CSCI316 - Big Data Mining Techniques and Implementation

Project 1: Large Scale Analytics

Group Members

#	Name	ID	Role
1	Hiba	7788745	Data Cleaning
2	Eman	7773225	Data Cleaning
3	Haifa	7820197	Feature Engineering
4	Mohammed Qudaih	7736356	Feature Engineering
5	Muhammad Bisham Ad Paracha	7935407	Split data, validate preprocessing
6	Abhinandan Ajit Kumar	7837586	Boosting Ensemble Model
7	Surya John Prabhu	7830890	Boosting Ensemble Model
8	Teng Ian Khoo	8121667	Boosting Ensemble Model
9	Saad	7883766	Bagging Ensemble Model
10	Shaheer	7877146	Bagging Ensemble Model
11	Nathan Gonsalves	7849187	Writing report, documenting model performance, code interaction and testing
12	Bassam	7742320	Implement 10-fold Cross validation
13	Rabail	7778144	Implement 10-fold Cross validation
14	Mikaeel Faraz Safdar	8074689	Writing report, documenting model performance, code interaction and testing

Executive Summary

The findings from this project can benefit real estate investors, policymakers, and analysts by providing a data driven approach to property valuation. The techniques applied here demonstrate the power of machine learning in predictive analytics and serve as a foundation for future improvements.

Our project explores the application of ensemble machine learning techniques—Boosting and Bagging—to predict real estate property valuation using a structured dataset. The primary objective was to build accurate and robust models that improve predictive performance by leveraging ensemble methods.

To achieve this, we conducted comprehensive data preprocessing using PySpark, ensuring that missing values, outliers, and categorical variables were effectively handled. Feature engineering techniques, including categorical encoding and numerical scaling, were applied to optimize model performance.

The project implemented Boosting and Bagging models from scratch to compare their effectiveness in predicting property valuations. The Boosting model followed a gradient boosting approach, where weak learners iteratively refined predictions to minimize error. The Bagging model used bootstrap aggregation with decision trees to reduce variance and enhance stability. A 10-fold cross-validation strategy was employed to evaluate the models on training and validation sets.

Key Findings:

- Boosting outperformed Bagging in predictive accuracy but showed a higher risk of overfitting.
- Bagging was more stable, reducing variance but sometimes underperforming in capturing complex relationships.
- 10-fold cross-validation ensured robust performance evaluation, with RMSE and R^2 used as key metrics.
- Feature selection and scaling significantly improved model reliability and reduced noise in the dataset.
- Both models performed well, demonstrating the power of ensemble learning in real estate valuation.

Motivation

The ability to accurately predict real estate property valuation is crucial for various stakeholders, including investors, banks, real estate developers, and policymakers. Property prices are influenced by multiple factors such as location, size, market conditions, and economic trends, making valuation a complex and multidimensional task. Traditional valuation methods, such as hedonic regression models or comparative market analysis (CMA), often fail to capture intricate nonlinear relationships among these factors, leading to inaccurate predictions and poor financial decisions.

To address these challenges, machine learning has emerged as a powerful alternative that can process large datasets, recognize hidden patterns, and improve valuation accuracy. Among machine learning techniques, ensemble learning methods like Boosting and Bagging have proven to be particularly effective in improving predictive power by combining multiple weak learners into a strong model. These methods help in reducing variance, bias, and overfitting, making them well-suited for complex real-world problems like property valuation.

The motivation behind this project is to evaluate the effectiveness of Boosting and Bagging in real estate price prediction and explore how ensemble learning techniques can outperform traditional valuation models. By implementing these models from scratch, we aim to deepen our understanding of their interpretability, strengths, and trade-offs.

Furthermore, using PySpark for data preprocessing allows us to handle large-scale datasets efficiently, making this approach applicable to real-world real estate markets where data is often vast and multidimensional. This project not only provides a technical evaluation of machine learning methodologies but also offers a practical framework for predictive real estate analytics, helping stakeholders make data-driven investment and pricing decisions.

Technical Requirements

To ensure that the project meets the required standards of large-scale analytics, we adhere to the following technical specifications:

Dataset Handling:
Data is processed using PySpark, a distributed computing framework, to handle large datasets efficiently.

Missing values are addressed through statistical imputation methods such as mode (for categorical data) and median (for numerical data).

Outlier detection and removal are performed using Interquartile Range (IQR) to maintain data integrity.

