République Algérienne Démocratique

et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université M'hamed Bougara - Boumerdès

Faculté des Sciences

Département d'Informatique

Domaine : Mathématiques et Informatique

Filière : Informatique

Spécialité : Ingénieur en informatique

Fia project

Presenter par : - Issaad Aya., Benfattoum Meriem Salsabil , Dahmani Assia , Nebak Rayan

Groupe 03.

1 . Dataset Description

This dataset is the well-known King County House Prices Dataset, which contains information about houses sold in King County, USA.

1.1. Attributes (Columns)

Common columns in this dataset include:

- id — Unique house identifier
- date — Date of sale
- price — Target variable (house price)
- bedrooms — Number of bedrooms
- sqft_living — House size (interior)
- sqft_lot — Land size
- floors — Number of floors
- waterfront — Is the house facing the water (0/1)
- view — Quality of the view (0–4)
- condition — House condition (1–5)
- grade — Housing grade (1–13)
    - sqft_above — Size above ground
    - sqft_basement — Basement size
- yr_built — Year the house was built
- yr_renovated — Year renovated
- zipcode — Postal code
- lat — Latitude
- long — Longitude
- sqft_living15 — Living room size of nearby houses
- sqft_lot15 — Lot size of nearby houses

1.2 Target Variable

price : This is the value we want to predict.

1.3 Data Dictionary:

Column	Type	Description	Possible Values
id	Integer	Unique house identifier	Unique number
date	String	Sale date	yyyy-mm-dd
price	Numeric	Target — sale price	continuous
bedrooms	Numeric	Number of bedrooms	0–10
bathrooms	Numeric	Number of bathrooms	0–8
sqft_living	Numeric	Interior size in ft²	continuous
sqft_lot	Numeric	Lot size in ft²	continuous
floors	Numeric	Number of floors	1–4
waterfront	Categorical	Waterfront view	0 = No, 1 = Yes
view	Categorical	View rating	0–4
condition	Categorical	House condition	1–5
grade	Categorical	Construction grade	1–13
sqft_above	Numeric	Area above ground	continuous
sqft_basement	Numeric	Basement area	continuous
yr_built	Numeric	Construction year	1900–2015
yr_renovated	Numeric	Renovation year	0 = never
zipcode	String	Postal code	e.g., 98001
lat	Numeric	Latitude	coordinate
long	Numeric	Longitude	coordinate
sqft_living15	Numeric	Nearby house size	continuous
sqft_lot15	Numeric	Nearby lot size	continuous

Total missing values (NULL + EMPTY): Total missing values in the whole dataset = 0.

1.4 Data Quality Analysis

1. Missing Values Overview

After analyzing the dataset:

- **No NULL values** were found in any column.
- **No empty string values** were detected.
- **Total missing values in the entire dataset = 0**

Although the dataset contains no formal missing values, some columns include **placeholder values (0)** that behave like missing data and must be treated carefully.

2. Columns With Missing-Like Values

2.1 `sqft_basement`

- Many rows contain **0**.
- A value of **0** may represent:
  - The house has **no basement**, or
  - The basement size was **not recorded**.
- This creates a mix of:
  - Valid basement sizes
  - Missing-like placeholders
- Must be handled as a special case before training the model.

2.2 `yr_renovated`

- Most values are **0**, except for a small percentage.
- Interpretation:
  - **0 = Never renovated**
  - A non-zero value = Year of renovation

- This column behaves like a **hybrid categorical feature**, not a continuous numeric attribute.

3. Recommended Handling Strategy

✔ For Numeric Features (e.g., `sqft_basement`)

If treating **0 as missing**:

- Replace using the **median** value
  or
- Create a binary indicator:

```
basement_exists = 1 if sqft_basement > 0 else 0
```

This prevents the model from misinterpreting "0" as a valid size.

✔ For Categorical / Hybrid Features (e.g., `yr_renovated`)

Transform into meaning categories:

yr_renovated	New Meaning
0	Never Renovated
> 0	Renovated

Additionally, create a useful binary feature:

```
is_renovated = 1 if yr_renovated > 0 else 0
```

✔ Final Summary

- The dataset contains **no official missing values**, but `sqft_basement` and `yr_renovated` contain structural placeholders that act like missing data.
- Proper handling improves model reliability and prevents misunderstanding of the data.

Outliers Detection

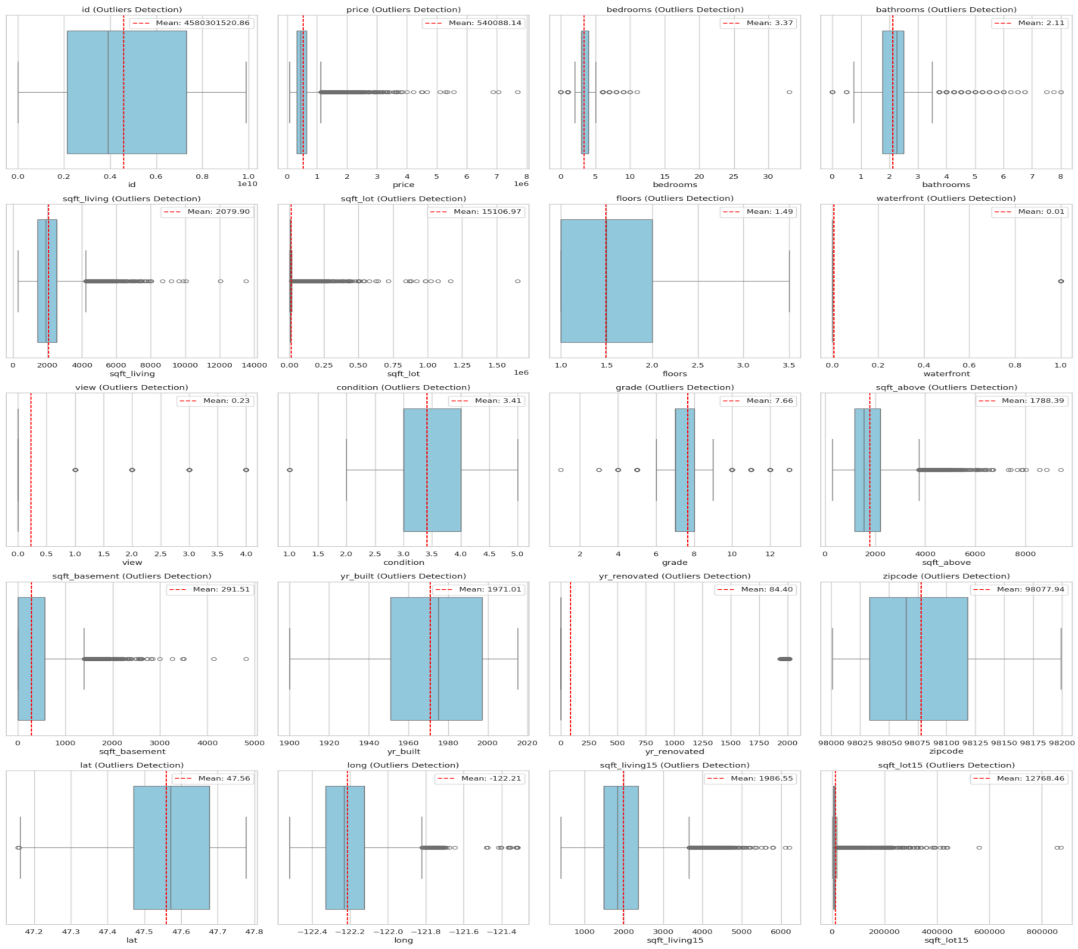
Possible outliers:

- Houses with > 10 bedrooms
- Houses priced above $5M
- Extremely large sqft_living (>10,000 ft²)

Detection methods:

- Boxplots Visualization
- IQR (Interquartile Range)
- Z-score
- Boxplot for a Single Column
- Histogram + KDE (for distribution inspection)

1. Outlier Detection using IQR Method :



2. Problem Formulation:

2.1 Type of ML Task:

Regression

- The goal is to **predict a continuous numerical value**: `price`.
- Regression algorithms are suitable since the target is numeric.

