Report on Housing Price Prediction Project

Group 8

Khuc Ngoc Nam

DS-AI 03 / 20230086

Le Trung Dung

DS-AI 03 / 20235489

Nguyen Vuong Trung Hieu

DS-AI 03 / 20235496

Nguyen Hai Duong

DS-AI 03 / 20235492

Thanh Huu Dat

ICT 01 / 20235909

Class: Introduction to AI - 2024.1/152630

Professor: Le Thanh Huong

Date: December 3rd, 2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

1.1 Problem Statement

In the real estate industry, investors must make informed decisions to prevent drastic losses and maximize their profit margins. However, the fluctuating real estate market, economic factors, and changing consumer demand make price prediction a complex and challenging task.

This research aims to develop an AI-based system capable of accurately predicting house prices based on factors such as location, size, number of bedrooms, and macroeconomic indicators, in order to help investors make the decisions mentioned above, thus maximizing their profits.

1.2 Background and Problems of Research

In recent years, the rapid development of AI technologies, particularly machine learning models, has opened up new opportunities for addressing complex problems, including real estate price prediction. Traditionally, price prediction methods relied heavily on expert knowledge and manual analysis. These methods, however, could not process large volumes of data and could not fully account for all factors affecting house prices. AI models, especially deep learning and machine learning can leverage and analyze data from various sources to generate more accurate predictions. Despite this, challenges remain, such as selecting the right input features and addressing market volatility.

1.3 Research Objectives and Conceptual Framework

1.3.1 Research Objectives

This research aims to build an AI model capable of accurately predicting house prices based on input factors such as location, size, number of bedrooms, year constructed, and economic indicators. Specifically, this research will focus on the following objectives:

- Analyze the factors influencing house prices and identify the most significant features.
- Develop a machine learning model that predicts house prices based on the collected data.
- Evaluate the model's accuracy and compare it with traditional prediction methods.
- Propose improvements for the AI model to enhance its accuracy in future house price predictions.

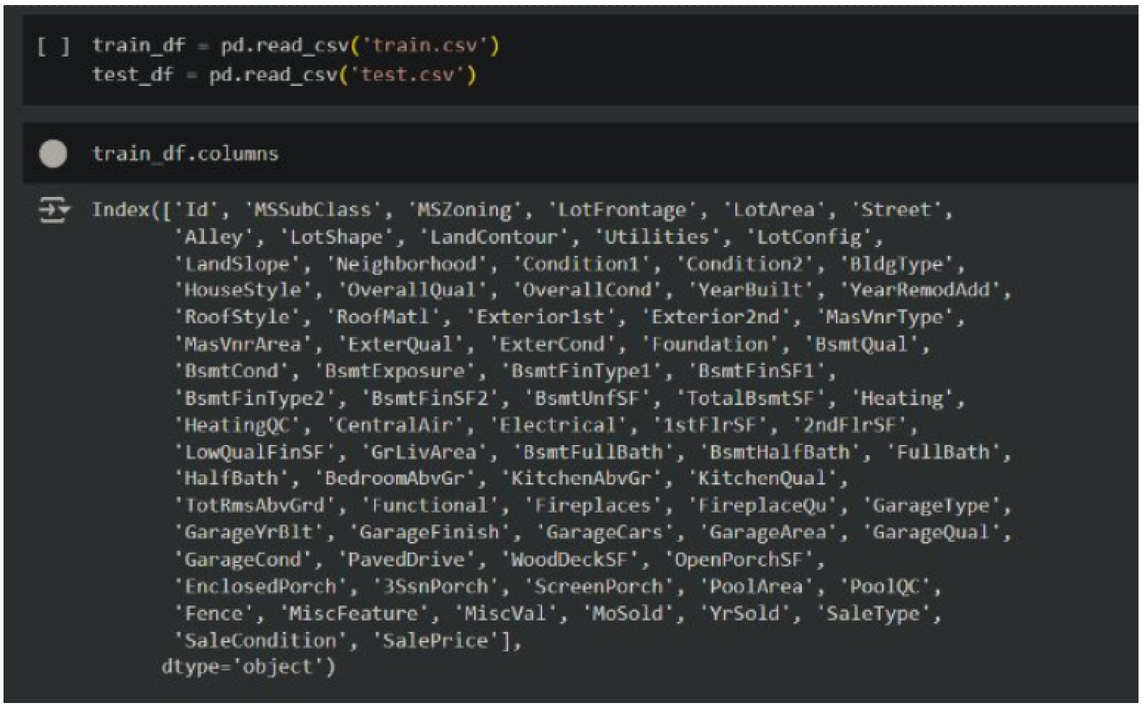
1.3.2 Conceptual Framework

This research will apply machine learning techniques to develop the house price prediction model. The input factors will include information about the house (size, number of rooms, age), geographic location (district, city, region), and economic indicators such as GDP growth rate, unemployment rate, and interest rates. Machine learning models, such as linear regression, decision trees, and neural networks, will be used to build the predictive model. The research will also evaluate different models to identify the best-performing one based on the available data and research goals.

2.1 Introduction to dataset

The Ames Housing dataset, compiled by Dean De Cock, is a leading dataset in the field of machine learning and data analysis. Often used for regression tasks, particularly for predicting housing prices, it is a modernized and expanded version of the often-cited Boston Housing dataset.

The dataset comprises various columns describing the house's properties, such as the sale price, building class, construction year, room configuration, and many other details.



```
[ ] train_df = pd.read_csv('train.csv')
    test_df = pd.read_csv('test.csv')
```

```
train_df.columns
```

```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

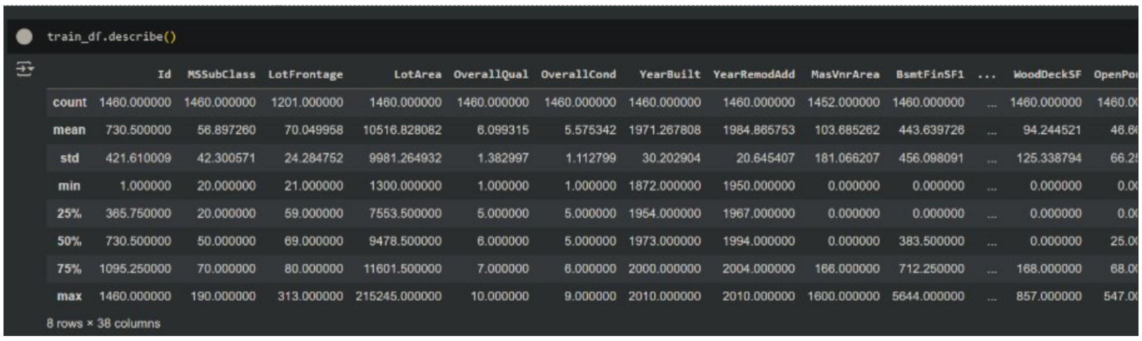
The dataset contained 2,919 entries, of which 1,460 was used for training and 1,459 was used as the test set to test the model's performance on unseen prices.

2.2 Data pre-processing

The following steps were taken to pre-process the data before use:

a, Data cleaning

- The dataset is first described to obtain the number of entries, means, standard deviations, minimum value, and maximum values at 25%, 50%, 75%, and 100% percentiles. This step helps to catch any invalid or missing values.

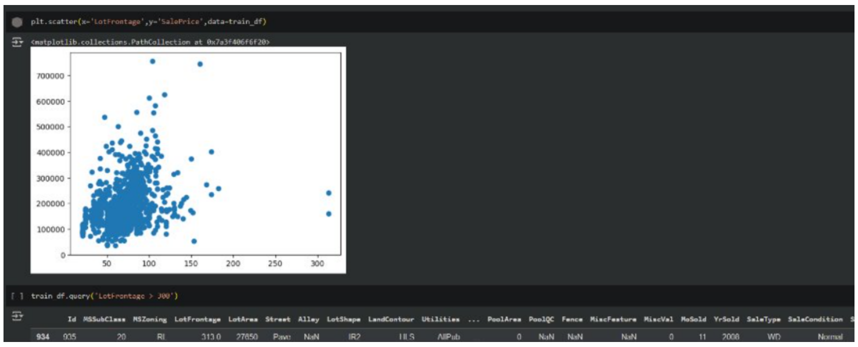


