Tampere University

Tommi Kinnunen

HOUSE PRICE PREDICTION USING MODEL STACKING

Master's Thesis
Faculty of Information Technology and Communication Sciences
July 2025

Abstract

Tommi Kinnunen: House Price Prediction Using Model Stacking
Master's Thesis
Tampere University
Master's Programme in Computing Sciences
July 2025
Supervisor: Professor Jaakko Peltonen

This thesis studies the use of model stacking to improve the predictive performance of regression models for real estate price prediction. Model stacking is an ensemble method that aims to learn an optimal meta-model that combines the predictions of multiple regression models. The objective was to compare several machine learning models, including XGBoost, CatBoost, lasso regression, neural networks, and support vector regression (SVR), in addition to model stacking and voting regression. Model performance was evaluated using mean absolute error (MAE), root mean squared error (RMSE), and the coefficient of determination (R^2) on a real estate dataset from Tampere.

The results indicate that the stacking ensemble outperforms individual base models and the voting regression model, achieving the highest R^2 score and the lowest MAE. This demonstrates that model stacking effectively combines the strengths of multiple base models, leading to more accurate predictions. However, all models, including the stacking ensemble, showed limitations when predicting prices for high-value properties, where performance was less robust due to the lack of such houses in the dataset.

While model stacking improved predictive accuracy, it required significantly longer training times compared to voting regression. Another drawback is the lack of interpretability of stacking, particularly when using complex base models like neural networks. Despite these limitations, stacking is found to be a suitable approach for tasks where predictive accuracy is the primary goal and longer training times are acceptable.

The thesis concludes by discussing potential areas for future research, including the integration of macroeconomic variables, evaluation of multi-layer stacking architectures, and the inclusion of additional meta-features. The findings provide valuable insights into the effectiveness of model stacking in real estate price prediction.

Keywords: model stacking, real estate, regression, machine learning, ensemble methods

Table of contents

1 Introduction **4**
1.1 Research questions . . . 4
1.2 Thesis structure . . . 5

2 Literature review **6**

3 Methodology **8**
3.1 Data and preprocessing . . . 8
3.1.1 Data selection . . . 8
3.1.2 Handling of categorical variables and one-hot encoding . . . 8
3.1.3 Missing and erroneous data handling . . . 9
3.1.4 Handling of numerical features . . . 9
3.1.5 Feature scaling . . . 10
3.2 Exploratory data analysis . . . 10
3.2.1 Descriptive statistics . . . 10
3.2.2 Visualizing the distribution of categorical variables . . . 11
3.2.3 Visualizing relationships between variables . . . 11
3.2.4 Distribution of house prices . . . 14
3.3 Theory of machine learning models . . . 16
3.3.1 Model stacking . . . 16
3.3.2 Voting regression . . . 18
3.3.3 Ridge regression . . . 19
3.3.4 Lasso . . . 19
3.3.5 Support vector regression . . . 20
3.3.6 Gradient boosted regression trees . . . 22
3.3.7 Extreme gradient boosting . . . 24
3.3.8 CatBoost . . . 26
3.3.9 Feed-forward neural network . . . 27
3.4 Research implementation and hyperparameter tuning . . . 29
3.5 Evaluation metrics . . . 32
3.5.1 Mean absolute error . . . 32
3.5.2 Root mean squared error . . . 32
3.5.3 Coefficient of determination (R^2) . . . 32

4 Results **34**

5 Discussion **39**
5.1 Limitations . . . 39
5.2 Future research . . . 40

References **41**

1 Introduction

Predicting housing prices is a crucial problem in the real estate market, where accurate price estimation plays a vital role in property investment and decision-making of individuals buying or selling houses. The prediction task involves estimating the value of properties based on a wide range of factors such as location, size, number of rooms, house type and building materials. The steady development of machine learning techniques over the years for tasks such as predicting real-estate prices means there are number of high accuracy models to choose from. However, despite the advancements, housing price prediction remains a challenging problem due to the complex and noisy nature of the data as well as influencing factors such as macro market trends.

To address these challenges and further improve predictive accuracy, model stacking has emerged as a promising approach in recent years, especially in the Kaggle competition community. Stacking is an ensemble learning technique that combines multiple models to make even better predictions. Unlike ensemble methods such as bagging or boosting, stacking trains a meta-model on the predictions of base models, enabling it to combine their outputs in a potentially more effective way to make the final prediction. This approach can lead to improved generalization and robustness in predictions by addressing the limitations of individual models.

1.1 Research questions

While individual machine learning models like Extreme gradient boosting have shown success in predicting house prices (Zaki et al. 2022), there is evidence that combining multiple models can result in better performance (Yang et al. 2023). Ensemble methods like voting regression aggregate predictions from different models by using a simple averaging or voting mechanism. Model stacking, on the other hand, provides a more sophisticated approach by training a meta-model to learn how to effectively combine the predictions from multiple base models. This thesis explores the effectiveness of model stacking in improving house price prediction accuracy compared to simpler ensemble techniques like voting regression.

The research questions addressed in this thesis are:
Question 1: How effectively does model stacking improve the prediction accuracy of models such as lasso, XGBoost, CatBoost, SVR, and neural networks?
Question 2: How does model stacking compare to voting regression in predicting house prices, specifically in a relatively small dataset from Tampere, Finland?

To answer these questions, the thesis aims to implement and evaluate model stacking applied to a real-estate dataset from Tampere. Furthermore, we use hyper-parameter tuning to make sure we get the best performance out of each of the model. Finally we compare the performance of model stacking with voting regression and their individual base models using prediction accuracy evaluation metrics.

1.2 Thesis structure

Firstly, Chapter 2 reviews the existing literature and research related to house price prediction and the use of model stacking in various regression tasks. The choice of the models studied in this thesis is also justified in this chapter based on the literature. Next, Chapter 3 presents the methodology, including data preprocessing steps, exploratory data analysis, and the theory behind the machine learning models used in this thesis. It also discusses the implementation of hyperparameter tuning and evaluation metrics. In Chapter 4, the results of the study are presented, comparing the performance of different models based on the evaluation metrics. Lastly, in Chapter 5, the results are interpreted, the limitations of the study are addressed, and possible directions for future research are suggested.

2 Literature review

Predicting real estate prices is continuously advancing toward more accurate and reliable predictions using more and more sophisticated machine learning methods. Pai et al. (2020) explored the use of machine learning techniques for predicting real estate transaction prices, employing models such as Least Squares Support Vector Regression (LSSVR), Classification and Regression Trees (CART), General Regression Neural Networks (GRNN), and Backpropagation Neural Networks (BPNN). Their research, utilizing transaction data from Taichung, Taiwan, found that attribute selection significantly improved model performance. In particular, LSSVR outperformed the other models in terms of prediction accuracy, highlighting the effectiveness of support vector-based models for real estate price forecasting.

Similarly, Zaki et al. (2022) applied XGBoost, a powerful gradient boosting algorithm, to predict house prices, comparing it with the traditional hedonic pricing model. Their results showed that XGBoost significantly outperformed the hedonic model, demonstrating its better predictive accuracy, regularization for overfitting, and ability to handle sparse data.

As machine learning has evolved, ensemble methods like model stacking have gained popularity for improving the predictive power of models. Stacking, first introduced by Wolpert (1992), combines the outputs of multiple base models and uses a meta-model to learn how to optimally combine their predictions. This approach can improve prediction accuracy by utilizing the strengths of various algorithms, and has been successfully applied to housing price prediction and other fields.

In their work, Yang et al. (2023) proposed a novel D-Stacking method, which combines diverse models like XGBoost and BP neural networks to predict housing prices. Their findings highlighted that stacking models with a variety of base learners leads to more accurate predictions, reinforcing the idea that model diversity is critical to improving ensemble performance.

Nnadozie et al. (2022) also utilized multi-level stacking for predicting real estate prices, employing models such as XGBoost and LightGBM. Their results demonstrated that stacking not only improves accuracy but also increases the model's robustness. This supports the idea that stacking can significantly boost the performance of models used for complex tasks such as real estate price prediction.

The success of stacking has been reproduced in other domains as well. Song et al. (2024) used stacking and voting hybrid learning methods for landslide susceptibility prediction, showing that ensemble techniques consistently outperform individual models in terms of predictive power. Similarly, Park et al. (2023) employed a stacking ensemble with models like Random Forest (RF), Support Vector Regression, and CatBoost for predicting construction project costs, illustrating the performance benefits of stacking.

Motivated by the success of model stacking in previous research highlighted in this section, this study selects a diverse set of base models for the stacking ensemble. The models selected for this research are lasso (Least Absolute Shrinkage and Selection Operator), support vector regression (SVR), XGBoost, CatBoost, and a feed-forward neural network.

Lasso (Tibshirani 1996), a linear model, is effective when dealing with high-dimensional data that require feature selection. Additionally, its ability to prevent overfitting through regularization helps the model to generalize to new data. Lasso is also chosen because of its simplicity, making it a good baseline model to which others can be compared.

Furthermore, SVR (Drucker et al. 1997) excels at capturing non-linear relationships in data while maintaining robustness to noise, which is particularly important in real estate price prediction. Its inclusion adds diversity to the ensemble, and it has demonstrated strong performance in prior research, such as Pai et al. (2020).

XGBoost (Chen and Guestrin 2016) is a gradient boosting tree algorithm that can handle complex and noisy data because of its boosting and regularization techniques. Similarly, CatBoost (Prokhorenkova et al. 2019), another gradient boosting tree method, is designed to handle categorical variables efficiently, which are common in real estate datasets. CatBoost has additional technical improvements over XGBoost that are reviewed in more detail in the Methodology section. Both of them are recently developed models that have shown great performance based on the literature reviewed.

Finally, the inclusion of a feed-forward neural network further improves the ability to model complicated, non-linear relationships in the data. Rampini and Re Cecconi (2022) demonstrated the effectiveness of artificial neural networks (ANNs) in predicting real estate prices in the Italian housing market. In their study, they compared three popular machine learning models: ElasticNet, XGBoost, and Neural Networks using a dataset from two cities in northern Italy. Their results showed that the ANN outperformed the other models in their task.

3 Methodology

3.1 Data and preprocessing

The data was collected from reported sales by real estate agents in Tampere between 9th November 2016 and 18th September 2024, by KVKL's (Kiinteistönvälitysalan Keskusliitto) Price Tracking Service; www.hintaseurantapalvelu.fi. This section outlines the steps taken to clean and preprocess the real estate dataset used in this study. The data underwent several stages of preprocessing, including variable selection, handling of missing values, transformation of categorical variables, and scaling of numerical features. After the data preprocessing steps, the dataset was split into training and testing sets, with 75% of the data used for training the models and the remaining 25% for testing. The cleaned and preprocessed dataset contained 15 features, one target variable (total price) and 28 111 samples.

3.1.1 Data selection

The dataset initially contained a variety of features related to real estate listings, such as property type, location, size, and price. For the purposes of this analysis, one feature was removed based on initial feature importance values from CatBoost and XGBoost. The remaining features were as follows:

- **Property characteristics**: Property type, Living area, Year of construction, Number of rooms, Floor number, Total floors, Property condition, Building material, Sauna, Balcony, Elevator, Beach.
- **Geographic information**: Neighborhood, Postal code.
- **Transaction details**: Total price, Maintenance fee, Transaction date.

3.1.2 Handling of categorical variables and one-hot encoding

Some categorical variables needed to be cleaned for consistency and to make them usable in the modeling process. In particular, the variable Neighborhood had various naming conventions, which were standardized by replacing inconsistent entries with uniform labels. For instance, several neighborhood names such as "Amuri, pyynikki" and "Amuri, keskusta" were combined into a single category "Amuri". Similarly, the Building material column was cleaned by identifying and grouping recognized building materials into short descriptive categories.

The floor variable was derived based on the values of floor number and total number of floors. For buildings that were apartment complexes, the floor information was categorized as "Ground", "Top", or "Middle", depending on the specific floor number. For non-apartment properties, the value was set to "Not an apartment / Unknown".

To address the issue of rare categories in categorical variables, categories that appeared fewer than 12 times were grouped into a new category labeled Other. This ensured that the model would not be overfitted by infrequent categories and that the training set would have all categories in the testing set and vice versa.