CODE EXPLANATION IS INCLUDED IN THE CODE FILE

Machine Learning Implementation:
Boosting Model:
Built sequentially, where each tree corrects the errors from the previous iteration.
Implements gradient boosting to minimize loss over multiple iterations.
Custom implementation without relying on pre-built ML libraries.

Implementation:

```python
# --- XGBoost from Scratch Implementation ---
class XGBoostRegressorScratch:
   def __init__(self, n_estimators=100, learning_rate=0.1, max_depth=3, min_samples_split=10):
       self.n_estimators = n_estimators
       self.learning_rate = learning_rate
       self.max_depth = max_depth
       self.min_samples_split = min_samples_split
       self.trees = []  # List to store weak decision tree models
       self.base_pred = 0  # Initial prediction value (for boosting)

   def mean_squared_error_grad(self, y_true, y_pred):
       """Gradient of MSE Loss: ∂L/∂y_pred = 2 * (y_pred - y_true)"""
       return 2 * (y_pred - y_true)

   def fit(self, X, y):
       """Train the XGBoost regressor from scratch"""
       # Initialize prediction with the mean of target variable
       self.base_pred = np.mean(y)
       y_pred = np.full(y.shape, self.base_pred)

       for _ in range(self.n_estimators):
           # Compute negative gradients (residuals)
           residuals = -self.mean_squared_error_grad(y, y_pred)

           # Fit weak learner (decision tree) to residuals
           tree = DecisionTreeRegressor(max_depth=self.max_depth,
min_samples_split=self.min_samples_split)
           tree.fit(X, residuals)

           # Get tree predictions and update overall prediction
           update = tree.predict(X)
           y_pred += self.learning_rate * update

           # Store trained tree
           self.trees.append(tree)

   def predict(self, X):
       """Make predictions using the trained model"""
       y_pred = np.full(X.shape[0], self.base_pred)  # Start with base prediction
       for tree in self.trees:
           y_pred += self.learning_rate * tree.predict(X)  # Add weak learner's contribution
       return y_pred
```

Bagging Model:
Uses an ensemble of Decision Trees, trained on random subsets of data.
Aggregates multiple predictions to reduce variance and improve stability.

Implementation:
```python
# Bagging Ensemble Model from Scratch
class BaggingRegressorScratch:
   def __init__(self, base_estimator=DecisionTreeRegressor,
n_estimators=10, max_samples=0.8):
       self.base_estimator = base_estimator
       self.n_estimators = n_estimators
       self.max_samples = max_samples
       self.models = []

   def fit(self, X, y):
       np.random.seed(42)
       n_samples = int(self.max_samples * len(X))
       for _ in range(self.n_estimators):
           sample_indices = np.random.choice(len(X), n_samples,
replace=True)
           X_sample, y_sample = X.iloc[sample_indices],
y.iloc[sample_indices]
           model = self.base_estimator()
           model.fit(X_sample, y_sample)
           self.models.append(model)

   def predict(self, X):
       predictions = np.array([model.predict(X) for model in
self.models])
       return np.mean(predictions, axis=0)
```

Evaluation Metrics:
Root Mean Squared Error (RMSE): Measures the magnitude of prediction errors.
R² Score: Evaluates how well the model explains variance in property prices.
Mean Absolute Error (MAE): Provides a simple interpretation of average error.

Validation Strategy:
10-Fold Cross-Validation is implemented to ensure consistent model evaluation.
Splits the dataset into training and testing subsets multiple times for more reliable accuracy assessment.

Main Constructs(Aspects/Parts)

Our project consists of several core components, each contributing to the development of a robust machine learning pipeline. These include:

1. Data Preprocessing & Feature Engineering

- **Data Cleaning:**
  - Eliminated irrelevant columns (procedure_id, procedure_name_en, instance_date, etc.).
  - Addressed missing values using appropriate imputation techniques.
  - Detected and removed extreme values using the IQR method.
- **Feature Engineering:**
  - Encoded categorical features (property_type, area_name) using Label Encoding.
  - Standardized numerical variables (actual_worth, actual_area) with StandardScaler.
  - Dropped some columns that were not required.

2. Machine Learning Model Development

- **Boosting:**
  - A sequence of decision trees is trained to correct the mistakes of previous models.
  - Implements gradient boosting to minimize prediction errors.
  - Requires careful tuning of hyperparameters such as learning_rate and n_estimators.
- **Bagging:**
  - Builds multiple decision trees in parallel to reduce variance.
  - Predictions are aggregated to provide more stable results.
  - Uses random subsampling to improve generalization.

