ds2-assignment1-katona

March 22, 2024

#

Data Science 2 - Assignment 1

Created by Zsófia Rebeka Katona

The goal of this assignment was to build predictive models to predict the property prices in New Taipei City, Taiwan. The tasks included building a simple benchmark model, linear and multivariate regression models and to explore other ensemble methods to improve prediction performance, such as RandomForest or Gradient Boosting. I assessed how my models perform with the full training set. Finally, I analyzed the business risks associated with wrong predictions and considered whether to launch a web app based on the best model's performance.

0.0.1 1. Predict real estate value (20 points)

[1]:
```
# Importing required libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

[2]:
```
# Setting the RandomState with Pseudo Random Number Generator
prng = np.random.RandomState(20240322)

# Importing the Taipei real estate dataset
real_estate_data = pd.read_csv("https://raw.githubusercontent.com/divenyijanos/
 ↪ceu-ml/2023/data/real_estate/real_estate.csv")

# Getting a validation set, which is 20% subsample of the original data
real_estate_sample = real_estate_data.sample(frac=0.2, random_state = prng)

# Checking the DataFrame
real_estate_data.head(5)
```

[2]:
```
   id  transaction_date  house_age  distance_to_the_nearest_MRT_station  \
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
```

```
4   5          2012.833        5.0                             390.56840

   number_of_convenience_stores  latitude  longitude  house_price_of_unit_area
0                            10  24.98298  121.54024                      37.9
1                             9  24.98034  121.53951                      42.2
2                             5  24.98746  121.54391                      47.3
3                             5  24.98746  121.54391                      54.8
4                             5  24.97937  121.54245                      43.1
```

[3]:
```python
# Checking the columns
print(real_estate_data.columns)
```

```
Index(['id', 'transaction_date', 'house_age',
       'distance_to_the_nearest_MRT_station', 'number_of_convenience_stores',
       'latitude', 'longitude', 'house_price_of_unit_area'],
      dtype='object')
```

I have considered choosing the following variables for the first set of features: `house_age`, `distance_to_the_nearest_MRT_station` and `number_of_convenience_stores`. The `transaction_date` variable had confusing values, such as `2012.833`, therefore I decided not to include it. Regarding the `latitude` and `longitude` columns, I have opted to include later in the ensemble methods, such as RandomForest or Gradient Boosting.

[4]:
```python
# Defining the features set
features = real_estate_sample[["house_age",
                               "distance_to_the_nearest_MRT_station",
                               "number_of_convenience_stores",
                               ]]

# Defining the target variable as outcome
outcome = real_estate_sample["house_price_of_unit_area"]

# Splitting the training and test samples and setting the test size to 30%
X_train, X_test, y_train, y_test = train_test_split(features, outcome,␣
 ↪test_size=0.3, random_state=prng)

# Printing the size of the training and the test samples
print(f"Size of the training set: {X_train.shape}, size of the test set:␣
 ↪{X_test.shape}")
```

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

0.0.2 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? (2 points)

The primary goal in real estate is to maximize the profit for property investors, or to minimize the risks. Therefore, making the wrong predictions can lead to wrong decisions, financial implications, for instance financial losses or incorrect pricing. Wrong pricing can lead to difficulties in selling the property, resulting in the amortization (while the property not sold), selling it below market value or missing an investment opportunity (for example, if the real estate sector is booming during that time period). Consdiering the impact of wrong predictions, an appropriate loss model for real estate case could be the RMSE or the RMSLE. These loss functions measure the average difference between the predicted and the actual house prices, providing a clear understanding of the magnitude of prediction errors.

[5]:
```python
# Defining the loss function (root mean squared logarithmic error)
def calculateRMSLE(prediction, y_obs):
    """
    Calculate the Root Mean Squared Logarithmic Error (RMSLE)
    between the predicted values and the observed values.

    Parameters:
    - prediction: array-like, the predicted values.
    - y_obs: array-like, the observed (true) values.

