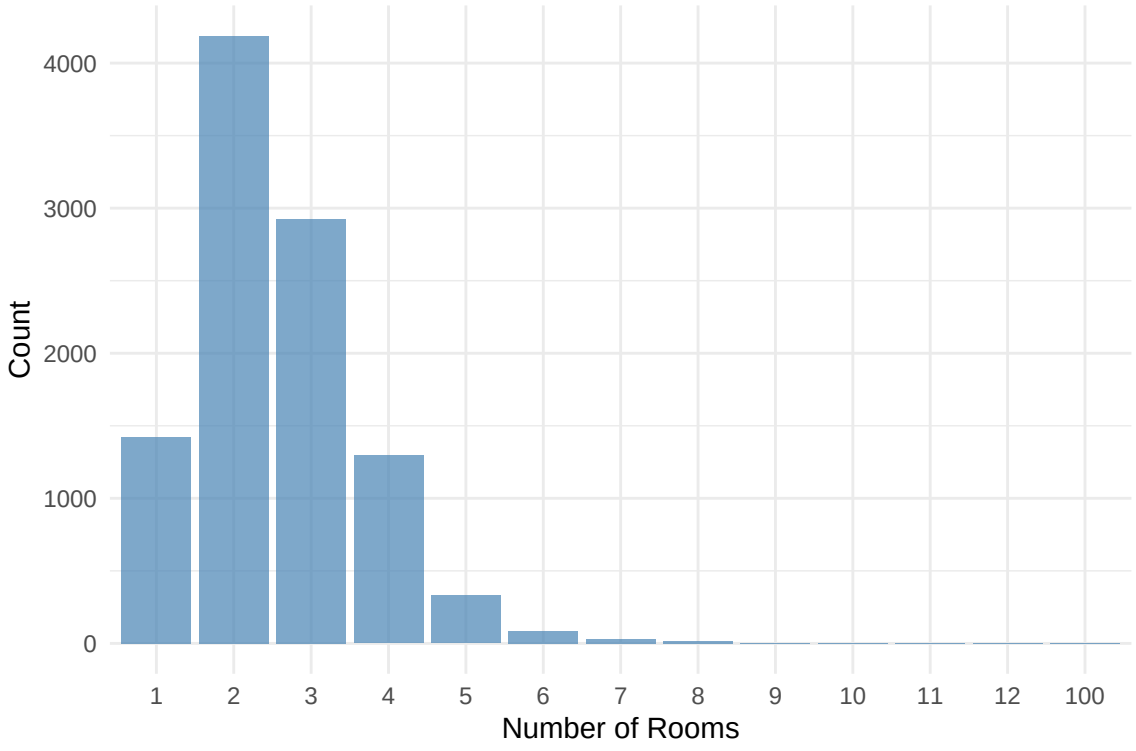

Figure 3: Count of Apartments by Number of Rooms

From the distribution, it is evident that the vast majority of apartments have a reasonable number of rooms, ranging between 1 and 8. However, there is one significant outlier, an apartment with an astonishing 100 rooms. While most of the data falls within expected ranges, such an outlier could have a disproportionate influence on our analysis and modeling.

One of our hypotheses about this dataset is that there should be a strong positive correlation between the number of rooms and the living space. Intuitively, as the number of rooms increases, the total living space is also expected to increase. Similarly, for smaller apartments, fewer rooms should correspond to smaller living spaces.
To visualize the relationship between living space and the number of rooms, we decided to exclude room counts that appear fewer than 10 times in the following visualization. This threshold ensures that the data we analyze is statistically meaningful. As a result, we focus on apartments with room counts greater than 8 for the following boxplot.

