Homework - Hyperparameter Tuning

DAT-5303 | Machine Learning

Team: 10

Step 1: Imports
Import the following packages and make sure to start replacing my comments with your own. Also, please tell me about which models you are tuning in the second cell.

In [3]:

```
# importing the libraries
import pandas as pd
import numpy as np
#  ML models
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
# train-test split
from sklearn.model_selection import train_test_split
# for hyperparameter tuning
from sklearn.model_selection import RandomizedSearchCV
```

In [6]:

```
# Reading the dataset
df = pd.read_excel('housing_feature_rich.xlsx')
df.head()
```

Out[6]:

	Order	Lot_Area	Overall_Qual	Overall_Cond	Mas_Vnr_Area	Total_Bsmt_SF	First_Flr_SF	Second_Flr_SF	Gr_Liv_Area	Full_Bath	...	NoRidge	NridgH
0	1	31770	6	5	112	1080	1656	0	1656	1	...	0	
1	2	11622	5	6	0	882	896	0	896	1	...	0	
2	3	14267	6	6	108	1329	1329	0	1329	1	...	0	
3	4	11160	7	5	0	2110	2110	0	2110	2	...	0	
4	5	13830	5	5	0	928	928	701	1629	2	...	0	

5 rows × 68 columns

In [8]:

```
# preparing x-variables
x_data = df.drop(['Sale_Price', 'log_Sale_Price'], axis = 1)
# preparing y-variable
y_data = df.loc[ : , 'Sale_Price']    # y-variable
```

We are tuning DecisionTreeRegressor and RandomForestRegressor

Step 2: Train-Test Split
Set up train-test split in the cell below. Set your *test_size* to 0.25 and your *random_state* to 219.

In [14]:

```
# train-test split with stratification
x_train, x_test, y_train, y_test = train_test_split(
            x_data,
            y_data,
            test_size    = 0.25,  # test size
            random_state = 219 )    # Random size
```

In [ ]:

```
```

Step 3: Check Available Hyperparameters

Call help( ) on your selected model types to see which hyperparameters are available for tuning. Do NOT tune random_state, n_jobs or anything related to the intercept of a model. Use as many cells as needed.

In [15]:

```
help(DecisionTreeRegressor)
```

```
|  random_state : int, RandomState instance or None, default=None
|      Controls the randomness of the estimator. The features are always
|      randomly permuted at each split, even if ``splitter`` is set to
|      ``"best"``. When ``max_features < n_features``, the algorithm will
|      select ``max_features`` at random at each split before finding the best
|      split among them. But the best found split may vary across different
|      runs, even if ``max_features=n_features``. That is the case, if the
|      improvement of the criterion is identical for several splits and one
|      split has to be selected at random. To obtain a deterministic behaviour
|      during fitting, ``random_state`` has to be fixed to an integer.
|      See :term:`Glossary <random_state>` for details.
|
|  max_leaf_nodes : int, default=None
|      Grow a tree with ``max_leaf_nodes`` in best-first fashion.
|      Best nodes are defined as relative reduction in impurity.
|      If None then unlimited number of leaf nodes.
|
|  min_impurity_decrease : float, default=0.0
|      A node will be split if this split induces a decrease of the impurity
|      greater than or equal to this value.
|
```

In [16]:

```
help(RandomForestRegressor)
```

```
|      .. versionchanged:: 0.18
|         Added float values for fractions.
|
|  min_samples_leaf : int or float, default=1
|      The minimum number of samples required to be at a leaf node.
|      A split point at any depth will only be considered if it leaves at
|      least ``min_samples_leaf`` training samples in each of the left and
|      right branches.  This may have the effect of smoothing the model,
|      especially in regression.
|
|      - If int, then consider `min_samples_leaf` as the minimum number.
|      - If float, then `min_samples_leaf` is a fraction and
|        `ceil(min_samples_leaf * n_samples)` are the minimum
|        number of samples for each node.
|
|      .. versionchanged:: 0.18
|         Added float values for fractions.
|
|  min_weight_fraction_leaf : float, default=0.0
|      The minimum weighted fraction of the sum total of weights (of all
```

Step 4: Model Development - Default Hyperparameters

Run each of your selected models using three (3) respective default hyperparameters that you feel would be good candidates for optimization, printing their R-Square values for the training and testing sets. Use as many cells as needed and make sure to label your default hyperparameters.