One of the preprocessing steps for the categorical variables involved applying One-Hot Encoding to transform these variables into a numerical format that can be used in machine learning models. One-Hot Encoding creates binary (0 or 1) columns for each category of the categorical feature, where a 1 is placed in the column corresponding to the category of the observation, and all other columns are assigned a 0. This ensures that the model does not assume any ordinal relationship between the categories.

3.1.3 Missing and erroneous data handling

Missing values in the dataset were handled in various ways depending on the variable type. For categorical variables, such as Beach, Elevator, Sauna, and Balcony, missing values were imputed with `FALSE`, assuming that the absence of information implied a lack of these features. For the Maintenance fee variable, missing values were replaced with zero, reflecting properties with no maintenance fee.

Rows with extreme or invalid price values (e.g., prices less than or equal to one) were filtered out of the dataset. Furthermore, clearly erroneous data was fixed. For example, there were a few samples where the number of rooms was 11, but the living area was so small that it was clear the property should only have 1 room. Likewise, a row with living area of 1 was removed from the dataset.

3.1.4 Handling of numerical features

In preparation for machine learning models, the transaction date variable was converted from a date format to a numeric representation. Additionally, two numerical features, maintenance fee and living area, exhibited skewness in their distributions, which could potentially affect model performance. The skewness was measured using `scipy.stats` library's `skew` function, which calculates the Fisher-Pearson coefficient of skewness:

$$g_1 = \frac{m_3}{m_2^{3/2}},$$

where m_3 and m_2 are the sample central moments defined as:

$$m_i = \frac{1}{N}\sum_{k=1}^{N}(x_k - \bar{x})^i,$$

where x_k represents each data point, $\bar{x}$ is the mean of the data, N is the number of data points, and i is the order of the moment (Joanes and Gill 1998). To address the skewness, a Box-Cox transformation was applied to normalize these features and reduce skewness (Box and Cox 1964).

3.1.5 Feature scaling

To ensure that all features were on a similar scale, which is important for some machine learning methods, the `RobustScaler` was applied to scale the numerical features. The `RobustScaler` is part of the `sklearn.preprocessing` module in the `scikit-learn` library and is used as shown in the code fragment below. Unlike the `StandardScaler`, which scales features by removing the mean and dividing by the standard deviation, the `RobustScaler` uses the median and the interquartile range for scaling. This approach makes the `RobustScaler` more robust in the presence of outliers compared to other scalers (Müller and Guido 2016, pp. 133).

```
from sklearn.preprocessing import RobustScaler
transformer = RobustScaler().fit(train_x)
train_x = transformer.transform(train_x)
test_x = transformer.transform(test_x)
```

3.2 Exploratory data analysis

In this section, we summarize the exploratory data analysis steps performed in the dataset, focusing on descriptive statistics, visualizations, and relationships between key variables. The main goal is to gain initial insights and uncovering underlying patterns in the dataset. This EDA was conducted before feature scaling or transformations.

3.2.1 Descriptive statistics

We started by examining the basic summary statistics of the numeric variables in the dataset to understand the range, mean, and spread of the features as well as detect potential outliers.

Variable	Living Area	Year built	Rooms	Total price	Price per square meter	Maintenance fee
Mean	70.22	1980.96	2.65	199 858.70	3 020.35	206.86
Std Dev	35.46	25.18	1.18	110 041.50	1 167.58	114.72
Min	12.00	1785	1	6 000.00	53.00	0.00
25th Percentile	48.00	1965	2	128 000.00	2 153.85	143.38
Median (50th)	62.50	1980	2	176 000.00	2 901.41	209.30
75th Percentile	83.00	2002	3	242 000.00	3 703.70	275.20
Max	889.00	2024	8	1 950 000.00	11 533.33	1 133.30

Table 3.1. Summary statistics for numeric variables after data cleaning

Table 3.1 includes the mean, standard deviation, minimum, and maximum values, as well as the 25th, 50th, and 75th percentiles for all numeric variables. We were able to detect and clean some outliers using these statistics. We can also see skewness in the house prices, as well as in the maintenance fee and living area variables.

3.2.2 Visualizing the distribution of categorical variables

We first visualized the distribution of the property condition and type using a bar plot.

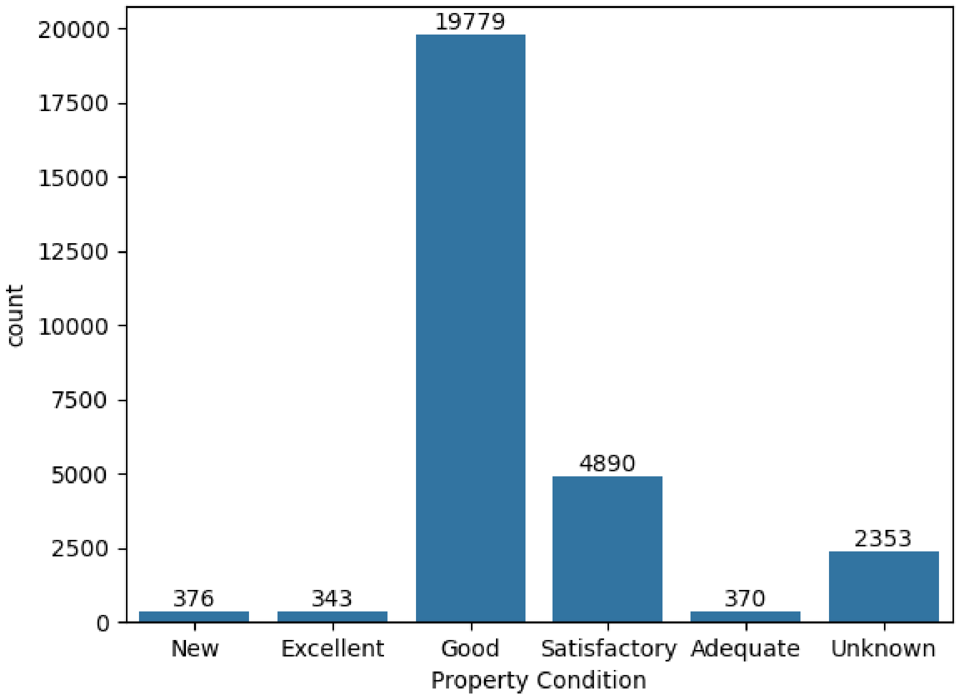

Figure 3.1. Bar plot of property condition variable.

from Figure 3.1 we can see that the vast majority of properties are evaluated as Good with only small portion being New, Excellent or Adequate. Furthermore, we can see from Figure 3.2 that most properties are either apartments (Kerrostalo) or row houses (Rivitalo).

3.2.3 Visualizing relationships between variables

We used a scatter plot to examine the relationship between the living area and the target variable. This plot helps to identify potential correlations or outliers between these two variables.

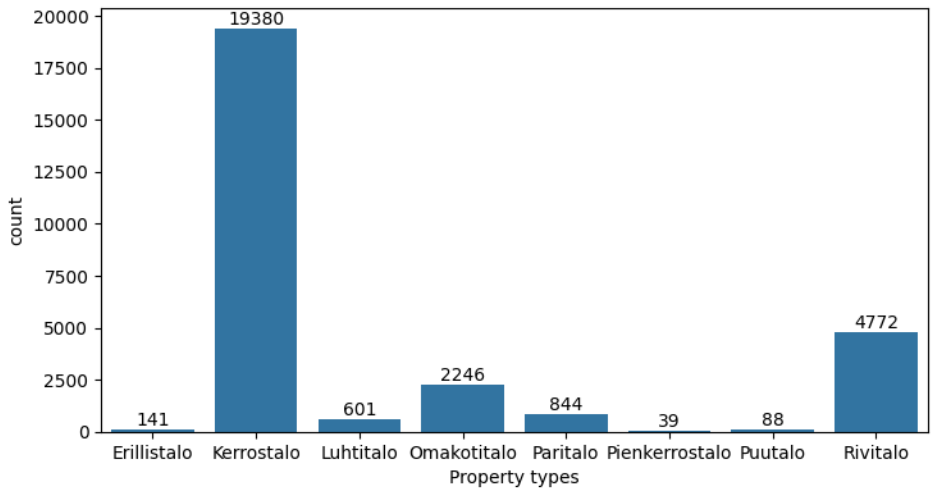

Figure 3.2. Bar plot of property types

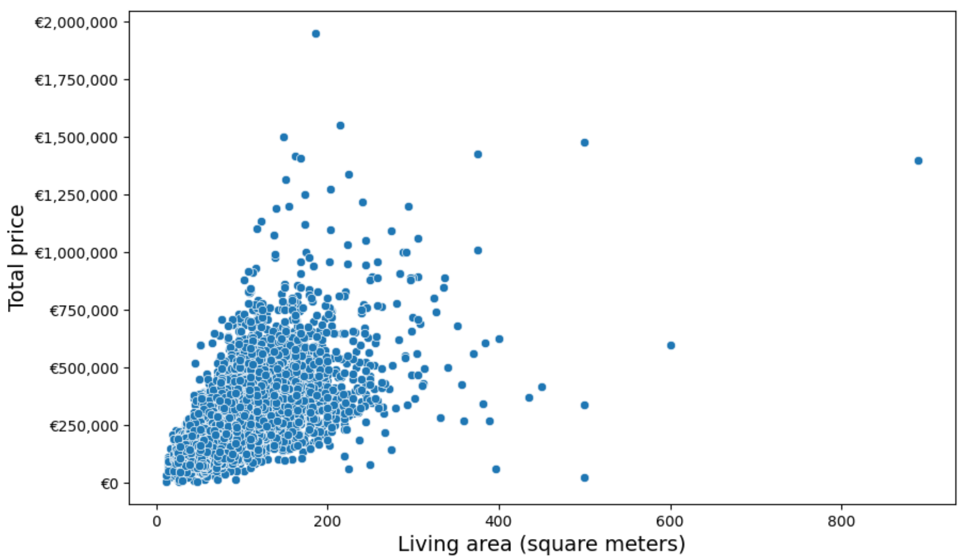

Figure 3.3. Scatter plot of living area vs. total price

From Figure 3.3, we observe a clear linear relationship between these variables, as well as the high variability in the price. Outlier values were examined, but none were removed or changed, as they could not be definitively classified as erroneous.

A box plot was used to visualize the distribution of price per square meter for the five most common property types. This allows us to visualize of the price distribution across different property types.

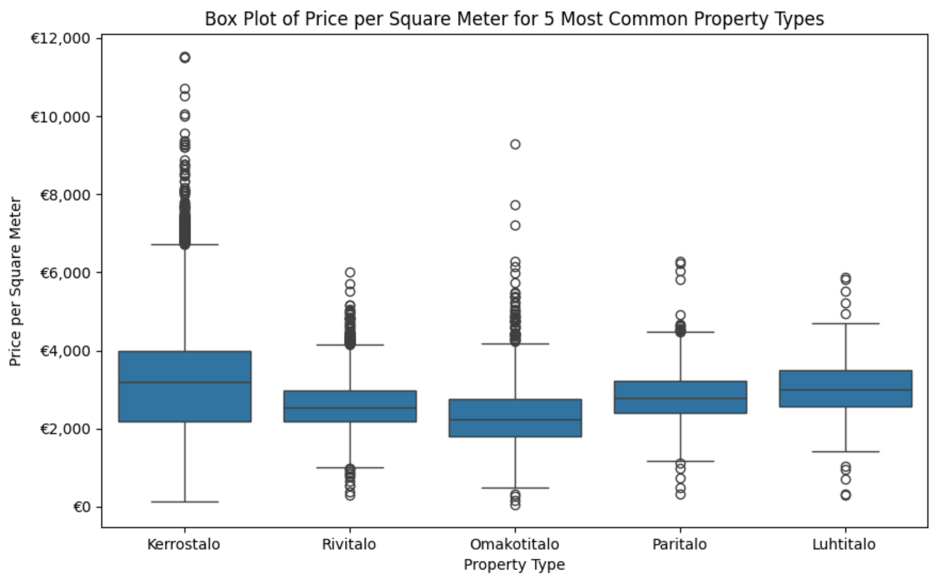

Figure 3.4. Box plot of property type vs. price per square meter

We can see from Figure 3.4 the distributions are similar except for apartments which has higher variance. In addition, apartments (Kerrostalo) and balcony access blocks (Luhtitalo) have higher price per square meter.