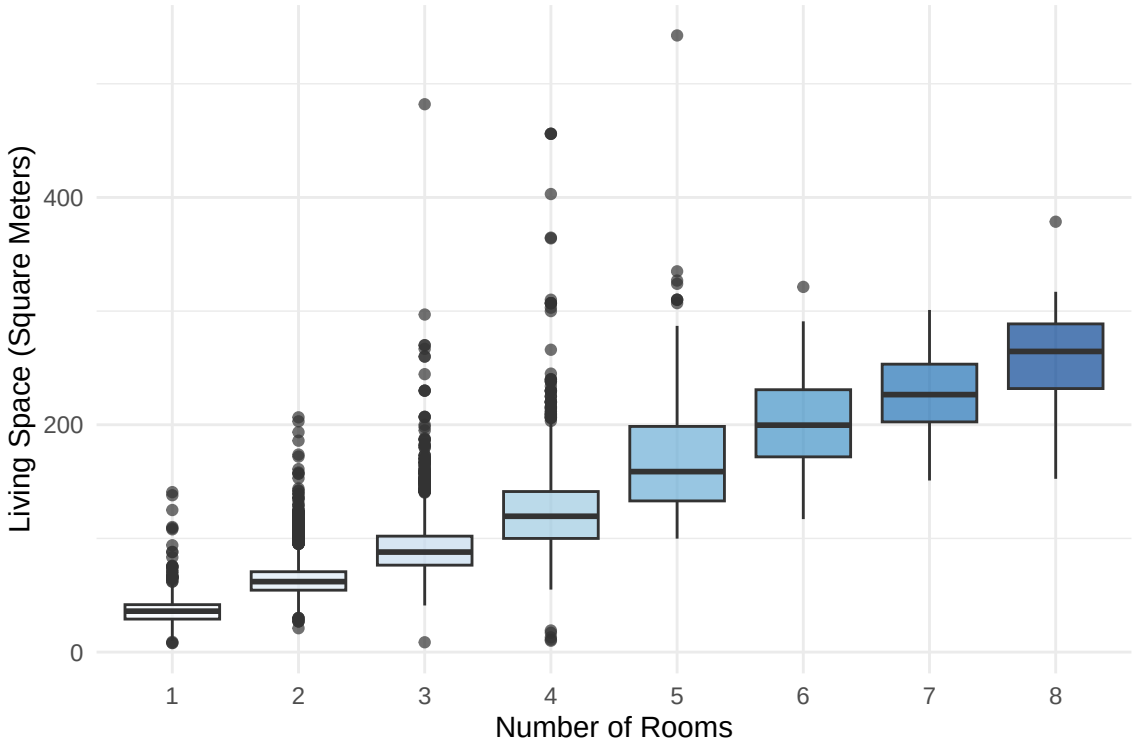

Figure 4: Living Space Distribution by Number of Rooms

The filtered boxplot confirms this hypothesis. Even for a room count like 8, which only has 16 occurrences, the trend remains consistent. Larger apartments have larger living spaces, which aligns with our expectations. A noteworthy observation is the prevalence of outliers in apartments with up to 5 rooms, especially for those with 3 and 4 rooms. These outliers are highly dispersed, potentially indicating that some of these entries might represent office spaces rather than standard apartments.

Another important aspect of our dataset is the condition of the apartments. This feature provides qualitative information about the state of the properties, with categories like *mint_condition*, *refurbished*, and *renovated*. However, the definitions of these categories are not immediately clear, particularly the distinction between "mint_condition" and the other classifications.
To better understand these categories, we compare the condition of the apartments to their rent values. This comparison allows us to infer potential meanings or differences between the conditions.

A clear order emerges among the apartment conditions. Apartments classified as *mint_condition* appear to represent the best quality. Similarly, *refurbished* properties seem to have a better condition than *renovated*.

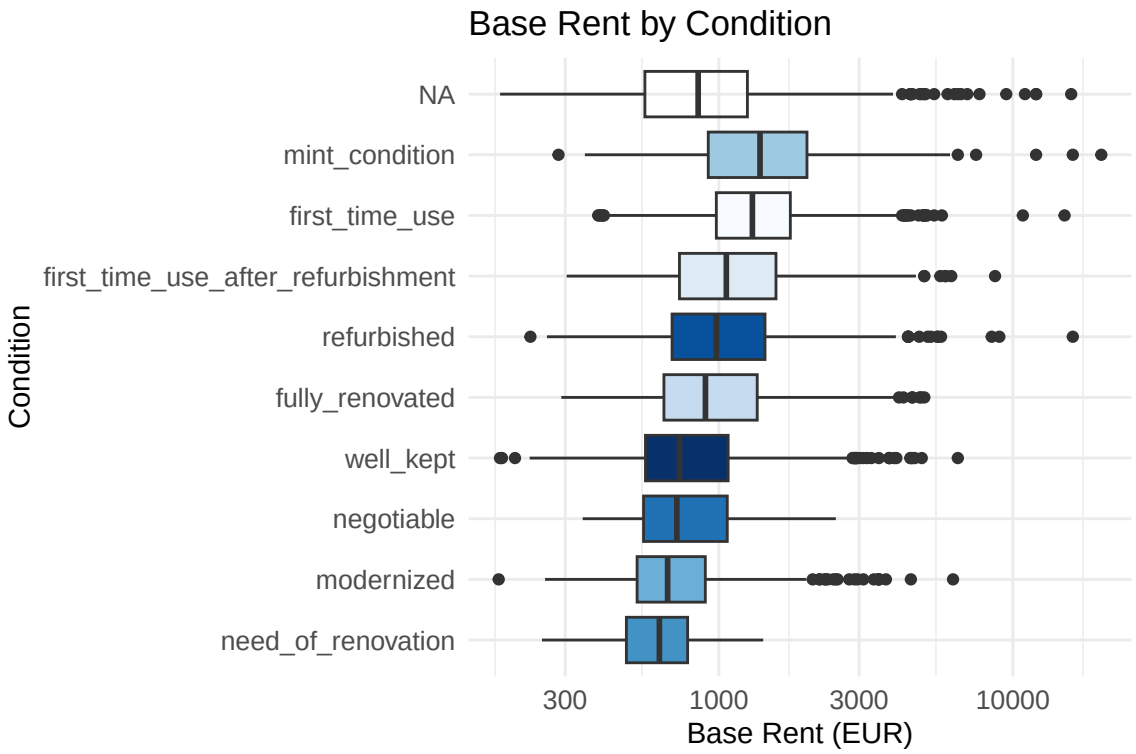

Figure 5: Base Rent by Conditions

This ordering aligns with the intuition that better maintained or new apartments tend to have higher rents. Outliers might represent unique cases, such as misclassifications, unusual rental agreements or exceptional property features.
Another observation is that the missing entries (`NA`) have a wide interquartile range and contain several outliers, which indicates that these entries are spread across categories and may not represent a single uniform condition. This variability could introduce noise into our analysis and modeling. To address this, we aim to impute the missing condition values.

