DATA SCIENCE 2: ASSIGNMENT 1

Ian Brandenburg(2304791)
GitHub Repo

The assignment is aimed at developing a predictive model that minimizes the loss in order to predict real estate prices in New Taipei City as accurately as possible. The project creates a linear model, multi-linear model, random forest model, and gradient boosted random forest model. Furthermore, feature engineering was conducted to transform some of the variables through squares, interations, and interacting the squqared variables.

Import Libraries

In [1]:
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor

import warnings
warnings.filterwarnings('ignore')
```

Import Data

We only are looking at 20% of the dataset.

In [2]:
```python
# seting the random state to `prng`
prng = np.random.RandomState(20240322)

# loading the data in directly from github
real_estate_data = pd.read_csv("https://raw.githubusercontent.com/divenyijanos/ceu-

# sectioning off 20% of the data into `real_estate_sample`, and setting the random_
real_estate_sample = real_estate_data.sample(frac=0.2, random_state = prng)
```

Set X (features) and Y (outcome) variables for predictive modelling

Setting the outcome to the `house_price_of_unit_area` variable, since this is what we are trying to predict

Set features to: `house_age` , `distance_to_the_nearest_MRT_station` , `number_of_convenience_stores` , `latitude` , `longitude`

Split test and train, with the test size being 30%

In [3]:
```
real_estate_sample.head()
```

Out[3]:

	id	transaction_date	house_age	distance_to_the_nearest_MRT_station	number_of_convenienc
372	373	2013.000	33.9	157.6052	
5	6	2012.667	7.1	2175.0300	
263	264	2013.417	3.9	2147.3760	
345	346	2012.667	0.0	185.4296	
245	246	2013.417	7.5	639.6198	

In [4]:
```
# setting the target variable, outcome variable, or y-variable
outcome = real_estate_sample["house_price_of_unit_area"]

# setting the features, or explanatory variables
features = real_estate_sample[['house_age', 'distance_to_the_nearest_MRT_station',

# splitting the `real_estate_sample` into the training and test sets
X_train, X_test, y_train, y_test = train_test_split(features, outcome, test_size=0.
print(f"Size of the training set: {X_train.shape}, size of the test set: {X_test.sh
```

```
Size of the training set: (58, 3), size of the test set: (25, 3)
```

The size of the training and test set will work for constructing the models and premptively working on them, but will need to be later tested on the full dataset. This is acting like a validation set.

1) Think about an appropriate loss function you can use to evaluate your predictive models. What is the risk (from a business perspective) that you would have to take by making a wrong prediction?

In [5]:
```
# defining the loss function
def calculateRMSLE(prediction, y_obs):
    return round(np.sqrt(
        np.mean(
            (
                np.log(np.where(prediction < 0, 0, prediction) + 1) -
                np.log(y_obs + 1)
            )**2
        )
    ), 4)
```

The Loss Function

This loss function is appropriate, including in a business context or the context of predicting real estate. This function can handle high value predictions, which can be important in price prediction. Unlike Mean Squared Error (MSE), RMSLE can handle asymmetry in prediction errors by comparing the log of predicted values with the log of actual values. Additionally, this function avoids negative predictions `(np.where(prediction < 0, 0, prediction))` here, which will allow for the results to remain interpretable. A negative price prediction would not make sense. The two primary business risks include underestimation and overestimation. Underestimating could cause a loss in revenue, due to setting prices too low, while overestimating could lead to properties not getting purchased or rented due to them being overpriced. Either way, there would be a loss in revenue.

2) Build a simple benchmark model and evaluate its performance on the hold-out set (using your chosen loss function).

In [6]:
```
# estimating benchmark model
benchmark = np.mean(y_train)
benchmark_result = ["Benchmark", calculateRMSLE(benchmark, y_train), calculateRMSLE
```

In [7]:
```
# collecting results into the results_df, so that it is repeatable throughout the c
result_columns = ["Model", "Train", "Test"]
results_df = pd.DataFrame([benchmark_result], columns=result_columns)
results_df
```

Out[7]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221

Here, the bench mark model was created based on the mean of the y variable in the training set. This is a very naive model, and not very accurate. There are definitely improvements that can be made. Additionally, the training and test RMSLE scores are not very close to each other.

3) Build a simple linear regression model using a chosen feature and evaluate its performance. Would you launch your evaluator web app using this model?

In [8]:
```
# building and fitting the simple linear regression model
lin_reg = LinearRegression().fit(X_train[["distance_to_the_nearest_MRT_station"]],