We visualized the relationship between property condition and price per square meter using a box plot. This plot helps us understand how the condition of a property impacts its price.

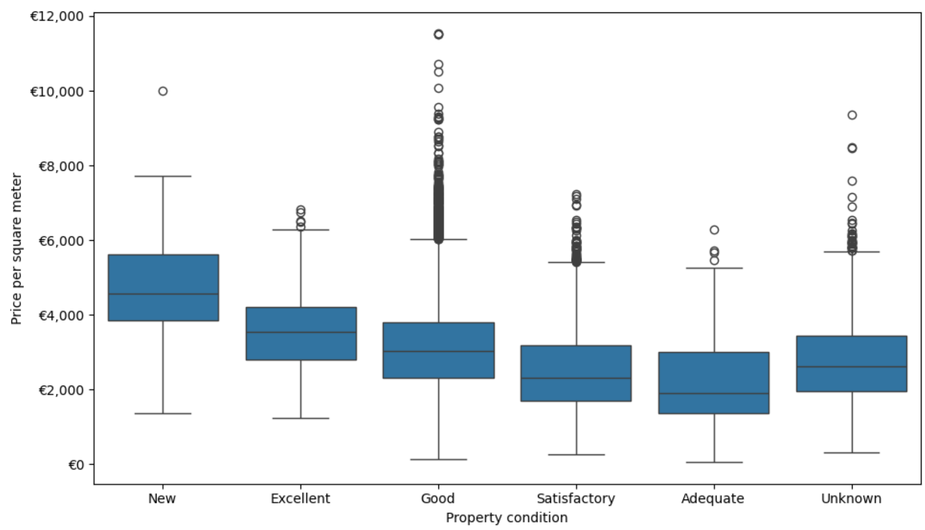

Figure 3.5. Box plot of property condition vs. price per square meter (`Neliöhinta`)

From Figure 3.5 we can clearly see that as the property condition goes down its price per square meter decreases.

We also examined the relationship between the year of construction and price per square meter using a scatter plot.

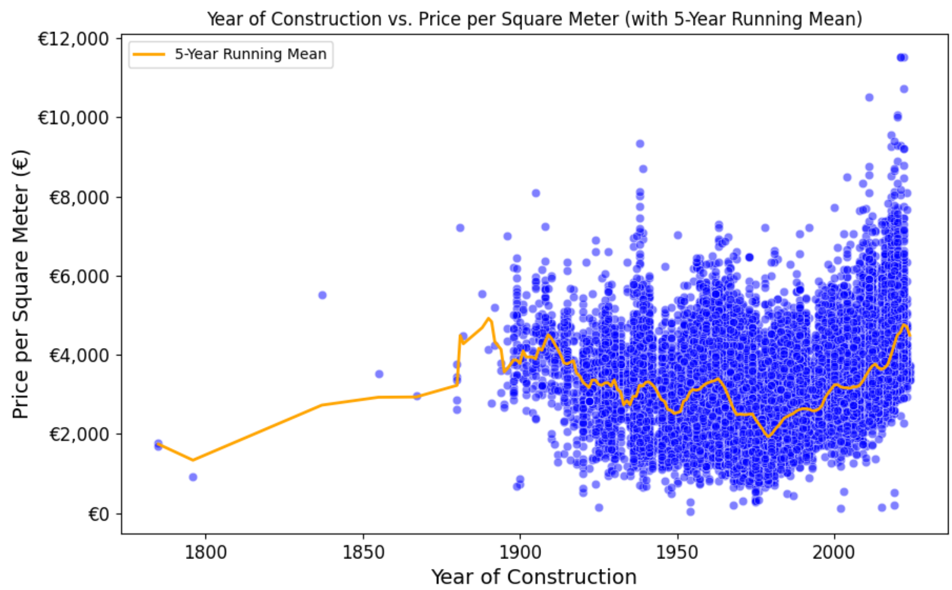

Figure 3.6. Scatter plot of year of construction vs. price per square meter

From Figure 3.6, it is unsurprising to see that newly built houses are generally more expensive. However, interestingly, houses that are around 100 years old or older appear to be more expensive on average as well. This trend could be due to a lack of data for houses that old, which might skew the results.

3.2.4 Distribution of house prices

A density plot was used to examine the distribution of the total price. This visualization helps to identify the distribution and skewness of the target variable.

From Figure 3.7 we can see that the distribution appears to be otherwise close to normal, but it shows a fair amount of skewness. We used a normal QQ plot to test how well the total price variable follows a normal distribution. Any deviations from the diagonal line indicate non-normality.

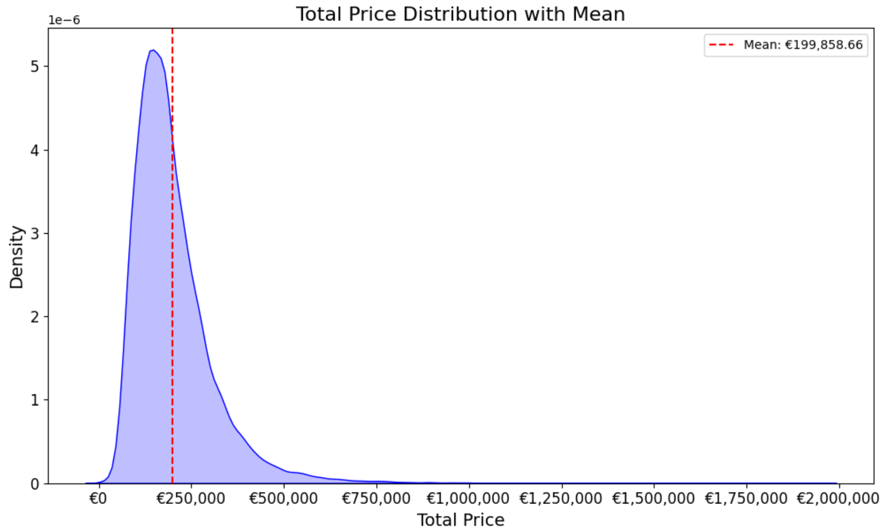

Figure 3.7. Density plot of total price

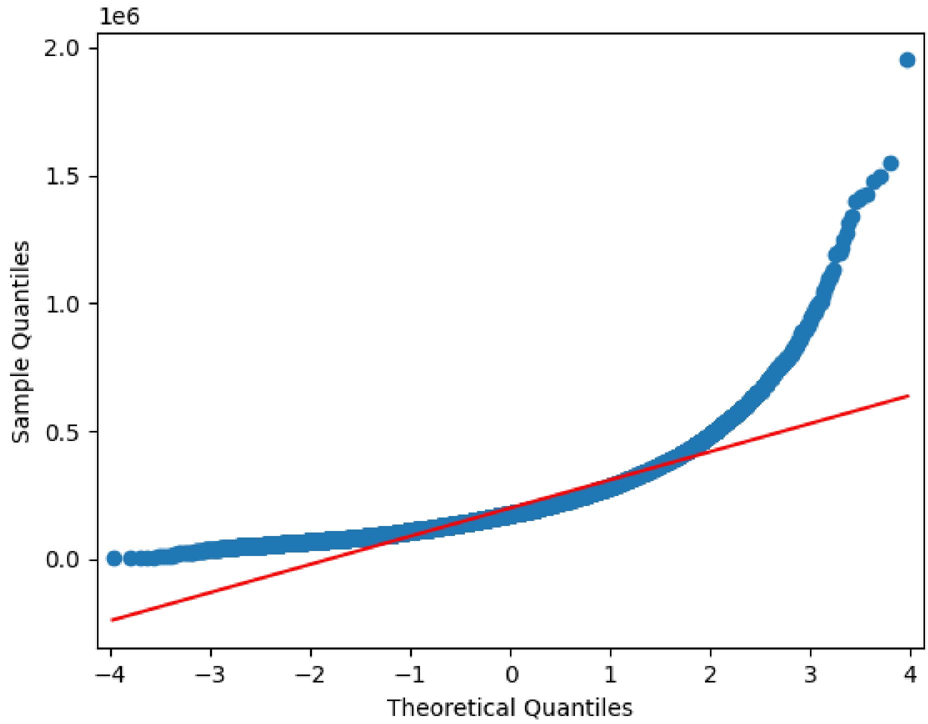

Figure 3.8. QQ plot of total price

From Figure 3.8 we can see deviation from normality in each tail ends of the graph. Because of this we applied a log transformation and visualized the results using a QQ plot.

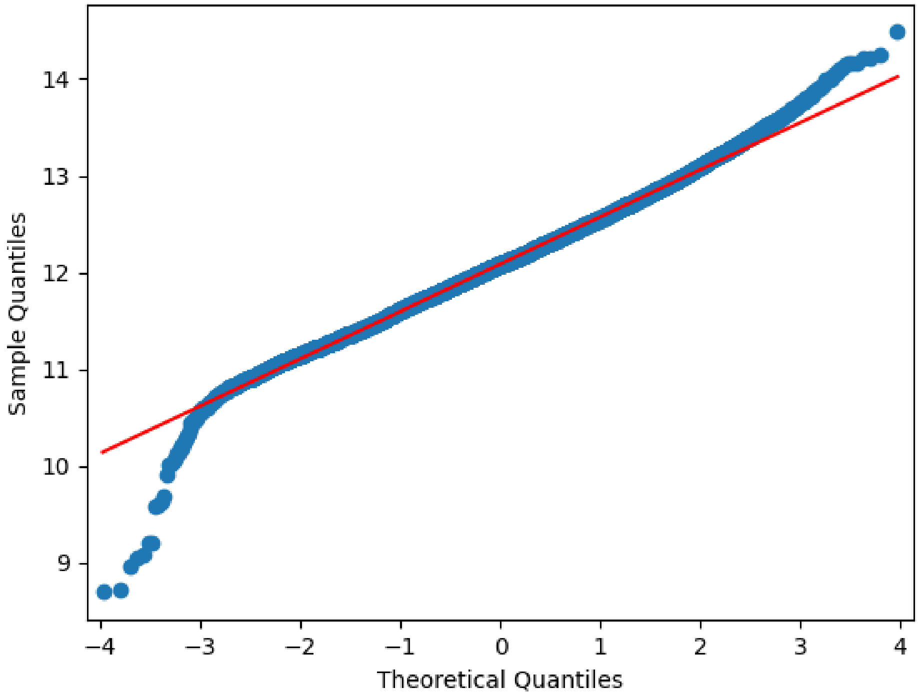

Figure 3.9. QQ plot of log-transformed total price

As we can see from Figure 3.9 the target variable now follows a normal distribution more closely. Only the lowest quantile values show non-normality. The transformed data is more suitable for modeling (Müller and Guido 2016, pp. 232).

3.3 Theory of machine learning models

This section contains the theoretical background of the machine learning models implemented in this thesis.

3.3.1 Model stacking

Model stacking, or stacked generalization (Wolpert 1992), is an ensemble learning technique that combines multiple predictive models to improve overall performance. The key idea behind stacking is to benefit from the strengths of different base models, allowing them to compensate for each other's potential biases and errors, leading to a more robust overall prediction. Breiman (1996) extended the concept of stacking to regression tasks by combining the predictions of base models using a simple linear model.

A typical stacking regression framework consists of two levels: the first level contains several base regression models, and the second level, often referred to as the meta-model, uses their outputs as inputs to learn the optimal way to make the final prediction (Brazdil et al. 2022, pp. 173–175). For example, the weights of a simple linear regression meta-model are computed by:

$$\hat{w} = \arg\min_{w} \sum_{i=1}^{N} \left[y_i - \sum_{m=1}^{M} w_m \hat{f}_m^{-i}(x_i) \right]^2 ,$$

with $\hat{f}_m^{-i}(x_i)$ being the prediction of the base model m on the dataset with the current observation x_i removed and M is the number of base models (Hastie et al. 2009, pp. 290). To avoid training the M base models N times, it is common to use k-fold cross-validation for training the base models. This approach still prevents overfitting by ensuring that the predictions for each data point comes from a model trained on a subset of the dataset that excludes that point, while also reducing the computation required, as the base models are trained only k times instead of N. The stacking regressor framework is illustrated in Figure 3.10 (Raschka 2018).