    Returns:
    - float: the RMSLE value rounded to four decimal places.
    """
    return round(np.sqrt(
        np.mean(
                (
                    np.log(np.where(prediction < 0, 0, prediction) + 1) -
                    np.log(y_obs + 1)
                )**2
        )
    ), 4)
```

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

[6]:
```python
# Estimating a benchmark by choosing y_train as our benchmark and taking
# ... its average
benchmark = np.mean(y_train)

# Calculating the RMSLE of the training and the test set based on the benchmark
benchmark_result = ["Benchmark", calculateRMSLE(benchmark, y_train),
                    calculateRMSLE(benchmark, y_test)]

# Defining the columns of our results_df DataFrame
result_columns = ["Model", "Train", "Test"]

# Adding the benchmark results to the DataFrame
results_df = pd.DataFrame([benchmark_result], columns=result_columns)
```

```python
# Displaying the results
results_df
```

[6]:
```
        Model   Train    Test
0  Benchmark  0.3434  0.3221
```

We can conclude that the train RMSLE is slightly higher (0.3434) compared to the test RMSLE (0.3221), which suggests that the test set is performing better with the simple benchmark model. This indicates that the model performs better on unseen data, and its performance generalizes well to new data.

0.0.4 Build a simple linear regression model using a chosen feature and evaluate its performance. Would you launch your evaluator web app using this model? (2 points)

I have chosen three features for my regressions: `house_age`, `distance_to_the_nearest_MRT_station` and `number_of_convenience_stores`. I have chosen `distance_to_the_nearest_MRT_station` to run the linear regression with. Therefore, I could use all the features for the multivariate regression later on.

[7]:
```python
from sklearn.linear_model import LinearRegression

# Initializing the linear regression model
lin_reg = LinearRegression().
 ↪fit(X_train[["distance_to_the_nearest_MRT_station"]],
                                 y_train)

# 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 RMSLE for the training and test 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([["Linear Regression", model_train_rmsle,
                              model_test_rmsle]],
                            columns=["Model", "Train", "Test"])

# Appending the results to the existing results_df DataFrame
results_df = pd.concat([results_df, model_result], ignore_index=True)

# Displaying the updated results
results_df
```

[7]:
```
                 Model   Train    Test
0            Benchmark  0.3434  0.3221
1  Linear Regression  0.2250  0.2305
```

The linear model is performing slighlty better comparing to the simple benchmark. (0.2250) for the training set and (0.2305) for the test set. The test set is performing weaker than the training set, suggesting that the model may be overfitting to the training data and might not generalize well to unseen data. After running linear regressions with each of the variables separately, I have found that `distance_to_the_nearest_MRT_station` is the feature that results in the lowest RMSE value.

0.0.5 Build a multivariate linear model with all the meaningful variables available. Did it improve the predictive power? (2 points)

[8]:
```python
# Setting the first group of features
features = ["house_age", "distance_to_the_nearest_MRT_station",
            "number_of_convenience_stores"]

# Initializing the linear regression model
multi_lin_reg = LinearRegression()
multi_lin_reg.fit(X_train[features], y_train)

# Predictions on training and testing sets
train_predictions_multi = multi_lin_reg.predict(X_train[features])
test_predictions_multi = multi_lin_reg.predict(X_test[features])

# Calculating the errors
multi_model_rmsle_train = calculateRMSLE(train_predictions_multi, y_train)
multi_model_rmsle_test = calculateRMSLE(test_predictions_multi, y_test)

# Preparing the model's results
multi_model_result = pd.DataFrame([["Multivariate Regression",
                                    multi_model_rmsle_train,
                                    multi_model_rmsle_test]],
                                  columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
results_df = pd.concat([results_df, multi_model_result], ignore_index=True)

# Displaying the updated results
results_df
```

[8]:
```
                       Model   Train    Test
0                  Benchmark  0.3434  0.3221
1          Linear Regression  0.2250  0.2305
2  Multivariate Regression  0.1993  0.2317
```

The multivariate regression improved our RMSLE scores more than the simple linear model, suggesting that including additional features in the model has helped capture more of the variability in the target variable. This resulted in more accurate predictions. Based on that, we can conclude that the selected features have meaningful relationships with the target variable and contribute positively to the predictive performance of the model. The training set RMSLE is lower than for the test set, which indicates that the model is still overfitting to the training data and doesn't generalize well on unseen data.

0.0.6 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&lo
2. Training more flexible models - e.g. random forest or gradient boosting (6 points)
```