2.2 Objective:

Develop a machine learning model capable of accurately predicting house prices based on property features.

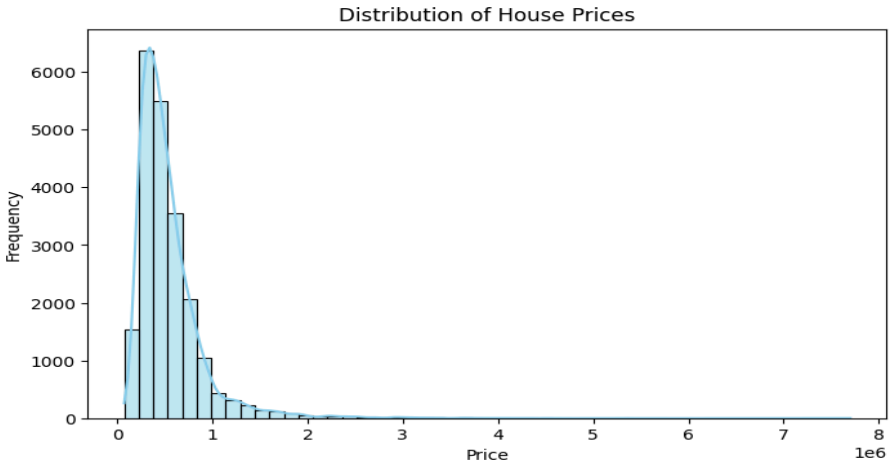
- Predictive accuracy is the main evaluation criterion.
- The model should generalize well to unseen data.

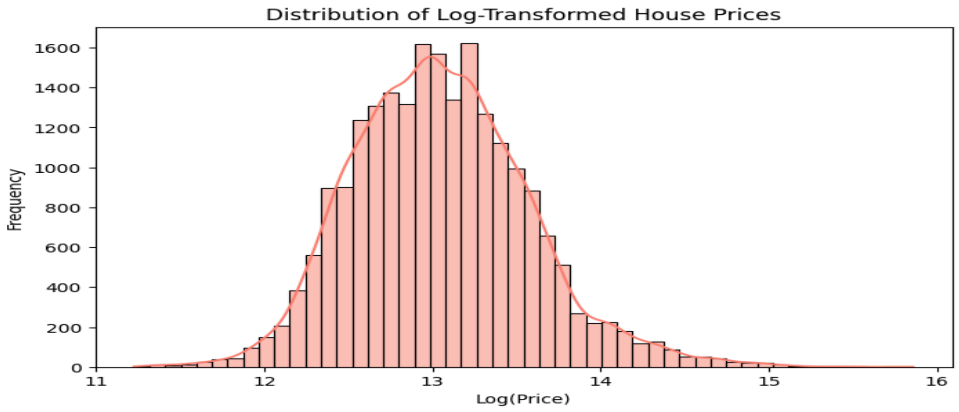
2.3 Data Balance Check:

- Since this is a regression problem, **class imbalance is not directly relevant**.
- However, it is important to check the **distribution of the target variable (`price`)**.

Price Distribution Insights

- The `price` data is **right-skewed**:
  - Most houses are **low to moderately priced**.
  - A few houses are **very expensive**, creating long tails.
- **Implication:** Right-skewed targets may affect model performance.
- **Common solution:** Apply a **log transformation** to normalize the distribution.





2.4 Evaluation Metrics

For regression tasks, we use metrics that evaluate how close the predicted values are to the actual target values (`price`).

1. Root Mean Squared Error (RMSE)

- Measures the **square root of the average squared differences** between predicted and actual values.
- Sensitive to **large errors (outliers)**.
- **Lower RMSE → better model performance**.

2. Mean Absolute Error (MAE)

- Measures the **average absolute differences** between predicted and actual values.
- Less sensitive to outliers than RMSE.
- **Lower MAE → better model performance**.

3. R^2 Score (Coefficient of Determination)

- Measures **proportion of variance explained** by the model.
- Value ranges from **0 to 1** (sometimes negative if the model is worse than baseline).
- **R^2 close to 1 → excellent model performance**.

✔ Goal

- **Lower RMSE and MAE** → better predictive accuracy.
- **R^2 close to 1** → model explains most of the variance in the target.

3. Data Preprocessing

3.1 Train / Validation / Test Split

Before training the model, we split the dataset into three sets:

Set	Proportion	Purpose
Training	70%	Learn patterns from the data
Validation	15%	Tune hyperparameters and prevent overfitting
Test	15%	Evaluate final model performance

Why this split is important:

- **Training set:** The model learns the relationships between features and the target.
- **Validation set:** Helps select the best model parameters and avoid overfitting.
- **Test set:** Provides an unbiased evaluation of the final model's predictive power.

3.2 Handling Missing Data (Specific to This Dataset)

Upon inspection of the dataset:

- **No NULL or empty values** were found in any column.
- Some columns contain **0 as placeholders**, which may represent missing-like values:
  - `sqft_basement` → 0 may indicate **no basement**
  - `yr_renovated` → 0 indicates **never renovated**

Handling Strategy for This Dataset

Numeric Columns

- Replace 0s only if they are **truly missing or meaningless**, otherwise leave them.
- For example, if treating `sqft_basement=0` as missing:S

	sqft_basement
0	0
1	400
2	0
3	910
4	0
5	1530
6	0
7	0
8	730
9	0

	sqft_basement
count	21613.000000
mean	291.509045
std	442.575043
min	0.000000
25%	0.000000
50%	0.000000
75%	560.000000
max	4820.000000

Create a binary indicator for basement existence:

	sqft_basement	basement_exists	sqft_basement_imputed
0	0	0	700
1	400	1	400
2	0	0	700
3	910	1	910
4	0	0	700
5	1530	1	1530
6	0	0	700
7	0	0	700
8	730	1	730
9	0	0	700

Key Takeaways: `sqft_basement` Handling

1️⃣ `basement_exists`

- Binary indicator of whether a basement exists.

- Helps the model distinguish **houses with no basement (0)** from real basement sizes.

