Group information

Family name	First name	Student ID	Email address	UTORID
Yin	Weilin	1005825047	alex.yin@mail.utoronto.ca	yinweili

Homework 2

Consider the `housing_price` dataset. The response measures the selling price of houses and the input contains numerous house attributes (see `housing_readme.md`) of different data types. The goal of this exercise is to predict the selling price of houses based on these attributes using tree-based method, including decicion trees, random forest, and gradient boosted trees.

In [55]:
```
# Modules
import inspect
import numpy as np
import pandas as pd

from sklearn import ensemble, metrics, model_selection, preprocessing, tree
from matplotlib import pyplot as plt
```

In [8]:
```
pd.options.display.max_rows = 500
pd.options.display.max_columns = 500
pd.options.display.width = 1000
```

1. Load the `housing_price` dataset available on Quercus (see `pd.read_csv`, mind the index column) and format the response as a `pd.Series`. Display descriptive statistics and compute the number of missing values for each variable.

In [2]:
```
housing_data = pd.read_csv("ensemble_data/housing_data.csv", index_col=0)
housing_target = pd.read_csv("ensemble_data/housing_target.csv", index_col=0, squeeze=True)
```

```
/var/folders/gg/jvlpw5wj401ckbfp3rh95lf80000gn/T/ipykernel_1175/3124336863.py:2: FutureWarn
ing: The squeeze argument has been deprecated and will be removed in a future version. Appe
nd .squeeze("columns") to the call to squeeze.


  housing_target = pd.read_csv("ensemble_data/housing_target.csv", index_col=0, squeeze=Tru
e)
```

In [9]:
```
# Descriptive Statistics
housing_data.describe()
```

Out[9]:

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	Ma
count	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000	145
mean	56.897260	70.049958	10516.828082	6.099315	5.575342	1971.267808	1984.865753	10
std	42.300571	24.284752	9981.264932	1.382997	1.112799	30.202904	20.645407	1
min	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.000000	1950.000000	
25%	20.000000	59.000000	7553.500000	5.000000	5.000000	1954.000000	1967.000000	
50%	50.000000	69.000000	9478.500000	6.000000	5.000000	1973.000000	1994.000000	
75%	70.000000	80.000000	11601.500000	7.000000	6.000000	2000.000000	2004.000000	16
max	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.000000	2010.000000	160

In [4]:
```
housing_target.describe()
```

Out[4]:
```
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

In [10]:
```
# number of missing values in housing_data
housing_data.isnull().sum()
```

Out[10]:

MSSubClass	0
MSZoning	0
LotFrontage	259
LotArea	0
Street	0
Alley	1369
LotShape	0
LandContour	0
Utilities	0
LotConfig	0
LandSlope	0
Neighborhood	0
Condition1	0
Condition2	0
BldgType	0
HouseStyle	0
OverallQual	0
OverallCond	0
YearBuilt	0
YearRemodAdd	0
RoofStyle	0
RoofMatl	0
Exterior1st	0
Exterior2nd	0
MasVnrType	8
MasVnrArea	8
ExterQual	0
ExterCond	0
Foundation	0
BsmtQual	37
BsmtCond	37
BsmtExposure	38
BsmtFinType1	37
BsmtFinSF1	0
BsmtFinType2	38
BsmtFinSF2	0
BsmtUnfSF	0
TotalBsmtSF	0
Heating	0
HeatingQC	0
CentralAir	0
Electrical	1
1stFlrSF	0
2ndFlrSF	0
LowQualFinSF	0
GrLivArea	0
BsmtFullBath	0
BsmtHalfBath	0
FullBath	0
HalfBath	0
BedroomAbvGr	0
KitchenAbvGr	0
KitchenQual	0
TotRmsAbvGrd	0
Functional	0
Fireplaces	0
FireplaceQu	690
GarageType	81
GarageYrBlt	81
GarageFinish	81
GarageCars	0
GarageArea	0
GarageQual	81