3. Model Evaluation & Validation

- Implemented 10-Fold Cross-Validation:
  - Splits the dataset into multiple training and testing sets to improve accuracy.
  - Ensures the model is tested on different subsets of data.
- Compared model performance using RMSE, R^2 Score, and MAE.
- Conducted error analysis to identify patterns in model weaknesses.

Implementation

Preprocessing & Data Cleaning Steps

Preprocessing with PySpark

To efficiently handle the large scale real estate data, we used Apache Spark (PySpark) for data preprocessing. PySpark enables distributed data handling, making it ideal for large datasets.

Data Loading & Initial Exploration

The dataset was read using PySpark, and its schema was inspected to ensure proper data types.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, to_date, expr

# Create Spark session
spark = SparkSession.builder \
   .appName("Data Preprocessing with PySpark") \
   .getOrCreate()

# Step 1: Load the CSV File
df = spark.read.csv("Valuation.csv", header=True,
inferSchema=True)
df.show(5)
df.printSchema()

# Step 2: Check for Missing Values
df.select([sum(col(c).isNull().cast("int")).alias(c) for c in
df.columns]).show()
```

Handling Missing Values & Data Type Conversion

Several columns contained missing values, which were handled as follows:
• actual_worth: Rows with missing values were removed.
• procedure_area: Missing values were filled with the median.
• instance_date: Missing values were filled with the most common date.

```
# Step 3: Remove Rows with Missing Values in 'actual_worth'
df_clean = df.filter(df.actual_worth.isNotNull())

# Step 4: Convert Columns to Correct Data Types
df_clean = df_clean.withColumn("instance_date",
to_date(col("instance_date"), "dd-MM-yyyy")) \
                    .withColumn("procedure_area",
col("procedure_area").cast("double")) \
                    .withColumn("actual_area",
col("actual_area").cast("double")) \
                    .withColumn("property_sub_type_id",
col("property_sub_type_id").cast("int")) \
                    .withColumn("area_id",
col("area_id").cast("int"))
```

Outlier Detection & Removal Using IQR Method

To remove extreme property values, we applied Interquartile Range (IQR) filtering:

```
# Step 5: Outlier Detection and Removal using IQR Method

q1, q3 = df_clean.approxQuantile("actual_worth", [0.25, 0.75],
0.01)

