DATA VISUALIZATION FOR DATA SCIENCE IN REAL ESTATE MARKET

BY

CHUA MIN XUAN

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION SYSTEMS (HONOURS) (DIGITAL ECONOMY TECHNOLOGY)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2026

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Systems (Honours) (Digital Economy Technology) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Ts Dr Zanariah binti Zainudin and my moderator, Ms. Jin, Tong, both who has given me this bright opportunity to engage in a machine learning project. Thank you for the guidance, patience, and understandings throughout the project. Besides, I also would like to thank my family and course mates for their love, support, and continuous encouragement throughout the course.

ABSTRACT

Housing is one of the basic human needs, and real estate has always had a high demand. It is crucial for stakeholders to correctly estimate the value of property, so that they are able to perform informed decisions. This research focuses on performing housing price prediction using machine learning, and constructing a dashboard integrated with automated ML pipeline. This research is based on a dataset consists of Singapore's HDB public house price. Selected models for this research include Linear Regression, Decision Tree, Support Vector Machine, Gradient Boost Machine, Random Forest, and Extreme Gradient Boosting. The necessary data preprocessing and feature engineering are performed, with customized column transformers applied to fulfil the need to handle high cardinality columns. After the baseline model comparison, the XGBoost, GBM, and Random Forest turned out to be the best 3 performers. These 3 models with baseline model are selected as candidates for hyperparameter tuning to select the final model, which will be the scope for project 2. Meanwhile, SVM turns out to be the worst performer among selected models. In the experiment, the DT shows significant overfitting. After hyperparameter tuning using Randomized Search and completed comparison, XGBoost with its optimal parameters is selected as the final model. A dashboard is later constructed, supported functionalities including data visualization, user upload CSV and automated model training, and single or batch housing price prediction. This project contributed to further the research from theoretical results to an actual developed system for housing price prediction with efficiency for production considered, helping end users to have a solution for making informed decisions.

Area of Study: Machine Learning, Data Visualization

Keywords: Housing Price Prediction, Data Preprocessing, Feature Engineering, Hyperparameter Tuning, Exploratory Data Analysis Model Deployment, Dashboard, Automated Machine Learning Pipeline

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ML	Machine Learning
OLS	Ordinary Least Square
GWR	Geographically Weighted Regression
GTWR	Geographically and Temporally Weighted Regression
MAE	Mean Absolute Error
R^2	R-Squared
RMSE	Root Mean Square Error
SVM	Support Vector Machine
GBM	Gradient Boost Machine
RF	Random Forest
DT	Decision Tree
XGBoost	Extreme Gradient Boosting
LR	Linear Regression
HDB	House Development Board
WSL2	Windows Subsystem for Linux 2
RAM	Random Access Memory
OOM	Out of RAM
CSV	Comma Separated Values

CHAPTER 1
Introduction

In the first chapter, the project background, motivation and contributions to the field, with the outline of the project are discussed and outlined. This is a research-based project to review and develop the house price prediction model, and in the end, build a web application which integrated automated machine learning pipeline that allows users to perform model training tasks with one-click, perform housing price regression with visualized dashboard, and allow user to estimate the house price based on attributes of one particular house.

1.1 Problem Statement and Motivation

One of the basic human needs is need for housing, and no matter which country is in the context, real estate has always had a high demand. For potential purchasers, property price is one of the major considerations for them to do purchase decision for a property [1]. Accurate property valuation will naturally become crucial for buyers to identify affordable and suitable homes, for sellers to set competitive prices, and for investors to evaluate potential returns. Similarly, policymakers and urban planners rely on housing price data to guide decisions that affect housing affordability and urban development.

In the field of housing price prediction, previous research largely used hedonic pricing modelling (HPM) technique to compare it with other models [2]. Ordinary least square (OLS, which technically is linear regression), is common choice as the model for HPM technique. However, traditional hedonic price models might not be able to capture the real influencers that actually affecting the price. Some other methods also have been proposed to introduce variable of space and time to the model, such as GWR and GTWR, but they still as not powerful as machine learning models to capture the pattern behind the complexity [3].

On the other hand, visualization of data is essential for both scientific and general purpose as graphs and charts are definitely more intuitive than just figures. Effective data visualizations are crucial in data mining, which presents trends and changes in

intuitive ways, and allows others both to understand the data and to gain insights from them [4]. However, previous studies which have been done related to the utilization of ML method to predict housing price only stopped at the modelling stage, and never further consider practicality in production such as efficiency, deployment and data visualization for the output in the form that could be easily understood and interpreted by non-technical users. Therefore, there is a need to further the study to the stage of model deployment and data visualization.

The main motivation for this research comes from the needs to enhance predictive performance and to improve the practicality of models for housing price prediction. Therefore, this study aims to evaluate multiple machine learning algorithms with hyperparameter tuning to identify the best-performing model for the selected dataset. This will provide a clear view of the impact of using different techniques to handle housing price prediction. Besides, this research will also bridge the gap of previous studies by developing a user-friendly application with graphic interface, integrated with machine learning model that able to predict the housing price, to visualize data for better understanding and enable general users to actually benefit from the robustness of machine learning.

1.2 Research Objectives

At the completion of the research, the listed objectives shall be fulfilled:

1. To perform exploratory data analysis (EDA) and data preprocessing on selected dataset,
2. To perform baseline model comparison based on selected processed dataset and chosen list of models, and perform evaluation using selected metrics,
3. To optimize models' performance by performing hyperparameter tuning, and select the best-performed model as the final model,
4. To construct an automated machine learning application integrated with dashboard capable of visualize data and price prediction dynamically based on user input.

Before modelling, the necessary exploratory data analysis (EDA) and data preprocessing, such as data cleaning, logarithmic transformation, and feature

engineering will apply on the dataset. Then, a list of models will be trained based on the selected dataset and evaluate the performance. Hyperparameter tuning will be performed to improve the performance of the model. Then, the models will go through the evaluation again, and the best performer will be picked based on the result. The final model shall be built based on the evaluation results using the selected dataset.

Finally, a dashboard using the selected final model integrated with the ML pipeline will be constructed. The app will be able to automate the data preprocessing and model training process, which user only needs to provide the relevant dataset for training. The dashboard will be able to visualize model's forecast results, aside from basic data visualization, and also allow user to do price estimation for houses.

1.3 Project Scope and Direction

The research majorly covers four scopes, which are modelling for housing price prediction, model evaluation, and build of automated ML pipeline with dashboard. The first scope tends to choose an existing dataset and build a housing price prediction model. As real estate markets may differ by countries and regions, and it is not practical to unify all of them. Therefore, a dataset based on Singapore's HDB public house will be used in this research to train the models. The dataset describes the characteristics of HDB public houses including their dwelling, building, spatial, and temporal characteristics.

The necessary data preprocessing will be performed. The models that will be included in the research will be machine learning models, which are LR, DT, SVM, RF, GBM, and XGBoost. Note that deep learning techniques will not be covered in this research, as they do not show significant differences nor improvements compared to traditional machine learning methods in terms of handling tabular data based on previous studies. At the end of the modelling, the performance of models will be evaluated based on R^2, RMSE, and MAE.

The next scope is to construct an ML application integrated with the pipeline and dashboard to ease and automate the workflow of machine learning. On the other hand, the application will have a dashboard not only can perform basic data visualization, but also capable of running the automated ML pipeline on demand which most if not all previous studies are missing. Users can understand the evolution of the housing price in the selected area, and the prediction for future price through the interactive

visualization. The users could also input the features of their housing, and the model will output the estimated price.

1.4 Contributions

Primarily, the contribution of this project is to convert information into actionable insights that could be meaningful to others by creating an interactive dashboard using data visualization software. This research will dive into a selected dataset and visualize it to produce a comprehensive dashboard that can be utilized to acquire meaningful insights and find out unseen relationships. Current related research mostly did not include the visualization for the output in the form that could be easily understood by non-experts and can be interact with and understand effectively. At the end of this research, an interactive dashboard will be carried out to visualize the output of the model.

Furthermore, this research will also perform the evaluation and select the best performed ML model and will be used to construct the previously mentioned dashboard upon the final stage of the project. This selected final ML model would further support the primary goal of helping stakeholders in the real estate market to estimate the value for properties accurately and conveniently and make informed decisions in terms of price, urban planning, etc.

1.5 Report Organization

In Chapter 2, previous related research, papers and proposed methodologies will be reviewed and summarized. Chapter 3, methodology, models, and metrics to be applied in this research will be further discussed. In Chapter 4, hardware environment and software stacks for the project is listed. Experimental setup, findings, and results obtained will be displayed in Chapter 5. The final evaluation and details of dashboard construction are discussed in detail in Chapter 6. Finally, conclusion for this project and recommendation for future works in the field are discussed in the last chapter, which is Chapter 7.

CHAPTER 2

Literature Review

Multiple studies have been conducted for the purpose to estimate the house price using machine learning in the field of real estate market. In this chapter we will go through and understand these related previous studies.

2.1 Previous studies on housing price prediction

Different studies have been done for the purpose of finding the best method to estimate the housing price. In this chapter, we will review the previous studies related to housing price prediction using hedonic regression methods, machine learning methods, and ensemble machine learning methods.

2.1.1 Housing Price Prediction using Hedonic Pricing Model

Traditionally, hedonic pricing models is commonly adopted as an effective to quantify the value of housing. Previous studies suggested that the commercial value of a property has a direct relationship with its associated attributes such as structural attributes and locational amenities. Hedonic pricing models, including OLS, GWR, and GTWR, treat a property as a good associated with multiple characteristics to understand the formation of its value and estimate its price [5-7]. Some previous studies related to the method have been carried out.

The authors of [5] use OLS model to figure out the effects of structural attributes and accessibility of neighbouring destinations on housing prices in core urban area pf Dalian, China. The dataset used in the research consists of 51395 instances and 15 features including one target variable. In results, the model shows that the adjusted R-squared (R^2) is 0.4276.
In [6], the authors use OLS model to find out which fundamental determinants provides more importance for prediction for housing market in the city of Santiago, Chile. The dataset used in this study includes 456000 instances and 11 variables. The OLS model shows 0.6074 of adjusted R^2 for the dataset.

[7] conducted the research of housing price prediction based on the case of Singapore's House Development Board (HDB) public house. In this study, authors chose OLS model and two GWR model, with one based on Euclidean distance and another based on travel time supported

by a big dataset of 30 million smartcard transactions. The study is mainly based on a dataset includes 21856 transactions which include transaction details, address details, and physical characteristics of properties.

In results, the OLS model shows 0.7 of R^2 and 0.695 of adjusted R^2 for the dataset, the Euclidean distance-based GWR model can reach 0.8873 of adjusted R^2, and 0.9589 of adjusted R^2 for travel time-based GWR model.

2.1.1.1 Limitations

The common metrics for model performance evaluation usually involves MAE, MSE, etc. [8], but [5-7] does not comply with this convention and the only relevant metric is adjusted R^2. Therefore, the evaluation for these hedonic pricing models is not all round and could not correctly evaluate their performance.

Furthermore, in [7], the travel time-based GWR model is supported by an additional dataset containing 30 million instances of smart card transactions, which the other two models did not have similar advantage. The advantages of data quality and richness for the mentioned model caused the experiment is partially skewed and result in the high performance of the mentioned model.

2.1.2 Housing Price Prediction using Machin Learning

In [3], 4 supervised machine learning methods are chosen to be assess, which are LR, DT, RF, and GBM. The researchers created a spatio-temporal lag variable to the dataset. The dataset used in this research contained 352024 data points with 46 features. The authors performed hyper-parameter tuning using Grid Search Cross-Validation for these models and evaluated the performance of different models using R^2, MAE, and RMSE. In summary, GBT and RF have better performance in housing price prediction.

In [9], the authors use the UCI Machine learning repository Boston housing dataset to develop the model for housing price prediction based on RF. The dataset consists of 506 instances and 14 features that represented the influencing factors of house price in Boston in 1978. In results, the authors of [9] compared the prediction difference and the model successfully achieved a prediction difference of ±5. The authors also used RMSE, MAE, and R^2 as the evaluation

metrics to evaluate the models' performance in the research. The RF model, by using the mentioned dataset, has achieved 0.9001 of R^2, 1.9001 of MAE, and 2.5890 of RMSE.

In the research of [10], the authors assessed the performance of SVM, RF, and GBM in the field of housing price prediction. The experiment is based on a dataset which consists of 39544 transactions from 1996 through 2014. The performance evaluation metrics including R^2, MSE, RMSE, and MAPE. In results, both RF and GBM perform better than SVM in terms of accuracy and all other aspects. However, the authors also pointed out that SVM is still an applicable model if time is constrained.

Authors of [11] chose Lasso and Ridge Regression, and Elastic Net Regression as models to be examined in the research. The dataset used to develop and examine the machine learning model is obtained from Kaggle. The dataset consists of 1460 instances and 81 features. Lasso Regression is the best performer among three selected models. The results show that the 0.9122 for Train R^2, 0.0821 for MAE, 0.1148 for MSE; the Test R-squared is 0.9113, 0.0786 for MAE, and 0.1073 for MSE.

In [12], the research is based on real estate market in Spain, particularly in Alicante. The dataset used in the study collected through web-scrapping and contained around 49,875 different properties for sale listed on a real estate portal in the time frame from May 2019 to December 2021. The authors used machine learning algorithms such as LR, RF, Extra Trees, GBM, XGBoost, and LightGBM in [12]. Authors used MAE, MSE, RMSE, and R^2 to evaluate the performance of different algorithms. Hyperparameter optimization was done with to improve the R^2 score of different models and minimizing prediction errors. Gradient Boosting methods (GBR, XGBM, LGBM) outperformed others with higher predictive accuracy and better handling of non-linear relationships. XGBoost and LGBoost are the fastest algorithms among the tested.

For [13], the authors performed web-scrapping to build the dataset to be used in this study. The dataset is based on Islamabad, Pakistan, consists of 44647 instances and 23 features including the target variable. The machine learning algorithms authors examined in the study include Bayesian Regression, LR, SVM, GBM, etc. Authors used MAPE, RMSE, and MAE. The results show that SVM is best performed among different algorithms.

The authors of [14] performed case study on Kuala Lumpur housing data, and assessed multiple regression analysis, ridge regression, LightGBM, and XGBoost. All 4 models have hyperparameter tuning using grid search applied. MAE, RMSE, and adjusted R^2 are used in the study as the evaluation metrics. The XGBoost model was the best performing model with 0.197 for RMSE and has highest adjusted R^2 score.

2.1.2.1 Limitations

Traditional methods and even some advanced machine learning approaches might face challenges in housing price prediction due to the use of outdated datasets. Some studies [9,11] rely on data that no longer reflects the current urban structure and socio-economic trends. Urban planning is constantly evolving due to factors such as population growth, infrastructure development, economic shifts, and changes in societal preferences. Outdated datasets fail to capture these transformations, leading to models that may perform well historically but are ineffective for predicting current housing prices accurately.

Besides, in [10], although hyperparameter tuning such as Stochastic Gradient Descent (SGD) and 5-Fold Cross-Validation applied on SVM, the researchers may not have conducted an exhaustive hyperparameter tuning such as grid search or random search for SVM like they did for the other two models, where more detailed grid search processes were applied. This may possibly lead to a biased experiment result. In contrast, experiment in [13] included SVM and GBM for the assessment. The result, however, shows that SVM performs better. Therefore, the experiment methodologies will be applied to assessed models equally in our research to achieve a fair and insightful result.

2.1.2.2 Strengths

In [3], the authors introduced a spatiotemporal lag variable in the research. The variable considered the spatially and temporally non-stationarity, which most previous studies neglected [3]. According to the results, this variable can significantly enhance the performance the model. Therefore, this could be applied to our research too to provide more accurate predictions.

2.2 Comparison Analysis Hedonic Pricing Model, Machine Learning, and Deep Learning

There is another significant research which utilized traditional hedonic pricing model, machine learning, and artificial neural network (ANN) to predict the housing price [15]. In this study,

the ANN is selected to estimate the housing price. The authors seleted ReLU as the activation function as is easy to optimize due to its similarity to linear function. The other models that have been assessed in the experiment include K Nearest Neighbour (KNN) model, RF model, and OLS model. The dataset used in the experiment consists of 1018 instances and 25 features including one target variable. RMSE, MAPE, and R^2 are the evaluation metrics. After hyperparameter tuning, the results show that RF outperformed other models in terms of every aspect in this research.

2.3 Summary

The literature review of previous studies on different methods for housing price prediction can give us some insight into the field. The review emphasized the models' performance in multiple investigations, focusing on criteria such as MAE, MSE, and R^2 score.

Based on previous studies, machine learning is generally performed better than traditional hedonic pricing models. Among machine learning models, the most chosen models include RF, GBM, LR, DT, and SVM. These models could be used in our experiment. Additionally, artificial neural network does not show advantages in this particular field.

Limitations of previous studies include outdated datasets [9,11] which might fail to reflect contemporary urban dynamics and trends and unequal hyperparameter tuning among models [10] led to potentially biased outcomes. Therefore, these issues need to be avoided in this research.

While significant progress has been made in applying machine learning to housing price prediction, most previous studies [3,9-15] have focused primarily on developing and evaluating models with only focused on the predictive power, without considering the efficiency in production environment. Further steps in creating dashboard and perform data visualization are also lacked. This limits the practical implementation of these methods, as stakeholders in the real estate market often lack the knowledge in technical area to interpret raw outputs and demand for efficiency. Therefore, the workflow in this research will be further to include these aspects.

CHAPTER 3

System Model

3.1 Research Methodology

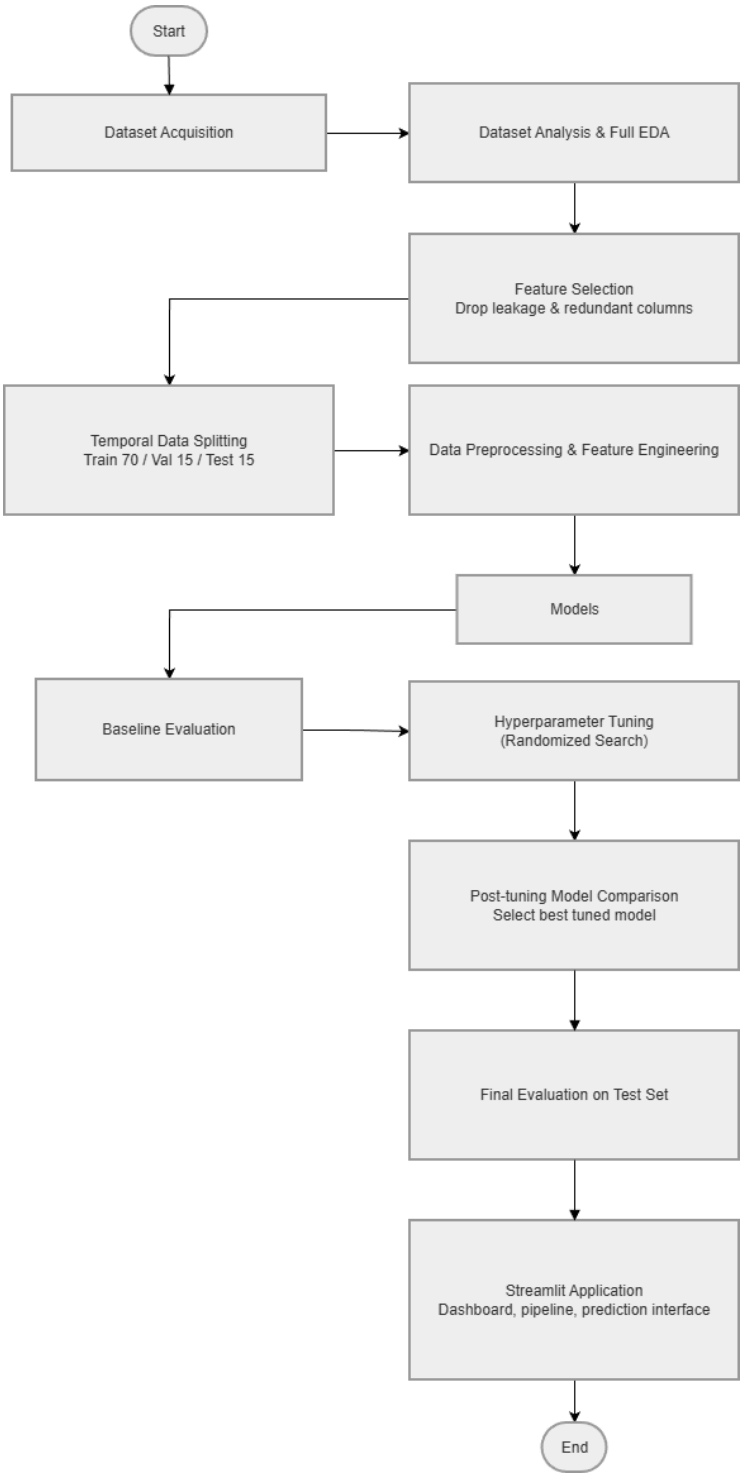


Figure 3.1 Methodology

The first step in any research is to acquire the data to be used for the experiment. This involves understanding statements of problems, the objectives and the scope of the research to find the suitable dataset. In our research, we will focus on the data visualization for data science in the field of real estate market, particularly with the case of Singapore's HDB flat resales as sample and aim to build a model for price prediction. The HDB resale transactions dataset that will be used in our research is from Kaggle.

Before any data manipulation steps, exploratory data analysis (EDA) will first be performed to find out the characteristics of dataset and help to determine the steps to be taken in further preprocessing steps. The EDA is focused on two aspects: the dataset itself, and the data distribution. Next, based on insights gathered in EDA stage, data preprocessing. This step includes data splitting, normalization, feature encoding, etc. Next, the selected dataset is split with 70:15:15, resulted in 3 sets which are train, validation, and test set, without shuffling the order of record and strictly followed the order by month to make sure there is no information leakage to the greatest extent.

After the datasets are prepared, selected models will be used to perform baseline model comparison using train set and validation set. LR, DT, SVM, RF, GBM, and XGBoost are selected in this research. For the comparison, R^2, adjusted R^2, RMSE, and MAE are used as performance metrics, and training time, R2/s, efficiency score, and time penalty are used as efficiency metrics. Overfitting analysis is also considered since the final model will need to be deployed to production environment.

Hyperparameter tuning is performed using Randomized Search, on the three best performing models and the baseline model using train set and validation set, with RMSE being used as the determinant. Full validation with all previously mentioned metrics is done later using validation set. Finally, the best performed model went through the final testing with test set.

In this project, the deployment stage is included. Once the model is finalized, a dashboard integrated with the full ML pipeline to streamline the whole workflow is constructed around the results. The dashboard will serve the purpose of displaying summaries of data, charts and graphs, and also allows users to perform housing price prediction dynamically. The functionalities is verified with unit tests.

3.2 Models

The selected models covered from the simplest to complex ensemble models to examine their capabilities to handle complex tabular data, without adding the complexity of neural networks.

3.2.1 Linear Regression

LR operates by identifying a straightforward, linear connection between various input features (like a house's size or age) and the outcome being predicted (like its price). It can process numerous factors at once. Because it clearly and simply shows how different variables influence the result, it's a foundational and commonly used prediction model [3]. It is the baseline model selected for the project due to its simplicity.

3.2.2 SVM

SVM technique works by finding a method to optimally separates the data points by looking for a specific boundary. For more complicated datasets where a simple straight line isn't enough, SVM employs a clever method known as the "kernel trick." This allows it to find a non-linear dividing line, making it a highly adaptable tool for various prediction tasks [10].

3.2.3 Decision Tree

DT functions much like a flowchart. It sorts data by making a sequence of if-then decisions. At each "internal node," the model asks a question about a particular feature, splitting the data down different "branches" based on the answer. This process continues until it reached to a final prediction, or “leaf node” [3].

3.2.4 Random Forest

RF improves on top of the DT concept by creating a large "forest" of individual trees. Each tree is built using the data that is randomly selected. For a final prediction, the model aggregates the outputs from all trees—typically by averaging their results. This "wisdom of the crowd" approach makes the model highly effective, stable, and capable of managing complex datasets with numerous features [9].

3.2.5 Gradient Boost Machine

GBM is a team-based technique that builds models in a step-by-step fashion. It begins with an initial model and then sequentially adds new ones, with each new addition specifically designed

to correct the errors made by the one before it. This iterative process of improving upon its mistakes makes GBM highly proficient at identifying and modelling complex patterns within the data [10].

3.2.6 Extreme Gradient Boosting

XGBoost is an enhanced and highly optimized version of the GBM [14]. It is widely recognized for its superior performance and computational speed, making it a popular choice in practical applications like online advertising and competitive data science. The combination can ensure that the selected final model is appropriate and efficient enough in production usage.

3.3 Evaluation Metrics

The performance of models will be evaluated from two aspects. One is the predictive power based on R^2, Adjusted R^2, RMSE, and MAE. Besides, another evaluation aspect is efficiency based on models' training time, R2/s, efficiency score, and time penalty.

3.3.1 R^2

The R^2 metric, or the determination coefficient, measures the scale of the variability in the outcome that can be interpreted by the model. Scored on a scale from 0 to 1, a higher value indicates that the model accounts for a larger percentage of the data's variance. R^2 is excellent for gauging a model's overall goodness-of-fit.

3.3.2 Adjusted R^2

Adjusted R2 is a modified version of R2. It also evaluates the goodness-of-fit in regression models, adjusted for the number of predictors, unlike R2, however, it penalizes the addition of irrelevant variables and only increases if a feature improves the model. It is crucial for determining whether a model is overfitting.

3.3.3 RMSE

RMSE provides a measure of the average size of prediction errors. It works by calculating the square root of the results, which is calculated by the average of squared differences between predicted and actual recorded values. Through squaring the errors, this metric places a much heavier penalty on larger mistakes, making it particularly sensitive to outliers. RMSE is useful when significant prediction errors are very costly.

3.3.4 MAE

MAE calculates the average scale of the errors in predictions. It is determined by calculating the average of the absolute value of the differences between the predicted and actual recorded values. This provides a direct and easily interpretable measure of the typical prediction error. MAE is preferred when you want a balanced assessment where large and small errors are weighed equally.

3.3.5 Training Time

The total consumed time to train a model, measured in second in the project. In a rapidly evolving market, it is unreasonable to ask the stakeholders to wait for a long time just to train an ML model, especially if the dataset is large. Training time is one of the most direct metrics that shows the efficiency of a model.

3.3.6 R^2/s

It treats time as a resource and measures the rate of variance explained per unit of time. It is used to identify models that provide high explanatory power with minimal time consumption. High R^2/s indicate models with high efficiency.

3.3.7 Efficiency

It is designed to evaluate the trade-off between predictive error and computational latency. The calculation applied a logarithmic transformation to the training time, which ensures that minor improvements in error are not affected by massive discrepancies in training speed. A lower score indicates a more "optimal" model that achieves a superior error-to-latency ratio.

Formula:

$$\text{RMSE} * \log_{10}(1 + \text{Training time})$$

3.3.8 Time Penalty

This is a relative comparison measures the increases of training time of a model compared to a selected baseline or itself in default parameters.

3.4 Timeline

The timeline for this project 1 and project 2 is drafted in this section.

Figure 3.2 shows the timeline in Gantt chart for all the work that needs to be done with the defined timeframe in this project 1. Works including report constructing, initial experiment preparation, EDA, data preprocessing, and baseline model comparison.

In Figure 3.3, a Gantt chart for project 2 is roughly drafted and outlined. The chart includes works that need to be done with the defined timeframe during the trimester. Works include revisiting and refinement of previous work, hyperparameter tuning, final testing, full dashboard development, and report writing.

All of the objects drafted on the charts are subject to adjust according to actual progress and requirements.

3.4.1 Project 1

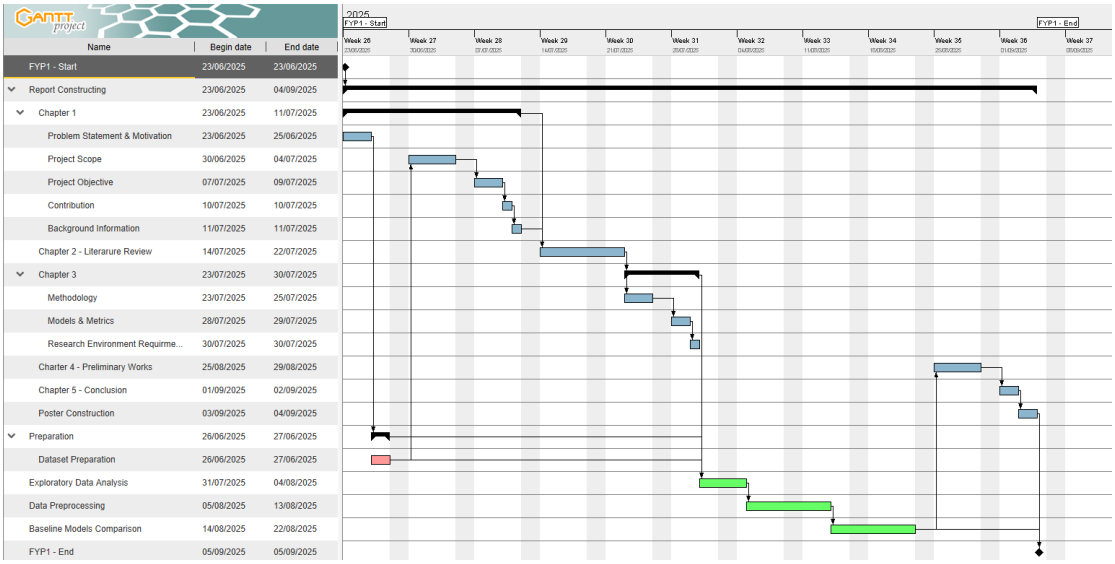


Figure 3.2 Timeline for Project 1

3.4.2 Project 2

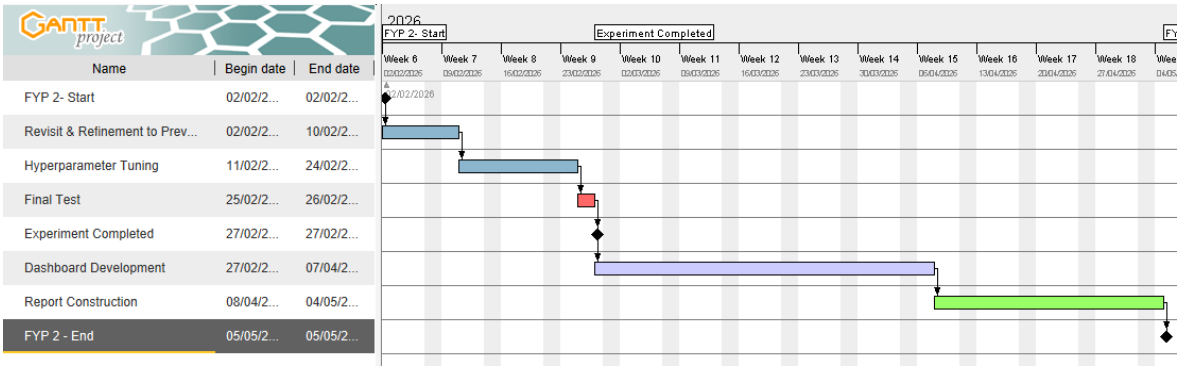


Figure 3.3 Timeline for Project 2

CHAPTER 4

System Design

4.1 Environment

4.1.1 Hardware

The experiment and development will majorly be done locally. Therefore, the hardware requirements involved in this project are computer to perform the necessary programming, experiment code execution, and dashboard development.