Exploring Correlations Between Variables

To better understand the relationships between variables in our dataset, we analyzed the correlations between them, focusing on how the predictors relate to the `baseRent`. This helps us identify expected patterns and uncover any unexpected relationships.

From the correlation analysis, we observed several expected relationships:

- Apartments with a high `baseRent` tend to have:
  - Higher `serviceCharges`, `heatingCosts` (additional rental costs)
  - Larger `livingSpace`, more rooms (`noRooms`)

These findings align with our expectations, as more expensive apartments generally offer more space and additional services.

One initially confusing correlation was between `thermalChar` and `yearConstructed`. After further research, we found that `thermalChar` represents the apartments energy efficiency level, which is often linked to the buildings construction year. Newer buildings tend to be more energy-efficient due to modern construction

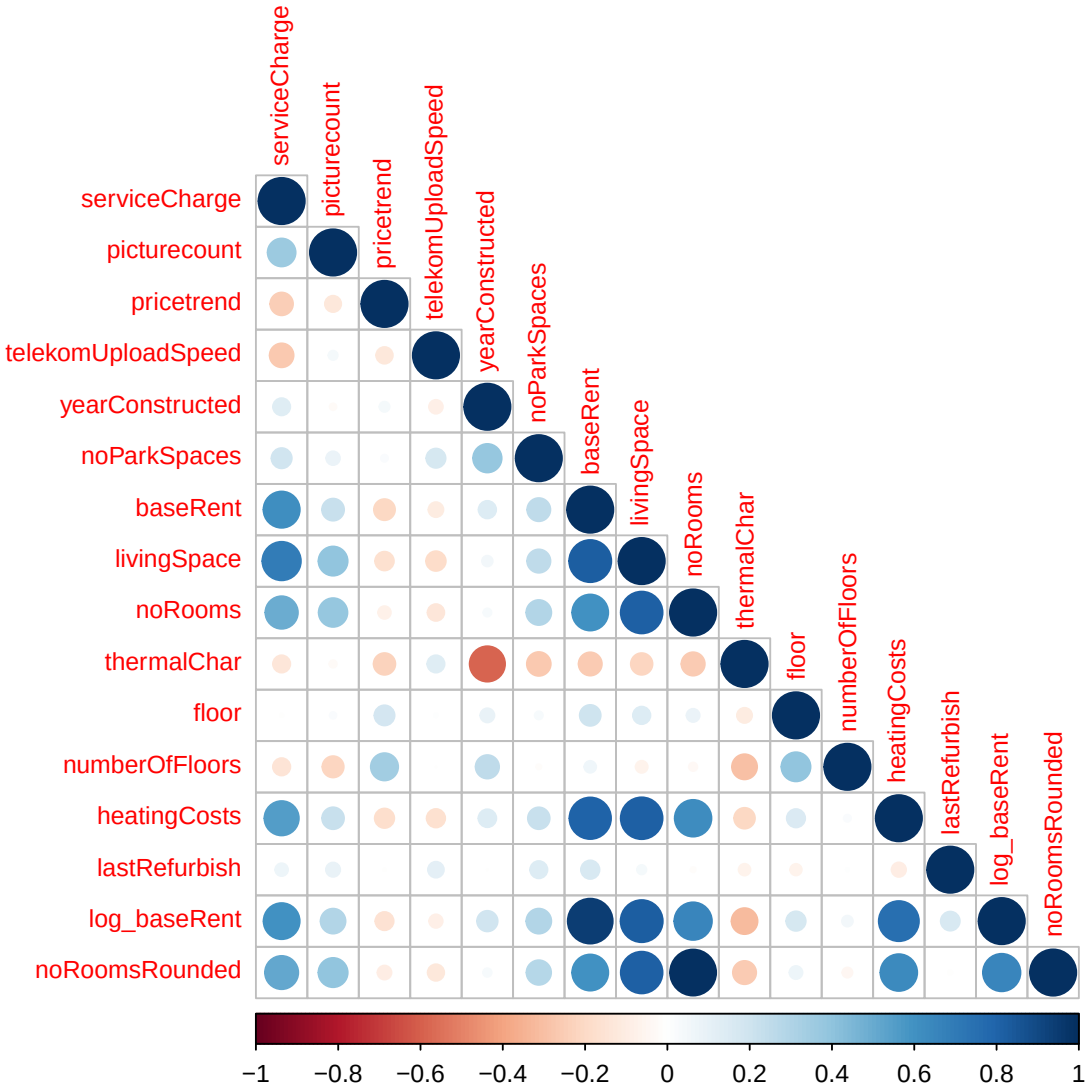

Figure 6: Correlation Matrix

standards and regulations.
Interestingly, this energy efficiency only shows a weak negative correlation with `heatingCosts`. Contrary to our inital expectations, more energy-efficient apartments appear to incur slightly higher heating costs. One possible explanation is the cost disparity between energy sources, in 2020 and even today, dirtier energy sources are often more affordable than renewable or sustainable alternatives.

Imputation

Missing values in the data can significantly impact model performance, especially for Support Vector Machines (SVMs). While tree models such as those in `rpart` can handle missing values, SVMs from the `e1071` package cannot. SVMs simply exclude rows with missing values, which can lead to a scenario where the model is trained on as few as 10 observations.