0.1 #### Including squares and interactions

[9]:
```
# Creating the squared variables
real_estate_sample["house_age_sq"] = real_estate_sample["house_age"] ** 2
real_estate_sample["distance_to_the_nearest_MRT_station_sq"] =␣
 ↪real_estate_sample["distance_to_the_nearest_MRT_station"] ** 2
real_estate_sample["number_of_convenience_stores_sq"] =␣
 ↪real_estate_sample["number_of_convenience_stores"] ** 2

# Creating the interactions
real_estate_sample["house_age_distance_interaction"] =␣
 ↪real_estate_sample["house_age"] *␣
 ↪real_estate_sample["distance_to_the_nearest_MRT_station"]
real_estate_sample["house_age_stores_interaction"] =␣
 ↪real_estate_sample["house_age"] *␣
 ↪real_estate_sample["number_of_convenience_stores"]
real_estate_sample["distance_stores_interaction"] =␣
 ↪real_estate_sample["distance_to_the_nearest_MRT_station"] *␣
 ↪real_estate_sample["number_of_convenience_stores"]

# Creating the interactions between the squared variables
real_estate_sample["house_age_distance_interaction_sq"] =␣
 ↪real_estate_sample["house_age"] *␣
 ↪real_estate_sample["distance_to_the_nearest_MRT_station"] *␣
 ↪real_estate_sample["house_age_sq"] *␣
 ↪real_estate_sample["distance_to_the_nearest_MRT_station_sq"]
real_estate_sample["house_age_stores_interaction_sq"] =␣
 ↪real_estate_sample["house_age"] *␣
 ↪real_estate_sample["number_of_convenience_stores"] *␣
 ↪real_estate_sample["house_age_sq"] *␣
 ↪real_estate_sample["number_of_convenience_stores_sq"]
```

```python
real_estate_sample["distance_stores_interaction_sq"] = real_estate_sample["distance_to_the_nearest_MRT_station"] * real_estate_sample["number_of_convenience_stores"] * real_estate_sample["distance_to_the_nearest_MRT_station_sq"] * real_estate_sample["number_of_convenience_stores_sq"]

real_estate_sample.head(5)
```

[9]:

```
      id  transaction_date  house_age  distance_to_the_nearest_MRT_station  \
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

     number_of_convenience_stores  latitude  longitude  \
372                             7  24.96628  121.54196
5                               3  24.96305  121.51254
263                             3  24.96299  121.51284
345                             0  24.97110  121.53170
245                             5  24.97258  121.54814

     house_price_of_unit_area  house_age_sq  \
372                      41.5       1149.21
5                        32.1         50.41
263                      31.7         15.21
345                      37.9          0.00
245                      40.8         56.25

     distance_to_the_nearest_MRT_station_sq  number_of_convenience_stores_sq  \
372                            2.483940e+04                               49
5                              4.730756e+06                                9
263                            4.611224e+06                                9
345                            3.438414e+04                                0
245                            4.091135e+05                               25

     house_age_distance_interaction  house_age_stores_interaction  \
372                      5342.81628                         237.3
5                       15442.71300                          21.3
263                      8374.76640                          11.7
345                         0.00000                           0.0
245                      4797.14850                          37.5

     distance_stores_interaction  house_age_distance_interaction_sq  \
372                    1103.2364                       1.525144e+11
5                      6525.0900                       3.682738e+12
263                    6442.1280                       5.873786e+11
```

```
345                    0.0000                  0.000000e+00
245                 3198.0990                  1.103950e+11

     house_age_stores_interaction_sq  distance_stores_interaction_sq
372                     1.336267e+07                    1.342783e+09
5                       9.663597e+03                    2.778174e+11
263                     1.601613e+03                    2.673548e+11
345                     0.000000e+00                    0.000000e+00
245                     5.273438e+04                    3.270964e+10
```

0.2 #### Handling the longitude and latitude variables

[10]:
```python
import numpy as np

# Calculating the distance from the city center based on the latitude and␣
↪longitude values

# Defining the coordinates of the city center of Taipei (Taipei Main Station)
city_center = (25.0478, 121.5170)

# Calculating the distance from each data point to the city center using the␣
↪Pythagorean theorem
def euclidean_distance(lat1, lon1, lat2, lon2):
    """
    Calculate the Euclidean distance between two points on the earth's surface
    using their latitude and longitude coordinates.
    """
    # Converting latitude and longitude values from degrees to radians
    lat_diff = np.radians(lat2 - lat1)
    lon_diff = np.radians(lon2 - lon1)

    # Approximating the distance using Pythagorean theorem
    distance = np.sqrt(lat_diff**2 + lon_diff**2)

    return distance

# Applying the function to calculate distances for each data point
real_estate_sample['distance_to_city_center'] =␣
↪euclidean_distance(real_estate_sample['latitude'],␣
↪real_estate_sample['longitude'], city_center[0], city_center[1])

# Checking the new column with the distance in coordinates
real_estate_sample.head(5)
```