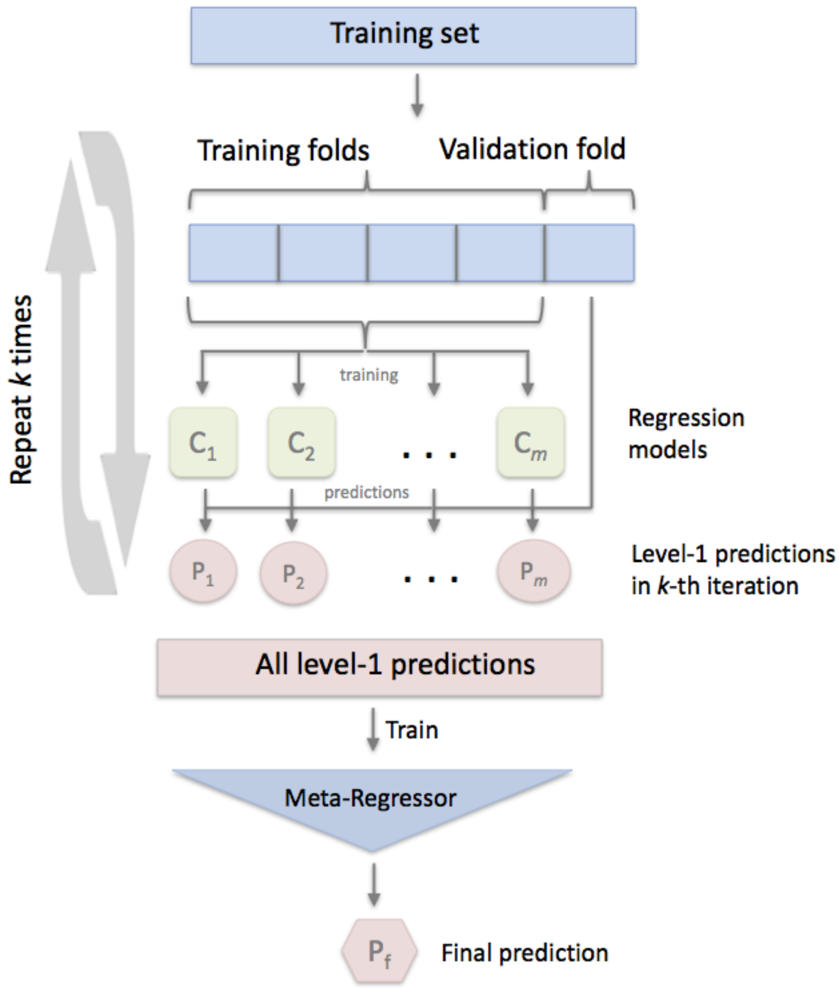

Figure 3.10. Stacking regressor framework (Raschka 2018)

Naturally, a question arises about the choice of an appropriate meta-model. In principle, any learning method can be applied. However, in practice, since most of the data complexity is explained by the base models and the meta-model only acts as a sort of final judge on how much trust to place in each base model's prediction, a simple linear model such as ridge regression is a suitable choice. (Witten et al. 2011, pp. 370–371)

A drawback of stacking is the lack of a standardized procedure for the choice of the stacking architecture. Some studies have explored more complex designs, such as multi-leveled stacking (Nnadozie et al. 2022) or the use of additional meta-features in the meta-level (Sill et al. 2009; Džeroski and Ženko 2004). However, the benefits of these approaches remain debated, as they tend to increase computational complexity while yielding only marginal gains in predictive performance. Moreover, studies in this area often differ in their experimental design and datasets, which complicates direct comparison and generalization of results.

For this study, five diverse base models were chosen based on their unique strengths, which are discussed in the Literature review section. These models are implemented in the usual stacking framework. Ridge regression was chosen as the meta-model because no additional meta-features were used, keeping the meta-level task relatively straightforward.

3.3.2 Voting regression

Voting regression, or averaging, is a simple ensemble method that utilizes the predictions of multiple different regressors. Given M different regression models $\{f_1, \ldots, f_M\}$, with $f_i(x)$ being the prediction of regressor i for data point x, the simple voting regression output is:

$$F(x) = \frac{1}{M} \sum_{i=1}^{M} f_i(x). \tag{3.1}$$

We can introduce weights w_i to each model in Equation 3.1:

$$F(x) = \sum_{i=1}^{M} w_i f_i(x), \tag{3.2}$$

where the weights are usually constrained by

$$w_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{M} w_i = 1.$$

(Zhou 2012, pp. 68–70.) The difficulty lies in defining the weights in Equation 3.2, and it is debated whether they offer significant improvements over simple averaging. Generally speaking, if the individual regressors all perform roughly equally well, then simple averaging is appropriate. On the other hand, if the models perform with varying success, then a weighted approach might achieve better results (Zhou 2012, pp. 71). Methods for defining weights usually include metrics on individual model performance on the training set, for example Chen and Zheng (2025) used relative root mean squared error rates to define the weights for averaging. In this study we use simple voting regression as a comparison to model stacking.

3.3.3 Ridge regression

Ridge regression is a regularization technique for linear regression models that introduces a penalty to shrink the model coefficients in proportion to their size. This method aims to improve prediction accuracy at the cost of higher bias. Ridge regression can be applied to help combat problems of collinearity and over-fitting. Mathematically, the ridge coefficients are obtained by minimizing a penalized ordinary least squares loss function:

$$\hat{\beta} = \arg\min_{\beta} \left\{ \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}. \tag{3.3}$$

- y_i: The observed values of the dependent variable.
- $\hat{y}_i$: The estimated value of the dependent variable for the i-th observation.
- β_j: The regression coefficients for the j-th predictor.
- λ: The regularization parameter, where $\lambda \geq 0$, controlling the strength of the penalty applied to the coefficients.

The second term, $\lambda \sum_{j=1}^{p} \beta_j^2$, is the penalty term, also known as L2 regularization. Since the coefficients are squared, larger coefficients are penalized more than smaller ones. The parameter λ serves as a tuning parameter that controls the magnitude of the penalty. The selection of λ is critical, as it influences the trade-off between bias and variance. Small values of λ result in a solution that closely approximates OLS regression, while larger values shrink the coefficients more significantly, introducing bias but potentially improving predictive accuracy. Cross-validation can used to determine optimal value of lambda. (Kuhn and Johnson 2013, pp. 123) Because the ridge regression solutions are not equivariant under scaling of the input variables, one usually standardizes the inputs before solving the Equation 3.3 (Hastie et al. 2009, pp. 63).

3.3.4 Lasso

Similar to ridge regression, the least absolute shrinkage and selection operator (lasso) is a regularization technique that uses a slightly different penalty term in the loss function (Tibshirani 1996):

$$\hat{\beta} = \arg\min_{\beta} \left(\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right). \tag{3.4}$$

The goal of lasso is to have the benefits of ridge regression of increasing prediction accuracy by shrinking the model coefficients while also allowing coefficients to be set to zero, thus working as a regularization technique and a feature selection method simultaneously (Tibshirani 1996). The penalty term in lasso causes the solutions to be non-linear, although efficient computing algorithms for solving the coefficients as λ is varied have been developed (Hastie et al. 2009, pp. 68–69).

3.3.5 Support vector regression

The underlying theory for support vector machines (SVMs) was developed by Vladimir Vapnik and Alexey Chervonenkis during the 1960s. The modern formulation of SVMs was published by Vapnik and Cortes (1995). Originally, the theory was developed for classification tasks, however, it was later extended to regression tasks by Vapnik in 1995 and further developed by Drucker et al. (1997). Much like the previously reviewed regression methods, the goal in support vector regression (SVR) is to minimize a loss function. The difference is that we are indifferent to errors smaller than some threshold ϵ. This is called the epsilon-insensitive loss function (Vapnik 2000, pp. 181–182).

Mathematically, SVR attempts to find a function $f(x) = x'w + w_0$ that minimizes the following objective:

$$\min_{w,w_0} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*), \tag{3.5}$$

where

- w is the weight vector,
- ξ_i and ξ_i^* are slack variables that allow some points to lie outside the epsilon margin,
- C is a regularization parameter controlling the bias-variance trade-off.

These are subject to constraints:

$$y_i - wx_i - w_0 \leq \epsilon + \xi_i$$

$$wx_i + w_0 - y_i \leq \epsilon + \xi_i^*$$

$$\xi_i, \xi_i^* \geq 0 \quad \forall i.$$

y_i represents the target value for each data point x_i, and ϵ defines the margin of error within which the size of the errors do not effect the loss function. Higher values of the regularization parameter C decrease bias and increase variance, while lower values do the opposite.

The corresponding Lagrangian dual formula can be shown to have a form where we minimize

$$\frac{1}{2}\sum_{i,j=1}^{N}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)\langle x_i, x_j\rangle + \epsilon\sum_{i=1}^{N}(\alpha_i^* + \alpha_i) - \sum_{i=1}^{N} y_i(\alpha_i^* - \alpha_i) \tag{3.6}$$

subject to the constraints

$$\sum_{i=1}^{N}(\alpha_i^* - \alpha_i) = 0$$

$$0 \le \alpha_i \le C, \quad 0 \le \alpha_i^* \le C \quad \forall i,$$

where α_i^* and α_i are Lagrange multipliers. The subset of the solution values $(\alpha_i^* - \alpha_i)$ that are nonzero are called the support vectors and they can be used to predict new values with

$$\hat{f}(x) = \sum_{i=1}^{N}(\alpha_i^* - \alpha_i)\langle x, x_i\rangle + w_0.$$

(Drucker et al. 1997; Hastie et al. 2009, pp. 435–436)

We can estimate nonlinear relationships in the data using the so-called kernel trick. Since our solution depends only on the inner products $\langle x_i, x_j\rangle$, we can solve the problem in a higher-dimensional feature space, where the solution becomes more linear. Using a kernel function $K(x_i, x_j)$, we can calculate the inner products in this transformed feature space without having to explicitly compute the transformed data points. Now the Equation 3.6 becomes

$$\frac{1}{2}\sum_{i,j=1}^{N}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)K(x_i, x_j) + \epsilon\sum_{i=1}^{N}(\alpha_i^* + \alpha_i) - \sum_{i=1}^{N} y_i(\alpha_i^* - \alpha_i) \tag{3.7}$$

and similarly we can predict new values with

$$\hat{f}(x) = \sum_{i=1}^{N}(\alpha_i^* - \alpha_i)K(x, x_i) + w_0.$$

The kernel function should be a positive semidefinite function. (Hastie et al. 2009, pp. 423–424, 436) A common choice, and one used in our models, is the radial basis function (RBF), defined as:

$$K(x, x_i) = \exp\left(-\gamma\|x - x_i\|^2\right), \tag{3.8}$$

where γ is a tuning parameter governing the width of the kernels. A small value of gamma implies wide and flat kernels resulting in a smoother model and the opposite is true for large values of gamma. For further details see Hastie and Zhu (2006). Fitting SVR to a nonlinear data using RBF kernel is visualized in Figure 3.11.

