In [1]:
```
## importing the required libraries


import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.impute import SimpleImputer
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
```

In [2]:
```
## Load the data
train_data = pd.read_csv('train.csv')

## Drop rows with missing target values
train_data.dropna(axis=0, subset=['SalePrice'], inplace=True)

## Define target (y) and features (X)
y = train_data['SalePrice']
X = train_data.drop(['SalePrice'], axis=1).select_dtypes(exclude=['object'])

## Now, ensure y is aligned with X after dropping missing values
y = y[X.index]
```

In [3]:
```
## Split the data into training and testing sets
## 25% of the data will be used for testing, the rest for training
## Feature (x) and target (y) splits
train_X, test_X, train_y, test_y = train_test_split(X.values, y.values, test_size=0.25, random_state=42)

## Initialise the imputer (you can choose strategy: 'mean', 'median', or 'most_frequent')
my_imputer = SimpleImputer()  # Default is 'mean', but can choose others

## Fit and transform the training data (impute missing values in the training set)
train_X = my_imputer.fit_transform(train_X)

## Transform the test data using the same imputer (to avoid data leakage)
test_X = my_imputer.transform(test_X)
```

```
## Optionally, you can convert back to DataFrame with the original column names after imputation
train_X = pd.DataFrame(train_X, columns=X.columns)
test_X = pd.DataFrame(test_X, columns=X.columns)
```

In [4]:
```
## Predictions using the Decision Tree algorithm

decision_model = DecisionTreeRegressor()
decision_model.fit(train_X, train_y)
predicted_decision_trees = decision_model.predict(test_X)
print ("Mean Absolute Error using Decision Tress :", mean_absolute_error(test_y, predicted_decision_trees))
```

```
Mean Absolute Error using Decision Tress : 25224.378082191783
```

In [5]:
```
## Predictions using the Random Forest algorithm

forest_model = RandomForestRegressor(n_estimators=100, max_depth=10)
forest_model.fit(train_X, train_y )
predicted_random_forest = forest_model.predict(test_X)
print("Mean Absolute Error using Random Forest:", mean_absolute_error(test_y, predicted_random_forest))
```

```
Mean Absolute Error using Random Forest: 17675.946473336273
```

In [6]:
```
## Predictions using the XGBoost algorithm

xg_model = XGBRegressor(n_estimators=100)
xg_model.fit(train_X, train_y)
predicted_XGBoost = xg_model.predict(test_X)
print("Mean Absolute Error using XGBoost: ", mean_absolute_error(test_y, predicted_XGBoost))
```

```
Mean Absolute Error using XGBoost:  18624.489811643834
```

In [7]:
```
## Let's improve the RandomForest Model as it was the best performing in MAE
## We will use GridSearchCV to tune the hyperparameters of the RandomForest Model

from sklearn.model_selection import GridSearchCV

## Define the hyperparameter grid to search
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2']
}


## Create a RandomForestRegressor model
rf_model = RandomForestRegressor()

## Perform grid search with cross-validation
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=2, n_jobs=-1, verbose=2)
grid_search.fit(train_X, train_y)

## Get the best parameters and best score
print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)

## Predict with the best model
best_rf_model = grid_search.best_estimator_
predicted_random_forest = best_rf_model.predict(test_X)
print("Mean Absolute Error using optimised Random Forest:", mean_absolute_error(test_y, predicted_random_forest))
```

```
Fitting 2 folds for each of 216 candidates, totalling 432 fits
c:\Users\samuel.mcdonnell\AppData\Local\anaconda3\envs\HousePrices\Lib\site-packages\numpy\ma\core.py:2820: RuntimeWarning: inv
alid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
Best parameters: {'max_depth': 30, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best score: 0.8349896844105849
Mean Absolute Error using optimised Random Forest: 16951.805671232876
```

In [8]:

```
## Get the feature importance scores
feature_importances = forest_model.feature_importances_

## Create a list of features with their corresponding importance scores
feature_importance_dict = {
    feature: importance for feature, importance in zip(X.columns, feature_importances)
}

## Sort the features by their importance score (in descending order)
sorted_features = sorted(feature_importance_dict.items(), key=lambda x: x[1], reverse=True)

## Display the top 10 most important features
top_10_features = sorted_features[:10]
```

```
print("Top 10 Features by Importance:")
for feature, importance in top_10_features:
    print(f"{feature}: {importance}")

## Random Forest inherently handles feature importance so it isn't necessary to include in tuning process. Still interesting t
```

```
Top 10 Features by Importance:
OverallQual: 0.5666270084021948
GrLivArea: 0.1251935508678321
TotalBsmtSF: 0.03610935941638093
BsmtFinSF1: 0.03176757360055653
2ndFlrSF: 0.03134538136165631
1stFlrSF: 0.026935964347982625
GarageCars: 0.024985106483516453
YearBuilt: 0.01790109932950991
GarageArea: 0.017185734383097943
LotArea: 0.016826869446280538
```