```
train_df.describe()
```

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	...	WoodDeckSF	OpenPo
count	1460.000000	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000	1452.000000	1460.000000	...	1460.000000	1460.0
mean	730.500000	56.897260	70.049958	10516.828082	6.099315	5.575342	1971.267808	1984.865753	103.685262	443.639726	...	94.244521	46.6
std	421.610009	42.300571	24.284752	9981.264932	1.382997	1.112799	30.202904	20.645407	181.066207	456.098091	...	125.338794	66.2
min	1.000000	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.000000	1950.000000	0.000000	0.000000	...	0.000000	0.0
25%	365.750000	20.000000	59.000000	7553.500000	5.000000	5.000000	1954.000000	1967.000000	0.000000	0.000000	...	0.000000	0.0
50%	730.500000	50.000000	69.000000	9478.500000	6.000000	5.000000	1973.000000	1994.000000	0.000000	383.500000	...	0.000000	25.0
75%	1095.250000	70.000000	80.000000	11601.500000	7.000000	6.000000	2000.000000	2004.000000	166.000000	712.250000	...	168.000000	68.0
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.000000	2010.000000	1600.000000	5644.000000	...	857.000000	547.0

8 rows × 38 columns

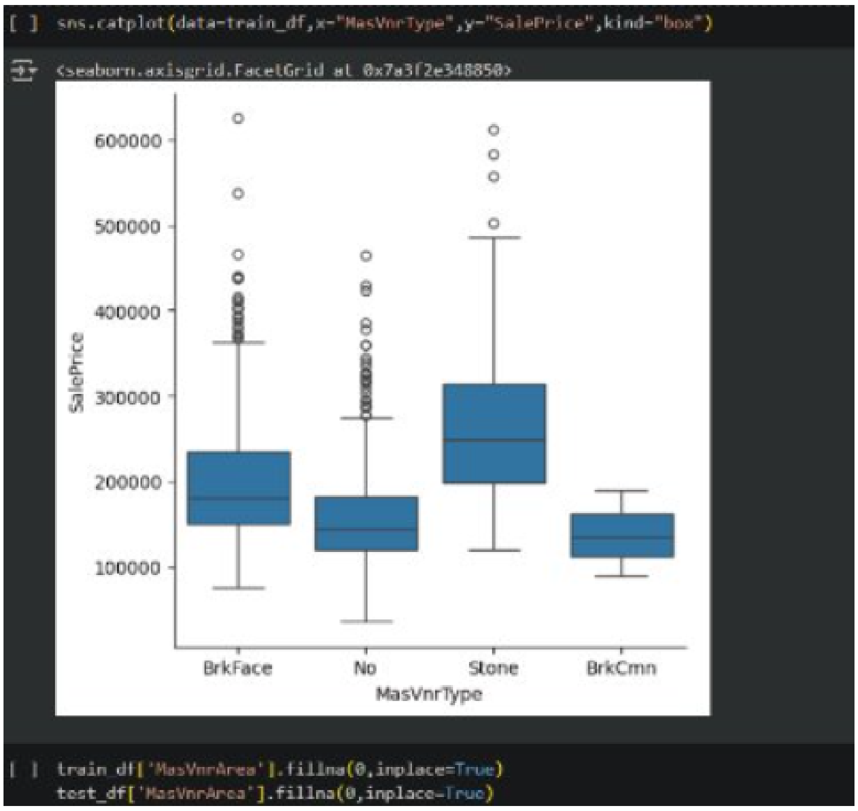
- Outliers and noise are cleaned out for numerical and categorical features. Missing values are added at this stage; features with too many missing values are excluded altogether.



```
plt.scatter(x='LotFrontage',y='SalePrice',data=train_df)
<matplotlib.collections.PathCollection at 0x7a3f406f6f20>
```

```
[ ] train df.query('LotFrontage > 300')
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoSold	YrSold	SaleType	SaleCondition	S
934	935	20	RL	313.0	27650	Pave	NaN	IR2	HLS	AllPub	...	0	NaN	NaN	NaN	0	11	2008	WD	Normal	

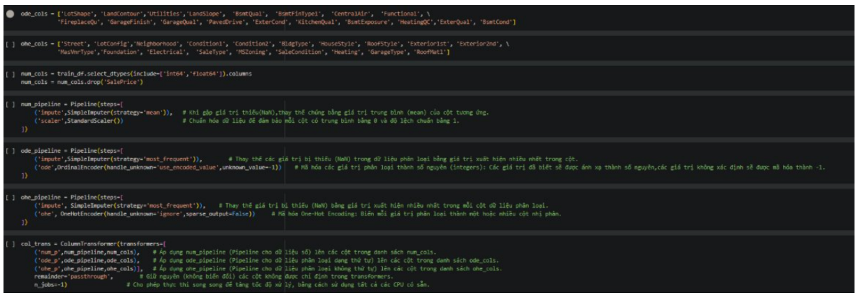


```
[ ] sns.catplot(data=train_df,x="MasVnrType",y="SalePrice",kind="box")
<seaborn.axisgrid.FacetGrid at 0x7a3f2e348850>
```

```
[ ] train_df['MasVnrArea'].fillna(0,inplace=True)
    test_df['MasVnrArea'].fillna(0,inplace=True)
```

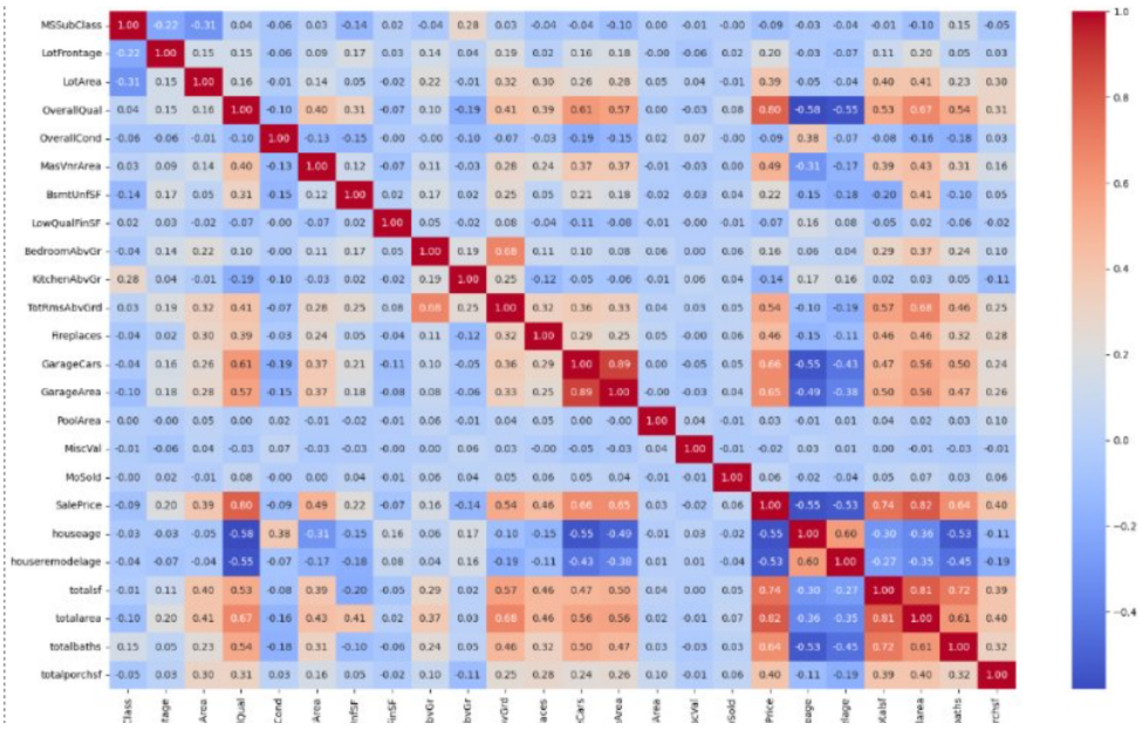
b, Data remodeling

- Some columns are replaced by a single column describing the relation between all of them. For example, a "TotalBath" column replaces all columns that describe the number of baths in different sections of the house.