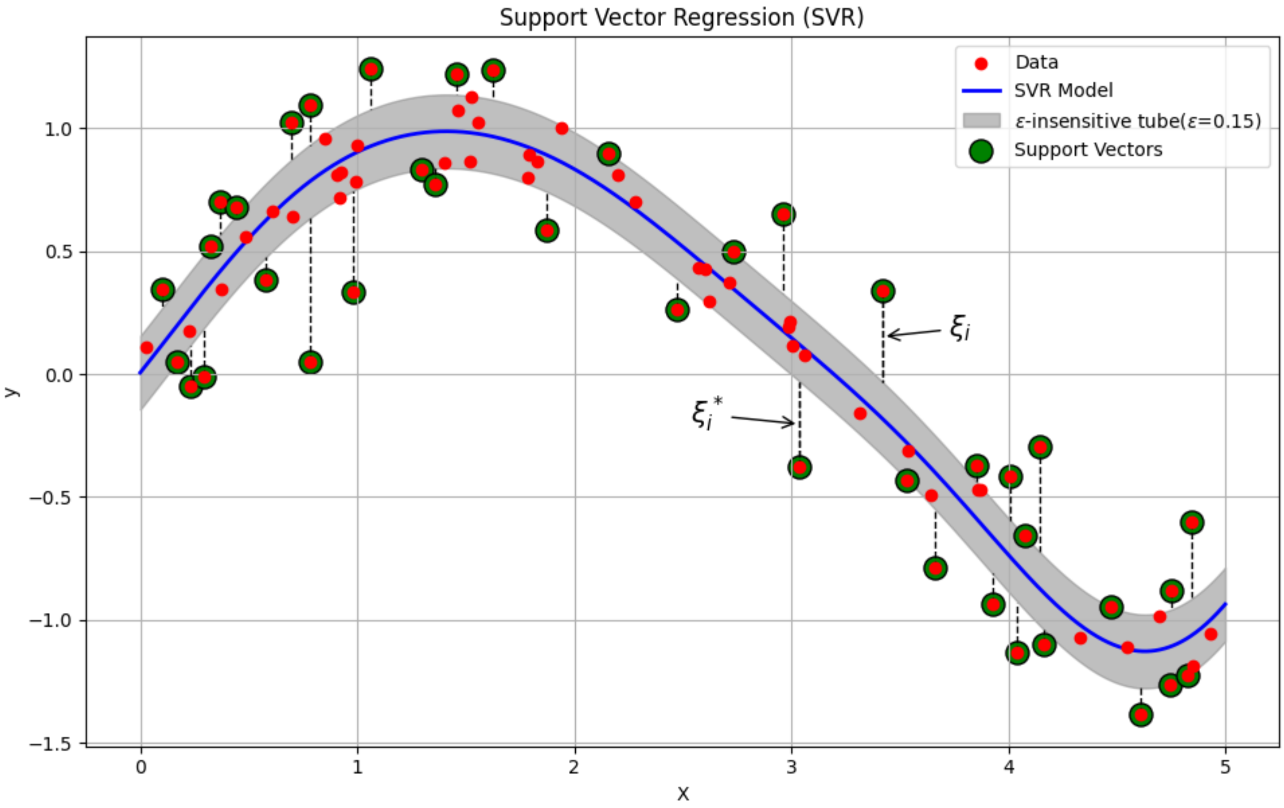

Figure 3.11. Support vector regression visualized with the epsilon-insensitive tube and slack variables ξ_i and ξ_i^*. Data was simulated from a noisy sinusoidal function.

3.3.6 Gradient boosted regression trees

In order to explain XGBoost, we will first look at the simpler gradient tree boosting method. In regression trees, the feature space is partitioned into rectangular areas, where new data points are given a prediction based on the samples in that area. The simplest prediction is just the average value of the samples in that rectangle. To find the partitions, we start with the entire dataset $D = (Y, X)$, where $Y = (y_1, y_2, \ldots, y_n)$ is the dependent variable, and $X = [x_{ij}]$ are the predictors, with $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, p$, where p is the number of features. The data is then split into two by going through every predictor X_j and trying every value of those predictors as the split point s. We select the split variable j and split point s by minimizing the sum of squared errors. (Hastie et al. 2009, pp. 307)

$$\min_{j,s} \left[\sum_{x_i \in R_1(j,s)} (y_i - \bar{y}_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - \bar{y}_2)^2 \right], \tag{3.9}$$

where

$$R_1(j,s) = \{X \mid X_j \le s\}, \quad R_2(j,s) = \{X \mid X_j > s\}$$

and $\bar{y}_1$ and $\bar{y}_2$ are the average value of the data points in the regions R_1 and R_2.

We continue this process iteratively on each of the regions. In order to prevent overfitting after the tree has been grown, it is pruned back to a smaller size, while minimizing the loss in predictive performance. Breiman et al. (1984) describes a method for pruning large trees in such a way, called cost-complexity pruning. The cost-complexity of a tree is defined as

$$C_\alpha(T) = \sum_{l=1}^{L} \sum_{x_i \in R_l} (y_i - \bar{y}_l)^2 + \alpha L. \tag{3.10}$$

- l is a terminal node (leaf node) of the tree T.
- R_l is the region corresponding to node l.
- $\bar{y}_l$ is the average value of data points in the region R_l.
- L is the number of terminal nodes in the tree.
- α is the regularization parameter.

We can find the smallest subtree $T_\alpha \subset T_0$ that minimizes $C_\alpha(T)$ for a given value of α. The regularization parameter governs the size of the optimal subtree. A small value of α results in a larger tree with $\alpha = 0$ resulting in the original tree T_0. Conversely a larger value α results in smaller trees. We can use cross-validation to choose the value of α. (Hastie et al. 2009, pp. 308)

We can now formally represent trees as

$$T(x; \{R_l\}_{l=1}^{L}) = \sum_{l=1}^{L} \gamma_l I(x \in R_l), \tag{3.11}$$

where γ_l is the prediction in the region R_l. In gradient tree boosting trees are fit in a forward stagewise manner and the boosted tree model is a sum of the built trees

$$F_m(x) = \sum_{m=0}^{M} T(x; \{R_{lm}\}_{l=1}^{L}). \tag{3.12}$$

In each step we must solve the regions and corresponding constant predictions for the next tree $\Theta_m = \{R_{lm}, \gamma_{lm}\}_{l=1}^{L}$

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} \mathcal{L}\left(y_i, F_{m-1}(x_i) + \sum_{l=1}^{L} \gamma_{lm} I(x_i \in R_{lm})\right), \tag{3.13}$$

where F_{m-1} is the current model. (Hastie et al. 2009, pp. 356–357) Finding the regions for (3.13) is a difficult problem, so instead we fit the next tree in each step to the negative gradient $-g_{im}$ or to so called "pseudoresponses" or "pseudoresiduals" using squared error loss (Friedman 1999, pp. 1193). Thus, (3.13) becomes

$$\tilde{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} (-g_{im} - \sum_{l=1}^{L} \gamma_{lm} I(x_i \in R_{lm}))^2, \tag{3.14}$$

where

$$g_{im} = \left[\frac{\partial \mathcal{L}(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)} \right]$$

for any differentiable loss function $\mathcal{L}$. (Hastie et al. 2009, pp. 359) Because the tree predicts constant values in each distinct region, the Equation 3.13 reduces to

$$\hat{\gamma}_{lm} = \arg\min_{\gamma} \sum_{x_i \in R_{lm}} \mathcal{L}\left(y_i, F_{m-1}(x_i) + \gamma\right), \tag{3.15}$$

when finding predictions given the regions R_{lm} (Friedman 2002, pp. 368-369).

For squared-error loss it can be shown that solution for (3.15) is

$$\hat{\gamma}_{lm} = \arg\min_{\gamma} \sum_{x_i \in R_{lm}} (y_i - F_{m-1}(x_i) - \gamma)^2$$

$$\frac{d}{d\gamma} \sum_{x_i \in R_{lm}} (y_i - F_{m-1}(x_i) - \gamma)^2 = 0$$

$$\gamma = \frac{1}{|R_{lm}|} \sum_{x_i \in R_{lm}} r_{lm}.$$

So optimal γ_{lm} is the average residuals r_{lm} in each region R_{lm}. Finally our current model $F_{m-1}(x)$ is updated in each region with

$$F_m(x) = F_{m-1}(x) + \nu \cdot \sum_{l=1}^{L} \hat{\gamma}_{lm} I(x \in R_{lm}),$$

where $0 < \nu \leq 1$ is a regularization parameter that can be thought of as the learning rate of the process (Friedman 1999, pp. 1203).

3.3.7 Extreme gradient boosting

Extreme gradient boosting (XGB) expands on gradient boosting from previous section by adding a regularization term to the objective (3.13) and by using second-order gradients to optimize the objective function. Formally the XGB objective in the m-th iteration for the next tree $\Theta_m = \{R_{lm}, w_{lm}\}_{l=1}^{L}$ is

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} \mathcal{L}(y_i, F_{m-1}(x_i) + F_m(x_i)) + \Omega(F_m), \tag{3.16}$$

where

$$\Omega(F) = \alpha L + \frac{1}{2}\lambda \|w\|^2, \tag{3.17}$$

with w being the leaf weights, αL being the cost complexity terms like in (3.10) and λ corresponding to L2 regularization used in ridge regression. Using the second-order Taylor approximation we get

$$\tilde{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} \left[\mathcal{L}(y_i, F_{m-1}(x_i)) + g_{im} F_m(x_i) + \frac{1}{2} h_{im} F_m(x_i)^2 \right] + \Omega(F_m), \tag{3.18}$$

Where g_{im} and h_{im} are the first and second-order gradients on the loss function respectively. The second-order gradient or Hessian is calculated as

$$h_{im} = \left[\frac{\partial^2 \mathcal{L}(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)^2} \right]. \tag{3.19}$$

The use of the second-order gradient, also known as Newton's method, allows for more precise and efficient optimization of the objective function (Goodfellow et al. 2016, pp. 89–92). Expanding Ω and $F(x)$, (3.18) changes to

$$\tilde{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} \left[g_{im} \sum_{l=1}^{L} w_{lm} I(x_i \in R_{lm}) + \frac{1}{2} h_{im} \sum_{l=1}^{L} w_{lm}^2 I(x_i \in R_{lm}) \right] + \alpha L + \frac{1}{2}\lambda \sum_{l=1}^{L} w_{lm}^2.$$

Simplifying further we get

$$\tilde{\Theta}_m = \arg\min_{\Theta_m} \sum_{l=1}^{L} \left[G_{lm} w_{lm} + \frac{1}{2}(H_{lm} + \lambda) w_{lm}^2 \right] + \alpha L, \tag{3.20}$$

where $G_{lm} = \sum_{i \in I_{lm}} g_{im}$ and $H_{lm} = \sum_{i \in I_{lm}} h_{im}$ with $I_{lm} = \{i | x_i \in R_{lm}\}$. From here we can calculate for a given tree structure the optimal weights by differentiating with respect to w_{lm} and setting the equation to zero.

$$\tilde{w}_{lm} = -\frac{G_{lm}}{H_{lm} + \lambda}. \tag{3.21}$$

Plugging the optimal weights to (3.20) we can calculate the optimal value as

$$\tilde{\Theta}_m = -\frac{1}{2} \sum_{l=1}^{L} \frac{G_{lm}^2}{H_{lm} + \lambda} + \alpha L. \tag{3.22}$$

This equation can be used to measure the quality of a tree. As before when constructing a tree, we use a greedy approach iteratively adding nodes to the tree. To find the optimal splits we maximize the gain score

$$\text{gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} + \frac{G^2}{H+\lambda}\right] + \alpha L, \tag{3.23}$$

where G_R, G_L and H_L, H_R are the sum of gradients and Hessians on the sets of right and left indices after the split respectively. (Chen and Guestrin 2016)

Additionally, XGB can use an approximate algorithm for finding splits instead of the exact greedy one. This can speed up computations when the dataset is large. Furthermore, XGB can handle sparse data during the split-finding process by learning an optimal default direction at each tree node. If a data value is missing, it is classified into the default direction. XGB also implements feature subsampling, a technique used in Random Forests, which helps prevent overfitting. To speed up computations further, XGB supports parallel learning by storing the data in in-memory units called blocks. For more details, refer to the original paper (Chen and Guestrin 2016).

3.3.8 CatBoost

CatBoost, developed by Yandex, is a gradient boosting tree algorithm designed to handle categorical features effectively and to overcome overfitting caused by target leakage present in gradient boosting methods such as XGBoost discussed in the previous section.