2️⃣ `sqft_basement_imputed`

- 0 values replaced by **median of non-zero basements (700 ft²)**.
- Preserves information for models that may misinterpret 0 as missing.

3️⃣ Original `sqft_basement`

- Remains intact, so you can use **either the original or imputed version** depending on model requirements.

This is a **best practice** for datasets with **structural zeros**, like `sqft_basement`.

Categorical Columns

No missing values, so no imputation needed.

Columns like `waterfront`, `view`, `condition`, `grade` are fully populated

3.3 Converting Categorical Variables

Machine learning models require **numeric input**, so categorical features must be encoded appropriately. The type of encoding depends on whether the feature is **ordered (ordinal)** or **unordered (nominal)**.

1️⃣ Target Encoding (Categorical Feature)

- Used for categorical features where the target variable may carry useful information.

Example in this dataset: `zipcode`

- Replaces each category with the average value of the target (price) for that category, helping the model capture the relationship between category and target without creating too many columns.

Label Encoding (Ordinal)

Used for ordered categories where the order matters.

Examples in this dataset: condition, grade

Converts categories to integer values reflecting their order.

```
data[['condition', 'grade']]:
```

	condition	grade
0	2	5
1	2	5
2	2	4
3	4	5
4	2	6
...	...	...
21608	2	6
21609	2	6
21610	2	5
21611	2	6
21612	2	5

Notes :

- After encoding, all features are numeric, ready for machine learning algorithms.
- One-Hot Encoding avoids introducing false order in nominal features.
- Label Encoding preserves the natural order for ordinal features.

3.4 Scaling Numeric Features

Scaling numeric features is an important preprocessing step for certain machine learning algorithms.

It ensures that features are on a similar scale, which improves **model performance and convergence**.

1️⃣ When Scaling is Important

Algorithms sensitive to feature scale include:

- **Linear Regression**
- **K-Nearest Neighbors (KNN)**
- **Support Vector Machines (SVM)**
- **Neural Networks**

2️⃣ Scaling Methods

StandardScaler

- Centers features to **mean = 0** and **std = 1**.
- Useful when features have different units and large variance.

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
0	0.886146	-0.866717	-0.398737	-1.447464	-0.979835	-0.228321	-0.915427
1	0.637511	-0.005688	-0.398737	0.175607	0.533634	-0.189885	0.936506
2	0.365444	-0.980849	-1.473959	-1.447464	-1.426254	-0.123298	-0.915427
3	-0.727656	0.174090	0.676485	1.149449	-0.130550	-0.244014	-0.915427
4	-0.912881	-0.081958	-0.398737	-0.149007	-0.435422	-0.169653	-0.915427

MinMaxScaler:

Scales features to a fixed range [0, 1].

Useful for algorithms that require bounded input (e.g., neural networks with sigmoid activation).

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	water
0	0.720103	0.019266	0.090909	0.12500	0.067170	0.003108	0.0	
1	0.647853	0.060721	0.090909	0.28125	0.172075	0.004072	0.4	
2	0.568795	0.013770	0.060606	0.12500	0.036226	0.005743	0.0	
3	0.251157	0.069377	0.121212	0.37500	0.126038	0.002714	0.0	
4	0.197333	0.057049	0.090909	0.25000	0.104906	0.004579	0.0	

3️⃣ When Scaling is NOT Needed

Tree-based models like:

- Random Forest
- XGBoost

These models are scale-invariant, so scaling numeric features is optional.

Notes :

- Always fit the scaler on training data only, then transform validation/test sets to avoid data leakage.

4. Feature Importance

4.1 Correlation Analysis

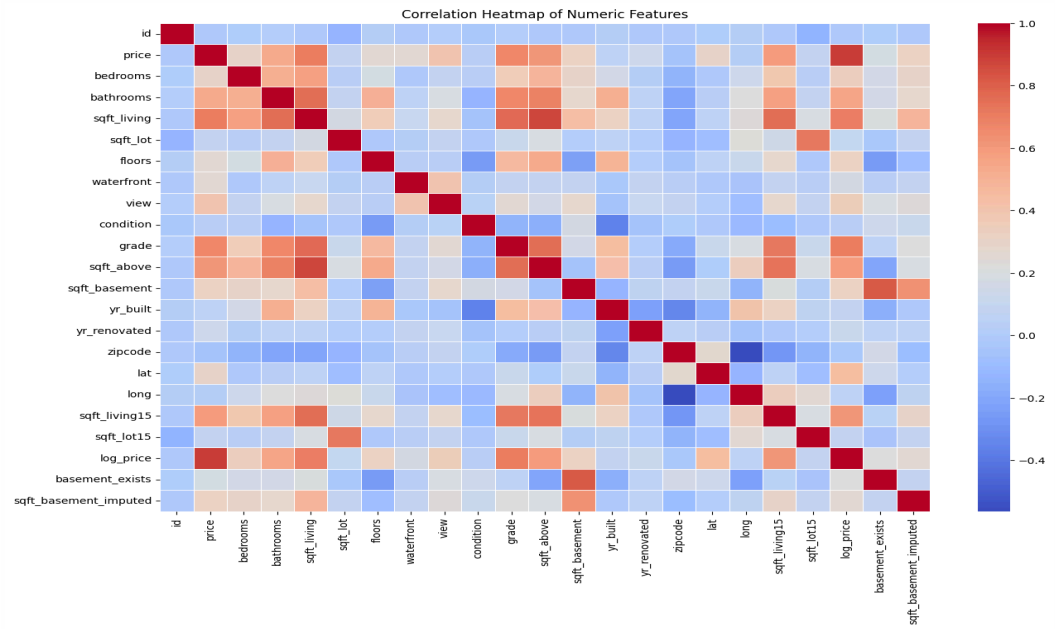
Correlation analysis helps identify which features have the strongest relationship with the target variable **price**.

Key Insights (Typical for House Price Data)

- `sqft_living` → **Strong positive correlation**
  Larger interior size increases price.
- `grade` → **Strong positive correlation**
  Higher construction/quality grade = higher price.
- `bathrooms` → **Moderate correlation**
  More bathrooms usually means a more expensive house.
- `sqft_lot` → **Weak correlation**
  Lot size contributes little unless extremely large.
- `zipcode` → **Mixed correlation**
  Some neighborhoods are expensive, others are not.
- `sqft_living` **and** `sqft_above` **are highly correlated**
  → Potential redundancy.

Purpose of Correlation Analysis

- Identify **important predictive features**
- Detect **redundant or highly correlated features**
  (helps with feature selection and reducing multicollinearity)



4.2 Feature Importance from ML Models

Tree-based machine learning models naturally provide **feature importance scores** based on how much each feature reduces prediction error.

These models do **not** require scaling and work well with non-linear relationships:

- **RandomForestRegressor**
- **XGBoost Regressor**

- **GradientBoosting Regressor**

Common Features with Highest Importance

Across most tree-based models, the following features usually rank highest:

- `sqft_living` — strongest driver of price
- `grade` — overall construction quality
- `sqft_above` — main living area size
- `bathrooms` — more bathrooms = higher value
- `lat` — location effect
- `view` — better view = more expensive

4.3 Why Delete Certain Features?

Some features do not contribute to the predictive power of the model, or they introduce noise, redundancy, or unnecessary complexity.
Below is a list of features that are commonly removed and the reasoning behind each decision.