# calculating the predictions on training and testing sets
train_predictions = lin_reg.predict(X_train[["distance_to_the_nearest_MRT_station"]
test_predictions = lin_reg.predict(X_test[["distance_to_the_nearest_MRT_station"]])

# calculating the RMSLE for the training and testing sets
model_train_rmsle = calculateRMSLE(train_predictions, y_train)
model_test_rmsle = calculateRMSLE(test_predictions, y_test)

# preparing the model's results
model_result = pd.DataFrame([["Simple Linear Regression", model_train_rmsle, model_
                            columns=["Model", "Train", "Test"])

# appending model_result to the existing results_df
results_df = pd.concat([results_df, model_result], ignore_index=True)
results_df
```

Out[8]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305

The simple linear regression was run using the `distance_to_the_nearest_MRT_station` variable. This model did significantly improved the RMSLE scores to 0.2305 in the test set. However, it is too naive and basic to be considered for evaluating an app, and thus should not be used. This model can be further improved.

4) Build a multivariate linear model with all the meaningful variables available. Did it improve the predictive power?

In [9]:
```
# reset features
features = ['house_age', 'distance_to_the_nearest_MRT_station', 'number_of_convenie

# building and training the model
lin_reg_multi = LinearRegression()
lin_reg_multi.fit(X_train[features], y_train)

# calculating the predictions on training and testing sets
train_predictions_multi = lin_reg_multi.predict(X_train[features])
test_predictions_multi = lin_reg_multi.predict(X_test[features])

# calculating the RMSLE for the training and testing sets
model_train_rmsle_multi = calculateRMSLE(train_predictions_multi, y_train)
model_test_rmsle_multi = calculateRMSLE(test_predictions_multi, y_test)

# preparing the model's results
model_result_multi = pd.DataFrame([["Multivariate Linear Regression", model_train_r
                            columns=["Model", "Train", "Test"])

# appending the model_result to the existing results_df
results_df = pd.concat([results_df, model_result_multi], ignore_index=True)
results_df
```

Out[9]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305
2	Multivariate Linear Regression	0.1993	0.2317

The multivatiate linear regression did not improve the loss score by very much, and furthered the gap between the train and test set scores as compared to the simple linear regression. However, these scores could still be improved, since three features is not very many in accurately predicting. Additionally, this could still yield over or underestimating prices with a large error margin.

5) Try to make your model (even) better. Document your process and its success while taking two approaches:

1. Feature engineering - e.g. including squares and interactions or making sense of latitude & longitude by calculating the distance from the city center, etc.
2. Training more flexible models - e.g. random forest or gradient boosting

Feature Engineering

```
In [10]: # squared terms
real_estate_sample['house_age_squared'] = real_estate_sample['house_age'] ** 2
real_estate_sample['distance_to_the_nearest_MRT_station_squared'] = real_estate_sam
real_estate_sample['number_of_convenience_stores_squared'] = real_estate_sample['nu

# interaction terms
real_estate_sample['age_x_distance'] = real_estate_sample['house_age'] * real_estat
real_estate_sample['age_x_stores'] = real_estate_sample['house_age'] * real_estate_
real_estate_sample['distance_x_stores'] = real_estate_sample['distance_to_the_neare