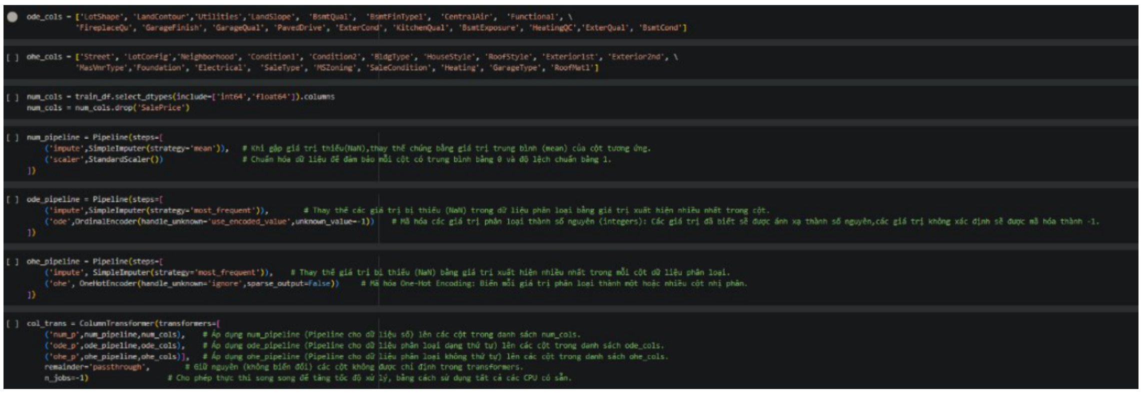


```
ode_cols = ['LotShape', 'LandContour','Utilities','LandSlope',  'BsmtQual',  'BsmtFinType1',  'CentralAir',  'Functional', \
           'FireplaceQu', 'GarageFinish', 'GarageQual', 'PavedDrive', 'ExterCond', 'KitchenQual', 'BsmtExposure', 'HeatingQC','ExterQual', 'BsmtCond']

[ ] ohe_cols = ['Street', 'LotConfig','Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'Exterior1st', 'Exterior2nd', \
           'MasVnrType','Foundation', 'Electrical',  'SaleType', 'MSZoning', 'SaleCondition', 'Heating', 'GarageType', 'RoofMatl']

[ ] num_cols = train_df.select_dtypes(include=['int64','float64']).columns
    num_cols = num_cols.drop('SalePrice')

[ ] num_pipeline = Pipeline(steps=[
        ('impute',SimpleImputer(strategy='mean')),   # Khi gặp giá trị thiếu(NaN),thay thế chúng bằng giá trị trung bình (mean) của cột tương ứng.
        ('scaler',StandardScaler())                  # Chuẩn hóa dữ liệu để đảm bảo mỗi cột có trung bình bằng 0 và độ lệch chuẩn bằng 1.
    ])

[ ] ode_pipeline = Pipeline(steps=[
        ('impute',SimpleImputer(strategy='most_frequent')),        # Thay thế các giá trị bị thiếu (NaN) trong dữ liệu phân loại bằng giá trị xuất hiện nhiều nhất trong cột.
        ('ode',OrdinalEncoder(handle_unknown='use_encoded_value',unknown_value=-1))    # Mã hóa các giá trị phân loại thành số nguyên (integers): Các giá trị đã biết sẽ được ánh xạ thành số nguyên,các giá trị không xác định sẽ được mã hóa thành -1.
    ])

[ ] ohe_pipeline = Pipeline(steps=[
        ('impute', SimpleImputer(strategy='most_frequent')),    # Thay thế giá trị bị thiếu (NaN) bằng giá trị xuất hiện nhiều nhất trong mỗi cột dữ liệu phân loại.
        ('ohe', OneHotEncoder(handle_unknown='ignore',sparse_output=False))     # Mã hóa One-Hot Encoding: Biến mỗi giá trị phân loại thành một hoặc nhiều cột nhị phân.
    ])

[ ] col_trans = ColumnTransformer(transformers=[
        ('num_p',num_pipeline,num_cols),    # Áp dụng num_pipeline (Pipeline cho dữ liệu số) lên các cột trong danh sách num_cols.
        ('ode_p',ode_pipeline,ode_cols),    # Áp dụng ode_pipeline (Pipeline cho dữ liệu phân loại dạng thứ tự) lên các cột trong danh sách ode_cols.
        ('ohe_p',ohe_pipeline,ohe_cols)],   # Áp dụng ohe_pipeline (Pipeline cho dữ liệu phân loại không thứ tự) lên các cột trong danh sách ohe_cols.
        remainder='passthrough',        # Giữ nguyên (không biến đổi) các cột không được chỉ định trong transformers.
        n_jobs=-1)                  # Cho phép thực thi song song để tăng tốc độ xử lý, bằng cách sử dụng tất cả các CPU có sẵn.
```

- A correlation matrix is compiled to show the correlation between individual features and their labels. High correlation labels can be later used as roots or key features for decision trees.

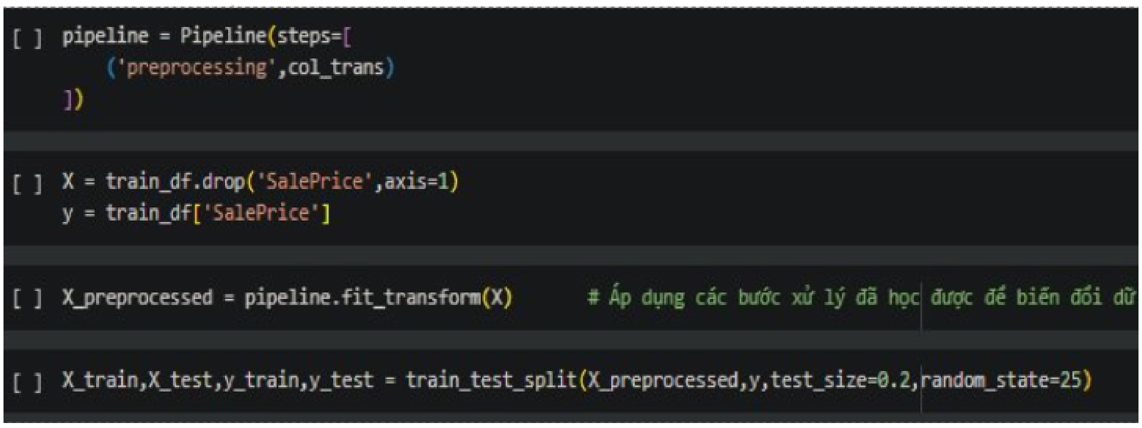


- Columns are classified for one-hot and ordinal encoders.



```
ode_cols = ['LotShape', 'LandContour','Utilities','LandSlope',  'BsmtQual',  'BsmtFinType1',  'CentralAir',  'Functional', \
            'FireplaceQu', 'GarageFinish', 'GarageQual', 'PavedDrive', 'ExterCond', 'KitchenQual', 'BsmtExposure', 'HeatingQC','ExterQual', 'BsmtCond']

[ ] ohe_cols = ['Street', 'LotConfig','Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'Exterior1st', 'Exterior2nd', \
            'MasVnrType','Foundation', 'Electrical',  'SaleType', 'MSZoning', 'SaleCondition', 'Heating', 'GarageType', 'RoofMatl']

[ ] num_cols = train_df.select_dtypes(include=['int64','float64']).columns
    num_cols = num_cols.drop('SalePrice')

[ ] num_pipeline = Pipeline(steps=[
        ('impute',SimpleImputer(strategy='mean')),   # Khi gặp giá trị thiếu(NaN),thay thế chúng bằng giá trị trung bình (mean) của cột tương ứng.
        ('scaler',StandardScaler())                  # Chuẩn hóa dữ liệu để đảm bảo mỗi cột có trung bình bằng 0 và độ lệch chuẩn bằng 1.
    ])

[ ] ode_pipeline = Pipeline(steps=[
        ('impute',SimpleImputer(strategy='most_frequent')),        # Thay thế các giá trị bị thiếu (NaN) trong dữ liệu phân loại bằng giá trị xuất hiện nhiều nhất trong cột.
        ('ode',OrdinalEncoder(handle_unknown='use_encoded_value',unknown_value=-1))    # Mã hóa các giá trị phân loại thành số nguyên (integers): Các giá trị đã biết sẽ được ánh xạ thành số nguyên,các giá trị không xác định sẽ được mã hóa thành -1.
    ])

[ ] ohe_pipeline = Pipeline(steps=[
        ('impute', SimpleImputer(strategy='most_frequent')),    # Thay thế giá trị bị thiếu (NaN) bằng giá trị xuất hiện nhiều nhất trong mỗi cột dữ liệu phân loại.
        ('ohe', OneHotEncoder(handle_unknown='ignore',sparse_output=False))     # Mã hóa One-Hot Encoding: Biến mỗi giá trị phân loại thành một hoặc nhiều cột nhị phân.
    ])

[ ] col_trans = ColumnTransformer(transformers=[
        ('num_p',num_pipeline,num_cols),    # Áp dụng num_pipeline (Pipeline cho dữ liệu số) lên các cột trong danh sách num_cols.
        ('ode_p',ode_pipeline,ode_cols),    # Áp dụng ode_pipeline (Pipeline cho dữ liệu phân loại dạng thứ tự) lên các cột trong danh sách ode_cols.
        ('ohe_p',ohe_pipeline,ohe_cols)],   # Áp dụng ohe_pipeline (Pipeline cho dữ liệu phân loại không thứ tự) lên các cột trong danh sách ohe_cols.
        remainder='passthrough',        # Giữ nguyên (không biến đổi) các cột không được chỉ định trong transformers.
        n_jobs=-1)                  # Cho phép thực thi song song để tăng tốc độ xử lý, bằng cách sử dụng tất cả các CPU có sẵn.
```

- Various further pre-processing steps, including splitting the dataset for training and testing, are taken. The dataset is now ready for use.



```
[ ] pipeline = Pipeline(steps=[
        ('preprocessing',col_trans)
    ])

[ ] X = train_df.drop('SalePrice',axis=1)
    y = train_df['SalePrice']

[ ] X_preprocessed = pipeline.fit_transform(X)        # Áp dụng các bước xử lý đã học được để biến đổi dữ

[ ] X_train,X_test,y_train,y_test = train_test_split(X_preprocessed,y,test_size=0.2,random_state=25)
```

2.3 Third-party Libraries

Notable third-party libraries used include the scikit-learn library for its various machine-learning modules, and the XGBoost library for its tree boosting algorithm (to be elaborated on later in the report).

CHAPTER 3. Model Evaluation

3.1 Evaluation parameters

3.1.1 Mean Squared Error

Mean Squared Error (MSE) is a common metric used to measure the accuracy of a model, particularly in regression tasks. It quantifies the difference between predicted values and actual values by calculating the average squared difference.

Formula:

The formula for MSE is:

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

Where:

- n: Number of data points.
- y_i: Actual value of the i-th observation.
- $\hat{y}_i$: Predicted value of the i-th observation.
- $(y_i - \hat{y}_i)^2$: Squared difference between actual and predicted values.