Description	Specifications
Processor	Intel Core i7-13620H (6P+4E)
Operating System	Ubuntu 24.04 LTS - WSL2
Graphic	Nvidia GeForce RTX 4060 Mobile 8GB GDDR6
Memory	32GB DDR5
Storage	2TB NVMe SSD

Table 4.1 Hardware Environment

4.1.2 Software

4.1.2.1 Python

Python is selected as the primary language for programming tasks in the project. It's a top choice in the data science community because its syntax is easy to read and it's supported by a massive collection of free, specialized libraries. This makes it perfect for quickly developing ideas, handling data efficiently, and using powerful machine learning tools.

4.1.2.2 Jupyter Lab

Jupyter Lab served as the interactive workspace for this research. Think of it as a digital lab notebook where you can write code, run it immediately to see the output, create charts, and write notes all in the same place. This setup is ideal for exploring data and experimenting with different models in a flexible, step-by-step manner.

4.1.2.3 NumPy

NumPy is a fundamental Python library used for working with numbers. Its main job is to provide a powerful tool for handling large arrays and matrices of numerical data. Because it's highly optimized for speed, it forms the backbone of many other scientific and data analysis libraries.

4.1.2.4 Pandas

For organizing and cleaning data, the Pandas library was essential. It provides easy-to-use data structures, most notably the DataFrame, which is like a powerful spreadsheet inside Python. Pandas simplified tasks like reading data files, handling missing values, and preparing datasets for analysis.

4.1.2.5 Scikit-learn

Scikit-learn is the go-to toolbox for machine learning in Python. The library offers a lot of of ready-to-use algorithms for different scientific tasks like regression and classification, including most of the models evaluated in this study. It also provides essential utilities for data preprocessing and for measuring a model's performance.

4.1.2.6 XGBoost

XGBoost is a specialized and high-performance library dedicated to the gradient boosting algorithm. It is renowned for its speed and accuracy, often being a key tool in winning data science competitions. Its advanced features make it particularly effective for building powerful predictive models on large datasets.

4.1.2.7 Matplotlib and Seaborn

To visualize the data, Matplotlib and Seaborn were used together. Matplotlib is a highly flexible library that offers detailed control for creating a wide variety of charts and graphs. Seaborn is built on top of Matplotlib and makes it simpler to create visually appealing and informative statistical plots, which are perfect for uncovering trends and patterns.

4.1.2.8 Hyperparameter Tuning Tools

To find the best settings for the machine learning models, RandomizedCV from Scikit-learn was used. This tool automates the process of hyperparameter tuning by probabilistically testing out different combinations of model parameters from the parameters grid. It is significantly more efficient than Grid Search which is a brute force method to drain out all possibilities [16].

4.1.2.9 PyTorch

PyTorch is an open-source ML library that is popular among the community. While it is more famous for deep learning (which is not a focus of this research), it was included in the toolkit for its ability to leverage the machine's discrete graphics card (GPU). This allows for significant speed-ups in computation, which can be beneficial even for traditional machine learning tasks when working with large amounts of data.

4.1.2.10 Streamlit

Streamlit is a modern Python framework that makes it incredibly simple to build and share interactive web applications for machine learning projects. Its straightforward approach allows a data script to be turned into a shareable web app with minimal effort, making it ideal for creating user-friendly interfaces to showcase model predictions and data visualizations.

4.2 Dataset

The dataset to be the case for the research is collected from Kaggle. The HDB Resale Flat Price dataset is an accumulation of information relating to the sale of Singapore's public housing apartments colloquially known as "flats". Each record is one resale flat price transaction, with information describing the transaction of the sale and the features of the apartment. The original data was retrieved from Singapore's National Data Repository [17].

Chapter 5

Experiment

In the project, the experiment included data visualization strategies exploration, data preprocessing strategies selection, baseline model comparison, and hyperparameter tuning have been conducted. These steps built a strong fundamental for the final model selection and construction of dashboard with automated ML pipeline.

5.1 Setting up

5.1.1 Libraries

Before starting the experiment workflow, necessary Python libraries need to be imported.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
```

Figure 5.1 Libraries import

5.1.2 Dataset

```
# Load the dataset
housing = pd.read_csv('hdb_resale_flat_transactions.csv')
```

Figure 5.2 Dataset reading

```
housing.head(10)
```

	Unnamed: 0	month	town	blk_no	road_name	building	postal	resale_price	storey_range	flat_type	...	y	latitude	longitude	closest_mrt_station	distance_to_mrt_meters	transport_type	line_color	distance_to_cbd	closest_pri_school	distance_to_pri_school_meters
0	0	2017-01-01	ANG MO KIO	406	ANG MO KIO AVENUE 10	NIL	560406	232000.0	10 TO 12	2 ROOM	...	38229.067463	1.362005	103.853880	Ang Mo Kio	999.941618	MRT	Red	8615.656983	TOWNSVILLE PRIMARY SCHOOL	216.125254
1	1	2017-01-01	ANG MO KIO	108	ANG MO KIO AVENUE 4	KEBUN BARU HEIGHTS	560108	250000.0	01 TO 03	3 ROOM	...	39220.009892	1.370966	103.838202	Ang Mo Kio	1268.958246	MRT	Red	9715.131951	ANG MO KIO PRIMARY SCHOOL	241.572335
2	2	2017-01-01	ANG MO KIO	602	ANG MO KIO AVENUE 5	YIO CHU KANG GREEN	560602	262000.0	01 TO 03	3 ROOM	...	40297.283149	1.380709	103.835368	Yio Chu Kang	1072.295164	MRT	Red	10828.819556	ANDERSON PRIMARY SCHOOL	777.155378
3	3	2017-01-01	ANG MO KIO	465	ANG MO KIO AVENUE 10	TECK GHEE HORIZON	560465	265000.0	04 TO 06	3 ROOM	...	38693.098657	1.366201	103.857201	Ang Mo Kio	945.371842	MRT	Red	9097.929095	TECK GHEE PRIMARY SCHOOL	698.165530
4	4	2017-01-01	ANG MO KIO	601	ANG MO KIO AVENUE 5	YIO CHU KANG GREEN	560601	265000.0	01 TO 03	3 ROOM	...	40334.052030	1.381041	103.835132	Yio Chu Kang	1095.197729	MRT	Red	10869.453109	ANDERSON PRIMARY SCHOOL	782.553222
5	5	2017-01-01	ANG MO KIO	150	ANG MO KIO AVENUE 5	YIO CHU KANG GROVE	560150	275000.0	01 TO 03	3 ROOM	...	39865.816461	1.376807	103.842018	Yio Chu Kang	636.984920	MRT	Red	10301.960155	MAYFLOWER PRIMARY SCHOOL	145.413709
6	6	2017-01-01	ANG MO KIO	447	ANG MO KIO AVENUE 10	CHONG BOON CENTRE	560447	280000.0	04 TO 06	3 ROOM	...	38834.409128	1.367479	103.855967	Ang Mo Kio	763.116307	MRT	Red	9230.421051	TECK GHEE PRIMARY SCHOOL	595.480953
7	7	2017-01-01	ANG MO KIO	218	ANG MO KIO AVENUE 1	ANG MO KIO GROVE	560218	285000.0	04 TO 06	3 ROOM	...	38573.450044	1.365119	103.841742	Ang Mo Kio	1019.729295	MRT	Red	9020.925725	ANG MO KIO PRIMARY SCHOOL	520.749753
8	8	2017-01-01	ANG MO KIO	447	ANG MO KIO AVENUE 10	CHONG BOON CENTRE	560447	285000.0	04 TO 06	3 ROOM	...	38834.409128	1.367479	103.855967	Ang Mo Kio	763.116307	MRT	Red	9230.421051	TECK GHEE PRIMARY SCHOOL	595.480953
9	9	2017-01-01	ANG MO KIO	571	ANG MO KIO AVENUE 3	CHENG SAN COURT	560571	285000.0	01 TO 03	3 ROOM	...	39119.248864	1.370055	103.854881	Ang Mo Kio	592.480064	MRT	Red	9509.248190	JING SHAN PRIMARY SCHOOL	389.285674

10 rows × 30 columns

Figure 5.3 Brief view of dataset

Through the code, a brief view of the dataset is displayed which included 30 columns. Note that there is an additional index column (Unnamed: 0), which is redundant for the experiment purpose. Therefore, the additional index column is dropped out of the dataset without hesitate. After this step, 29 columns remaining.

```
# Drop the extra index column
housing = housing.drop('Unnamed: 0', axis = 1)
housing.head(10)
```

	month	town	blk_no	road_name	building	postal	resale_price	storey_range	flat_type	flat_model	...	y	latitude	longitude	closest_mrt_station	distance_to_mrt_meters	transport_type	line_color	distance_to_cbd	closest_pri_school	distance_to_pri_school_meters
0	2017-01-01	ANG MO KIO	406	ANG MO KIO AVENUE 10	NIL	560406	232000.0	10 TO 12	2 ROOM	Improved	...	38229.067463	1.362005	103.853880	Ang Mo Kio	999.941618	MRT	Red	8615.656983	TOWNSVILLE PRIMARY SCHOOL	216.125254
1	2017-01-01	ANG MO KIO	108	ANG MO KIO AVENUE 4	KEBUN BARU HEIGHTS	560108	250000.0	01 TO 03	3 ROOM	New Generation	...	39220.009892	1.370966	103.838202	Ang Mo Kio	1268.958246	MRT	Red	9715.131951	ANG MO KIO PRIMARY SCHOOL	241.572335
2	2017-01-01	ANG MO KIO	602	ANG MO KIO AVENUE 5	YIO CHU KANG GREEN	560602	262000.0	01 TO 03	3 ROOM	New Generation	...	40297.283149	1.380709	103.835368	Yio Chu Kang	1072.295164	MRT	Red	10828.819556	ANDERSON PRIMARY SCHOOL	777.155378
3	2017-01-01	ANG MO KIO	465	ANG MO KIO AVENUE 10	TECK GHEE HORIZON	560465	265000.0	04 TO 06	3 ROOM	New Generation	...	38693.098657	1.366201	103.857201	Ang Mo Kio	945.371842	MRT	Red	9097.929095	TECK GHEE PRIMARY SCHOOL	698.165530
4	2017-01-01	ANG MO KIO	601	ANG MO KIO AVENUE 5	YIO CHU KANG GREEN	560601	265000.0	01 TO 03	3 ROOM	New Generation	...	40334.052030	1.381041	103.835132	Yio Chu Kang	1095.197729	MRT	Red	10869.453109	ANDERSON PRIMARY SCHOOL	782.553222
5	2017-01-01	ANG MO KIO	150	ANG MO KIO AVENUE 5	YIO CHU KANG GROVE	560150	275000.0	01 TO 03	3 ROOM	New Generation	...	39865.816461	1.376807	103.842018	Yio Chu Kang	636.984920	MRT	Red	10301.960155	MAYFLOWER PRIMARY SCHOOL	145.413709
6	2017-01-01	ANG MO KIO	447	ANG MO KIO AVENUE 10	CHONG BOON CENTRE	560447	280000.0	04 TO 06	3 ROOM	New Generation	...	38834.409128	1.367479	103.855967	Ang Mo Kio	763.116307	MRT	Red	9230.421051	TECK GHEE PRIMARY SCHOOL	595.480953
7	2017-01-01	ANG MO KIO	218	ANG MO KIO AVENUE 1	ANG MO KIO GROVE	560218	285000.0	04 TO 06	3 ROOM	New Generation	...	38573.450044	1.365119	103.841742	Ang Mo Kio	1019.729295	MRT	Red	9020.925725	ANG MO KIO PRIMARY SCHOOL	520.749753
8	2017-01-01	ANG MO KIO	447	ANG MO KIO AVENUE 10	CHONG BOON CENTRE	560447	285000.0	04 TO 06	3 ROOM	New Generation	...	38834.409128	1.367479	103.855967	Ang Mo Kio	763.116307	MRT	Red	9230.421051	TECK GHEE PRIMARY SCHOOL	595.480953
9	2017-01-01	ANG MO KIO	571	ANG MO KIO AVENUE 3	CHENG SAN COURT	560571	285000.0	01 TO 03	3 ROOM	New Generation	...	39119.248864	1.370055	103.854881	Ang Mo Kio	592.480064	MRT	Red	9509.248190	JING SHAN PRIMARY SCHOOL	389.285674

10 rows × 29 columns

Figure 5.4 Dataset reading (removed additional index)

5.2 Exploratory Data Analysis (EDA)

The initial data exploration including visualized plots and graph is important for understanding the structure and characteristics of the dataset before proceeding to the preprocessing and evaluation stage.

5.2.1 Analysis on Dataset

```
Data columns (total 29 columns):
 #   Column                        Non-Null Count   Dtype
---  ------                        --------------   -----
 0   month                         212701 non-null  object
 1   town                          212701 non-null  object
 2   blk_no                        212701 non-null  object
 3   road_name                     212701 non-null  object
 4   building                      212701 non-null  object
 5   postal                        212701 non-null  int64
 6   resale_price                  212701 non-null  float64
 7   storey_range                  212701 non-null  object
 8   flat_type                     212701 non-null  object
 9   flat_model                    212701 non-null  object
 10  lease_commence_date           212701 non-null  object
 11  remaining_lease_years         212701 non-null  int64
 12  remaining_lease_months        212701 non-null  int64
 13  floor_area_sqm                212701 non-null  float64
 14  floor_area_sqft               212701 non-null  float64
 15  price_per_sqft                212701 non-null  float64
 16  planning_area_ura             212701 non-null  object
 17  region_ura                    212701 non-null  object
 18  x                             212701 non-null  float64
 19  y                             212701 non-null  float64
 20  latitude                      212701 non-null  float64
 21  longitude                     212701 non-null  float64
 22  closest_mrt_station           212701 non-null  object
 23  distance_to_mrt_meters        212701 non-null  float64
 24  transport_type                212701 non-null  object
 25  line_color                    212701 non-null  object
 26  distance_to_cbd               212701 non-null  float64
 27  closest_pri_school            212701 non-null  object
 28  distance_to_pri_school_meters 212701 non-null  float64
dtypes: float64(11), int64(3), object(15)
```

Figure 5.5 Columns (features) list

The dataset consists of 29 columns (features), and 212701 rows (records). The data type for the columns are majorly object and float64, and one column with int64.

```
# Check for missing values
missing_values = housing_eda.isnull().sum()
missing_percentage = (missing_values / len(housing_eda)) * 100
missing_housing = pd.DataFrame({
    'Missing Count': missing_values,
    'Missing Percentage': missing_percentage
}).sort_values('Missing Count', ascending=False)
print(missing_housing[missing_housing['Missing Count'] > 0])
```

```
Empty DataFrame
Columns: [Missing Count, Missing Percentage]
Index: []
```

Figure 5.6 Missing value checking

The displayed output shows that the dataset has no missing values.

```
duplicates = housing_eda.duplicated().sum()
print(f"Number of duplicate rows: {duplicates}")
```

```
Number of duplicate rows: 304
```

Figure 5.7 Duplicates checking

The duplicated() method naturally is comparing values of every column within one row and check with previous rows to see if there are any existing same row and mark the current row as duplicate if so. The output shows that dataset contains 304 rows of duplicates.

```
if duplicates > 0:
    print("Removing duplicate rows...")
    housing = housing.drop_duplicates()
    housing_eda = housing_eda.drop_duplicates()
    print(f"Original dataset shape after removing duplicates: {housing.shape}")
    print(f"Copied dataset shape after removing duplicates: {housing_eda.shape}")
```

```
Removing duplicate rows...
Original dataset shape after removing duplicates: (212397, 29)
Copied dataset shape after removing duplicates: (212397, 29)
```

Figure5.8 Duplicates removal

After dropping duplicates, there are 212397 rows remaining.

```
town: 26 unique values
['ANG MO KIO' 'BEDOK' 'BISHAN' 'BUKIT BATOK' 'BUKIT MERAH' 'BUKIT PANJANG'
'BUKIT TIMAH' 'CENTRAL AREA' 'CHOA CHU KANG' 'CLEMENTI']

blk_no: 2743 unique values
['406' '108' '602' '465' '601' '150' '447' '218' '571' '534']

road_name: 576 unique values
['ANG MO KIO AVENUE 10' 'ANG MO KIO AVENUE 4' 'ANG MO KIO AVENUE 5'
'ANG MO KIO AVENUE 1' 'ANG MO KIO AVENUE 3' 'ANG MO KIO AVENUE 9'
'ANG MO KIO AVENUE 8' 'ANG MO KIO AVENUE 6' 'ANG MO KIO STREET 52'
'BEDOK NORTH AVENUE 4']

building: 635 unique values
['NIL' 'KEBUN BARU HEIGHTS' 'YIO CHU KANG GREEN' 'TECK GHEE HORIZON'
'YIO CHU KANG GROVE' 'CHONG BOON CENTRE' 'ANG MO KIO GROVE'
'CHENG SAN COURT' 'CHENG SAN VIEW' 'KEBUN BARU PALM VIEW']

postal: 9660 unique values
[560406 560108 560602 560465 560601 560150 560447 560218 560571 560534]

storey_range: 17 unique values
['10 TO 12' '01 TO 03' '04 TO 06' '07 TO 09' '13 TO 15' '19 TO 21'
'22 TO 24' '16 TO 18' '34 TO 36' '28 TO 30']

flat_type: 7 unique values
['2 ROOM' '3 ROOM' '4 ROOM' '5 ROOM' 'EXECUTIVE' '1 ROOM'
'MULTI-GENERATION']

  flat_model: 21 unique values
  ['Improved' 'New Generation' 'DBSS' 'Standard' 'Apartment' 'Simplified'
  'Model A' 'Premium Apartment' 'Adjoined flat' 'Model A-Maisonette']

  planning_area_ura: 31 unique values
  ['ANG MO KIO' 'BEDOK' 'BISHAN' 'BUKIT BATOK' 'BUKIT MERAH' 'BUKIT PANJANG'
  'BUKIT TIMAH' 'TANGLIN' 'OUTRAM' 'ROCHOR']

  region_ura: 5 unique values
  ['NORTH-EAST REGION' 'EAST REGION' 'CENTRAL REGION' 'WEST REGION'
  'NORTH REGION']

  closest_mrt_station: 136 unique values
  ['Ang Mo Kio' 'Yio Chu Kang' 'Tanah Merah' 'Bedok' 'Kembangan' 'Eunos'
  'Marymount' 'Bishan' 'Bukit Gombak' 'Bukit Batok']

  transport_type: 2 unique values
  ['MRT' 'LRT']

  line_color: 7 unique values
  ['Red' 'Green' 'Orange' 'Purple' 'Grey' 'Blue' 'Brown']

  closest_pri_school: 169 unique values
  ['TOWNSVILLE PRIMARY SCHOOL' 'ANG MO KIO PRIMARY SCHOOL'
  'ANDERSON PRIMARY SCHOOL' 'TECK GHEE PRIMARY SCHOOL'
  'MAYFLOWER PRIMARY SCHOOL' 'JING SHAN PRIMARY SCHOOL'
  "ST. ANTHONY'S CANOSSIAN PRIMARY SCHOOL" 'FENGSHAN PRIMARY SCHOOL'
  'BEDOK GREEN PRIMARY SCHOOL' 'YU NENG PRIMARY SCHOOL']
```

Figure 5.9 Unique values checking

Next, the cardinality of categorical columns is checked. The cardinality of these columns varies from 2 to 9660 unique values in the column. This indicates that single One-hot encoder is not efficient enough to handle these columns, but separate encoding strategies are needed in the next preprocessing stage.

```
def check_if_all_numeric(series):
    try:
        # Try to convert all values to numeric
        pd.to_numeric(series, errors='raise')
        return True, "All values are numeric"
    except (ValueError, TypeError):
        # Find non-numeric values
        non_numeric = pd.to_numeric(series, errors='coerce').isna() & series.notna()
        non_numeric_values = series[non_numeric].unique()
        return False, f"Non-numeric values found: {non_numeric_values}"

# Usage
is_numeric, message = check_if_all_numeric(housing_eda['blk_no'])
print(f"{message}")
```

```
Non-numeric values found: ['588C' '588D' '588A' ... '478A' '479B' '479C']
```

Figure 5.10 Detailed unique values checking for blk_no

The previous output only displays numerical values for ‘blk_no’ column, but the data type of the column is recognized as ‘object’. Therefore, the unique value within the column is further investigated. A function is created to examine if the column contains alphabetical (non-numerical) values. The output indicates that the column is containing non-numerical values, which cut off the option to directly convert the column to numerical column.

5.2.2 Analysis on Data Distribution

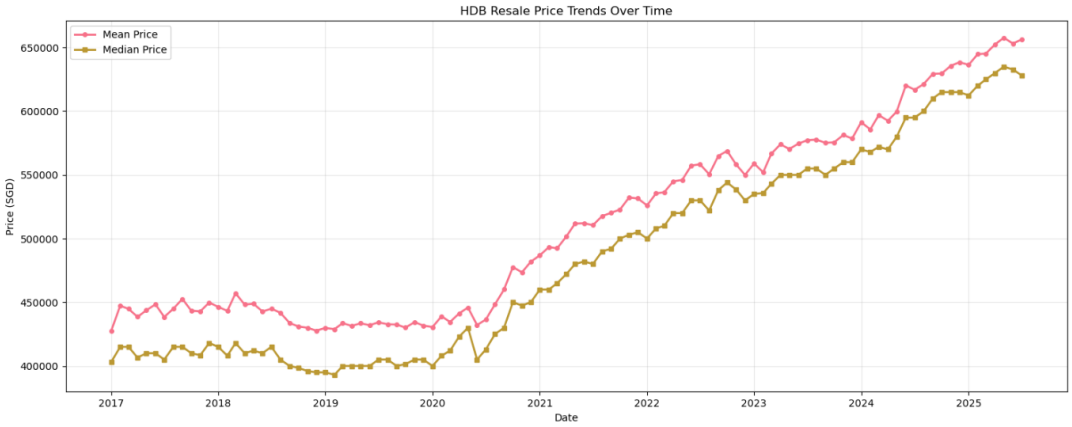


Figure 5.11 Price trend graph

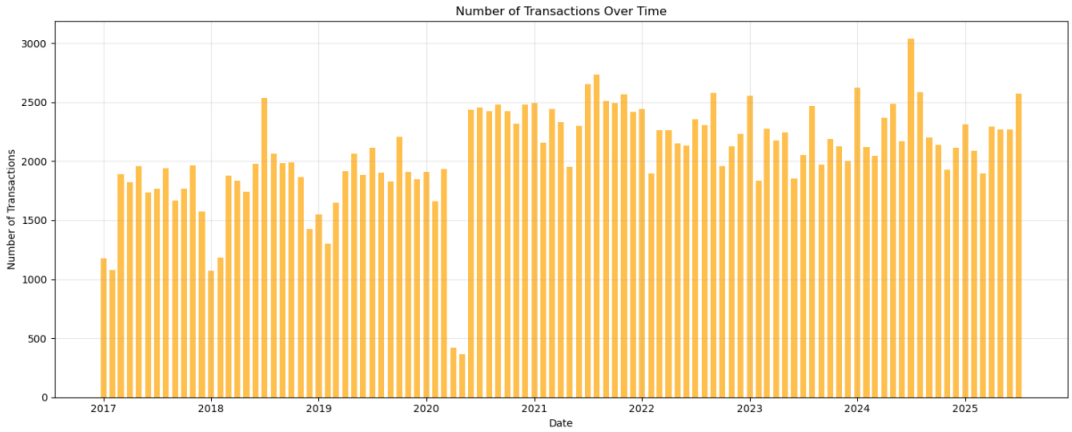


Figure 5.12 Transactions frequency chart

A line chart was chosen to plot the mean and median resale prices over time, as it is the most effective method for visualizing trends across a continuous interval. This was supplemented with a bar chart to show the corresponding transaction volume for each month. Based on the graphs, the dataset recorded the transactions from 2017 until July 2025. The graph shows the price trend, which is a clear upward trend in both mean and median resale prices over the period covered by the dataset, reflecting the overall appreciation of the Singapore property market.

On the other hand, the volume of transactions is not constant; it fluctuates, showing potential seasonality or reactions to market events and policy changes. This hints that temporal features might have an impact on the resale price. Therefore, we need to handle the feature carefully.

Figure 5.13 Geospatial plotting

The scatter plot of property longitude and latitude was generated. The resale price was encoded onto a colour scale, effectively transforming the raw coordinate data into an intuitive price heat map of Singapore. This strategy makes complex geospatial patterns instantly understandable. The resulting visualization clearly highlights price "hotspots" in the central and southern regions, which are geographically closer to the CBD and other key amenities. It also reveals relative "cold spots" in outlying areas. This direct visualization of spatial price disparity is a powerful insight for any potential buyer or market analyst and directly achieves the project's goal of making complex data accessible.

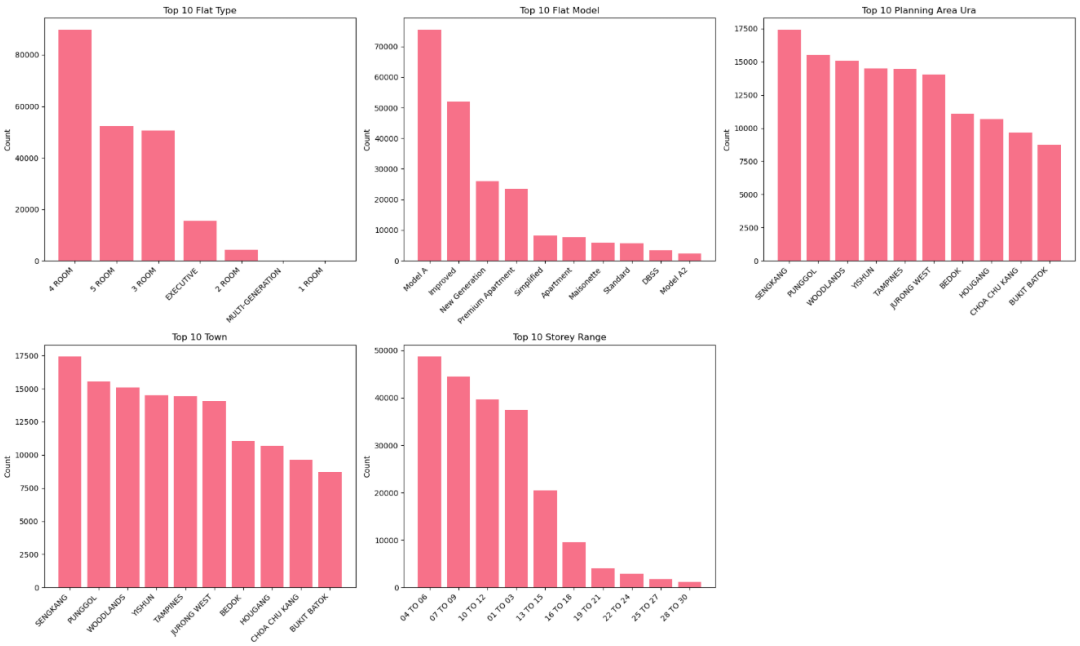


Figure 5.14 Categorical column analysis

Simple bar charts were used to display the frequency counts of key categorical features like town and flat type. This is the standard and most direct way to compare the magnitude of discrete categories. 4-room are the most frequent room type occurred in the transactions, followed by 5-room and 3-room. Besides, towns like Sengkang, Punggol, and Woodlands have the highest transaction volumes, indicating they are highly active markets.

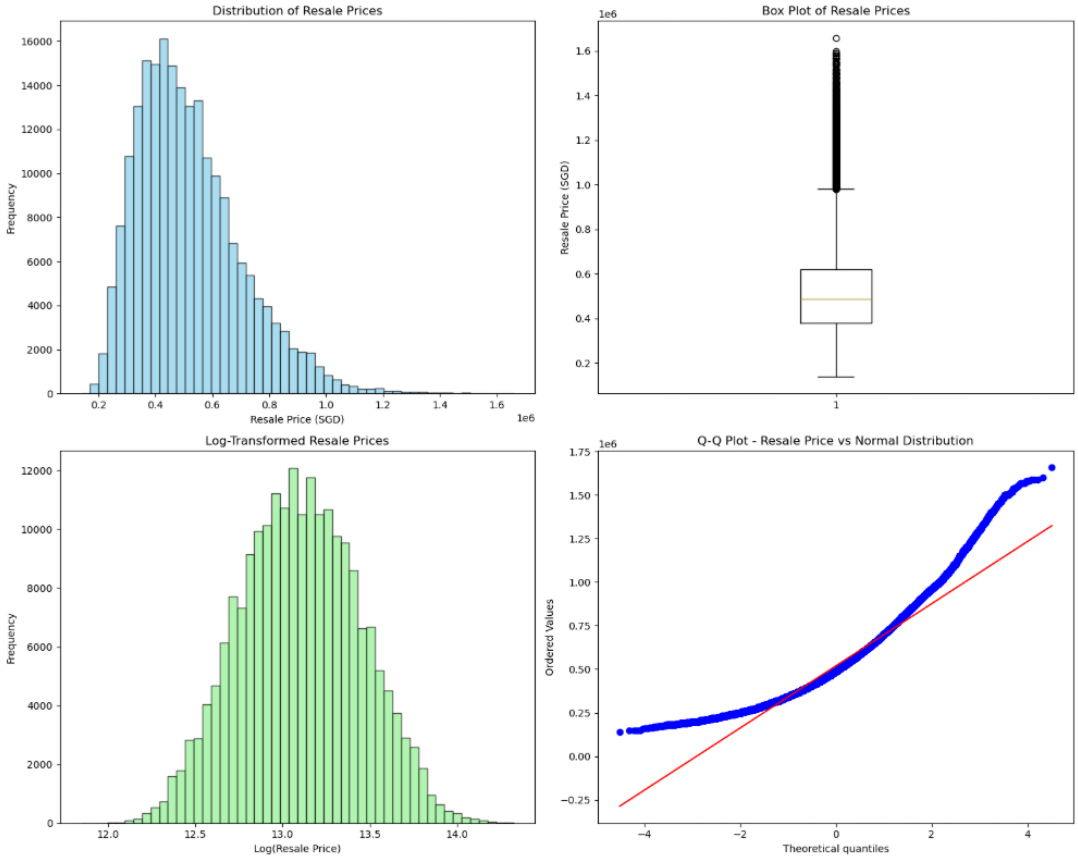


Figure 5.15 Target variable analysis

A multi-plot approach was applied for thoroughly investigate the distribution of the target. This included a histogram for frequency distribution, a box plot to identify statistical spread and outliers, and a Q-Q plot to formally check for normality. The histogram and Q-Q plot together prove that the prices are not normally distributed but are strongly right-skewed. The box plot further emphasizes this by explicitly showing that a large number of high-value properties exist as outliers. This is not only a valuable insight into market inequality but also a critical technical finding. These visualizations provide a clear justification for the subsequent log-transformation of the target variable.

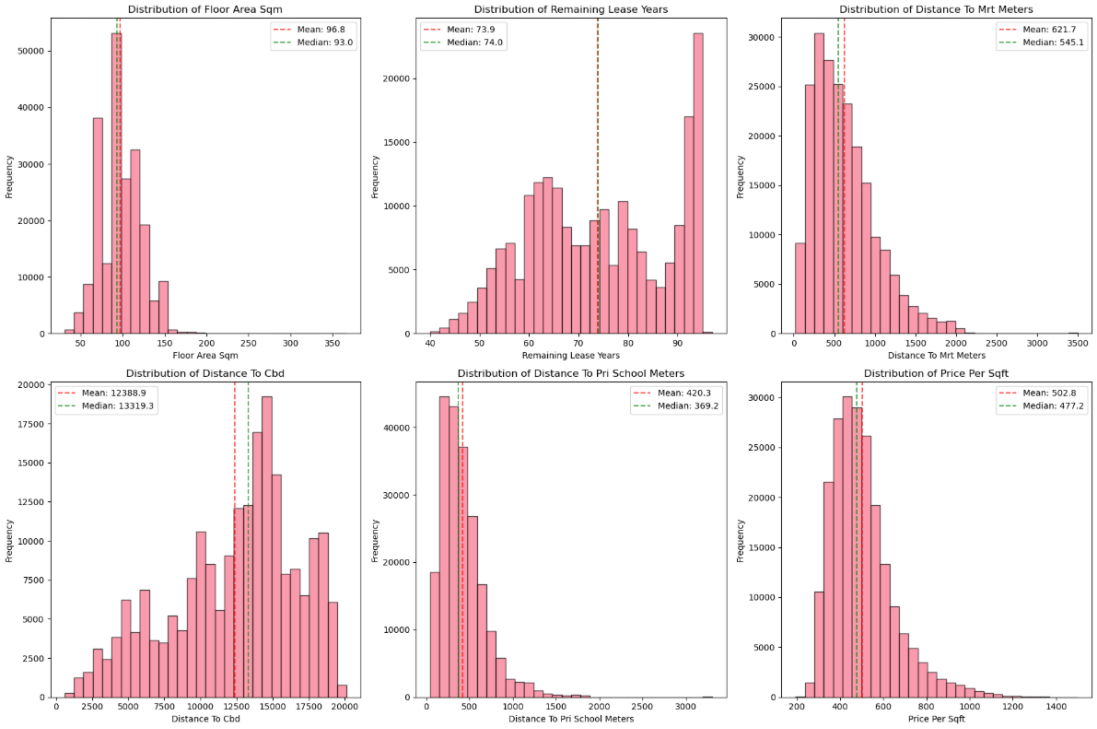


Figure 5.16 Numerical features distribution analysis

The charts show the data distribution of independent numerical variables. Based on the charts, floor area, distance to MRT, distance to primary school, and price per square foot are right-skewed, while remaining lease years and distance to CBD tend to be left-skewed. This shows the complexity of real-world data.

Even though it is possible to use transformation techniques like log-transformation and Box-Cox transformation to handle the skewness, it is decided to not to normalize the distribution of these independent variables during the data preprocessing stage, as the current state of distribution can help to assess different models' capabilities to handle real-world complex data. The extra step of normalization will also make the workflow unnecessarily complex. Box-Cox transformation will also increase the difficulty to interpret the data as it is not as direct as log-transformation. Here, a hypothesis can be made that tree-based models can better understand these skewed columns, as they do not care about the magnitude of the data.

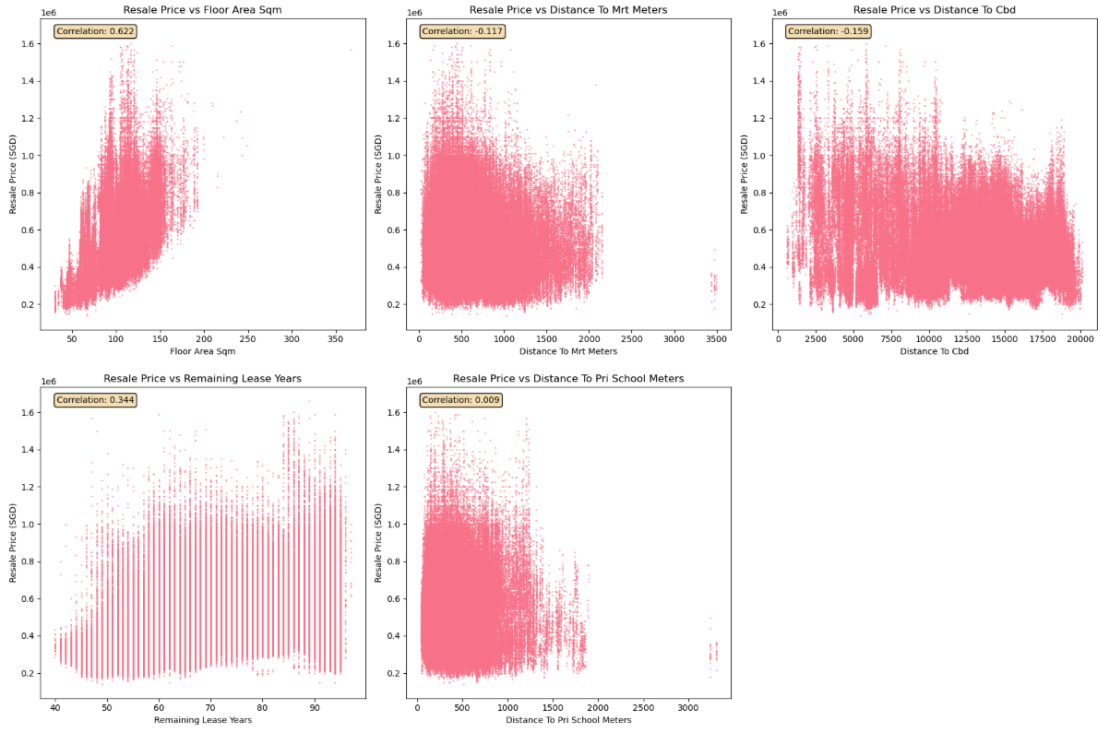


Figure 5.17 Numerical features vs target variable

To understand the factors driving price, scatter plots were created to visualize the relationship between key numerical columns (e.g., floor_area_sqft, distance_to_mrt_meters) with the resale_price. It clearly stated a strong positive trend between a flat's area and its price, and the negative trend where price decreases as the distance from an MRT station increases. This visual exploration forms the basis for understanding which factors are most influential and provides a qualitative precursor to the statistical correlation analysis.

5.2.3 Summary & Insights

Different EDA steps and data visualization strategies have been applied in this stage. A clear view of the dataset characteristics has been shaped, which is helpful to determine the data preprocessing methods to be taken in the next stage. The visualization techniques applied such as geospatial plotting and trend graphs can be useful to integrate into the dashboard of the final AutoML application.

5.3 Data Preprocessing

Based on insights gained from previous EDA section, appropriate data preprocessing strategies can be determined and applied to the dataset to ensure the data quality and model performance.

5.3.1 Initial Feature Selection