# interactions between squared terms
real_estate_sample['age_squared_x_distance_squared'] = real_estate_sample['house_ag
real_estate_sample['age_squared_x_stores_squared'] = real_estate_sample['house_age_
real_estate_sample['distance_squared_x_stores_squared'] = real_estate_sample['dista

real_estate_sample.head()
```

Out[10]:

	id	transaction_date	house_age	distance_to_the_nearest_MRT_station	number_of_convenienc
372	373	2013.000	33.9	157.6052	
5	6	2012.667	7.1	2175.0300	
263	264	2013.417	3.9	2147.3760	
345	346	2012.667	0.0	185.4296	
245	246	2013.417	7.5	639.6198	

Feature engineering was conducted in order to add more features to be analyzed and considering in the upcoming flexible models.

- *Squared Terms*: Each of the three main variables looked at in the multivariate model were squared.
- *Interaction terms*: Each of the three variables were interacted to identify possible interaction associations.
- *Interactions between Squared Terms*: Each of the squared terms were interacted to get a greater indepth perspective.

Distance from City Center using Lat/Long

In [11]:
```
# coordinates of New Taipei City center
city_center_lat = 25.0143
city_center_lon = 121.4672

def haversine(lat1, lon1, lat2, lon2):
    # radius of the Earth in kilometers
    R = 6371.0
    # convert latitude and longitude from degrees to radians
    lat1_rad = np.radians(lat1)
    lon1_rad = np.radians(lon1)
    lat2_rad = np.radians(lat2)
    lon2_rad = np.radians(lon2)
    # computer differences in coordinates
    dlat = lat2_rad - lat1_rad
    dlon = lon2_rad - lon1_rad
    # apply the Haversine formula
    a = np.sin(dlat / 2)**2 + np.cos(lat1_rad) * np.cos(lat2_rad) * np.sin(dlon / 2
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))
    distance = R * c
    return distance

# calculating the the distance for each property
real_estate_sample['distance_to_city_center'] = real_estate_sample.apply(
    lambda row: haversine(row['latitude'], row['longitude'], city_center_lat, city_
real_estate_sample.head()
```

Out[11]:

	id	transaction_date	house_age	distance_to_the_nearest_MRT_station	number_of_convenienc
372	373	2013.000	33.9	157.6052	
5	6	2012.667	7.1	2175.0300	
263	264	2013.417	3.9	2147.3760	
345	346	2012.667	0.0	185.4296	
245	246	2013.417	7.5	639.6198	

The function above, using the assistance of chatgpt, calculates the distances from the city center of Taipei New City by utilizing the `latitude` and `longitude` values. It appends a new column named `distance_to_city_center` to the real_esate_sample. This allows us to use this variable in the predictive models.

Random Forest Model

In [12]:
```
# defining the features and target variable
features = ['house_age', 'distance_to_the_nearest_MRT_station', 'number_of_convenie
            'latitude', 'longitude', 'house_age_squared', 'distance_to_the_nearest_
            'number_of_convenience_stores_squared', 'age_x_distance', 'age_x_stores
            'age_squared_x_distance_squared', 'age_squared_x_stores_squared', 'dist
            'distance_to_city_center']
outcome = 'house_price_of_unit_area'

# splitting the data into training and testing sets
features_fe = real_estate_sample[features]
outcome_fe = real_estate_sample[outcome]
X_train_fe, X_test_fe, y_train_fe, y_test_fe = train_test_split(features_fe, outcom

# defining the pipeline for the random forest
pipe_rf = Pipeline([
    ("random_forest", RandomForestRegressor(random_state=prng))
])

# fitting the model on the training data
pipe_rf.fit(X_train_fe, y_train_fe)

# calculating the RMSLE on the prediction
train_error = calculateRMSLE(pipe_rf.predict(X_train_fe), y_train_fe)
test_error = calculateRMSLE(pipe_rf.predict(X_test_fe), y_test_fe)