3.1.2 R^2 Score

R^2 Score, or the Coefficient of Determination, is a statistical measure used to evaluate the performance of a regression model. It indicates how well the model's predictions fit the actual data and represents the proportion of the variance in the dependent variable that is predictable from the independent variables.

Formula:

The formula for R^2 is:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

Where:

- SS_{res}: Residual sum of squares, calculated as:

$$SS_{\text{res}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  It measures the variance of errors (difference between actual and predicted values).

- SS_{tot}: Total sum of squares, calculated as:

$$SS_{\text{tot}} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

It measures the variance of the actual data points relative to their mean.

Where:

- y_i: Actual value for the i-th observation.
- $\hat{y}_i$: Predicted value for the i-th observation.
- $\bar{y}$: Mean of the actual values.

The following models differ in their methodology, but in usage after training has been completed, the method of usage remains the same: a prediction is given after a set of factors have been inputted.

4.0.1 Linear Regression

a, Methodology

The overarching idea of this model is to process the given values to create a linear prediction formula.

Assume m houses in the training data set, each with corresponding house price values of $y_1, y_2, ..., y_m$. The linear prediction formula takes on the following general form:

$$\hat{y} = f(x) = w_0 + x^{(1)}w_1 + x^{(2)}w_2 + ... + x^{(n)}w_n$$

$$= \begin{bmatrix} 1 & x^{(1)} & ... & x^{(n)} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ w_n \end{bmatrix}$$

$$Given\ x = \begin{bmatrix} 1 & x^{(1)} & ... & x^{(n)} \end{bmatrix}^T$$

$$\implies w = \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ w_n \end{bmatrix} \implies \hat{y} = f(x) = x^T w$$

in which x is a vector containing the features of the house whose property is being predicted, and w is the vector containing the weights of this linear model. For each i in $\overline{1, m}$, we have the following:

$$\hat{y}_1 = x_i^T \cdot w$$

$$Given\ Y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_m \end{bmatrix} \implies \hat{Y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ . \\ . \\ \hat{y}_m \end{bmatrix} = \begin{bmatrix} x_1^T \\ x_2^T \\ . \\ . \\ x_m^T \end{bmatrix} \times \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ w_n \end{bmatrix}$$

$$Given\ X = \begin{bmatrix} x_1 & x_2 & ... & x_m \end{bmatrix} \implies X^T = \begin{bmatrix} x_1^T \\ x_2^T \\ . \\ . \\ x_m^T \end{bmatrix}$$

Thus, $\hat{Y} = X^T w$

The objective is to, by optimizing the w vector, minimize the $L2$ loss function which represents the difference between the predicted and actual values:

$$\mathcal{L}(w) = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 = (Y - \hat{Y})^T (Y - \hat{Y})$$

As $\mathcal{L}(w) \geq 0\ \forall w$,the objective becomes to find a w such that

$$\frac{\nabla \mathcal{L}(w)}{\nabla w} = 0$$

for this function to be optimized.

$$Given\ \mathcal{L}(w) = (Y - X^T w)^T (Y - X^T w)$$

$$= (Y^T - w^T X)(Y - X^T w)$$

$$Y^T Y - w^T XY - Y^T X^T w + w^T X X^T w$$

As w^T is of dimension $1 \times (n+1)$, X is of dimension $(n+1) \times m$, and Y is of dimension $m \times 1$, it is concluded that $w^T XY$ is of dimension 1×1.

As such,

$$w^T XY = (w^T XY)^T = Y^T X^T w$$

$$\implies \mathcal{L}(w) = Y^T Y - 2Y^T X^T w + w^T X X^T w$$

$$\implies \frac{\nabla \mathcal{L}(w)}{\nabla w} = \frac{d(Y^T Y)}{dw} = 2\frac{d(Y^T X^T w)}{dw} + \frac{d(w^T X X^T w)}{dw}$$

$$= 0 - 2Y^T X^T + 2w^T X X^T$$

From the above, it can be concluded that:

$$\frac{\nabla \mathcal{L}(w)}{\nabla w} = 0$$

$$\iff 2w^T X X^T = 2Y^T X^T$$

$$\iff w^T X X^T = Y^T X^T$$

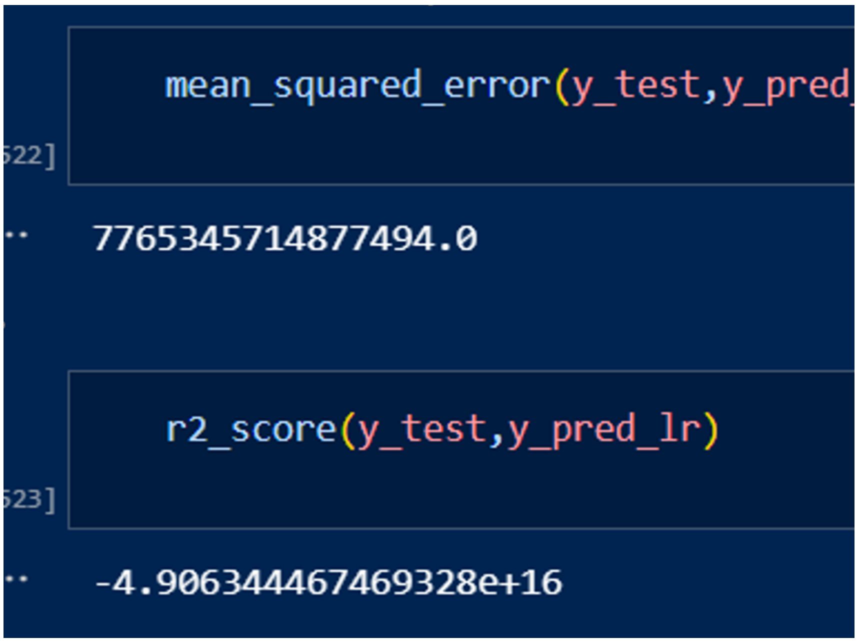
$$\iff X X^T w = XY$$

One of the following cases will occur based on the invertibility of XX^T:

- If XX^T is invertible, the equation has one unique solution of $w = (XX^T)^{-1}XY$.
- If XX^T is non-invertible, Python supports finding a pseudoinverse for non-invertible matrices.
  We define $(XX^T)^\dagger$ as the pseudoinverse of XX^T $\implies$ The solution to the equation becomes $w = (XX^T)^\dagger XY$.

b, Results

The following result were obtained on the test set:



```
522]    mean_squared_error(y_test,y_pred
···     7765345714877494.0

523]    r2_score(y_test,y_pred_lr)
···     -4.906344467469328e+16
```

c, Evaluation

Despite performing reasonably well, the Linear Regression model is not without its setbacks; namely, some features x_i may be either insignificant or excessively

large, and the model may conform too greatly to these outliers in the training set, resulting in larger model complexity.
This phenomenon, known as $overfitting$, causes a model to perform poorly on the test set due to conforming too greatly to noise on the training set. This can be very clearly seen on the results on the test set, where the model failed to provide any sort of reasonable prediction.

4.0.2 Ridge Regression

a, Methodology

This model, fundamentally a modification of the Linear Regression model, aims to alleviate overfitting, the main drawback of its predecessor.

To achieve this goal, l_2 regularization terms are added to the original $L2$ loss function, resulting in the following new loss function:

$$\mathcal{J}(w) = \mathcal{L}(w) + \lambda ||w||_2^2$$

in which $||w||_2^2$ is the square sum of all coefficients within the vector w.

λ denotes the penalty coefficient (also known as the ridge coefficient), which must be adjusted appropriately to ensure maximum performance by the model.

The above modifications serve to reduce the weights of insignificant features within the $\mathcal{L}(w)$ function for them to approach 0, as well as reduce the weights of features that are excessively large.

In specific, we have the following:

$$\mathcal{J}(w) = \mathcal{L}(w) + \lambda ||w||_2^2$$

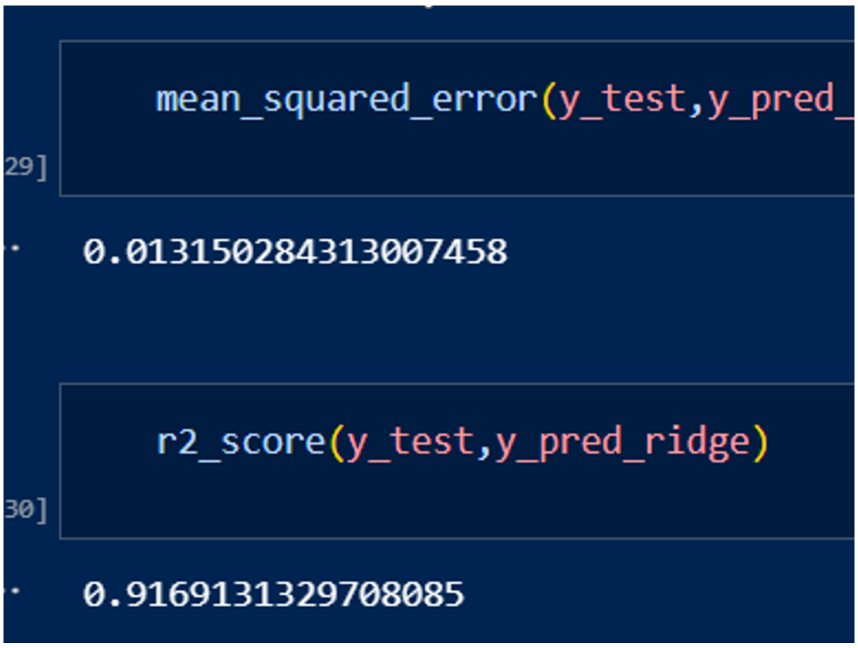
$$= Y^TY - w^TXY - w^TXY - Y^TX^Tw + w^TXX^Tw + \lambda w^Tw$$

$$= Y^TY - 2Y^TX^T + w^TXX^Tw + \lambda w^Tw$$

Similar to the Linear Regression model, the objective is to minimize $\mathcal{J}(w)$ by ensuring that its derivative with respect to w is equal to 0.