```
# Drop the 6 columns
col_to_drop = ['price_per_sqft', # target leakage
               'floor_area_sqm',
               'blk_no',
               'road_name',
               'building',
               'postal',
               'x',
               'y'] # redundant

housing_preprocessed = housing_preprocessed.drop(columns = col_to_drop, axis = 1)
```

Figure 5.18 Initial drop

Mathematically, price_per_sqft is highly correlated to the target (price_per_sqft = resale_price/floor_area_sqft), which might cause target leakage if it is included for model training. It is also not suitable for multi-target regression, as the manual calculation is much simpler than doing regression.

According to metadata and the result in EDA section, there are floor_area_sqm and floor_area_sqft, which essentially represent the same meaning which is the size of the house. As there is price_per_sqft which uses the same unit, the floor_area_sqft is kept and another one is dropped.

On the other hand, although blk_no, road_name, building, and postal can be treated as geographical attributes, there are already latitude and longitude which can represent the geographical attributes of the house in a more precise way, and town and planning_area_ura to provide high-level grouping of location.

According to the metadata of the dataset, x and y are coordinate of the house displayed in CRS SVY21 (Singaporean Coordinate Reference System SVY21), which are the exact same use to latitude and longitude. Therefore, these columns are dropped at the beginning.

5.3.2 Data Splitting

```
# Define feature (X) and target (y)
X  =  housing_preprocessed.drop('resale_price', axis = 1)
y = housing_preprocessed['resale_price']
```

```
# Sort by month to preserve temporal order
sort_idx = X['month'].sort_values().index
X = X.loc[sort_idx]
y = y.loc[sort_idx]

# Temporal split by position
n = len(X)
train_end = int(n * 0.70)
val_end  = int(n * 0.85)

X_train, y_train = X.iloc[:train_end],        y.iloc[:train_end]
X_val,   y_val   = X.iloc[train_end:val_end], y.iloc[train_end:val_end]
X_test,  y_test  = X.iloc[val_end:],          y.iloc[val_end:]
```

```
print("Train set shape:", X_train.shape)
print("Validation set shape:", X_val.shape)
print("Test set shape:", X_test.shape)
```

```
Train set shape: (148677, 20)
Validation set shape: (31860, 20)
Test set shape: (31860, 20)
```

Figure 5.19 Data splitting

The whole dataset is first sorted in order by month to maintain the time-series characteristics of the data. The dataset is later split following 70:15:15 ratio. This is to ensure there is no information leakage, e.g. the model is trained with future data during the later stages to the greatest extent.

5.3.3 Log Transformation

```
# Apply log transformation to the target variable
y_train_log = np.log(y_train)
y_val_log = np.log(y_val)
y_test_log = np.log(y_test)

print("Log transformation applied to target variables:")
print(f"Original mean: {y_train.mean():.2f}, std: {y_train.std():.2f}")
print(f"Log-transformed mean: {y_train_log.mean():.4f}, std: {y_train_log.std():.4f}")
```

```
Log transformation applied to target variables:
Original mean: 518755.12, std: 183401.43
Log-transformed mean: 13.0997, std: 0.3447
```

Figure 5.20 Log-transformation on target variable

As shown in the EDA section, the target variable is right-skewed and includes quite a number of outliers. Therefore, it is necessary to perform log-transformation on it to eliminate the skewness and outliers and also establish a more linear relationship between independent and dependent variables.

5.3.4 Feature Encoding

```
# Categorize different types of column
# 1. Temporal columns
temporal_col = ['month', 'lease_commence_date']
lease_col = ['remaining_lease_years', 'remaining_lease_months',]

# 2. Categorical columns
high_cardinality_col = ['town', 'flat_model', 'planning_area_ura',
                        'closest_mrt_station', 'closest_pri_school']

low_cardinality_col = ['flat_type', 'region_ura',
                       'transport_type', 'line_color']

ordinal_col = ['storey_range']

# 3. Numerical columns (except target variable)
numerical_col = ['floor_area_sqft', 'latitude', 'longitude',
                 'distance_to_mrt_meters', 'distance_to_cbd',
                 'distance_to_pri_school_meters']
```

Figure 5.21 Column categorization

```
preprocessor = ColumnTransformer(
    transformers=[
        # Extract temporal features with cyclical encoding
        ('temporal', TemporalFeatureExtractor(), temporal_col),

        # Create total remaining lease months feature
        ('lease', LeaseFeatureCreator(), lease_col),

        # Ordinal encoding for ordinal column
        ('ordinal', OrdinalEncoder(categories=[storey_order]), ordinal_col),

        # Target encoding with cross-validation
        ('target_encode', CVTargetEncoderWrapper(n_folds=5, smoothing=1.0), high_cardinality_col),

        # One-hot encoding for low cardinality columns
        ('onehot', OneHotEncoder(sparse_output=False, drop='first', handle_unknown='ignore'), low_cardinality_col),

        # Standard scaling for numerical features
        ('numerical', StandardScaler(), numerical_col)
    ],
    remainder='drop',  # Drop any remaining columns
    verbose_feature_names_out=False
)
```

Figure 5.22 Encoding pipeline

Before encoding the features, the existing columns are divided into several groups according to their characteristics. Then, a column transformer is constructed to handle these columns with appropriate methods respectively.

For the temporal columns, 2 custom encoders are created. The custom encoders served the purpose of performing cyclical encoding on temporal columns. Cyclical encoding is a method used to encode date into numerical features using sine and cosine transformations to preserve the wrap-around nature of the data. It represents them as coordinates on a circle, where nearby values on the original cycle are also numerically close in the new representation.

On the other hand, another custom encoder is created for high-cardinality columns. The custom encoder served the purpose of performing target encoding to these high-cardinality columns, as they are not suitable for traditional One-hot encoding which will create enormous new columns, which can cause high-dimensionality issues. Note that target encoding will bring in the risk of information leakage due to its characteristics, therefore the custom encoder integrated K-fold cross validation to prevent the leakage.

For other remaining columns, common encoder or scaler are applied such as ordinal encoder for storey range as it has an order naturally from low to high, One-hot encoder for low-cardinality columns, and standard scaler for numerical columns.

```
Transforming train data...
     Shape: (148677, 40)

Transforming validation data...
     Shape: (31860, 40)

Transforming test data...
     Shape: (31860, 40)
```

Figure 5.23 Shape of transformed dataset

After feature encoding, the new data frame contains 40 numerical features.

5.3.5 Correlation and Multicollinearity Analysis

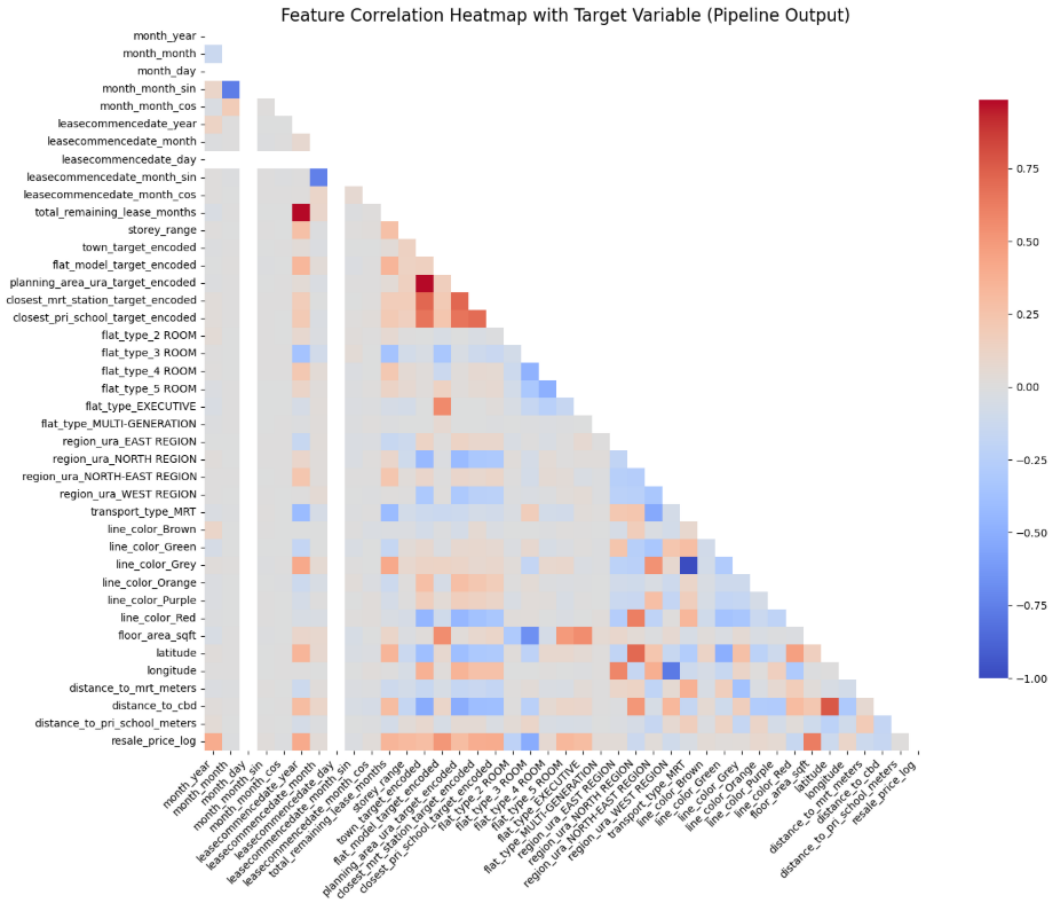


Figure 5.24 Correlation heatmap

A correlation heatmap is created to understand the correlation between features. As a result, most of the features did not show clear linear relationships with each other. The highest correlation is 0.6617, which is floor_area_sqft with log-transformed resale_price. The lowest

correlation can go down to no correlation at all, including month_day and leasecommencedate_day, which both are the results of feature encoding.