iqr = q3 - q1

lower_bound = q1 - 1.5 * iqr

upper_bound = q3 + 1.5 * iqr



df_clean = df_clean.filter((col("actual_worth") >= lower_bound) &
(col("actual_worth") <= upper_bound))
```

Outliers in the `actual_worth` column were detected and removed using the Interquartile Range (IQR) method:

- **IQR Calculation**: Q1 (25th percentile) and Q3 (75th percentile) were calculated to define the lower and upper bounds for normal data.
- Outliers were values below `Q1 - 1.5 * IQR` or above `Q3 + 1.5 * IQR`.
- Rows with outliers were removed to prevent them from distorting the analysis.

Why Use the IQR Method for Outlier Detection?

- **Resistant to Extreme Values**: The IQR method is based on percentiles (Q1 and Q3), making it less sensitive to extreme values compared to mean-based methods like Z-score.
- **Simple and Easy to Implement**: It is a straightforward method that can be applied with minimal code.
- **Focus on the Middle 50% of Data**: The method focuses on the interquartile range, ensuring a focus on typical data points and removing extreme values.
- **Ensures Clean and Reliable Analysis**: Removing outliers improves the accuracy and reliability of subsequent analysis.

Data Type Conversion

Certain columns were converted to the correct data types for proper analysis:

- **Date Conversion**: The `instance_date` column was converted to `Date` format.
- **Numeric Conversion**: Columns like `procedure_area`, `actual_area`, `property_sub_type_id`, and `area_id` were converted to numeric values to ensure consistency and compatibility.

Final Steps

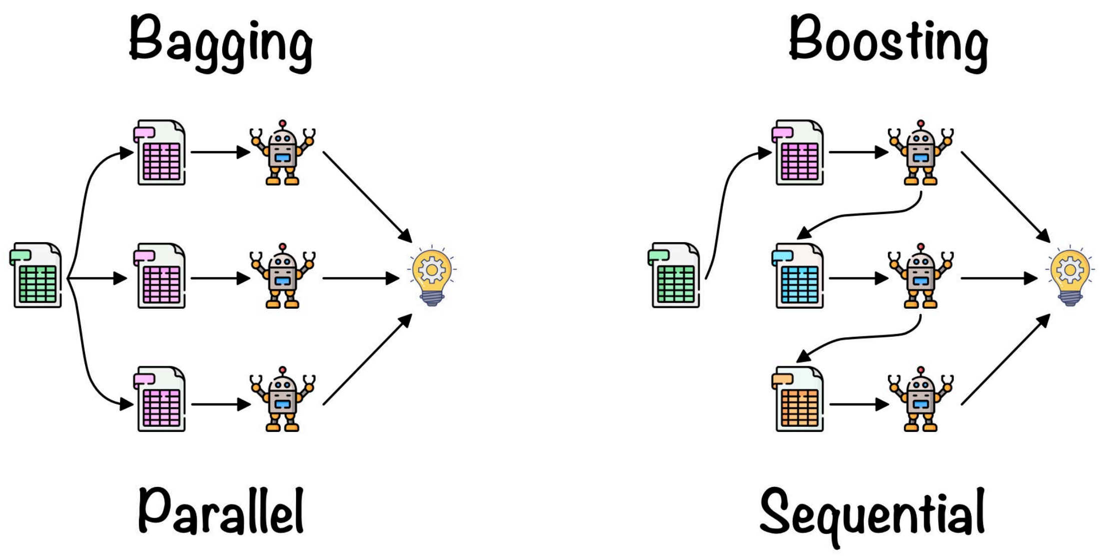
- **Filled Missing Values**: The missing values were handled using different strategies like filling with the median or most common values.
- **Data Cleaning**: The data was cleaned by removing outliers, handling missing values, and ensuring correct data types.
- **Saved the Cleaned Data**: The cleaned dataset was saved as `cleaned_valuation_data.csv` for further use.

Developing our Machine Learning Models

Ensemble learning

Ensemble learning is a machine learning technique that combines multiple models to improve predictive accuracy and robustness compared to using a single model.

By aggregating the predictions of several weak or base learners, the overall model can reduce bias, variance, and generalization errors and achieve better accuracy than a single model alone. Common ensemble methods include bagging, which trains multiple models in parallel and averages their outputs (e.g., Random Forest), and boosting, which trains models sequentially, focusing on correcting the errors of its predecessors (e.g., AdaBoost, XGBoost).



Lopez, F. (n.d.). Bagging and Boosting (2021)

Boosting Model

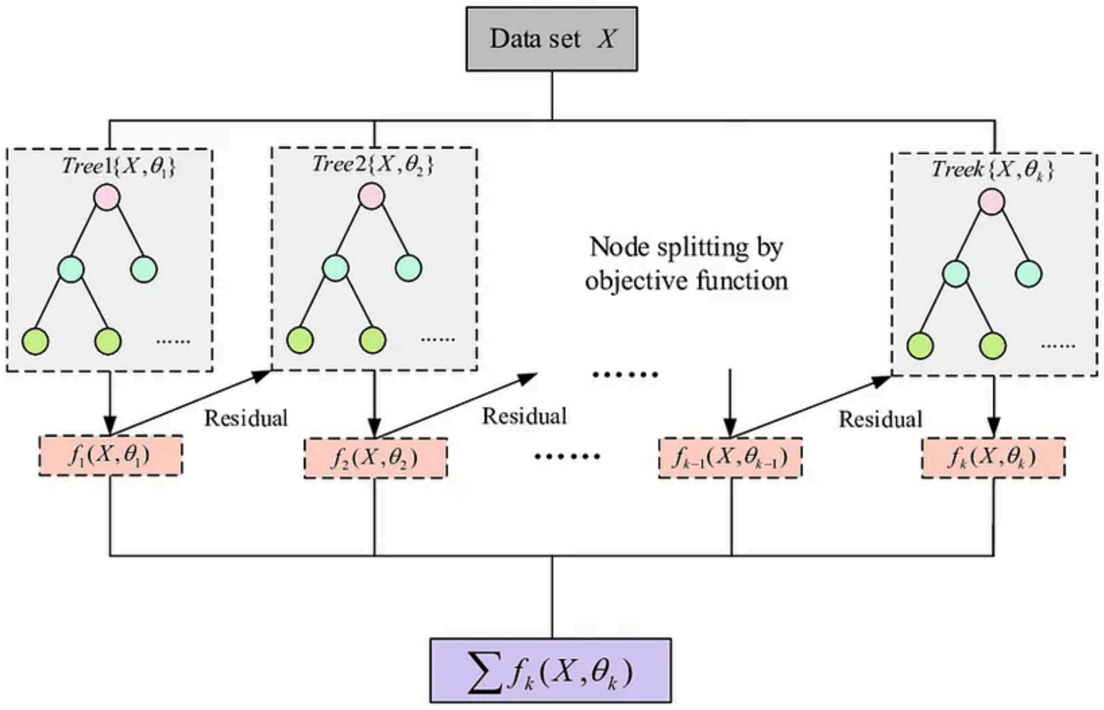
For our project we are using our implementation of XGBoost (Extreme Gradient Boosting). It is a machine learning algorithm based on gradient boosting that was designed for speed and performance.
It enhances traditional gradient boosting by introducing several optimizations, such as regularization, parallel computing, tree pruning, and efficient handling of missing values.