[10]:
```
      id  transaction_date  house_age  distance_to_the_nearest_MRT_station  \
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

     number_of_convenience_stores  latitude  longitude  \
372                             7  24.96628  121.54196
5                               3  24.96305  121.51254
263                             3  24.96299  121.51284
345                             0  24.97110  121.53170
245                             5  24.97258  121.54814

     house_price_of_unit_area  house_age_sq  \
372                      41.5       1149.21
5                        32.1         50.41
263                      31.7         15.21
345                      37.9          0.00
245                      40.8         56.25

     distance_to_the_nearest_MRT_station_sq  number_of_convenience_stores_sq  \
372                            2.483940e+04                               49
5                              4.730756e+06                                9
263                            4.611224e+06                                9
345                            3.438414e+04                                0
245                            4.091135e+05                               25

     house_age_distance_interaction  house_age_stores_interaction  \
372                      5342.81628                         237.3
5                       15442.71300                          21.3
263                      8374.76640                          11.7
345                         0.00000                           0.0
245                      4797.14850                          37.5

     distance_stores_interaction  house_age_distance_interaction_sq  \
372                    1103.2364                       1.525144e+11
5                      6525.0900                       3.682738e+12
263                    6442.1280                       5.873786e+11
345                       0.0000                       0.000000e+00
245                    3198.0990                       1.103950e+11

     house_age_stores_interaction_sq  distance_stores_interaction_sq  \
372                     1.336267e+07                    1.342783e+09
5                       9.663597e+03                    2.778174e+11
263                     1.601613e+03                    2.673548e+11
345                     0.000000e+00                    0.000000e+00
245                     5.273438e+04                    3.270964e+10
```

```
     distance_to_city_center
372                 0.001488
5                   0.001481
263                 0.001482
345                 0.001363
245                 0.001421
```

As the latitude and longitude coordinates are not directly convertible to distances in kilometers, the values in the `distance_to_city_center` column are not representative of actual distances in kilometers. Instead, they represent differences in latitude and longitude coordinates.

0.3 #### Feature engineering

[11]:
```
# Setting the second set of features with feature engineering
features_fe = real_estate_sample[[
    "house_age_sq",
    "distance_to_the_nearest_MRT_station_sq",
    "number_of_convenience_stores_sq",
    "house_age_distance_interaction",
    "house_age_stores_interaction",
    "distance_stores_interaction",
    "house_age_distance_interaction_sq",
    "house_age_stores_interaction_sq",
    "distance_stores_interaction_sq",
    "distance_to_city_center"
]]

# Redeclaring the prng before the next split to obtain the same test set
prng = np.random.RandomState(20240322)

# Splitting the data again with the feature engineered variables
X_train_fe, X_test_fe, y_train_fe, y_test_fe = train_test_split(features_fe,␣
 ↪outcome, test_size=0.3, random_state=prng)

# Printing the size of the training and the test samples
print(f"Size of the training set: {X_train_fe.shape}, size of the test set:␣
 ↪{X_test_fe.shape}")
```

```
Size of the training set: (58, 10), size of the test set: (25, 10)
```

[12]:
```
# Initializing the linear regression model with FE
multi_lin_reg_fe = LinearRegression()
multi_lin_reg_fe.fit(X_train_fe, y_train_fe)

# Predictions on training and the test sets
train_predictions_multi_fe = multi_lin_reg_fe.predict(X_train_fe)
test_predictions_multi_fe = multi_lin_reg_fe.predict(X_test_fe)
```

```python
# Calculating the errors
multi_model_rmsle_train_fe = calculateRMSLE(train_predictions_multi_fe,␣
↪y_train_fe)
multi_model_rmsle_test_fe = calculateRMSLE(test_predictions_multi_fe, y_test_fe)

# Preparing the model's results
multi_model_result_fe = pd.DataFrame([["Multivariate Regression with FE",␣
↪multi_model_rmsle_train_fe, multi_model_rmsle_test_fe]],
                            columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
results_df = pd.concat([results_df, multi_model_result_fe], ignore_index=True)

# Displaying the updated results
results_df
```

[12]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Linear Regression	0.2250	0.2305
2	Multivariate Regression	0.1993	0.2317
3	Multivariate Regression with FE	0.1532	0.3236

We can see that the feature engineered multivariate regression gave us surprising results. We can see that the training RMLSE (0.1532) has improved compared to the original multivariate regression. However, the test RMSLE (0.3236) increased compared to the previous models. This indicates that the feature engineered model may have overfitted to the training data, and it failed to generalize well to unseen data.