Model_1 (Decision Tree Regressor)

In [29]:

```
# decision tree regressor
model_1  = DecisionTreeRegressor(max_depth=None,
                                 min_samples_split=2,
                                 min_samples_leaf=2)
# fit the train set
model_1.fit(x_train, y_train)
# predict on test set
y_pred = model_1.predict(x_test)
```

In [30]:

```
from sklearn.metrics import r2_score
```

In [31]:

```
# Get the train score
train_score = r2_score(y_train, model_1.predict(x_train))
print("R-squared score on training set:{}".format(train_score))
```

```
R-squared score on training set:0.9888964688340652
```

In [33]:

```
# Get the test score
test_score = r2_score(y_test, y_pred)
print("R-squared score on test set:{}".format(test_score))
```

```
R-squared score on test set:0.7779934527369415
```

In [ ]:

```

```

Model_2 (RandomForestRegressor)

In [41]:

```
# random forest regressor
model_2 = RandomForestRegressor(n_estimators=100,
                                max_depth=None,
                                min_samples_split=2)
# fit the train set
model_2.fit(x_train, y_train)
# predict on test set
y_pred2 = model_2.predict(x_test)
```

In [42]:

```
# Get the train score
train_score = r2_score(y_train, model_2.predict(x_train))
print("R-squared score on training set:{}".format(train_score))
```

```
R-squared score on training set:0.9827194619927093
```

In [43]:

```
# Get the test score
test_score = r2_score(y_test, y_pred2)
print("R-squared score on test set:{}".format(test_score))
```

```
R-squared score on test set:0.8785104994567583
```

In [ ]:

```

```

Step 5: Hyperparameter Tuning Part I

Develop ranges for each of the hyperparameters you would like tune. Write a code to calculate how many models you are building (based on the combinations of hyperparameters you are tuning).

Requirement: You must tune at least three hyperparameters per model.
Recommendation: Keep the number of models (i.e., iterations) you are building to:

- Less than 2,000 for Random Forest and GBM (not including cross-validation)
- Less than 10,000 for all other model types (not including cross-validation)

Insert as many code cells as you need for the number of model types you are tuning.

Model_1 (Decision Tree Regressor)

In [45]:

```
# hyperparameter ranges
max_depth_range = np.arange(1,6)
min_samples_split_range = np.arange(2,11)
min_samples_leaf_range = np.arange(1,11)
max_leaf_nodes_range = np.arange(1,11)
```

In [47]:

```
# caluculating the number of possible combinations of ML Model
a1 = len(np.arange(1,6))
a2 = len(np.arange(2,11))
a3 = len(np.arange(1,11))
a4 = len(np.arange(1,11))
# total combinations
total_num = a1 * a2 * a3 * a4

print(" The combinatins of hyperparameters is:{}".format(total_num))
```

```
 The combinatins of hyperparameters is:4500
```

In [ ]:

```

```

Model_2 (RandomForestRegressor)¶

In [56]:

```
# hyperparameter ranges
n_estimators_range = np.arange(100, 600, 100)
max_depth_range = np.arange(6,11)
min_samples_split_range = np.arange(5,11)
min_samples_leaf_range = np.arange(5,11)
```

In [62]:

```
# caluculating the number of possible combinations of ML Model
b1 = len(np.arange(100, 600, 100))
b2 = len(np.arange(6,11))
b3 = len(np.arange(5,11))
b4 = len(np.arange(2,11))
# total combinations
total_num2 = b1 * b2 * b3 * b4

print(" The combinatins of hyperparameters is:{}".format(total_num2))
```

```
 The combinatins of hyperparameters is:1350
```

In [ ]:

```

```

Step 6: Hyperparameter Tuning Part II

Create a hyperparameter grid for each of the models you are tuning. Note that this is the dictionary step we conducted in class. Use as many cells as needed for this task.

In [ ]:

```

```

Model_1 (Decision Tree Regressor)

In [58]:

```
param_grid1 = {'max_depth' : np.arange(1,6),
               'min_samples_split': np.arange(2,11),
               'min_samples_leaf' : np.arange(1,11),
               'max_leaf_nodes' : np.arange(1,11)}
```

In [ ]:

```

```

In [ ]:

```

```

Model_2 (RandomForestRegressor)

In [63]:

```
param_grid2 = {'n_estimators' : np.arange(100,600,100),
               'max_depth': np.arange(6,11),
               'min_samples_split' : np.arange(5,11),
               'max_leaf_nodes' : np.arange(2,11)}
```

In [ ]:

```

```

Step 7: Hyperparameter Tuning Part III

a) Complete the remaining steps for hyperparameter tuning. While your code runs, observe the processing time it takes to tune. Use as many cells as needed for this task.

