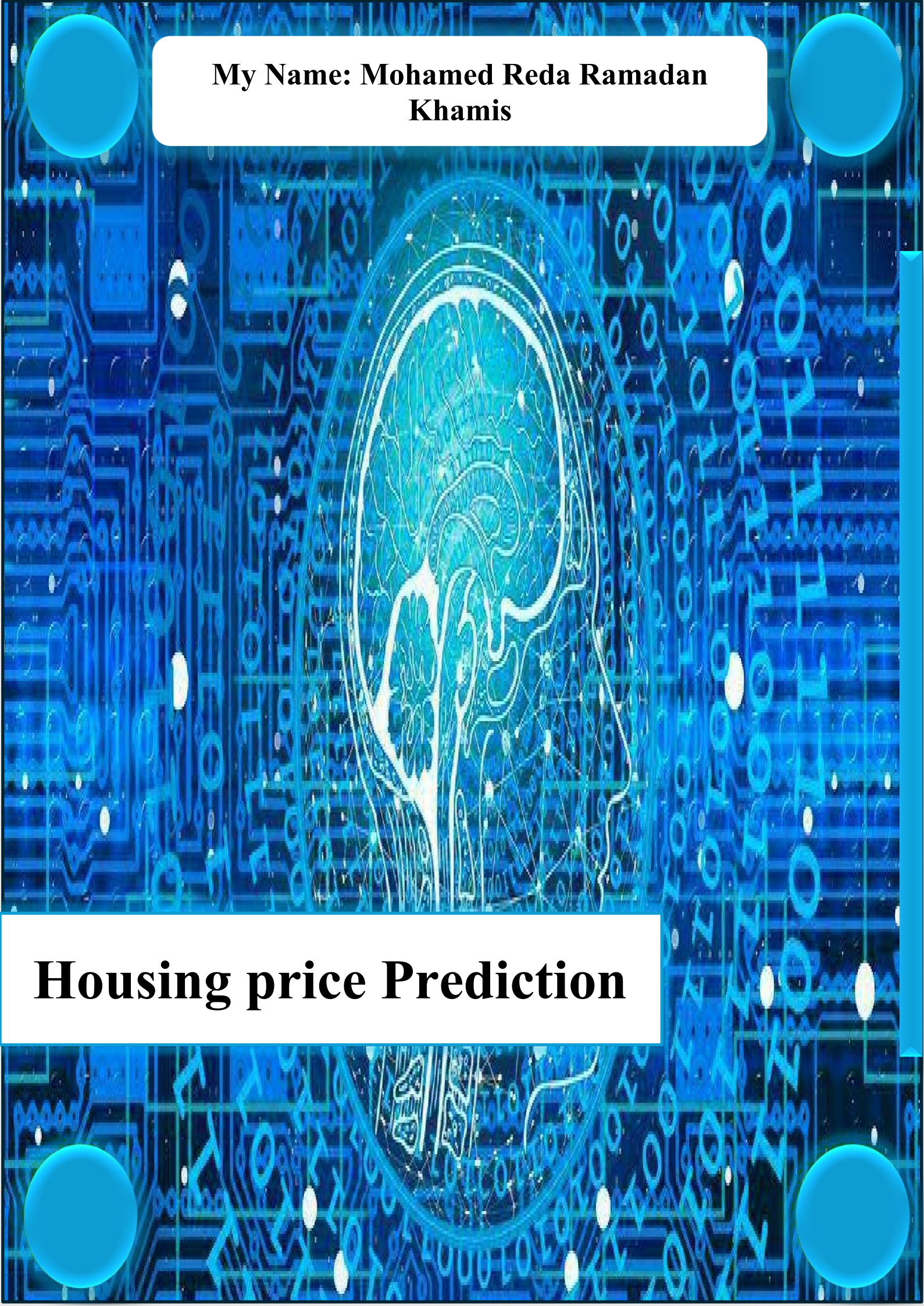
My Name: Mohamed Reda Ramadan Khamis



Housing price Prediction

Overview

- **housing market** is a critical sector in the economy, with pricing being influenced by numerous factors such as location, size, age, and amenities of a property. Accurately predicting housing prices is essential for buyers, sellers, and real estate professionals. This project leverages data mining techniques to analyze and predict housing prices based on various attributes.

- This project aims to assist buyers, sellers, and real estate agents in making informed decisions based on historical and current market data.

- **Housing prices** are influenced by a complex combination of factors, making them a vital area of study for real estate developers, policymakers, and investors. Accurate prediction of housing prices not only supports individual buyers and sellers but also drives informed decision-making in urban planning, taxation, and investment strategies. This project employs data mining techniques to analyze historical housing data, identify key determinants of price variation, and develop models for reliable price prediction.

- **The goal of this project** is to accurately predict housing prices based on a set of features such as location, size, and amenities. By understanding the relationships between these features and prices, the project aims to provide a comprehensive solution for stakeholders in the housing market.

1 Data Understanding

- **Data Understanding means studying and understanding the data, including knowing:**
  - ✓ **What is the task or the target of the data.**
  - ✓ **How much each column is important to the target.**

Drop Un-necessary Columns

```
df.drop(["Id"], axis=1, inplace=True)
display(df.head())  #default 5
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley
0	60	RL	65.0	8450	Pave	NaN
1	20	RL	80.0	9600	Pave	NaN
2	60	RL	68.0	11250	Pave	NaN
3	70	RL	60.0	9550	Pave	NaN
4	60	RL	84.0	14260	Pave	NaN

2 Check for Dtypes

- **Sometimes there are mistakes in representing the columns datatypes.**
  - ✓ **A numerical column could be represented as categorical, or a categorical column represented as object datatype, and so on.**
- **It's our responsibility to check for such mistakes and correct them.**

Check for Dtypes

```
pd.DataFrame({"Dtypes": df.dtypes, "Num_Uniq": df.nunique()}).T
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	BldgType
Dtypes	int64	object	float64	int64	object	object	object	object	object	object	object	object	object	object	object
Num_Uniq	15	5	110	1073	2	2	4	4	2	5	3	25	9	8	5

```
cols_to_change = ['MSSubClass', 'OverallQual', 'OverallCond',
'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'BedroomAbvGr',
'KitchenAbvGr', 'Fireplaces', 'GarageCars', 'PoolArea', 'MoSold', 'YrSold',
'MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature', 'SaleType',
'SaleCondition']

df[cols_to_change] = df[cols_to_change].astype("category")
pd.DataFrame({"Dtypes": df.dtypes, "Num_Uniq": df.nunique()}).T
```

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	BldgType
Dtypes	category	category	float64	int64	category	category	category	category	category	category	category	category	category	category	category
Num_Uniq	15	5	110	1073	2	2	4	4	2	5	3	25	9	8	5

3 Handle Null Values

- **Null Values must be handled, so that the data becomes complete and ready for machine learning.**
- **There are 3 options to handle Null values, that depend on the Null Values amount in each column:**
  - ✓ **If the column contains a small number of Null Values, you can simply delete rows that contain null values in this column.**
  - ✓ **If the column contains a huge number of Null Values, you should delete this column.**

- ✓ If the column contains a large number of Null Values, but not very large, you can replace null values with mean, median, or Mode.

- **In the 3 option:**
  - ✓ We replace Null Values with Mode if the column is categorical.
  - ✓ We replace Null Values with Mean if the column is numerical & normally distributed.
  - ✓ We replace Null Values with Median if the column numerical & not-normally distributed.

```
null = df.isnull().sum()
null_ratio = null / df.shape[0]
pd.DataFrame({"Null": null, "Null_ratio": null_ratio}).T
```

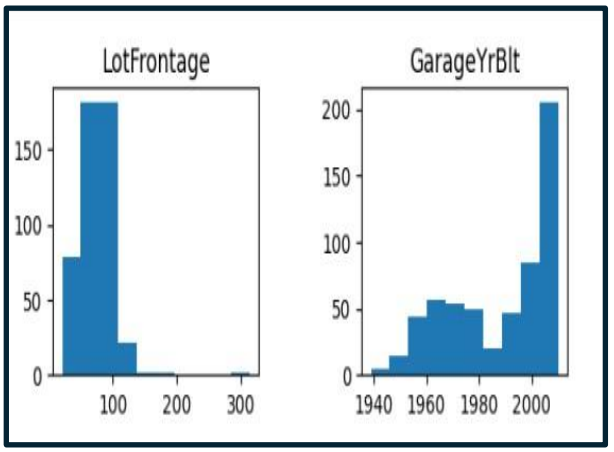
	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	BldgType
Null	0.0	0.0	259.000000	0.0	0.0	1369.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Null_ratio	0.0	0.0	0.177397	0.0	0.0	0.937671	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

OverallCond	YearBuilt	YearRemodAdd	RoofStyle	RoofMatl	Exterior1st	Exterior2nd	MasVnrType	MasVnrArea	ExterQual	ExterCond	Foundation	BsmtQual	BsmtCond	BsmtExposure
0.0	0.0	0.0	0.0	0.0	0.0	0.0	872.00000	8.000000	0.0	0.0	0.0	37.000000	37.000000	38.000000
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.59726	0.005479	0.0	0.0	0.0	0.025342	0.025342	0.026027

BsmtFinType1	BsmtFinSF1	BsmtFinType2	BsmtFinSF2	BsmtUnfSF	TotalBsmtSF	Heating	HeatingQC	CentralAir	Electrical	1stFlrSF	2ndFlrSF	LowQualFinSF	GrLivArea	BsmtFullBath	BsmtHa
37.000000	0.0	38.000000	0.0	0.0	0.0	0.0	0.0	0.0	1.000000	0.0	0.0	0.0	0.0	0.0	
0.025342	0.0	0.026027	0.0	0.0	0.0	0.0	0.0	0.0	0.000685	0.0	0.0	0.0	0.0	0.0	

FireplaceQu	GarageType	GarageYrBlt	GarageFinish	GarageCars	GarageArea	GarageQual	GarageCond	PavedDrive	WoodDeckSF	OpenPorchSF	EnclosedPorch	3SsnPorch	ScreenPorch	PoolAr
690.000000	81.000000	81.000000	81.000000	0.0	0.0	81.000000	81.000000	0.0	0.0	0.0	0.0	0.0	0.0	
0.472603	0.055479	0.055479	0.055479	0.0	0.0	0.055479	0.055479	0.0	0.0	0.0	0.0	0.0	0.0	

```
c1 = "MasVnrType MasVnrArea Electrical".split()
df.dropna(subset=c1, inplace=True)

c2 = "BsmtQual BsmtCond BsmtExposure BsmtFinType1 BsmtFinType2
GarageType GarageFinish GarageQual GarageCond".split()
modes = dict(df[c2].mode().iloc[0])

df.fillna(modes, inplace=True)

c2 = ["LotFrontage", "GarageYrBlt"]
medians = dict(df[c2].median())
medians

df.fillna(medians, inplace=True)

c3 = "Alley FireplaceQu PoolQC Fence MiscFeature".split()
df.drop(c3, axis=1, inplace=True)

null = df.isnull().sum()
null_ratio = null / df.shape[0]
pd.DataFrame({"Null": null, "Null_ratio": null_ratio}).T
```



	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	Condition2	BldgType	HouseStyle
Null	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Null_ratio	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

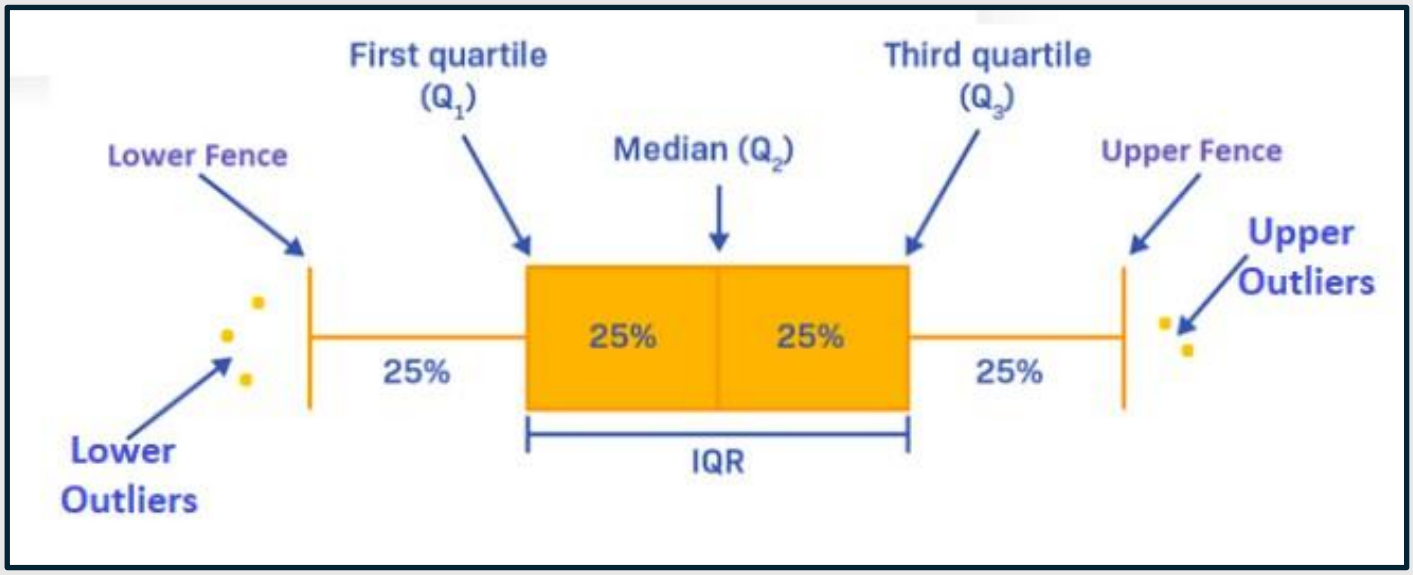
BsmtFinType1	BsmtFinSF1	BsmtFinType2	BsmtFinSF2	BsmtUnfSF	TotalBsmtSF	Heating	HeatingQC	CentralAir	Electrical	1stFlrSF	2ndFlrSF	LowQualFinSF	GrLivArea	BsmtFullBath
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