As such:

$$\frac{\nabla \mathcal{J}(w)}{\nabla w} = 0$$



```
29]   mean_squared_error(y_test,y_pred_

      0.013150284313007458

30]   r2_score(y_test,y_pred_ridge)

      0.9169131329708085
```

$$\iff \frac{d(Y^TY)}{dw} - \frac{d(Y^TX^Tw)}{dw} + \frac{d(w^TXX^Tw)}{dw} + \lambda\frac{d(w^Tw)}{dw} = 0$$

$$\iff 2w^TXX^T - 2Y^TX^T + \lambda 2w^T = 0$$

$$\iff w^T(XX^T + \lambda I) = Y^TX^T$$

$$\iff (XX^T + \lambda I)^T w = XY$$

$$\iff (XX^T + \lambda I)w = XY(*)$$

With the appropriate λ, we have $XX^T + \lambda I > XX^T$

$$\implies (XX^T + \lambda I)^{-1} < (XX^T)^{-1}$$

Thus, the solution of $(*)$ becomes $w = (XX^T + \lambda I)^{-1}XY$.

b, Results

The following results were obtained on the test set:

c, Evaluation

With the addition of the ridge constant, he coefficients within the vector w have decreased in value, thus reducing overfitting and the complexity of the Linear Regression model. This can be very clearly seen in the results, where the accuracy has been massively improved.

4.1 Random Forest

4.1.1 Decision Tree

A *decision tree* is a model composed of a collection of "questions" organized hierarchically in the shape of a tree, commonly called *conditions, splits or tests*. Each non-leaf node contains a condition, and each leaf node contains a prediction. Decision trees are commonly represented with the root node (the first node) at the top.

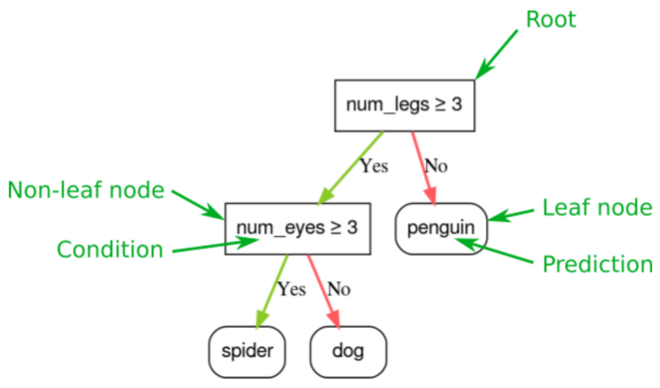


Figure 4.1: An example of a decision tree

The inference of a decision tree model is computed by routing an example from the root (at the top) to one of the leaf nodes (at the bottom) according to the conditions. The value of the reached leaf is the decision tree's prediction. The set of visited nodes is called the *inference path*.

Types of conditions:

Axis-aligned conditions and oblique conditions: An axis-aligned condition involves only a single feature, whereas an oblique condition involves multiple.

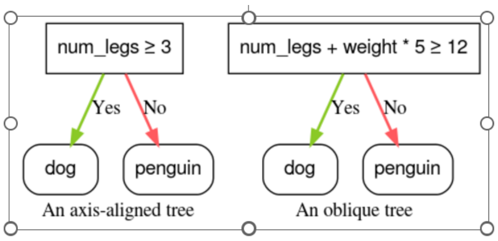


Figure 4.2: Example of decision tree

Binary and non-binary conditions:

- Conditions with two possible outcomes (e.g. true or false) are called binary conditions. Decision trees containing only binary conditions are called binary decision trees.
- Non-binary conditions have more than two possible outcomes. Therefore, non-binary conditions have more discriminative power than binary conditions. Decisions containing one or more non-binary conditions are called non-binary.

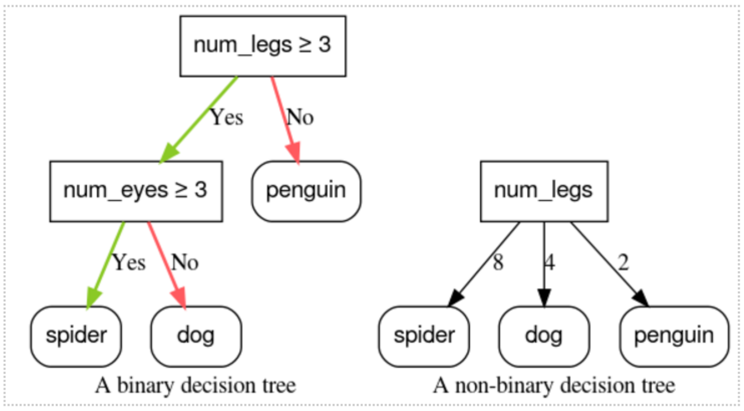


Figure 4.3: Example of Decision tree with binary and non-binary conditions

4.1.2 Random Forest Training

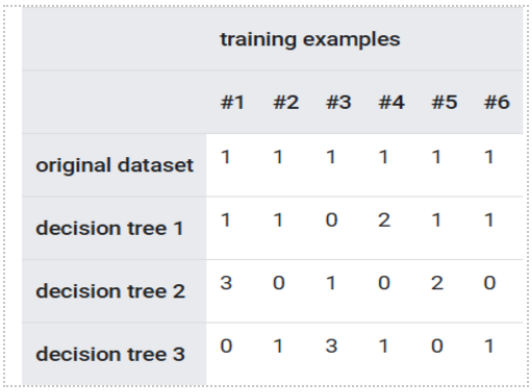
a, Bagging

Bagging (portmanteau of *bootstrap aggregating*) means training each decision tree on a random subset of the examples in the training set. In other words, each decision tree in the random forest is trained on a different subset of examples.

Bagging is peculiar. Each decision tree is trained on the same number of examples as in the original training set. For instance, if the original training set contains 60 examples, then each decision tree is trained on 60 examples. However, bagging only trains each decision tree on a subset (typically, 67 percent) of those examples. So, some of those 40 examples in the subset must be reused while training a given decision tree. This reuse is called training *with replacement*."

The following table shows how bagging could distribute six examples across three decision trees. Notice the following:

- Each decision tree trains on a total of six examples.
- Each decision tree trains on a different set of examples.
- Each decision tree reuses certain examples. For example, example #4 is used twice in training decision tree 1; therefore, the learned weight of example #4 is effectively doubled in decision tree 1.



	training examples					
	#1	#2	#3	#4	#5	#6
original dataset	1	1	1	1	1	1
decision tree 1	1	1	0	2	1	1
decision tree 2	3	0	1	0	2	0
decision tree 3	0	1	3	1	0	1

Figure 4.4: *Bagging six training examples across three decision trees. Each number represents the number of times a given training example (#1-6) is repeated in the training dataset of a given decision tree (1-3).*

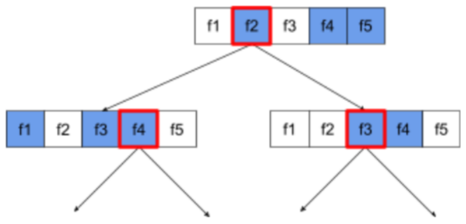
In bagging, each decision tree is almost always trained on the total number of examples in the original training set, as training on more or fewer examples usually results in degradation in the quality of the random forest.

The sampling of examples is sometimes done "without replacement"; that is, a training example cannot be present more than once among all of the training sets. For example, in the preceding table, all values would be either 0 or 1.

b, Attribute Sampling

Attribute sampling means that instead of looking for the best condition over all available features, only a random subset of features are tested at each node. The set of tested features is sampled randomly at each node of the decision tree.

The following decision tree illustrates the attribute / feature sampling. Here a decision tree is trained on 5 features (f1-f5). The blue nodes represent the tested features while the white ones are not tested. The condition is built from the best tested features (represented with a red outline).



The ratio of attribute sampling is an important regularization hyperparameter. The preceding Figure used a approximately 3/5 ratio. Many random forest implementations test, by default, 1/3 of the features for regression and sqrt(number of features) for classification.

In TF-DF, the following hyperparameters control attribute sampling:

- num-candidate-attributes
- num-candidate-attributes-ratio

For example, if num-candidate-attributes-ratio=0.5, half of the features will be tested at each node.

Disabling Decision Tree Regularization

Individual decision trees in a random forest are trained without pruning. This produces overly complex trees with poor predictive quality. Instead of regularizing individual trees, the trees are ensembled producing more accurate overall predictions.