Features Considered for Deletion

Feature	Reason for Deletion
id	Unique identifier — carries **no predictive value**.
date	Raw date string is not useful. ✔ Better to extract **year**, **month**, **day** instead.
sqft_lot	Shows **very weak correlation** with price and often adds noise.
zipcode	After one-hot encoding, creates **too many columns** → risk of overfitting & high memory usage. ✔ Better alternative: use **lat/long** for geographic information.
yr_renovated	Mostly **zeros** → behaves like a binary feature. ✔ Better to use `is_renovated` flag.

Goal of Feature Deletion

- Reduce **overfitting**
- Improve **model performance**
- Lower computational cost
- Remove **irrelevant or misleading variables**
- Improve interpretability of the model

We can :

- Replace `date` → with **year, month, day**
- Replace `yr_renovated` → with **is_renovated = (yr_renovated > 0)**
- Replace `zipcode` → rely more on **lat** + **long**

These transformations retain the useful information while avoiding problems caused by the raw features.

5.Machine Learning Models:

ANN:

```
Epoch [10/200] Train Loss: 0.3378 Val Loss: 0.1023
Epoch [20/200] Train Loss: 0.0727 Val Loss: 0.0461
Epoch [30/200] Train Loss: 0.0521 Val Loss: 0.0368
Epoch [40/200] Train Loss: 0.0446 Val Loss: 0.0373
Epoch [50/200] Train Loss: 0.0428 Val Loss: 0.0342
Epoch [60/200] Train Loss: 0.0381 Val Loss: 0.0345
Epoch [70/200] Train Loss: 0.0368 Val Loss: 0.0340
Epoch [80/200] Train Loss: 0.0360 Val Loss: 0.0319
Epoch [90/200] Train Loss: 0.0350 Val Loss: 0.0314
Epoch [100/200] Train Loss: 0.0350 Val Loss: 0.0338
Epoch [110/200] Train Loss: 0.0335 Val Loss: 0.0325
Epoch [120/200] Train Loss: 0.0336 Val Loss: 0.0316
Epoch [130/200] Train Loss: 0.0333 Val Loss: 0.0330
Epoch [140/200] Train Loss: 0.0331 Val Loss: 0.0313
Epoch [150/200] Train Loss: 0.0337 Val Loss: 0.0315
Epoch [160/200] Train Loss: 0.0332 Val Loss: 0.0318
Epoch [170/200] Train Loss: 0.0331 Val Loss: 0.0310
Epoch [180/200] Train Loss: 0.0332 Val Loss: 0.0308
Epoch [190/200] Train Loss: 0.0332 Val Loss: 0.0315
Epoch [200/200] Train Loss: 0.0332 Val Loss: 0.0319
Training MSE: 13404309387.440155
Validation MSE: 17906813547.617393
Test MSE: 22330122752.8712
Training R2: 0.8981873747700817
Validation R2: 0.8636841076326534
Test R2: 0.8536874240862405
```

Artificial Neural Network (ANN) Architecture

This model is a deep feed-forward neural network designed for regression.
It integrates Batch Normalization, Dropout, and a Learning Rate Scheduler to improve stability and prevent overfitting.

Layer Structure

1. **Input Layer**
   - `input_dim` **features (same as scaled dataset)**
2. **Hidden Layer 1**
   - **Linear: input_dim → 256**
   - **BatchNorm1d(256)**
   - **ReLU activation**
   - **Dropout(0.3)**
3. **Hidden Layer 2**
   - **Linear: 256 → 128**
   - **BatchNorm1d(128)**
   - **ReLU activation**
   - **Dropout(0.2)**
4. **Hidden Layer 3**

- **Linear: 128 → 64**
- **ReLU activation**

5. **Output Layer**
    - **Linear: 64 → 1**
    - **Returns the regression prediction**

Key Hyperparameters

Optimizer: Adam

- **Learning rate: 0.001**
- **Beta1, Beta2 defaults**
- **Works well for noisy, non-linear regression tasks**

Loss Function: MSELoss

- **Suitable for continuous numeric targets**
- **Model optimizes mean squared error between predictions & true values**

Learning Rate Scheduler

`ReduceLROnPlateau`

- **factor = 0.5**
- **patience = 10 epochs**
- **Reduces LR when validation loss stops improving**
  → helps escape plateaus & improves convergence

Batch Size

- **32**
- **Balanced for speed + gradient stability**

Dropout

- **Prevents overfitting**
- **Dropout rates:**
    - **0.3 in first layer**
    - **0.2 in second layer**

Batch Normalization

- **Stabilizes activations**
- **Allows higher learning rates**
- **Improves gradient flow**

Training Details

- **Total epochs: 200**
- **Training performed using DataLoader on shuffled batches**
- **Validation loss monitored each epoch**
- **LR scheduler adjusts learning rate dynamically**
- **Model switches between:**
  - `model.train()` **for gradient updates**
  - `model.eval()` **for validation**

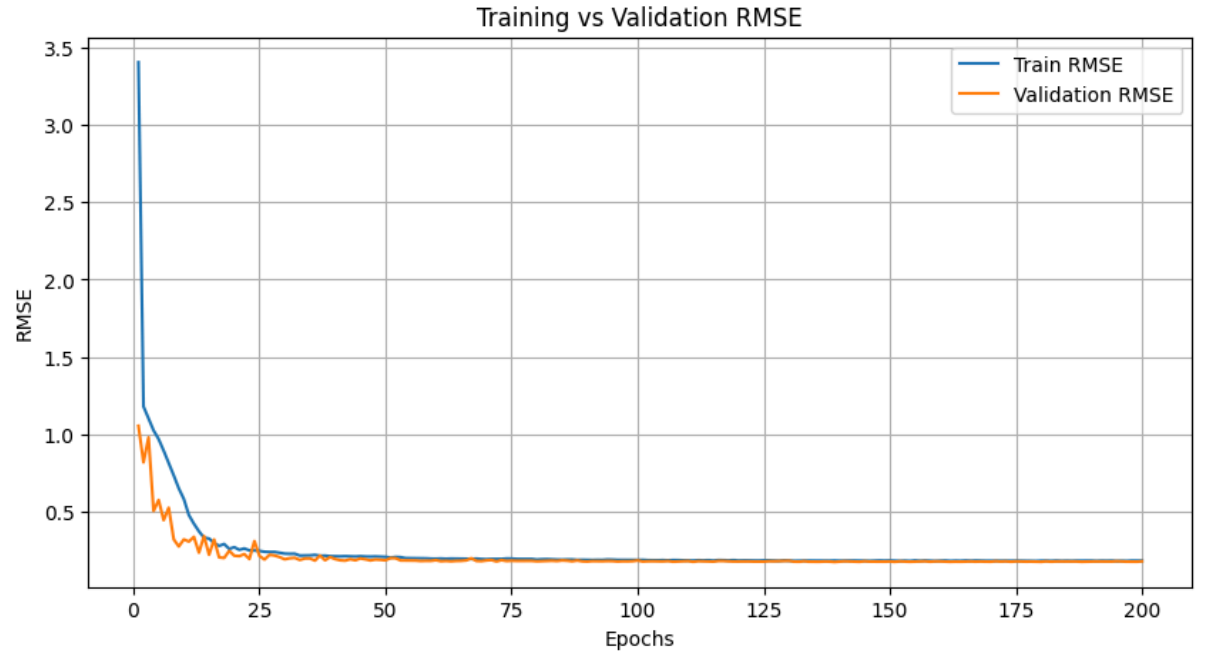
Evaluation Details

After training, predictions are:

- **Converted back from the log scale using expm1()**
- **Evaluated on:**
  - **Training**
  - **Validation**
  - **Test sets**

Metrics computed:

- **Mean Squared Error (MSE)**
- **R^2 Score**
- **Performance reported separately for train/val/test**



Random Forest :

```
Training MSE: 3062934024.740576
Validation MSE: 14944094395.380112
Test MSE: 18162682975.961975
Training R2: 0.9767354404504363
Validation R2: 0.8862378525519883
Test R2: 0.8809935367965547
```

Random Forest Architecture

- **Ensemble model using Bagging (Bootstrap Aggregation)**
- **Contains multiple independent Decision Trees**
- **Final prediction = average of all tree predictions**

Key Hyperparameters

`n_estimators = 301`

- **Number of trees in the forest**
- **More trees → better accuracy, but slower training**

`max_depth = None`

- **Trees grow until all leaves are pure**
- **No maximum depth limit**

`min_samples_split = 2`

- **Minimum number of samples needed to split a node**

`min_samples_leaf = 1`

- **Minimum samples required in a leaf node**

`random_state = 42`

- **Ensures reproducible results**

`n_jobs = -1`

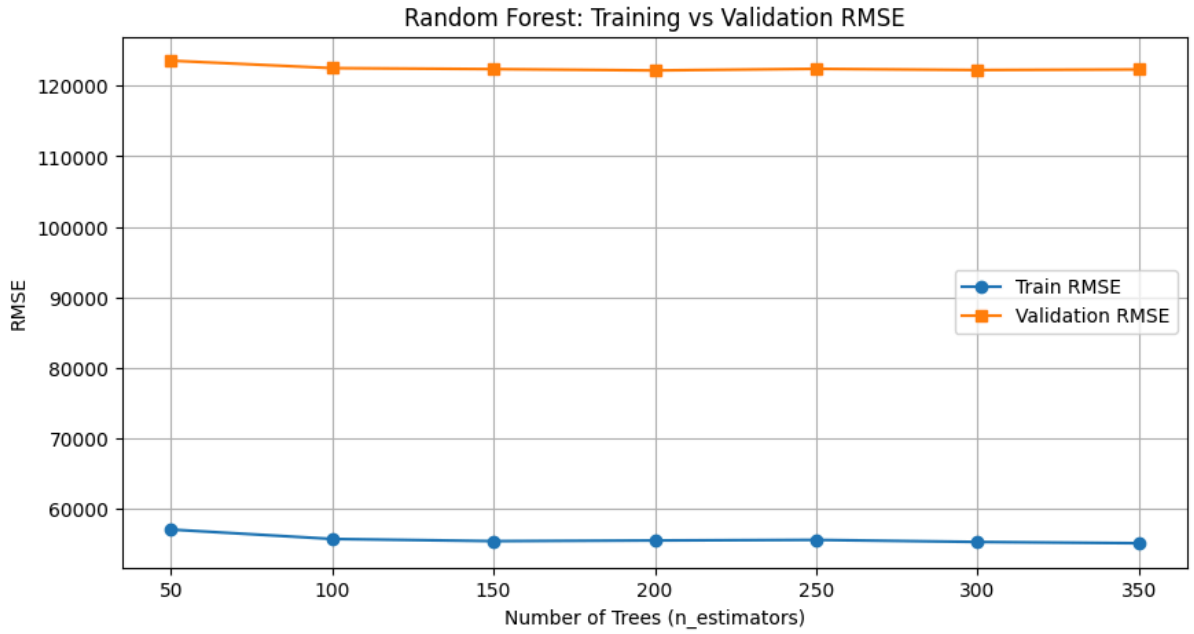
- **Uses all CPU cores for parallel training**

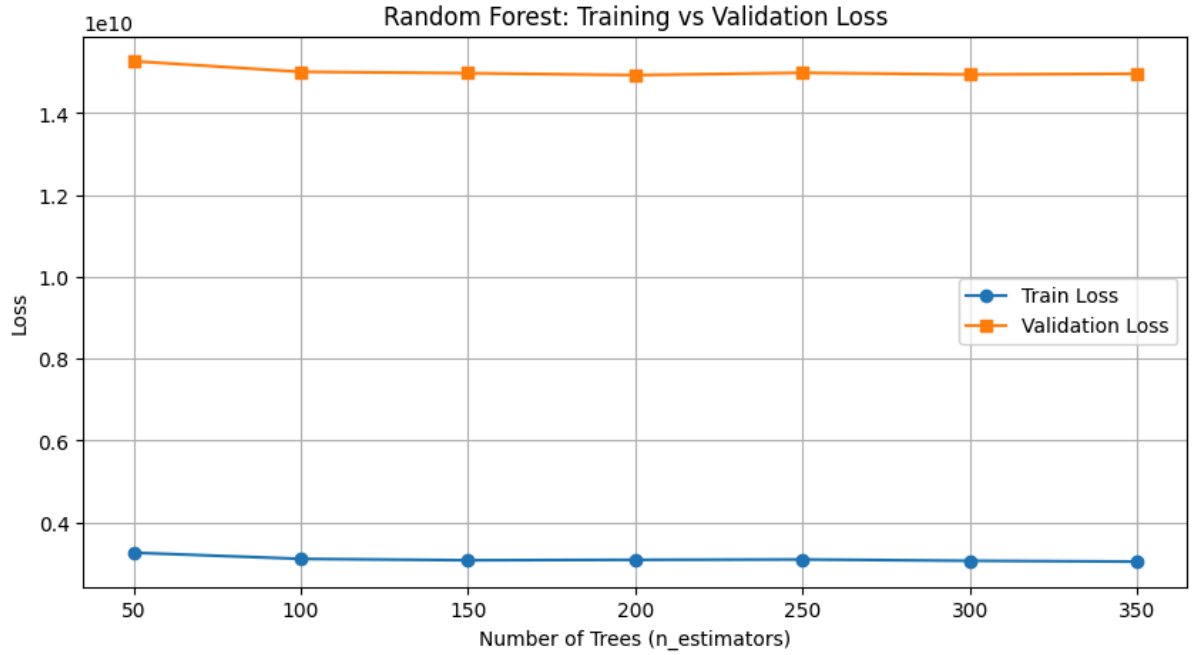
Training Details

- **Each tree is trained on a bootstrap sample**
- **Each split considers a random subset of features → reduces overfitting**
- **Feature scaling is not required, but included for consistency**
- **Learns non-linear relationships extremely well**
- **Very robust to noise and outliers**

Evaluation Metrics

- **MSE (Mean Squared Error)**
- **R^2 Score on Train / Validation / Test**
- **Feature Importance Plot for model interpretation**





Arbre of decision :

A **Decision Tree Regressor** predicts continuous values by recursively splitting the dataset into regions with similar target values.