# preparing the the model's results and concatinating to results_df
model_result_rf = pd.DataFrame([["FE Random Forest", train_error, test_error]],
                               columns=["Model", "Train", "Test"])
results_df = pd.concat([results_df, model_result_rf], ignore_index=True)
results_df
```

Out[12]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305
2	Multivariate Linear Regression	0.1993	0.2317
3	FE Random Forest	0.0815	0.1449

Here, the feature engineered random forest shows to be performing better in the training set, but the test sample is not performing much better. Nevertheless, the FE Random Forest test RMSLE score is better than the multivariate linear regression, impliying the possibility of some of the feature engineered variables adding mor explanation of the variance in the price variable. With a large gap between the train and test scores, this suggests an overfitting, possibly due to the small number of observations in the training and test sets and a large number of features.

Gradient Boosted RF Model

In [13]:
```
# defining the pipeline with XGBRegressor
pipe_xgb = Pipeline([
    ("gradient_boosting", XGBRegressor(random_state=prng))
])

# fitting the model on the training data
pipe_xgb.fit(X_train_fe, y_train_fe)

# calculating the RMSLE on the prediction
train_error_xgb = calculateRMSLE(pipe_xgb.predict(X_train_fe), y_train_fe)
test_error_xgb = calculateRMSLE(pipe_xgb.predict(X_test_fe), y_test_fe)

# preparing and concatinating the model's results to results_df
model_result_xgb = pd.DataFrame([["FE Gradient Boosted RF", train_error_xgb, test_e
                                columns=["Model", "Train", "Test"])

results_df = pd.concat([results_df, model_result_xgb], ignore_index=True)
results_df
```

Out[13]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305
2	Multivariate Linear Regression	0.1993	0.2317
3	FE Random Forest	0.0815	0.1449
4	FE Gradient Boosted RF	0.0195	0.1890

The feature engineered gradient boosted random forest does not seem like it's doing better, but with further investigtion, it seems to be overfitting as a result of the difference between the training and test RMSLE scores. The RMSLE score on the test set is actually higher in the gradient boosted model compared to the random forest model, with 0.1890 in the gradient boosted model and a 0.1449 in the random forest model.

6) Would you launch your web app now? What options you might have to further improve the prediction performance?

I would not launch my web app now, because the data seems to be overfitting. To further improve, increasing the number of observations should help minimzing the overfitting. This could be done through collecting more observations, or potentially boost strapping. The issue with boostrapping here is that there are very few observations, which might not be representative of the entire dataset. So, collecting more observations would be ideal. Additionally, adjusting the hyperparameters of the models could further strengthen the models.

7) Rerun three of your previous models (including both flexible and less flexible ones) on the full train set. Ensure that your test result remains comparable by keeping that dataset intact.

In [14]:
```
# excluding the existing X_test
real_estate_full = real_estate_data.loc[~real_estate_data.index.isin(X_test.index)]
print(f"Size of the full training set: {real_estate_full.shape}")
```

```
Size of the full training set: (389, 8)
```

Here, the full dataset is added on, excluding the X test set. This will allow us to see the improvement of the models when more data is added.

In [15]:
```
# setting the training values
X_full_train = real_estate_full[['house_age', 'distance_to_the_nearest_MRT_station'
y_full_train = real_estate_full['house_price_of_unit_area']
```

The full dataset was used as a training set, and the X and Y variables are set manually above the same as they were for the multilinear regression from before.

Multivariate Linear Regression

In [16]:
```
# initializing the linear regression model
lin_reg_multi_full = LinearRegression()

# building and fitting the simple linear regression model
lin_reg_multi_full.fit(X_full_train, y_full_train)

# calculating predictions on the full training set and the original test set
train_predictions_full = lin_reg_multi_full.predict(X_full_train)
test_predictions_full = lin_reg_multi_full.predict(X_test)

# calculating the RMSLE for the full training set and original test set
model_train_rmsle_full = calculateRMSLE(train_predictions_full, y_full_train)
model_test_rmsle_full = calculateRMSLE(test_predictions_full, y_test)