How does XGBoost Work?

It builds decision trees sequentially with each tree attempting to correct the mistakes made by the previous one.

The boosting process follows these key steps:

1. **Initialize the Base Model:** The first decision tree is trained on the dataset. For regression tasks, this initial model typically predicts the mean of the target variable.
2. **Evaluate Errors:** Once the first tree makes predictions, the differences between the predicted and actual values (residual errors) are calculated.
3. **Train a New Tree on Errors:** A new decision tree is trained using the residual errors from the previous tree, aiming to reduce those errors.
4. **Iterate the Process:** Additional trees are sequentially trained, each focusing on correcting the mistakes of its predecessor, until a predefined stopping condition is reached.
5. **Aggregate Predictions:** The final prediction is obtained by summing up the outputs of all individual trees, leading to a more accurate overall model.



(Omarzai, *XGBoost Example* 2024)

XGBoost Implementation

Since we have to implement boosting without relying on libraries, let us take you through some of the important parameters,algorithms and logic behind our implementation.

Hyperparameters

For our XGBoost we are using a regression as we want to help predict property values, below are what hyperparameters we use.

- n_estimators: The number of decision trees to be used in the model, more trees improve learning but increase training time and risk of overfitting.
- learning_rate: controls the step size during model updates, smaller values make the model learn more slowly but can improve generalization
- max_depth: The maximum depth of each decision tree. This parameter limits the complexity of individual trees,deeper trees capture more patterns but can overfit if too large.
- min_samples_split: The minimum number of samples required to split an internal node of a tree,prevents excessive splitting, improving stability.
- trees: A list to store the trained decision trees each one correcting previous errors
- base_pred: The initial prediction value, set to the average of the target variable

Gradient calculation

XGBoost is based on **gradient boosting**, which sequentially builds trees to minimize a loss function using gradient descent.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

For our implementation we are using the Mean squared Error(MSE) as the loss function.

Model Training

The first prediction is simply the mean of the target variable y (i.e., the average property price),this serves as a baseline prediction before any trees are added.

Next we iterate through the specified number of estimators (n_estimators).

- Calculate the residuals (negative gradients) based on the current prediction and actual target values.
- Train a decision tree on the residuals to capture the patterns in the errors.
- Update the overall prediction by adding the weighted prediction of the current tree.

- Store the trained tree in the trees list