```
--- Features with VIF > 10 (High Multicollinearity) ---
                           feature           VIF
0                        month_year  7.659517e+02
1                       month_month  1.277406e+01
5            leasecommencedate_year  2.664069e+04
6           leasecommencedate_month  1.243557e+01
10     total_remaining_lease_months  2.629092e+04
12             town_target_encoded  2.806653e+01
14  planning_area_ura_target_encoded  2.662939e+01
17                 flat_type_2 ROOM  5.457539e+01
18                 flat_type_3 ROOM  5.131506e+02
19                 flat_type_4 ROOM  7.081337e+02
20                 flat_type_5 ROOM  5.525405e+02
21              flat_type_EXECUTIVE  2.087965e+02
24          region_ura_NORTH REGION  1.420061e+01
26           region_ura_WEST REGION  1.337867e+01
27               transport_type_MRT           inf
30                  line_color_Grey           inf
34                  floor_area_sqft  1.352185e+01
35                         latitude  2.273348e+01
36                        longitude  1.324557e+01
38                  distance_to_cbd  1.541989e+01
```

Figure 5.25 Multicollinearity checking

Multicollinearity has also been investigated. Half of the features have shown a high variance inflation factor (VIF) score, with some even achieved infinity. Although they did not show strong linear correlations, they will be kept and used for model training to evaluate the performance of models to handle real world data.

5.3.6 Output for the next stage

After the preprocessing stage, original dataset is divided into the required train set and validation set for the following stages, and the test set for final stage with 40 features. The next stage (baseline model training and evaluation) will be using train and validation set.

5.4 Baseline Model Comparison

Model / Metrics	Linear Regression	SVM	Decision Tree	Random Forest	GBM	XGBoost
RMSE	58116.14	90172.99	67615.04	50503.35	58791.01	43791.16
MAE	41937.05	72165.69	48262.51	36983.83	42160.86	32062.79
R^2	0.8929	0.7421	0.8550	0.9191	0.8904	0.9392
Adjusted R^2	0.8927	0.7417	0.8548	0.9190	0.8902	0.9391
Training Time (s)	0.0959	27.0918	1.7207	13.8801	25.6832	0.3147
R^2/s	9.3067	0.0274	0.4969	0.0662	0.0347	2.9841
Efficiency	2312.17	130622.68	29390.96	59220.52	83849.93	5203.88
Time Penalty	1.0000	282.3931	17.9359	144.6804	267.7106	3.2805

Table 5.1: Comparison of Baseline Models

6 selected models have been trained and validated using train and validation sets. After training, the model's performance is validated, and the table shows the summary of the results from 2 perspectives, one is accuracy including R^2, Adjusted R^2, RMSE, MAE, and one is efficiency including training time, R^2 per seconds, efficiency score, and time penalty to the baseline model.

5.4.1 RMSE and MAE

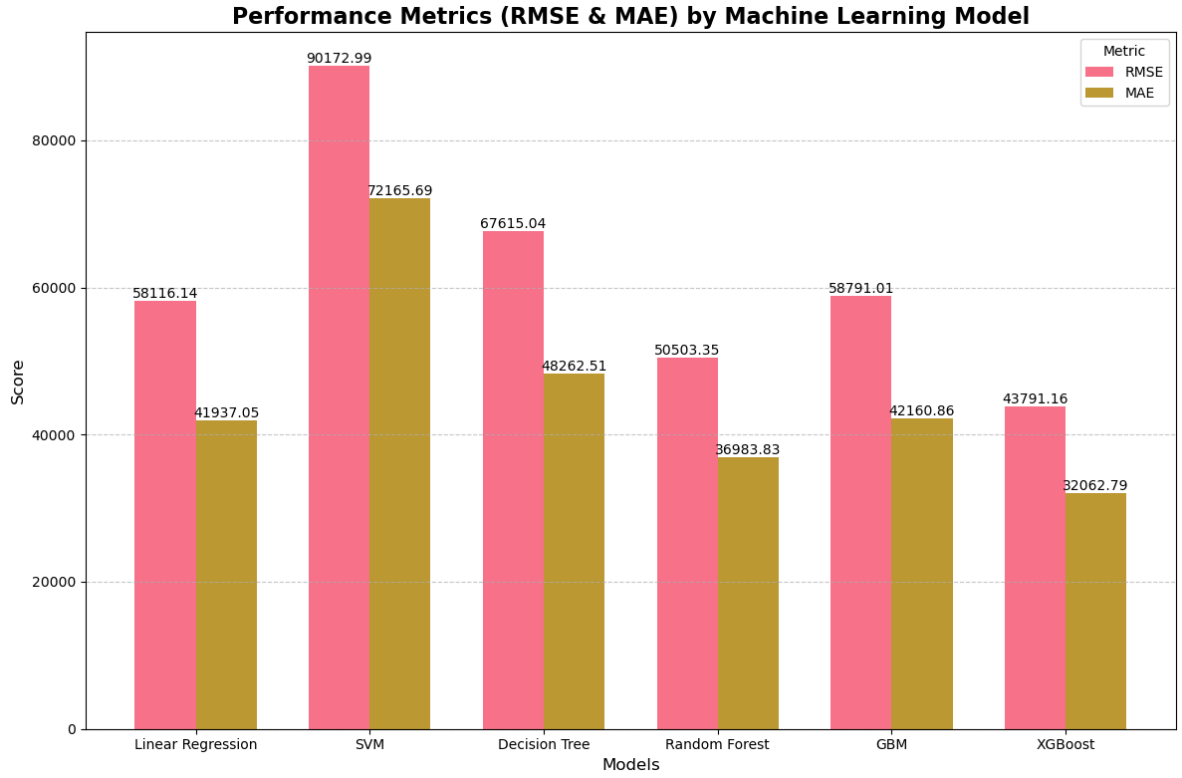


Figure 5.26 Clustered chart (RMSE, MAE)

The XGBoost model demonstrated the highest precision with an RMSE of $43,791.16, indicating it has the smallest average deviation from actual resale prices. Random Forest followed as a strong second at $50,503.35. Both Linear Regression ($58,116.14) and GBM ($58,791.01) showed moderate error levels, while the Decision Tree ($67,615.04) is worse than them. The SVM model performed the worst among all models in this metric, with a significantly higher RMSE of $90,172.99, suggesting it struggled to capture the variance in the HDB price dataset.

In terms of MAE, XGBoost again led the pack with an MAE of $32,062.79, meaning its predictions are typically within roughly $32k of the true price. Random Forest remained competitive at $36,983.83. Linear Regression ($41,937.05) and GBM ($42,160.86) have their results closed to each other, representing a middle ground in predicting power. The Decision Tree recorded an MAE of $48,262.51, while SVM showed the largest average error at $72,165.69, nearly double that of the top-performing ensemble models.

5.4.2 R^2 and Adjusted R^2

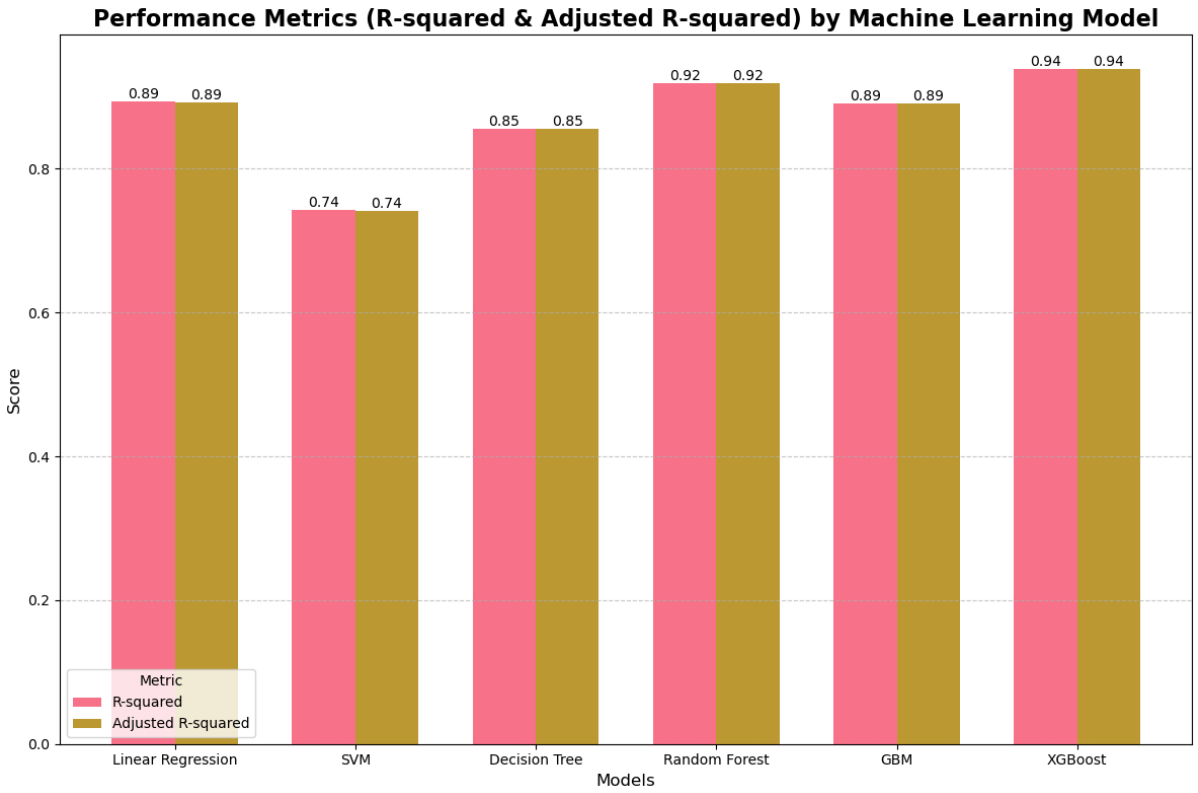


Figure 5.27 Clustered chart (R^2, Adjusted R^2)

The baseline LR model achieved 0.891 for R^2, meaning it explained about 89% of the variancE. This provided a reasonable benchmark for explanatory power. The SVM fell below the baseline with an R^2 of 0.717, confirming its inability to adequately capture variability in the dataset.

In contrast, the Decision Tree reached an R^2 of 0.947, showing stronger explanatory power than the baseline, though this came with potential risks of overfitting. The GBM performed similarly, with an R^2 of 0.933, also surpassing the baseline. Random Forest and XGBoost delivered the highest explanatory power, achieving R^2 values of 0.974 and 0.973, respectively, both significantly outperforming the baseline. These results indicate that ensemble methods not only reduce error but also provide the best model fit to the data.

Across all models, the Adjusted R^2 values were remarkably close to their standard counterparts, suggesting that the 40 features generated by the preprocessing pipeline are meaningful and do not introduce significant noise. XGBoost led with 0.9391, followed by Random Forest (0.9190), Linear Regression (0.8927), and GBM (0.8902). The minimal drop-off from the standard R^2 indicates that the model complexity is well-justified by the dataset size.

5.4.3 Training Time

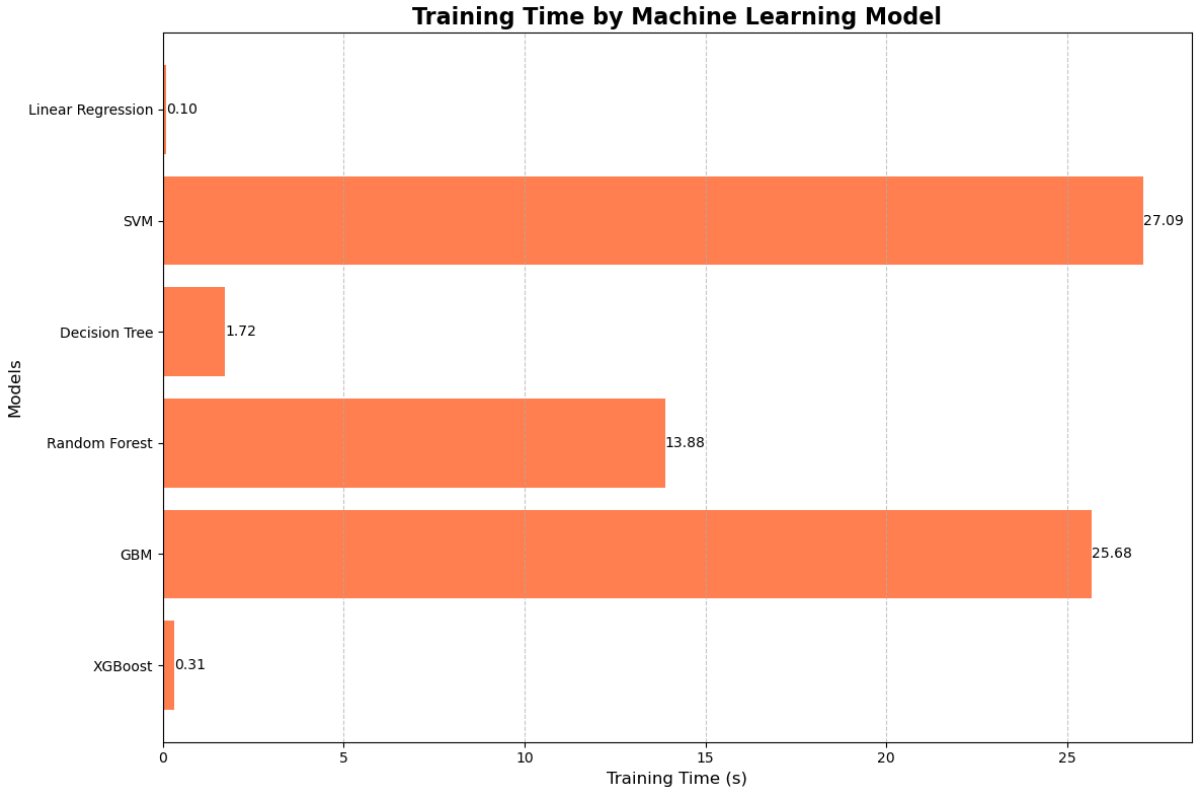


Figure 5.28 Bar chart (training time in s)

There was a massive disparity in computational efficiency; Linear Regression was the fastest, completing in just 0.096 seconds, followed closely by XGBoost at 0.315 seconds. The Decision Tree took 1.72 seconds, while the ensemble models required more resources: Random Forest took 13.88 seconds, and GBM took 25.68 seconds. SVM was the most computationally expensive, requiring 27.09 seconds—over 280 times longer than the baseline linear model—despite its lower accuracy. This is due to the native support of GPU acceleration from the library [18]. This highlights XGBoost's efficiency advantage in real-world applications.

5.4.4 R^2/s

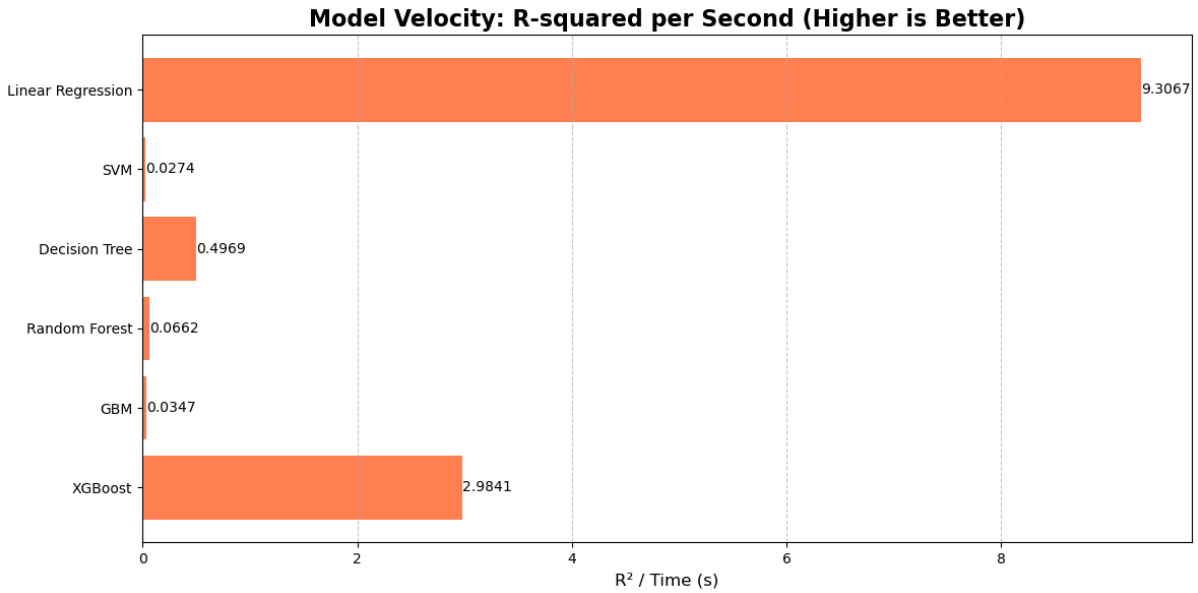


Figure 5.29 Bar chart (R^2/s)

Linear Regression as the baseline model is the clear winner in this section, with an efficiency of 9.3067 R^2/s. XGBoost followed with a strong 2.9841, proving it is highly efficient given its high accuracy. The other models dropped off significantly, with the Decision Tree (0.4969), Random Forest (0.0662), and GBM (0.0347) showing much lower efficiency. SVM was the least efficient at 0.0274.

5.4.5 Efficiency

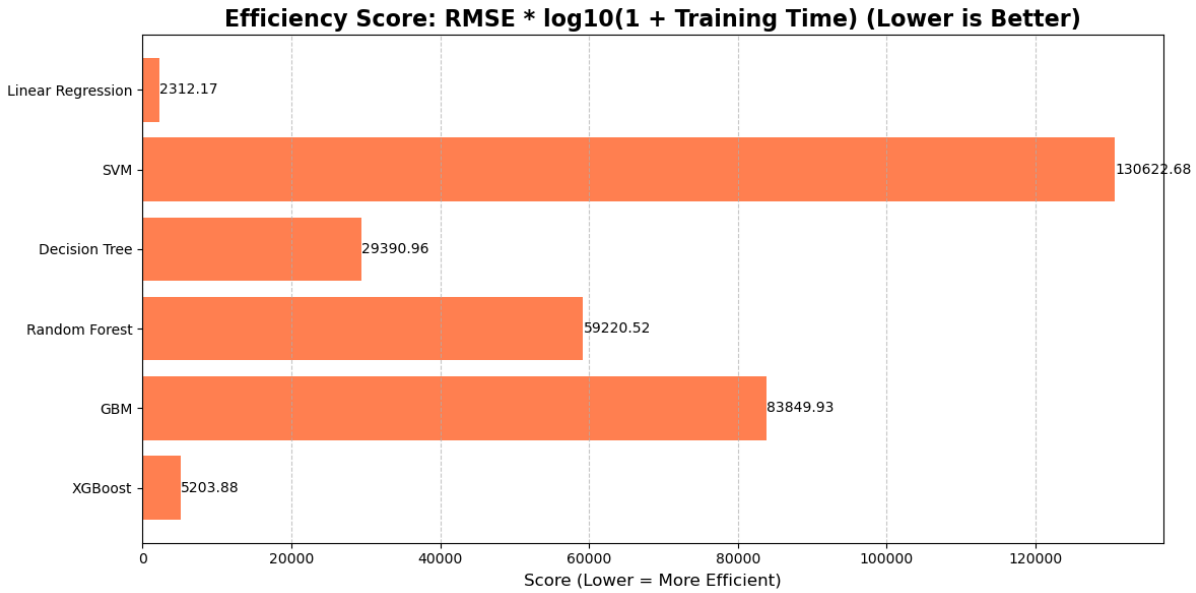


Figure 5.30 Bar Chart (Efficciency)

Linear Regression achieved the best (lowest) score of 2312.17, primarily due to its extremely fast training. XGBoost gained a strong 5203.88, suggesting it is the most optimal model when considering both high accuracy and speed. In contrast, the SVM had the worst efficiency score at 130,622.68, penalized heavily for both its high error rate and long training duration. The other models are in between, with the Decision Tree (29390.66), Random Forest (59220.52), and GBM (83849.93)

5.4.6 Time Penalty

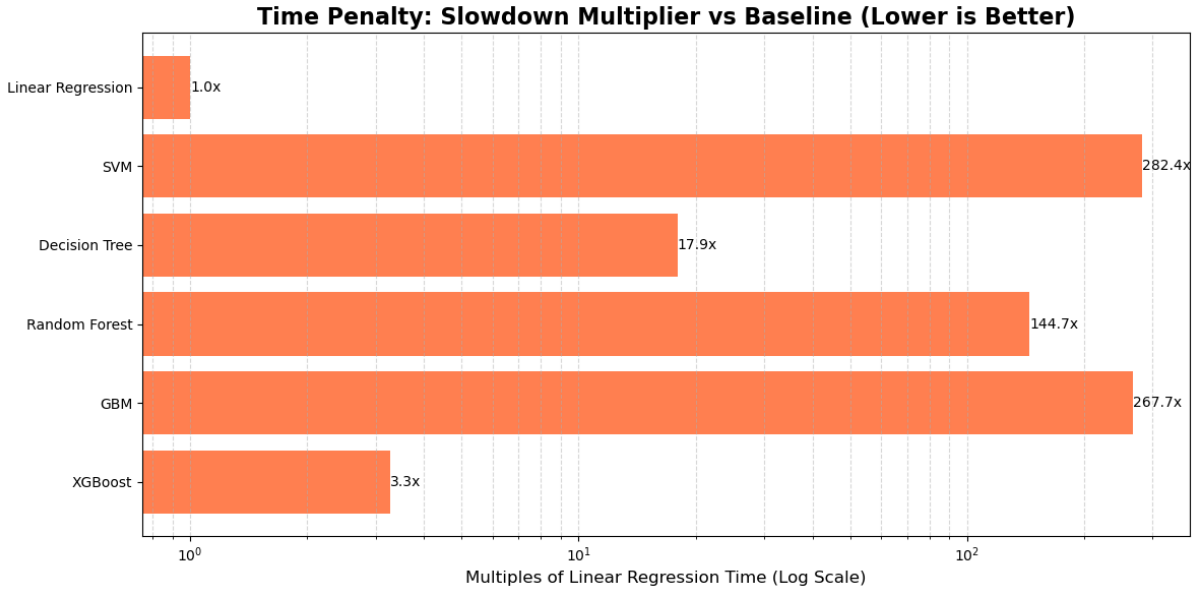


Figure 5.31 Bar Chart (Time Penalty)

Using Linear Regression as the 1x baseline, the XGBoost model is 3.28x slower which is negligible considering the predictive power. The Decision Tree is 17.9x slower, while the more complex ensemble models like Random Forest (144.7x) and GBM (267.7x) represent significant jumps in processing requirements. The SVM represents the most extreme trade-off, being 282.4x slower than the baseline while actually providing less accurate estimation.

5.4.7 Overfitting Analysis

Model / Dataset	Linear Regression	SVM	Decision Tree	Random Forest	GBM	XGBoost
Validation Set	0.8929	0.74221	0.8550	0.9191	0.8904	0.9392
Train Set	0.8787	0.7694	0.9998	0.9962	0.9278	0.9745

Table 5.2 Overfitting Analysis Summary

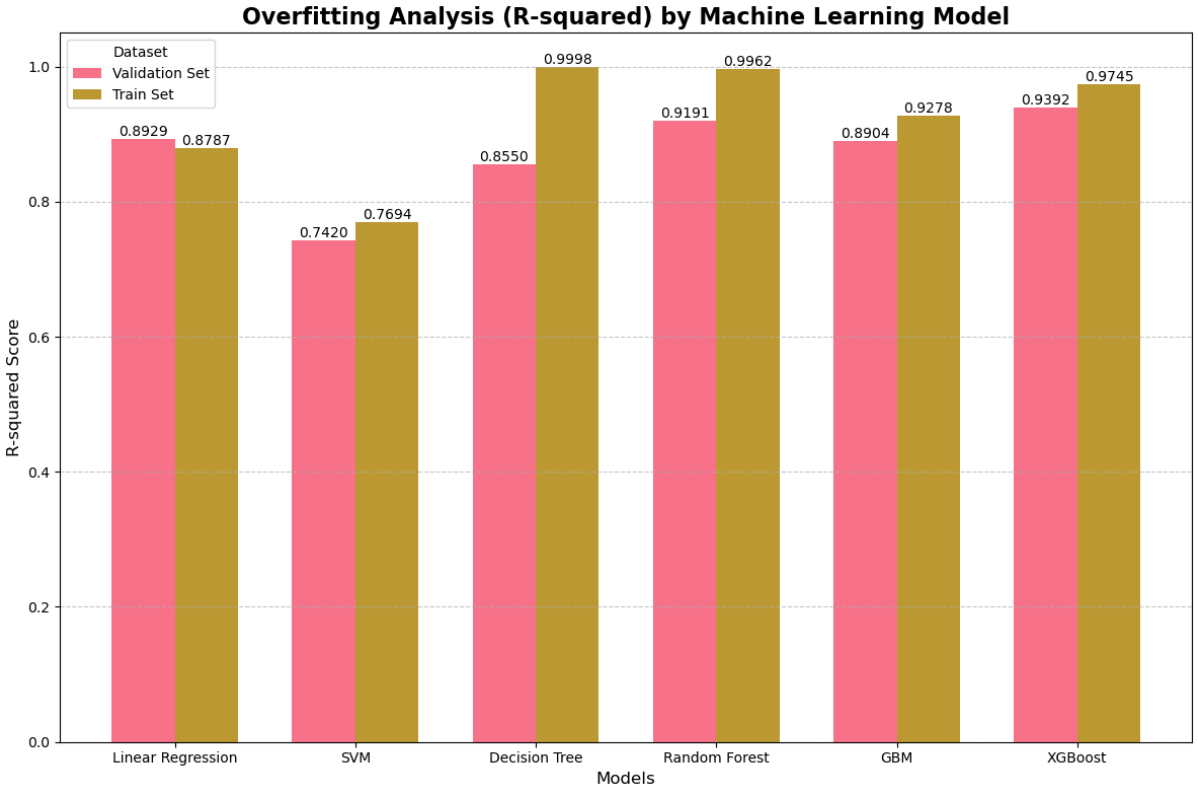


Figure 5.32 Overfitting Analysis (R^2)

Comparing R^2 scores between training and validation sets provides further insight into model generalization. The baseline Linear Regression achieved 0.889 on training and 0.891 on validation, showing almost identical performance. This consistency indicates little to no overfitting, but the moderate scores suggest underfitting, as the model does not fully capture the complexity of the dataset. Similarly, SVM recorded 0.713 on training and 0.717 on validation, consistent but lower than the baseline, reinforcing its underfitting behavior.

The Decision Tree, however, achieved an almost perfect R^2 of 0.99998 on training but dropped to 0.947 on validation, indicating strong overfitting—memorizing the training data but generalizing less effectively. Random Forest achieved 0.996 on training and 0.974 on validation, showing high accuracy with only mild overfitting. XGBoost displayed the best balance, with 0.976 on training and 0.973 on validation, achieving high accuracy with minimal loss in generalization. GBM maintained consistent performance across sets, with 0.933 on training and 0.933 on validation, outperforming the baseline without signs of severe overfitting. Overall, the analysis shows that while the Decision Tree is prone to overfitting, ensemble methods like Random Forest and XGBoost strike the best balance between fitting the training data and maintaining predictive power on the validation set. Conversely, Linear Regression and SVM do not overfit but remain limited.

5.4.8 Feature Importance

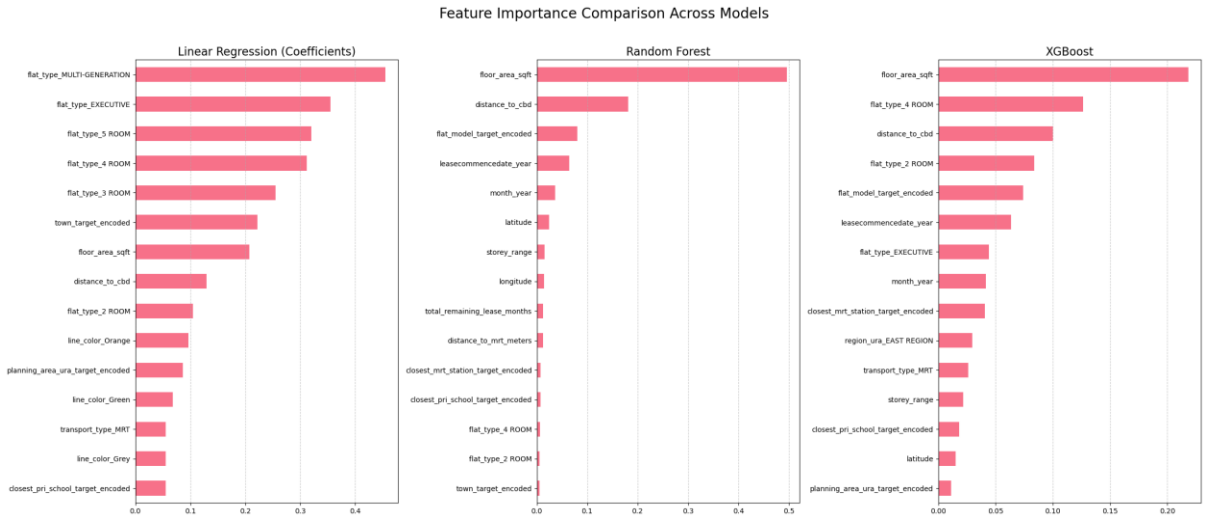


Figure 5.33 Feature importance

The feature importance analysis reveals that across all types of models, housing prices are primarily driven by floor area (size of the unit), transaction timing (month and year), and proximity to the central business district (CBD), confirming the dominant role of structural and locational attributes in price determination. In the Linear Regression model, categorical encodings such as flat types, MRT proximity, and even line color indicators appear influential, which reflects the model's sensitivity to dummy variables and its limited ability to capture non-linear interactions.

By contrast, the Random Forest model places overwhelming emphasis on floor area, with month of sale and distance to CBD following at a distance, suggesting a strong reliance on a small set of dominant predictors. XGBoost demonstrates a more balanced distribution of feature importance: while floor area remains the most critical factor, temporal variables, distance to CBD, flat model, remaining lease, and storey range also contribute meaningfully, indicating that the model integrates both structural and categorical variables more effectively. Collectively, the results highlight that while model-specific biases exist, the consistent importance of floor area, timing, and location underscores their central role in shaping property values.

5.4.9 Analysis in Baseline Result

Overall, Linear Regression served as the baseline model, delivering moderate predictive accuracy with the advantage of being the fastest to train. However, it underfit the data and left significant room for improvement. SVM performed worse than the baseline across all metrics, showing both high error and low explanatory power.

Decision Tree and GBM improved on the baseline but showed limitations—Decision Tree by overfitting, and GBM by achieving only moderate accuracy gains. The strongest results came from ensemble methods. XGBoost achieved the lowest and highest explanatory power, and even achieved the second fastest training speed. Random Forest matched XGBoost in accuracy while training time is longer.

Taken together, these results demonstrate that while Linear Regression provided a useful baseline, XGBoost and Random Forest significantly outperformed it, with XGBoost offering the best balance between accuracy, efficiency, and generalization.

In summary, ensemble tree-based methods, particularly XGBoost, provide the most reliable and well-rounded approach for housing price prediction with this dataset as of now. The result also shows the hypothesis that was made at the end of section 4.2.2 is correct, which tree-based algorithms can better interpret the complex, non-linear variables.

5.5 Hyperparameter Tuning

After the baseline comparison, 4 selected models (LR, RF, GBM, XGBoost) have been carried over to the hyperparameter tuning stage. The process followed several procedures, such as the following:

1. **Grid Definition**: For each model, a parameters grid is defined which includes the key parameters that will affect the model's performance and the range of applicable value.
2. **Predefined Split**: As the dataset follows time-series trend, cross-validation is not applicable. Splitting of train set and validation set in hyperparameter tuning is predefined at the beginning of the stage.
3. **Performance Evaluation**: RMSE is the primary metric used in this stage, particularly in randomized search. Comprehensive evaluation is done after the best parameters combinations are found, with complete set of metrics.
4. **Validation**: The found combinations of best parameters were carried over to run the full validation with validation set.

5.5.1 Parameters Grid

5.5.1.1 Linear Regression

```
lr_param_dist = {
    'fit_intercept': [True, False],
    'positive':      [True, False],
}
```

Figure 5.34 Parameters Grid (LR)

Linear Regression has minimal hyperparameters as it is a relatively simple, closed-form model with no stochastic or regularization components. The search space was limited to two Boolean parameters:

- **fit_intercept [True, False]**: The parameter is to determine whether to calculate the intercept for the model. In practice, it controls whether a bias term is added. It is set to True by default, and no intercept will be used if set to False [19].
- **positive [True, False]**: The parameter is set to False by default. It is used to control the polarity constraint of the coefficient, which the coefficient will be forced to be positive if set to True [19].

5.5.1.2 Random Forest

```
rf_param_dist = {
    'n_estimators':      randint(100, 500),
    'max_depth':         [None] + list(range(10, 30)),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf':  randint(1, 10),
    'max_features':      uniform(0.3, 0.7),    # fraction of features per split
    'bootstrap':         [True, False],
}
```

Figure 5.35 Parameters Grid (RF)

The Random Forest search space covered six selected parameters governing both ensemble structure and individual tree complexity.

- **n_estimators [100, 500]**: The parameter controls the number of trees, 100 by default [20].
- **max_depth [None, 10, 30]**: The maximum depth of the tree, which None is the default value, which nodes will continue to expand until all the leaves fulfil certain constraints i.e. all are pure or the number of contained samples is less than min_samples_split [20]. This parameter may help with balancing depth against the risk of overfitting.
- **min_samples_split [2, 20]**, **min_samples_leaf [1, 10]**: The parameters control the minimum samples number that are required to split a node internally, and a leaf to have [20]. Both act as regularisation by setting minimum thresholds for a node to split or to remain a leaf, preventing the trees from memorising noise.
- **max_features [0.3, 0.7]**: The parameter controls the features number that will be considered when searching for optimal split [19]. The value is drawn from a range of float numbers unlike others mentioned which are in integer. In practice, it controls the fraction of features considered at each split.
- **bootstrap [True, False]**: The parameter controls whether bootstrap samples are used when training the trees. It is set to True by default, and the whole dataset is used to train each tree if set to False [20].

5.5.1.3 GBM

```
gbr_param_dist = {
    'n_estimators':   randint(100, 500),
    'max_depth':      randint(3, 8),
    'learning_rate':  uniform(0.01, 0.29),
    'subsample':      uniform(0.6, 0.4),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf':  randint(1, 10),
    'max_features':   uniform(0.3, 0.7),
}
```

Figure 5.36 Parameters Grid (GBM)

The GBM parameter grid reflects the sequential, additive nature of gradient boosting where each successive tree corrects residuals from prior trees.

- **n_estimators [100, 500]**, **learning_rate [0.01, 0.29]**: The parameters control the number of boosting stages, and the contribution of trees respectively. Both parameters are jointly searched as there is a trade-off between them, in which a smaller learning rate generally requires more trees to converge, and vice versa [21].
- **max_depth [3, 8]**: The parameter controls the depth of individual regression estimatiors, which also constrains the number of nodes within the tree. The parameter defaults the value to 3 [21].
- **subsample [0.6, 0.4]**: The parameter is set to 1.0 by default. It introduces stochasticity by training each tree on a random sub-sample of the data. Stochastic Gradient Boosting is the result if smaller than 1.0 [21].
- **min_samples_split [2, 20]**, **min_samples_leaf [1, 10]**: The parameters control the minimum samples number that are required to split a node internally, and a leaf to have [21]. Both act as regularisation by setting minimum thresholds for a node to split or to remain a leaf, preventing the trees from memorising noise.
- **max_features [0.3, 0.7]**: The parameter controls the features number that will be considered when searching for optimal split [21].

5.5.1.4 XGBoost