```
GarageCond         81
PavedDrive          0
WoodDeckSF          0
OpenPorchSF         0
EnclosedPorch       0
3SsnPorch           0
ScreenPorch         0
PoolArea            0
PoolQC           1453
Fence            1179
MiscFeature      1406
MiscVal             0
MoSold              0
YrSold              0
SaleType            0
SaleCondition       0
dtype: int64
```

In [11]:
```
housing_target.isnull().sum()
```

Out[11]:
```
0
```

2. Recode string variables and missing values using dummy variables (see `pd.get_dummies` and `pd.fillna`). Should you standardise the input data? Explain how missing values are handled by the model.

- Decision Tree: ignore the missing value and split based on available data
- Random Forest: RF is building trees based on a set of randomly selected features. If there are missing value in some of the features, the algorithm will compute the impurity of that feature using available data and assign a weight to each feature and the weight is proportional to the number of non-missing data in that feature.
- Gradient Boosting Trees: This algorithm assign a direction to the missing value when building the tree. It calculate the derivative of loss function w.r.t. the predicted value and update the direction of the missing value. With many iterations, this will refine the model.

Should I standardize the input data?

- for Decision Tree and random forest, the scaling of the input data does not affect the split of the tree. No standarization required for these two models
- For Gradient Boosting Tree, it uses gradient descent to minimize the loss function which is sensitive to the scale of the feature. Thus, standardization is recommended for Gradient boosting tree.

In [14]:
```
housing_data_1 = pd.DataFrame()
for col in housing_data.columns:
    if housing_data[col].dtype == object:
        housing_data_1 = pd.concat([housing_data_1, pd.get_dummies(housing_data[col], prefi
    else:
        housing_data_1[col] = housing_data[col].fillna(housing_data[col].mean())
```

Out[14]:
```
Index(['MSSubClass', 'MSZoning_C (all)', 'MSZoning_FV', 'MSZoning_RH', 'MSZoning_RL', 'MSZo
ning_RM', 'LotFrontage', 'LotArea', 'Street_Grvl', 'Street_Pave',
       ...
       'SaleType_ConLw', 'SaleType_New', 'SaleType_Oth', 'SaleType_WD', 'SaleCondition_Abno
rml', 'SaleCondition_AdjLand', 'SaleCondition_Alloca', 'SaleCondition_Family', 'SaleConditi
on_Normal', 'SaleCondition_Partial'], dtype='object', length=288)
```

3. Randomly split the observations into a training sample (75%) and a test sample (25%). Fit a decision tree model that minimises the squared error to the training sample.

In [40]:
```
x_train, x_test, y_train, y_test = model_selection.train_test_split(housing_data_1, housing

dt = tree.DecisionTreeRegressor(min_samples_leaf=9, random_state=1)
dt.fit(x_train, y_train)
y_pred = dt.predict(x_test)
mse = metrics.mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 1083015082.807594
```

4. Compute the predictive performance on the training and the test sample (see `sklearn.tree`) and comment on the results. Inspect the arguments of the model (see `inspect.signature`) and propose 3 different ways to improve the generalisation performance of the tree model.

- Increase the max_depth of the model. The model is more flexible but in the risk of overfitting
- Increase the max_leaf_nodes so that the tree can have more split
- Specify a min_impurity_decrease value which only split when the decrease of impurity is over this value.

In [26]:
```
train_mse = dt.score(x_train, y_train)
test_mse = dt.score(x_test, y_test)
print("Training Score:", train_mse)
print("Test Score:", test_mse)

# inspect the arguments of the model
args = inspect.signature(tree.DecisionTreeRegressor)
print(args)
```

```
Training Score: 0.8787659822105276
Test Score: 0.8386744682849232
(*, criterion='squared_error', splitter='best', max_depth=None, min_samples_split=2, min_sa
mples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_
nodes=None, min_impurity_decrease=0.0, ccp_alpha=0.0)
```

5. Fit a random forest model with 25 trees to the training data (see `sklearn.ensemble`) and compute the predictive performance on both samples. How does it compare to the performance of the tree model? Explain why.

Compared to the Decision Tree, Random Forest gives better performance on both training set and test set. This is because random forest built trees based on randomly selected features. therefore, overfitting issue is reduced and model can understand the pattern in the data better. The random forest average the prediction so that the variance is reduced as well.

In [44]:
```
rf = ensemble.RandomForestRegressor(n_estimators=25, random_state=1)
rf.fit(x_train, y_train)
rf_train_mse = rf.score(x_train, y_train)
rf_test_mse = rf.score(x_test, y_test)
print("Training Score:", rf_train_mse)
print("Test Score:", rf_test_mse)
```

```
Training Score: 0.9719753384241091
Test Score: 0.8950142533441221
```

6. Compute the out-of-bag score (see the `oob_score` parameter of the model) and explain how this estimate compares to the test score.

Even though the out-of-bag score can reflect the generalized performance of the model, it is less accurate than the test score, especially for small dataset. It also underestimates the test error because it is only using a small portion of the entire dataset for each tree whereas the test score using all. Therefore, it cannot replace the test set.

In [45]:
```
rf_oob = ensemble.RandomForestRegressor(n_estimators=25, random_state=1, oob_score=True)
rf_oob.fit(x_train, y_train)
oob_score = rf_oob.oob_score_
print(f"OOB score: {oob_score}" )
```