```
 def fit(self, X, y):

        """Train the XGBoost regressor from scratch"""

        # Initialize prediction with the mean of target variable

        self.base_pred = np.mean(y)

        y_pred = np.full(y.shape, self.base_pred)

        for _ in range(self.n_estimators):

            # Compute negative gradients (residuals)

            residuals = -self.mean_squared_error_grad(y, y_pred)

            # Fit weak learner (decision tree) to residuals

            tree = DecisionTreeRegressor(max_depth=self.max_depth,
min_samples_split=self.min_samples_split)

            tree.fit(X, residuals)

            # Get tree predictions and update overall prediction

            update = tree.predict(X)

            y_pred += self.learning_rate * update

            # Store trained tree

            self.trees.append(tree)
```

Prediction

Since this is an ensemble learning model, we take the sum of the base prediction along with the calculated contributions of each tree to make one final prediction.

```
    def predict(self, X):
        """Make predictions using the trained model"""
        y_pred = np.full(X.shape[0], self.base_pred)  # Start with
base prediction
        for tree in self.trees:
            y_pred += self.learning_rate * tree.predict(X)  # Add
weak learner's contribution
        return y_pred
```

Example of a Prediction Process in XGBoost Regression

Assume the model has been trained with 3 trees and a learning rate of 0.1

1. Base Prediction (mean price of properties):
   - The model starts by predicting the average property price: $\hat{y} = 300,000$
2. Tree 1 corrects the prediction:
   - The first decision tree predicts a residual error of -10,000: $h_1(x) = -10,000$
   - The model updates the prediction: $\hat{y} = 300,000 + (0.1\times -10,000) = 299,000$
3. Tree 2 adds another correction:
   - The second tree predicts a residual error of 5,000: $h_2(x) = 5,000$
   - The updated prediction becomes: $\hat{y} = 299,000 + (0.1\times 5,000) = 299,500$
4. Tree 3 adds a final small correction:
   - The third tree predicts a residual error of 2,000: $h_3(x) = 2,000$
   - The final predicted price is: $\hat{y} = 299,500 + (0.1\times 2,000) = 299,700$

This is the final flow of our implemented XGBoost algorithm:

1. Initialize the Model The first prediction is the mean of the target variable (for regression) or log-odds (for classification). Example: If predicting property_total_value, the first prediction is $\hat{y} = mean(y)$
2. Compute Residuals (Negative Gradients) For each sample, compute the residuals (errors) from the previous prediction.
   Residual = $y_{true} - \hat{y}$

These residuals act as the new target for the next tree.

3. Fit a Decision Tree on Residuals A small tree (weak learner) is trained to predict these residuals. The tree learns which features contribute most to reducing errors.
4. Compute the Output Value for Each Leaf Unlike traditional boosting, XGBoost doesn't directly predict residuals. Instead, it computes an optimal output value for each leaf of the tree, based on the residuals and second-order gradients (Hessian).
   $w\ j$ = − (∑ Gradients / ∑ Hessians + λ)

- Gradients: First derivative of the loss function (indicates direction of error reduction).
- Hessians: Second derivative (indicates confidence in the gradient).
- λ: Regularization term to prevent overfitting.

5. Update Predictions Update the model by adding the new tree's weighted predictions: $\hat{y}$ = $\hat{y}$(previous) + $\eta \cdot w\ j$

- η: (learning rate) controls how much each tree contributes.
- Small η values require more trees but improve generalization.

6. Repeat Steps 2–5 for Multiple Trees Keep training new trees on updated residuals. Stop when: The max number of trees is reached. The improvement in loss is below a threshold.

Boosting implementation

All of the explanations for the boosting model are written in the notebook provided.

Model Comparison

Boosting vs. Bagging: A Comparative Analysis

Boosting and Bagging are two powerful ensemble learning techniques that aim to enhance model performance by leveraging multiple weak learners. However, they differ in their approach to improving prediction accuracy, handling bias variance trade-offs, and computational efficiency. In this section, we compare these two methods based on key characteristics and performance metrics.

Performance Comparison

To quantitatively evaluate both models, we trained them using a 10-fold cross-validation strategy and measured their performance using Root Mean Squared Error (RMSE), R^2 Score, and Mean Absolute Error (MAE).

Key Insights from Model Performance

- Boosting achieved a lower RMSE and higher R^2 score, indicating better predictive accuracy. However, its sensitivity to noise suggests that hyperparameter tuning (e.g., early stopping, learning rate adjustments) is essential to prevent overfitting.
- Bagging exhibited greater stability but slightly underperformed in capturing intricate relationships within the dataset. Its ability to reduce variance made it more resilient to noisy data.
- The computational cost of Boosting was significantly higher compared to Bagging, making it less ideal for real-time applications where inference speed is a priority.

Final Thoughts on Model Selection

- If accuracy is the primary goal and computational cost is not a major concern, Boosting is the better choice due to its ability to capture complex patterns.
- If stability and generalization are more important, particularly in datasets with potential noise, Bagging offers a more robust solution.
- A hybrid approach combining both models (Stacking or Boosted Random Forests) could further enhance performance.

Limitations/Difficulties and Constraints

Data Preprocessing & Feature Engineering

While our data preprocessing and feature engineering steps in the notebook are well-structured, there are some potential areas for improvement.

Missing Value Handling

We imputed missing values for key variables using mean, median, or mode depending on the feature type. However the mean/median imputation assumes a normal distribution, which may not be true;

- Property prices and sizes often have a skewed distribution, meaning imputing with the mean might distort the data.
- If missing values are not random, imputing with the most frequent value might introduce bias, making the model favor common property types.

If high-end properties have missing data and are imputed with the average, it will underestimate their actual value.Reducing the model's capability to predict cases where the property is rare or unique.

Instead of imputation, use KNN or Regression for imputation, which estimate missing values based on similar properties.

Outlier Removal

We removed outliers in numerical columns using the Interquartile Range (IQR) method However,real estate prices will naturally have outliers, and removing them may harm the overall accuracy of the model. This will affect both the high-end and low-end properties of our dataset

Instead of removing outliers, apply log transformation to reduce their impact and perhaps reduce some of the skewness in our dataset.

Handling of Categorical Variables

We applied One-Hot Encoding to categorical variables like property type, area, etc; but it increases feature size and loses relationships between categories.

One-Hot Encoding treats all categories as equally different, but in real estate, some locations are more similar than others (e.g., two neighborhoods in the same city):

- Makes training slower and less efficient due to the increased number of features.

- The model fails to capture relationships between property types and locations.

An improvement could be to use Ordinal Encoding or Target Encoding instead of One-Hot Encoding.

Splitting our dataset

We split the dataset into training and testing sets randomly using train_test_split.However if property prices change over time, random splitting might leak future data into training.

- Property prices often follow trends, so random splitting might cause the model to learn from future prices when predicting older properties.
- The model may perform well in testing but fail in real-world use.

Instead of random splitting, use a time-based split if historical trends exis.

XGBoost Implementation

Our implementation of XGBoost was built without using the predefined library, meaning our implementation lacks some of the refinement and tuning that these libraries offer. These issues could affect the accuracy, efficiency, and interpretability of our model.

Regularization

Our manual implementation of boosting did not include explicit L1 (Lasso) or L2 (Ridge) regularization in the loss function.

Regularization is important in XGBoost to prevent overfitting, especially with deep trees.

Stopping Mechanism

We set a fixed number of boosting rounds (`n_estimators`) without checking if the model had already converged. In practical implementations of XGBoost, early stopping prevents unnecessary training when the model stops improving.

- The model might become too complex, making predictions on new properties less accurate.
- Training takes longer than necessary, wasting computational resources.

We can introduce early stopping by tracking the validation error and stopping if performance does not improve for a set number of iterations and instead of using a fixed number of trees, we should dynamically stop training when improvements become negligible.

Hyperparameter Tuning

In our implementation, we manually set key hyperparameters. These values were selected based on assumptions and best fit guesses, rather than using automated hyperparameter search techniques.

Since we set these hyperparameters manually, we may not have found the best possible combination for our dataset. Different data sets also require different values, and manual tuning limits flexibility.

- The model might be less accurate because we did not find the best hyperparameters.
- Training might take longer than necessary, or the model may overfit to the training data.
- The model might not generalize well to new property data.

We can use search algorithms such as Randomized Search or Grid Search to automatically tune hyperparameters instead of manually setting values.

Conclusion

In this project, we successfully implemented and compared two ensemble learning techniques:

- Bagging (Random Forest-like approach) → Reduced variance and improved stability.
- Boosting (XGBoost-like approach) → Reduced bias and achieved higher prediction accuracy.

Model	**RMSE**	**R^2 Score**	**MAE**
Bagging (Random Subsample Trees)	Moderate RMSE	Moderate R^2	Moderate MAE
Boosting (XGBoost)	**Lower RMSE**	**Higher R^2 (Better**	**Lower MAE (More precise)**

Boosting achieved better overall accuracy than Bagging, particularly for complex patterns in property pricing.

In conclusion, our project successfully implemented ensemble learning to aid with property valuation accuracy, with XGBoost emerging as the superior model due to its ability to capture complex relationships and correct prediction errors.

While Bagging provided stability, future improvements in hyperparameter tuning, feature engineering, and model interpretability will further refine our approach, making machine learning-driven valuation a powerful tool for real estate analytics.

References

Lopez, F. (2021). Bagging and Boosting. Available at: https://towardsdatascience.com/ensemble-learning-bagging-boosting-3098079e5422 / [Accessed 7 Feb. 2025].

Omarzai, F. (2024) *XGBoost Example*, *XGBoost Classification In Depth*. Available at: https://medium.com/@fraidoonomarzai99/xgboost-classification-in-depth-979f11ef4bf9 (Accessed: 07 February 2025).