One of the challenges faced by machine learning algorithms is the handling of categorical variables. In general, machine learning algorithms require numerical input, which requires the encoding of categorical features into numeric values, for example, using one-hot encoding. However, one-hot encoding can result in a huge number of features and sparse data. In their paper, Prokhorenkova et al. consider target statistics to be an efficient method to address this issue. Target statistics simply replace the categories x_k^i by estimating the expected target value $\hat{x}_k^i$ of the k-th training example for each categorical feature i. A simple approach is to estimate it as the average value of y for data points with the same category x_k^i, with a smoothing prior p, as:

$$\hat{x}_k^i = \frac{\sum_{j=1}^n I_{\{x_j^i = x_k^i\}} \cdot y_j + ap}{\sum_{j=1}^n I_{\{x_j^i = x_k^i\}} + a},$$

where $a > 0$ is a smoothing parameter, and p is commonly the average target value in the dataset. (Prokhorenkova et al. 2019)

An issue arises with this technique as the target statistics are calculated using y_k which is the target for x_k, resulting in target leakage, which can cause overfitting. To address this, CatBoost uses ordered target statistics. The idea is to compute the target statistic for x_k on a subset of examples $D_k \subset D \setminus \{x_k\}$ excluding x_k. This is effectively done by introducing an artificial "time" through a random permutation σ

of the training examples. For each training example k, the target statistic is computed using the preceding examples or "history" in the permutation, which is expressed as $D_k = \{x_j : \sigma(j) < \sigma(k)\}$.

$$\hat{x}_k^i = \frac{\sum_{x_j \in D_k} I_{\{x_j^i = x_k^i\}} \cdot y_j + ap}{\sum_{x_j \in D_k} I_{\{x_j^i = x_k^i\}} + a}.$$

Another issue that Prokhorenkova et al. (2019) address is the target leakage caused by calculating the gradients in tree boosting. The so-called prediction shift occurs because, at each iteration of the boosted tree, the next tree is estimated using the gradient which relies on the target values of the same data points the current tree model F_{m-1} was built on. Subsampling, which is also used by XGBoost does not fully solve this problem. Instead, CatBoost introduces the concept of ordered boosting, which is closely related to the idea of ordered target statistics. In ordered boosting, CatBoost constructs each tree in a way that the gradients for a given data point are based only on data that came before it in a randomly sampled permutation of the training data. This ordered approach is applied both to the computation of target statistics for categorical features and to the gradient estimation in boosting. To reduce the potential variance that can be caused by relying on a single permutation, CatBoost generates multiple random permutations. First, $s + 1$ permutations of the training data are created. Permutations $\sigma_1, ..., \sigma_s$ are used for tree construction and σ_0 is used for computing the leaf values of the final tree. At each iteration of the boosting algorithm, a permutation σ_r is sampled from $\{\sigma_1, ..., \sigma_s\}$ and used to calculate the target statistics for categorical variables. Then the next tree is constructed using gradients that are calculated using only the previous examples in the permutation. That tree is then used to update our current model. For further details and the full algorithm, see (Prokhorenkova et al. 2019).

3.3.9 Feed-forward neural network

A neural network consists of layers of nodes (or neurons) that take numerical inputs and output a value based on those inputs. These nodes are connected to form a network in such a way that the outputs of one layer become the inputs for the next. In a regression task, the final layer consists of a single neuron, and its output attempts to estimate the target variable. The first layer of nodes also called the input layer represents the feature vector of our dataset. The layers between the input and output layers are called the hidden layers. In a simple feed-forward neural network (FNN), information flows in only one direction, meaning there are no loops in the network. We can represent the nodes in the hidden layers as weighted sum of inputs plus a bias term. The weights and bias govern the output value of each node. Mathematically, we write the j-th neuron $z_j^{(l)}$ of the layer l as:

$$z_j^{(l)} = \sum_{i=1}^{K} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)},$$

where

- $w_{ji}^{(l)}$: The weight associated with the connection between the i-th neuron in the previous layer $(l-1)$ and the j-th neuron in layer l. $i = \{1, ..., K\}$
- $a_i^{(l-1)}$: The activation of the i-th neuron in the previous layer $(l-1)$.
- $b_j^{(l)}$: The bias term for the j-th neuron in layer l.

These neurons are then transformed using a non-linear differentiable activation function $f(z)$, which produces the output of the neuron. (Bishop 2006, pp. 227)

$$a_j^{(l)} = f(z_j^{(l)}).$$

An example of a common activation function is the ReLU (rectified linear unit): $f(x) = \max(0, x)$. For regression tasks, the final output neuron does not have an activation function, allowing the output to yield any real number.

To learn optimal weights and biases for each neuron we need to minimize the loss function given N data points.

$$\sum_{i=1}^{N} L(y_i, \hat{y}_i),$$

where $\hat{y}_i$ is the prediction of our network. Since finding the minimum analytically is infeasible, we use gradient descent to take steps in the direction that maximally minimizes the loss. To do this, we compute the partial derivatives of the loss function with respect to the weights and to the biases and update those parameters by taking a step towards the negative gradient. We start by computing the error at the output layer and calculating the partial derivatives for the nodes using the chain rule.

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}}$$

$$\frac{\partial L}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}.$$

Using these derivatives, we can then calculate the partial derivatives for the neurons in the previous layer starting with the output.

$$\frac{\partial L}{\partial a_i^{(l-1)}} = \sum_{j=1}^{N_l} \frac{\partial L}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial a_i^{(l-1)}},$$

where N_l is the number of nodes in the layer l. This process continues iteratively through each layer, from the output layer back to the input layer, updating the gradients at each step. This method of propagating the gradient from output backwards is known as backpropagation. Finally, we update the weights using the computed partial derivatives.

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \cdot \frac{\partial L}{\partial w_{ji}^{(l)}}$$

$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \frac{\partial L}{\partial b_j^{(l)}},$$

where $\eta > 0$ is the learning rate. (Bishop 2006, pp. 236–245) Learning with gradient descent can be done in batches or online. A batch can refer to the entire dataset or to a minibatch, which consists of randomly and uniformly drawn samples from the training set (Goodfellow et al. 2016, pp. 152). Online learning, on the other hand, updates the models weights after each individual training sample. Learning is done by iterating over the training samples multiple times with one iteration being referred to as an epoch. It is common to initialize weights with small random values near zero, which makes the model start out nearly linear and gradually become nonlinear as training progresses. Starting with large weights can lead to poor solutions due to instability during training. Furthermore, standardizing inputs to have zero mean and unit variance improves training by ensuring balanced regularization across all input features and allowing for meaningful initialization of weights. (Hastie et al. 2009, pp. 397–400) Since the loss surface often contains many local minima, it may be necessary to run gradient descent multiple times with different random initializations to find a sufficient minimum (Bishop 2006, pp. 240).

To prevent overfitting, early stopping is often used, which halts training before the model reaches the global minimum of the loss function. This can be done with a validation set, as rising validation error signals overfitting. Another effective regularization method is dropout, which offers a computationally efficient way to prevent overfitting. During training, dropout randomly "drops" non-output neurons by setting their output to zero, effectively simulating different sub-networks each time a minibatch is processed. This process is repeated across training iterations using a random binary mask to determine which units are active. The probability of dropping nodes is a fixed hyperparameter. Instead of training multiple models independently, dropout trains sub-networks that shares parameters, making it more memory-efficient than traditional bagging. (Goodfellow et al. 2016, pp. 246–248, 258–259)

3.4 Research implementation and hyperparameter tuning

The research was conducted following the flowchart shown in Figure 3.12. Initially, the data was collected and then cleaned. Exploratory data analysis was performed on the cleaned dataset, which helped identify additional features that required cleaning and provided further insights for subsequent preprocessing steps. After preprocessing, initial models, XGBoost and CatBoost, were trained to assess feature importance. One feature that did not contribute to model predictions was dropped. Next, the models underwent hyperparameter tuning. The tuning process began with the base models, followed by stacking, where the meta-model was fine-tuned. Finally, the optimized models were trained and evaluated. The results were then analyzed, leading to conclusions that address the research questions.

The models were implemented in Python using several key libraries: `xgboost`, `catboost`, `scikit-learn`, and `TensorFlow Keras`. The hyperparameter tuning process was conducted using cross-validation techniques with `GridSearchCV` (GSCV), `LassoCV`, and `RidgeCV`. Cross-validation is a statistical technique used to assess the performance and generalizability of machine learning models. In k-fold cross-validation, the dataset is randomly divided into k equal-sized folds. The model is then trained k times, each time using $k-1$ folds for training and the remaining fold for testing. This process is repeated for each fold, and the overall performance is averaged across all k iterations (Hastie et al. 2009, pp. 241–242).

GSCV, a method from `scikit-learn`, automates the process of hyperparameter selection by exhaustively searching through a user-defined grid of parameters. It performs cross-validation for each combination of parameters and returns the set that yields the best performance based on a scoring metric, in this case, the negative mean squared error.

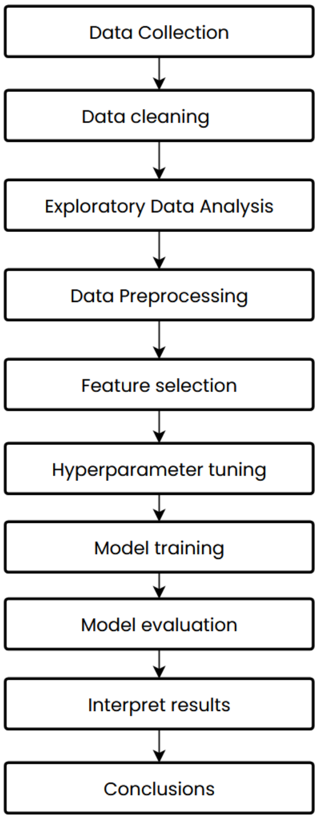

Figure 3.12. Flowchart depicting the research process.

GSCV was used to tune the hyperparameters for SVR and XGBoost. For CatBoost the built-in `grid_search` was used, which is functionally the same as GSCV. For XGBoost, three hyperparameters were tuned: `max_depth`, `n_estimators`, and `learning_rate`. The `max_depth` parameter controls the depth of individual decision trees, with deeper trees having more capacity to model complex relationships, but also a higher risk of overfitting. The `n_estimators` parameter determines the number of boosting rounds, or trees, to fit. The `learning_rate`, also known as `eta`, controls the step size shrinkage applied during the update of feature weights after each boosting step. Smaller values of `learning_rate` reduce the contribution of each new tree, making the boosting process more conservative and helping to prevent overfitting.

GSCV was performed using 5-fold cross-validation. The best combination of hyperparameters, based on the negative mean squared error, was found to be:

$$\texttt{learning\_rate} = 0.05, \quad \texttt{max\_depth} = 5, \quad \texttt{n\_estimators} = 1000$$

For CatBoost, the hyperparameters that were tuned included `depth`, `learning_rate`, and `l2_leaf_reg`. The `depth` and `learning_rate` parameters are similar to those in XGBoost. The `l2_leaf_reg` parameter controls the L2 regularization applied to the leaf values of the trees, helping to prevent overfitting by penalizing large leaf values.

The default 3-fold cross-validation was used for CatBoost. The best combination of hyperparameters was found to be:

$$\texttt{learning\_rate} = 0.05, \quad \texttt{depth} = 6, \quad \texttt{l2\_leaf\_reg} = 3$$

For SVR, the hyperparameter tuning process was also performed using GSCV with 5-fold cross-validation. The hyperparameters tuned for SVR were `C` and `epsilon`, the effects of which are discussed in Section 3.3.5. The optimized hyperparameters were:

$$\texttt{C} = 1, \quad \texttt{epsilon} = 0.05$$

The neural network was implemented using `TensorFlow/Keras`. The architecture of the model, including the number of hidden layers, the number of neurons per layer, the learning rate, and the dropout rate, was optimized through a process of trial and error. The model's architecture consists of the following layers:

- An input layer with 169 features.
- Four hidden layers, 640 neurons for the first and 256 for the rest, all using the `ReLU` activation function.
- Dropout layers with a rate of 0.2, applied after each hidden layer to prevent overfitting.
- A final dense layer with a single neuron for the output prediction.

Additionally, two callback functions were used to improve the training process. The first, `EarlyStopping`, monitors the validation loss and stops training when it stops improving, restoring the model weights to the best performing point. The patience for early stopping was set to 16 epochs. The second, `ReduceLROnPlateau`, was configured to reduce the learning rate by a factor of 0.2 if the validation loss has not improved for 7 consecutive epochs. Reducing the learning rate helps the model converge more accurately to a local minimum of the loss function, potentially avoiding overshooting the minimum. The model was compiled using the Adam optimizer with an initial learning rate of 0.001 and was trained to minimize the mean squared error loss.

Although this architecture was the best found through manual adjustments, it is important to note that it is most likely not the optimal architecture and further fine-tuning and experimentation would be required to improve the model.

`LassoCV` and `RidgeCV` were used to optimize the lasso and ridge regression models. Both models automatically perform cross-validation to identify the optimal `alpha` parameter, which controls the strength of regularization. For lasso regression, the optimal `alpha` was found to be 1.279×10^{-5}, which suggests that a very small amount of regularization was applied. The optimal `alpha` found for the ridge regression model at the second level in stacking was approximately 2.28.

3.5 Evaluation metrics

In this research, the performance of the models are evaluated using three common metrics: Mean absolute error (MAE), root mean squared error (RMSE), and R^2 (coefficient of determination). These metrics were chosen for their ability to evaluate the accuracy of regression models.