```
OOB score: 0.7936874232245372
```

7. Explain how we can estimate variable importance when using tree-based models and compute these estimates (see `feature_importances_` method) and identify the 10 most important variables. How do you expect correlation among input variables to affect these estimates?

There are two ways to measure the importance of variables, Gini importance and mean decrease impurity. Gini importance measures the decrease in impurity by each feature over all the decision trees. Features that cause higher reduction in impurity are considered more important. The mean decrease impurity method evaluates the significance of a feature by determining how much the average squared error (MSE) of the model increases when the feature values are randomly shuffled within the out-of-bag (OOB) samples.

In [47]:
```
rf_importance = rf.feature_importances_
indices = np.argsort(rf_importance)[::-1][:10]
top_features = [x_train.columns[i] for i in indices] # type: ignore
top_features
```

Out[47]:
```
['OverallQual',
 'GrLivArea',
 'TotalBsmtSF',
 '1stFlrSF',
 'BsmtFinSF1',
 'LotArea',
 'GarageCars',
 '2ndFlrSF',
 'GarageArea',
 'ExterQual_Ex']
```

The correlation between two variables can affect the estimation on the importance of the variables. If two variables are highly correlated, then they could be both estimated as important variable whereas in fact only one of them truly is. In this case, feature importance is not accurate.

8. Optimise the tuning parameters `n_estimators` and `max_features` of the model using grid-search 5-fold cross-validation (see `model_selection.GridSearchCV`), and find a combination of parameters that improves on question 5 (see the `best_params_` and `best_score_` methods).

In [52]:
```
param_grid = {
    'n_estimators': [5, 10, 15, 20, 25, 50, 100, 500, 1000],
    'max_features': ['sqrt', 'log2']
}
```

```
rf = ensemble.RandomForestRegressor(random_state=1)

grid_search = model_selection.GridSearchCV(rf, param_grid, cv=5)
grid_search.fit(x_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Train Score:", grid_search.best_score_)
print("Test Score:", grid_search.best_estimator_.score(x_test, y_test)) # type: ignore
```

```
Best Parameters: {'max_features': 'sqrt', 'n_estimators': 1000}
Best Train Score: 0.8303188611835907
Test Score: 0.8656453266416815
```

9. Explain the advantage of gradient boosting over individual tree. Fit a gradient boosting model with 100 trees to the training data. How does the generalisation performance compare to the decision tree and the random forest models?

Gradient Boosting builds simple models first,them build more complex versions of the previous model by learning from the mistakes previous models made. In this case, unlike simple model decision trees which can overfit the data, Gradient boosting can reduce variance and bias. There are several advantages of Gradient boosting over individual tree.

- the ability of handling nonlinear relationships between features and target variables
- is able to capture interactions between features
- can handle missing data and outliers rather than just ignore them
- can provide feature importance measures

In [53]:
```
gb = ensemble.GradientBoostingRegressor(n_estimators=100, random_state=1)
gb.fit(x_train, y_train)
gb_train_mse = gb.score(x_train, y_train)
gb_test_mse = gb.score(x_test, y_test)
print("Training Score:", gb_train_mse)
print("Test Score:", gb_test_mse)
```

```
Training Score: 0.9682749521594058
Test Score: 0.9187566484151645
```

In [54]:
```
print(f"the test MSE of decision tree: {test_mse}")
print(f"the test MSE of random forests: {rf_test_mse}")
print(f"the test MSE of gradient boosting tree: {gb_test_mse}")
```