Model_1 (Decision Tree Regressor)

In [60]:

```
# instanciating the model
model_1 = DecisionTreeRegressor() #
# RandomizedSearchXCV
model_1_cv = RandomizedSearchCV(estimator           = model_1,
                                param_distributions = param_grid1,
                                n_iter              = 1000,
                                cv                  = 3,
                                random_state        = 219)
# fitting the model
model_1_cv.fit(x_data, y_data)

# printing the optimal parameters and best score
print(f"Tuned Parameters: {model_1_cv.best_params_}")
print(f"Tuned R-Square:   {model_1_cv.best_score_}")
```

```
        nan 0.46842401 0.46842401 0.69160329 0.46842401 0.61350647
 0.70380722 0.70380722 0.68244981 0.65393609 0.46842401 0.69744349
 0.46842401 0.65393609 0.61350647        nan 0.69377788 0.65393609
 0.67060541 0.46842401 0.61350647 0.61350647 0.46842401 0.46842401
 0.61350647 0.67060541 0.69132263 0.56043804 0.69744349 0.61350647
        nan 0.70379256 0.61350647 0.46842401 0.46842401 0.69349723
        nan 0.61350647 0.56043804 0.69362943        nan 0.69961809
        nan 0.61350647 0.46842401 0.46842401 0.68385801 0.61350647
 0.46842401 0.61350647 0.61350647 0.46842401        nan 0.46842401
 0.69377788 0.46842401 0.68385801 0.46842401 0.61350647 0.61350647
 0.61350647 0.61350647 0.46842401 0.46842401 0.46842401 0.69145483
 0.7162684  0.65393609 0.46842401 0.56043804 0.46842401 0.69744349
 0.67060541 0.56043804 0.46842401 0.61350647 0.65393609 0.61350647
 0.56043804 0.56043804 0.46842401 0.56043804 0.61350647 0.61350647
 0.69961809        nan 0.6914695  0.46842401 0.71638128 0.46842401
 0.46842401 0.46842401 0.69132263 0.46842401 0.70379256        nan
 0.46842401        nan 0.46842401 0.61350647 0.70379256 0.6914695
        nan 0.61350647        nan 0.68244981 0.61350647 0.46842401
        nan        nan 0.46842401 0.56043804 0.46842401 0.68385801
```

In [ ]:

```

```

Model_2 (RandomForestRegressor)

In [*]:

```
# instanciating the model
model_2 = RandomForestRegressor()
# RandomizedSearchXCV
model_2_cv = RandomizedSearchCV(estimator           = model_2,
                                param_distributions = param_grid2,
                                n_iter              = 1000,
                                cv                  = 3,
                                random_state        = 219)
# fitting the model
model_2_cv.fit(x_data, y_data)

# printing the optimal parameters and best score
print(f"Tuned Parameters:  {model_2_cv.best_params_}")
print(f"Tuned R-Square:    {model_2_cv.best_score_}")
```

In [ ]:

```

```

b) Explore the hyperparameter tuning results for one of your tuned models. Look for hyperparameter combinations that tied for first place. Test each of these models using train-test split. If no models tied for first place, test your top three models.

In [ ]:

```
def tuning_results(cv_results, n=5):
    """
This function will display the top "n" models from hyperparameter tuning,
based on "rank_test_score".

PARAMETERS
----------
cv_results = results dictionary from the attribute ".cv_results_"
n          = number of models to display
    """
    param_lst = []

    for result in cv_results["params"]:
        result = str(result).replace(":", "=")
        param_lst.append(result[1:-1])


    results_df = pd.DataFrame(data = {
        "Model_Rank" : cv_results["rank_test_score"],
        "Mean_Test_Score" : cv_results["mean_test_score"],
        "SD_Test_Score" : cv_results["std_test_score"],
        "Parameters" : param_lst
    })


    results_df = results_df.sort_values(by = "Model_Rank", axis = 0)
    return results_df.head(n = n)
```

In [ ]:

```
tuning_results(model_1_cv_results_, n=5)
```

In [ ]:

```

```