In our initial data overview we identified the number of missing values in each column. To improve our understanding, we will visualize the top 10 columns with the highest percentage of missing values. These columns are also potential candidates for exclusion if we decide to reduce our model size. The following plot displays the absolute number of missing values out of a total of 10,304 entries.

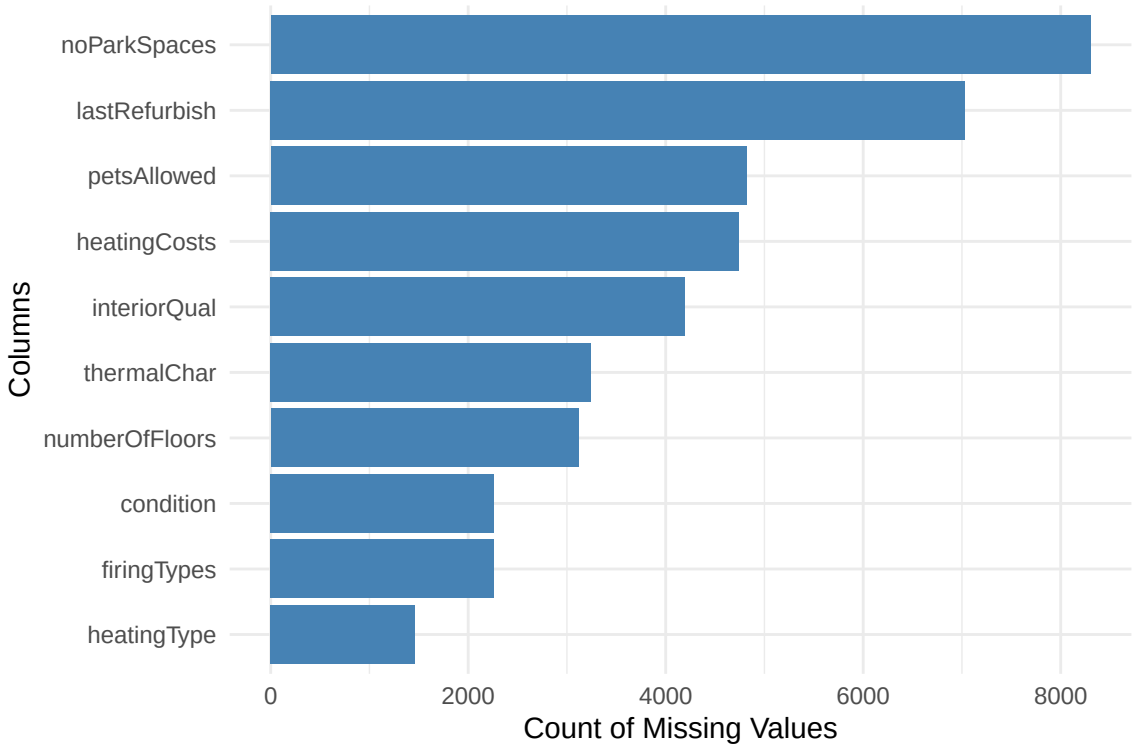

Figure 7: Top 10 Columns with Most Missing Values

For some columns, such as `lastRefurbished`, missing values have specific meanings. For example, indicating that the apartment has never been renovated. For the remaining columns we assume that the missing values represent missing data.
To address the values in `lastRefurbished` we will impute missing values with the year of construction, as this represents the most plausible time for the first or last renovation.
For all other columns we will use *Gibbs sampling* to impute values. This approach preserves the underlying distribution of variables and minimizes information loss during model fitting.

The imputation process using the *MICE* algorithm will generate five different datasets. For model training

we will build models on all five datasets and aggregate their results to produce a final output. For simplicity in visualizations, such as decision tree plots, we will use the first dataset generated by Gibbs sampling.

Model Overview

The following chapter begins with an overview of the machine learning algorithms we plan to use for our models. The content in this section is primarily based on our lecture slides, with citations provided for any external sources. Subsequently, we will train various models and evaluate them.

Regression Trees

Tree models are supervised learning algorithms used for both regression and classification. They work by splitting the dataset into smaller subsets based on decision rules derived from the features and forming a tree structure. Each split improves the prediction accuracy of the model on the training data. The process continues until a stopping criterion, such as minimum number of observations in a node or maximum tree depth, is met. The tree's leaf nodes represent the final predictions. For regression trees, this prediction is typically the average of the target variable within that leaf.
Regression trees are particularly useful for predicting continuous values as they can easily capture non-linear relationships. In our case, we aim to predict `baseRent` using a regression tree model.

The following procedure shows how regression trees work:

1. **Initialization**:
   The process begins with the entire dataset. The mean of the target variable Y serves as an initial prediction, as it minimizes the **Residual sum of squares (RSS)**:

   $$RSS = \sum_{i=1}^{n} (y_i - \hat{y})^2,$$

   where $\hat{y}$ is the mean of Y.

2. **Splitting the data**:
   At each node, the dataset is split into two regions R_1 and R_2, based on feature X_j and threshold x:

   $$R_1 : X_j < x, \quad R_2 : X_j \geq x.$$

   The goal is to find the split that minimizes the RSS for the two regions:

   $$RSS = \sum_{i \in R_1} (y_i - \hat{y}_1)^2 + \sum_{i \in R_2} (y_i - \hat{y}_2)^2,$$

   where $\hat{y}_1$ and $\hat{y}_2$ are the mean values of Y in R_1 and R_2, respectively.