```
the test MSE of decision tree: 0.8386744682849232
the test MSE of random forests: 0.8950142533441221
the test MSE of gradient boosting tree: 0.9187566484151645
```

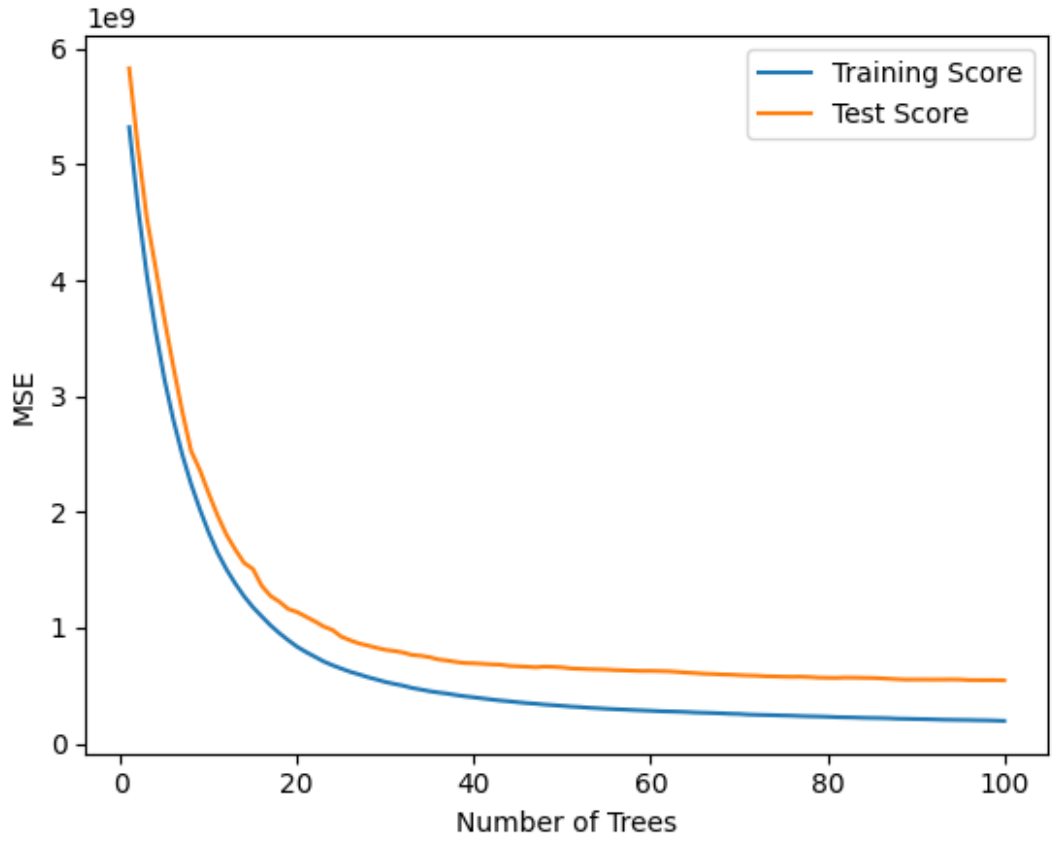
By comparing the test score, we can easily see that the gradient boosting algorithm outperform both random forest and individual decision tree. However, one thing to note is that this comparison only look at the prediction accuracy, not considering the model interpretability, complexity, and robustness.

10. Plot the evolution of the training and test scores with each boosting iteration. Show the impact of increasing (e.g. 0.5) or decreasing (e.g. 0.01) the model's learning rate on the optimisation path.

By comparing the high learning rate, medium learning rate, and low learning rate, we can see that high learning rate will cause model coverage too quickly, whereas low learning rate would cause the model too slow to converge. The disadvantage for high learning rate is that, the model may overfit the data and cause model not robust enough to the new unseen data. This is because each tree has stronger influence in the

final prediction. In the contrast, low learning rate would make each tree less influential to the final prediction, therefore, it would require lots of trees to reach the performance of a higher learning rate model.

In [58]:
```
gb = ensemble.GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, random_state=1
gb.fit(x_train, y_train)
train_pred = list(gb.staged_predict(x_train))
test_pred = list(gb.staged_predict(x_test))
train_scores = [metrics.mean_squared_error(y_train, pred) for pred in train_pred]
test_scores = [metrics.mean_squared_error(y_test, pred) for pred in test_pred]
plt.plot(np.arange(1, 101), train_scores, label='Training Score')
plt.plot(np.arange(1, 101), test_scores, label='Test Score')
plt.xlabel('# of Trees')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



In [60]:
```
gb_high_learningrate = ensemble.GradientBoostingRegressor(n_estimators=100, learning_rate=0

gb_high_learningrate.fit(x_train, y_train)
train_pred_high_learning_rate = list(gb_high_learningrate.staged_predict(x_train))
test_pred_high_learning_rate = list(gb_high_learningrate.staged_predict(x_test))
train_scores_high_learningrate = [metrics.mean_squared_error(y_train, pred) for pred in tra
test_scores_high_learningrate = [metrics.mean_squared_error(y_test, pred) for pred in test_

gb_low_learingrate = ensemble.GradientBoostingRegressor(n_estimators=100, learning_rate=0.0
gb_low_learingrate.fit(x_train, y_train)
train_pred_low_learning_rate = list(gb_low_learingrate.staged_predict(x_train))
test_pred_low_learning_rate = list(gb_low_learingrate.staged_predict(x_test))
train_scores_low_learningrate = [metrics.mean_squared_error(y_train, pred) for pred in trai
test_scores_low_learningrate = [metrics.mean_squared_error(y_test, pred) for pred in test_p
plt.plot(np.arange(1, 101), train_scores, label='Train, LR=0.1')
plt.plot(np.arange(1, 101), test_scores, label='Test, LR=0.1')
plt.plot(np.arange(1, 101), train_scores_high_learningrate, label='Train, LR=0.5')
plt.plot(np.arange(1, 101), test_scores_high_learningrate, label='Test, LR=0.5')
```

```
plt.plot(np.arange(1, 101), train_scores_low_learningrate, label='Train, LR=0.01')
plt.plot(np.arange(1, 101), test_scores_low_learningrate, label='Test, LR=0.01')
plt.xlabel('# of Trees')
plt.ylabel('MSE')
plt.legend()
plt.show()
```