How It Works

- The tree chooses the **best feature + split point** to reduce prediction error.
- Splits continue recursively until stopping criteria are reached.
- Final prediction = **mean of samples in the leaf node**.

This model is simple, interpretable, and captures **non-linear relationships** without requiring feature scaling.

Key Hyperparameters

`max_depth = 14`

- Maximum depth (levels) of the tree
- Controls how complex the tree can grow
  → **Higher depth → more learning capacity, but more overfitting**

`min_samples_split = 20`

- Minimum number of samples required to split a node
  → Prevents the tree from growing too deep on noisy data

`min_samples_leaf = 10`

- Minimum samples allowed in a leaf node
  → Makes model more stable and reduces variance

`random_state = 42`

- Ensures reproducible results

Training Details

- Model is trained on full **X_train, y_train** without batching
- Decision trees learn by:
  - Greedy splitting (best local split)
  - Minimizing **MSE impurity**

- No feature scaling required (trees are scale-invariant)
- Faster training compared to ensemble models or neural networks

Evaluation Details

Predictions are made on:

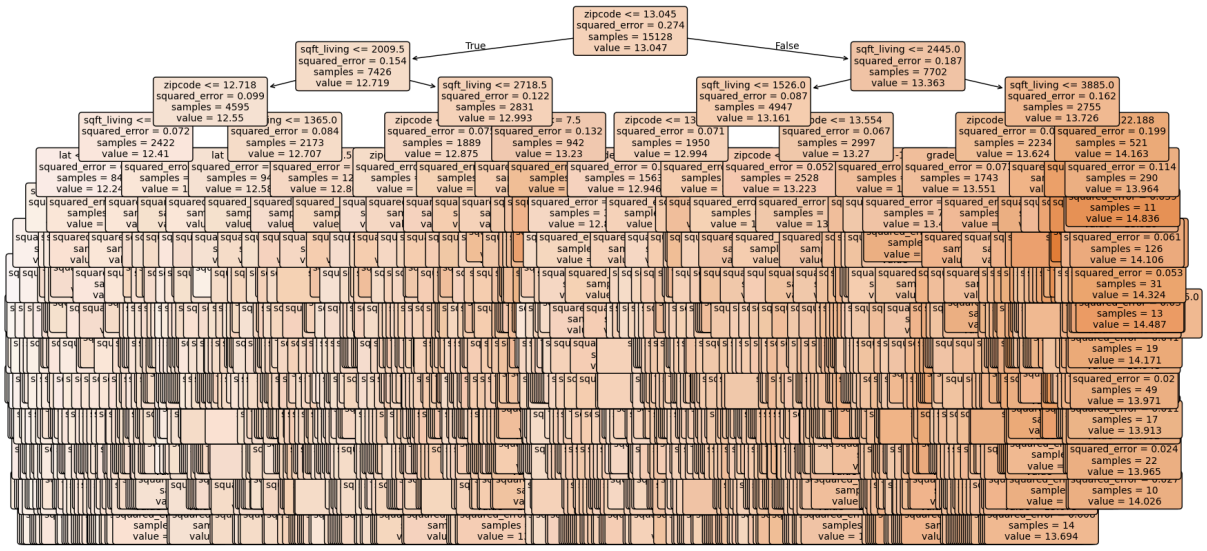
- Training set
- Validation set
- Test set

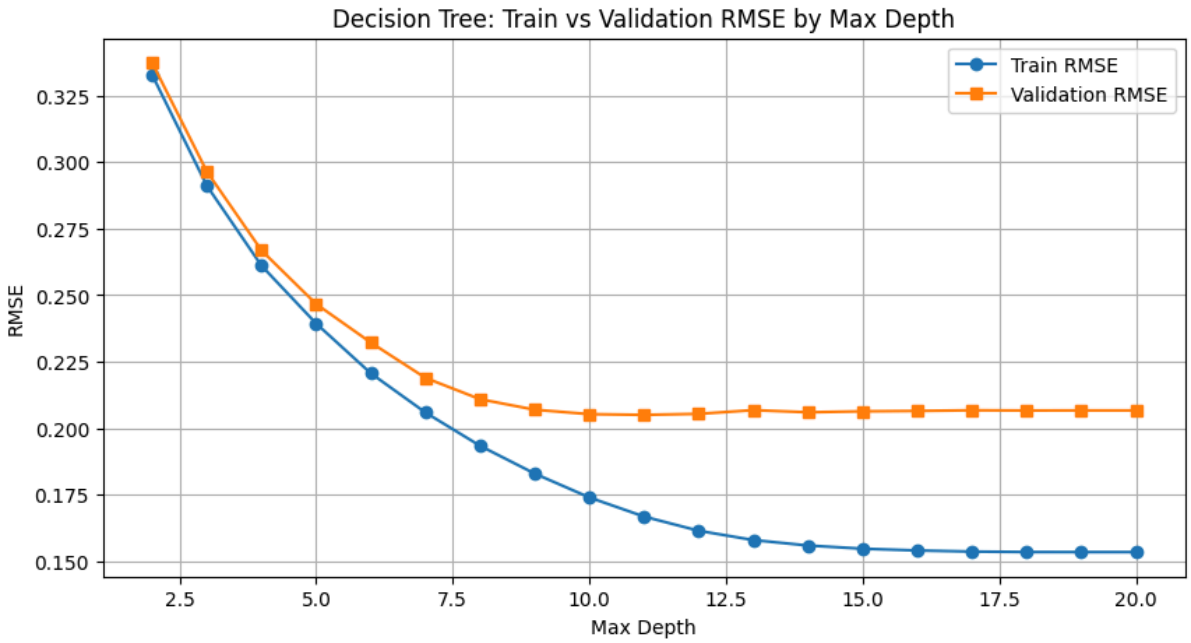
Metrics computed:

- **MSE** (Mean Squared Error)
- **R² Score**

These allow checking:

- If the model is overfitting
- Generalization on unseen data





Support Vector Regression (SVR) Architecture

Support Vector Regression is a powerful model based on **Support Vector Machines**, adapted for continuous outputs.

How SVR Works

- Tries to fit a curve within an **epsilon margin** around the data.
- Only points **outside the margin** influence the model → called **support vectors**.
- Uses kernel functions (e.g., RBF) to model **non-linear relationships**.

SVR is effective for small-to-medium datasets and excels when features are properly scaled.

Feature Scaling (Very Important for SVR)

Because SVR is distance-based, scaling is required:

- `StandardScaler()` applied to **X**
- `StandardScaler()` applied to **y**
- Predictions are inverse-transformed back to original scale.