0.4 ### RandomForest with feature engineering

[13]:
```python
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor

# Defining the steps of the pipeline
steps = [
    ("random_forest", RandomForestRegressor(random_state = prng))
]
# Creating the pipeline object
pipe_rf = Pipeline(steps)

# Fitting the pipeline to the training data
pipe_rf.fit(X_train_fe, y_train_fe)

# Calculating the errors
train_error_rf = calculateRMSLE(pipe_rf.predict(X_train_fe), y_train_fe)
test_error_rf = calculateRMSLE(pipe_rf.predict(X_test_fe), y_test_fe)
```

```
# Preparing the model's results
result_rf = pd.DataFrame([["Random Forest with FE", train_error_rf,␣
↪test_error_rf]],
                            columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
results_df = pd.concat([results_df, result_rf], ignore_index=True)

# Displaying the results
results_df
```

[13]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Linear Regression	0.2250	0.2305
2	Multivariate Regression	0.1993	0.2317
3	Multivariate Regression with FE	0.1532	0.3236
4	Random Forest with FE	0.0682	0.2411

The RandomForest improves our RMSLE scores. It performs well on the test set (0.2411), but even better on the training set (0.0682). This suggests that the RandomForest model may have captured complex patterns present in the training data more effectively than the previous models, potentially leading to overfitting.

0.5 ### Gradient Boosting with feature engineering

[14]:
```
from sklearn import tree

# Defining the steps of the pipeline
steps = [
    ("deep_tree", tree.DecisionTreeRegressor(max_depth = 10, random_state =␣
↪prng))
]

# Creating the pipeline object
pipe_tree_deep = Pipeline(steps)

# Fitting the pipeline to the training data
pipe_tree_deep.fit(X_train_fe, y_train_fe)

# Calculating the errors
train_error_gradient = calculateRMSLE(pipe_tree_deep.predict(X_train_fe),␣
↪y_train_fe)
test_error_gradient = calculateRMSLE(pipe_tree_deep.predict(X_test_fe),␣
↪y_test_fe)

# Preparing the model's results
```

```python
result_gradient = pd.DataFrame([["Gradient Boosting with FE",␣
↪train_error_gradient, test_error_gradient]],
                                columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
results_df = pd.concat([results_df, result_gradient], ignore_index=True)

# Displaying the results
results_df
```

[14]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Linear Regression	0.2250	0.2305
2	Multivariate Regression	0.1993	0.2317
3	Multivariate Regression with FE	0.1532	0.3236
4	Random Forest with FE	0.0682	0.2411
5	Gradient Boosting with FE	0.0195	0.2138

From the RMSLE scores, it can be concluded that the feature engineered Gradient Boosting model is experiencing overfitting. Both scores have shown improvement compared to the previous models. The training set shows a relatively low RMSLE (0.0195), and the RMSLE score for the test is (0.2138). This indicates that the model is learning the training data too closely, leading to a lack of generalization ability when applied to unseen data.

0.5.1 Would you launch your web app now? What options you might have to further improve the prediction performance? (2 points)

Given the current RMSLE scores, I would advise not to launch the web app now as Gradient Boosting - the model with the lowest test RMSLE - is overfitting. There are several ways to improve the model, such as additional steps in feature engineering, perhaps making sense of and include `transaction_date` in the features. Moreover, we could introduce hyperparameter tuning or regularization parameters, such as lambda. We could also introduce other models, such as decision trees and feature engineered decision trees. We could also conduct a more precise model evaluation with cross-validation, which could provide more insights into the models' robustness and generalization ability.