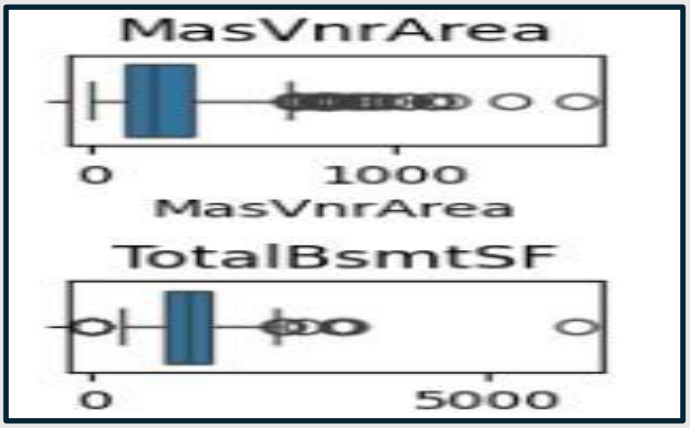
FullBath	HalfBath	BedroomAbvGr	KitchenAbvGr	KitchenQual	TotRmsAbvGrd	Functional	Fireplaces	GarageType	GarageYrBlt	GarageFinish	GarageCars	GarageArea	GarageQual
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

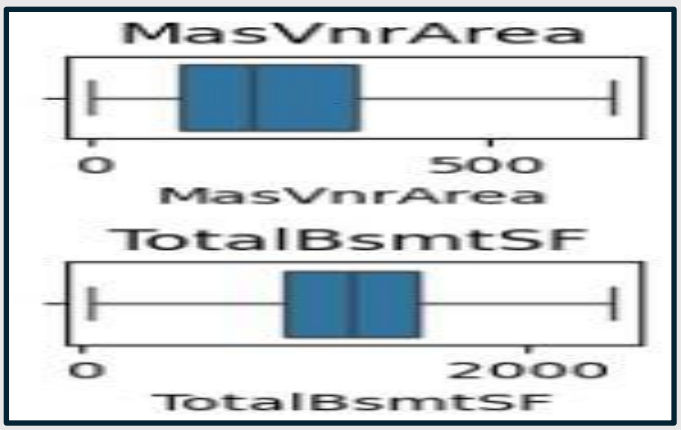
4 Handle Outliers

- **Outliers are values within the data, that has extremely large or extremely small values.**
- **Sometimes we need to remove outliers because they could cause many problems; such as, introducing misleading information about the data.**
- **Also some statistical measures could be sensitive to outliers; such as mean, and this is another reason for why we would like to remove outliers.**

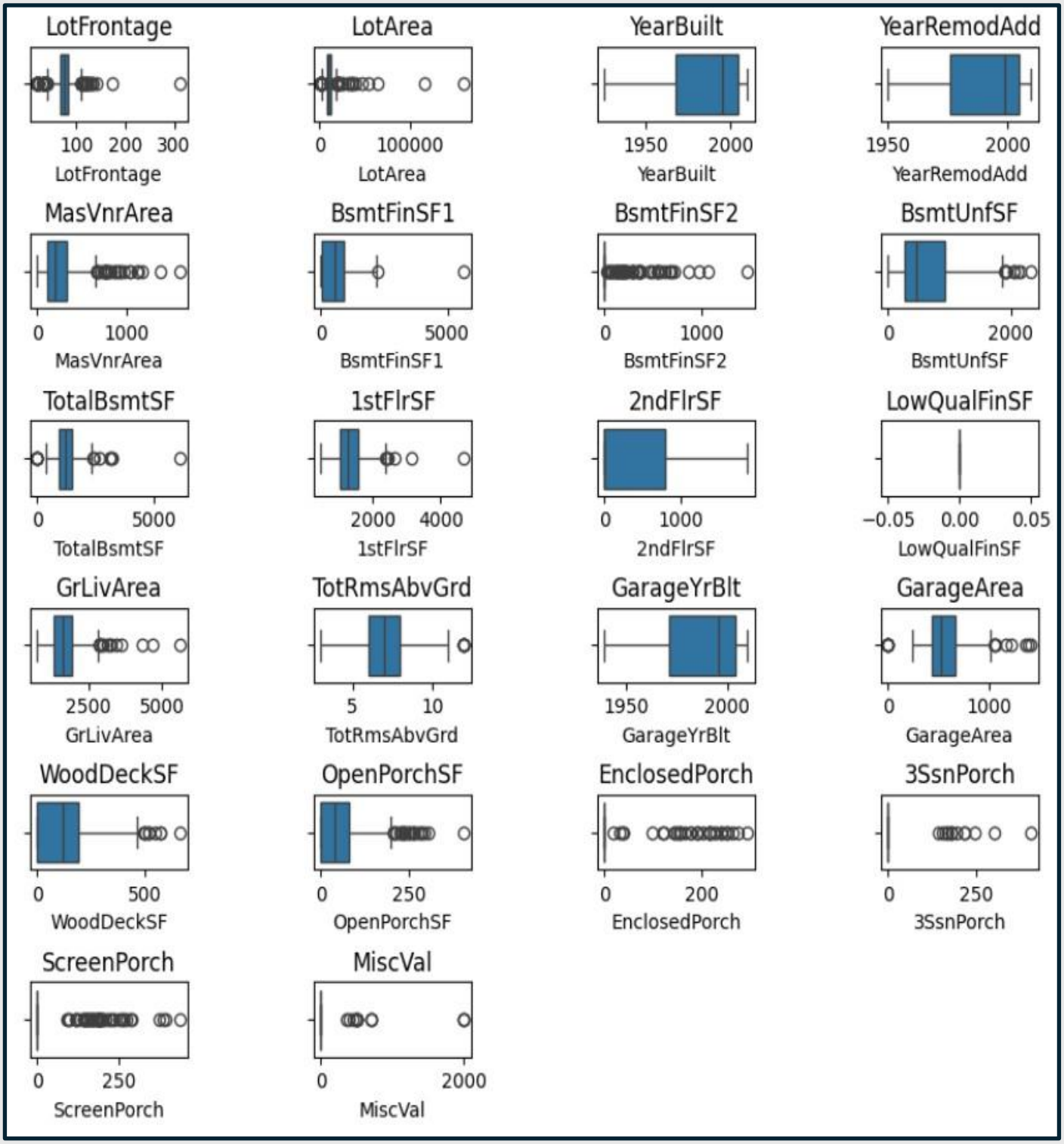
- **To check for outliers we Quartiles, where:**
  - ✓ **All values, greater than the Upper-Fence, are considered to be outliers.**
  - ✓ **All values, smaller than the Lower-Fence, are considered to be outliers.**



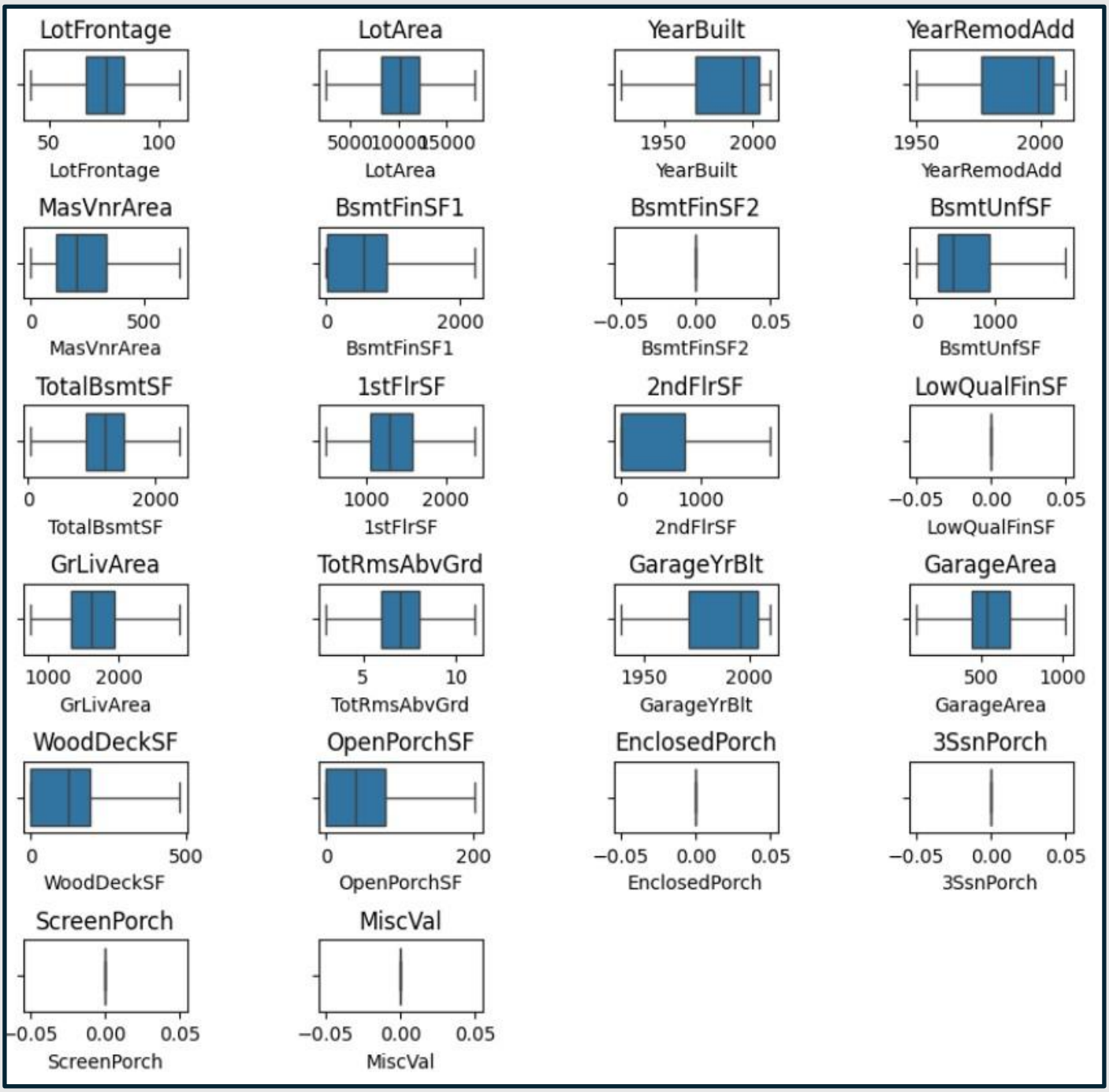




```
plt.figure(figsize=(10, 9))
for i, col in enumerate(num_cols[:-1]):
    plt.subplot(6, 4, i+1)
    plt.title(col)
    sns.boxplot(df[col], orient="h")
plt.subplots_adjust(wspace=.8, hspace=1.5)
plt.show()
```



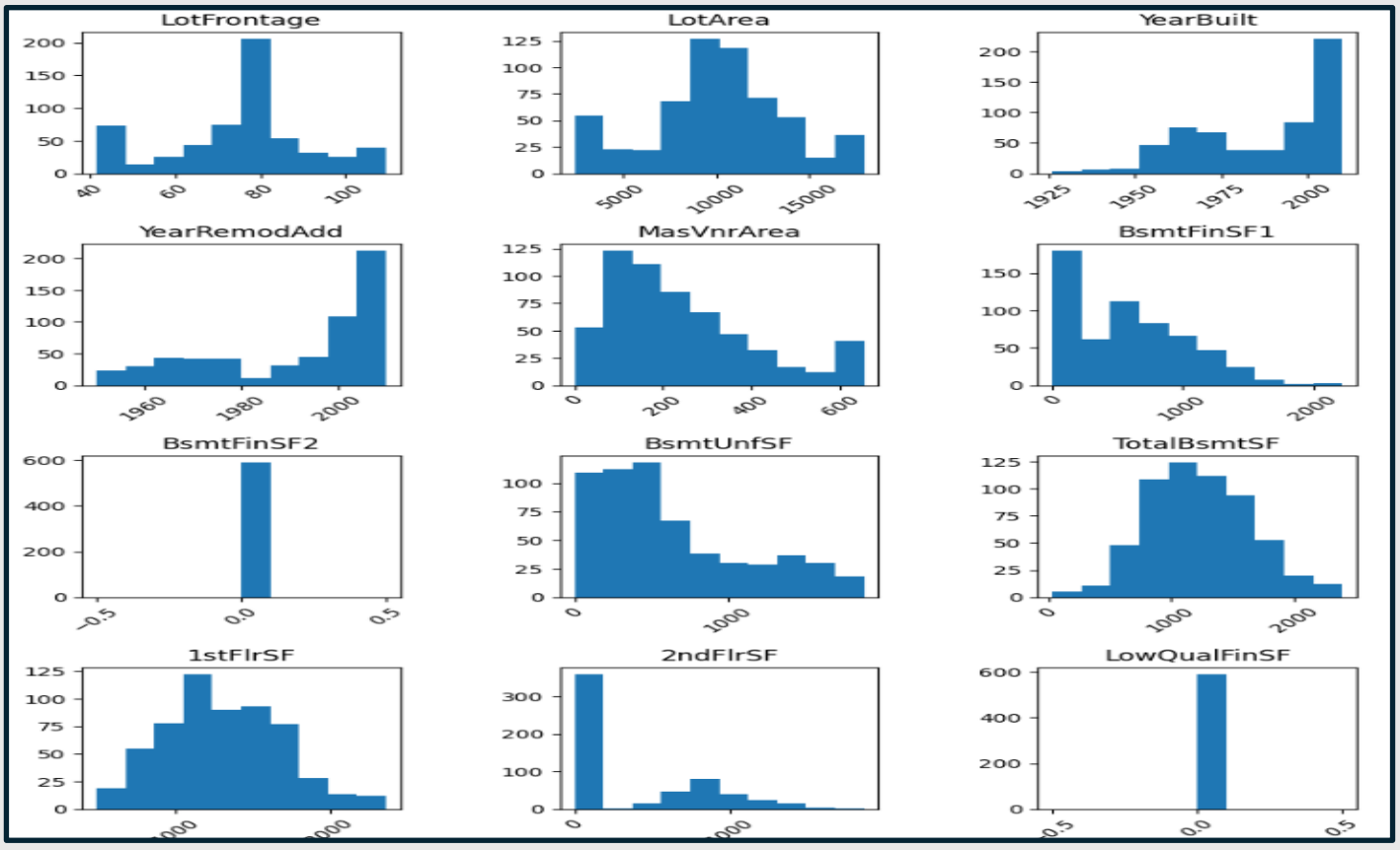
```
for col in num_cols[:-1]:
    Q1 = np.quantile(df[col], .25)
    Q3 = np.quantile(df[col], .75)
    IQR = Q3 - Q1
    upper = Q3 + 1.5 * IQR
    lower = Q1 - 1.5 * IQR
    upper_outliers = df[df[col] > upper][col].values
    lower_outliers = df[df[col] < lower][col].values
    df[col].replace(upper_outliers, upper, inplace=True)
    df[col].replace(lower_outliers, lower, inplace=True)
```