# preparing the model's results
model_result_full = pd.DataFrame([["Multivariate Linear Regression (FULL)", model_t
                            columns=["Model", "Train", "Test"])

# concatinating model_result to results_df
results_df = pd.concat([results_df, model_result_full], ignore_index=True)
results_df
```

Out[16]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305
2	Multivariate Linear Regression	0.1993	0.2317
3	FE Random Forest	0.0815	0.1449
4	FE Gradient Boosted RF	0.0195	0.1890
5	Multivariate Linear Regression (FULL)	0.2639	0.2152

The multivariate linear regression run on the original test set using the full dataset did not see a significant improvement. There is a separation between the training and test RMSLE values, with the training set being larger than the tests. The test set RMSLE value did decrease by a few, which could suggest that the addition in data did help a small amount.

Feature Engineering

Feature engineering was conducted again on the full dataset. This was done so that we can see the benefit of adding more data on the feature engineered models.

In [17]:
```
# squared terms
real_estate_full['house_age_squared'] = real_estate_full['house_age'] ** 2
real_estate_full['distance_to_the_nearest_MRT_station_squared'] = real_estate_full[
real_estate_full['number_of_convenience_stores_squared'] = real_estate_full['number

# interaction terms
real_estate_full['age_x_distance'] = real_estate_full['house_age'] * real_estate_fu
real_estate_full['age_x_stores'] = real_estate_sample['house_age'] * real_estate_fu
real_estate_full['distance_x_stores'] = real_estate_full['distance_to_the_nearest_M

# interactions between squared terms
real_estate_full['age_squared_x_distance_squared'] = real_estate_full['house_age_sq
real_estate_full['age_squared_x_stores_squared'] = real_estate_full['house_age_squa
real_estate_full['distance_squared_x_stores_squared'] = real_estate_full['distance_

real_estate_full.head()
```

Out[17]:

	id	transaction_date	house_age	distance_to_the_nearest_MRT_station	number_of_convenience_sto
0	1	2012.917	32.0	84.87882	
1	2	2012.917	19.5	306.59470	
2	3	2013.583	13.3	561.98450	
3	4	2013.500	13.3	561.98450	
5	6	2012.667	7.1	2175.03000	

The distance from the center using the latitude and longitude is calculated again for the full dataset.

In [18]:
```
# calculating the distance for each property
real_estate_full['distance_to_city_center'] = real_estate_full.apply(
    lambda row: haversine(row['latitude'], row['longitude'], city_center_lat, city_
real_estate_full.head()
```

Out[18]:

	id	transaction_date	house_age	distance_to_the_nearest_MRT_station	number_of_convenience_sto
0	1	2012.917	32.0	84.87882	
1	2	2012.917	19.5	306.59470	
2	3	2013.583	13.3	561.98450	
3	4	2013.500	13.3	561.98450	
5	6	2012.667	7.1	2175.03000	

In [19]:
```
# setting the training values using the full dataset and engineered features
X_full_train_fe = real_estate_full[['house_age', 'distance_to_the_nearest_MRT_stati
            'latitude', 'longitude', 'house_age_squared', 'distance_to_the_nearest_
            'number_of_convenience_stores_squared', 'age_x_distance', 'age_x_stores
            'age_squared_x_distance_squared', 'age_squared_x_stores_squared', 'dist
            'distance_to_city_center']]

y_full_train_fe = real_estate_full['house_price_of_unit_area']
```

Random Forest Model with the Feature Engineered Full Dataset

In [20]:
```
# defining the pipeline
pipe_rf = Pipeline([
    ("random_forest", RandomForestRegressor(random_state=prng))
])

# fitting the model on the training data
pipe_rf.fit(X_full_train_fe, y_full_train_fe)

# calculating the RMSLE on the predictions
train_error = calculateRMSLE(pipe_rf.predict(X_full_train_fe), y_full_train_fe)
test_error = calculateRMSLE(pipe_rf.predict(X_test_fe), y_test_fe)