3. **Recursive splitting**:
   This process of splitting is repeated recursively. At every step, the algorithm evaluates all possible splits for every feature and selects the one that minimizes the RSS.

4. **Stopping criteria**:
   If we continue splitting until each node contained only a single observation, then each node would perfectly predict the value of its observation, leading to an overfitted model with an RSS of 0.
   That is why we stop the tree growth when a predefined stopping criterion is met. This could be:

   - The minimum number of observations in a node is reached.

- The maximum depth is reached.
- Further splits fail to reduce the RSS significantly.

5. **Pruning with penalized least squares**:
   Overfitting is a common problem with regression trees when they grow too large. To address this, the tree is pruned using **Penalized least squares (PLS)**:

$$PLS(\alpha) = \sum_{j=1}^{|T_\alpha|} \sum_{i \in R_j} (y_i - \hat{y}_j)^2 + \alpha |T_\alpha|,$$

   where:

   - $|T_\alpha|$ is the number of terminal nodes (leaf nodes) in the pruned tree T_α,
   - $\alpha \geq 0$ is a complexity parameter controlling the trade-off between tree size and prediction accuracy.

   The parameter α is chosen by using cross-validation, which ensures the pruned tree balances model complexity and performance.

6. **Choosing the best subtree**:
   To find the optimal tree:

   - The full tree is grown.
   - The pruning process evaluates subtrees by varying α and calculates the relative error using cross-validation.
   - The subtree corresponding to the smallest cross-validation error is selected.

Ensemble Methods

To understand how tree ensemble methods work, we need to know what bootstrapping is. Bootstrapping is a statistical technique that involves sampling with replacement from a dataset to create multiple subsets. Each bootstrap sample is the same size as the original dataset, but some observations may appear multiple times while others may be excluded. This technique is commonly used to generate diverse training subsets or to augment small datasets. In ensemble methods like bagging and random forests, bootstrapping enhances generalization and robustness by reducing variance only slightly increasing bias.

Bagging

Bagging is an ensemble learning technique that enhances accuracy and reduces variance by combining the predictions of multiple models. It involves training several instances of the same machine learning algorithm on different subsets of the data and then aggregating their predictions to form a final output. For regression tasks, this typically involves averaging the predictions. For classification tasks, the class with the majority vote is selected.

Bagging reduces model variance by averaging out individual errors, which leads to more robust and reliable predictions. This technique is especially useful when using base models that are prone to high variance, such as decision trees. An ensemble of decision trees is commonly referred to as a *forest*.

Random Forests

Random forests enhance the concept of *forests* by introducing additional randomness during the tree building process. At each node split, only a random subset of features is considered, which prevents the model from relying too heavily on the same dominant predictors. This also allows less important predictors to capture

more subtle patterns in the data. Typically, the number of features m considered at each split is given by $m = \lfloor \sqrt{p} \rfloor$, where p is the total number of features. This approach helps to reduce overfitting caused by a few strong predictors, which leads to more robust and reliable predictions.

Boosting

Boosting is an ensemble learning technique in machine learning that aims to improve the performance of weak models, such as a small tree (called *stump*) by combining their predictions to create a strong predictive model. The core idea is to iteratively train a series of models, where each new model focuses on correcting the mistakes made by the previous ones.

The process begins with a simple model, like a stump, which makes predictions on the training data. The instances that were misclassified or had large errors are then given higher weights, so that subsequent models pay more attention to correcting these errors.
In the next iteration, a new model is trained on the weighted data, where the misclassified have higher weights. After training, it is added to the ensemble and its predictions are combined with those of the previous models. The respect to errors the weights are updated again. This process repeats for several iterations, with each new model gradually improving the overall predictions.
The final prediction is made by aggregating the predictions of all models. In algorithms like AdaBoost, this is done by weighted voting, where better performing models have more influence on the result.

Boosting helps to reduce bias and variance. Initially, a simple model may have high bias and low variance, but as boosting progresses, the bias decreases, which improves the accuracy of our predictions. However, the increased flexibility and strong focus on errors can also lead to higher variance, making the model more sensitive to noise. To prevent overfitting regularization techniques and early stopping are often used.

We want to test the following boosting algorithms on our dataset:

- **Gradient Boosting**: Focuses on minimizing a loss function by fitting new models to the residuals of previous predictions.
- **XGBoost**: An optimized version of gradient boosting that enhances speed, performance and uses regularization techniques.

Support Vector Machines

Support vector machines (SVMs) are supervised learning algorithms that can be used for both classification and regression tasks. They work by finding a hyperplane that best separates data points into different classes. SVMs excel in situations where relationships between variables are complex and potentially non-linear. This chapter provides an overview of how SVMs work, with a focus on their extension to regression tasks and the kernel trick, which enables handling non-linear relationships effectively.

Linear Support Vector Machines

In classification SVMs aim to identify a hyperplane $f(x)$ that satisfies:

$$\hat{y}(x) = \text{sign}(f(x)), \quad f(x) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j,$$

where the hyperplane separates the data into distinct classes while maximizing a margin, which is the distance between the hyperplane and the closest data points from each class, known as *support vectors*.