We expect the training and test accuracy of a random forest to differ. The training accuracy of a random forest is generally much higher (sometimes equal to 100%). However, a very high training accuracy in a random forest is normal and does not indicate that the random forest is overfitted.

The two sources of randomness (bagging and attribute sampling) ensure the relative independence between the decision trees. This independence corrects the overfitting of the individual decision trees. Consequently, the ensemble is not overfitted. We'll illustrate this non-intuitive effect in the next unit.

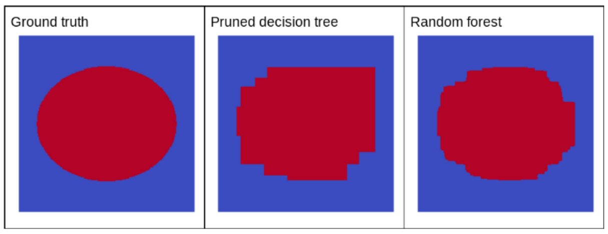
Pure random forests train without maximum depth or minimum number of observations per leaf. In practice, limiting the maximum depth and minimum number of observations per leaf is beneficial. By default, many random forests use the following defaults:

- maximum depth of 16
- minimum number of observations per leaf of 5.
- You can tune these hyperparameters.

c, Noise clarity

Why would random noise improve the quality of a random forest? To illustrate the benefits of random noise, the next figure shows the predictions of a classical (pruned) decision tree and a random forest trained on a few examples of simple two-dimensional problem with an ellipse pattern.

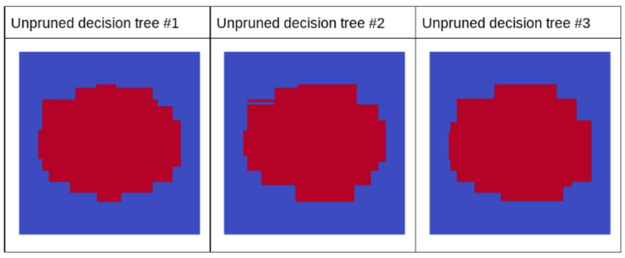
Ellipses patterns are notoriously hard for decision tree and decision forest algorithms to learn with axis-aligned conditions, so they make a good example. Notice that the pruned decision tree can't get the same quality of prediction as the random forest.



The next plot shows the predictions of the first three unpruned decision trees of the random forest; that is, the decision trees are all trained with a combination of:

- bagging
- attribute sampling
- disabling pruning

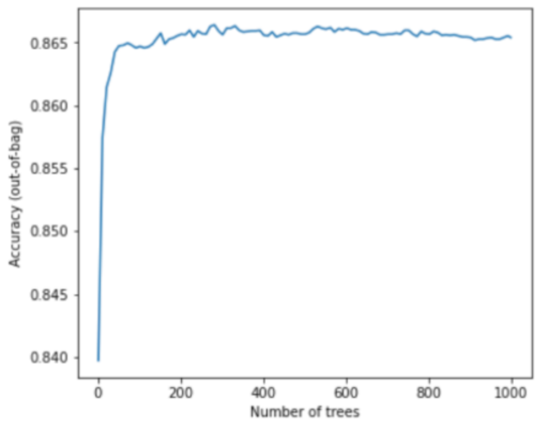
Notice that the individual predictions of these three decision trees are worse than the predictions of the pruned decision tree in the preceding figure. However, since the errors of the individual decision trees are only weakly correlated, the three decision trees combine in an ensemble to create effective predictions.



Because the decision trees of a random forest are not pruned, training a random forest does not require a validation dataset. In practice, and especially on small datasets, models should be trained on all the available data.

When training a random forest, as more decision trees are added, the error almost always decreases; that is, the quality of the model almost always improves. Yes, adding more decision trees almost always reduces the error of the random forest. In other words, adding more decision trees cannot cause the random forest to overfit. At some point, the model just stops improving. Leo Breiman famously said, "Random Forests do not overfit, as more trees are added".

For example, the following plot shows the test evaluation of a random forest model as more decision trees are added. The accuracy rapidly improves until it plateaus around 0.865. However, adding more decision trees does not make accuracy decrease; in other words, the model does not overfit. This behavior is (mostly) always true and independent of the hyperparameters.



Accuracy stays constant as more decision trees are added to the random forest.

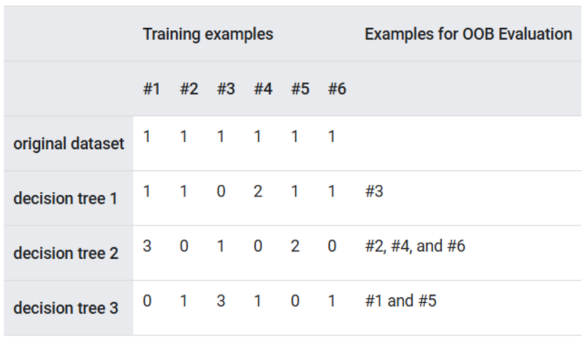
d, Out-of-bag evaluation

Random forests do not require a validation dataset. Most random forests use a technique called out-of-bag-evaluation (OOB evaluation) to evaluate the quality of the model. OOB evaluation treats the training set as if it were on the test set of a cross-validation.

As explained earlier, each decision tree in a random forest is typically trained on approximately 67% of the training examples. Therefore, each decision tree does not see approximately 33% of the training examples. The core idea of OOB-evaluation is as follows: -To evaluate the random forest on the training set.

-For each example, only use the decision trees that did not see the example during training.

The following table illustrates OOB evaluation of a random forest with 3 decision trees trained on 6 examples (this is the same table as in the Bagging section). The table shows which decision tree is used with which example during OOB evaluation.



	Training examples						Examples for OOB Evaluation
	#1	#2	#3	#4	#5	#6	
original dataset	1	1	1	1	1	1	
decision tree 1	1	1	0	2	1	1	#3
decision tree 2	3	0	1	0	2	0	#2, #4, and #6
decision tree 3	0	1	3	1	0	1	#1 and #5

Figure 4.5: Table: OOB Evaluation - the numbers represent the number of times a given training example is used during training of the given example

In the example shown in the table, the OOB predictions for training example 1 will be computed with decision tree #3 (since decision trees 1 and #2 used this example for training). In practice, on a reasonable size dataset and with a few decision trees, all the examples have an OOB prediction.

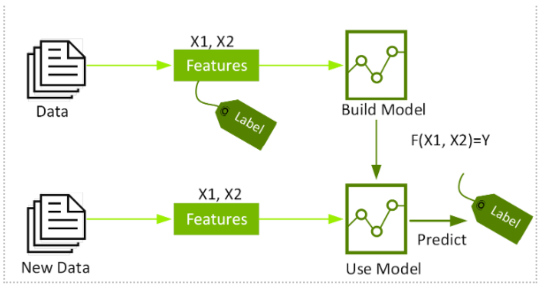
OOB evaluation is also effective to compute permutation variable importance for random forest models. Remember that permutation variable importance measures the importance of a variable by measuring the drop of model quality when this variable is shuffled. The random forest "OOB permutation variable importance" is a permutation variable importance computed using the OOB evaluation.

4.2 XGBoost

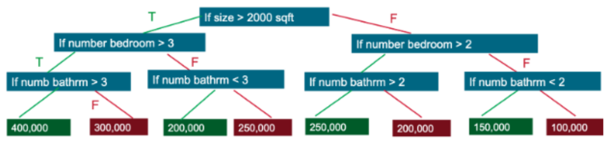
XGBoost, which stands for **Extreme Gradient Boosting**, is a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library. It provides parallel tree boosting and is the leading machine-learning library for regression, classification, and ranking problems.

To understand XGBoost, it is vital to first grasp the machine learning concepts and algorithms that XGBoost builds upon: supervised machine learning, decision trees, ensemble learning, and gradient boosting.

Supervised machine learning uses algorithms to train a model to find patterns in a dataset with labels and features and then uses the trained model to predict the labels on a new dataset's features.

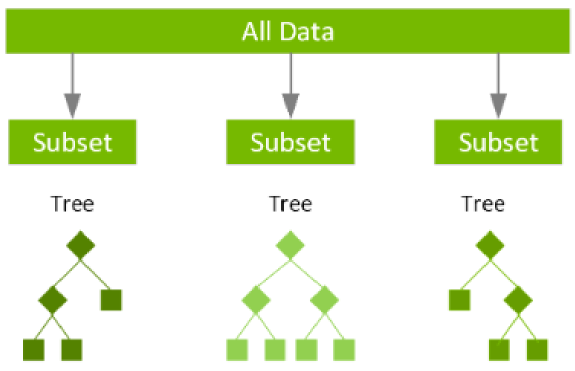


Decision trees create a model that predicts the label by evaluating a tree of if-then-else true/false feature questions, and estimating the minimum number of questions needed to assess the probability of making a correct decision. Decision trees can be used for classification to predict a category, or regression to predict a continuous numeric value. In the simple example below, a decision tree is used to estimate a house price (the label) based on the size and number of bedrooms (the features).



A Gradient Boosting Decision Trees (GBDT) is a decision tree ensemble learning algorithm similar to random forest, for classification and regression. Ensemble learning algorithms combine multiple machine learning algorithms to obtain a better model.