3.5.1 Mean absolute error

Mean absolute error (MAE) is metric used in regression tasks, which measures the average size of the errors in the model's predictions, giving equal weight to each error. MAE is defined as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| ,$$

where:

- y_i is the true value of the target variable,
- $\hat{y}_i$ is the predicted value from the model,
- n is the total number of data points.

A lower MAE indicates better performance, as it means that the model's predictions are closer to the true values.

3.5.2 Root mean squared error

Root mean squared error (RMSE) is another metric that measures the average magnitude of the error between the predicted and actual values. RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

Again, a lower value of RMSE indicates a better model performance. Unlike MAE, which gives equal weights to each error regardless of their size, RMSE is sensitive to large errors, as the differences between predicted and actual values are squared. As discussed by Chai and Draxler (2014), the RMSE is appropriate when errors follow a normal distribution and when large errors are undesirable and need to be penalized more heavily.

3.5.3 Coefficient of determination (R^2)

The coefficient of determination, denoted as R^2, is another important metric for evaluating regression models. R^2 indicates how well the model's predictions fit the observed data. It is calculated as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

where $\bar{y}$ is the mean of the actual values. The value of R^2 usually ranges from 0 to 1:

- $R^2 = 1$ indicates that predictions perfectly fit the data.
- $R^2 = 0$ indicates that the model does not explain any of the variability in the target variable.

An R^2 value closer to 1 indicates that the model explains a high proportion of the variance in the data. It is an intuitive metric, as it provides a direct interpretation of the model's goodness of fit. However, R^2 can also technically be negative, which happens when the model's predictions are worse than simply predicting the mean value of the target variable for all instances. It should be noted that R^2 alone does not provide information about the magnitude of errors and it is affected by the amount of variance of the target variable, which is why it is often used in conjunction with other evaluation metrics (Kuhn and Johnson 2013, pp. 95–97).

4 Results

This chapter presents the predictive performance of the regression models used in this study evaluated on the test set. Table 4.1 summarizes the performance of each model. A notable observation across the models is the significant difference between the MAE and RMSE values. As discussed in the Evaluation metrics section, RMSE penalizes larger errors more heavily than MAE. Therefore, this discrepancy suggests the presence of outliers or small amount of large prediction errors in the dataset.

This assumption is supported by the predicted vs actual value plots shown in Figures 4.1 and 4.2. These plots correspond to the final stacking model and provide a visualization of its performance. In both figures, several outlier predictions can be observed, particularly properties that are predicted to be significantly more expensive than their actual sale prices. These deviations may be due to factors that lower the actual prices, which the dataset doesn't contain information about.

Additionally, Figure 4.2 shows a slight increase in variance among the predictions as property prices increase. This pattern suggests that the model shows increased uncertainty when predicting high-value properties, possibly due to the lack of such cases in the relatively small dataset.

Both of these factors result in getting moderately good predictions on average shown by the MAE values but not by RMSE. Therefore the models perform well for properties within the typical price range but struggle with more extreme cases.

Model	**R^2**	**MAE**	**RMSE**
XGBoost	0.916	19 170.11	35 462.50
CatBoost	0.921	18 526.08	33 015.73
Lasso	0.858	26 880.63	45 378.86
NeuralNet	0.906	20 768.03	36 541.57
SVR	0.907	19 682.54	35 763.12
Stacking	**0.923**	**18 144.69**	**32 870.71**
Voting	0.918	19 069.80	34 570.01

Table 4.1. Performance of the studied regression models on the test set

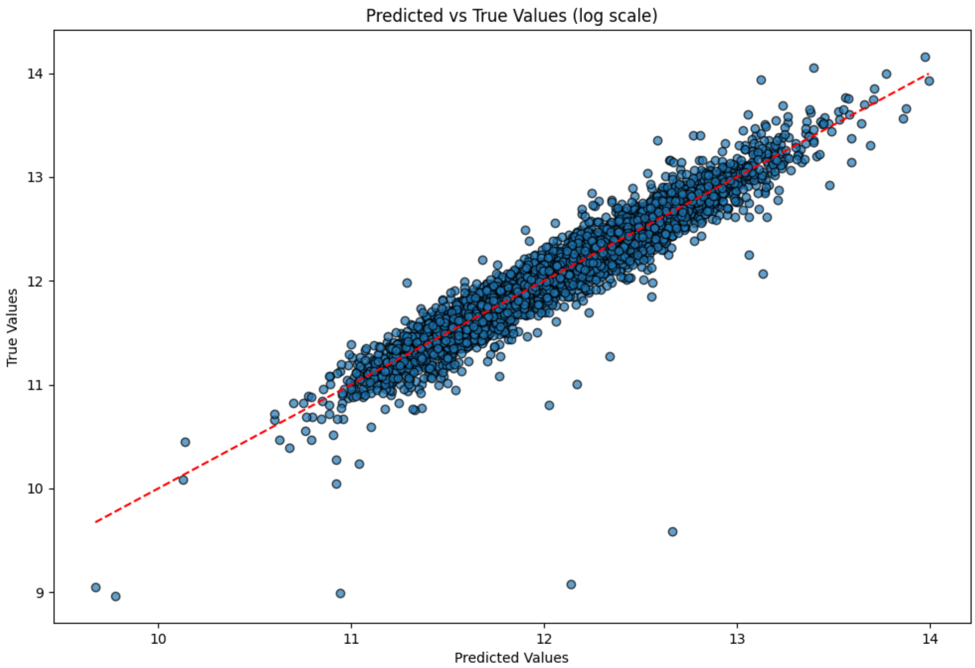

Figure 4.1. log-transformed predicted vs actual house prices for the stacking model

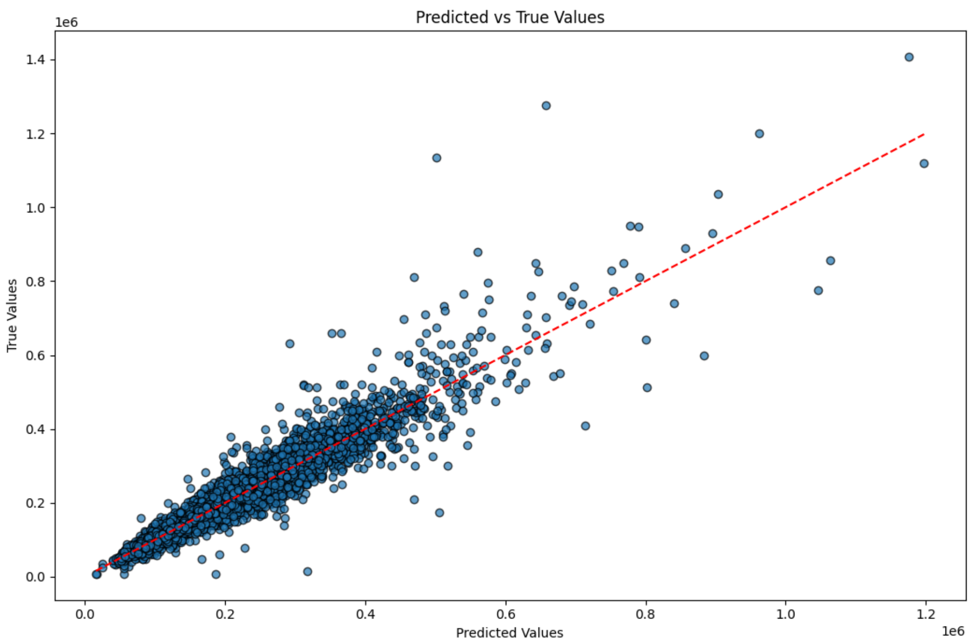

Figure 4.2. Predicted vs actual house prices for the stacking model in the original scale.

Among the individual base learners, CatBoost achieved the highest R^2 score of 0.921, followed closely by XGBoost and SVR with 0.916 and 0.907, respectively. These results demonstrate the value of handling categorical data with ordered target statistics, as well as the use of ordered boosting discussed in Section 3.3.8. These techniques are likely the reason behind CatBoost's improved performance compared to the other base models. The lasso regression model showed the weakest performance across all evaluation metrics. The relatively poor performance of the lasso model highlights the non-linear and complex nature of the data, making it less suitable for linear models.

The stacking ensemble achieved the best overall performance, with an R^2 score of 0.923, a mean absolute error of 18 144.69, and a root mean squared error of 32 870.71. For this dataset, model stacking achieved a higher level of predictive accuracy than the simple voting regression model. Voting regression also showed strong performance, outperforming most individual models. Its performance, however, is likely hindered by giving equal weight to weaker models such as lasso.

While model stacking outperforms voting in terms of accuracy, it requires significantly longer training times compared to voting regression. In this study, the time it took to fit the model stacking regressor was 16.5 minutes, while the voting regressor took 3.5 minutes.

To better understand the contribution of each base model to the performance of the stacking ensemble, the coefficients of the meta-model were examined. Table 4.2 shows the learned weights assigned by ridge regression to each of the five base regressors. Higher weights indicate greater relative importance in the final prediction.

Base model	**Meta-model coefficients**
Lasso	0.0242
FNN	0.0600
XGBoost	0.3230
CatBoost	0.4893
SVR	0.1172

Table 4.2. Weights of base models in the stacking ensemble estimated by the ridge regression meta-model.

The meta-model assigned the highest weight to CatBoost, with a value of 0.4893, followed by XGBoost, with a weight of 0.3230. These results indicate that the gradient boosting models were the most significant to the final predictions. SVR was assigned a moderate weight of 0.1172, while the feed-forward neural network and lasso received lower weights, at 0.0600 and 0.0242, respectively. The minimal contribution of the lasso model corresponds with its weaker performance in earlier analysis.

From Figures 4.3, 4.4, and 4.5, we can see the feature importance values of the three highest weighted models in the stacking ensemble. The feature importance for the SVR model was calculated using the `permutation_importance` function from the `sklearn.inspection` module. For CatBoost and XGBoost, the feature importance values were computed using the built-in feature importance methods. For XGBoost the `total_gain` parameter was used, while for CatBoost the default feature importance method was used.

Across all three models, the Living area and Year of construction features show relatively high importance. Additionally, location-related features appear to be significant. The Neighborhood feature holds high importance for all three models, while the Postal code feature is especially important for SVR and CatBoost.

On the other hand, the boolean features of Beach, Balcony, Elevator, and Sauna, as well as the Building material and Maintenance fee features, show the least importance across these models. The Property condition and Number of rooms features show medium levels of importance.

Thus it seems that location, size, age, and condition of the houses are the most important factors in predicting house prices.

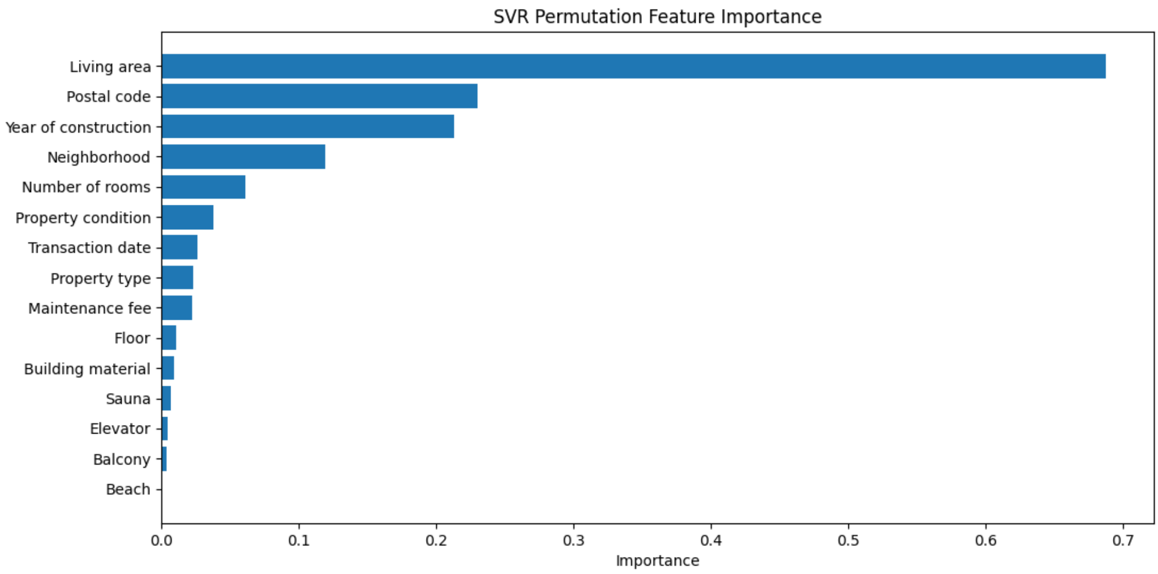

Figure 4.3. Permutation feature importance of SVR.