0.5.2 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. (Hint: extend the code snippet below.) Did it improve the predictive power of your models? Where do you observe the biggest improvement? Would you launch your web app now? (4 points)

[15]:
```python
# Defining the full real estate training set
real_estate_full = real_estate_data.loc[~real_estate_data.index.isin(X_test.
↪index)].copy()


# checking the size of the full training set
```

```python
print(f"Size of the full training set: {real_estate_full.shape}")
```

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

[16]:
```python
# Redefining the training set features
X_train_full = real_estate_full[["house_age",
                                 "distance_to_the_nearest_MRT_station",
                                 "number_of_convenience_stores",
                                 ]]

# Redefining the target variable
y_train_full = real_estate_full["house_price_of_unit_area"]
```

0.6 ### Linear Regression with full training set

[17]:
```python
# Initializing the linear regression model
lin_reg_full = LinearRegression().
 ↪fit(X_train_full[["distance_to_the_nearest_MRT_station"]], y_train_full)

# Predictions on training and testing sets
train_predictions_full = lin_reg_full.
 ↪predict(X_train_full[["distance_to_the_nearest_MRT_station"]])
test_predictions_full = lin_reg_full.
 ↪predict(X_test[["distance_to_the_nearest_MRT_station"]])

# Calculating RMSLE for the training and testing sets
model_train_rmsle_full = calculateRMSLE(train_predictions_full, y_train_full)
model_test_rmsle_full = calculateRMSLE(test_predictions_full, y_test)

# Preparing the model's results
model_result_full = pd.DataFrame([["Linear Regression (Full)",␣
 ↪model_train_rmsle_full, model_test_rmsle_full]],
                            columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
results_df = pd.concat([results_df, model_result_full], ignore_index=True)

# Displaying the updated results
results_df
```

[17]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Linear Regression	0.2250	0.2305
2	Multivariate Regression	0.1993	0.2317
3	Multivariate Regression with FE	0.1532	0.3236
4	Random Forest with FE	0.0682	0.2411

```
5        Gradient Boosting with FE  0.0195  0.2138
6         Linear Regression (Full)  0.3477  0.2211
```

The linear regression with the full data slighlty different results to the original linear model taking all of the previous models into consideration. The training set RMSLE (0.3477) is slightly higher, and the test RMSLE (0.2211) is lower compared to the original linear regression. This means that the full data model might not fit the training data as well as the original model, but it can make accurate predictions on unseen data.

0.7 ### RandomForest with full sample

Because I trained RandomForest and Gradient Boosting models with feature engineering before, I needed to make sure that the variables I engineered for those models are defined again. This ensures that the results remain comparable.

[18]:
```
# Feature engineering with the full dataset
real_estate_full["house_age_sq"] = real_estate_full["house_age"] ** 2
real_estate_full["distance_to_the_nearest_MRT_station_sq"] =␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station"] ** 2
real_estate_full["number_of_convenience_stores_sq"] =␣
 ↪real_estate_full["number_of_convenience_stores"] ** 2

# Creating the interactions
real_estate_full["house_age_distance_interaction"] =␣
 ↪real_estate_full["house_age"] *␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station"]
real_estate_full["house_age_stores_interaction"] =␣
 ↪real_estate_full["house_age"] *␣
 ↪real_estate_full["number_of_convenience_stores"]
real_estate_full["distance_stores_interaction"] =␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station"] *␣
 ↪real_estate_full["number_of_convenience_stores"]

# Creating the interactions between the squared variables
real_estate_full["house_age_distance_interaction_sq"] =␣
 ↪real_estate_full["house_age"] *␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station"] *␣
 ↪real_estate_full["house_age_sq"] *␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station_sq"]
real_estate_full["house_age_stores_interaction_sq"] =␣
 ↪real_estate_full["house_age"] *␣
 ↪real_estate_full["number_of_convenience_stores"] *␣
 ↪real_estate_full["house_age_sq"] *␣
 ↪real_estate_full["number_of_convenience_stores_sq"]
```

```
real_estate_full["distance_stores_interaction_sq"] =␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station"] *␣
 ↪real_estate_full["number_of_convenience_stores"] *␣
 ↪real_estate_full["distance_to_the_nearest_MRT_station_sq"] *␣
 ↪real_estate_full["number_of_convenience_stores_sq"]

# Including the distance variable
real_estate_full['distance_to_city_center'] =␣
 ↪euclidean_distance(real_estate_full['latitude'],␣
 ↪real_estate_full['longitude'], city_center[0], city_center[1])

# Defining the feature engineered features again
features_fe = real_estate_full[[
    "house_age_sq",
    "distance_to_the_nearest_MRT_station_sq",
    "number_of_convenience_stores_sq",
    "house_age_distance_interaction",
    "house_age_stores_interaction",
    "distance_stores_interaction",
    "house_age_distance_interaction_sq",
    "house_age_stores_interaction_sq",
    "distance_stores_interaction_sq",
    "distance_to_city_center"
]]
```

```
[19]: # Defining the steps of the pipeline
steps = [
    ("random_forest", RandomForestRegressor(random_state = prng))
]

# Creating the pipeline object
pipe_rf_full = Pipeline(steps)

# Fitting the pipeline to the training data
pipe_rf_full.fit(X_train_full, y_train_full)

# Calculating the errors
train_error_rf_full = calculateRMSLE(pipe_rf_full.predict(X_train_full),␣
 ↪y_train_full)
test_error_rf_full = calculateRMSLE(pipe_rf_full.predict(X_test), y_test)

# Preparing the model's results
result_rf_full = pd.DataFrame([["Random Forest (Full)", train_error_rf_full,␣
 ↪test_error_rf_full]],
                            columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
```