However, real world datasets often contain overlapping classes, making perfect separation impossible. To address this SVMs use a *soft margin* that allows some misclassifications. The trade off between margin width and classification errors is controlled by the penalty parameter C. The optimization problem for a

soft margin SVM is:

$$\text{maximize } M, \quad \text{subject to } y_i f(x_i) \geq M(1-\xi_i), \quad \xi_i \geq 0, \quad \sum_{i=1}^{n} \xi_i \leq C,$$

where ξ_i are slack variables representing the extent of misclassification.

While linear SVMs are effective for linearly separable data, they struggle with non-linear relationships, where a simple hyperplane cannot adequately separate the classes.

Non-linear Support Vector Machines

To overcome the limitations of linear SVMs, data can be projected into higher-dimensional feature spaces, where a linear hyperplane can separate the data. This involves mapping the input x from its original space $\mathbb{R}^p$ to a higher-dimensional feature space $\mathbb{R}^q$, where $q > p$:

$$\phi(x) : \mathbb{R}^p \to \mathbb{R}^q.$$

In this transformed space, a linear decision boundary may successfully classify the data. However, explicitly computing $\phi(x)$ is computationally inefficient, especially for large datasets or high-dimensional mappings.

The *kernel trick* addresses this challenge by avoiding explicit computation of $\phi(x)$. Instead, SVMs calculate the inner product in the feature space $\langle \phi(x_i), \phi(x_j) \rangle$, indirectly using a kernel function $K(x_i, x_j)$. This enables efficient computation in the higher-dimensional space without ever explicitly performing the transformation. The decision function then takes the form:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i),$$

where S represents the set of support vectors that define the hyperplane, enhancing computational efficiency by focusing only on these critical points. The values α_i are the Lagrange multipliers corresponding to each support vector x_i. These multipliers are determined during the training process and reflect the importance of each support vector in defining the optimal hyperplane.
By leveraging the kernel trick, SVMs maintain computational efficiency while gaining the flexibility to tackle complex, non-linear problems.

Regression with Support Vector Machines (Rasifaghihi, 2023)

Support vector regression (SVR) is an extension of SVMs for regression tasks. The goal of SVR is to identify a function $f(x)$ that accurately predicts the target variable, while minimizing model complexity to enhance generalization and reduce overfitting.

A distinctive feature of SVR is the introduction of an ϵ-insensitive tube around the regression hyperplane. This tube defines a tolerance margin within which deviations between predicted and actual target values are not penalized. This creates a balance between model complexity and its ability to generalize effectively.

For a linear function $f(x) = wx + b$, this means to minimizing w, which makes it less sensitive to small variations in the input data.
The SVR problem is formulated as a convex optimization task with the following objectives:
1. Ensure that the deviations between predictions $f(x)$ and actual target values y remain within ϵ for all training data points. 2. Minimize the complexity of the function $f(x)$ by keeping w small.

In cases where the optimization problem is infeasible due to noisy data or overlapping regions, *slack variables* (ξ_i^+ and ξ_i^-) are introduced. These variables allow for deviations beyond the ϵ-insensitive tube, representing data points that fall outside the tube.

The optimization problem can be expressed as:

$$\min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i^+ + \xi_i^-),$$

subject to the following constraints:

$$y_i - f(x_i) \leq \epsilon + \xi_i^+, \quad f(x_i) - y_i \leq \epsilon + \xi_i^-, \quad \xi_i^+, \xi_i^- \geq 0.$$

The regularization strength is inversely related to C, meaning higher values allow the model to prioritize accuracy at the expense of simplicity.

SVR assigns zero prediction error to points within the ϵ-insensitive tube. For points outside the tube (slack variables), the penalty is proportional to the magnitude of their deviation. The tolerance for small deviations and penalties for larger errors make SVR more robust to overfitting.

Model Implementation

TODO

- geo_plz contains 214 levels on 10000 rows, likely that we get plzs in test data which arent in train data,
- we also did some sort of dimension reduction on the geo_plzs by adding the districts and have subdistricts inside the regio3
- code for all models can be found in eric_model
- we will do HPO only on the first gibbs otherwise it takes too long and as gibbs leaves the distributations etc same parameters are likely to still be near the optimal param

a description of your fitting process for each method including, a summary of how you arrived at your final model, the choice of hyperparameters and how you made this choice, (8P)

- on imputed data
- with outliers
- without outliers only at the end for the final comparison with the best model

```
train_data1 <- train_data[[1]]
test_data1 <- test_data[[1]]
```

Regression Trees

In this section, we focus on training a regression tree using the `rpart` package and visualizing the results.

To determine the optimal depth or number of splits for our tree, we use cross-validation. The `rpart` package simplifies this process by calculating the cross-validated error for each of the subtrees. These cross-validation results are stored in a complexity parameter table (`cptable`), which provides the cross-validation errors for the subtrees. This table helps us to assess the complexity needed when we prune our tree. Below, we display the first 10 rows of the `cptable` for our tree:

```
default_tree <- rpart(
  baseRent ~ .,
  data = train_data1,
  method = "anova"
)
print(head(default_tree$cptable, 10))
```

```
##       CP nsplit rel error xerror  xstd
## 1  0.398      0      1.00   1.00 0.089
## 2  0.134      1      0.60   0.60 0.071
## 3  0.078      2      0.47   0.52 0.052
## 4  0.047      3      0.39   0.43 0.051
## 5  0.030      4      0.34   0.38 0.034
## 6  0.024      5      0.31   0.37 0.032
## 7  0.018      6      0.29   0.35 0.032
## 8  0.015      7      0.27   0.33 0.032
## 9  0.011      8      0.26   0.32 0.032
## 10 0.010      9      0.25   0.32 0.032
```

From the `cptable`, we can use two approaches to prune the tree:
1. **Lowest Cross-Validation Error**: Prune the tree to the complexity parameter corresponding to the lowest cross-validated error (`xerror`).
2. **Standard Deviation Rule**: Use the complexity parameter with the simplest tree whose error is within one standard deviation of the minimum cross-validated error. This approach is recommended by the authors of the `rpart` package.