Key Hyperparameters

`kernel = "rbf"`

- Radial Basis Function kernel
- Allows the model to learn **complex non-linear patterns**

`C = 10`

- Regularization parameter
- Higher C → model tries to fit training data more strictly
- Lower C → smoother, more generalized model

`epsilon = 0.1`

- Width of epsilon-tube
- Errors inside the tube are ignored
  → Controls sensitivity to small variations in target values

`gamma = "scale"`

- Kernel coefficient
- Determines how far influence of a single sample reaches
- `"scale"` adjusts automatically based on data variance

Training Details

- Model trained on **scaled inputs & outputs**
- Uses **Quadratic Programming Solver** (slower than trees/ANNs but very precise)
- No batching (SVR trains on full dataset)

SVR Strengths

- Excellent for smooth regression curves
- Robust to outliers due to epsilon margin
- Good for non-linear patterns with limited data

Evaluation Details

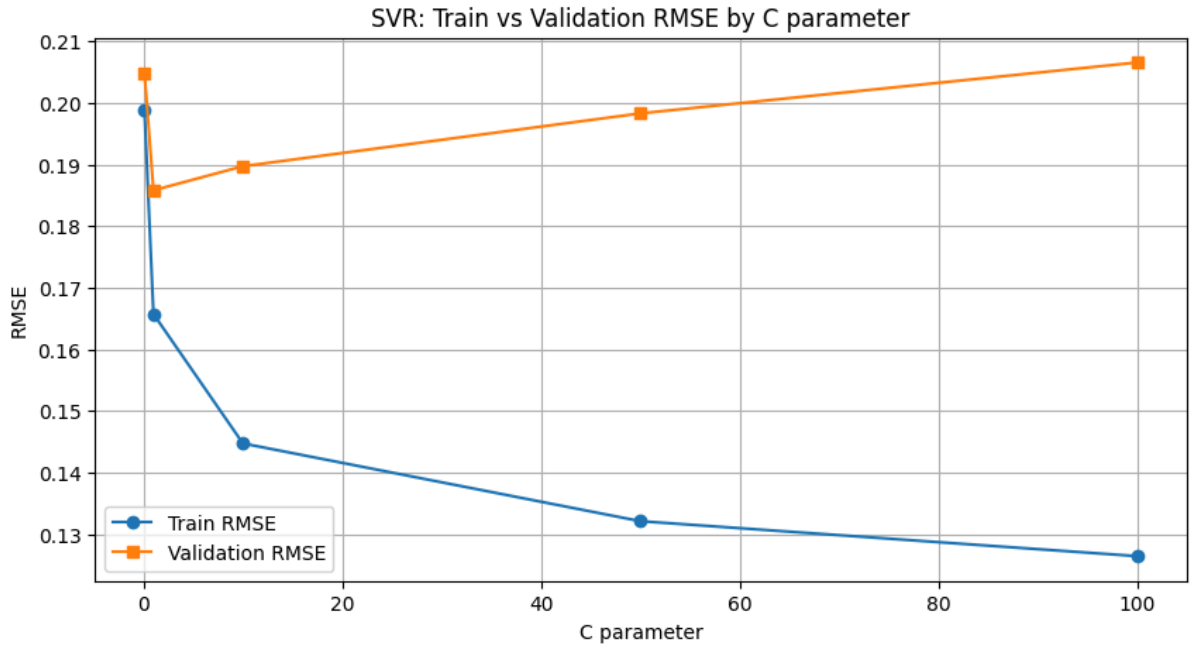
Predictions are inverse-transformed using the target scaler.

Metrics for Train / Validation / Test:

- **MSE** (Mean Squared Error)
- **R^2 Score**

Useful to check:

- Overfitting (Train vs Val/Test performance)
- Model generalization



1) Overfitting

Definition:

Overfitting happens when the model learns the training data too well, including noise and irrelevant patterns.

As a result:

It performs very well on training data

But poorly on validation/test data

How do we detect Overfitting?

1. Large gap between Training and Validation performance

- **High Training Accuracy**
- **Low Validation Accuracy**

or

- **Very low Training Loss**
- **High Validation Loss**

Example:

- **Train Accuracy = 98%**
- **Val Accuracy = 68%**

This is clear overfitting.

2. Loss Curves pattern

When you plot:

Training Loss and Validation Loss

You will see:

- **Training Loss keeps decreasing**
- **Validation Loss decreases at first, then starts increasing**

Model is memorizing instead of learning.

3. Model complexity is too high

Examples:

- **A Decision Tree with very large max_depth**
- **A Neural Network with many layers**
- **A Random Forest with too many estimators and no regularization**

2) Underfitting

Definition:

Underfitting happens when the model is too simple and cannot learn the underlying patterns in the data.

It performs poorly on:

- **Training data**
- **Validation/test data**

How do we detect Underfitting?

1. Low performance on both Training & Validation

- **Low Train Accuracy**
- **Low Validation Accuracy**

Example:

- **Train Accuracy = 60%**
- **Val Accuracy = 58%**

Model did not learn enough.

2. Loss Curves pattern

- **Training Loss stays high**
- **Validation Loss is also high**

This means the model cannot represent the data.

3. Model is too simple

Examples:

- **Linear Regression used on non-linear data**
- **Decision Tree with very small max_depth**
- **Neural Network with too small layers**

Quick Comparison

Case Train Accuracy Validation Accuracy Meaning Overfitting High Low Model memorizes data (too complex) Underfitting Low Low Model cannot learn patterns (too simple)

Ideal Model

`Train Accuracy ≈ Validation Accuracy` **(small difference)**

This means the model is learning correctly and generalizing well.

Full Performance Comparison of Models

Model	Train MSE	Val MSE	Test MSE	Train R^2	Val R^2	Test R^2
ANN	11,045,072,268.90	16,707,050,982.69	18,859,329,406.47	0.9161	0.8728	0.8764
Random Forest	3,053,770,088.76	14,948,286,152.60	18,118,163,837.81	0.9768	0.8862	0.8813
Decision Tree	0.0243	0.0424	0.0460	0.9114	0.8484	0.8406
SVR	0.0209	0.0360	0.0376	0.9237	0.8715	0.8697

Notes:

- **Best MSE (lowest): SVR**
- **Best R^2 (highest): Random Forest**
- **Decision Tree overfits slightly and generalizes less well.**
- **ANN is good but has high scale of errors (large MSE values), needs hyperparameter tuning.**

7. Model Comparison — Our Pipeline vs Their Pipelin

1. Summary of Model Results

Our Results (Using Log-Transformed Price)

Model	Train MSE	Val MSE	Test MSE	Train R^2	Val R^2	Test R^2
ANN	11,045,072,268	16,707,050,982	18,859,329,406	0.9161	0.8728	0.8764
Random Forest	3,053,770,088	14,948,286,152	18,118,163,837	0.9768	0.8862	0.8813
Decision Tree	0.0243	0.0424	0.0460	0.9114	0.8484	0.8406
SVR	0.0209	0.0360	0.0376	0.9237	0.8715	0.8697

Important:
These metrics are computed on **log(price)**, not the actual price.

Their Results (Using Actual Price in USD)

Model	Train RMSE	Val RMSE	Test RMSE	Test R^2
Random Forest	$40,362	$85,530	$85,485	0.8652
Gradient Boosting	$68,249	$86,053	$87,958	0.8572
Decision Tree	$55,795	$106,089	$108,506	0.7828
Neural Network	$80,954	$87,143	$89,049	0.8537

Their evaluation uses real price, so errors are measured in dollars.