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

# Displaying the results
results_df
```

[19]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Linear Regression	0.2250	0.2305
2	Multivariate Regression	0.1993	0.2317
3	Multivariate Regression with FE	0.1532	0.3236
4	Random Forest with FE	0.0682	0.2411
5	Gradient Boosting with FE	0.0195	0.2138
6	Linear Regression (Full)	0.3477	0.2211
7	Random Forest (Full)	0.0782	0.1385

RandomForest with the full sample performs better in terms of the test set. While the training set RMSLE (0.0782) is close to the original RandomForest's RMSLE, the test set is almost half of the original RandomForest's test RMSLE score (0.1385). This suggests that RF with the full training set effectively generalized from the training data to the test data, imporving its predictive performance to the original RandomForest, but the new model is still overfitting.

0.8 ### Gradient Boosting with full sample

[20]:
```
# Defining the steps of the pipeline
steps = [
    ("deep_tree", tree.DecisionTreeRegressor(max_depth = 10, random_state =␣
 ↪prng))
]

# Creating the pipeline object
pipe_tree_deep_full = Pipeline(steps)

# Fitting the pipeline to the training data
pipe_tree_deep_full.fit(X_train_full, y_train_full)

# Calculating the errors
train_error_gradient_full = calculateRMSLE(pipe_tree_deep_full.
 ↪predict(X_train_full), y_train_full)
test_error_gradient_full = calculateRMSLE(pipe_tree_deep_full.predict(X_test),␣
 ↪y_test)

# Preparing the model's results
result_gradient_full = pd.DataFrame([["Gradient Boosting (Full)",␣
 ↪train_error_gradient_full, test_error_gradient_full]],
                                  columns=["Model", "Train", "Test"])

# Appending model_result to the existing results_df DataFrame
```

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

# Displaying the results
results_df
```

[20]:

	Model	Train	Test
0	Benchmark	0.3434	0.3221
1	Linear Regression	0.2250	0.2305
2	Multivariate Regression	0.1993	0.2317
3	Multivariate Regression with FE	0.1532	0.3236
4	Random Forest with FE	0.0682	0.2411
5	Gradient Boosting with FE	0.0195	0.2138
6	Linear Regression (Full)	0.3477	0.2211
7	Random Forest (Full)	0.0782	0.1385
8	Gradient Boosting (Full)	0.0736	0.1628

The Gradient Boosting model with the full data displays similar training set RMSLE value (0.0736). However, the RMSLE score for the test has improved significantly compared to the original GradientBoosting model (0.1628). Regardless, as the training RMSLE score is lower than the error score for the test set, the Gradient Boosting model - the model with the lowest test RMSLE score - is still overfitting.

0.9 ### Conclusion

The app could be launched, but there would be some risk. Even the best performing model, Gradient Boosting on the full training sample, is still overfitting. Incorrect predictions could lead to wrong decisions, and could result in mispricing of the properties in New Taipei City. As the full training sample performed better compared to the models with the 20% subsample, we could assume that more data could improve our predictive models. Despite further data collection being time-consuming and expensive, I believe launching the app could result in bigger financial losses.