The following plot illustrates the first complexity parameters and cross-validation errors for our tree:

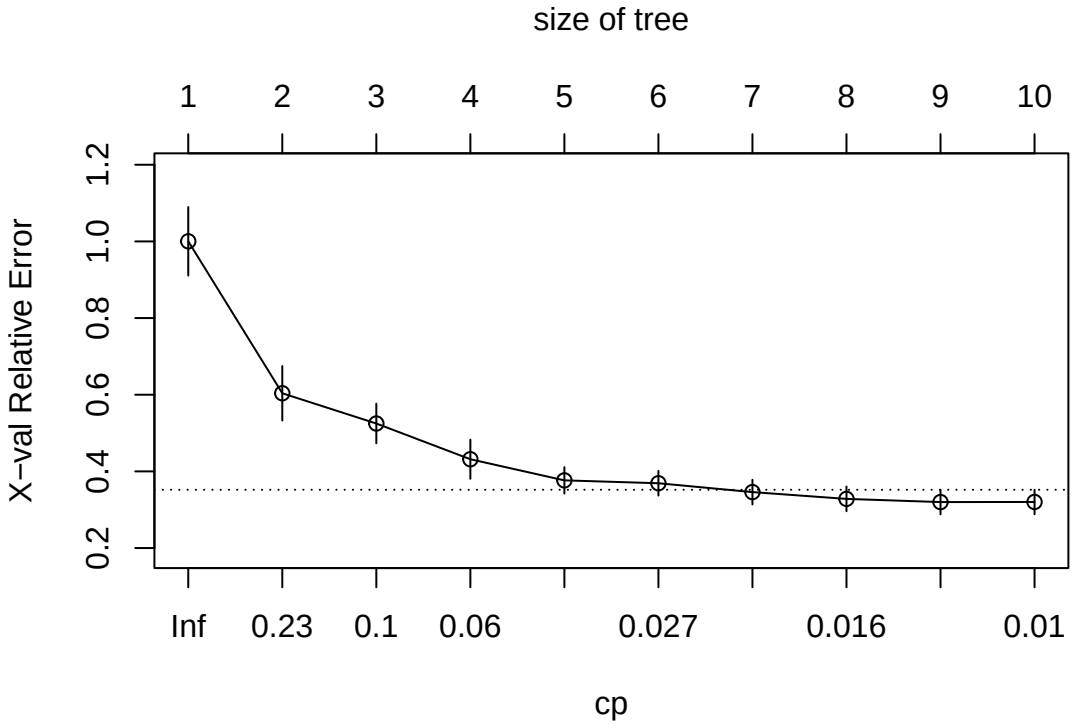

Figure 8: Default Tree Complexity

Using these methods, we pruned the full tree. The first splits of the pruned tree are shown below:

The pruning process helps to create a more interpretable tree while maintaining or even enhancing its predictive performance. Below, we compare the performance of the full tree and the two pruning techniques on the test data, averaged across all five imputed datasets. Note that the *full tree* does not necessarily imply only one node per leaf, other stopping parameters influence the structure. Having one terminal node per observation would be excessive and unnecessary.