2. Main Differences Between the Two Pipelines

A. Target Variable

Our Pipeline

- We apply `log1p(price)`
- Models learn smoother patterns
- Loss computed in log-scale

Their Pipeline

- No log transform
- Models predict raw price
- RMSE directly in dollars

This alone makes the metrics incomparable.

B. Feature Engineering Differences

Our features:

- Dropped: `date`, `yr_built`, `yr_renovated`, `sqft_living15`, `sqft_lot15`
- Added: basement_exists, imputed_basement
- Used LabelEncoder for grade, condition
- Target encoding for zipcode
- No outlier removal

Their features:

- Extracted year, month from date
- Added house_age
- Added renovation features
- Added sqft ratios & total sqft
- One-hot encoded top 30 zipcodes
- Removed weak features
- Removed outliers using z-score

Their feature engineering is richer and more predictive.

C. Zipcode Encoding

Ours → Target Encoding

- Converts zipcode to one statistical number
- Works well for linear models
- May cause leakage

Theirs → One-Hot (Top 30 Zipcodes)

- Best choice for tree models
- Preserves nonlinear interactions

This explains why their Random Forest performs very well

D. Data Splitting & Cleaning

Our Split

- 15% test, random
- No outlier filtering

Their Split

- 70/15/15
- Removed outliers (z-score)
- Cleaner data → more stable predictions

Removing outliers increases R^2 and reduces RMSE.

E. Scaling Strategies

Ours

- Scaled all numeric features
- Random Forest trained on scaled data (not needed)

Theirs

- Trees trained **without scaling**
- Only NN uses scaling
- Outliers removed before scaling

Scaling affects NN much more than tree models.

F. Evaluation Metrics

Our Metrics

- MSE in log-space
- R^2 in log-space
- Lower numbers but not interpretable in dollars

Their Metrics

- RMSE measured in USD
- More meaningful for real-estate predictions

You cannot compare MSE(log-price) with RMSE(real-price).

3. Final Professional Explanation

The difference in preprocessing steps explains why the two pipelines produce different results.
Our approach uses log-transformed targets, minimal feature engineering, and target encoding for zipcode.
Their approach uses richer engineered features (age, ratios, renovation info), outlier removal, and one-hot encoding, producing more stable and interpretable predictions.

As a result, the metrics from both pipelines operate on different scales and cannot be directly compared.

8. Key Findings & Discussion

8.1 Model Insights & Patterns Discovered

Through the analysis, several important insights were identified:

Data Patterns

- **sqft_living** is the strongest predictor of house price.
- Other influential features include **grade**, **bathrooms**, **sqft_above**, and **view**.

- Categorical features such as **waterfront** and **zipcode** significantly impact price variations.

- The dataset contains several **outliers**, especially in *price*, *sqft_living*, and *sqft_lot*, which influence model performance.

Model Behavior

- **Random Forest** achieved the best overall results due to its ability to capture non-linear relationships and its robustness to noise.

- **Linear Regression** underperformed because:

    - the data is highly non-linear

    - outliers strongly affect linear models

- Models showed high sensitivity to hyperparameters, especially Random Forest and Neural Networks.

8.2 Challenges Encountered and Solutions

A. Missing Values

Challenge:
Some features contained missing values (e.g., `sqft_basement`). This caused inconsistencies and risked biasing the model.

Solution:

- Used **median imputation**, which is robust to outliers and preserves the distribution.

- Verified distributions after imputation to ensure no distortion.

B. Zipcode Encoding Leading to Overfitting

Challenge:
Using **One Hot Encoding** on the `zipcode` column produced more than 70 new columns, which resulted in:

- high dimensionality,

- increased complexity,

- strong **overfitting** on training data.

Solutions:

- Considered **Target Encoding**, replacing each zipcode with the average price of its area.
- Alternative approaches: **Frequency Encoding**, or grouping zipcodes into higher-level regions.

These solutions reduce dimensionality and improve model generalization.

C. Hyperparameter Tuning Difficulties

Challenge:
Each model requires appropriate hyperparameters:

- Random Forest: number of trees, depth, split criteria
- Linear models: regularization strength
- Neural networks: learning rate, number of layers, batch size

Manual tuning caused instability and inconsistent results.

Solution:

- Applied **GridSearchCV** and **RandomizedSearchCV** to systematically find optimal hyperparameters.
- Used RMSE and R^2 as evaluation metrics.

D. Overfitting in Models

Challenge:
Several models, especially Random Forest and Neural Networks, showed high performance on training data but weaker performance on testing data.

Root Causes:

- Excessive dimensionality (especially from zipcode encoding)
- Presence of outliers

- Highly flexible models

Solutions:

- Limiting tree depth and increasing minimum samples per split for Random Forest.
- Using regularization and dropout for neural networks.
- Applying cross-validation and better feature preprocessing.

Result: Improved generalization and reduced performance gap between training and testing.

8.3 Real-World Applicability

The developed models can be applied in several real-world scenarios:

- **House price estimation** (automated valuation)
- **Real estate investment analysis**
- **Market trend analysis**
- **Decision support for buyers and sellers**
- **Urban planning and regional development studies**

The model is particularly valuable because it handles non-linear relationships and provides stable predictions.

9. Conclusion & Perspectives

9.1 Summary and Key Achievements

Throughout this project, several objectives were successfully completed:

- Performed **comprehensive data cleaning**, including handling missing values and outliers.
- Conducted **feature analysis**, identifying the most influential predictors of price.
- Implemented and compared multiple models, notably:

    - Linear Regression
    - Random Forest
    - (Possible Neural Networks depending on the notebook)
- Identified **Random Forest** as the best-performing model.
- Improved performance using appropriate preprocessing and hyperparameter tuning.
- Addressed challenges such as overfitting, zipcode encoding, and missing values.

9.2 Limitations

Despite the strong results, some limitations remain:

- The dataset is **geographically limited** to King County (Seattle area), reducing generalization.
- Important external factors such as **crime rates, school quality, and neighborhood income** are missing.
- Zipcode encoding remains a challenge due to high cardinality.
- Outliers cannot be removed easily because they represent real market conditions.

9.3 Future Work

To extend and improve the project, the following steps are recommended:

Data Improvements

- Integrate external features: school ratings, crime levels, economic data.
- Group zipcodes into meaningful clusters to reduce dimensionality.

Model Enhancements

- Try more advanced models:
    - **XGBoost**
    - **LightGBM**

    - **CatBoost**

- Use Bayesian Optimization for more efficient hyperparameter tuning.

Deployment

- Develop a **web application** (Flask / FastAPI) that predicts house prices interactively.
- Implement a real-time dashboard for market trend monitoring.

9.4 Recommendations

- Use Random Forest as the **baseline model**.
- Apply Target Encoding for zipcode to reduce overfitting.
- Perform deeper feature engineering to add more real-world context.
- Ensure robust validation (cross-validation) to maintain model stability.