Both random forest and GBDT build a model consisting of multiple decision trees. The difference is in how the trees are built and combined.



Random forest uses a technique called bagging to build full decision trees in parallel from random bootstrap samples of the data set. The final prediction is an average of all of the decision tree predictions.

The term "gradient boosting" comes from the idea of "boosting" or improving a single weak model by combining it with a number of other weak models in order to generate a collectively strong model. Gradient boosting is an extension of boosting where the process of additively generating weak models is formalized as a gradient descent algorithm over an objective function. Gradient boosting sets targeted outcomes for the next model in an effort to minimize errors. Targeted outcomes for each case are based on the gradient of the error (hence the name gradient boosting) with respect to the prediction.

GBDTs iteratively train an ensemble of shallow decision trees, with each iteration using the error residuals of the previous model to fit the next model. The final prediction is a weighted sum of all of the tree predictions. Random forest "bagging" minimizes the variance and overfitting, while GBDT "boosting" minimizes the bias and underfitting.

XGBoost is a scalable and highly accurate implementation of gradient boosting that pushes the limits of computing power for boosted tree algorithms, being built largely for energizing machine learning model performance and computational speed. With XGBoost, trees are built in parallel, instead of sequentially like GBDT. It follows a level-wise strategy, scanning across gradient values and using these partial sums to evaluate the quality of splits at every possible split in the training set.

Interesting metrics and features include:

- Regularization: XGBoost has an option to penalize complex models through both L1 and L2 regularization. Regularization helps in preventing overfitting
- Handling sparse data: Missing values or data processing steps like one-hot

encoding make data sparse. XGBoost Classifier incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data

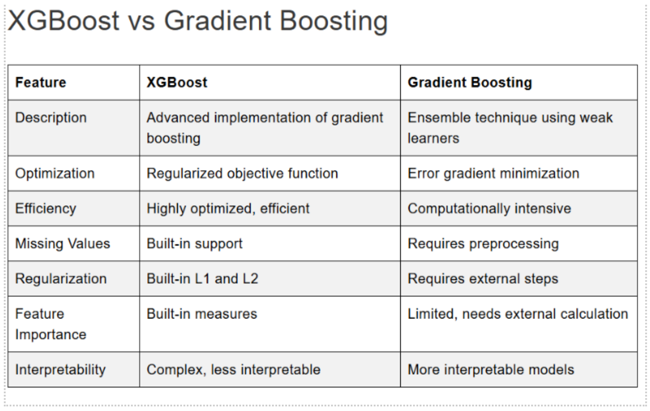
- Weighted quantile sketch: Most existing tree based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data
- Block structure for parallel learning: For faster computing, XGBoost Classifier can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sort and store data in in-memory units called blocks. Unlike other algorithms, this approach enables subsequent iterations to reuse the data layout instead of computing it again.
- This feature also serves useful for steps like split finding and column subsampling
- Cache awareness: In XGBoost machine learning, Scala requires non-continuous memory access to obtain the gradient statistics by row index. Hence, Tianqi Chen designed XGBoost to optimize hardware usage. This optimization occurs by allocating internal buffers in each thread, where the workflow can store the gradient statistics. And these parallel tree make better XGboost algorithms with the help of julia and java lanuages.
- Out-of-core computing: This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

Benefits and attributes of XGBoost include:

- High accuracy: Delivers high accuracy and consistently outperforms other machine learning algorithms in many predictive modeling tasks.
- Scalability: Highly scalable and can handle large datasets with millions of rows and columns.
- Efficiency: The design ensures computational efficiency, allowing it to quickly train models on large datasets.
- Flexibility: It supports a variety of data types and objectives, including regression, classification, and ranking problems.
- Regularization: Incorporates regularization techniques to avoid overfitting and improve generalization performance.
- Interpretability: Provides feature importance scores that can help users understand

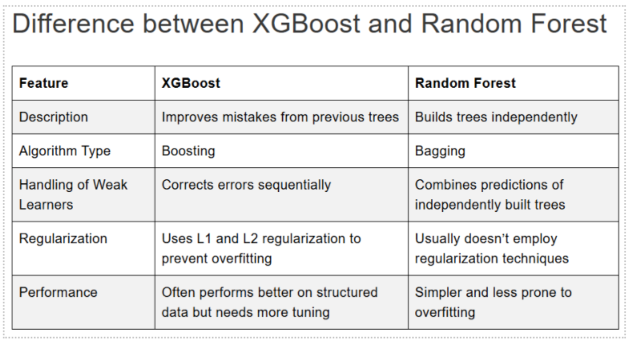
which features are most important for making predictions.

- Open-source: Serves as an open-source library widely used and supported by the data science community.



XGBoost vs Gradient Boosting

Feature	XGBoost	Gradient Boosting
Description	Advanced implementation of gradient boosting	Ensemble technique using weak learners
Optimization	Regularized objective function	Error gradient minimization
Efficiency	Highly optimized, efficient	Computationally intensive
Missing Values	Built-in support	Requires preprocessing
Regularization	Built-in L1 and L2	Requires external steps
Feature Importance	Built-in measures	Limited, needs external calculation
Interpretability	Complex, less interpretable	More interpretable models



Difference between XGBoost and Random Forest

Feature	XGBoost	Random Forest
Description	Improves mistakes from previous trees	Builds trees independently
Algorithm Type	Boosting	Bagging
Handling of Weak Learners	Corrects errors sequentially	Combines predictions of independently built trees
Regularization	Uses L1 and L2 regularization to prevent overfitting	Usually doesn't employ regularization techniques
Performance	Often performs better on structured data but needs more tuning	Simpler and less prone to overfitting

4.3 K-Nearest Neighbor(KNN)

KNN is one of the most basic yet essential classification algorithms in machine learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining, and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data). We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:

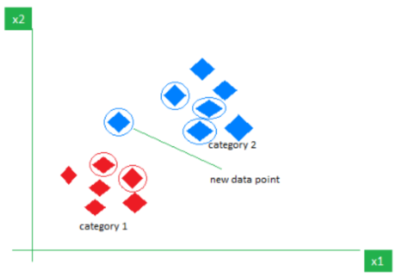


Figure 4.6: KNN Algorithm working visualization

Now, given another set of data points (also called testing data), allocate these points to a group by analyzing the training set. Note that the unclassified points are marked as 'White'.

Intuition Behind KNN Algorithm

If we plot these points on a graph, we may be able to locate some clusters or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbors belong to. This means a point close to a cluster of points classified as 'Red' has a higher probability of getting classified as 'Red'.

Intuitively, we can see that the first point (2.5, 7) should be classified as 'Green', and the second point (5.5, 4.5) should be classified as 'Red'.

Why do we need a KNN algorithm?

(K-NN) algorithm is a versatile and widely used machine learning algorithm that is primarily used for its simplicity and ease of implementation. It does not require any assumptions about the underlying data distribution. It can also handle both numerical and categorical data, making it a flexible choice for various types of datasets in classification and regression tasks. It is a non-parametric method that makes predictions based on the similarity of data points in a given dataset. K-NN is less sensitive to outliers compared to other algorithms.

The K-NN algorithm works by finding the K nearest neighbors to a given data point based on a distance metric, such as Euclidean distance. The class or value of the data point is then determined by the majority vote or average of the K neighbors. This approach allows the algorithm to adapt to different patterns and make predictions based on the local structure of the data.

Distance Metrics Used in KNN Algorithm

As we know that the KNN algorithm helps us identify the nearest points or the groups for a query point. But to determine the closest groups or the nearest points for a query point we need some metric. For this purpose, we use below distance metrics:

Euclidean Distance

This is nothing but the cartesian distance between the two points which are in the plane/hyperplane. *Euclidean distance* can also be visualized as the length of the straight line that joins the two points which are into consideration. This metric helps us calculate the net displacement done between the two states of an object.

$$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^{d}(x_j - X_{ij})^2}$$

Manhattan Distance

Manhattan Distance metric is generally used when we are interested in the total distance traveled by the object instead of the displacement. This metric is calculated by summing the absolute difference between the coordinates of the points in n-dimensions.

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Minkowski Distance

We can say that the Euclidean, as well as the Manhattan distance, are special cases of the *Minkowski distance*.

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

From the formula above we can say that when p = 2 then it is the same as the formula for the Euclidean distance and when p = 1 then we obtain the formula for the Manhattan distance.

The above-discussed metrics are most common while dealing with a Machine Learning problem but there are other distance metrics as well like Hamming Distance which come in handy while dealing with problems that require overlapping comparisons between two vectors whose contents can be Boolean as well as string values.

How to choose the value of k for KNN Algorithm?

The value of k is very crucial in the KNN algorithm to define the number of neighbors in the algorithm. The value of k in the k-nearest neighbors (k-NN) algorithm should be chosen based on the input data. If the input data has more outliers or noise, a higher value of k would be better. It is recommended to choose an odd value for k to avoid ties in classification. methods can help in selecting the best k value for the given dataset.