```
xgb_param_dist = {
    'n_estimators':     randint(100, 500),
    'max_depth':        randint(3, 10),
    'learning_rate':    uniform(0.01, 0.29),    # samples from [0.01, 0.30]
    'subsample':        uniform(0.6, 0.4),       # samples from [0.6, 1.0]
    'colsample_bytree': uniform(0.6, 0.4),
    'min_child_weight': randint(1, 10),
    'reg_alpha':        uniform(0, 1),           # L1
    'reg_lambda':       uniform(0.1, 2.9),       # L2
}
```

Figure 5.37 Parameters Grid (XGBoost)

XGBoost extends gradient boosting with built-in L1 and L2 regularisation, making its parameter grid the richest among the four models.

- **n_estimators [100, 500]**, **learning_rate over [0.01, 0.29]**: The parameters control the number of boosting stages, and the contribution of trees respectively [22]. Both parameters are jointly searched as there is a trade-off between them, which a smaller learning rate generally requires more trees to converge, and vice versa.
- **subsample over [0.6, 0.4]**: The parameter is set to 1.0 by default. It introduces stochasticity by training each tree on a random sub-sample of the data. Stochastic Gradient Boosting is the result if smaller than 1.0 [22].
- **max_depth [3, 10]**: The parameter controls the depth of individual regression estimatiors, which also constrains the number of nodes within the tree. The parameter defaults the value to 6 [22].
- **colsample_bytree [0.6, 0.4]**: It controls the subsample ratio of columns when training each tree, adding another dimension of feature-level stochasticity [22].
- **min_child_weight [1, 10]**: The parameter defines the minimum Hessian instance weights sum that is needed for a child node. The value sets to 1 by default, and the algorithm will be more conservative as the value becomes larger [22].
- **reg_alpha (L1, [0, 1])**, **reg_lambda (L2, [0.1, 2.9])**: Both parameters penalise large leaf weights directly in the objective function, the model will become more conservative as any of the parameters increased [22].

5.6 Tuned Models Comparison

Model / Metrics	Linear Regression	Linear Regression (Tuned)	Random Forest	Random Forest (Tuned)
RMSE	58116.14	58116.14	50503.35	49092.61
MAE	41937.05	41937.05	36983.83	36043.30
R^2	0.8929	0.8929	0.9191	0.9235
Adjusted R^2	0.8927	0.8927	0.9190	0.9234

Table 5.3 Performance Summary after Hyperparameter Tuning (LR & RF)

Model / Metrics	GBM	GBM (Tuned)	XGBoost	XGBoost (Tuned)
RMSE	58791.01	38460.95	43791.16	40774.77
MAE	42160.86	28118.80	32062.79	29483.65
R^2	0.8904	0.9531	0.9392	0.9473
Adjusted R^2	0.8902	0.9530	0.9391	0.9472

Table 5.4 Performance Summary after Hyperparameter Tuning (GBM & XGBoost)

Model / Metrics	Linear Regression (Tuned)	Random Forest (Tuned)	GBM (Tuned)	XGBoost (Tuned)
Training Time (s)	0.2976	35.0114	173.4550	0.6727
R^2/s	3.0006	0.0264	0.0055	1.4081
Efficiency	6574.39	76409.71	86217.28	9110.16
Time Penalty (to baseline model)	3.1016	364.9444	1808.0229	7.0123
Time Penalty (to default parameters)	3.1016	2.5224	6.7536	2.1375

Table 5.5 Efficiency Summary after Hyperparameter Tuning

Model / Dataset	Linear Regression	Linear Regression (Tuned)	Random Forest	Random Forest (Tuned)
Validation Set	0.8929	0.8929	0.9191	0.9235
Train Set	0.8787	0.8787	0.9962	0.9931

Table 5.6 Overfitting Analysis Summary (LR & RF)

Model / Dataset	GBM	GBM (Tuned)	XGBoost	XGBoost (Tuned)
Validation Set	0.8904	0.9531	0.9392	0.9473
Train Set	0.9278	0.9818	0.9745	0.9777

Table 5.7 Overfitting Analysis Summary (GBM & XGBoost)

5.6.1 RMSE and MAE

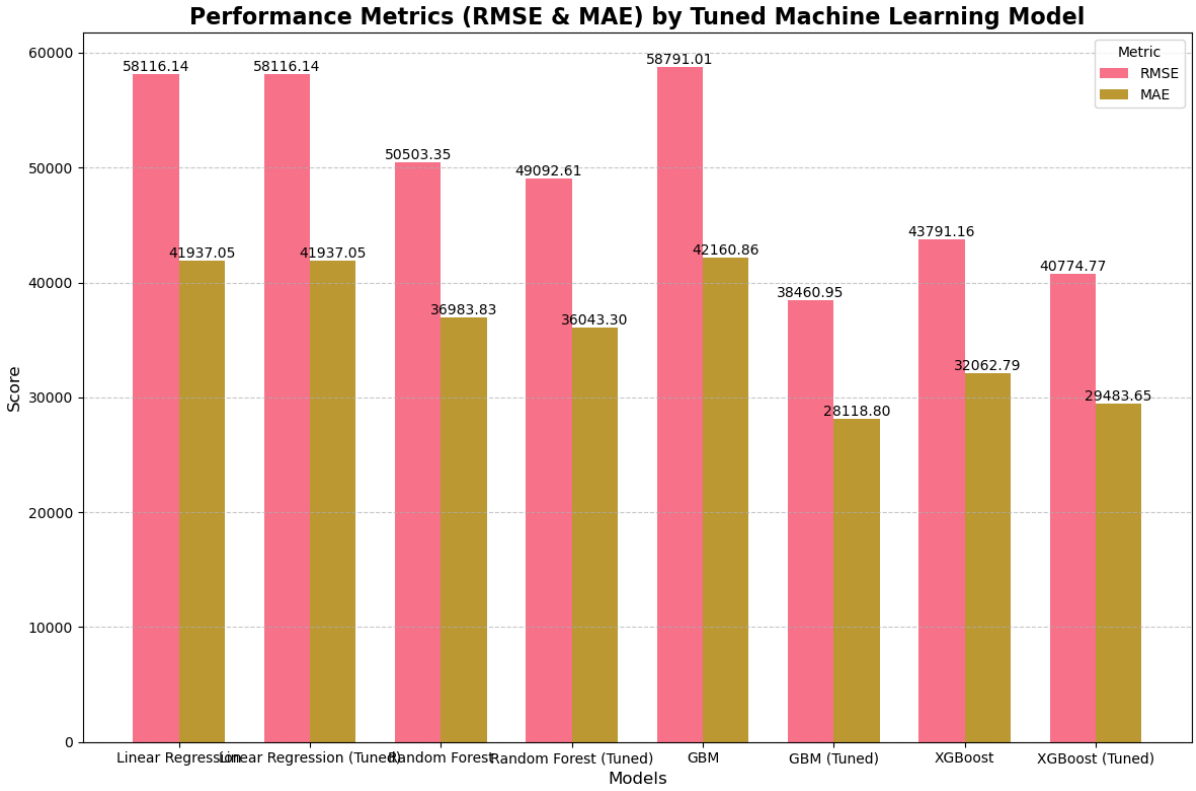


Figure 5.38 Clustered chart after Hyperparameter Tuning (RMSE, MAE)

Hyperparameter tuning significantly improved the predictive accuracy of the boosting models, with GBM (Tuned) achieving the lowest overall error of $38,460.95, a substantial drop from its baseline. XGBoost (Tuned) followed with a strong $40,774.77, while Random Forest (Tuned) showed a slight improvement to $49,092.61. Notably, Linear Regression (Tuned) remained unchanged at $58,116.14, as the search confirmed that the default parameters were already optimal for the linear baseline.

The average prediction error followed a similar downward trend, with GBM (Tuned) leading at $28,118.80, indicating that its predictions are typically off by only about $28k. XGBoost (Tuned) was nearly as precise with an MAE of $29,483.65. Random Forest (Tuned) improved to $36,043.30, while Linear Regression (Tuned) stayed at its baseline value of $41,937.05, showing that tuning had the most profound impact on the non-linear ensemble models.

5.6.2 R^2 and Adjusted R^2

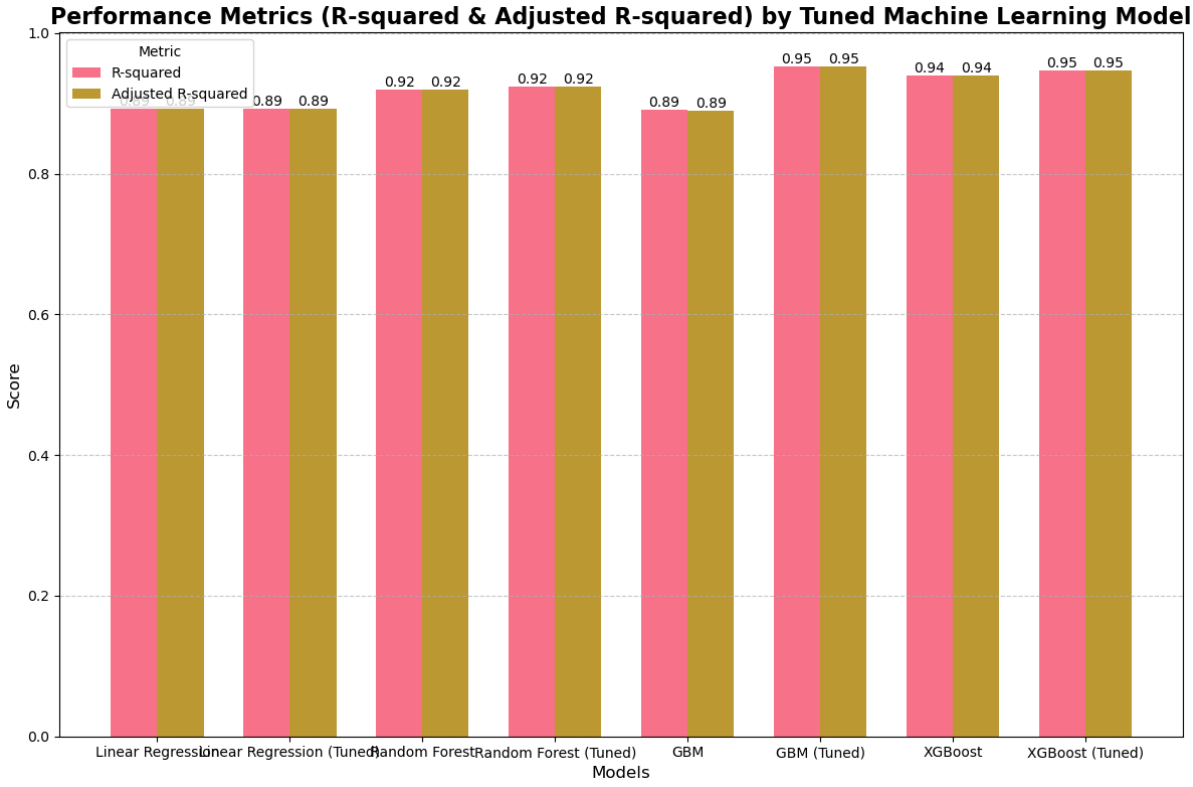


Figure 5.39 Clustered chart after Hyperparameter Tuning (R^2, Adjusted R^2)

The explanatory power of the models reached new heights after tuning, particularly for GBM (Tuned), which now explains 95.31% of the variance in resale prices. XGBoost (Tuned) also saw an increase to 0.9473, maintaining its position as a top-tier performer. Random Forest (Tuned) rose to 0.9235, while Linear Regression (Tuned) remained at 0.8929. These scores confirm that the tuned ensemble models are exceptionally well-suited for the complexities of the real estate market.

The Adjusted R^2 values mirrored the standard R^2 scores, with GBM (Tuned) leading at 0.9530 and XGBoost (Tuned) at 0.9472. Random Forest (Tuned) followed with 0.9234, while Linear Regression (Tuned) held steady at 0.8927. The continued proximity between these two metrics indicates that even with more complex tuned configurations, the models are not being penalized for unnecessary feature inclusion.

5.6.3 Training Time

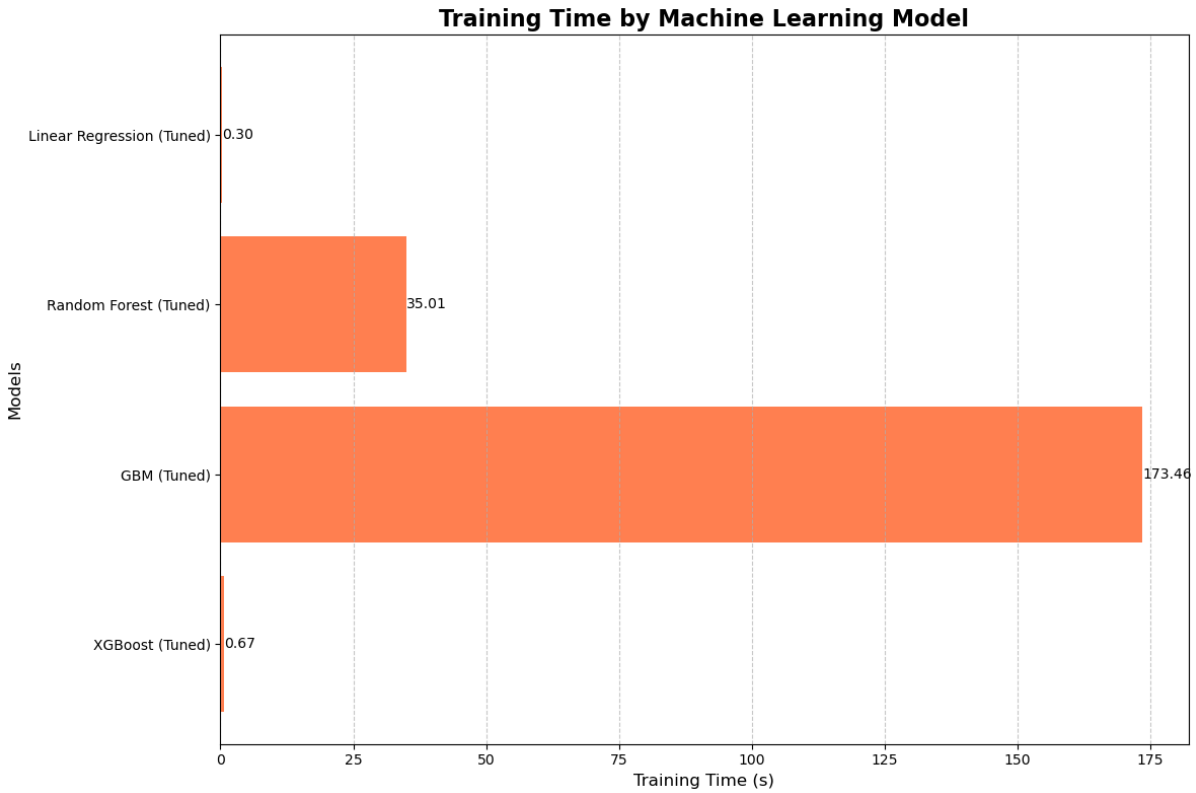


Figure 5.40 Bar chart after Hyperparameter Tuning (training time in s)

Hyperparameter tuning resulted in significantly longer training durations, especially for the tree-based ensemble models. GBM (Tuned) became the most time-intensive, requiring 173.46 seconds to train, while Random Forest (Tuned) took 35.01 seconds. On the other hand, XGBoost (Tuned) remained impressively efficient, training in just 0.67 seconds. Linear Regression (Tuned) took 0.30 seconds, a slight increase from its baseline but still nearly instantaneous.

5.6.4 R^2/s

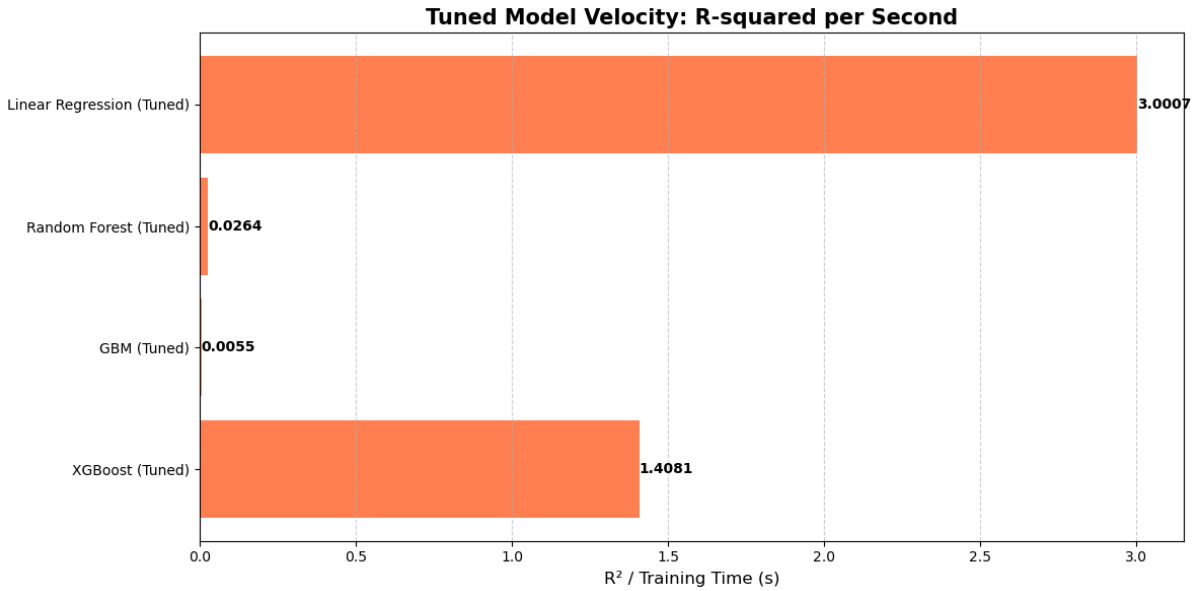


Figure 5.41 Bar chart after Hyperparameter Tuning (R^2/s)

The velocity metric highlights the impressive efficiency of LR (Tuned) and XGBoost (Tuned), which achieved 3.0007 and 1.4081 R^2/s, respectively. Because of their massive training times, Random Forest (Tuned) (0.0264) and GBM (Tuned) (0.0055) had very low velocity. This has shown that while GBM is the most accurate, it requires significantly more computational resources for every R^2 explained.

5.6.5 Efficiency

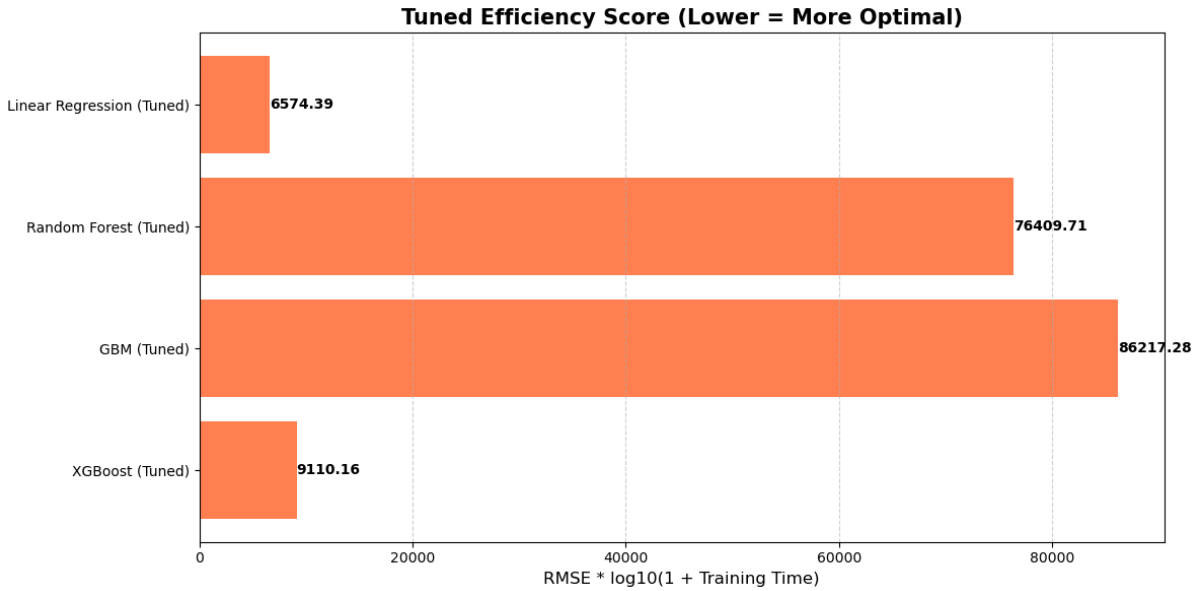


Figure 5.42 Bar chart after Hyperparameter Tuning (Efficiency)

Linear Regression (Tuned) achieved the top place with a score of 6574.39. XGBoost (Tuned) was close at second place at 9110.16, proving it is superior in high performance predictions that require efficiency. The scores for Random Forest (Tuned) (76,409.71) and GBM (Tuned) (86,217.28) showed a heavy penalty for their training times despite the lower RMSE.

5.6.6 Time Penalty

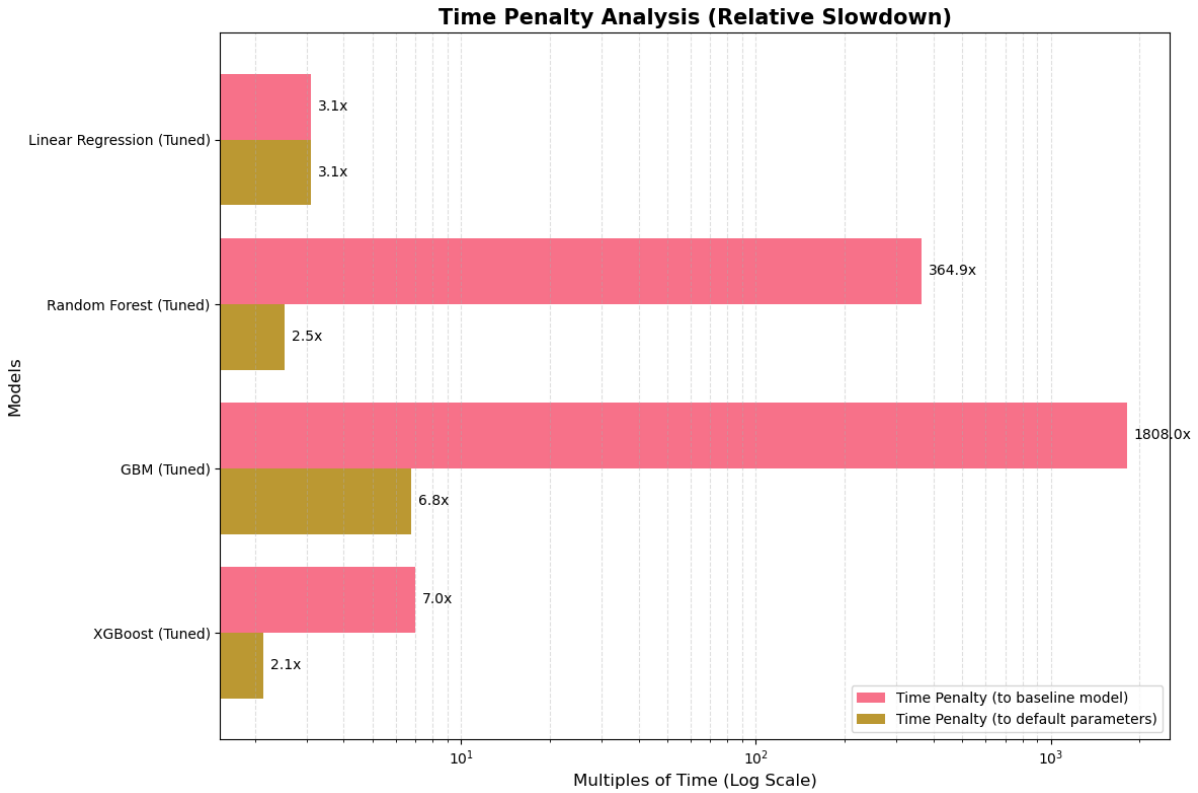


Figure 5.43 Bar chart after Hyperparameter Tuning (Time penalty)

The slowdown caused by tuning was most extreme for GBM, which was 1808.0x slower than the original linear baseline and 6.8x slower than its own default settings. Random Forest was 364.9x slower than the baseline. In comparison, XGBoost (Tuned) proved to be highly practical, with only a 7.0x penalty relative to the baseline and 2.1x slowdown compared to its default model, making it the most balanced model in the final evaluation.

5.6.7 Overfitting Analysis

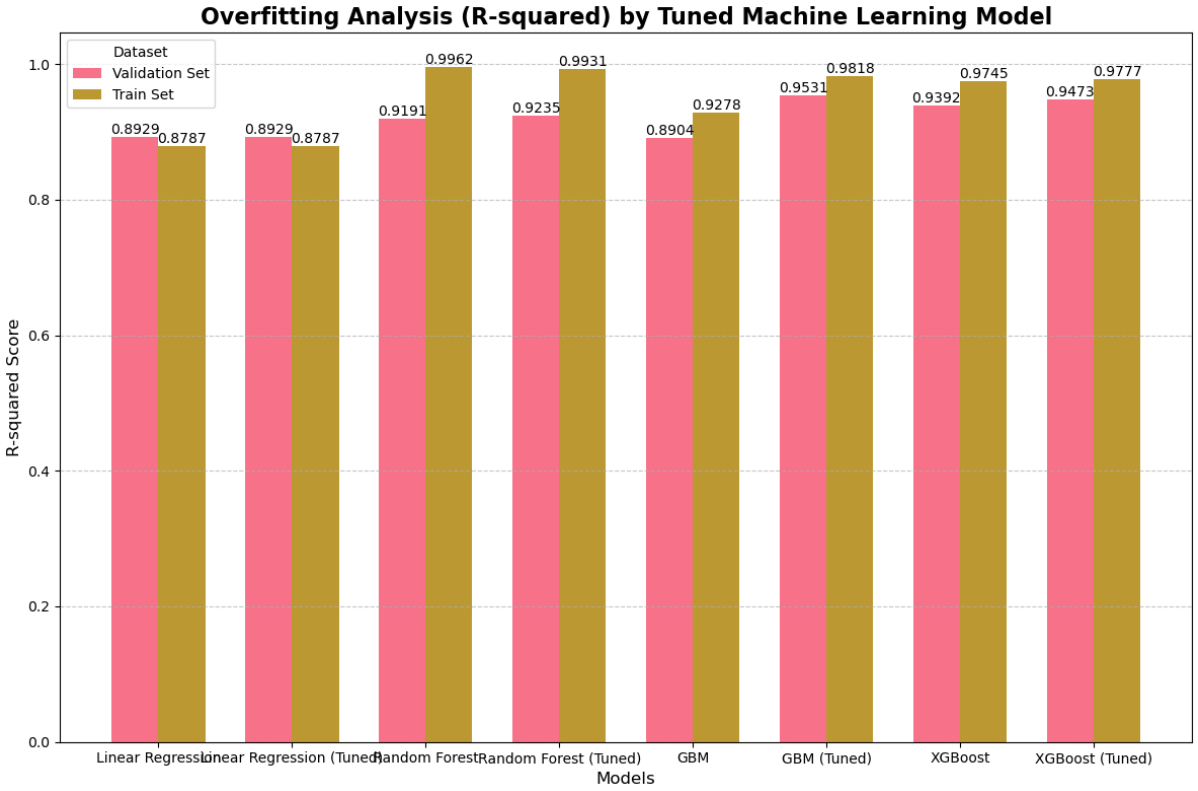


Figure 5.44 Overfitting Analysis after Hyperparameter Tuning (R^2)

The tuned Linear Regression shows identical performance to the baseline. This consistency indicates the hyperparameter tuning does not affect in the context. Random Forest after tuning has achieved 0.9931 on training and 0.9235 on validation, which slightly improved from the default parameters as it explained more in validation set and less overfitting in training set. GBM after hyperparameter tuning has a significant improvement in performance from default. It achieved 0.9818 in training set and 0.9531 in validation set. Despite that, the performance can also be interpreted as it shown certain “tendency to become overfitted” on the other hand. XGBoost meanwhile did not fluctuate much, which has achieved 0.9777 in training set and 0.9473 in validation set. The performance is almost identical to the default model.

5.6.8 Summary

Overall, models other than LR showed improvements in terms of the predictive power, but with the side effect that the training time significantly increased.

The improvement recorded on GBM's predictive power is the most aggressive compared to the others. This may be due to the found best parameters with Randomized Search, which aggressively increase the number of trees and the maximum depth. In results, the predictive power is the highest after tuning with the training speed significantly slowed down to the slowest correspondingly. Same speculation applied to the RF model after tuning, despite its condition is still better than GBM to a certain extent.

XGBoost, meanwhile, maintained the training time under a second, with a close enough predictive power to GBM after tuning. This made XGBoost the ideal candidate for the model to be selected as the final model.

Chapter 6

Final Model Evaluation and Dashboard

6.1 Final Evaluation

After the comparison, XGBoost with tuned parameters is selected as the final model. It is comparable to tuned GBM which is the absolute best one in terms of predictive power, with significantly better in efficiency which made its difference in predictive power negligible. The final model is trained with a new training set which has the original training set merged with validation set and evaluated with the test set that remained untouched before this stage. The following tables and charts illustrated the results.

6.1.1 Performance

Metrics	Value
RMSE	51999.24
MAE	40661.48
R^2	0.9302
Adjusted R^2	0.9301
Training Time (s)	0.7185
R^2/s	1.2946
Efficiency	12227.67

Table 6.1 Final model performance

XGBoost with its best parameters for this dataset is selected as the final model for the project. The results are obtained with the test set which isolated in all previous stages. The final model achieved $51999.24 of RMSE and $40661.48 of MAE. It is capable of explaining 93.02% of the data, and the adjusted R^2 is almost identical with the standard R^2 which are 0.9301 and 0.9302 respectively. The model training is done within 0.7185s, with 1.2946 of R^2/s and 12227.67 of efficiency score.

The model's performance slightly declined compared to its behaviour in the previous stage. This could be due to the market experienced fluctuation caused by policy changes or market sentiment in the time frame that the test set covered. Despite that, the performance is acceptable considering its efficiency.

6.1.2 Feature Importance

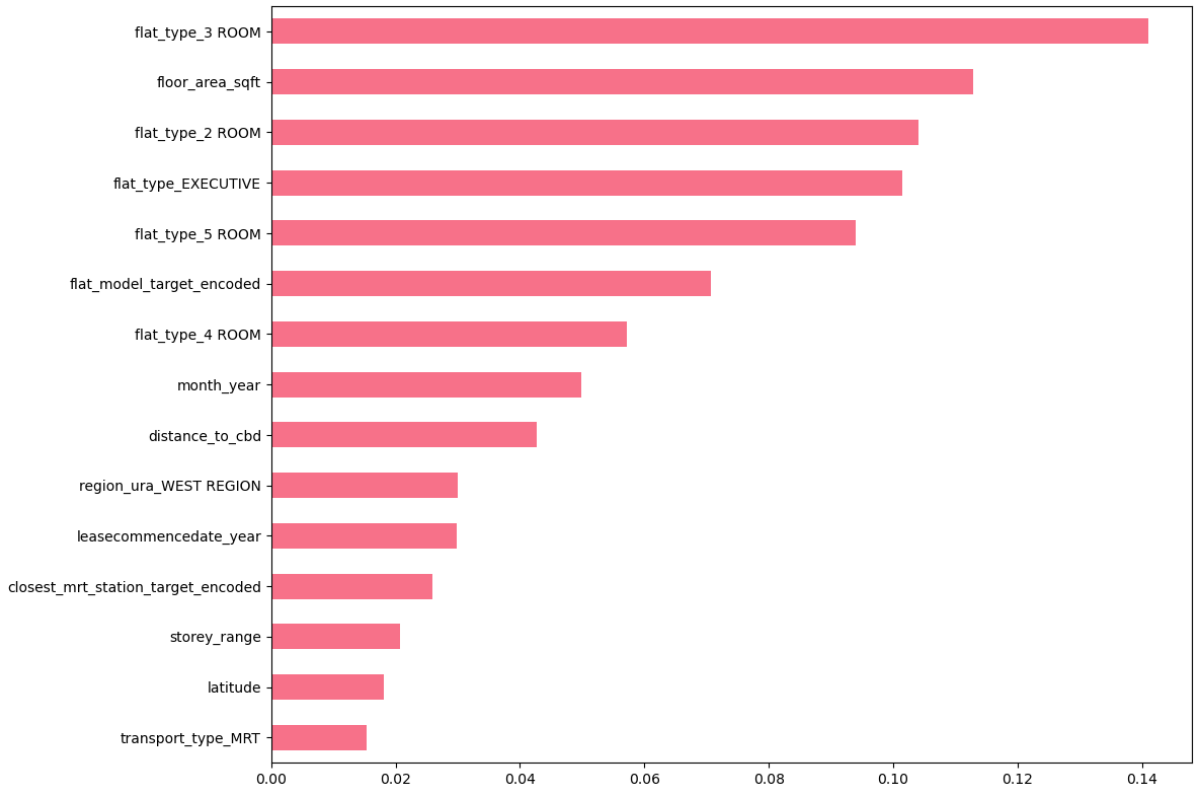


Figure 6.1 Feature Importance (final model)

The final model behaved slightly different from the default XGBoost in terms of treating the features, despite it still demonstrated a balanced distribution of feature importance: the number or type of rooms, in this context, has become one of the most important features in the housing price prediction, while floor area remains a critical factor. Temporal variables, distance to CBD, flat model, remaining lease, and storey range also contribute meaningfully, indicating that the model integrates both structural and categorical variables more effectively.

6.1.3 Residuals by Flat Type

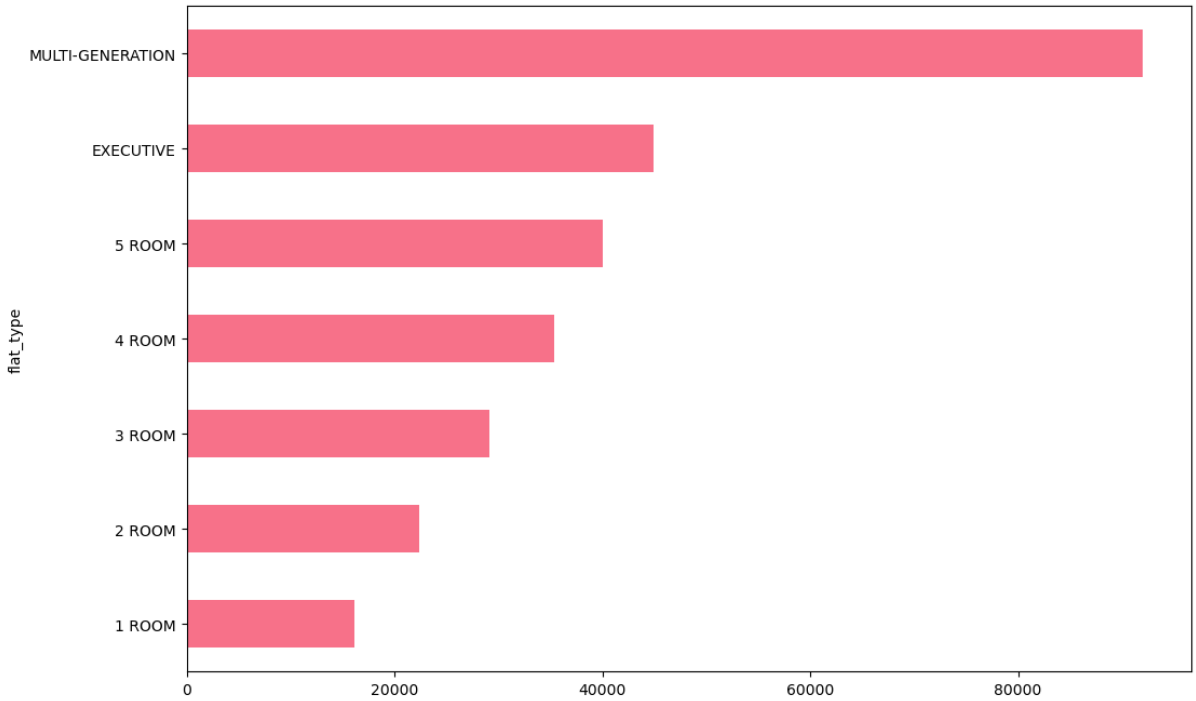