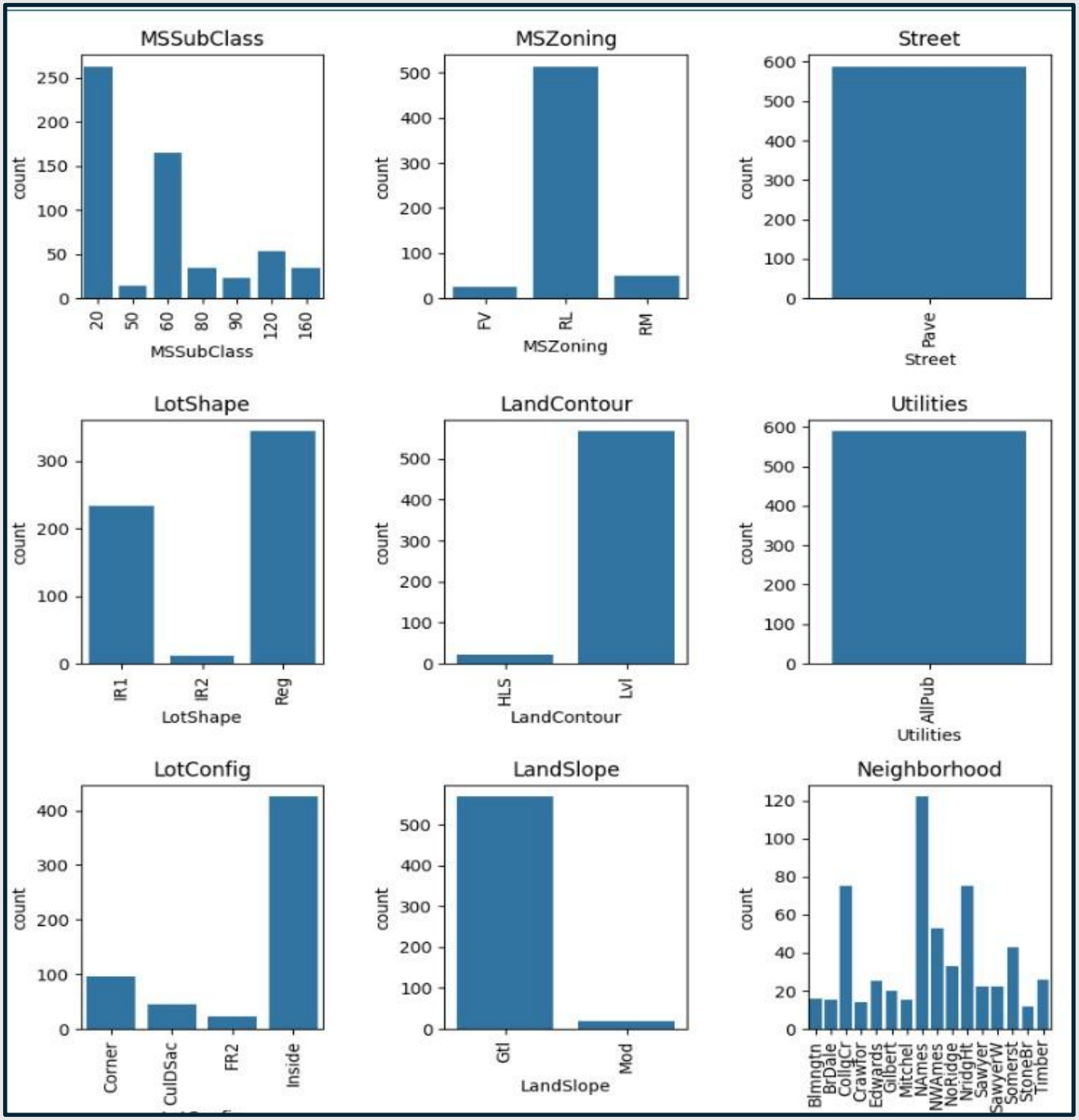
5 Visualization

- **Visualization is the process of creating graphs about our data, that helps use well understand & explore our data.**
- **Types of Graphs:**
  - ✓ **Data Distribution Graphs.**
  - ✓ **Outlier Detection Graphs.**
  - ✓ **Relationship Graphs.**
- **The most famous libraries used for visualization are Matplotlib & Seaborn.**

```
plt.figure(figsize=(10, 25))
for i, col in enumerate(num_cols):
    plt.subplot(8, 3, i+1)
    plt.title(col)
    plt.xticks(rotation=45)
    plt.hist(df[col])
plt.subplots_adjust(wspace=.5, hspace=.5)
plt.show()
```



```
plt.figure(figsize=(10, 75))
for i, col in enumerate(cat_cols):
    plt.subplot(18, 3, i+1)
    plt.title(col)
    plt.xticks(rotation=90)
    sns.countplot(x=col, data=df)
plt.subplots_adjust(wspace=.5, hspace=.5)
plt.show()
```



6 Sampling

- **Sampling is a data preprocessing technique where a subset of the data is selected to represent the entire dataset. It is used to reduce the size of the dataset for faster processing, especially in scenarios where the original dataset is too large to handle efficiently.**

```
# Perform random sampling to reduce the dataset to 80% of its original size
df = df.sample(frac=0.8, random_state=42)
```

7 Attribute Transformation

- **Attribute transformation alters the data by replacing a selected attribute by one or more new attributes, functionally dependent on the original one, to facilitate further analysis**

Goals of Attribute Transformation :

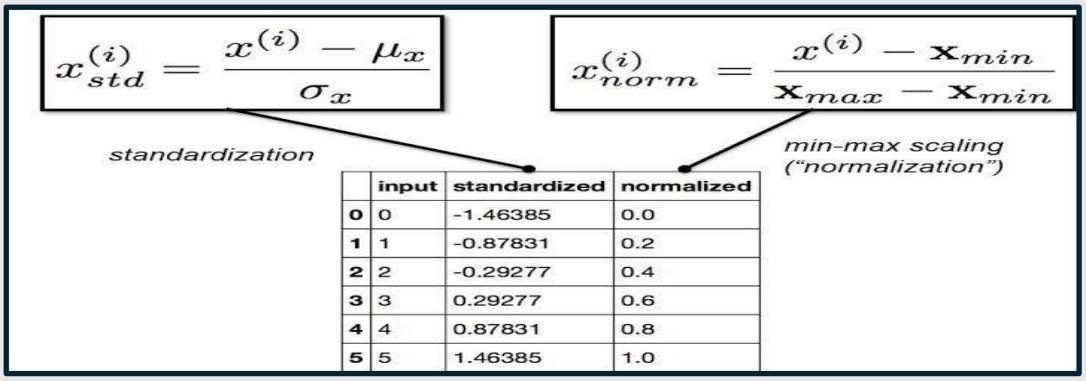
- **Normalization: Scale the data to bring it within a specific range (e.g., [0, 1] or [-1, 1]).**
- **Standardization: Adjust the attributes to have a mean of 0 and a standard deviation of 1.**
- **Reducing Skewness: Handle attributes with highly skewed distributions.**

```
# Transform "YearBuilt" to represent the age of the house in years
df['HouseAge'] = 2024 - df['YearBuilt']
```

```
# Drop the original "YearBuilt" column to avoid redundancy
df.drop(columns=['YearBuilt'], inplace=True)
```

8 Normalization

- **Normalization** is a data preprocessing technique used to transform features in a dataset to a common scale, improving the performance and accuracy of machine learning algorithms. The main goal of normalization is to eliminate the potential biases and distortions caused by the different scales of features. Some common normalization methods include min-max scaling, z-score standardization, and log transformation.



	input	standardized	normalized
0	0	-1.46385	0.0
1	1	-0.87831	0.2
2	2	-0.29277	0.4
3	3	0.29277	0.6
4	4	0.87831	0.8
5	5	1.46385	1.0

```
num_cols = X.select_dtypes("number").columns
scaler = MinMaxScaler()
X[num_cols] = scaler.fit_transform(X[num_cols])
```

	LotFrontage	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtUnfSF	TotalBsmtSF	1stFlrSF	2ndFlrSF	GrLivArea	TotRmsAbvGrd	GarageYrBlt	GarageArea	WoodDeckSF	OpenPorchSF	HouseAge
1290	0.507353	0.233333	0.131498	0.456500	0.073200	0.475170	0.354531	0.000000	0.191083	0.375	0.352113	0.422414	0.472917	0.000000	0.547619
688	0.272059	0.950000	0.000000	0.425436	0.251426	0.588739	0.496025	0.000000	0.317056	0.500	0.957746	0.511853	0.291667	0.000000	0.035714
1273	1.000000	0.933333	0.128440	0.323692	0.159130	0.418598	0.463169	0.000000	0.287804	0.250	0.281690	0.237069	0.000000	0.000000	0.607143
796	0.433824	0.450000	0.226300	0.000000	0.350086	0.265897	0.425013	0.000000	0.253833	0.500	0.535211	0.469828	0.287500	0.000000	0.392857
506	0.566176	0.733333	0.191131	0.171075	0.210582	0.315664	0.308426	0.451923	0.549186	0.625	0.760563	0.408405	0.379167	0.400000	0.202381
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1366	0.389706	0.816667	0.241590	0.284975	0.127304	0.356497	0.211447	0.485043	0.492097	0.500	0.845070	0.534483	0.000000	0.434568	0.130952
1295	0.419118	0.300000	0.256881	0.457400	0.019096	0.432635	0.301537	0.000000	0.143902	0.250	0.408451	0.211207	0.741667	0.000000	0.500000
1043	0.654412	0.666667	0.151376	0.488464	0.206869	0.612558	0.555909	0.425748	0.746402	0.875	0.718310	0.454741	0.400000	0.597531	0.238095
235	0.000000	0.350000	0.923547	0.161171	0.066304	0.190610	0.000000	0.269231	0.113234	0.250	0.450704	0.185345	0.000000	0.000000	0.464286
545	0.507353	0.633333	0.272171	0.451097	0.000000	0.497288	0.473768	0.430556	0.677518	0.750	0.690141	0.747845	0.000000	0.000000	0.261905

470 rows × 15 columns

9 Encoding

- **Encoding means representing the string values as numbers so that the machine can understand them, where computers can only apply mathematical operations over numbers.**

Encoding Techniques are:

- **Ordinal Encoding:**
  - ✓ Used for ordinal columns.
- **One Hot Encoding:**
  - ✓ Used for nominal columns with small number of unique values.
- **Binary Encoding:**
  - ✓ Used for nominal columns with large number of unique values.

```
ordinal_cols = ['ExterQual', 'BsmtQual', 'BsmtExposure',
'BsmtFinType1', 'HeatingQC', 'KitchenQual', 'Foundation',
'GarageType', 'GarageFinish']
pd.DataFrame(df[ordinal_cols].nunique()).T
```

	ExterQual	BsmtQual	BsmtExposure	BsmtFinType1	HeatingQC	KitchenQual	Foundation	GarageType	GarageFinish
0	3	3	4	6	3	3	2	3	3

```
nominal_cols = ['MSZoning', 'LotShape', 'LandContour', 'LotConfig',
'Neighborhood', 'HouseStyle', 'RoofStyle', 'Exterior1st', 'Exterior2nd',
'MasVnrType']
pd.DataFrame(df[nominal_cols].nunique()).T
```

MSZoning	LotShape	LandContour	LotConfig	Neighborhood	HouseStyle	RoofStyle	Exterior1st	Exterior2nd	MasVnrType
3	3	2	4	16	5	2	6	6	3

10 Dimension Reduction(PCA)

It transforms a large data set into a lower dimensional space, extracting the most informative features while preserving the most relevant information from the initial data set.

```
# Use PCA to reduce dimensions to 10 principal components
pca = PCA(n_components=10)
pca_features = pca.fit_transform(X[num_cols])

# Create a new DataFrame for PCA features
pca_df = pd.DataFrame(pca_features, columns=[f'PCA_{i+1}' for i in range(10)])

# Add PCA features back to the dataset
X = pd.concat([X.reset_index(drop=True), pca_df], axis=1)
```

PCA_1	PCA_2	PCA_3	PCA_4	PCA_5	PCA_6	PCA_7	PCA_8	PCA_9	PCA_10
-0.680760	-0.096487	-0.117001	0.400599	-0.144940	0.245338	-0.087448	-0.036708	-0.146309	0.064363
0.152872	0.590197	-0.000420	0.217509	-0.146898	0.013702	-0.227573	-0.232509	-0.028516	-0.030651
-0.479135	0.007360	-0.238639	0.183412	0.059167	-0.060177	-0.492547	0.341512	0.516101	-0.124890
-0.434615	0.078086	-0.141396	-0.136506	-0.211818	0.102753	-0.039486	0.016856	-0.022303	0.132907
0.101607	-0.064831	0.368524	-0.135213	-0.013759	0.257207	-0.196581	0.053601	-0.013666	-0.088477
...	...	...	...	...	...	...	...	...	...
0.100913	0.111811	0.513256	-0.140278	0.201986	-0.107120	-0.189700	-0.015332	-0.065365	0.003557
-0.689861	0.020971	0.015169	0.504717	-0.278245	0.395680	0.144751	0.078476	-0.102496	-0.084456
0.365592	-0.365511	0.192782	0.118995	0.183602	0.317138	-0.271996	-0.241975	-0.017701	-0.190801
-0.826563	0.004854	0.466756	-0.076911	-0.163157	-0.443570	0.583030	0.128647	-0.017489	-0.032213
0.072692	-0.238962	0.261326	0.153489	-0.117004	-0.307905	-0.418835	-0.266664	-0.071315	0.123555

11 Feature Subset Selection

Feature Selection is the most critical pre-processing activity in any machine learning process. It intends to select a subset of attributes or features that makes the most meaningful contribution to a machine learning activity.

```
selector = SelectKBest(score_func=f_classif, k=10)
X_selected = selector.fit_transform(X, y)
selected_features = X.columns[selector.get_support()]
X = X[selected_features.to_list()]
```

