1. Data and Data Preparation

1.1 Data Usage

The data used tracks HDB prices from 2017-March 2023. Earlier data was not used because the backdated data will not provide an accurate reflection of HDB prices of today, and the amount of data points (148, 277) was sizeable enough to train a robust model for the assignment's purpose of predicting future HDB prices.

1.2 Data Preparation

	month	town	flat_type	block	street_name	storey_range	floor_area_sqm	flat_model	lease_commence_date	remaining_lease	resale_price
0	2017-01	ANG MO KIO	2 ROOM	406	ANG MO KIO AVE 10	10 TO 12	44.0	Improved	1979	61 years 04 months	232000.0
1	2017-01	ANG MO KIO	3 ROOM	108	ANG MO KIO AVE 4	01 TO 03	67.0	New Generation	1978	60 years 07 months	250000.0
2	2017-01	ANG MO KIO	3 ROOM	602	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 years 05 months	262000.0
3	2017-01	ANG MO KIO	3 ROOM	465	ANG MO KIO AVE 10	04 TO 06	68.0	New Generation	1980	62 years 01 month	265000.0
4	2017-01	ANG MO KIO	3 ROOM	601	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 years 05 months	265000.0
...	...	...	...	...	...	...	...	...	...	...	...
148272	2023-03	YISHUN	5 ROOM	386	YISHUN RING RD	10 TO 12	127.0	Improved	1988	64 years 05 months	629000.0
148273	2023-03	YISHUN	5 ROOM	221	YISHUN ST 21	01 TO 03	126.0	Improved	1985	61 years 04 months	565000.0
148274	2023-03	YISHUN	5 ROOM	335C	YISHUN ST 31	13 TO 15	112.0	Improved	2015	91 years 09 months	625000.0
148275	2023-03	YISHUN	5 ROOM	820	YISHUN ST 81	07 TO 09	122.0	Improved	1988	64 years 06 months	650000.0
148276	2023-03	YISHUN	EXECUTIVE	356	YISHUN RING RD	01 TO 03	146.0	Maisonette	1988	64 years 06 months	800000.0

148277 rows × 11 columns

Fig. 1. Raw Data from "resale-flat-prices-based-on-registration-date-from-jan-2017-onwards.csv"

The raw data (Fig. 1) had several columns which was omitted from my analysis. The columns I excluded were "block", "street_name" and "lease_commence_date". The reasons for exclusion are as follows:

"block": Since the blocks are in numerical form, they imply an ordinal trend. This is not a huge problem as the blocks can be one-hot encoded into new columns, removing bias towards the ordering. However, when we start to think about different towns with the same block numbers, any two towns with the same block number will fall under the same category although having the same block numbers in different towns do not really have any strong relationship with each other. This results in us having to one-hot encode every unique block number in each town, which will in turn result in a huge number of new features. To avoid complicating the model with a large number of features, "block" was excluded. Moreover, information about the neighbourhood is encapsulated in the "town" feature, which was included in the model.

"street_name": Although there is a lower number of unique street names as compared to block (multiple blocks could be on the same street), "street_name" was excluded similarly to the above reasoning for "block". Too many new one-hot encoded features could overcomplicate the model, and the information about the neighbourhood is mostly encapsulated in the "town" feature.

"remaining_lease": This was excluded instead of "lease_commence_date" for ease of use. "remaining_lease" is essentially a new feature that was derived from "lease_commence_date", by taking (present time of the year – lease_commence_date). From a user perspective, if "remaining_lease" were to be used instead of "lease_commence_date", the "remaining_lease" has to be recalculated by adding the number of months that has passed since the model was trained to every row of the column. This makes the model not very user-friendly since the user essentially has to retrain the model himself to retrieve the most updated model, before inputting his features to predict a price. Although this retraining process could be automated, it will still be an inconvenient step because retraining the model takes compute power and time, both of which can be avoided by just using the "lease_commence_date" feature instead in training the initial model.

After dropping the aforementioned columns, the trimmed data is as such (Fig. 2).

	month	town	flat_type	storey_range	floor_area_sqm	flat_model	lease_commence_date	resale_price
0	2017-01	ANG MO KIO	2 ROOM	10 TO 12	44.0	Improved	1979	232000.0
1	2017-01	ANG MO KIO	3 ROOM	01 TO 03	67.0	New Generation	1978	250000.0
2	2017-01	ANG MO KIO	3 ROOM	01 TO 03	67.0	New Generation	1980	262000.0
3	2017-01	ANG MO KIO	3 ROOM	04 TO 06	68.0	New Generation	1980	265000.0
4	2017-01	ANG MO KIO	3 ROOM	01 TO 03	67.0	New Generation	1980	265000.0
...	...	...	...	...	...	...	...	...
148272	2023-03	YISHUN	5 ROOM	10 TO 12	127.0	Improved	1988	629000.0
148273	2023-03	YISHUN	5 ROOM	01 TO 03	126.0	Improved	1985	565000.0
148274	2023-03	YISHUN	5 ROOM	13 TO 15	112.0	Improved	2015	625000.0
148275	2023-03	YISHUN	5 ROOM	07 TO 09	122.0	Improved	1988	650000.0
148276	2023-03	YISHUN	EXECUTIVE	01 TO 03	146.0	Maisonette	1988	800000.0

148277 rows × 8 columns

Fig. 2. Trimmed Data

The features “month”, “flat_type”, “storey_range”, “town” and “flat_model” are not in a numerical form that a regression model can be trained from. For these features, two different types of encoding are used to convert these features into a numerical form. The method used for each column is as described:

Ordinal Encoding: “month”, “flat_type”, “storey_range”
One-Hot Encoding: “town”, “flat_model”

The reason ordinal encoding is used on the aforementioned features is because “month”, “flat_type” and “storey_range” are categorical features with a natural ordering. For example, “2017-01” comes before “2017-02” and they are mapped to “0” and “1” respectively using ordinal encoding. Whereas in the features “town” and “flat_model”, we do not want to impose any ordering on the transformed data, hence One-Hot Encoding is used to create a binary column for each category.

This results in our final processed data having 52 features in total, excluding the resale_price column which will form our y data (Fig. 3) (total columns = 53). The respective encodings for each feature can be found in the appendix.

	month	flat_type	storey_range	town_ANG MO KIO	town_BEDOK	...	flat_model_Type S1	flat_model_Type S2	floor_area_sqm	lease_commence_date	resale_price
0	0.0	1.0	3.0	1.0	0.0	...	0.0	0.0	44.0	1979.0	232000.0
1	0.0	2.0	0.0	1.0	0.0	...	0.0	0.0	67.0	1978.0	250000.0
2	0.0	2.0	0.0	1.0	0.0	...	0.0	0.0	67.0	1980.0	262000.0
3	0.0	2.0	1.0	1.0	0.0	...	0.0	0.0	68.0	1980.0	265000.0
4	0.0	2.0	0.0	1.0	0.0	...	0.0	0.0	67.0	1980.0	265000.0
...	...	...	...	...	...	...	...	...	...	...	...
148272	74.0	4.0	3.0	0.0	0.0	...	0.0	0.0	127.0	1988.0	629000.0
148273	74.0	4.0	0.0	0.0	0.0	...	0.0	0.0	126.0	1985.0	565000.0
148274	74.0	4.0	4.0	0.0	0.0	...	0.0	0.0	112.0	2015.0	625000.0
148275	74.0	4.0	2.0	0.0	0.0	...	0.0	0.0	122.0	1988.0	650000.0
148276	74.0	5.0	0.0	0.0	0.0	...	0.0	0.0	146.0	1988.0	800000.0

148277 rows × 53 columns

Fig. 3. Final Post-processed Dataset (truncated to display 10 columns)

1.3 Other Considerations

In the data-processing stage, I have considered the use of normalisation methods such as mean-normalisation to ensure that the scales of each of the features were the same. However, if we think carefully about our training data, we notice that the features generated by one-hot encoding “town” and “flat-model” generated sparse matrices. Mean normalisation will not be able to handle the high variability of the input data, as it contains a mix of both sparse and dense matrices.