Figure 6.2 Residual by Flat Type

Plots are constructed to view the situation of absolute mean residual value in different conditions. Based on the plot, the difference between predicted price and actual price is ascending with the number of rooms, with “executive” type of flat on top of the other type of flat with different number of rooms and being the second highest. “Multi-generation” flat is significantly causing the model’s prediction diverged from the actual price. This indicated potential shortcomings of the model or the current pipeline to handle the correlation between flat type and the housing price, especially the categorical type such as “executive” and “multi-generation”.

6.1.4 Residuals by Town

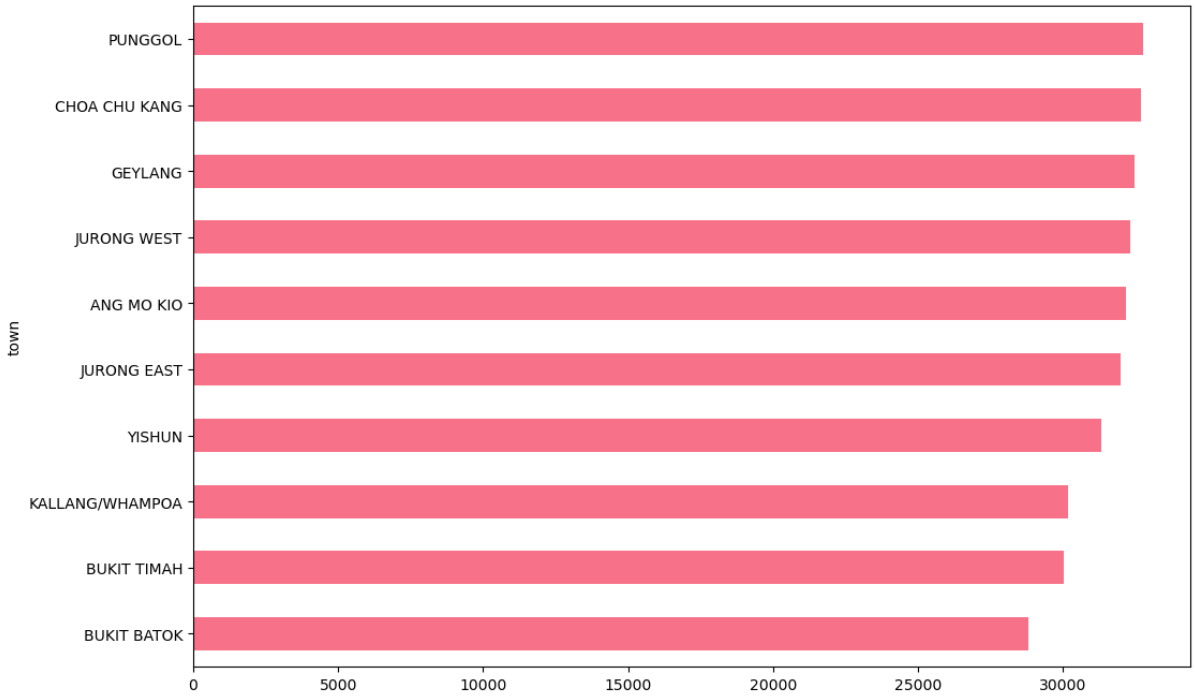


Figure 6.3 Residual by Town

Unlike the result in flat type, the residual by town is relatively consistent. The smallest and the largest residual by town has the difference less than $10k, and the residuals range from $25k to $35k. This indicated that the model or the current pipeline successfully captured the correlation between located towns and the actual resale price. On the other hand, it can also be interpreted as the influence of the town is relatively consistent with the resale price.

6.1.5 Residuals over Time

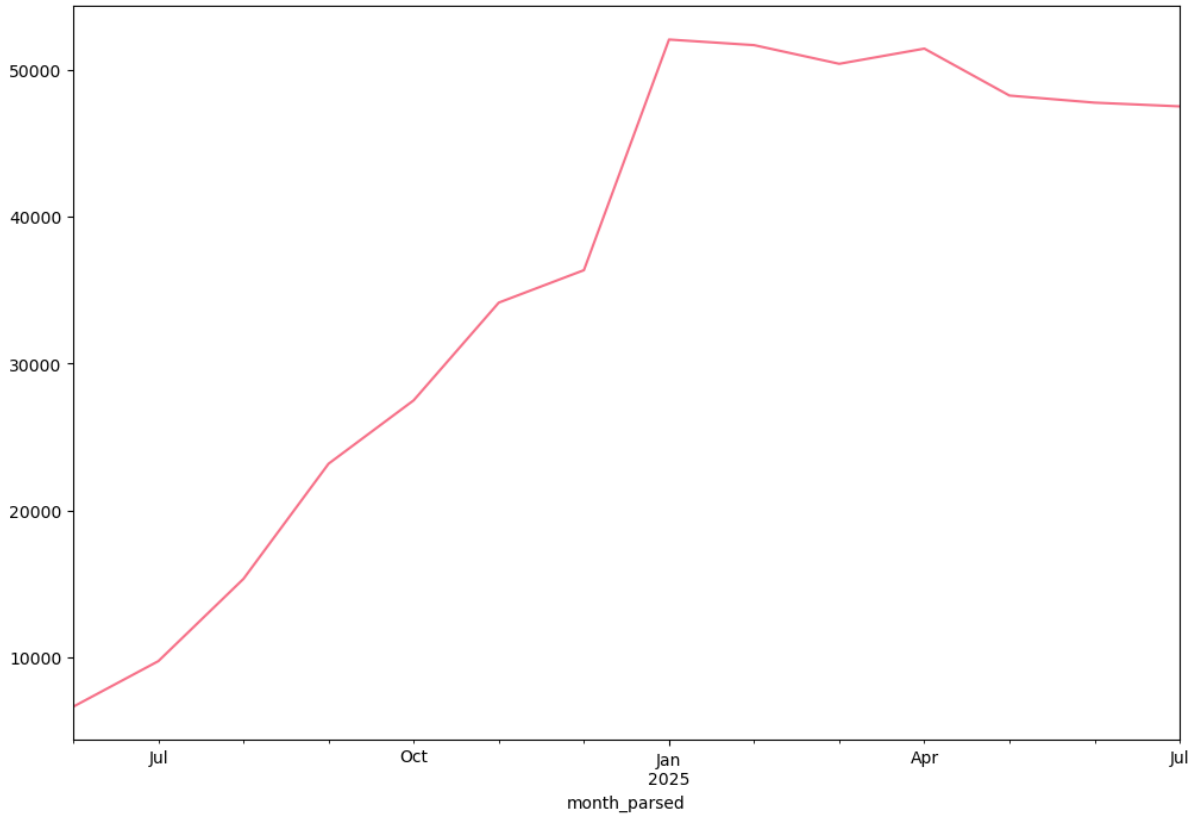


Figure 6.4 Residual over Time

The residual over time has shown a fluctuation. The line of residual started to climb up since Q3 2024 and spiked to the highest point in January 2025. Despite the possibility that this is related to the capabilities of the model or the current pipeline, it is also likely to be affected by the changes in the market or policy, or the market sentiment. For example, the launch of new flats in January 2025 could be the reason that caused the housing price to fluctuate non-linearly at the time [23].

6.2 Dashboard

After the construction of ML pipeline and best performed model for the dataset, the tested components have been carried over to build the dashboard using Streamlit. The application is available in Dockerfile, which allows users to build it locally with consistency. The following sections demonstrated the interface of the dashboard and its functionalities.

(Full source code: https://github.com/chmnxn/hdb-price-dashboard)

6.2.1 Upload and Overview

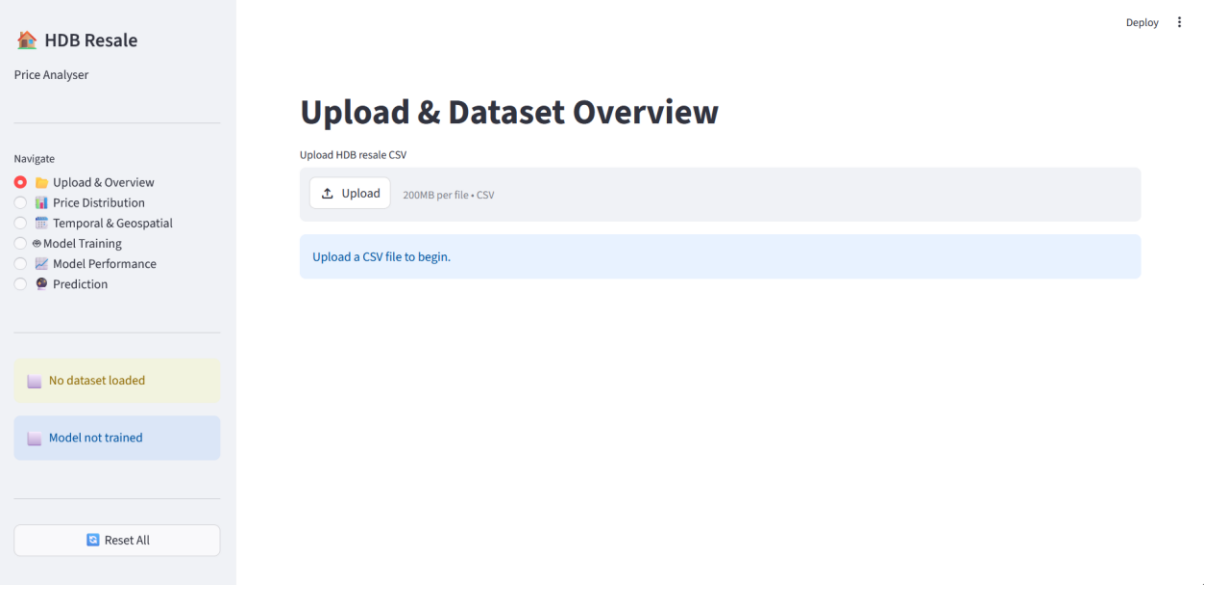


Figure 6.5 Dashboard screenshot #1

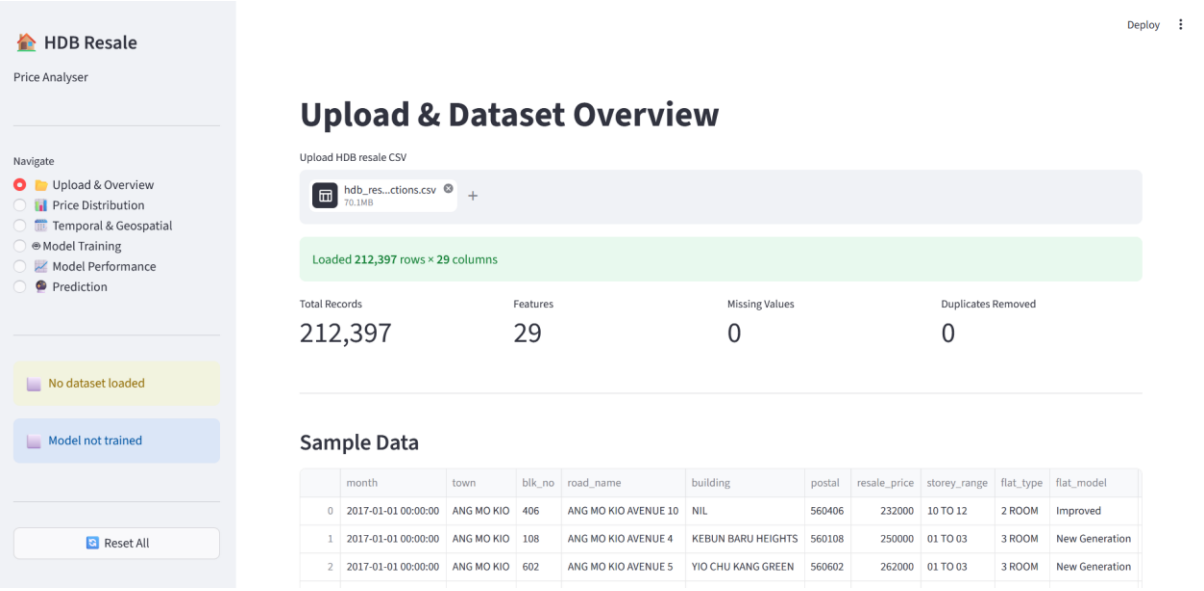


Figure 6.6 Dashboard screenshot #2

The landing page is for dataset upload and overview. As figure 6.5 shown, the page is empty by default, with a button for “upload” and file picker will pop out when user clicks it. After user uploaded a dataset that fulfil the requirements to have same features as the dataset used in experiment, the brief description of the dataset will appear on the page. A snapshot of the dataset is followed.

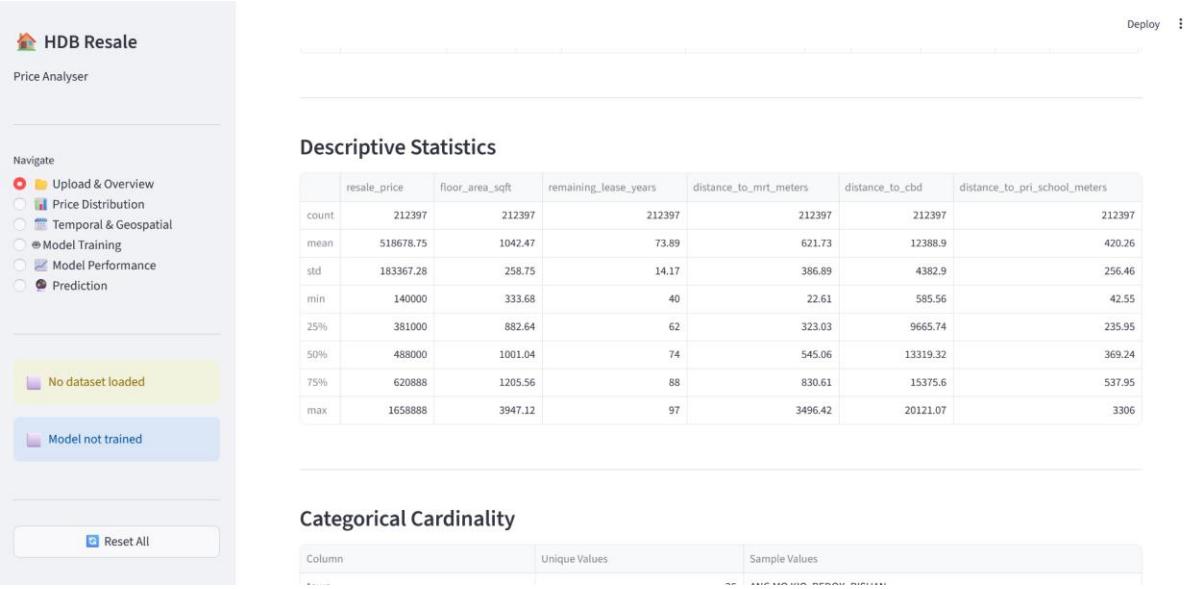


Figure 6.7 Dashboard screenshot #3

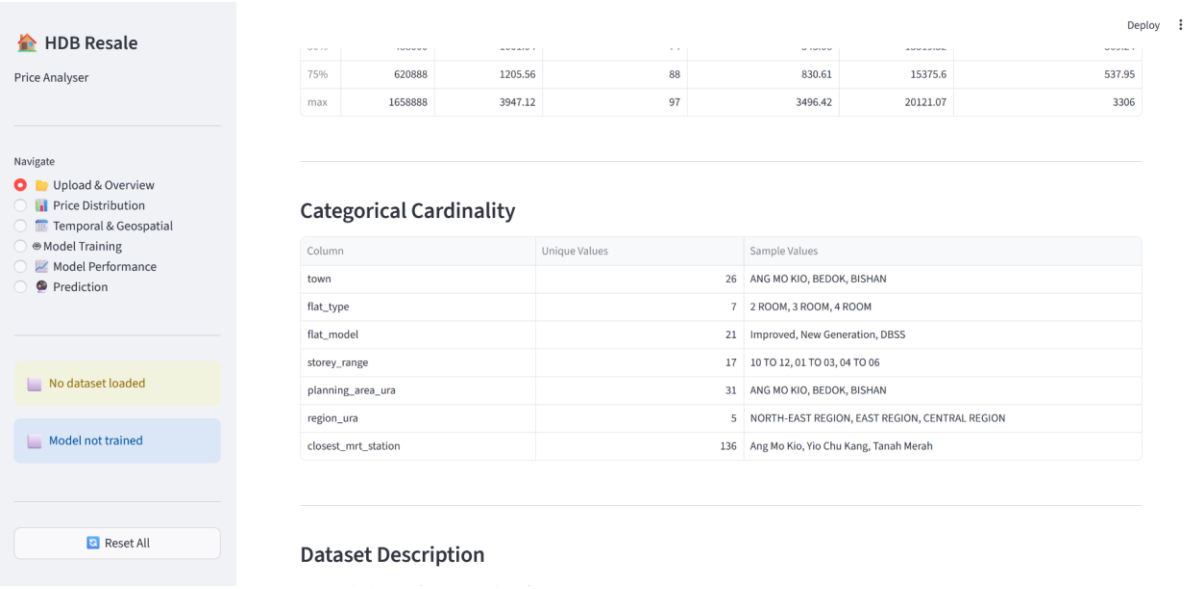


Figure 6.8 Dashboard screenshot #4

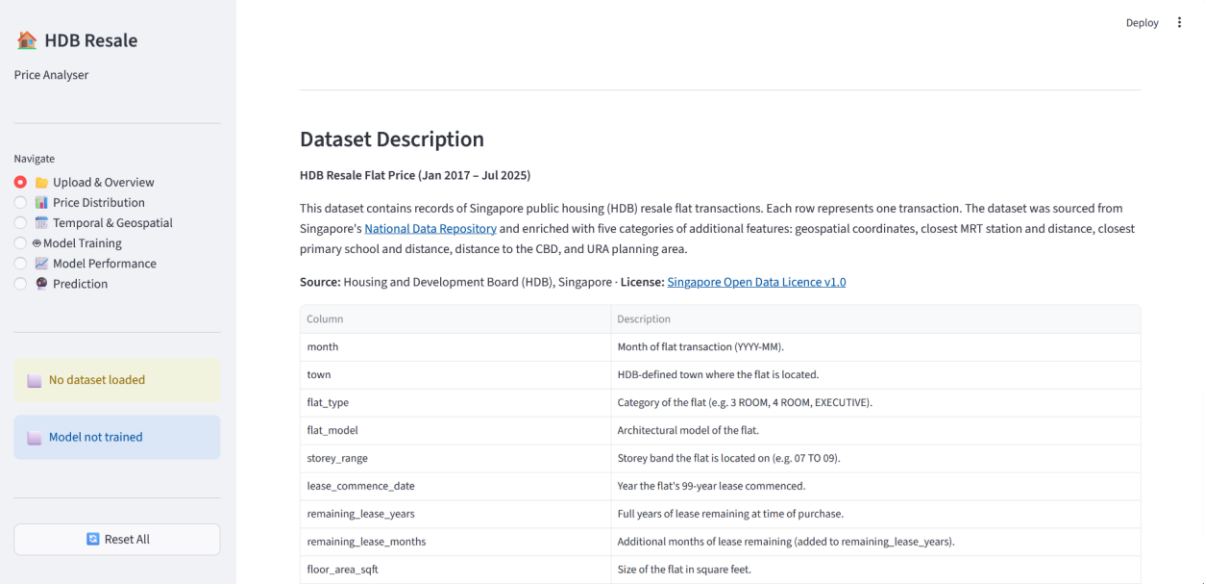


Figure 6.9 Dashboard screenshot #5

Besides of that, other descriptions such as statistics of the data (e.g. mean, median, etc.), cardinality of categorical features, and metadata of the dataset will also display on the page. Overall, this page tends to provide a clear landscape about the dataset, which helps to enable user to make informed decisions in the later sections.

6.2.2 Price Distribution

Figure 6.10 Dashboard screenshot #6

Figure 6.11 Dashboard screenshot #7

Figure 6.12 Dashboard screenshot #8

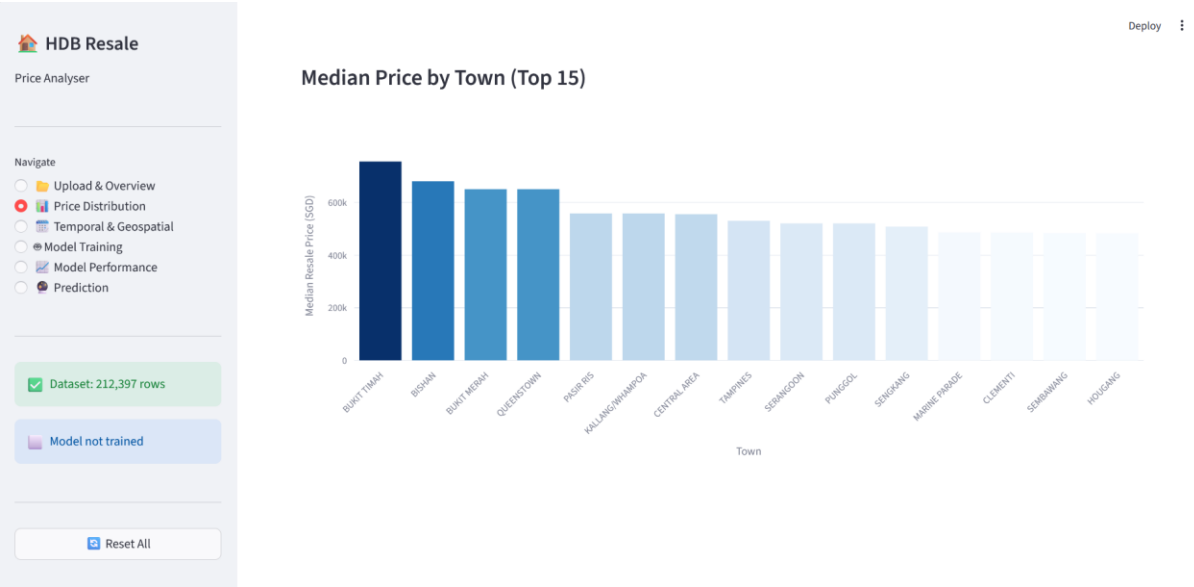


Figure 6.13 Dashboard screenshot #9

The page provides a statistical overview of the housing price and its relationship with some key categorical features. The first plot is a side-by-side comparison of the resale price distribution at the original scale and after log transformation, which described the reason for the log transformation applied during model training as the original distribution has shown to be right skewed with a long tail of high-value transactions, while the log-transformed distribution approximates normality. On the other hand, two box plot which one described the resale price by flat type and another by storey range. It also contains a bar chart of the fifteen towns with the highest median resale price, providing a geographic comparison of price levels across different towns.

6.2.3 Temporal and Geospatial Analysis

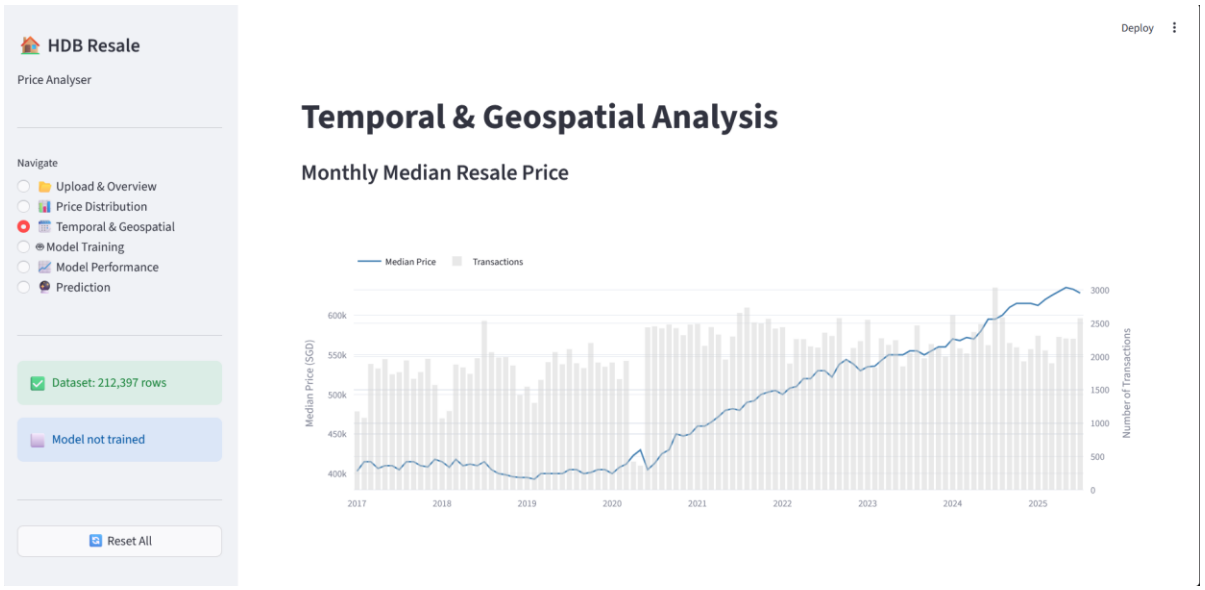


Figure 6.14 Dashboard screenshot #10

Figure 6.15 Dashboard screenshot #11

Figure 6.16 Dashboard screenshot #12

The page examines the dataset from the perspectives of temporal and geospatial. It included a dual-axis time-series chart showing monthly median resale price on the primary axis and monthly transaction volume as a bar chart on the secondary axis. Besides, there is also a geospatial scatter plot mapping individual transactions by latitude and longitude, coloured by resale price on a red-yellow-green gradient. A sample size slider allows the user to control the number of points rendered, balancing visual density against browser performance. Hovering over individual points reveals the town, flat type, and exact resale price. A horizontal bar chart is also prepared to visualize the Pearson correlation between each numeric feature and the target variable.

6.2.4 Model Training

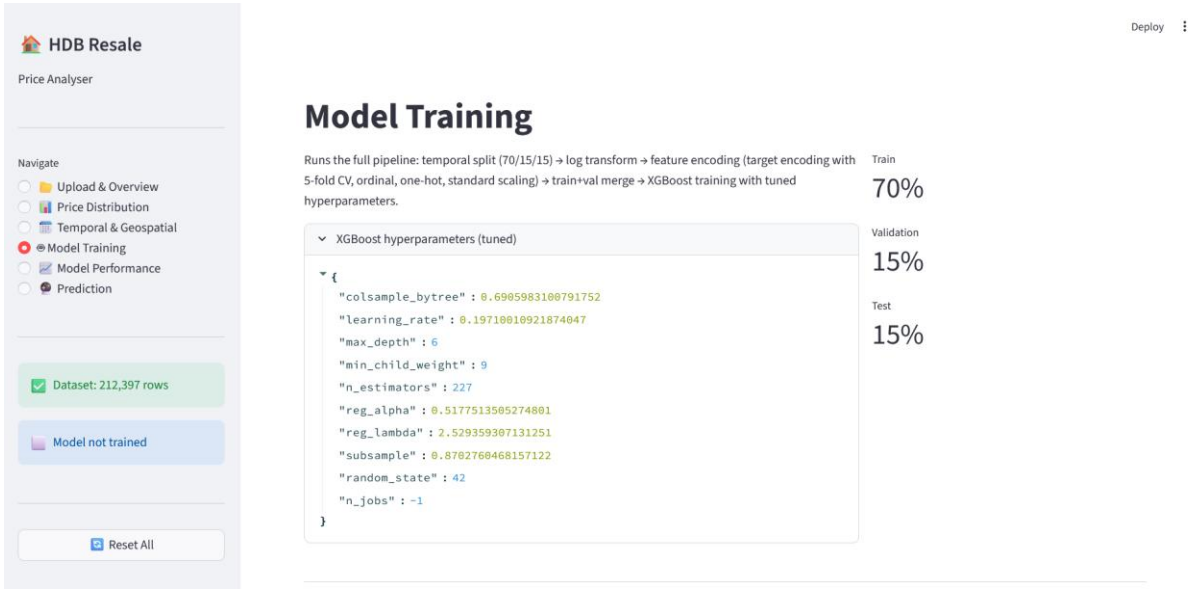


Figure 6.17 Dashboard screenshot #13

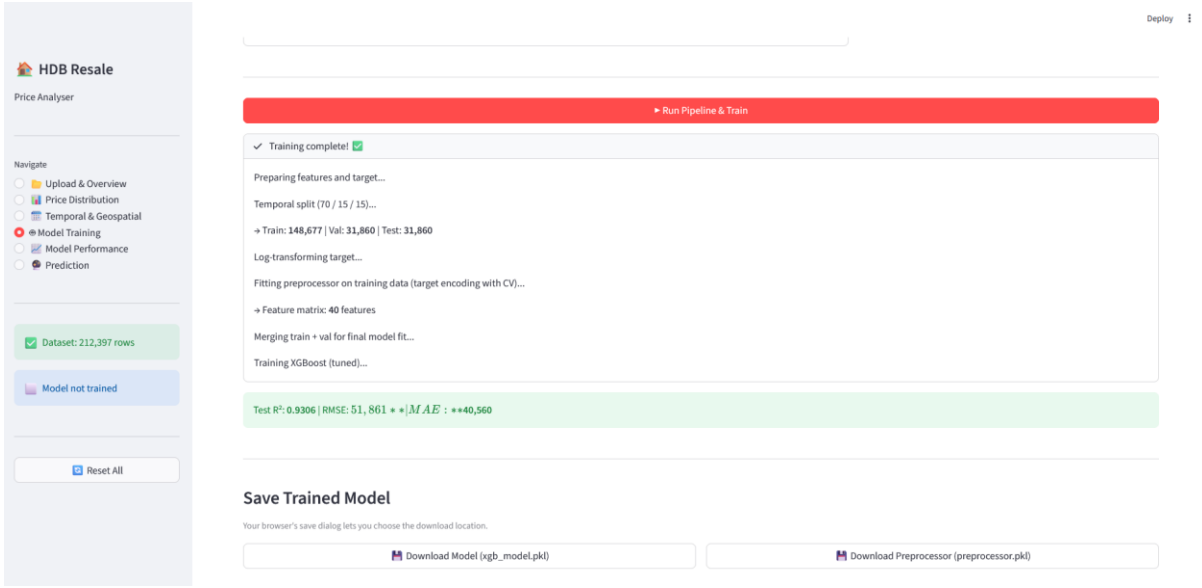


Figure 6.18 Dashboard screenshot #14

The page contains the full machine learning pipeline as an interactive process. The page displays the fixed train, validation, and test split ratio of 70:15:15 and provides an expandable section showing the complete set of XGBoost optimal hyperparameters selected during the randomised search tuning phase. When the user clicks the Run Pipeline & Train button, the pipeline executes sequentially with a live progress log rendered, reporting the output of each stage. When completed, a success banner displays the test set R^2, RMSE, and MAE. All

pipeline outputs, including the trained model, fitted preprocessor, raw and processed test sets, predictions, and evaluation metrics, are stored in Streamlit's session state to persist across page navigation without triggering retraining. Once training is complete, two download buttons become available: one for the trained model and one for the fitted preprocessor, which either button will trigger the browser's save dialog.

6.2.5 Model Performance

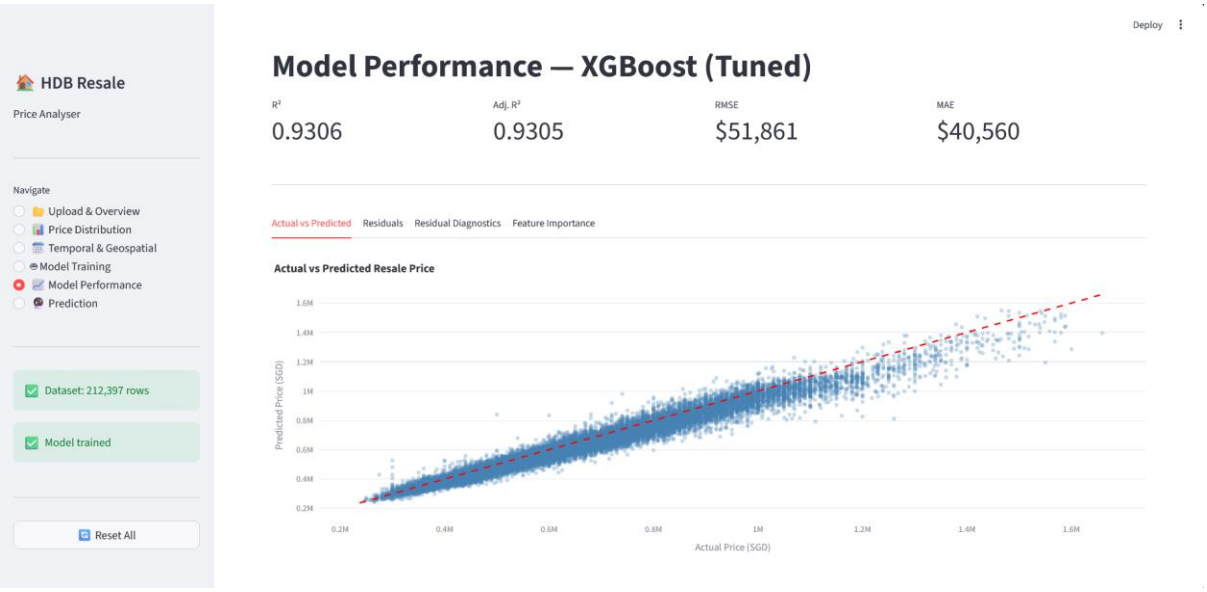


Figure 6.19 Dashboard screenshot #15

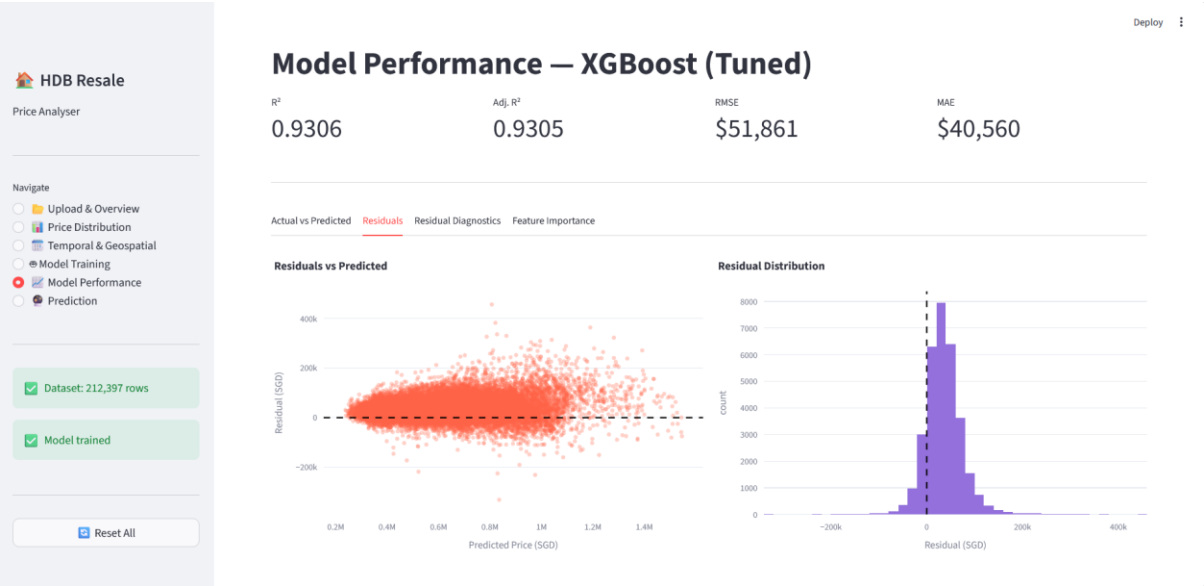


Figure 6.20 Dashboard screenshot #16

The page displays a comprehensive diagnostic evaluation of the trained model. At the top of the page, the four primary evaluation metrics — R^2, Adjusted R^2, RMSE, and MAE — are displayed as summary cards. The remainder of the page is organised into four tabs. The first tab, Actual vs Predicted, predicted resale prices against actual transaction prices as a scatter chart. The second tab, Residuals, presents a scatter plot of residuals against predicted values and a histogram of the residual.

Based on the current captured plots, the model is likely tended to predict conservatively or pessimistically.

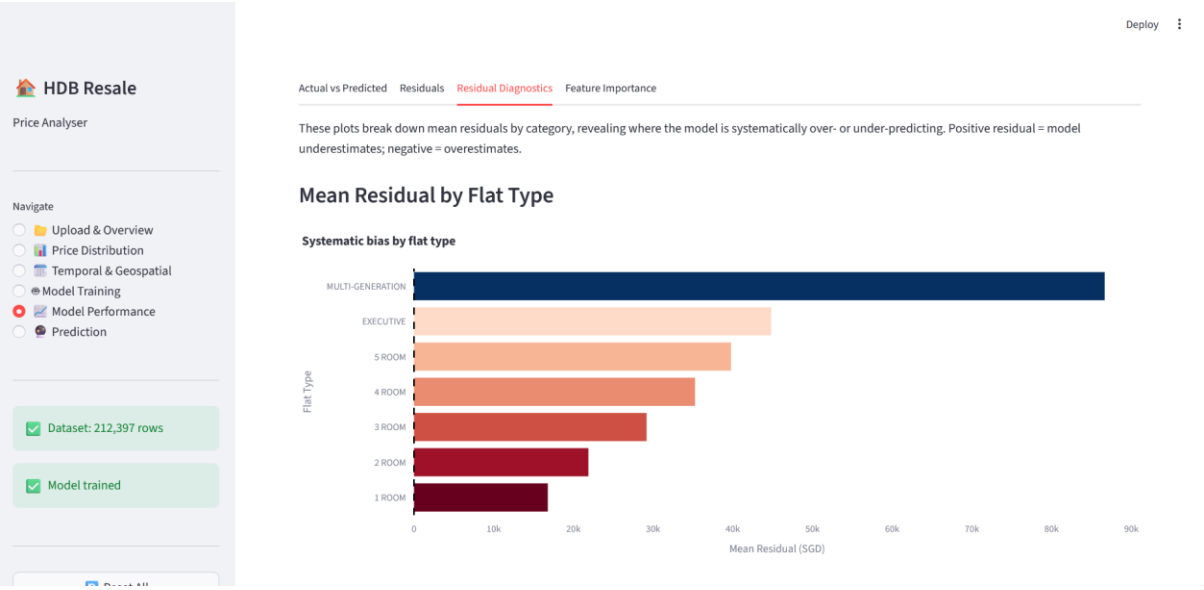


Figure 6.21 Dashboard screenshot #17

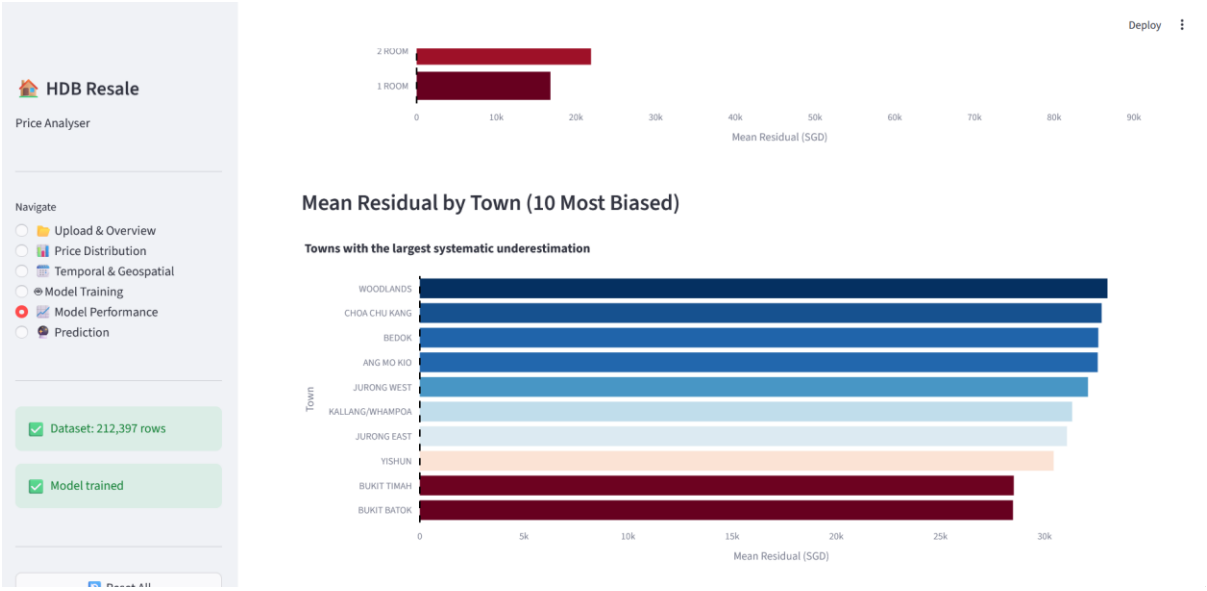


Figure 6.22 Dashboard screenshot #18

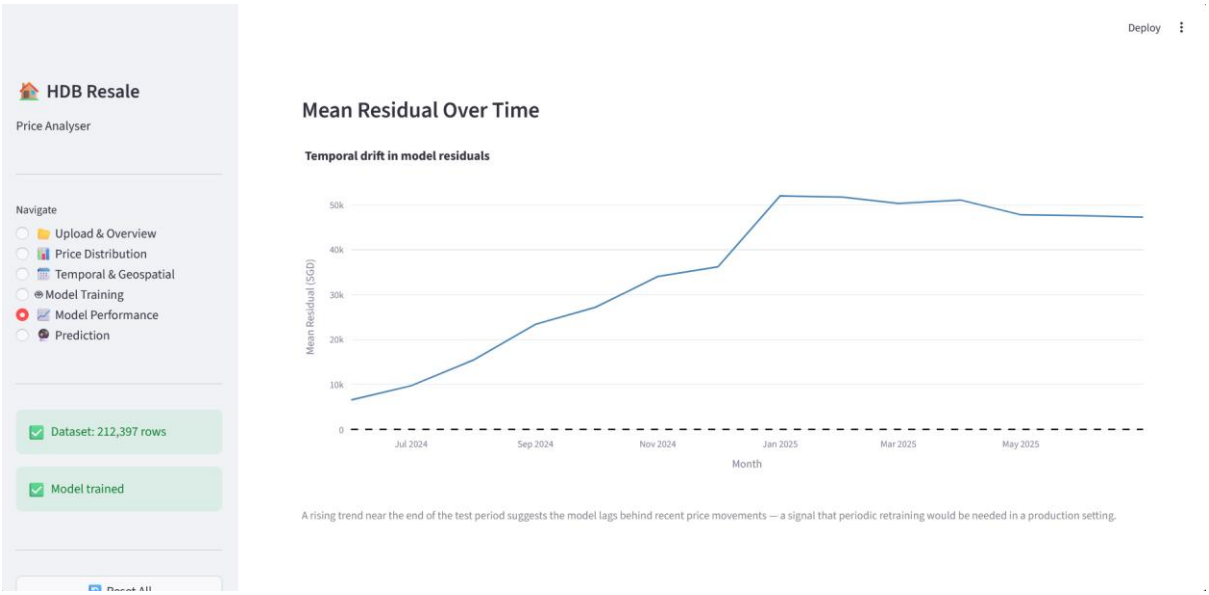


Figure 6.23 Dashboard screenshot #19

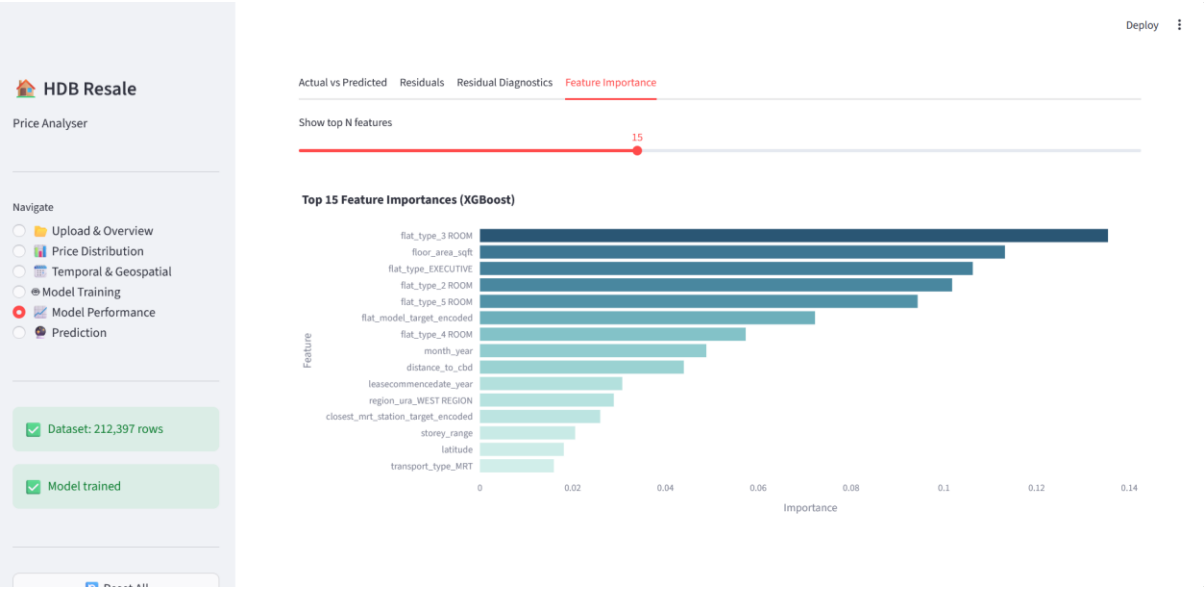


Figure 6.24 Dashboard screenshot #20

The third tab, Residual Diagnostics, contains three plots adapted from the final evaluation in experiment. Mean residual by flat type reveals whether the model inaccurately estimates particular flat categories. Mean residual by town, restricted to the ten most biased towns, identifies geographic segments where the model's assumptions break down. Mean residual over time shows whether prediction error drifts as the test period progresses. The fourth tab, Feature Importance, renders a horizontal bar chart of model's feature importances with an adjustable slider allowing the user to select the number of features to display.

6.2.6 Prediction

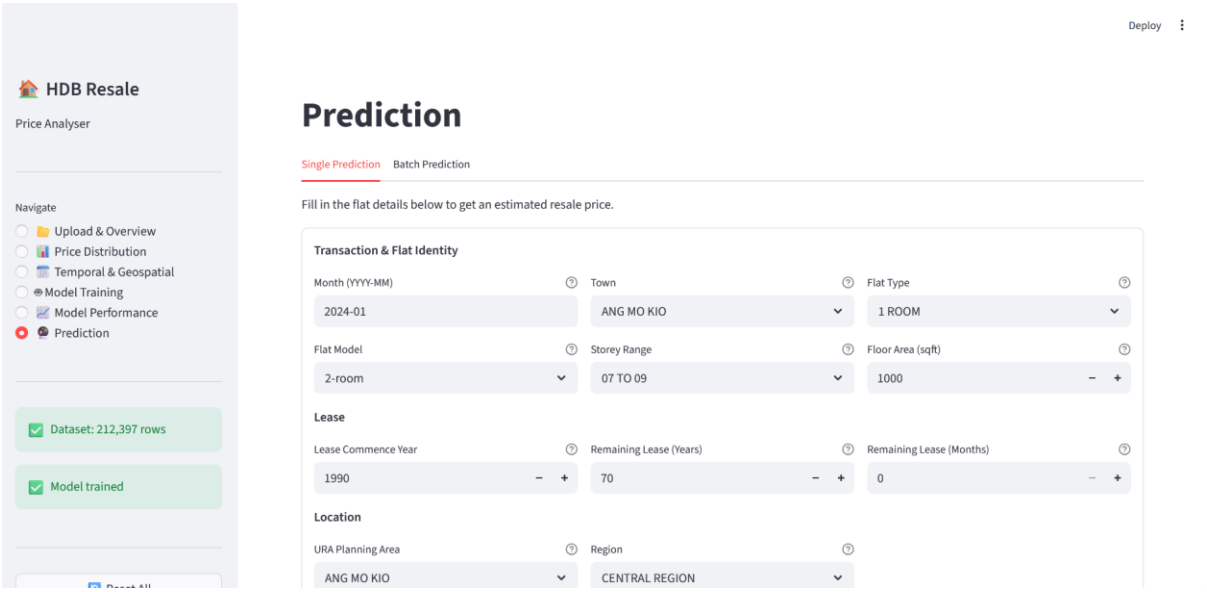


Figure 6.25 Dashboard screenshot #21

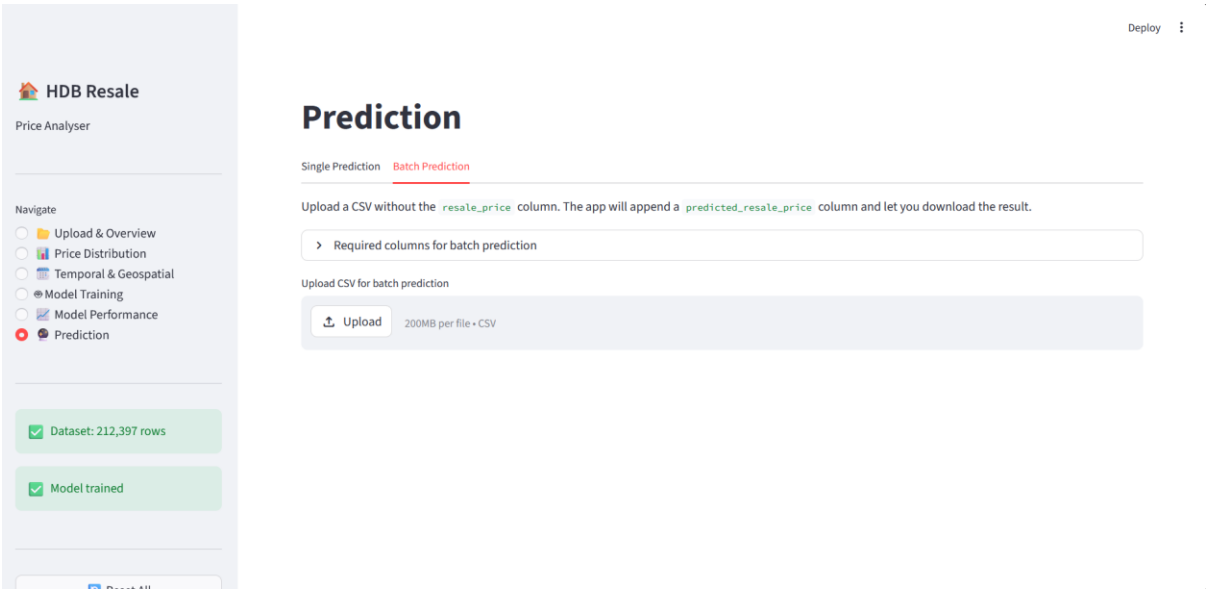


Figure 6.26 Dashboard screenshot #22

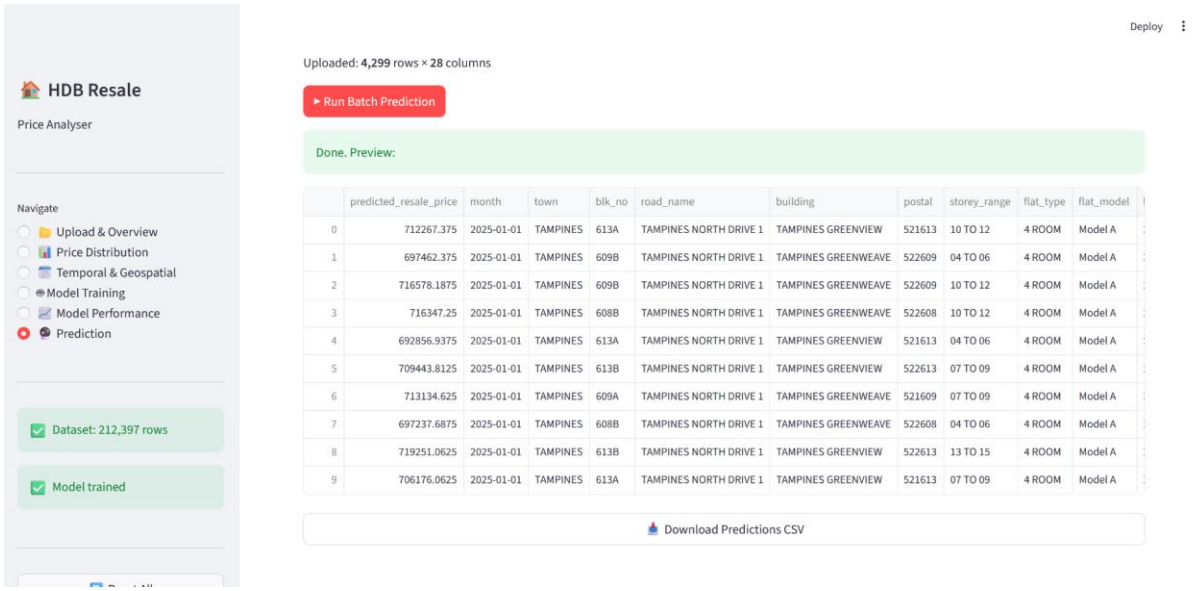


Figure 6.27 Dashboard screenshot #23

The Prediction page provides single and batch prediction using the fitted preprocessor and the trained model. The Single Prediction tab is an input form divided into four sections that cover all features captured by the model. Dropdown fields are automatically populated with the unique values captured in the dataset, falling back to hardcoded lists of known valid categories otherwise. Every field includes a "hint" which is accessible via the question mark icon to display the corresponding column description. The result is displayed alongside a confidence range to provide user with a sense of the estimate's expected precision, which helps them to make informed decisions. The Batch Prediction tab accepts a CSV file containing multiple rows of housing features without a resale price column. The system validates the uploaded file for column completeness and rejects files that contain a resale price column. User clicks Run Batch Prediction to generate predictions for all rows simultaneously. A preview of the results is displayed in the interface, and a download button allows the user to export it as a CSV file.

6.3 System Testing

6.3.1 Unit Test

Automated unit tests were created using the pytest and target all functions in modules.py. The tests are executable with the command "pytest test_modules.py -v" with no dependency on external resources.

The following test classes and their covered scope are as described below:

- **TestLoadAndClean**: verifies that the function module correctly accepts both file path strings and BytesIO file-like objects produced by file uploader, drops the unnecessary column, and removes duplicate rows from the loaded DataFrame.
- **TestPrepareXY**: verifies that function module separates the target column from the feature set, drops all selected columns to drop during the experiment pipeline, and does not raise an error when any of those columns are unavailable.
- **TestTemporalSplit**: verifies that the function module returns exactly six output arrays, that their sizes sum to the total input size, that the default 70/15/15 ratio is applied correctly, which the split is chronological, and that X and y indices remain aligned after splitting.
- **TestLogTransform**: verifies that function module produces the correct log values for known inputs and that the transformation is correctly invertible by np.exp().
- **TestEvaluate**: verifies the metric computation functions.
- **TestPredict**: verifies that the function correctly inverts the log transformation applied during training, that the np.clip(0) guard prevents any negative price predictions regardless of model output, and that the output shape matches the number of input rows.
- **TestMergeTrainVal**: verifies that the function concatenates training and validation arrays to the correct combined shape, and that the train portion appears before the validation portion in the merged output.
- **TestValidateTrainColumns**, **TestValidatePredictColumns**, **TestValidateBatchColumns**: verify the input validation layer. Tests confirm that a well-formed DataFrame returns no errors, that missing columns are reported by name, that out-of-range numeric values are detected, columns with mixed string/integer types do not cause a crash, and that batch prediction function rejects any file containing a resale_price column.

Results:

```
============================================================ test session starts =============================================================
platform linux -- Python 3.12.7, pytest-9.0.3, pluggy-1.6.0 -- /home/chmnxn/anaconda3/envs/mlenv/bin/python3.12
cachedir: .pytest_cache
rootdir: /home/chmnxn/streamlit-app
plugins: anyio-4.7.0
collected 36 items

test_modules.py::TestLoadAndClean::test_accepts_filepath PASSED                                                                      [  2%]
test_modules.py::TestLoadAndClean::test_accepts_bytesio PASSED                                                                       [  5%]
test_modules.py::TestLoadAndClean::test_drops_unnamed_index_column PASSED                                                            [  8%]
test_modules.py::TestLoadAndClean::test_removes_duplicate_rows PASSED                                                                [ 11%]
test_modules.py::TestPrepareXY::test_separates_target PASSED                                                                         [ 13%]
test_modules.py::TestPrepareXY::test_drops_leakage_columns PASSED                                                                    [ 16%]
test_modules.py::TestPrepareXY::test_missing_drop_cols_are_ignored PASSED                                                            [ 19%]
test_modules.py::TestTemporalSplit::test_returns_six_parts PASSED                                                                    [ 22%]
test_modules.py::TestTemporalSplit::test_sizes_sum_to_total PASSED                                                                   [ 25%]
test_modules.py::TestTemporalSplit::test_default_ratio_70_15_15 PASSED                                                               [ 27%]
test_modules.py::TestTemporalSplit::test_split_is_chronological_not_random PASSED                                                    [ 30%]
test_modules.py::TestTemporalSplit::test_x_and_y_indices_are_aligned PASSED                                                          [ 33%]
test_modules.py::TestLogTransform::test_single_array PASSED                                                                          [ 36%]
test_modules.py::TestLogTransform::test_multiple_arrays_returns_tuple PASSED                                                         [ 38%]
test_modules.py::TestLogTransform::test_inverse_is_exp PASSED                                                                        [ 41%]
test_modules.py::TestEvaluate::test_perfect_predictions PASSED                                                                       [ 44%]
test_modules.py::TestEvaluate::test_returns_all_keys PASSED                                                                          [ 47%]
test_modules.py::TestEvaluate::test_rmse_is_always_non_negative PASSED                                                               [ 50%]
test_modules.py::TestEvaluate::test_adj_r2_penalises_more_features PASSED                                                            [ 52%]
test_modules.py::TestEvaluate::test_known_mae_value PASSED                                                                           [ 55%]
test_modules.py::TestPredict::test_inverse_log_transform PASSED                                                                      [ 58%]
test_modules.py::TestPredict::test_no_negative_predictions PASSED                                                                    [ 61%]
test_modules.py::TestPredict::test_output_shape_matches_input PASSED                                                                 [ 63%]
test_modules.py::TestMergeTrainVal::test_concatenates_correctly PASSED                                                               [ 66%]
test_modules.py::TestMergeTrainVal::test_train_comes_before_val PASSED                                                               [ 69%]
test_modules.py::TestValidateTrainColumns::test_valid_df_returns_no_errors PASSED                                                    [ 72%]
test_modules.py::TestValidateTrainColumns::test_missing_column_detected PASSED                                                       [ 75%]
test_modules.py::TestValidateTrainColumns::test_missing_target_detected PASSED                                                       [ 77%]
test_modules.py::TestValidateTrainColumns::test_out_of_range_numeric_detected PASSED                                                 [ 80%]
test_modules.py::TestValidateTrainColumns::test_string_numeric_column_handled PASSED                                                 [ 83%]
test_modules.py::TestValidatePredictColumns::test_valid_df_returns_no_errors PASSED                                                  [ 86%]
test_modules.py::TestValidatePredictColumns::test_missing_column_detected PASSED                                                     [ 88%]
test_modules.py::TestValidatePredictColumns::test_out_of_range_distance_detected PASSED                                              [ 91%]
test_modules.py::TestValidateBatchColumns::test_rejects_file_with_resale_price PASSED                                                [ 94%]
test_modules.py::TestValidateBatchColumns::test_accepts_file_without_resale_price PASSED                                             [ 97%]
test_modules.py::TestValidateBatchColumns::test_still_catches_missing_columns PASSED                                                 [100%]

============================================================ 36 passed in 2.28s ==============================================================
```

Figure 6.28 Automated Unit Tests Passed

The system passed all 36 automated tests within 3 seconds.

6.3.2 Functional Testing

The following test cases cover the Streamlit dashboard's user-facing functionality across all six pages. Each test was executed manually with the application running locally, due to the automated test method is practically not able to correctly simulate and evaluate the user experience.

ID	Description	Expected	Result
Page: Upload and Overview			
TC-01	Upload a valid CSV file.	Dataset loads successfully, KPI metrics (row count, feature count, missing values, duplicates removed) update correctly, sample data table and descriptive statistics render.	Pass

TC-02	Upload a CSV with a missing required column (e.g. town removed).	An error message naming the missing column is displayed and the page halts before rendering any statistics.	Pass
TC-03	Upload a CSV where a numeric column (latitude) contains a value outside the valid range.	An error message identifying the column, the number of offending rows, and example values are displayed.	Pass
TC-04	Upload a CSV where a numeric column (lease_commence_date) is stored as string dtype due to mixed values.	Validation completes without a TypeError; any out-of-range values are still reported correctly.	Pass
TC-05	Navigate to any other page without uploading a dataset first.	A warning message is displayed, and the page content does not render.	Pass
Page: Price Distribution			
TC-06	With a valid dataset loaded, navigate to the Price Distribution page.	Original scale and log scale histograms, flat type box plot, storey range box plot, and top towns bar chart all render without error.	Pass
TC-07	Verify storey range box plot ordering.	Storey ranges appear in ascending order (01 TO 03 first, highest band last) rather than alphabetical order.	Pass
Page: Temporal and Geospatial			
TC-08	Verify the monthly trend dual-axis chart.	Median price line plotted on the left axis, transaction count bars on the right axis, both visible simultaneously.	Pass
TC-09	Adjust the sample size slider for the geospatial scatter plot.	The map re-renders with the updated number of points without error.	Pass
TC-10	Verify correlation bar chart direction.	floor_area_sqft and remaining_lease_years show positive correlation with resale_price; distance features show negative correlation.	Pass
Page: Model Training			

TC-11	Click "Run Pipeline & Train" with a valid dataset loaded.	the st.status progress log displays each pipeline stage sequentially, training completes, and a success banner showing Test R^2, RMSE, and MAE appears.	Pass
TC-12	Verify that the pipeline does not retrain when navigating away from and back to the Model Training page.	Session state preserves the trained model; the training button must be explicitly clicked to retrain.	Pass
TC-13	Click "Download Model" after training.	Browser save dialog appears; a valid .pkl file is downloaded that can be loaded with joblib.load().	Pass
TC-14	Click "Download Preprocessor" after training.	Browser save dialog appears; a valid .pkl file is downloaded.	Pass
TC-15	Attempt to download before training.	Download buttons are not shown; an info message prompts the user to train first.	Pass
TC-16	Click "Reset All" in the sidebar after training.	All session state is cleared, sidebar status indicators revert to "not trained" and "no dataset", and the training page reverts to its initial state.	Pass
Page: Model Performance			
TC-17	Navigate to Model Performance before training.	An info message prompts the user to train the model first; no charts render.	Pass
TC-18	Navigate to Model Performance after training.	All four metrics (R^2, Adjusted R^2, RMSE, MAE) display correctly, and all four tabs (Actual vs Predicted, Residuals, Residual Diagnostics, Feature Importance) render without error.	Pass
TC-19	Verify the Actual vs Predicted scatter chart.	A red dashed diagonal line representing perfect prediction is overlaid on the scatter plot.	Pass

TC-20	Verify the Residual Diagnostics tab.	Three charts render (mean residual by flat type, mean residual by the 10 most biased towns, and mean residual over time), each with a zero-reference line.	Pass
TC-21	Adjust the "Show top N features" slider in the Feature Importance tab.	The bar chart updates to show the selected number of features in descending importance order.	Pass
Page: Prediction – Single Prediction			
TC-22	Submit the prediction form with all fields populated with valid values.	An estimated resale price is displayed in SGD, accompanied by a confidence range derived from the model MAE.	Pass
TC-23	Verify tooltip availability.	Hovering the "?" icon next to any form field displays the corresponding column description from the dataset metadata.	Pass
TC-24	Navigate to Prediction before training.	A warning message is displayed and the page halts before rendering the form.	Pass
Page: Prediction – Batch Prediction			
TC-25	Upload a valid batch CSV (all required columns present, no resale_price column).	Row count is displayed, "Run Batch Prediction" button appears.	Pass
TC-26	Click "Run Batch Prediction".	Predictions complete, a preview DataFrame with predicted_resale_price as the first column is shown, and a "Download Predictions CSV" button appears.	Pass
TC-27	Download the predictions CSV.	Browser save dialog appears; the downloaded file contains the original columns plus a predicted_resale_price column.	Pass

TC-28	Upload a CSV that contains a resale_price column (i.e. a training file uploaded by mistake).	An error message warns that the file appears to be a training dataset and the prediction does not proceed.	Pass
TC-29	Upload a batch CSV with a missing required column.	An error message naming the missing column is displayed and the prediction does not proceed.	Pass

Table 6.2 Functionalities Tests & Results

6.4 Challenges

Several challenges are encountered in the project. First, the dataset selected for this research is considerably new which is first uploaded in 2024, and there is currently no other previous work that uses the exact same data as this. Despite synthetic data could be a potential approach for benchmarking, this step is not taken in this particular research, as the quality and complexity of synthetic data is just not as significant as real-world data. On the other hand, this research will focus on internal model comparisons, which include comparison with baseline model (linear regression) and comparisons of different hyperparameter for models.

On the other hand, there are certain constraints from the Streamlit framework itself from the development point of view, especially in terms of its testability. Many feature tests need to be done manually due to the automated test can neither render the plot nor simulate the file upload process, which both are important features in the dashboard. Also, the technical stack chosen for the dashboard is to save the training outcomes in the form of session state; despite giving flexibility, it created some challenges for automated testing at the same time. Eventually the manual way is the most efficient way to evaluate the functionalities in this scenario despite it is time consuming.

Even Randomized Search is utilized in the project, it is still resource-consuming in terms of not just computational latency, but also computational resources, given the size and dimensionality of the dataset. Under observation, the experiment has crashed multiple times due to OOM issue, which eventually forced a reconfiguration of the computing environment to reserve as much RAM as it can during the execution of experiment. Around 3 hours is required to execute the full experiment notebook with the hardware condition specified in chapter 4.

6.5 Objective Evaluation

6.5.1 Exploratory data analysis (EDA) and data preprocessing on selected dataset

The objective was accomplished in the first of third section in the experiment. Several charts and plots were carried out to visualize the data distribution and characteristics of the dataset. Some of the constructed plots and charts have also been carried over to the latter dashboard construction to enhance the readability of the dashboard. A general data preprocessing pipeline is also constructed to process the selected dataset to make it "digestible" for ML models.

6.5.2 Baseline model comparison and evaluation

The comparison was done in the experiment. 6 selected models have been placed in the same starting line (dataset went through the general preprocessor) and have been compared from the aspects of both performance and efficiency. The worst performed model and overfitted model have been filtered after this stage, while the remaining have been carried over to the next stage. Therefore, this particular objective has been accomplished.

6.5.3 Hyperparameter tuning and final model selection

This has been accomplished in the experiment. The models have been tuned by Randomized Search can compared with each other, and also with the baseline and default models. XGBoost with its best parameters, was the best all-rounder among all models based on the results. It has been selected as the final model to be used for dashboard construction.

6.5.4 Dashboard construction

This has been accomplished by using Streamlit, which allows developers to easily migrate the written code in experiment to become an actual application. The dashboard inherited the experiment outcomes and added a layer of user interface, which enabled end users to easily deploy and utilize it in different way such as data understanding visually, automated model training, housing price prediction, etc.

CHAPTER 7

Conclusion and Recommendation

7.1 Conclusion

The project successfully investigated and evaluated the application of ML and data visualization in real estate market. EDA, data preprocessing, model comparison in default parameters and after hyperparameter-tuning, and the final construction of dashboard with automated ML pipeline have all contributed to the success of the project.

This research investigated the application of machine learning and data visualization, comparing 6 ML models with different architectures (LR, SVM, DT, RF, GBM, XGBoost) from the aspects of predictive power and also efficiency. XGBoost is the selected all-rounder that balanced both sides (RMSE of $51999.24, MAE of $40661.48, R2 of 0.9302, training time of 0.7185s, and efficiency score of 12227.67, with test set). SVM, meanwhile, has proved to be the worst performer among 6, indicated it is not capable enough to capture the complexity of the market and the feature space, which is not suitable to be used for housing price prediction. Other models are either moderately performed or lack efficiency despite their strong predictive power, GBM in this case for example, which consumed nearly 3 minutes while only slightly better than XGBoost.

The preprocessing pipeline with ideal processes to handle different types of features is also successfully constructed, which able to process the housing dataset to be utilized in ML process. In terms of feature importance, flat type and the area of the flat are proved to be the significant factors that heavily affect the housing price.

The project also successfully furthers the steps from theoretical results, to actually utilize the outcomes in real-world production. A dashboard was built on top of the outcomes of the experiment using Stremlit, which integrated the automated ML pipeline to enable end users (stakeholders) to actually utilize the power of machine learning to help them to make inform decisions.

7.2 Recommendation

For future works in this area, there are certain improvements that can be considered. First, the dataset used in this project is too clean and already includes certain additional features that may help to boost the models' performance, which cannot fully reflect the difficulties in real

production practices. Therefore, research using rougher datasets is needed to further consolidate the outcome of this project.

Besides, different data preprocessing methods can be further investigated in future research, as this project only constructed one general pipeline that applied to all models. There could be some room for improvement for other models' performance if the preprocessing pipeline is tailored to meet the characteristics of different model architectures.

On the other hand, the current implementation of the dashboard has some other improvements that can be made, such as model persistence with import and export functionalities, or database to store the housing data. These improvements can enable the framework to become more practical in real world situations. Future development works are recommended to build the dashboard as a complete full stack application with dedicated frontend and backend, instead of using all-in-one Streamlit for more robust functionalities.

Lastly, the project utilized Randomized Search for hyperparameter tuning. Despite being efficient, it might have missed certain combinations that actually better than current results. Future research can attempt to use Grid Search for a more credible outcome.

REFERENCES

[1] Kang, Yuhao, et al. "Human settlement value assessment from a place perspective: Considering human dynamics and perceptions in house price modeling." Cities 118 (2021): 103333.

[2] Mohd, Thuraiya, et al. "An overview of real estate modelling techniques for house price prediction." Charting a Sustainable Future of ASEAN in Business and Social Sciences: Proceedings of the 3rd International Conference on the Future of ASEAN (ICoFA) 2019—Volume 1. Springer Singapore, 2020.

[3] Soltani, Ali, et al. "Housing price prediction incorporating spatio-temporal dependency into machine learning algorithms." Cities 131 (2022): 103941.

[4] Waskom, Michael L. "Seaborn: statistical data visualization." Journal of Open Source Software 6.60 (2021): 3021.

[5] Luo, Huanhuan, Shengchuan Zhao, and Ronghan Yao. "Determinants of housing prices in Dalian city, China: Empirical study based on hedonic price model." Journal of Urban Planning and Development 147.2 (2021): 05021017.

[6] Vergara-Perucich, José Francisco. "Testing housing price drivers in Santiago de Chile: A hedonic price approach." Critical Housing Analysis 10.2 (2023): 44-57.

[7] Cao, K., Diao, M., & Wu, B. (2018). A Big Data–Based Geographically Weighted Regression Model for Public Housing Prices: A Case Study in Singapore. Annals of the American Association of Geographers, 1–14. doi:10.1080/24694452.2018.1470925

[8] Botchkarev, Alexei. "Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology." arXiv preprint arXiv:1809.03006 (2018).

[9] Adetunji, Abigail Bola, et al. "House price prediction using random forest machine learning technique." Procedia Computer Science 199 (2022): 806-813.

[10] Ho, Winky KO, Bo-Sin Tang, and Siu Wai Wong. "Predicting property prices with machine learning algorithms." Journal of Property Research 38.1 (2021): 48-70.

[11] Basysyar, Fadhil M., and Gifthera Dwilestari. "House price prediction using exploratory data analysis and machine learning with feature selection." Acadlore Trans. AI Mach. Learn 1.1 (2022): 11-21.

[12] Mora-Garcia, Raul-Tomas, Maria-Francisca Cespedes-Lopez, and V. Raul Perez-Sanchez. "Housing price prediction using machine learning algorithms in COVID-19 times." Land 11.11 (2022): 2100.

[13] Imran, Imran, et al. "Using machine learning algorithms for housing price prediction: the case of Islamabad housing data." Soft Computing and Machine Intelligence 1.1 (2021): 11-23.

[14] Abdul-Rahman, Shuzlina, et al. "Advanced machine learning algorithms for house price prediction: case study in Kuala Lumpur." International Journal of Advanced Computer Science and Applications 12.12 (2021).

[15] Yazdani, Mahdieh. "Machine learning, deep learning, and hedonic methods for real estate price prediction." arXiv preprint arXiv:2110.07151 (2021).

[16] Subaşı, Nadir. "Comprehensive analysis of grid and randomized search on dataset performance." *European Journal of Engineering and Applied Sciences* 7.2 (2024): 77-83.

[17] T. Lim, “Singapore resale flat prices (JAN-17 to jun-25),” Kaggle, https://www.kaggle.com/datasets/lzytim/hdb-resale-prices (accessed Aug. 12, 2025).

[18] “XGBoost GPU support□,” XGBoost GPU Support - xgboost 3.0.5 documentation, https://xgboost.readthedocs.io/en/stable/gpu/ (accessed Sep. 10, 2025).

[19] “Linearregression,” scikit, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (accessed Apr. 7, 2026).

[20] “Randomforestregressor,” scikit, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html (accessed Apr. 7, 2026).

[21] “Gradientboostingregressor,” scikit, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html (accessed Apr. 7, 2026).

[22] “XGBoost parametersℑ,” XGBoost Parameters - xgboost 3.2.0 documentation, https://xgboost.readthedocs.io/en/stable/parameter.html (accessed Apr. 7, 2026).

[23] “2025 BTO Supply,” Ministry of National Development (MND), https://www.mnd.gov.sg/newsroom/speeches/view/2025-bto-supply (accessed Apr. 26, 2026).

APPENDIX

Metadata

Column Information:

'month' - The month of flat transaction.

'town' - The HDB-defined town of the flat.

'blk_no' - An address subcomponent - the block number of the flat.

'road_name' - An address subcomponent - the road name of the flat.

'building' - An address subcomponent - the building name of the flat.

'postal' - An address subcomponent - the postal code of the flat.

'resale_price' - The price that the (resale) flat was transacted at.

'storey_range' - The storey range of the flat.

'flat_type' - The category of the flat.

'flat_model' - The flat model.

'lease_commence_date' - The date at which the flats lease commenced (flats are sold by the govenment on a 99-year lease term)

'remaining_lease_years' - The number of years of lease remaining for the flat at the time of purchase.

'remaining_lease_months' - The number of months of lease remaining for the flat at the time of purchase. 'remaining_lease_years' + 'remaining_lease_month' gives the total lease remaining at time of purchase.

'floor_area_sqm' - The size of the flat measured in square metres.

'floor_area_sqft' - The size of the flat measured in square feet

'price_per_sqft' - The price paid per square feet - a simple division of 'resale_price' and 'floor_area_sqft'

'planning_area_ura' - The URA planning area in which the flat is sited

'region_ura' - The cardinal direction of Singapore in which the flat is sited

'x' - The x coordinate of the flat (based on CRS: SVY21)

'y' - The y coordinate of the flat (based on CRS: SVY21)

'latitude' - The latitude coordinate of the flat

'longitude' - The latitude coordinate of the flat

'closest_mrt_station' - The closest MRT station to the flat

'distance_to_mrt_meters' - The displacement between the flat and the closest MRT station in metres (note that this is displacement and not distance)

'transport_type' - The transport type of the closest MRT station. MRT or LRT.

'line_color' - The line of the closest MRT staion. The line colour is analogous to the operational train routes (i.e North-South Line, Circle Line etc.)

'distance_to_cbd' - The displacement between the flat and Raffles Place MRT station in metres, which is chosen as a proxy for the centre of the Central Business District

'closest_pri_school' - The closest primary school to the flat

'distance_to_pri_school_meters' - The displacement between the flat and the closest primary school in metres

APPENDIX

Custom Column Transformers

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler, FunctionTransformer
from sklearn.model_selection import KFold, TimeSeriesSplit
import warnings
warnings.filterwarnings('ignore')

# Custom transformer for temporal features (modified to handle multiple date columns)
class TemporalFeatureExtractor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.feature_names_out_ = None
        self.input_columns_ = None

    def fit(self, X, y=None):
        if isinstance(X, pd.DataFrame):
            self.input_columns_ = X.columns.tolist()
        return self

    def transform(self, X):
        if isinstance(X, pd.DataFrame):
            X_df = X
        else:
            # If numpy array, assume column names from fit
            X_df = pd.DataFrame(X, columns=self.input_columns_)
```

```
        all_features = []

        # Process each date column
        for col in X_df.columns:
            # Convert to datetime
            date_series = pd.to_datetime(X_df[col])

            # Extract temporal features for this column
            col_features = np.column_stack([
                date_series.dt.year,
                date_series.dt.month,
                date_series.dt.day,
                np.sin(2 * np.pi * date_series.dt.month / 12),  # month_sin
                np.cos(2 * np.pi * date_series.dt.month / 12)   # month_cos
            ])

            all_features.append(col_features)

        # Concatenate all features horizontally
        return np.column_stack(all_features)

    def get_feature_names_out(self, input_features=None):
        if input_features is None and self.input_columns_ is not None:
            input_features = self.input_columns_
        elif input_features is None:
            input_features = ['date_col']

        feature_names = []
        for col in input_features:
            # Create unique feature names for each date column
            col_prefix = col.replace('_', '').replace(' ', '').lower()
            feature_names.extend([
                f'{col_prefix}_year',
```

```
            f'{col_prefix}_month',
            f'{col_prefix}_day',
            f'{col_prefix}_month_sin',
            f'{col_prefix}_month_cos'
        ])

    return feature_names

# Custom transformer for lease features
class LeaseFeatureCreator(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        if isinstance(X, pd.DataFrame):
            X_array = X.values
        else:
            X_array = X

        # Assuming columns are [remaining_lease_years, remaining_lease_months]
        total_months = X_array[:, 0] * 12 + X_array[:, 1]

        return total_months.reshape(-1, 1)

    def get_feature_names_out(self, input_features=None):
        return ['total_remaining_lease_months']

# PyTorch-accelerated Target Encoder with Cross-Validation
class TargetEncoder(BaseEstimator, TransformerMixin):
    def __init__(self, n_folds=5, smoothing=1.0):
```

```
        self.n_folds = n_folds
        self.smoothing = smoothing
        self.global_means_ = {}
        self.encodings_ = {}
        self.feature_names_in_ = None
        self.feature_names_out_ = None
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self._is_fitted = False

    def fit(self, X, y):
        if isinstance(X, pd.DataFrame):
            self.feature_names_in_ = X.columns.tolist()
            X_df = X
        else:
            # If X is numpy array, we need column names from somewhere else
            self.feature_names_in_ = [f'col_{i}' for i in range(X.shape[1])]
            X_df = pd.DataFrame(X, columns=self.feature_names_in_)

        # Convert target to PyTorch tensor
        if hasattr(y, 'values'):
            y_values = y.values
        else:
            y_values = y

        y_tensor = torch.tensor(y_values, dtype=torch.float32, device=self.device)

        for col in self.feature_names_in_:
            # Get unique categories and create mapping
            unique_cats = X_df[col].unique()
            cat_to_idx = {cat: idx for idx, cat in enumerate(unique_cats)}

            # Convert categories to indices tensor
            cat_indices = torch.tensor(
```

```
            [cat_to_idx[cat] for cat in X_df[col]],
            dtype=torch.long,
            device=self.device
        )

        # Calculate global mean for each category using PyTorch
        n_categories = len(unique_cats)
        category_sums = torch.zeros(n_categories, device=self.device)
        category_counts = torch.zeros(n_categories, device=self.device)

        # Use scatter_add for efficient aggregation
        category_sums.scatter_add_(0, cat_indices, y_tensor)
        category_counts.scatter_add_(0, cat_indices, torch.ones_like(y_tensor))

        # Apply smoothing and calculate means
        global_mean = y_tensor.mean()
        smoothed_means = (
            (category_sums + self.smoothing * global_mean) /
            (category_counts + self.smoothing)
        )

        # Store encodings
        self.encodings_[col] = {
            unique_cats[i]: smoothed_means[i].cpu().item()
            for i in range(n_categories)
        }
        self.global_means_[col] = global_mean.cpu().item()

    self.feature_names_out_ = [f'{col}_target_encoded' for col in self.feature_names_in_]
    self._is_fitted = True
    return self

  def transform(self, X):
```

```
        if not self._is_fitted:
            raise ValueError("This TargetEncoder instance is not fitted yet.")

        if isinstance(X, pd.DataFrame):
            X_df = X
        else:
            X_df = pd.DataFrame(X, columns=self.feature_names_in_)

        encoded_features = []
        for col in self.feature_names_in_:
            # Apply encoding with fallback to global mean
            encoded_col = X_df[col].map(self.encodings_[col]).fillna(self.global_means_[col])
            encoded_features.append(encoded_col.values)

        return np.column_stack(encoded_features)

    def fit_transform_cv(self, X, y):
        """Fit and transform with cross-validation to prevent overfitting"""
        if isinstance(X, pd.DataFrame):
            self.feature_names_in_ = X.columns.tolist()
            X_df = X
        else:
            self.feature_names_in_ = [f'col_{i}' for i in range(X.shape[1])]
            X_df = pd.DataFrame(X, columns=self.feature_names_in_)

        kf = TimeSeriesSplit(n_splits=self.n_folds)
        encoded_features = []

        # Convert target to PyTorch tensor
        if hasattr(y, 'values'):
            y_values = y.values
        else:
            y_values = y
```

```
    y_tensor = torch.tensor(y_values, dtype=torch.float32, device=self.device)

    for col in self.feature_names_in_:
        encoded_values = np.zeros(len(X_df))

        for train_idx, val_idx in kf.split(X_df):
            # Get training fold data
            X_train_fold = X_df.iloc[train_idx]
            y_train_fold = y_tensor[train_idx]

            # Create temporary encoder for this fold
            unique_cats = X_train_fold[col].unique()
            cat_to_idx = {cat: idx for idx, cat in enumerate(unique_cats)}

            # Convert categories to indices
            cat_indices = torch.tensor(
                [cat_to_idx.get(cat, -1) for cat in X_train_fold[col]],
                dtype=torch.long,
                device=self.device
            )

            # Filter out unknown categories
            valid_mask = cat_indices >= 0
            if valid_mask.sum() > 0:
                cat_indices_valid = cat_indices[valid_mask]
                y_train_valid = y_train_fold[valid_mask]

                # Calculate fold encodings
                n_categories = len(unique_cats)
                category_sums = torch.zeros(n_categories, device=self.device)
                category_counts = torch.zeros(n_categories, device=self.device)
```

```
                category_sums.scatter_add_(0, cat_indices_valid, y_train_valid)
                category_counts.scatter_add_(0, cat_indices_valid, torch.ones_like(y_train_valid))

                # Apply smoothing
                fold_mean = y_train_valid.mean()
                smoothed_means = (
                    (category_sums + self.smoothing * fold_mean) /
                    (category_counts + self.smoothing)
                )

                # Create fold encoding dictionary
                fold_encoding = {
                    unique_cats[i]: smoothed_means[i].cpu().item()
                    for i in range(n_categories)
                }

                # Encode validation fold
                for idx in val_idx:
                    cat = X_df.iloc[idx][col]
                    encoded_values[idx] = fold_encoding.get(cat, fold_mean.cpu().item())

            encoded_features.append(encoded_values)

        # Fit on full data for future transforms
        self.fit(X, y)
        return np.column_stack(encoded_features)

    def get_feature_names_out(self, input_features=None):
        if self.feature_names_out_ is None:
            if input_features is not None:
                return [f'{col}_target_encoded' for col in input_features]
            else:
                return [f'target_encoded_{i}' for i in range(len(self.feature_names_in_ or []))]
```

```
        return self.feature_names_out_

# Custom wrapper for CV Target Encoding in ColumnTransformer
class CVTargetEncoderWrapper(BaseEstimator, TransformerMixin):
    def __init__(self, n_folds=5, smoothing=1.0):
        self.n_folds = n_folds
        self.smoothing = smoothing
        self.encoder = None
        self._target = None

    def fit(self, X, y=None):
        if y is not None:
            self._target = y
            self.encoder = TargetEncoder(n_folds=self.n_folds, smoothing=self.smoothing)
            self.encoder.fit(X, y)
        return self

    def transform(self, X):
        if self.encoder is None:
            raise ValueError("Encoder not fitted. Call fit first.")
        return self.encoder.transform(X)

    def fit_transform(self, X, y=None):
        if y is not None:
            self._target = y
            self.encoder = TargetEncoder(n_folds=self.n_folds, smoothing=self.smoothing)
            return self.encoder.fit_transform_cv(X, y)
        else:
            return self.fit(X, y).transform(X)

    def get_feature_names_out(self, input_features=None):
        if self.encoder is not None:
            return self.encoder.get_feature_names_out(input_features)
```

```
    elif input_features is not None:
        return [f'{col}_target_encoded' for col in input_features]
    else:
        return []
```

APPENDIX

Unit Test

```
import io
import numpy as np
import pandas as pd
import pytest

from modules import (
    load_and_clean,
    prepare_xy,
    temporal_split,
    log_transform,
    evaluate,
    predict,
    merge_train_val,
    validate_train_columns,
    validate_predict_columns,
    validate_batch_columns,
    REQUIRED_TRAIN_COLS,
    REQUIRED_PREDICT_COLS,
    COLS_TO_DROP,
)

# ── Fixtures ──────────────────────────────────────────────────────────────────


def _make_row(**overrides) -> dict:
    """Return a minimal valid row matching REQUIRED_PREDICT_COLS."""
    base = {
        'month':                '2023-01',
        'lease_commence_date':  1990,
```

```
        'remaining_lease_years':         70,
        'remaining_lease_months':        0,
        'storey_range':                  '07 TO 09',
        'town':                          'ANG MO KIO',
        'flat_model':                    'Model A',
        'planning_area_ura':             'Ang Mo Kio',
        'closest_mrt_station':           'ANG MO KIO',
        'closest_pri_school':            'ANG MO KIO PRIMARY SCHOOL',
        'flat_type':                     '4 ROOM',
        'region_ura':                    'North Region',
        'transport_type':                'MRT',
        'line_color':                    'Red',
        'floor_area_sqft':               1000,
        'latitude':                      1.37,
        'longitude':                     103.85,
        'distance_to_mrt_meters':        400,
        'distance_to_cbd':               8000,
        'distance_to_pri_school_meters': 300,
    }
    base.update(overrides)
    return base


def _make_df(n=10, shuffled=False, with_price=True, with_duplicates=False) ->
pd.DataFrame:
    """Build a minimal valid DataFrame for pipeline testing."""
    months = pd.date_range('2020-01', periods=n, freq='MS').strftime('%Y-%m').tolist()
    rows = [_make_row(month=m) for m in months]
    if with_price:
        for i, r in enumerate(rows):
            r['resale_price'] = 300_000 + i * 10_000
    if with_duplicates:
        rows.append(rows[0].copy())
    df = pd.DataFrame(rows)
```

```
    if shuffled:
        df = df.sample(frac=1, random_state=42).reset_index(drop=True)
    return df

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════
# load_and_clean
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════


class TestLoadAndClean:

    def test_accepts_filepath(self, tmp_path):
        df = _make_df()
        p = tmp_path / "data.csv"
        df.to_csv(p, index=False)
        result = load_and_clean(str(p))
        assert isinstance(result, pd.DataFrame)
        assert len(result) == len(df)

    def test_accepts_bytesio(self):
        df = _make_df()
        buf = io.BytesIO(df.to_csv(index=False).encode())
        result = load_and_clean(buf)
        assert isinstance(result, pd.DataFrame)

    def test_drops_unnamed_index_column(self, tmp_path):
        df = _make_df()
        p = tmp_path / "data.csv"
        df.to_csv(p, index=True)   # index=True produces 'Unnamed: 0'
        result = load_and_clean(str(p))
```

```
        assert 'Unnamed: 0' not in result.columns

    def test_removes_duplicate_rows(self, tmp_path):
        df = _make_df(n=5, with_duplicates=True)
        assert len(df) == 6
        p = tmp_path / "data.csv"
        df.to_csv(p, index=False)
        result = load_and_clean(str(p))
        assert len(result) == 5

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
# prepare_xy
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════

class TestPrepareXY:

    def test_separates_target(self):
        df = _make_df()
        X, y = prepare_xy(df)
        assert 'resale_price' not in X.columns
        assert y.name == 'resale_price'
        assert len(X) == len(y)

    def test_drops_leakage_columns(self):
        df = _make_df()
        # Add leakage columns that should be dropped
        df['price_per_sqft'] = df['resale_price'] / 1000
        df['floor_area_sqm'] = 100
        X, _ = prepare_xy(df)
```

```
        for col in COLS_TO_DROP:
            assert col not in X.columns

    def test_missing_drop_cols_are_ignored(self):
        """prepare_xy should not crash if COLS_TO_DROP columns are absent."""
        df = _make_df()
        X, y = prepare_xy(df)   # none of COLS_TO_DROP are in _make_df
        assert len(X) == len(df)

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
# temporal_split
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════

class TestTemporalSplit:

    def _split(self, n=100, shuffled=True):
        df = _make_df(n=n, shuffled=shuffled)
        X, y = prepare_xy(df)
        return temporal_split(X, y)

    def test_returns_six_parts(self):
        result = self._split()
        assert len(result) == 6

    def test_sizes_sum_to_total(self):
        n = 100
        X_tr, X_val, X_te, y_tr, y_val, y_te = self._split(n=n)
        assert len(X_tr) + len(X_val) + len(X_te) == n
        assert len(y_tr) + len(y_val) + len(y_te) == n
```

```
    def test_default_ratio_70_15_15(self):
        n = 100
        X_tr, X_val, X_te, *_ = self._split(n=n)
        assert len(X_tr) == 70
        assert len(X_val) == 15
        assert len(X_te) == 15

    def test_split_is_chronological_not_random(self):
        """Train months must all be earlier than val months,
        which must all be earlier than test months."""
        X_tr, X_val, X_te, *_ = self._split(n=60, shuffled=True)
        train_max = pd.to_datetime(X_tr['month']).max()
        val_min   = pd.to_datetime(X_val['month']).min()
        val_max   = pd.to_datetime(X_val['month']).max()
        test_min  = pd.to_datetime(X_te['month']).min()
        assert train_max <= val_min, "Train bleeds into val (not chronological)"
        assert val_max   <= test_min, "Val bleeds into test (not chronological)"

    def test_x_and_y_indices_are_aligned(self):
        X_tr, X_val, X_te, y_tr, y_val, y_te = self._split()
        assert list(X_tr.index)  == list(y_tr.index)
        assert list(X_val.index) == list(y_val.index)
        assert list(X_te.index)  == list(y_te.index)

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
# log_transform
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
```

```
class TestLogTransform:

    def test_single_array(self):
        a = np.array([1.0, np.e, np.e**2])
        result = log_transform(a)
        np.testing.assert_allclose(result, [0.0, 1.0, 2.0])

    def test_multiple_arrays_returns_tuple(self):
        a = np.array([1.0, 2.0])
        b = np.array([3.0, 4.0])
        result = log_transform(a, b)
        assert isinstance(result, tuple)
        assert len(result) == 2

    def test_inverse_is_exp(self):
        """log_transform followed by np.exp should recover original values."""
        original = np.array([100_000.0, 350_000.0, 800_000.0])
        recovered = np.exp(log_transform(original))
        np.testing.assert_allclose(recovered, original, rtol=1e-10)

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
# evaluate
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════

class TestEvaluate:

    def test_perfect_predictions(self):
        y = np.array([100_000.0, 200_000.0, 300_000.0])
        m = evaluate(y, y, n_features=5)
```

```
        assert m['RMSE']   == pytest.approx(0.0)
        assert m['MAE']    == pytest.approx(0.0)
        assert m['R2']     == pytest.approx(1.0)

    def test_returns_all_keys(self):
        # n=20, n_features=2 → denominator = 20 - 2 - 1 = 17
        rng  = np.random.default_rng(0)
        y    = rng.uniform(100_000, 800_000, 20)
        pred = y + rng.normal(0, 5_000, 20)
        m = evaluate(y, pred, n_features=2)
        assert set(m.keys()) == {'RMSE', 'MAE', 'R2', 'Adj_R2'}

    def test_rmse_is_always_non_negative(self):
        y    = np.random.default_rng(0).uniform(100_000, 800_000, 50)
        pred = y + np.random.default_rng(1).normal(0, 10_000, 50)
        m = evaluate(y, pred, n_features=10)
        assert m['RMSE'] >= 0

    def test_adj_r2_penalises_more_features(self):
        y    = np.random.default_rng(0).uniform(100_000, 800_000, 100)
        pred = y + np.random.default_rng(1).normal(0, 10_000, 100)
        m_few  = evaluate(y, pred, n_features=2)
        m_many = evaluate(y, pred, n_features=50)
        assert m_few['Adj_R2'] > m_many['Adj_R2']

    def test_known_mae_value(self):
        # n=10, n_features=1 → denominator = 10 - 1 - 1 = 8
        # All errors are exactly 10_000, so MAE = 10_000
        y    = np.array([100_000.0, 200_000.0, 300_000.0, 400_000.0, 500_000.0,
                         600_000.0, 700_000.0, 800_000.0, 900_000.0, 1_000_000.0])
        pred = y + 10_000.0
        m = evaluate(y, pred, n_features=1)
        assert m['MAE'] == pytest.approx(10_000.0)
```

```
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════
# predict
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════

class TestPredict:

    class _DummyModel:
        """Stub model that returns log-scale values directly."""
        def __init__(self, log_values):
            self._vals = np.array(log_values)
        def predict(self, X):
            return self._vals

    def test_inverse_log_transform(self):
        log_prices = np.log([300_000.0, 500_000.0, 800_000.0])
        model = self._DummyModel(log_prices)
        result = predict(model, X=None)
        np.testing.assert_allclose(result, [300_000.0, 500_000.0, 800_000.0], rtol=1e-6)

    def test_no_negative_predictions(self):
        """clip(0) must ensure all predictions are non-negative."""
        # Force a very negative log value to simulate edge case
        model = self._DummyModel([-1000.0, np.log(500_000)])
        result = predict(model, X=None)
        assert np.all(result >= 0)

    def test_output_shape_matches_input(self):
        n = 20
```

```
        model = self._DummyModel(np.log(np.full(n, 400_000.0)))
        result = predict(model, X=np.zeros((n, 5)))
        assert result.shape == (n,)

#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
# merge_train_val
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════

class TestMergeTrainVal:

    def test_concatenates_correctly(self):
        X_tr  = np.ones((70, 5))
        X_val = np.ones((15, 5)) * 2
        y_tr  = np.zeros(70)
        y_val = np.ones(15)
        X_out, y_out = merge_train_val(X_tr, X_val, y_tr, y_val)
        assert X_out.shape == (85, 5)
        assert y_out.shape == (85,)

    def test_train_comes_before_val(self):
        X_tr  = np.array([[1], [2]])
        X_val = np.array([[3], [4]])
        y_tr  = np.array([10.0, 20.0])
        y_val = np.array([30.0, 40.0])
        X_out, y_out = merge_train_val(X_tr, X_val, y_tr, y_val)
        np.testing.assert_array_equal(X_out[:2], X_tr)
        np.testing.assert_array_equal(X_out[2:], X_val)
        np.testing.assert_array_equal(y_out[:2], y_tr)
        np.testing.assert_array_equal(y_out[2:], y_val)
```

```
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════
# Validation functions
#
═══════════════════════════════════════════════════════════════════════
═══════════════════════════════════


class TestValidateTrainColumns:

    def test_valid_df_returns_no_errors(self):
        df = _make_df(with_price=True)
        assert validate_train_columns(df) == []

    def test_missing_column_detected(self):
        df = _make_df(with_price=True).drop(columns=['town'])
        errors = validate_train_columns(df)
        assert len(errors) == 1
        assert 'town' in errors[0]

    def test_missing_target_detected(self):
        df = _make_df(with_price=False)
        errors = validate_train_columns(df)
        assert any('resale_price' in e for e in errors)

    def test_out_of_range_numeric_detected(self):
        df = _make_df(with_price=True)
        df.loc[0, 'latitude'] = 99.0   # clearly outside Singapore
        errors = validate_train_columns(df)
        assert any('latitude' in e for e in errors)

    def test_string_numeric_column_handled(self):
```

```
        """Mixed-type numeric columns (str + int) must not crash."""
        df = _make_df(with_price=True)
        df['floor_area_sqft'] = df['floor_area_sqft'].astype(str)
        errors = validate_train_columns(df)   # must not raise
        assert isinstance(errors, list)

class TestValidatePredictColumns:

    def test_valid_df_returns_no_errors(self):
        df = pd.DataFrame([_make_row()])
        assert validate_predict_columns(df) == []

    def test_missing_column_detected(self):
        df = pd.DataFrame([_make_row()])
        df = df.drop(columns=['floor_area_sqft'])
        errors = validate_predict_columns(df)
        assert any('floor_area_sqft' in e for e in errors)

    def test_out_of_range_distance_detected(self):
        df = pd.DataFrame([_make_row(distance_to_mrt_meters=99999)])
        errors = validate_predict_columns(df)
        assert any('distance_to_mrt_meters' in e for e in errors)

class TestValidateBatchColumns:

    def test_rejects_file_with_resale_price(self):
        df = _make_df(with_price=True)   # has resale_price
        errors = validate_batch_columns(df)
        assert any('resale_price' in e for e in errors)

    def test_accepts_file_without_resale_price(self):
        df = pd.DataFrame([_make_row()])   # no resale_price
        assert validate_batch_columns(df) == []
```

```
    def test_still_catches_missing_columns(self):
        df = pd.DataFrame([_make_row()])
        df = df.drop(columns=['town', 'flat_type'])
        errors = validate_batch_columns(df)
        assert len(errors) > 0
```

POSTER

FINAL YEAR PROJECT · BACHELOR OF INFORMATION SYSTEMS (HONS) (DIGITAL ECONOMY TECHNOLOGY)

Data Visualization for Data Science in Real Estate Market

Chua Min Xuan
23ACB01414
Supervisor: Ts Dr Zanariah binti Zainudin
UTAR FICT
FEB 2026

0.9302	$51999.24	$40661.48	212K+
R^2 SCORE	RMSE (SGD)	MAE (SGD)	TRANSACTIONS

ABSTRACT

This research focuses on performing housing price prediction using machine learning, and constructing an dashboard integrated with automated ML pipeline. A complete machine learning process from EDA to data preprocessing, model training, and hyperparameter tuning built a strong foundation for the project, while extend the scope to build a interactive dashboard allows the power of ML to effictievly enable end users to make informed decisions.

Housing is one of the basic human needs, and real estate has always had a high demand. It is crucial for stakeholders to correctly estimate the value of property, so that they are able to perform informed decisions.

01

METHODOLOGY

Data Collection & Cleaning
HDB resale records Jan 2017 – Jul 2025. Deduplication, null check, feature enrichment.

Exploratory Data Analysis
Temporal trends, geospatial distribution, price distribution, correlation analysis.

Feature Engineering & Preprocessing
Log transform · Target / ordinal / one-hot encoding · Standard scaling · VIF check.

Baseline Model Comparison
6 models evaluated — Linear Reg, SVM, Decision Tree, Random Forest, GBM, XGBoost.

Hyperparameter Tuning
Randomized search on top 4 models. Best params selected per model.

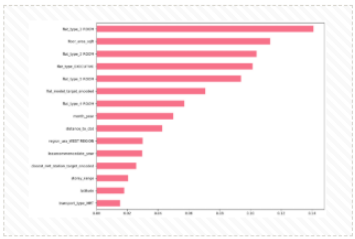
Final Evaluation on Test Set
XGBoost (tuned) — residual diagnostics, feature importance analysis.

MODEL COMPARISON (VALIDATION SET)

MODEL	R^2	RMSE (SGD)	MAE (SGD)
Linear Regression	0.8929	58,116	41,937
SVM	0.4061	136,829	120,746
Decision Tree	0.8554	67,508	48,263
Random Forest	0.9194	50,400	36,933
GBM (Tuned)	0.9492	40,001	29,520
★ XGBoost (Tuned)	**0.9473**	**40,775**	**29,484**

KEY PREDICTIVE FEATURES

- Floor area (sqft)
- Distance to MRT
- Storey range
- Transaction month
- Remaining lease
- Distance to CBD
- Town / Planning area
- Flat type & model



STREAMLIT DASHBOARD

Upload & Overview — CSV upload, validation, dataset stats & metadata

Dashboard — Price distribution, temporal trends, geospatial map

Model Training — Full pipeline, live progress, model download

Performance — Metrics, residual diagnostics, feature importance

Prediction — Single form & batch CSV inference with export

KEY FINDINGS

- XGBoost outperforms all baseline models after hyperparameter tuning.
- Floor area and flat types are the strongest price predictors.
- Distance to MRT and CBD show significant negative correlation.
- Temporal encoding captures market trend effects across years.

CONCLUSION

A robust model was built, and a easy to use dashboard built on top of it. Project objective have been accomplished to bridge the gap between theoretical results and practical usage.

FUTURE WORK

- Extend to other housing markets (Malaysia, Hong Kong).
- Examine on other dataset with different feature space
- Model versioning and periodic retraining pipeline.
- Specialized preprocessing for different model architecture

Data: Housing & Development Board (HDB), Singapore · National Data Repository
Stack: Python · XGBoost · scikit-learn · Streamlit · Plotly · Docker