	OverallQual	ExterQual	BsmtQual	TotalBsmtSF	GrLivArea	KitchenQual	GarageCars	GarageArea	HouseAge	PCA_1
0	5	2	3	0.475170	0.191083	2	2	0.422414	0.547619	-0.680760
1	8	3	4	0.588739	0.317056	3	2	0.511853	0.035714	0.152872
2	6	2	3	0.418598	0.287804	4	1	0.237069	0.607143	-0.479135
3	6	2	3	0.265897	0.253833	2	2	0.469828	0.392857	-0.434615
4	8	3	4	0.315664	0.549186	3	2	0.408405	0.202381	0.101607
...	...	...	...	...	...	...	...	...	...	...
465	7	3	4	0.356497	0.492097	3	2	0.534483	0.130952	0.100913
466	5	2	3	0.432635	0.143902	2	1	0.211207	0.500000	-0.689861
467	7	2	4	0.612558	0.746402	3	2	0.454741	0.238095	0.365592
468	6	2	3	0.190610	0.113234	2	1	0.185345	0.464286	-0.826563
469	7	3	4	0.497288	0.677518	3	3	0.747845	0.261905	0.072692

470 rows × 10 columns

12 Display Processed Data

	OverallQual	ExterQual	BsmtQual	TotalBsmtSF	GrLivArea	KitchenQual	GarageCars	GarageArea	HouseAge	PCA_1
0	5	2	3	0.475170	0.191083	2	2	0.422414	0.547619	-0.680760
1	8	3	4	0.588739	0.317056	3	2	0.511853	0.035714	0.152872
2	6	2	3	0.418598	0.287804	4	1	0.237069	0.607143	-0.479135
3	6	2	3	0.265897	0.253833	2	2	0.469828	0.392857	-0.434615
4	8	3	4	0.315664	0.549186	3	2	0.408405	0.202381	0.101607
...	...	...	...	...	...	...	...	...	...	...
465	7	3	4	0.356497	0.492097	3	2	0.534483	0.130952	0.100913
466	5	2	3	0.432635	0.143902	2	1	0.211207	0.500000	-0.689861
467	7	2	4	0.612558	0.746402	3	2	0.454741	0.238095	0.365592
468	6	2	3	0.190610	0.113234	2	1	0.185345	0.464286	-0.826563
469	7	3	4	0.497288	0.677518	3	3	0.747845	0.261905	0.072692

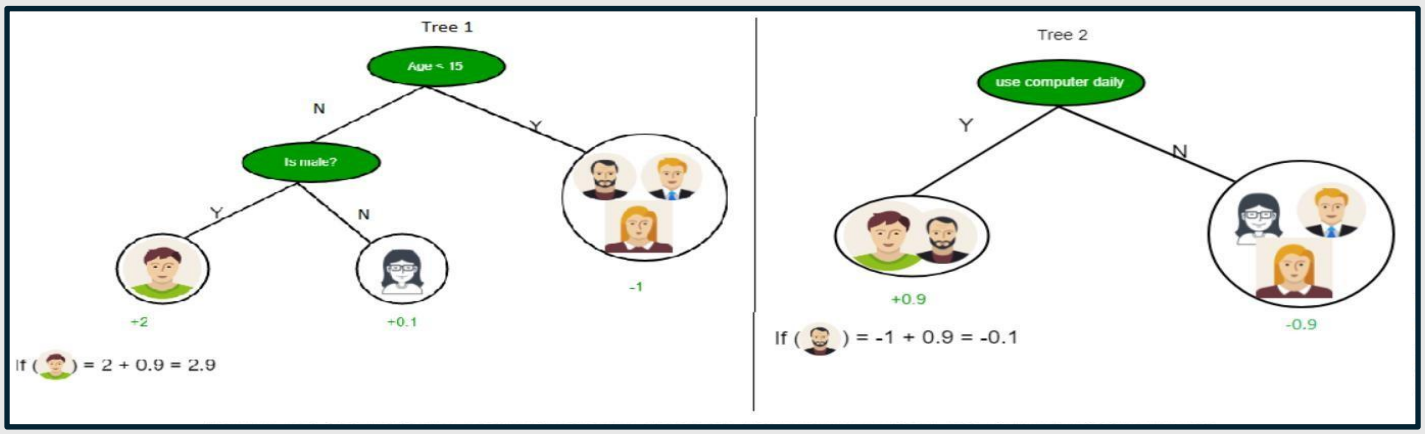
470 rows × 10 columns

```
# Assuming X_df is your preprocessed DataFrame
X_df.to_csv("X_df.csv", index=False) # Save as CSV
```

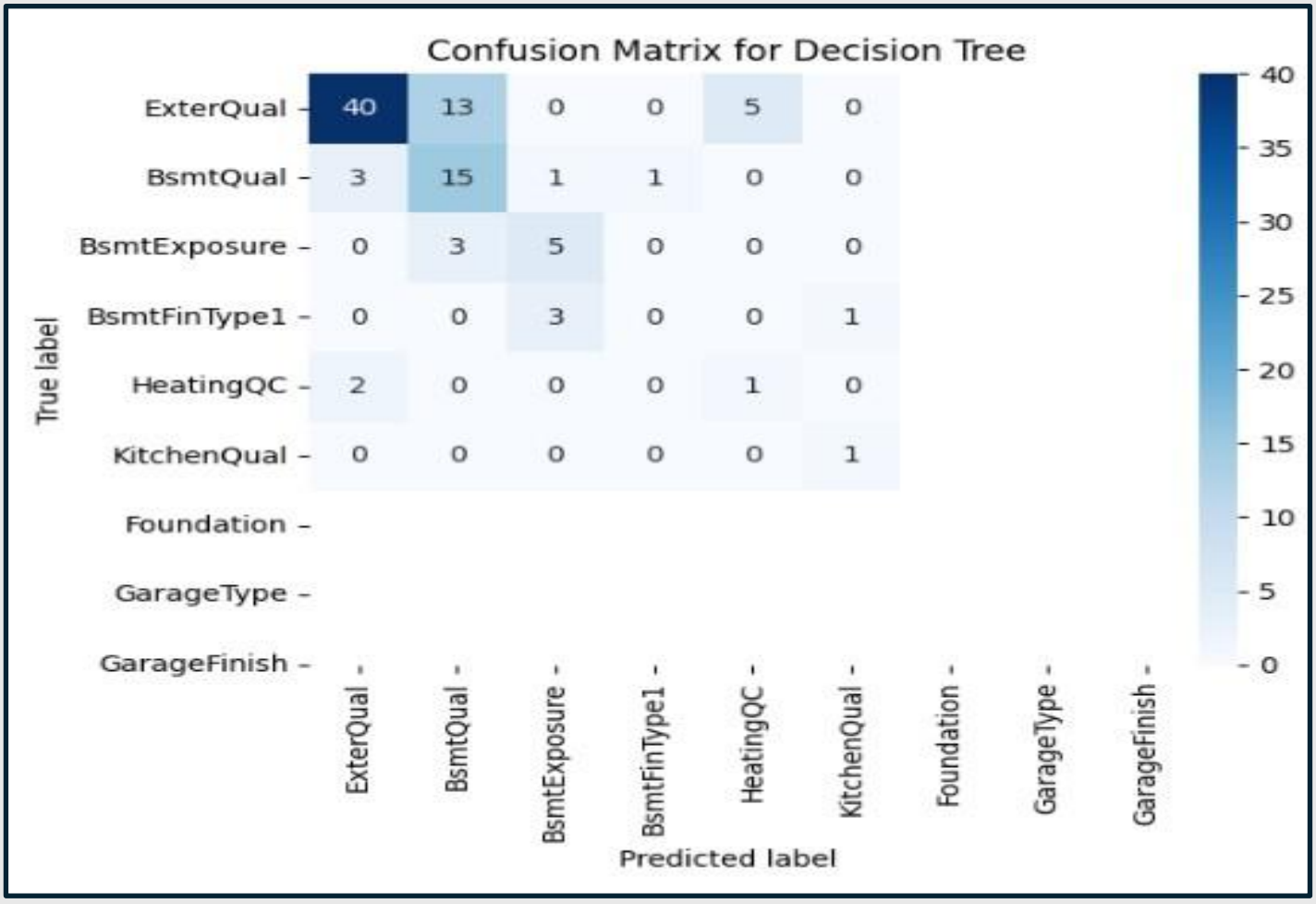
1 Decision Tree Classifier

- **A decision tree** is a type of supervised learning algorithm that is commonly used in machine learning to model and predict outcomes based on input data. It is a tree-like structure where each internal node tests on attribute, each branch corresponds to attribute value and each leaf node represents the final decision or prediction. The decision tree algorithm falls under the category of supervised learning.

- **Root Node:** A decision tree's root node, which represents the original choice or feature from which the tree branches, is the highest node.
- **Internal Nodes (Decision Nodes):** Nodes in the tree whose choices are determined by the values of particular attributes. There are branches on these nodes that go to other nodes.
- **Leaf Nodes (Terminal Nodes):** The branches' termini, when choices or forecasts are decided upon. There are no more branches on leaf nodes.
- **Branches (Edges):** Links between nodes that show how decisions are made in response to particular circumstances.
- **Splitting:** The process of dividing a node into two or more sub-nodes based on a decision criterion. It involves selecting a feature and a threshold to create subsets of data.
- **Parent Node:** A node that is split into child nodes. The original node from which a split originates.



```
# Initialize the classifier model
dt_model = DecisionTreeClassifier(random_state=42)

# Train the model
dt_model.fit(X_train, y_train)

# Predict the target variable
y_pred_dt = dt_model.predict(X_test)

# Print the predicted values
print("Predictions from Decision Tree Classifier (y_pred_dt):")
print(y_pred_dt)

# Metrics
accuracy_dt = accuracy_score(y_test, y_pred_dt) * 100
recall_dt = recall_score(y_test, y_pred_dt, average='macro') * 100
precision_dt = precision_score(y_test, y_pred_dt, average='macro') * 100
f1_dt = f1_score(y_test, y_pred_dt, average='macro') * 100

# Create a DataFrame to display the metrics
metrics_data = {
    "Metric": ["Accuracy", "Recall", "Precision", "F1 Score"],
    "Decision Tree": [accuracy_dt, recall_dt, precision_dt, f1_dt]}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the results in DataFrame
display(metrics_df)

# Confusion Matrix visualization
cm_dt = confusion_matrix(y_test, y_pred_dt)
sns.heatmap(cm_dt, annot=True, fmt='d', cmap='Blues',
xticklabels=labels, yticklabels=labels)
plt.title("Confusion Matrix for Decision Tree")
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()
```



Classification Report as DataFrame:

	precision	recall	f1-score	support
100k-200k	0.888889	0.689655	0.776699	58.000000
200k-300k	0.483871	0.750000	0.588235	20.000000
300k-400k	0.555556	0.625000	0.588235	8.000000
400k-500k	0.000000	0.000000	0.000000	4.000000
<100k	0.166667	0.333333	0.222222	3.000000
>500k	0.500000	1.000000	0.666667	1.000000
accuracy	0.659574	0.659574	0.659574	0.659574
macro avg	0.432497	0.566331	0.473676	94.000000
weighted avg	0.709334	0.659574	0.668643	94.000000

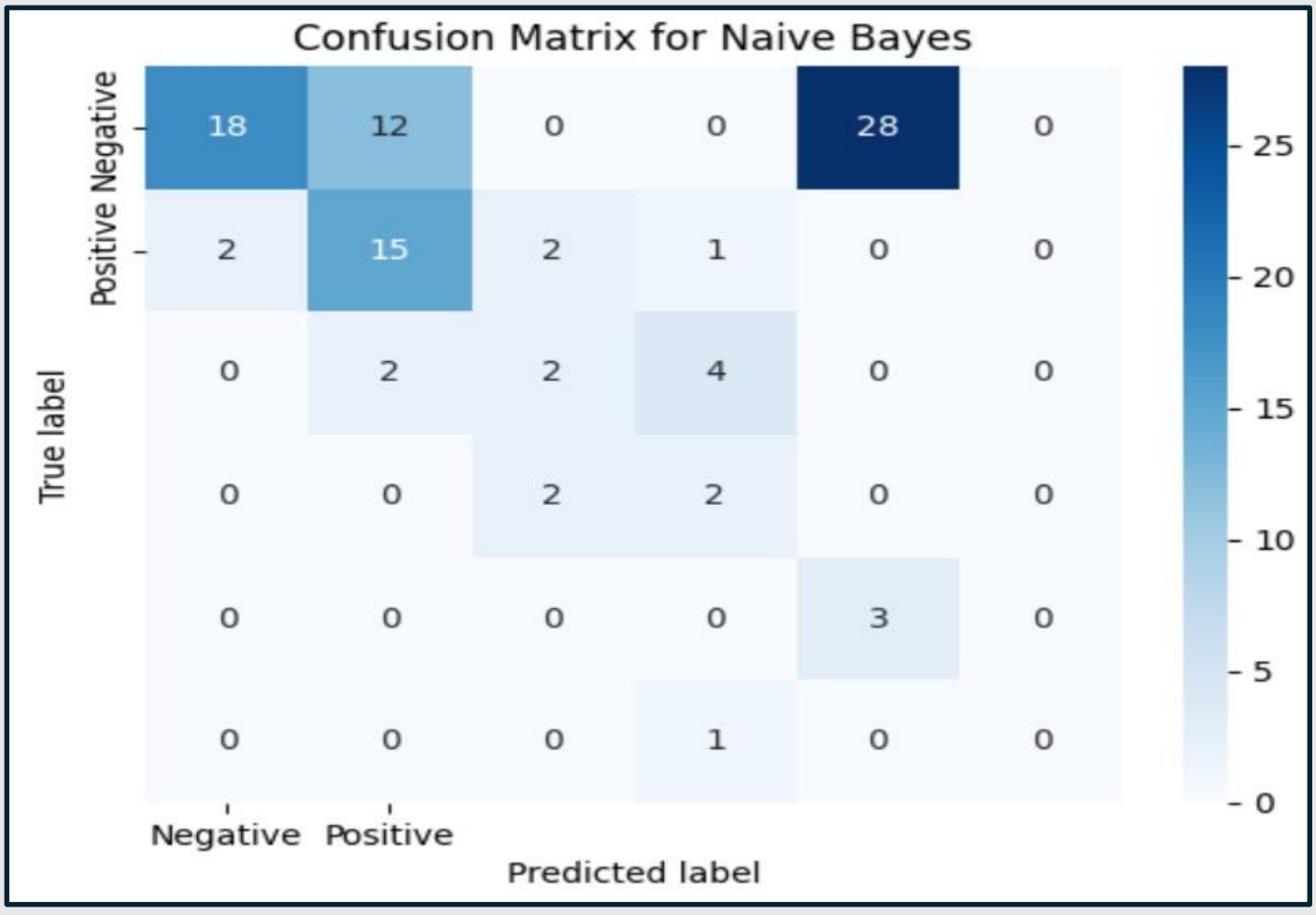
2 Naïve Bayes Classifier

- **Naive Bayes classifiers** are a collection of classification algorithms based on Bayes' Theorem. It is not a single algorithm but a family of algorithms where all of them share a common principle, i.e. every pair of features being classified is independent of each other. To start with, let us consider a dataset.