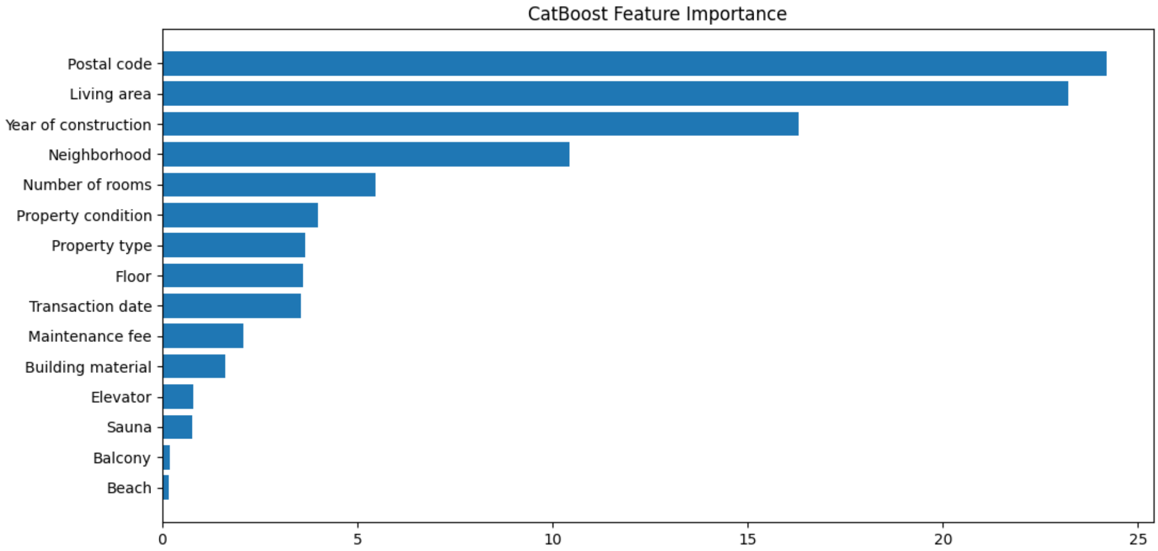

Figure 4.4. CatBoost feature importance values.

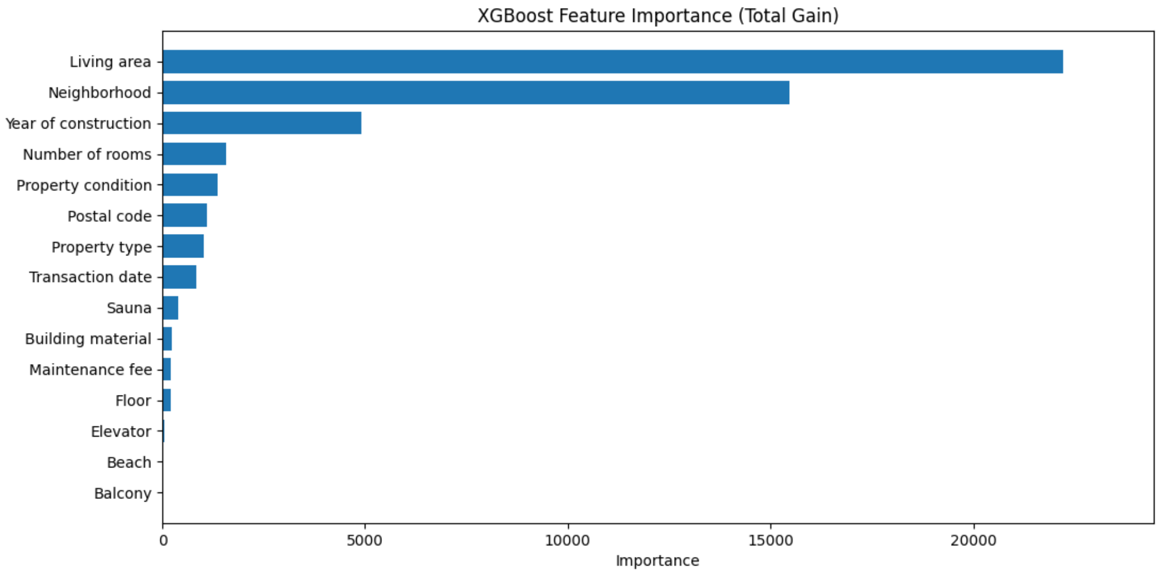

Figure 4.5. XGBoost feature importance values.

5 Discussion

The goal of this study was to investigate how well model stacking improves the performance of its base models and how it compares to the simpler ensemble method of voting regression, particularly for predicting house prices using a relatively small local dataset. Chapter 3 detailed the methodology, starting with data selection, preprocessing, and feature scaling. Exploratory data analysis (EDA) was conducted to better understand the data, detect outliers, and identify any issues requiring cleaning. The theory behind the used models in this study was also examined, along with the implementation details and hyperparameter tuning process. Evaluation metrics including MAE, RMSE, and R^2 were used to assess model performance.

Our findings align with existing literature, which shows that stacking ensembles can improve prediction accuracy compared to individual base models. The meta-model effectively combines the predictions of these base models, leading to improved performance relative to voting regression. The stacking ensemble demonstrated its ability to utilize the strengths of its most effective base models while minimizing the influence of weaker models. The final stacking model provided reliable predictions for average-priced homes, although its performance was less robust for properties with uncommon price levels.

While stacking produces more accurate predictions, a significant drawback is that it requires more training time than voting regression. Each base model must be trained multiple times using k-fold cross-validation to generate out-of-fold predictions, which serve as features for the meta-model in order to prevent overfitting. Additionally, training and tuning the meta-model itself requires extra time. This can be computationally expensive, especially with large datasets and complex models. Furthermore, the final model in a stacking ensemble can be difficult to interpret, especially when using complex base models such as neural networks.

Thus, stacking is particularly suitable for tasks where predictive accuracy is the primary goal, and longer training times can be tolerated. Additionally, stacking is appropriate for tasks where high interpretability is not required.

5.1 Limitations

Despite the insights provided by this study, several limitations must be acknowledged. One significant limitation is the absence of macroeconomic market indicators in the dataset. Factors such as interest rates, inflation, GDP, and unemployment rates can significantly influence real estate prices. The exclusion of these variables may have constrained the predictive power of the models.

Another limitation comes from the limited occurrence of high-priced homes within the relatively small dataset. The lack of data could have affected the model's ability to accurately predict prices for high-value properties. A larger dataset, with more high-priced homes, would likely improve the model's overall prediction accuracy.

The dataset also includes a number of outliers, which can distort model predictions. Although some techniques were employed to address this issue, the addition of more features could help minimize prediction errors caused by outliers and lead to more robust models. Features such as lot size, renovation history, parking availability, and energy source, for example, could help predict unexpectedly priced houses more accurately.

Finally, it is important to note that this study relied on a single dataset, and as such, the results may not be generalizable to other datasets.

5.2 Future research

As noted previously in Section 3.3.1, one of the challenges with model stacking is the lack of a standardized method for selecting the stacking architecture. While some studies favor multi-layer stacking or the inclusion of additional meta-features, the true impact of these modifications remains unclear. Future research should aim to evaluate whether these approaches lead to substantial improvements in model performance across a variety of datasets.

A particularly interesting avenue for further study involves investigating whether the inclusion of the original training data in the meta-model enhances predictions. This approach could provide the meta-model with more information, potentially improving performance. However, it could also overcomplicate the task at the meta-level, leading to a decline in accuracy. Perhaps only a subset of the original training data should be passed to the meta-level. Additionally, further research is needed to study how meta features affect predictions depending on the choice of the meta-model.

Furthermore, as mentioned earlier, the inclusion of macroeconomic indicators such as interest rates, inflation, and GDP may significantly impact model performance in predicting real estate prices. Future studies could include these variables into the dataset and evaluate how they affect prediction accuracy.

References

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Box, G. E. P., & Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)*, *26*(2), 211–243.

Brazdil, P., van Rijn, J. N., Soares, C., & Vanschoren, J. (2022). *Metalearning: Applications to automated machine learning and data mining* (2nd ed.). Springer Nature.

Breiman, L. (1996). Stacked regressions. *Machine Learning*, *24*(1), 49–64.

Chai, T., & Draxler, R. R. (2014). Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature. *Geoscientific Model Development*, *7*(3), 1247–1250.

Chen, S., & Zheng, W. (2025). RRMSE-enhanced weighted voting regressor for improved ensemble regression. *PLOS ONE*, *20*(3).

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). ACM.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273–297.

Drucker, H., Burges, C. J. C., Kaufman, L., Smola, A., & Vapnik, V. (1996). Support vector regression machines. *Advances in Neural Information Processing Systems*, *9*, 155–161.

Džeroski, S., & Ženko, B. (2004). Is combining classifiers with stacking better than selecting the best one? *Machine Learning*, *54*(3), 255–273.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, *29*(5), 1189–1232.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, *38*(4), 367–378.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed.). Springer.

Hastie, T., & Zhu, J. (2006). Comment. *Statistical Science*, *21*(3), 352–357.

Joanes, D. N., & Gill, C. A. (1998). Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society. Series D (The Statistician)*, *47*(1), 183–189.

Kuhn, M., & Johnson, K. (2013). *Applied predictive modeling*. Springer.

Müller, A. C., & Guido, S. (2016). *Introduction to machine learning with Python: A guide for data scientists*. O'Reilly Media.

Nnadozie, L., Matthias, D., & Bennett, E. O. (2022). A model for real estate price prediction using multi-level stacking ensemble technique. *European Journal of Computer Science and Information Technology*, *10*(3), 33–45.

Pai, P.-F., & Wang, W.-C. (2020). Using machine learning models and actual transaction data for predicting real estate prices. *Applied Sciences*, *10*(17), 5832.

Park, U., Kang, Y., Lee, H., & Yun, S. (2022). A stacking heterogeneous ensemble learning method for the prediction of building construction project costs. *Applied Sciences*, *12*(19), 9729.

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: Unbiased boosting with categorical features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (pp. 6639–6649).

Rampini, L., & Re Cecconi, F. (2022). Artificial intelligence algorithms to predict Italian real estate market prices. *Journal of Property Investment & Finance*, *40*(6), 588–611.

Raschka, S. (2018). [Figure]. MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack. *The Journal of Open Source Software*, *3*(24). Retrieved from https://rasbt.github.io/mlxtend/user_guide/regressor/StackingCVRegressor/

Sill, J., Takacs, G., Mackey, L., & Lin, D. (2009). Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*.

Song, Y., Song, Y., Wang, C., Wu, L., Wu, W., Li, Y., Li, S., & Chen, A. (2024). Landslide susceptibility assessment through multi-model stacking and meta-learning in Poyang County, China. *Geomatics, Natural Hazards and Risk*, *15*(1).

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, *58*(1), 267–288.

Vapnik, V. N. (2000). *The nature of statistical learning theory* (2nd ed.). Springer.

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, *5*(2), 241–259.

Witten, I. H., Frank, E., & Hall, M. A. (2011). *Data mining: Practical machine learning tools and techniques* (3rd ed.). Morgan Kaufmann.

Yang, Z., Zhu, X., Zhang, Y., Nie, P., & Liu, X. (2023). A housing price prediction method based on stacking ensemble learning optimization method. In *2023 IEEE 10th International Conference on Cyber Security and Cloud Computing (CSCloud) / 2023 IEEE 9th International Conference on Edge Computing and Scalable Cloud (EdgeCom)* (pp. 96–101).

Zhou, Z.-H. (2012). *Ensemble methods: Foundations and algorithms* (1st ed.). Taylor & Francis.

Zaki, J., Nayyar, A., Dalal, S., & Ali, Z. H. (2022). House price prediction using hedonic pricing model and machine learning techniques. *Concurrency and Computation: Practice and Experience*, *34*(27).