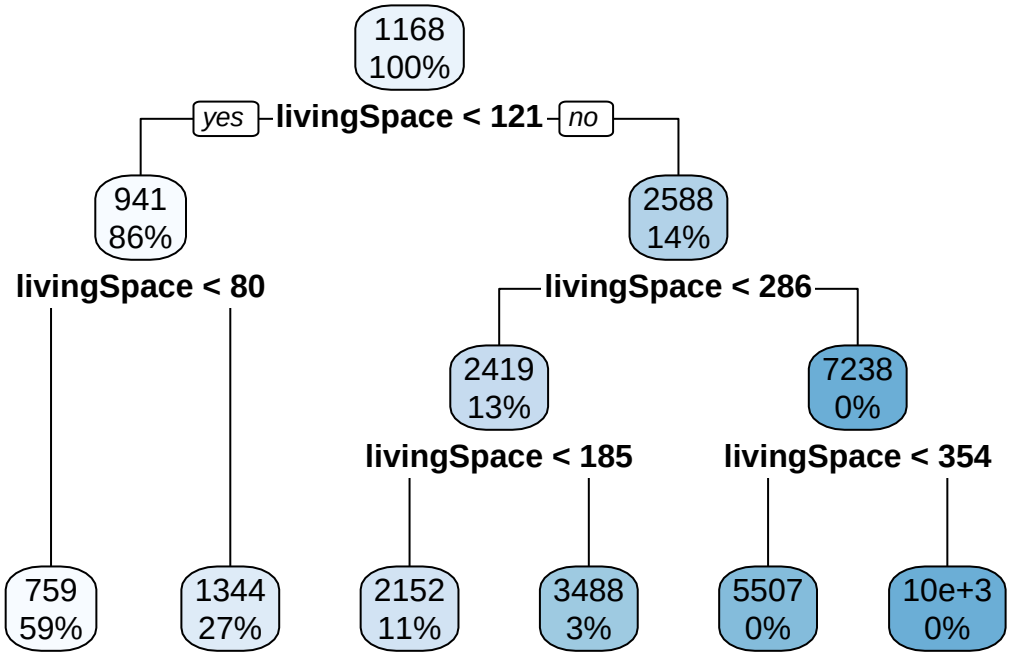

Figure 9: Tree First Splits

Method	Average MSE	Average RMSE	Average MAE
Full Tree	130,515.9	361.22	204.53
Best xError	129,591.7	359.91	207.78
Best (Authors' Method)	165,022.5	405.1	253.89

From the table we observe the following:
1. **Full Tree**: Surprisingly, the full tree performs well on the test dataset, achieving competitive results in MSE and RMSE and the best results for mean absolute error. This suggests a high degree of variance in the dataset.
2. **Best xError**: This pruning technique achieves significantly better results compared to the method recommended by the authors of the `rpart` package, as it has the lowest MSE and RMSE out of all methods.
3. **Authors Method**: While this approach prioritizes simplicity and generalization, it results in higher error values compared to the other methods. This means that this pruning method may not be the optimal choice for this dataset.

Support Vector Machines

In this section, we focus on training a SVM for regression using the `e1071` package.

SVM Regression is a distance based algorithm, which means they calculate distances between data points to make decisions. One key consideration when using SVMs is that variables with larger scales can dominate the distance calculations, potentially leading to inaccurate results. This is because the SVMs decision boundary relies on calculating the distance between points. Therefore normalizing or standardizing data is crucial to ensure that each feature contributes equally to the model.

For tuning the SVM parameters and validating its performance, we performed cross-validation on one of our imputed datasets with the following settings:

- **Kernel Function**: (`polynomial`, `radial`, `sigmoid`, `linear`)
  The kernel function defines the type of decision boundary the SVM will create. It maps the input features to a higher-dimensional space where linear separation is possible.
- **Cost (C)**: (0.5, 1, 2, 4)
  The cost parameter controls the trade-off between minimizing training error and ensuring good generalization to new data. It penalizes misclassification of data points. A higher cost leads to a more complex model that fits the training data closely, which may result in overfitting.
- **Gamma ( )**: (0.01, 0.5, 1, 2)
  Gamma influences individual training points when using non-linear kernels (e.g., *radial*, *polynomial*). A high gamma value causes a potentially overfitted model. A low gamma value results in a smoother decision boundary that is less sensitive to individual data points. (Günay, 2021)
- **Degree**: (2, 3, 6, 10)
  The degree parameter applies only to the *polynomial* kernel. It specifies the degree of the polynomial used to compute the kernel. A higher degree increases the complexity which could lead to overfitting.
- **Coef0**: (0, 1, 5, 10)
  The coef0 parameter is relevant only for *polynomial* and *sigmoid* kernels. It controls the impact of the higher-dimensional feature space on the decision boundary. (lejlot, 2013)

In this case the results of SVM fitting over all imputed datasets are as follows:

Method	Average MSE	Average RMSE	Average MAE
HPO SVM	114,781.2	338.71	169.77

When comparing SVMs to simple regression trees, we can see that SVMs performed slightly better. Decision trees are more interpretable, but also prone to overfitting, especially when there are many features or complex relationships in the data. SVMs tend to generalize better, which leads to lower error metrics.
While both models have their advantages, the SVMs performance suggests it is better suited for this particular dataset, likely due to its improved ability to handle outliers and avoid overfitting.

Gradient Boosting

- include comparison of decision tree vs ensemble

XGBoost

Ensemble Methods

Outlier Detection

Smaller Model

Model Evaluation

- include comparison of ensmble tree methods and svms

Evaluation on current apartments

Why it might fail? Market Demand and Supply: An imbalance between the number of available apartments and the demand from potential tenants leads to price fluctuations. High demand with limited supply results in increased rents. Economic Conditions: The general economic situation and income development of the population in Berlin play a role in determining rental prices. (https://eichenglobal.com/en/blog/mietspiegel-2024-berlin/) Government Policies: Regulations such as rent caps or tenant protection laws can influence rental prices by limiting increases and affecting market dynamics. (https://www.pwc.de/en/real-estate/rent-cap-and-regulation-of-the-real-estate-market.html)

References

Bar, C. (2020). Apartment rental offers in germany. In *Kaggle*. https://www.kaggle.com/datasets/corrieaar/apartment-rental-offers-in-germany/data

Günay, G. (2021). Understanding parameters of SVM. In *Kaggle*. https://www.kaggle.com/code/gorkemgunay/understanding-parameters-of-svm

lejlot. (2013). Scikit-learn SVC coef0 parameter range. In *Stack Overflow*. https://stackoverflow.com/questions/21390570/scikit-learn-svc-coef0-parameter-range

Rasifaghihi, N. (2023). From theory to practice: Implementing support vector regression for predictions in python. In *Medium*. https://medium.com/@niousha.rf/support-vector-regressor-theory-and-coding-exercise-in-python-ca6a7dfda927