- **Naïve Bayes algorithm** is used for classification problems. It is highly used in text classification. In text classification tasks, data contains high dimension (as each word represent one feature in the data). It is used in spam filtering, sentiment detection, rating classification etc. The advantage of using naïve Bayes is its speed. It is fast and making prediction is easy with high dimension of data.

```
# Initialize the model
nb_model = GaussianNB()

# Train the model
nb_model.fit(X_train, y_train)

# Predict the target variable
y_pred_nb = nb_model.predict(X_test)

# Print the predicted values for Naive Bayes
print("Predictions from Naive Bayes (y_pred_nb):")
print(y_pred_nb)

# Confusion Matrix
cm_nb = confusion_matrix(y_test, y_pred_nb)

# Metrics
accuracy_nb = accuracy_score(y_test, y_pred_nb)*100
recall_nb = recall_score(y_test, y_pred_nb,average='macro')*100
precision_nb = precision_score(y_test, y_pred_nb,average='macro')*100
f1_nb = f1_score(y_test, y_pred_nb,average='macro')*100
```

```
# Create a DataFrame to display the metrics
metrics_data = {
    "Metric": ["Accuracy", "Recall", "Precision", "F1 Score"],
    "Naive Bayes": [accuracy_nb, recall_nb, precision_nb, f1_nb]}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the results in DataFrame
display(metrics_df)

# Confusion Matrix visualization
sns.heatmap(cm_nb, annot=True, fmt='d', cmap='Blues',
xticklabels=["Negative", "Positive"], yticklabels=["Negative",
"Positive"])
plt.title("Confusion Matrix for Naive Bayes")
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()
```



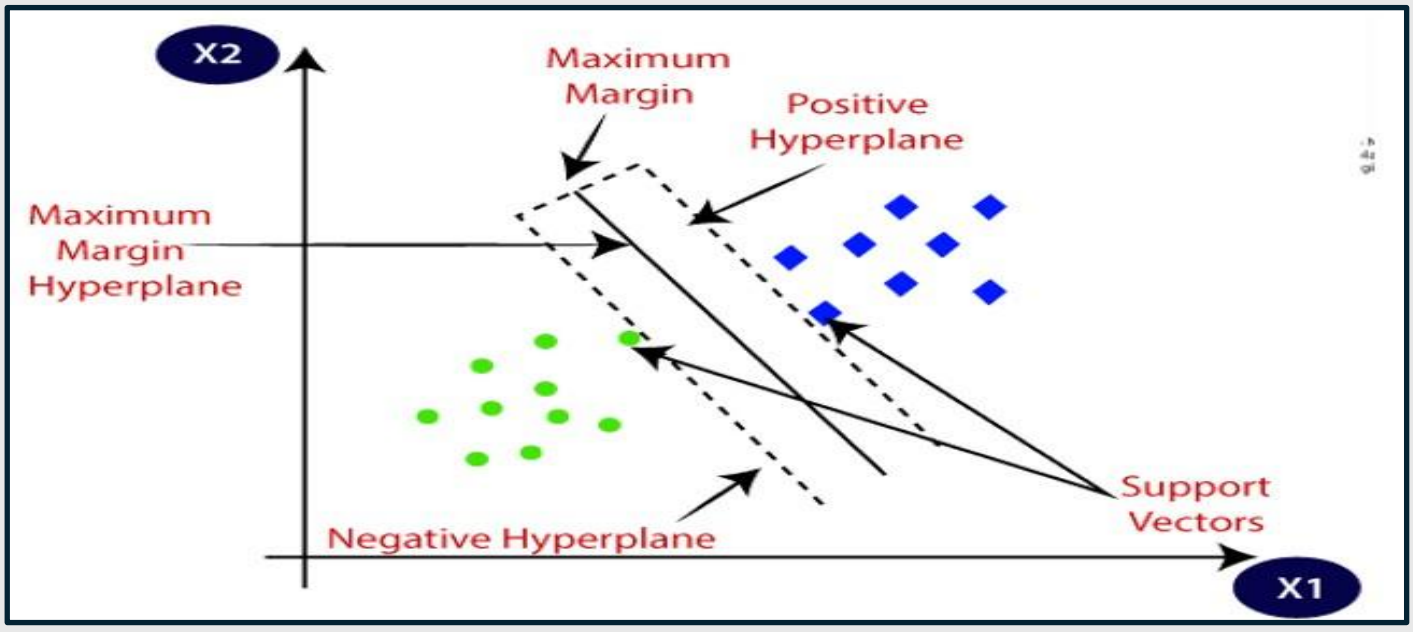
```
...   Classification Report as DataFrame:
```

...

	precision	recall	f1-score	support
100k-200k	0.900000	0.310345	0.461538	58.000000
200k-300k	0.517241	0.750000	0.612245	20.000000
300k-400k	0.333333	0.250000	0.285714	8.000000
400k-500k	0.250000	0.500000	0.333333	4.000000
<100k	0.096774	1.000000	0.176471	3.000000
>500k	0.000000	0.000000	0.000000	1.000000
accuracy	0.425532	0.425532	0.425532	0.425532
macro avg	0.349558	0.468391	0.311550	94.000000
weighted avg	0.707466	0.425532	0.459176	94.000000

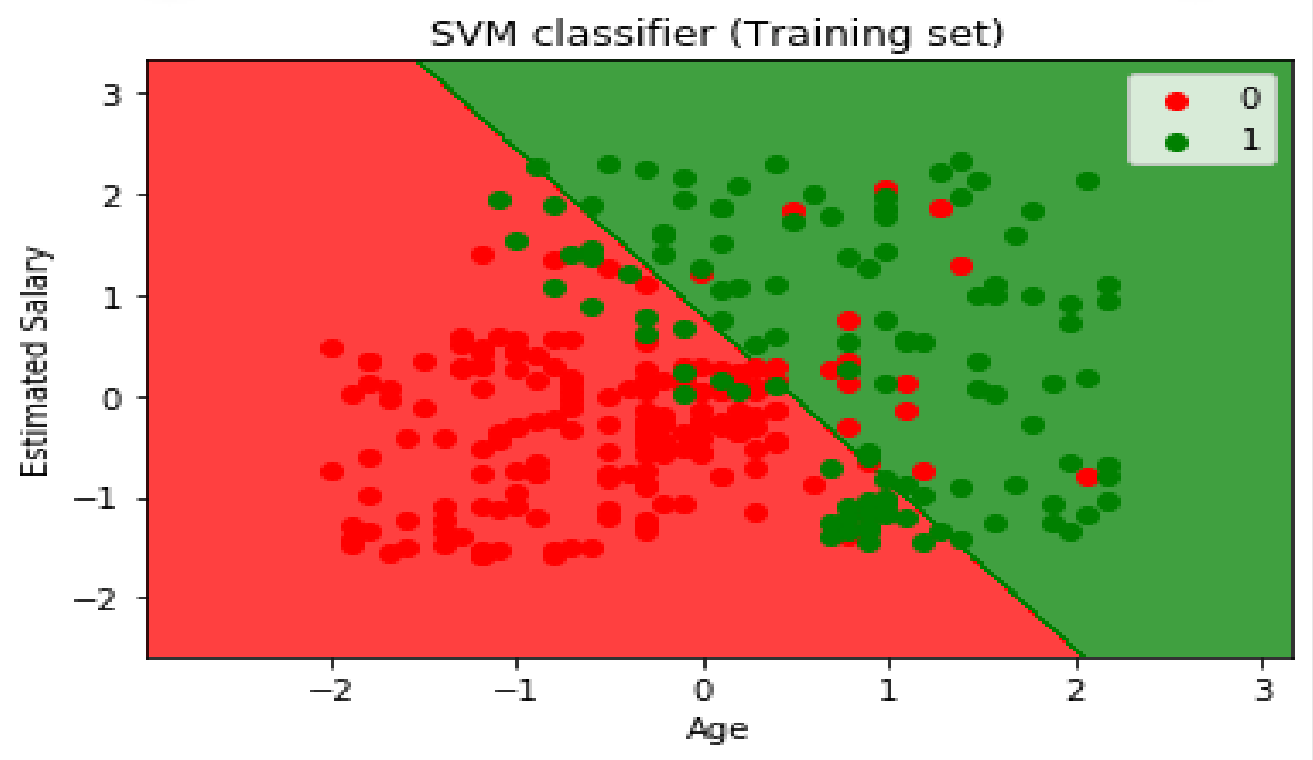
3 SVM

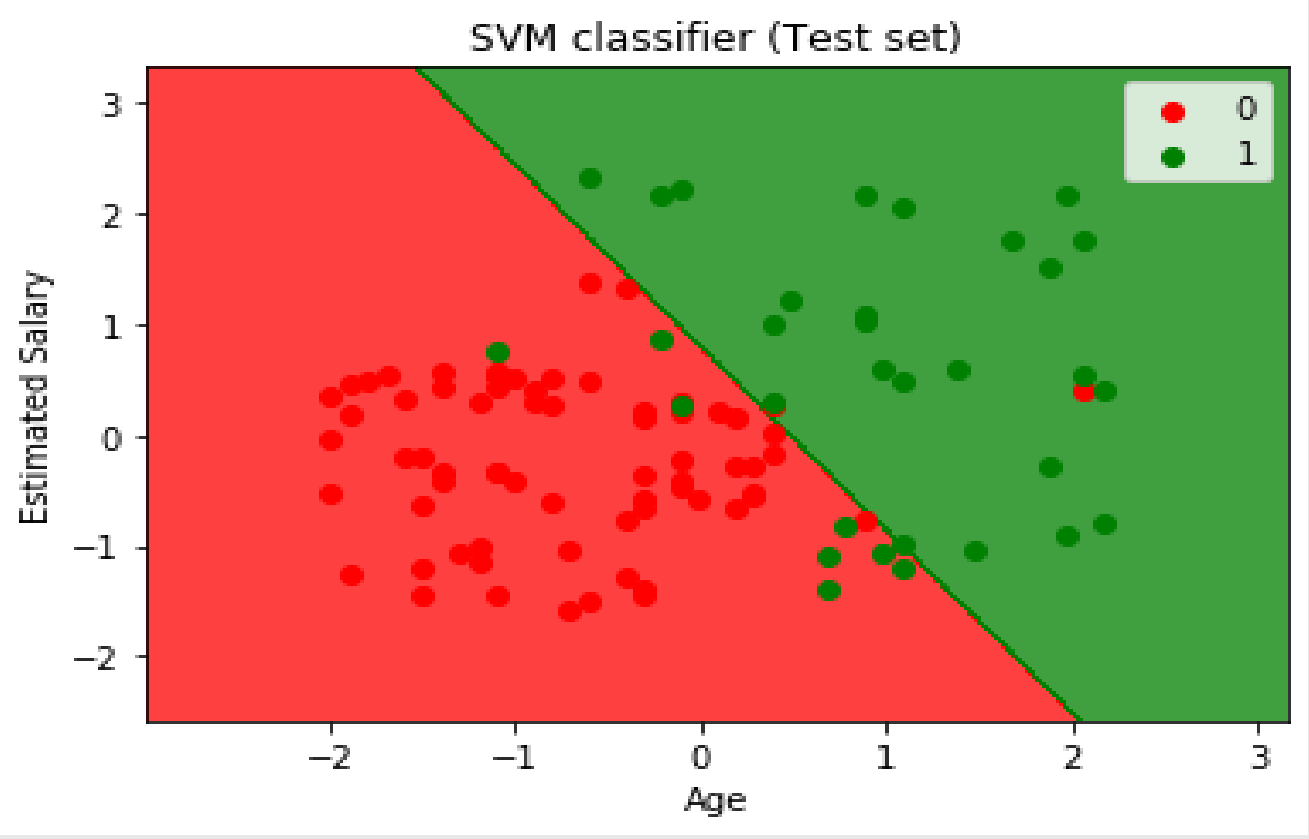
- **The goal of the SVM algorithm is to create the best decision boundary that can segregate n-dimensional space into classes. This best decision boundary is called a Hyperplane.**
- **SVM chooses the extreme points/vectors that help in creating the Hyperplane. These extreme cases are called as Support Vectors.**