# preparing the model's results and concatinating to results_df
model_result_rf = pd.DataFrame([["FE Random Forest (FULL)", train_error, test_error
                               columns=["Model", "Train", "Test"])
results_df = pd.concat([results_df, model_result_rf], ignore_index=True)
results_df
```

Out[20]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305
2	Multivariate Linear Regression	0.1993	0.2317
3	FE Random Forest	0.0815	0.1449
4	FE Gradient Boosted RF	0.0195	0.1890
5	Multivariate Linear Regression (FULL)	0.2639	0.2152
6	FE Random Forest (FULL)	0.0811	0.1100

The feature engineered random forest modul using the full training set suggests an improvement in the predictive modelling, both compared to the random forest model conducted on 20% of the dataset, as well as the multivariate linear regression run on the full dataset. The RMSLE scores on the test set are more similar to each other, and are the lowest seen so far for the test set. This implies that the addition of data improved the model. The gradient boosting will be run to compare to the random forest and see if there is further improvement.

Gradient Boosted Random Forest with the Feature Engineered Full Training Set

In [21]:

```
# defining the pipeline with XGBRegressor
pipe_xgb = Pipeline([
    ("gradient_boosting", XGBRegressor(random_state=prng))
])

# fitting the model on the training data
pipe_xgb.fit(X_full_train_fe, y_full_train_fe)

# calculating the RMSLE on the predictions
train_error_xgb = calculateRMSLE(pipe_xgb.predict(X_full_train_fe), y_full_train_fe
test_error_xgb = calculateRMSLE(pipe_xgb.predict(X_test_fe), y_test_fe)

# preparing and concatinating the model's results to results_df
model_result_xgb = pd.DataFrame([["FE Gradient Boosted RF (FULL)", train_error_xgb,
                                columns=["Model", "Train", "Test"])

results_df = pd.concat([results_df, model_result_xgb], ignore_index=True)
results_df
```

Out[21]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Simple Linear Regression	0.2250	0.2305
2	Multivariate Linear Regression	0.1993	0.2317
3	FE Random Forest	0.0815	0.1449
4	FE Gradient Boosted RF	0.0195	0.1890
5	Multivariate Linear Regression (FULL)	0.2639	0.2152
6	FE Random Forest (FULL)	0.0811	0.1100
7	FE Gradient Boosted RF (FULL)	0.0272	0.1081

Here, the feature engineered gradient boosted random forest using the full training data shows to have a better test score than the random forest run on the full training set. However, with the difference between the training and test set RMSLE in the gradient boosted model, it suggests that there is potentially some overfitting occuring. Nevertheless, there is significant improvement in the the gradient boosted model using the full training data as compared to using 20% of the data. This suggests that the additional data improves the models.

Did it improve the predictive power of your models?

The addition of data did improve the predictive power of my models, which was seen mostly in the gradient boosting and random forest models. It was not seen as strongly in the multivariate linear regression, likely due to the small number of features included in this model.

Where do you observe the biggest improvement?

The most significant improvment in RMSLE score from the 20% dataset model to the full dataset model was the gradient boosting model, which was initially at a 0.1890 RMSLE score and went down to 0.1081 once the additional data was added in. However, this might not be the best model due to the significant difference between the training and test set RMSLE in the gradient boosting. So, the ideal model to go with at this point would be the Random Forest model on the feature engineered full data set. This is because it has nearly the same RMSLE score as compared to the gradient boosting, but a more comparable training RMSLE.

Would you launch your web app now?

There would still be some risk in launching the app now, because of the RMSLE of the ideal model in the test set being 0.11. This could result in over- or underestimating the prices of real estate in New Taipei City. Furthermore, more data may need to be collected to better fit the models for an app launch and create more accurate predictions. The addition of data clearly improved the models perdictive capabilities, so furthering this would enhance the effectiveness of an app. So no, I would not launch the web app now, I would collect more data first.