Moreover, in predicting future prices of HDB properties, the input data for the “month” feature will always exceed the scale of the data for this particular feature. This is because the input value for a property when we want to predict the price in “2023-4” will be “75”, and this is clearly outside the scale of 0-74 of the training data for the particular feature. This might cause the model to make inaccurate predictions. In this case, I have also considered re-normalising the data based on the input values that a user inputs, but this will require re-training the model based on newly-scaled training data (note: not training with the unseen data, but **newly-scaled training data**). This makes the prediction process unnecessarily long as it requires computation and time, thus not user-friendly. This case of input value falling outside the scale of the training data also applies to “storey-range”, if a user were to predict the price of the property that is situated above floor 51.

Lastly, not all models require the data to be mean-normalised to provide meaningful predictions. Models such as linear regression and support vector regression benefit from mean-normalised data because these models are sensitive to the scale of the input features. However, the input features of my model do not have **vastly different** scales such that it will heavily disrupt the training process by giving more weight to certain features over others. In the case of K-Nearest Neighbours regression, mean-normalised data could ensure that the distances between data points are calculated more accurately, thus improving its performance. On the other hand, for Decision Tree Regression, Random Forest Regression and XGBoost regression, mean-normalising the data may not be as beneficial. This is because these models make splits based on threshold values of the training features, hence the scale of the features do not impact the splits. In fact, the tree-based models are designed to handle data with scales that vary and do not require the training data to be mean-normalised.

2. Methodology

2.1 Splitting Training and Testing Data

After data-processing, the next step taken was to consider how to split the data into training and test sets in a way that can best evaluate any models. The strategy taken was to use data from the year 2022 to present time as the testing data, as it is practical to train data based on an earlier time period in order to get a good gauge of the prices in the future. Data from the start of 2017 to the end of 2021 was used as the training data. The resulting split resulted in a training data size of 116, 676 and

testing data of size 31, 601, which roughly translates to 78.7% of the dataset used as training data and 21.3% of the total dataset as testing data.

2.2 GridSearchCV with K-Fold Cross Validation

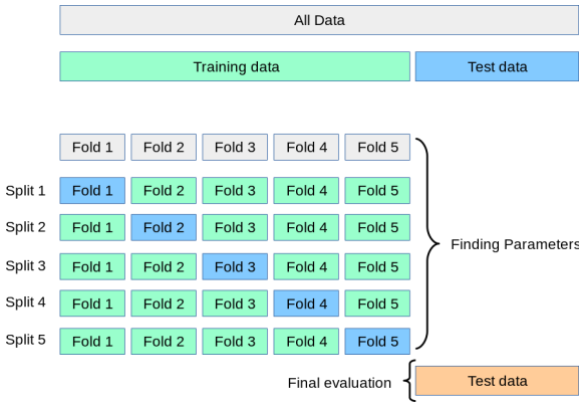

Fig. 4. Cross Validation with 5 folds (image source: scikit-learn.org)

In training the regression models, I made use of grid search in conjunction with 5-fold cross validation (Fig. 4) to tune hyperparameters for the models. Each combination of hyperparameters produced 5 different models, each trained with the 4 other folds of the split and then validated against each different fold to get the aggregate performance of the model across all 5 splits. By doing so, this gives us a more reliable estimate of the model's performance on unseen data, thereby ensuring that the models are generalising well to unseen data and preventing the case of overfitting. Fig. 5 shows an example of how the parameter grid was defined for GridSearch to exhaustively search over the range of values for each hyperparameter to find the optimal combination of hyperparameters that results in a model with the best performance over the training data.

```
# Define parameter grid for XGBoost regression
param_grid = {'learning_rate': [0.1, 0.2, 0.3],
              'n_estimators': [300, 400, 500],
              'max_depth': [5, 6, 7]}
```

Fig. 5. Performing Grid Search over the features for XGBoost regression

The same scoring function (Fig. 6) was defined for all models used in my analysis. The scoring function was defined in such a way that it takes **both the R^2 and the Root Mean Squared Error** (RMSE) as metrics for evaluating the performance of the model during the grid search process. These two metrics are used in conjunction with each other provides a more comprehensive picture of the model's performance, as opposed to using only one of either of the metrics.

However, in selecting the final model with the best parameters, R^2 score is used as this was defined as the refitting criteria (Fig. 6). R^2 was chosen as the metric to choose the best set of hyperparameters because it gives us good information about the model's fit, since a high R^2 indicates that the model is capturing important relationships in the data, translating to a high proportion of the variance of the y values (prices) being explained by the input features of the model.

```
# Create scoring functions
scoring = {'r2': 'r2', 'rmse': make_scorer(mean_squared_error, squared=False, greater_is_better=False)}

# Perform grid search
grid = GridSearchCV(xgbr, param_grid, scoring=scoring, refit='r2', cv=5)
grid.fit(X_train, y_train)
```

Fig. 6. Scoring function defined for all models and Refitting Data based on R^2 of the model

3. Model Evaluation and Model Selection

After the optimal set of hyperparameters for each model was found, the complete set of training data was used to retrain the model, then this retrained model is evaluated against the test set (Year 2022 to present) to obtain the metric "RMSE of predictions" and "R^2 of predictions". For each model, a baseline model was first trained without the use of grid search, so as to set some basis for comparison after the hyperparameters of each model were tuned and also to find the training duration for each model. This information was useful in deciding the range of each hyperparameter to tune when using grid search in the later part of my analysis, as the computational times of each model varied greatly. The results of the baseline models are as follows:

Baseline Models						
Model Name / Scoring Function	Linear Regression	Support Vector Regression	Decision Tree Regression	Random Forest Regression	K Nearest Neighbours Regression	XGBoost Regression
R^2	0.86814	0.75541	0.84148	0.86628	0.87163	0.95757
Root Mean-Squared-Error of Predictions	87861.84	112455.61	91683.17	84644.19	89850.92	51328.07
R^2 of Predictions	0.73359	0.56357	0.70991	0.75275	0.72139	0.90908

At first glance, the results of the XGBoost regression model seem promising, even without any hyperparameter tuning of the model. Noting the training times of the baseline models in mind, I noticed that the SVR took exceptional long just to train a single model. Hence, I did not proceed with hyperparameter tuning on the model, as searching across 2 parameters each with 3 values, and performing 5-fold cross validation on top of that is enough to increase the time taken by 45 times. Since what an SVR does is that it finds the hyperplane that maximally separates data points while still allowing a margin of error, this would be computationally expensive in this case of a high-dimensional feature-space. Keeping reproducibility of the models and user-friendliness in mind, I have chosen to exclude SVR from the hyperparameter-tuning process. Moreover, the baseline model showed disappointing results as compared to the other baseline models as well. Linear Regression is also not included in the second table as there are no hyperparameters to tune in the model other than specifying whether to fit the intercept.

With the results of the baseline models in mind, I went on to tune the hyperparameters of each model. In the hyperparameter-tuning process, I performed several iterations of grid search on each model, each time specifying values of the parameters that are closer to the best parameters outputted by grid search. Note that Linear Regression does not have any hyperparameters to tune, hence it is excluded from the following table.

Hyperparameter-Tuned Models				
Model Name / Scoring Function	Decision Tree Regression	Random Forest Regression	K Nearest Neighbours Regression	XGBoost Regression
R^2	0.96798	0.90460	0.99673	0.97060
Root Mean-Squared-Error of Predictions	61267.15	72327.34	76457.91	47758.06
R^2 of Predictions	0.87046	0.81947	0.79826	0.92129
Parameters Searched Across	'criterion': ['squared_error', 'friedman_mse'] 'max_depth': [None, 5, 10] 'min_samples_leaf': [2, 4, 6]	'n_estimators': [400, 500, 600] 'max_depth': [5, 10, 15] 'min_samples_leaf': [1, 2, 4]	'n_neighbors': [6, 7, 8, 9] 'weights': ['uniform', 'distance'] 'p': [1, 2] 'algorithm': ['ball_tree', 'kd_tree']	'learning_rate': [0.1, 0.2, 0.3] 'n_estimators': [300, 400, 500] 'max_depth': [5, 6, 7]
Best Parameters Used	'criterion': 'friedman_mse' 'max_depth': None 'min_samples_leaf': 6	'max_depth': 15 'min_samples_leaf': 1 'n_estimators': 400	'algorithm': 'ball_tree' 'n_neighbors': 8 'p': 1 'weights': 'distance'	'learning_rate': 0.2 'max_depth': 6 'n_estimators': 500

Chosen Model: XGBoost Regression

Overall, the performance of all the models have improved after performing hyperparameter-tuning, especially in the case of Decision Tree regression which made the greatest improvement. At first glance, KNN Regression actually has the highest R^2 within the training dataset (0.99673). However, when it comes to the RMSE of the predictions in the test data and R^2 of predictions in the test data, it loses out to the XGBoost regression algorithm, having values of 76457.91 and 0.79826 respectively. From the huge drop in the R^2 between the training and testing data of KNN Regression, we can tell that the KNN Regression model is overfitted, and is not generalising well to new data.

XGBoost Regression has the best overall results in terms of the 3 evaluation metrics I have focused on. Not only does it have the smallest RMSE of the test data among the other models, it also has the highest R^2 with the test data and the second highest R^2 with the training data. This tells us that the XGBoost Regression model is able to explain a large proportion of the variance in training data. The low RMSE also suggests that the model is making relatively accurate predictions on unseen data, which is crucial because it implies that the model is not overfitting to the training data since it generalises well with unseen data. The predictive power of the XGBoost Regression model is also reinforced by the high R^2 value of the test data, implying that is it able to explain a large proportion of the variance in the test data. The impressive results of these 3 metrics combined together suggests that the XGBoost Regression model is a suitable model for this particular regression task, leading me to choose it as my final model for the regression task. Moreover, the baseline model performance showed extremely impressive results on its own without any hyperparameter tuning, which turns out to be an advantage in terms of reproducibility and user-friendliness when it comes to deploying the model for the layperson.

There are several reasons why XGBoost regression outperformed the other models in this regression task. Whereas models such as Decision Tree Regression and Random Forest Regression are more prone to overfitting and require a lot of hyperparameter tuning work to prevent overfitting, XGBoost employs regularisation techniques such as L1 and L2 regularisation to prevent overfitting of the model with respect to the training data. Since the number of features is large (52), preventing overfitting is an important task that XGBoost regression could handle well. The fact that the baseline XGBoost regression model already performed well without tuning any hyperparameters is a testament to that. XGBoost regression was also designed to handle high-dimensional data well, whereas the other models such as Support Vector Regression and K Nearest Neighbours might drop in performance when dealing with high-dimensional data, or require more steps in pre-processing such as mean normalisation to effectively handle these data. Additionally, XGBoost regression excels at capturing non-linear relationships between the input features and the target "y" variable, which is crucial in tasks like predicting property prices whereby one cannot instinctively do so. Although Decision Tree regression and Random Forest Regression are capable of capturing these non-linear relationships as well, they are not as flexible in learning the relationships between the input features themselves. Also, the XGBoost regression model utilises ensemble learning, meaning that it combines the results of multiple weak models such as decision trees to produce a more robust model. Other models such as Support Vector Regression and K Nearest Neighbours regression do not utilise ensemble learning, thereby limiting their performance in this regression task. Lastly, XGBoost regression is convenient in my use-case whereby I did not perform mean normalisation to the data beforehand, since it handles data with input features of different scales well, unlike models such as Linear Regression and Support Vector Regression as aforementioned in an earlier section.

4. Adapting the Model to the General Public

4.1 How to Use Proposed Model To Determine Fair Price

The XGBoost Regression model is first retrained with ALL the data in the dataset (2017-2023 Current), with the following parameters {'learning_rate': 0.2, 'max_depth': 6, 'n_estimators': 500}

The end user would take the following steps to determine the fair price of a property:

1. With the dataset "201791to202303_processed.csv"in the same folder as the jupyter notebook, run the code in "UsersNotebook_Jo-Wayn.ipynb". The model is fitted directly based on the processed data, and it only takes **2 minutes** to train, thus making it extremely easy to use.
2. The User will be prompted with all the details of the property to be filled in, one-by-one. The first field is shown below as an example. For the flat model field, since there are many options, the user can refer to the chart in the appendix to see the available options. There is also input validation in-built into the code. Once the input is invalid, the code exits. This avoids misleading the user.

```
Enter year-month, e.g 2023-05: 2017-03
```

3. After the User finishes filling in all the blanks, the estimated price of the property is generated:

```
Enter year-month, e.g 2023-05: 2017-03
Enter number of rooms, e.g 3 or 'executive', or 'multi-generation': 5
Enter the storey number, e.g 33: 25
Enter the name of the town, e.g Hougang: Punggol
Enter flat model, e.g Standard: Standard
Enter floor area in sqm: 80
Enter your lease commencement year, e.g 2003: 2001


The estimated resale price as of 2017-03 for a 5 room flat on the 25 floor in Punggol, with flat model
 as Standard, floor area in sqm as 80.0 with its lease commencement year in 2001 is [469263.4]
```

4.1 Notable Findings

The model is capable of making accurate predictions prices of any HDB properties from 2017 to the current month (2023-03). HDB owners who are considering selling their property or buying a new property ought to try plotting out the prices of the properties against time to make the best decisions. The price movement chart for a 5-room flat on the 25th floor in Punggol, with flat model as Standard and floor area as 80sqm and lease commencement year in 2001 is as follows. Based on this chart, the price of this property has been appreciating year really quickly over the last 2 years, but it cannot be generalized to all properties. Users can also consider plotting the data by month instead of by year to scrutinise the micro-trends in price changes.

Appendix

	month
0	2017-01
1	2017-02
2	2017-03
3	2017-04
4	2017-05
...	...
70	2022-11
71	2022-12
72	2023-01
73	2023-02
74	2023-03

Appx !. Mapping for month (truncated)

	flat_type
0	1 ROOM
1	2 ROOM
2	3 ROOM
3	4 ROOM
4	5 ROOM
5	EXECUTIVE
6	MULTI-GENERATION

Appx. 2. Mapping for flat_type

	storey_range
0	01 TO 03
1	04 TO 06
2	07 TO 09
3	10 TO 12
4	13 TO 15
5	16 TO 18
6	19 TO 21
7	22 TO 24
8	25 TO 27
9	28 TO 30
10	31 TO 33
11	34 TO 36
12	37 TO 39
13	40 TO 42
14	43 TO 45
15	46 TO 48
16	49 TO 51

Appx. 3. Mapping for storey range

	town
0	ANG MO KIO
1	BEDOK
2	BISHAN
3	BUKIT BATOK
4	BUKIT MERAH
5	BUKIT PANJANG
6	BUKIT TIMAH
7	CENTRAL AREA
8	CHOA CHU KANG
9	CLEMENTI
10	GEYLANG
11	HOUGANG
12	JURONG EAST
13	JURONG WEST
14	KALLANG/WHAMPOA
15	MARINE PARADE
16	PASIR RIS
17	PUNGGOL
18	QUEENSTOWN
19	SEMBAWANG
20	SENGKANG
21	SERANGOON
22	TAMPINES
23	TOA PAYOH
24	WOODLANDS
25	YISHUN

Appx. 4. Mapping for town

	flat_model
0	2-room
1	3Gen
2	Adjoined flat
3	Apartment
4	DBSS
5	Improved
6	Improved-Maisonette
7	Maisonette
8	Model A
9	Model A-Maisonette
10	Model A2
11	Multi Generation
12	New Generation
13	Premium Apartment
14	Premium Apartment Loft
15	Premium Maisonette
16	Simplified
17	Standard
18	Terrace
19	Type S1
20	Type S2

Appx. 5. Mapping for flat model