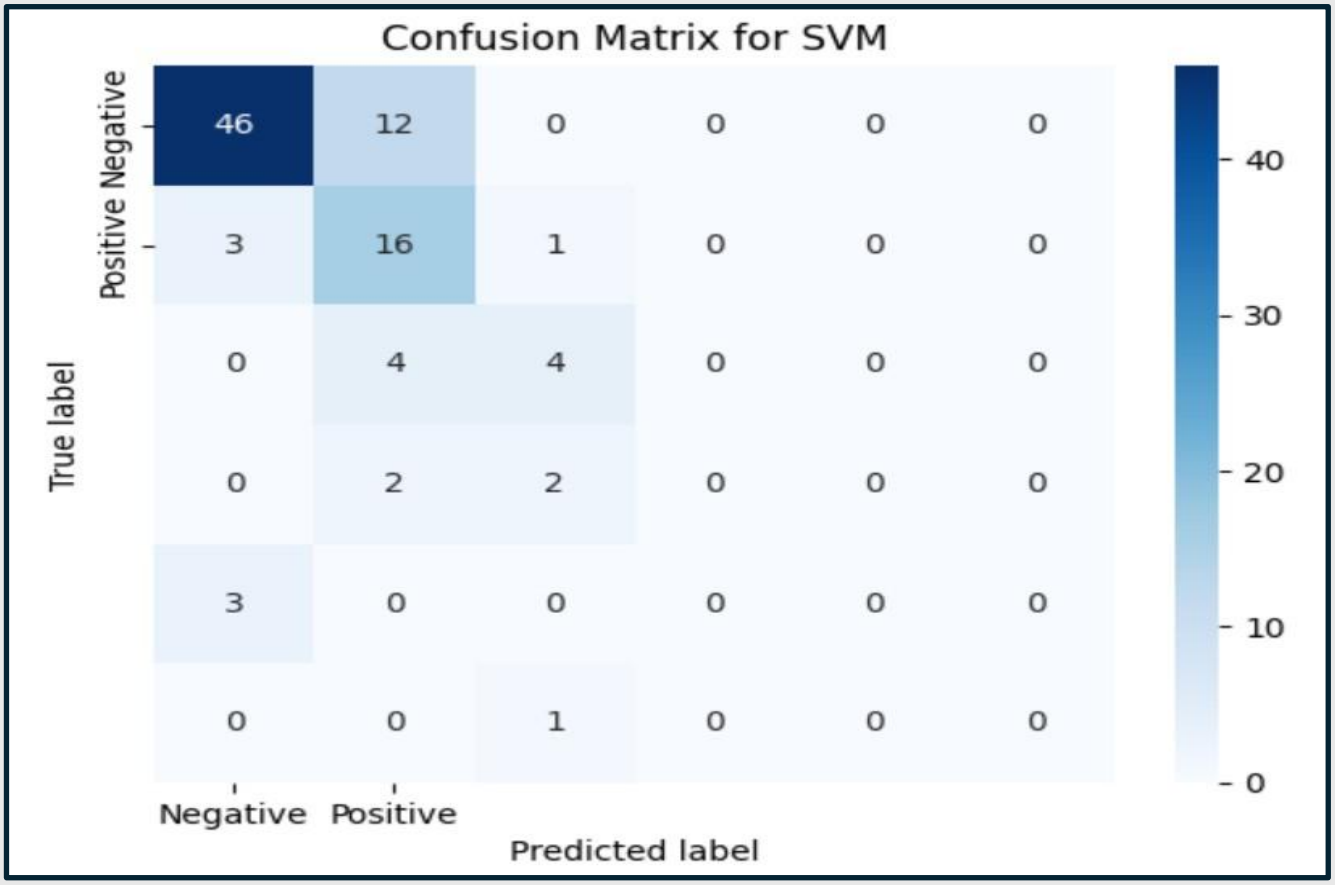


```
# Initialize the model
svm_model = SVC(random_state=42)

# Train the model
svm_model.fit(X_train, y_train)

# Predict the target variable
y_pred_svm = svm_model.predict(X_test)

# Print the predicted values for Support Vector Machine (SVM)
print("Predictions from Support Vector Machine (SVM):")
print(y_pred_svm)

# Confusion Matrix
cm_svm = confusion_matrix(y_test, y_pred_svm)

# Metrics
accuracy_svm = accuracy_score(y_test, y_pred_svm) *100
recall_svm = recall_score(y_test, y_pred_svm,average='weighted') * 100
precision_svm = precision_score(y_test, y_pred_svm,average='weighted')
* 100
f1_svm = f1_score(y_test, y_pred_svm,average='weighted') * 100

# Create a DataFrame to display the metrics
metrics_data = {
    "Metric": ["Accuracy", "Recall", "Precision", "F1 Score"],
    "Support vector machine(SVM)": [accuracy_svm, recall_svm,
precision_svm, f1_svm]}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(metrics_data)
# Display the results in DataFrame
display(metrics_df)
```

Sklearn SVM


SVM classifier (Training set)
Estimated Salary
Age
0
1
3
2
1
0
−1
−2
−2
−1
0
1
2
3


SVM classifier (Test set)
Estimated Salary
Age
0
1
3
2
1
0
−1
−2
−2
−1
0
1
2
3

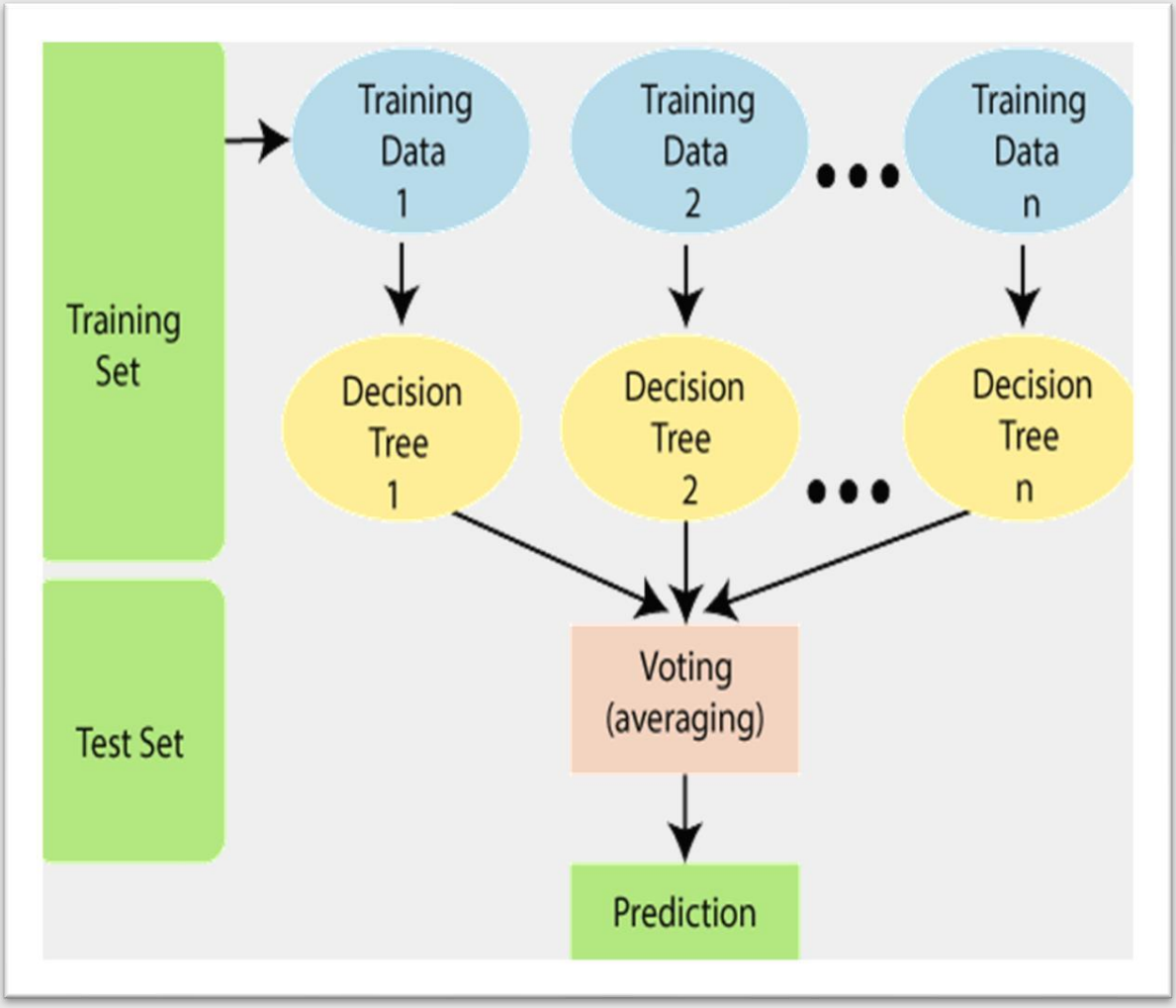


Classification Report as DataFrame:

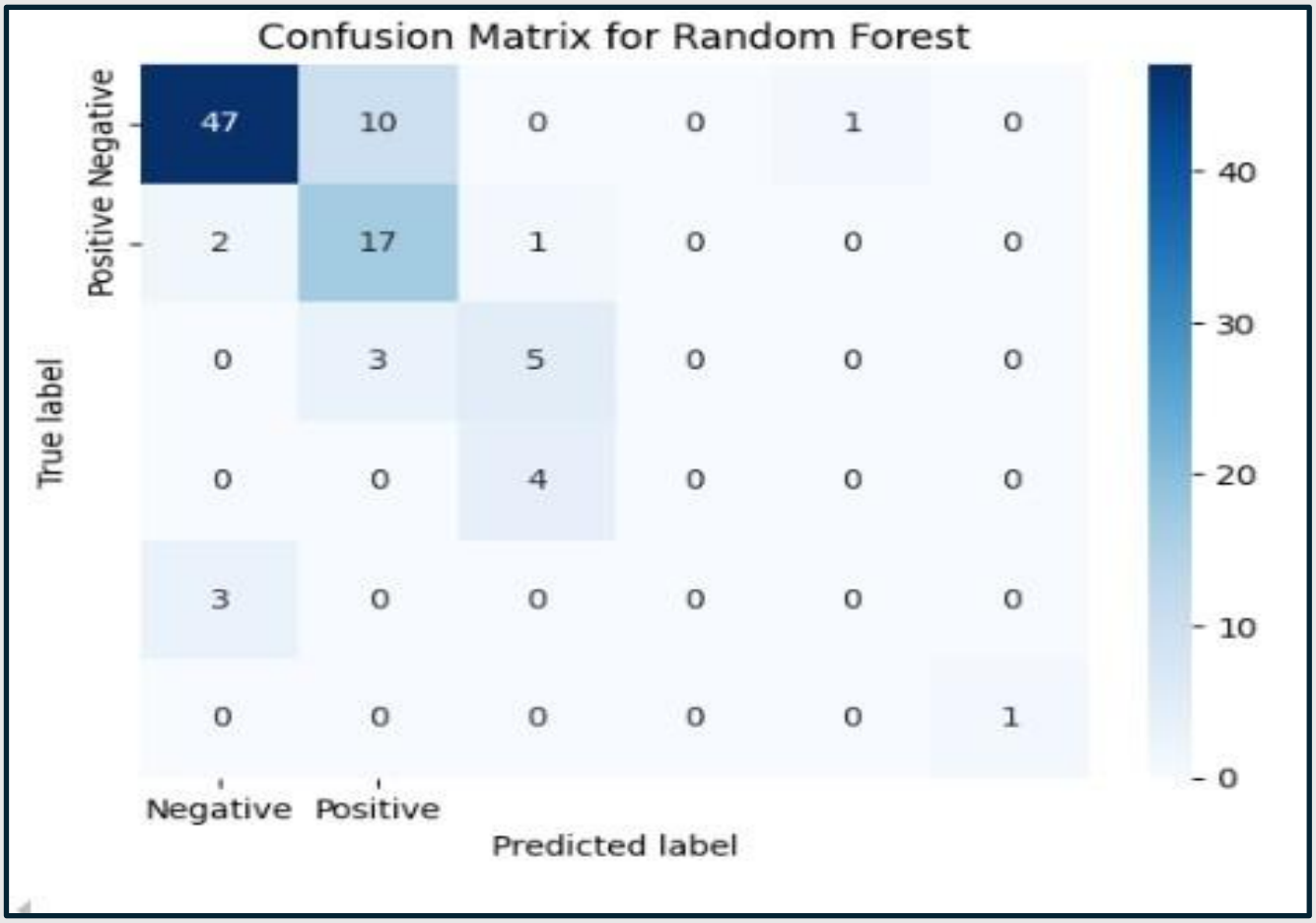
	precision	recall	f1-score	support
100k-200k	0.884615	0.793103	0.836364	58.000000
200k-300k	0.470588	0.800000	0.592593	20.000000
300k-400k	0.500000	0.500000	0.500000	8.000000
400k-500k	0.000000	0.000000	0.000000	4.000000
<100k	0.000000	0.000000	0.000000	3.000000
>500k	0.000000	0.000000	0.000000	1.000000
accuracy	0.702128	0.702128	0.702128	0.702128
macro avg	0.309201	0.348851	0.321493	94.000000
weighted avg	0.688505	0.702128	0.684691	94.000000

4 Random Forest Classifier

- **Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset.**
- **Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, it predicts the final output.**
- **The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.**



```
# Initialize the model
rf_model = RandomForestClassifier(random_state=42)

# Train the model
rf_model.fit(X_train, y_train)

# Predict the target variable
y_pred_rf = rf_model.predict(X_test)

# Print the predicted values for Random Forest Classifier
print("Predictions from Random Forest Classifier:")
print(y_pred_rf)

# Confusion Matrix
cm_rf = confusion_matrix(y_test, y_pred_rf)

# Metrics
accuracy_rf = accuracy_score(y_test, y_pred_rf) * 100
recall_rf = recall_score(y_test, y_pred_rf,average='weighted') * 100
precision_rf = precision_score(y_test, y_pred_rf,average='weighted') * 100
f1_rf = f1_score(y_test, y_pred_rf,average='weighted') * 100

# Create a DataFrame to display the metrics
metrics_data = {
    "Metric": ["Accuracy", "Recall", "Precision", "F1 Score"],
    "Random Forest Classifier": [accuracy_rf, recall_rf, precision_rf,
f1_rf]}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(metrics_data)
# Display the results in DataFrame
display(metrics_df)

# Confusion Matrix visualization
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues',
xticklabels=["Negative", "Positive"], yticklabels=["Negative",
"Positive"])
plt.title("Confusion Matrix for Random Forest")
```




```
Classification Report as DataFrame:
```

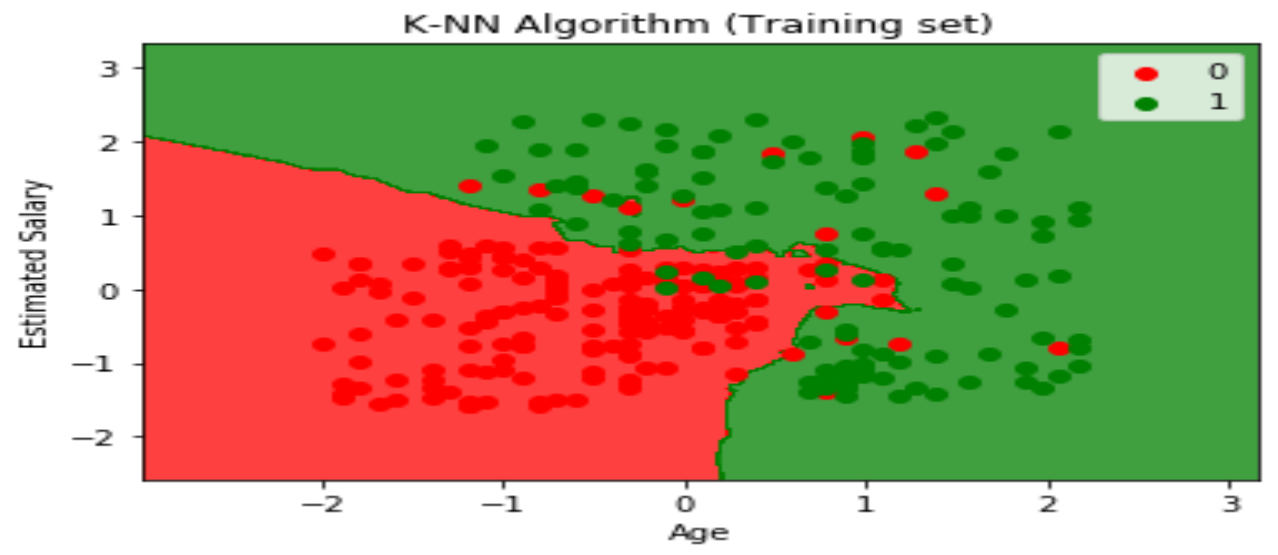
	precision	recall	f1-score	support
100k-200k	0.903846	0.810345	0.854545	58.000000
200k-300k	0.566667	0.850000	0.680000	20.000000
300k-400k	0.500000	0.625000	0.555556	8.000000
400k-500k	0.000000	0.000000	0.000000	4.000000
<100k	0.000000	0.000000	0.000000	3.000000
>500k	1.000000	1.000000	1.000000	1.000000
accuracy	0.744681	0.744681	0.744681	0.744681
macro avg	0.495085	0.547557	0.515017	94.000000
weighted avg	0.731451	0.744681	0.729873	94.000000

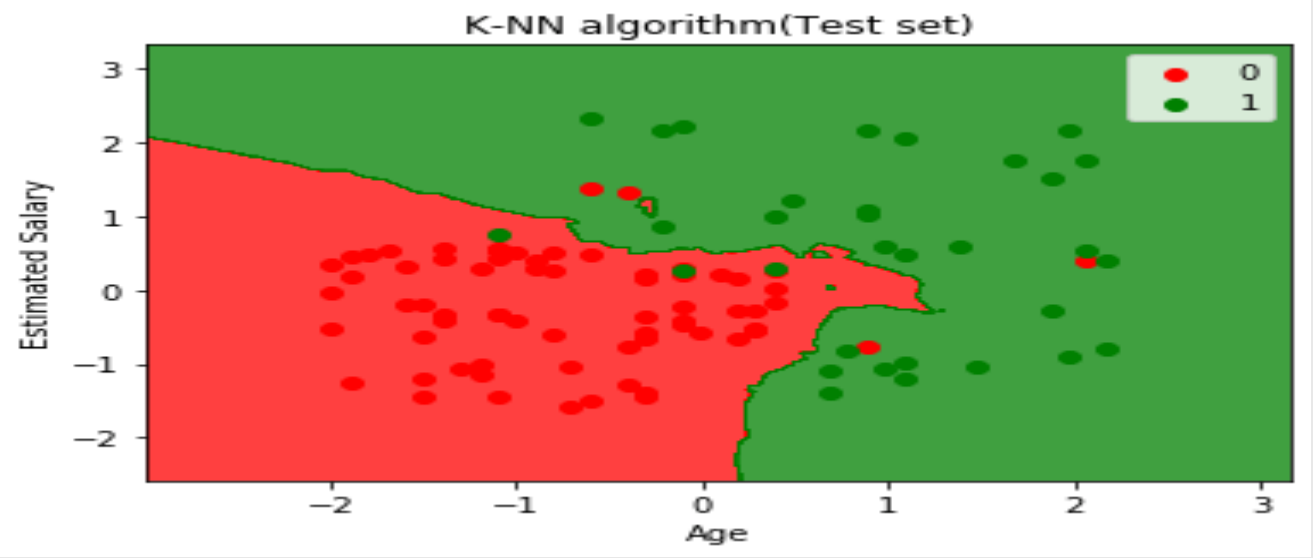
5 Knn

- **K-Nearest Neighbour** is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- **K-NN** algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- **K-NN algorithm** stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K-NN algorithm.
- Suppose there are two classes, i.e., Class A and Class B, and we need to classify a new data point to one of these classes. To solve this type of problem, we need a K-NN algorithm. So that, we can easily identify the category or class of a particular dataset.

```
# Initialize the model
knn_model = KNeighborsClassifier()

# Train the model
knn_model.fit(X_train, y_train)

# Predict the target variable
y_pred_knn = knn_model.predict(X_test)

# Print the predicted values for K-Nearest Neighbors (KNN)
print("Predictions from K-Nearest Neighbors (KNN):")
print(y_pred_knn)

# Confusion Matrix
cm_knn = confusion_matrix(y_test, y_pred_knn)

# Metrics
accuracy_knn = accuracy_score(y_test, y_pred_knn) * 100
recall_knn = recall_score(y_test, y_pred_knn,average='weighted') * 100
```

```
precision_knn = precision_score(y_test, y_pred_knn,average='weighted')
* 100
f1_knn = f1_score(y_test, y_pred_knn,average='weighted') * 100

# Create a DataFrame to display the metrics
metrics_data = {
    "Metric": ["Accuracy", "Recall", "Precision", "F1 Score"],
    "K-Nearest Neighbors (KNN)": [accuracy_knn, recall_knn,
precision_knn, f1_knn]}

# Convert the dictionary to a DataFrame
metrics_df = pd.DataFrame(metrics_data)
# Display the results in DataFrame
display(metrics_df)

# Confusion Matrix visualization
sns.heatmap(cm_knn, annot=True, fmt='d', cmap='Blues',
xticklabels=["Negative", "Positive"], yticklabels=["Negative",
"Positive"])
plt.title("Confusion Matrix for KNN")
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()
```





Confusion Matrix of all Models

	Metric	Decision Tree	Naive Bayes	Support vector machine(SVM)	Random Forest Classifier	K-Nearest Neighbors (KNN)
0	Accuracy	65.957447	42.553191	70.212766	74.468085	70.212766
1	Recall	56.633142	46.839080	70.212766	74.468085	70.212766
2	Precision	43.249701	34.955815	68.850486	73.145117	72.244521
3	F1 Score	47.367642	31.155026	68.469088	72.987320	69.961521

```
from tkinter import *
from tkinter import ttk, messagebox
from datetime import datetime
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
import pandas as pd
import numpy as np

# Importing X_df from the extracted Data Mining Project file
X_df = pd.read_csv("X_df.csv")
house_price=pd.read_csv("house_price.csv")
class Am_ML:
    def __init__(self, root):
        self.root = root
        self.root.geometry("1350x700+0+0")
        self.root.title("House Price Prediction")
        self.root.configure(bg="Light Yellow")
        now = datetime.now()
        current_date = now.strftime("%Y-%m-%d")
        current_time = now.strftime("%H:%M:%S")

        # Title Label
        title = Label(self.root, text="House Price Prediction",
compound=CENTER,
                      font=("times new roman", 40, "bold"), bg="#010c48",
fg="white", anchor="w", padx=20)
        title.place(x=0, y=0, relwidth=1, height=70)

        image_frame = LabelFrame(self.root, text="Some pictures of
houses", font=("times new roman", 16, "bold"), bg="Yellow",
fg="black")
        image_frame.place(x=875, y=120, width=400, height=550)
```

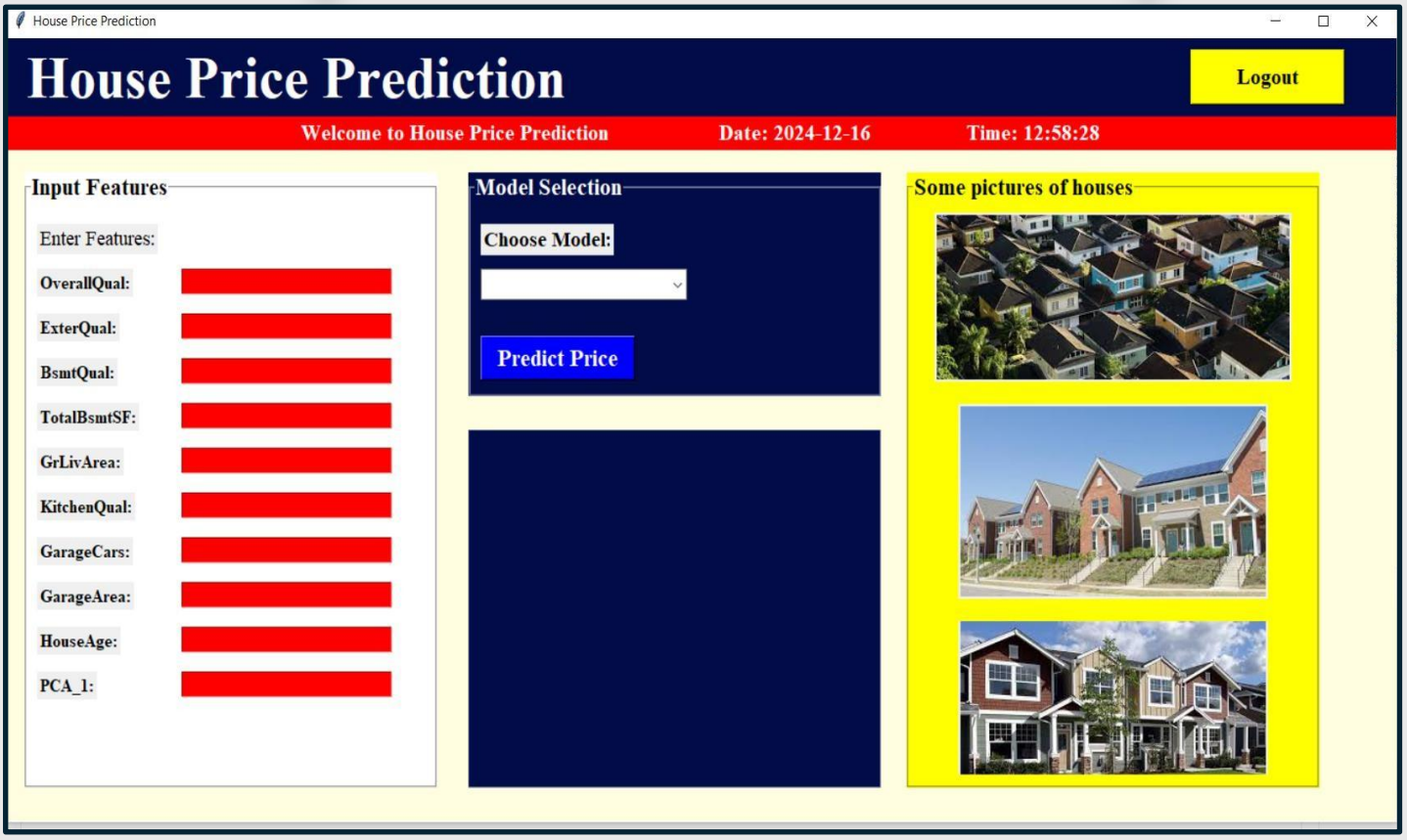
```
        self.img_1 = PhotoImage(file="C:/Users/pc/OneDrive - Benha
University (Faculty Of engineering)/Desktop/Data mining
project/download (2).png")
        label_img1 = Label(image_frame, image=self.img_1)
        label_img1.pack(pady=10)

        self.img_2 = PhotoImage(file="C:/Users/pc/OneDrive - Benha
University (Faculty Of engineering)/Desktop/Data mining
project/Image_2.png")
        label_img2 = Label(image_frame, image=self.img_2)
        label_img2.pack(pady=10)

        self.img_3 = PhotoImage(file="C:/Users/pc/OneDrive - Benha
University (Faculty Of engineering)/Desktop/Data mining
project/Image_3.png")
        label_img3 = Label(image_frame, image=self.img_3)
        label_img3.pack(pady=10)


        # Logout Button
        btn_logout = Button(self.root, text="Logout", font=("times new
roman", 15, "bold"), bg="yellow", cursor="hand2",
command=self.logout)
        btn_logout.place(x=1150, y=10, height=50, width=150)

        # Clock Label
        self.lbl_clock = Label(self.root, text=f"Welcome to House Price
Prediction \t\t Date: {current_date} \t\t Time: {current_time}",
                               font=("times new roman", 15,"bold"), bg="red",
fg="white")
        self.lbl_clock.place(x=0, y=70, relwidth=1, height=30)

        # Frame for User Inputs
        frame1 = LabelFrame(self.root, text="Input Features", font=("times
new roman", 16,"bold"), bg="White", fg="black")
        frame1.place(x=20, y=120, width=400, height=550)
```

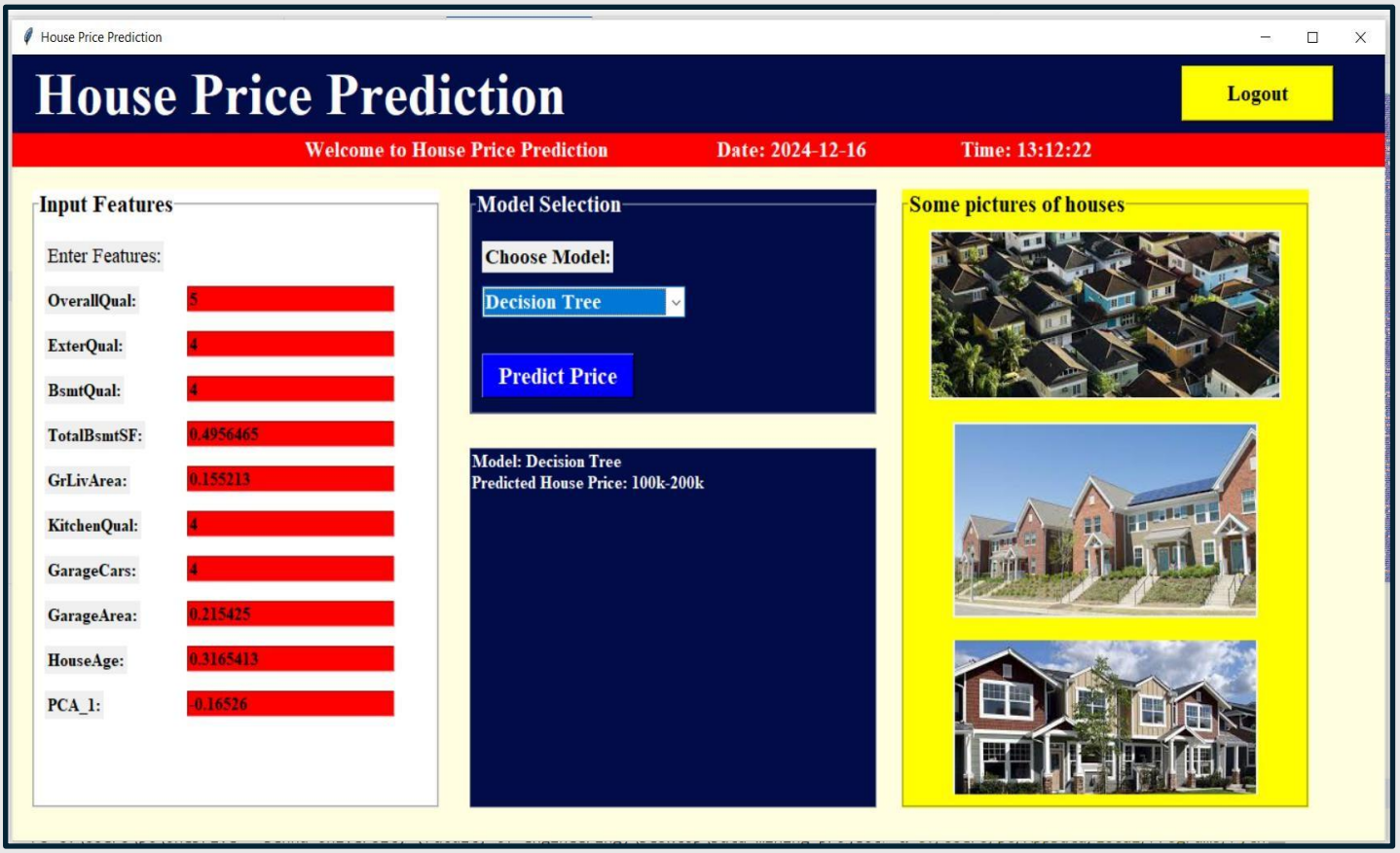
```
        Label(frame1, text="Enter Features:", font=("times new roman",
14)).place(x=10, y=20)
        self.feature_entries = {}

        # Create Dynamic Input Fields from X_df
        self.features = X_df.columns.tolist() # Use columns of X_df as
features

        y_offset = 60
        for feature in self.features:
            Label(frame1, text=f"{feature}:", font=("times new roman",
12,"bold")).place(x=10, y=y_offset)
            entry = Entry(frame1, font=("times new roman",
12,"bold"),bg="red", width=25)
            entry.place(x=150, y=y_offset)
            self.feature_entries[feature] = entry
            y_offset += 40

        # Frame for Model Selection
        frame2 = LabelFrame(self.root, text="Model Selection", font=("times
new roman", 16 , "bold"), bg="#010c48", fg="White")
        frame2.place(x=450, y=120, width=400, height=200)

        Label(frame2, text="Choose Model:", font=("times new roman",
14,"bold")).place(x=10, y=20)
        self.model_var = StringVar()
        self.model_combobox = ttk.Combobox(frame2,
textvariable=self.model_var, font=("times new roman", 14,"bold"),
state="readonly")
        self.model_combobox['values'] = ("Decision Tree", "Naive Bayes",
"SVM", "Random Forest", "KNN")
        self.model_combobox.place(x=10, y=60, width=200)

        Button(frame2, text="Predict Price", font=("times new roman", 16,
"bold"), bg="blue", fg="white", cursor="hand2",
command=self.predict_price).place(x=10, y=120, width=150, height=40)
```

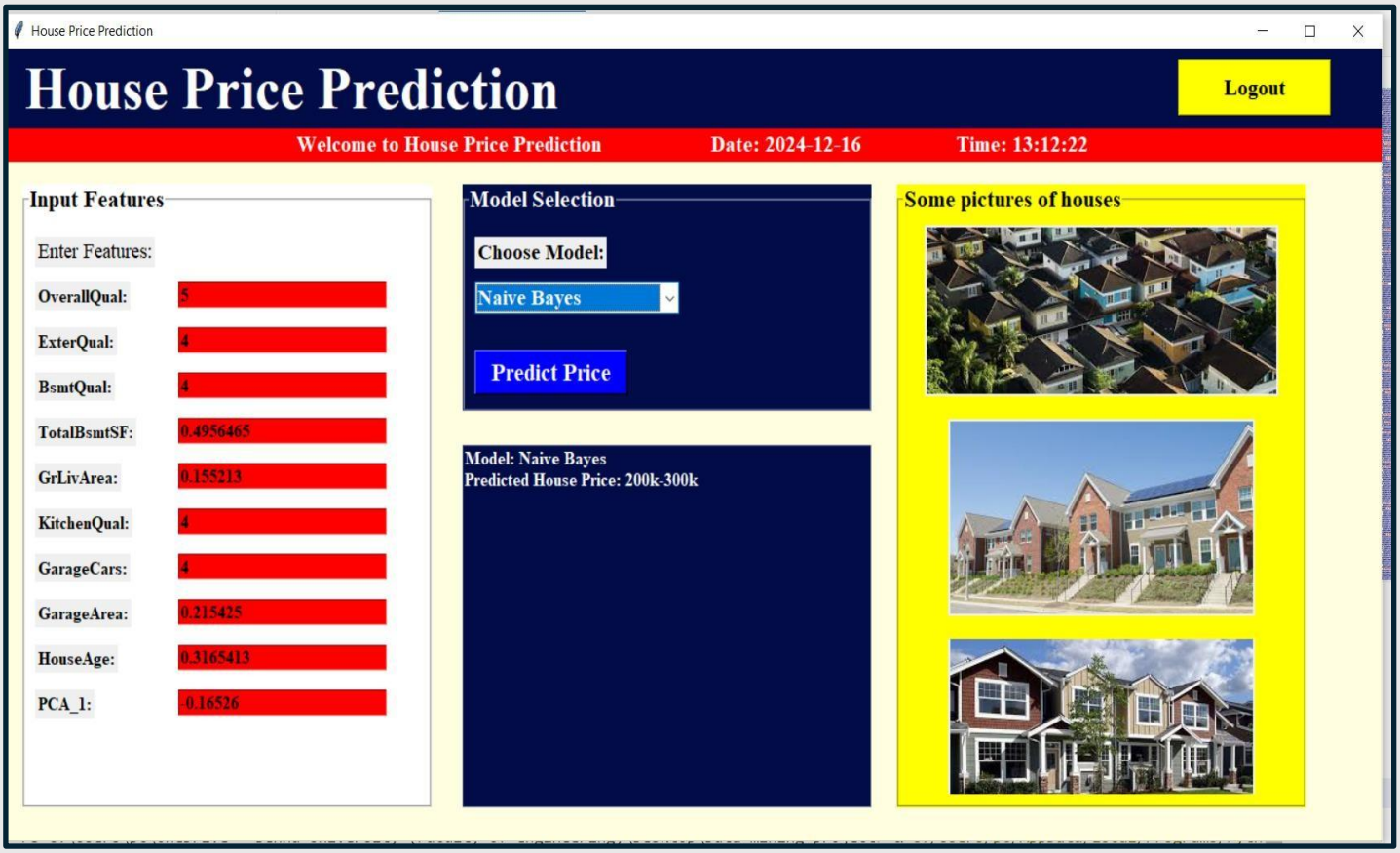
```
    def predict_price(self):
        try:
            # Collect user input for features
            input_data = {}
            for feature, entry in self.feature_entries.items():
                value = entry.get()
                if value == "":
                    messagebox.showerror("Error", f"Please enter a value for
{feature}.")
                    return
                try:
                    # Convert input to float and store in input_data dictionary
                    input_data[feature] = float(value)
                except ValueError:
                    messagebox.showerror("Error", f"Please enter a valid
numeric value for {feature}.")
                    return

            # Prepare data from X_df (already preprocessed in your notebook)
            X = X_df

            # Placeholder for target (y) - Assuming you have target column
'price' in your notebook processing
            y = house_price

            # Reshape y to be a 1D array for scikit-learn
            y = y.values.ravel()  # Correcting the target variable

            # Split data into training and testing sets
            X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

            # Initialize model based on user choice
            model_name = self.model_var.get()
            if model_name == "Decision Tree":
                model = DecisionTreeClassifier()
            elif model_name == "Naive Bayes":
                model = GaussianNB()
```

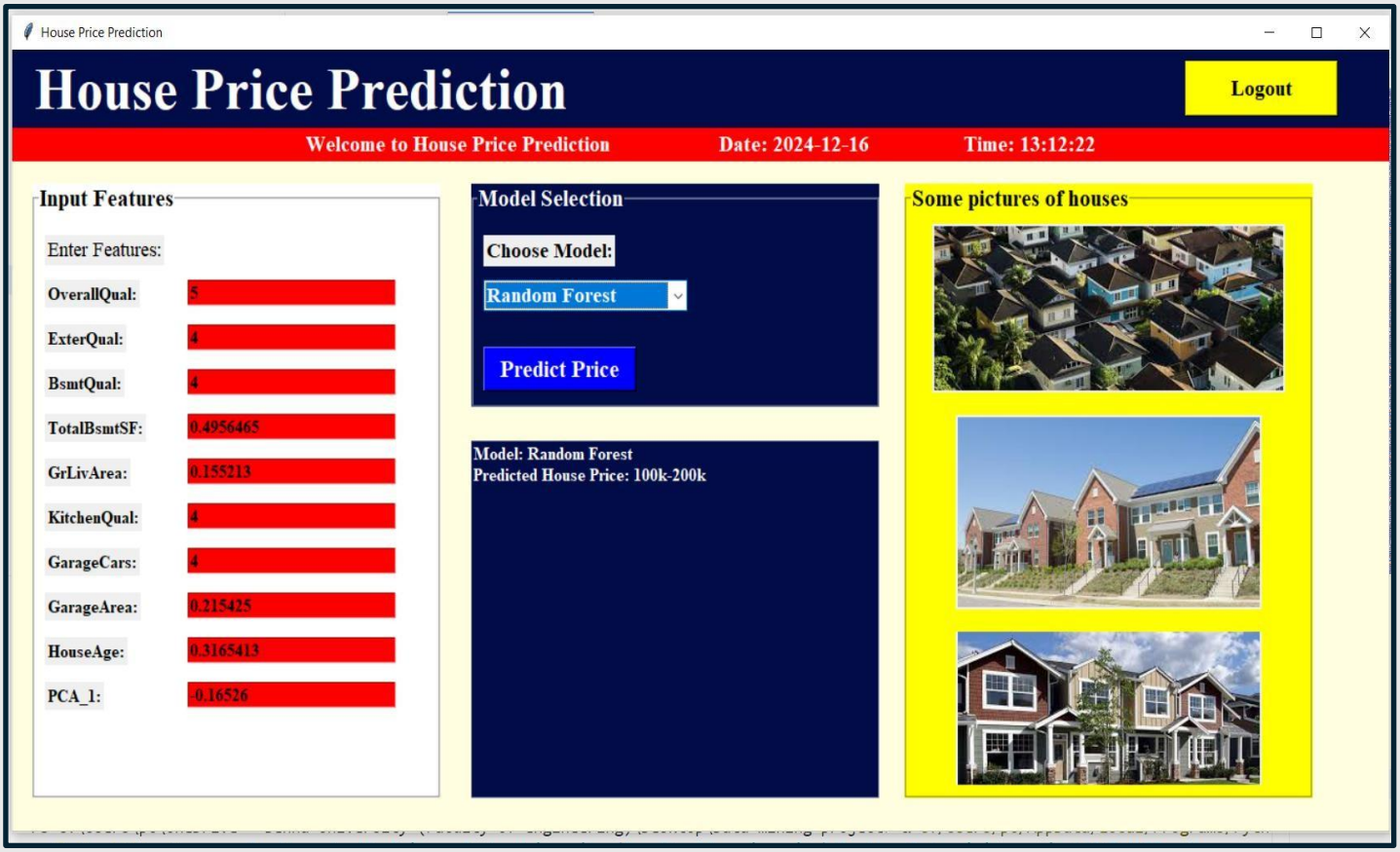
```
elif model_name == "SVM":
            model = SVC()
        elif model_name == "Random Forest":
            model = RandomForestClassifier()
        elif model_name == "KNN":
            model = KNeighborsClassifier()
        else:
            messagebox.showerror("Error", "Please select a model.")
            return

        # Train the model
        model.fit(X_train, y_train)

        # Predict house price based on user input
        input_df = pd.DataFrame([input_data])
        prediction = model.predict(input_df)[0]

        # Display output
        self.output_text.delete("1.0", END)
        self.output_text.insert(END, f"Model: {model_name}\n")
        self.output_text.insert(END, f"Predicted House Price:
{prediction}\n")

    except Exception as e:
        messagebox.showerror("Error", f"An error occurred: {str(e)}")

if __name__ == "__main__":
    root = Tk()
    obj = Am_ML(root)
    root.mainloop()
```

Screenshots






House Price Prediction
House Price Prediction
Logout
Welcome to House Price Prediction
Date: 2024-12-16
Time: 13:12:22
Input Features
Enter Features:
OverallQual: 5
ExterQual: 4
BsmtQual: 4
TotalBsmtSF: 0.4956465
GrLivArea: 0.155213
KitchenQual: 4
GarageCars: 4
GarageArea: 0.215425
HouseAge: 0.3165413
PCA_1: -0.16526
Model Selection
Choose Model:
Naive Bayes
Predict Price
Model: Naive Bayes
Predicted House Price: 200k-300k
Some pictures of houses


House Price Prediction
House Price Prediction
Logout
Welcome to House Price Prediction
Date: 2024-12-16
Time: 13:12:22
Input Features
Enter Features:
OverallQual: 5
ExterQual: 4
BsmtQual: 4
TotalBsmtSF: 0.4956465
GrLivArea: 0.155213
KitchenQual: 4
GarageCars: 4
GarageArea: 0.215425
HouseAge: 0.3165413
PCA_1: -0.16526
Model Selection
Choose Model:
Random Forest
Predict Price
Model: Random Forest
Predicted House Price: 100k-200k
Some pictures of houses


Thank you