Algorithm for K-NN

```
1. DistanceToNN = sort(distance from 1st example, distance from $k$th example)
2. For value $i = 1$ to number of training records:
     • Dist = distance(test example, $i$th example)
     • If (Dist < any example in DistanceToNN):
         – Remove the example from DistanceToNN and value.
         – Put new example in DistanceToNN and value in sorted order.
3. Return average of value.
```

Fit using k-NN is more reasonable than 1-NN. k-NN affects very less from noise if the dataset is large.

Remarks

In the k-NN algorithm, we can see jumps in prediction values due to unit changes in input. The reason for this is the change in neighbors. To handle this situation, we can use weighting of neighbors in the algorithm. If the distance from a neighbor is high, we want less effect from that neighbor. If the distance is low, that neighbor should be more effective than others.

Workings of KNN algorithm

The K-Nearest Neighbors (KNN) algorithm operates on the principle of similarity, where it predicts the label or value of a new data point by considering the labels or values of its K nearest neighbors in the training dataset.

Step 1: Selecting the Optimal Value of K

- K represents the number of nearest neighbors that needs to be considered while making predictions.

Step 2: Calculating Distance

- To measure the similarity between target and training data points, Euclidean distance is used.
- Distance is calculated between each of the data points in the dataset and the target point.

Step 3: Finding Nearest Neighbors

- The k data points with the smallest distances to the target point are the nearest neighbors.

Step 4: Voting for Classification or Taking Average for Regression

- In the classification problem, the class labels of k-nearest neighbors are determined by performing majority voting. The class with the most occurrences among the neighbors becomes the predicted class for the target data point.
- In the regression problem, the class label is calculated by taking the average of the target values of k nearest neighbors. The calculated average value becomes the predicted output for the target data point.

Let X be the training dataset with n data points, where each data point is represented by a d-dimensional feature vector X_i, and Y be the corresponding labels or values for each data point in X.

Given a new data point x, the algorithm calculates the distance between x and each data point X_i in X using a distance metric, such as Euclidean distance:

$$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^{d} (x_j - X_{ij})^2}$$

The algorithm selects the K data points from X that have the shortest distances to x.

- For **classification tasks**, the algorithm assigns the label y that is most frequent among the K nearest neighbors to x.
- For **regression tasks**, the algorithm calculates the average or weighted average of the values y of the K nearest neighbors and assigns it as the predicted value for x.

Advantages of the KNN Algorithm

- **Easy to implement** – The complexity of the algorithm is not that high.
- **Adapts Easily** – As per the working of the KNN algorithm, it stores all the data in memory storage. Hence, whenever a new example or data point is added, the algorithm adjusts itself as per that new example and contributes to future predictions.
- **Few Hyperparameters** – The only parameters required in the training of a KNN algorithm are the value of k and the choice of the distance metric to use

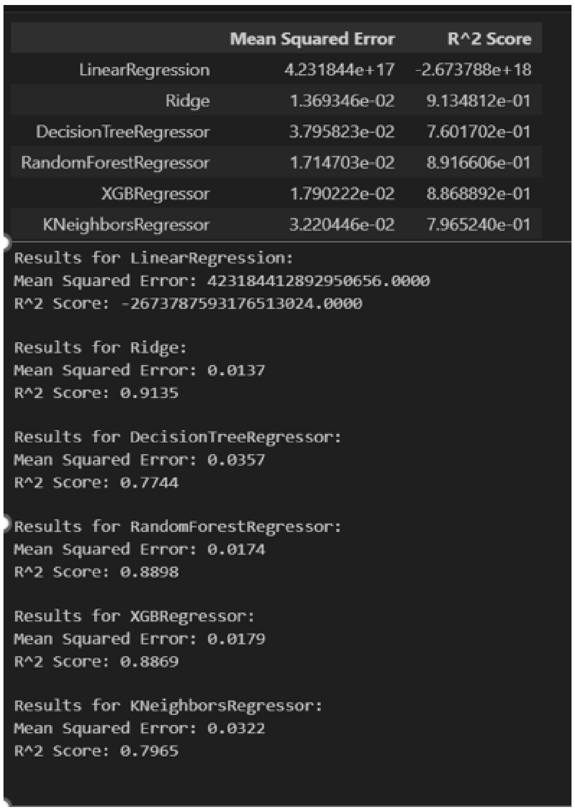
for evaluation.

Disadvantages of the KNN Algorithm

- **Does not scale** – The KNN algorithm is also considered a *Lazy Algorithm*. This implies it takes a lot of computing power and memory storage. This makes the algorithm time-consuming and resource-intensive.
- **Curse of Dimensionality** – The algorithm is affected by the *curse of dimensionality*, which means it faces challenges in classifying data points properly when the dimensionality is too high. For more information, refer to Curse of Dimensionality.
- **Prone to Overfitting** – Due to the curse of dimensionality, the algorithm is prone to overfitting. Generally, feature selection and dimensionality reduction techniques are applied to mitigate this issue.

CHAPTER 5. CONCLUSIONS

5.1 Model performance

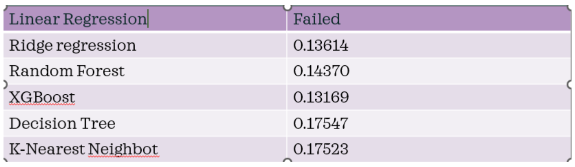


	Mean Squared Error	R^2 Score
LinearRegression	4.231844e+17	-2.673788e+18
Ridge	1.369346e-02	9.134812e-01
DecisionTreeRegressor	3.795823e-02	7.601702e-01
RandomForestRegressor	1.714703e-02	8.916606e-01
XGBRegressor	1.790222e-02	8.868892e-01
KNeighborsRegressor	3.220446e-02	7.965240e-01

```
Results for LinearRegression:
Mean Squared Error: 423184412892950656.0000
R^2 Score: -2673787593176513024.0000

Results for Ridge:
Mean Squared Error: 0.0137
R^2 Score: 0.9135

Results for DecisionTreeRegressor:
Mean Squared Error: 0.0357
R^2 Score: 0.7744

Results for RandomForestRegressor:
Mean Squared Error: 0.0174
R^2 Score: 0.8898

Results for XGBRegressor:
Mean Squared Error: 0.0179
R^2 Score: 0.8869

Results for KNeighborsRegressor:
Mean Squared Error: 0.0322
R^2 Score: 0.7965
```

Figure 5.1: Mean squared error and R^2 score of models

The table compares the performance of six regression models based on their Mean Squared Error (MSE) and R_2 score. Ridge Regression achieves a high R_2 score of 0.91 with relatively low error, indicating strong predictive accuracy. Random Forest Regressor and XGBoost Regressor also perform well, with R_2 scores of 0.89 and 0.87, respectively, and low MSE values, suggesting their robustness in capturing data patterns. Decision Tree Regressor and K-Neighbors Regressor have slightly lower R_2 scores, showing moderate accuracy. Linear Regression performs poorly, with a massive MSE and a highly negative R_2, indicating a failure to model the dataset with any degree of accuracy.



Linear Regression	Failed
Ridge regression	0.13614
Random Forest	0.14370
XGBoost	0.13169
Decision Tree	0.17547
K-Nearest Neighbot	0.17523

Figure 5.2: Result

The table presents the performance of various regression models in predicting house prices in Kaggle, excluding Linear Regression, which failed. XGBoost demonstrates the best performance with the lowest error (0.13169), followed closely by Ridge Regression with an error of 0.13614. Random Forest achieves a slightly higher error of 0.14370, indicating solid performance. Decision Tree and K-Nearest Neighbors have the highest errors of 0.17547 and 0.17523, respectively, reflecting comparatively less accurate predictions. Overall, XGBoost and Ridge Regression emerge as the top-performing models for this task.

5.2 Conclusions

Within the context of this dataset, it can be seen that the XGBoost, Gradient-Boosted Decision Tree, Random Forest, and Ridge Regression models provided reasonable performance. Each of these models performed well under the conditions provided, yielding sufficiently accurate predictions for practical application in fields such as real estate investment and market forecasting. These results suggest that the models can serve as valuable tools for investors, buyers, and real estate companies, providing data-driven insights that can help them make more informed decisions.

However, it is also readily apparent that some challenges remain, mainly that the quality and richness of the data have major ramifications on the performance of the models. For example, outliers or incorrect feature values may lead to overfitting or underfitting, which in turn affects the generalization capability of the models. Within this project, various steps were taken to address these issues, namely moving towards less-prone models such as Ridge Regression, as well as utilizing Random Forests as a way to improve on the predictive quality of Decision trees. Steps were also taken to address data quality issues and ensure that the dataset is as clean and comprehensive as possible in order to improve the robustness of the predictions.

5.3 Future Work

Future improvements to these models may include expanding the dataset to include more detailed features. More granular features may be added to the dataset, allowing for more predictive accuracy as more parameters are available to the models to come to their predictions.

Deep learning such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), could be applied to enhance the model's predictive ability even further.

SHORT NOTICES ON REFERENCE

Dataset: `https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data`

Implementations: `https://drive.google.com/drive/folders/13UGM-1ftZJD2xtCMMubMEp9sNcKX3z-2`

KNN:`https://www.geeksforgeeks.org/k-nearest-neighbours/`

Linear and ridge :`https://machinelearningcoban.com/2016/12/28/linearregression`

Tree:`https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost`