Introduction

The CSV file contains real estate data with the following columns:

- `id` : A unique identifier for each record.
- `transaction_date` : The date of the property transaction, represented as a year and fraction of the year.
- `house_age` : The age of the house at the time of the transaction, in years.
- `distance_to_the_nearest_MRT_station` : The distance to the nearest Mass Rapid Transit station, in meters.
- `number_of_convenience_stores` : The number of convenience stores within walking distance.
- `latitude` : The geographical latitude of the property.
- `longitude` : The geographical longitude of the property.
- `house_price_of_unit_area` : The price of the house per unit area, which is our target variable for prediction.

Import libraries

In [211…

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import PolynomialFeatures, MinMaxScaler
from sklearn.linear_model import Lasso
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
```

In [212…

```python
# set the seed
prng = np.random.RandomState(20240319)
real_estate_data = pd.read_csv("https://raw.githubusercontent.com/divenyijanos/ceu-ml/2023/data/real_estate/real_estate.csv", inde
real_estate_sample = real_estate_data.sample(frac=0.2, random_state=prng)
# outcome
outcome = real_estate_sample["house_price_of_unit_area"]
# features
features = real_estate_sample.drop(columns=["house_price_of_unit_area"])
X_train, X_test, y_train, y_test = train_test_split(features, outcome, test_size=0.3, random_state=prng)
# print the size of the training and test set
print(f"Size of the training set: {len(X_train)}, size of the test set: {len(X_test)}")
```

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

In [213…

```python
# check the sample
real_estate_sample
```

Out[213]:

	transaction_date	house_age	distance_to_the_nearest_MRT_station	number_of_convenience_stores	latitude	longitude	house_price_of_unit_area
id							
213	2013.333	17.6	1805.66500	2	24.98672	121.52091	31.1
16	2013.583	35.7	579.20830	2	24.98240	121.54619	50.5
127	2013.083	38.6	804.68970	4	24.97838	121.53477	62.9
257	2012.667	14.6	339.22890	1	24.97519	121.53151	26.5
239	2013.083	12.8	732.85280	0	24.97668	121.52518	40.6
...	...	...	...	...	...	...	...
100	2013.417	6.4	90.45606	9	24.97433	121.54310	62.2
299	2013.333	16.7	4082.01500	0	24.94155	121.50381	16.7
156	2013.167	13.8	4082.01500	0	24.94155	121.50381	15.6
59	2013.500	30.3	4510.35900	1	24.94925	121.49542	22.6
256	2013.417	31.5	5512.03800	1	24.95095	121.48458	17.4

83 rows × 7 columns

Question 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?

Answer 1

We choose the Root Mean Squared Error (RMSE) as our loss function for predicting property prices due to its practical advantage. RMSE is in the same units as the target variable, making it straightforward to interpret.

Incorrect predictions in property pricing can lead to significant business consequences. Overestimating prices may result in prolonged listing periods without sales, undermining the credibility of the platform and leading to increased holding costs for sellers. On the other hand, underestimating property values could facilitate rapid sales, but at the expense of potential earnings, leaving sellers unhappy and potentially decreasing their trust in the platform's ability to provide accurate market valuations.

In [214…

```
# define a function to calculate the RMSE
def rmse(y, y_hat):
    return np.sqrt(mean_squared_error(y, y_hat))
```

Question 2

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

In [215…

```
# build the benchmark model
benchmark = np.mean(y_train)

# evaluate the benchmark model on the test and train sets using root mean squared error
rmse_benchmark = rmse(y_test, [benchmark]*len(y_test))
rmse_benchmark_train = rmse(y_train, [benchmark]*len(y_train))

# put everything into a dataframe
results_df = pd.DataFrame({'Model': ['Benchmark'],'Train': [rmse_benchmark_train], 'Test': [rmse_benchmark]})
results_df
```

Out[215]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567

The benchmark model, which predicts the average price of the training set for all instances, is the simplest form of model we could use for this regression task. Its performance, evaluated using the RMSE, results in scores of approximately 12.28 on the training set and 12.03 on the test set. These results serve as a baseline for the complexity and accuracy of more sophisticated models; any advanced model should aim to surpass this basic benchmark to prove its efficiency in predicting real estate prices more accurately.

Question 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 [216…

```
# build a simple linear regression model
from sklearn.linear_model import LinearRegression
# create a linear regression model
model = LinearRegression()
# fit the model
model.fit(X_train[['distance_to_the_nearest_MRT_station']], y_train)

# evaluate the model on the test and train sets using root mean squared error
rmse_simple = rmse(y_test, model.predict(X_test[['distance_to_the_nearest_MRT_station']]))
rmse_simple_train = rmse(y_train, model.predict(X_train[['distance_to_the_nearest_MRT_station']]))

# put everything into the results dataframe
results_df = pd.concat([results_df, pd.DataFrame({'Model': ['Simple linear regression'],'Train': [rmse_simple_train], 'Test': [rms
                       ignore_index=True)
results_df
```

Out[216]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198

The simple linear regression model, using 'distance to the nearest MRT station' as the predictor, shows an improvement in performance over the benchmark model, with RMSE scores of approximately 8.38 on the training set and 6.87 on the test set. This improvement suggests that the distance to the nearest MRT station is a significant factor in predicting real estate prices. However, while the model performs better than the benchmark, it is still quite simplistic and may not capture all the complexities and variables influencing property prices. Therefore, before launching the evaluator web app using this model, it would be advisable to explore more sophisticated models and include additional features to ensure a more accurate and reliable prediction system.

Question 4

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

In [217…

```
# build a multiple linear regression model
model2 = LinearRegression()
# fit the model
model2.fit(X_train, y_train)

# evaluate the model on the test and train sets using root mean squared error
rmse_multiple = rmse(y_test, model2.predict(X_test))
rmse_multiple_train = rmse(y_train, model2.predict(X_train))

# put everything into the results dataframe
results_df = pd.concat([results_df, pd.DataFrame({'Model': ['Multiple linear regression'],'Train': [rmse_multiple_train], 'Test':
                       ignore_index=True)
results_df
```

Out[217]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198
2	Multiple linear regression	7.535398	6.241648

The multiple linear regression model, which incorporates all available meaningful variables, shows a further improvement in performance compared to both the benchmark and the simple linear regression models. The RMSE scores for the multiple regression are approximately 7.54 on the training set and 6.24 on the test set. This reduction in RMSE compared to the simple linear regression model indicates that including more variables helps capture a broader range of factors influencing property prices, thus improving the model's predictive power.

Question 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 [218…

```
# featue engineering
real_estate_data['year'] = real_estate_data['transaction_date'].apply(lambda x: int(x))
real_estate_data['month'] = (real_estate_data['transaction_date'] % 1) * 12
real_estate_data['age_distance_interaction'] = real_estate_data['house_age'] * real_estate_data['distance_to_the_nearest_MRT_stati
real_estate_data['house_age_squared'] = real_estate_data['house_age'] ** 2
real_estate_data['distance_squared'] = real_estate_data['distance_to_the_nearest_MRT_station'] ** 2
# central point of the city of New Taipei
central_lat, central_long = 25.013, 121.537
# calculate the distance from the center
real_estate_data['distance_from_center'] = np.sqrt((real_estate_data['latitude'] - central_lat) ** 2 + (real_estate_data['longitud
# more features
real_estate_data['log_distance_to_MRT'] = np.log1p(real_estate_data['distance_to_the_nearest_MRT_station'])
real_estate_data['log_age'] = np.log1p(real_estate_data['house_age'])  # Logarithm of house age
real_estate_data['number_of_convenience_stores_squared'] = real_estate_data['number_of_convenience_stores'] ** 2
real_estate_data['log_distance_from_center'] = np.log1p(real_estate_data['distance_from_center'])

# get the sample data
real_estate_sample = real_estate_data.sample(frac=0.2, random_state=prng)
# outcome
outcome = real_estate_sample["house_price_of_unit_area"]
# features
features = real_estate_sample.drop(columns=["house_price_of_unit_area"])
# split the data
X_train, X_test, y_train, y_test = train_test_split(features, outcome, test_size=0.3, random_state=prng)
# print the size of the training and test set
print(f"Size of the training set: {len(X_train)}, size of the test set: {len(X_test)}")
```

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

In [219…

```
# build model 3 with the new features
model3 = LinearRegression()
# fit the model
model3.fit(X_train, y_train)
```

```
# evaluate the model on the test and train sets using root mean squared error
rmse_multiple_feature = rmse(y_test, model3.predict(X_test))
rmse_multiple_feature_train = rmse(y_train, model3.predict(X_train))

# put everything into the results dataframe
results_df = pd.concat([results_df, pd.DataFrame({'Model': ['Multiple linear regression with new features'],'Train': [rmse_multipl
                       ignore_index=True)
results_df
```

Out[219]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198
2	Multiple linear regression	7.535398	6.241648
3	Multiple linear regression with new features	5.618501	7.580872

In [220…

```
# build model 4 with LASSO
# search for the best alpha
alphas = np.linspace(0.01, 5, 100)

# create a list to store the RMSEs
rmse_train = []
rmse_test = []

# loop through the alphas
for alpha in alphas:
    # create a LASSO model
    model4 = Lasso(alpha=alpha, max_iter=100000)
    # fit the model
    model4.fit(X_train, y_train)
    # evaluate the model on the test and train sets using root mean squared error
    rmse_train.append(rmse(y_train, model4.predict(X_train)))
    rmse_test.append(rmse(y_test, model4.predict(X_test)))

# find the best alpha
best_alpha = alphas[np.argmin(rmse_test)]
print(f"The best alpha is: {best_alpha}")

# create a LASSO model
model4 = Lasso(alpha=best_alpha, max_iter=100000)
# fit the model
model4.fit(X_train, y_train)
# evaluate the model on the test and train sets using root mean squared error
rmse_lasso = rmse(y_test, model4.predict(X_test))
rmse_lasso_train = rmse(y_train, model4.predict(X_train))

# put everything into the results dataframe
results_df = pd.concat([results_df, pd.DataFrame({'Model': ['LASSO'],'Train': [rmse_lasso_train], 'Test': [rmse_lasso]})],
                       ignore_index=True)
results_df
```

```
The best alpha is: 0.01
```

Out[220]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198
2	Multiple linear regression	7.535398	6.241648
3	Multiple linear regression with new features	5.618501	7.580872
4	LASSO	6.096429	7.842076

The introduction of additional features in the third model, "Multiple linear regression with new features," significantly improved the training RMSE to 5.62, but slightly worsened the test RMSE to 7.58 compared to the standard multiple linear regression, indicating potential overfitting. We tried to get a better RMSE with LASSO but the results indicate that model #2 is still the best.

Train more flexible models

In [221…

```
# build model 5 with random forest

# Define the pipeline steps
pipeline_steps = [
    ('scaler', StandardScaler()),
    ('rf', RandomForestRegressor(random_state=prng))
]

# Create the pipeline
rf_pipeline = Pipeline(pipeline_steps)
```

```python
# Define the parameter grid
param_grid = {
    'rf__n_estimators': np.linspace(10, 400, 5).astype(int),
    'rf__max_features': [0.05, 0.2, 0.25, 0.5, 1.0],
    'rf__max_depth': np.linspace(5, 50, 5).astype(int)
}

# Create the grid search object
grid_search = GridSearchCV(estimator=rf_pipeline, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)

# Fit the grid search
grid_search.fit(X_train, y_train)

# Best parameters
best_params_rf = grid_search.best_params_
print(f"Best parameters: {best_params_rf}")

# Update the pipeline with the best parameters
rf_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestRegressor(
        n_estimators=best_params_rf['rf__n_estimators'],
        max_features=best_params_rf['rf__max_features'],
        max_depth=best_params_rf['rf__max_depth'],
        random_state=prng
    ))
])

# Fit the final model
rf_pipeline.fit(X_train, y_train)

# calculate the RMSE
rmse_rf_train = rmse(y_train, rf_pipeline.predict(X_train))
rmse_rf_test = rmse(y_test, rf_pipeline.predict(X_test))

# put everything into the results dataframe
results_df = pd.concat([results_df, pd.DataFrame({'Model': ['Random Forest'],'Train': [rmse_rf_train], 'Test': [rmse_rf_test]})],
                        ignore_index=True)
results_df
```

```
Best parameters: {'rf__max_depth': 5, 'rf__max_features': 1.0, 'rf__n_estimators': 10}
```

Out[221]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198
2	Multiple linear regression	7.535398	6.241648
3	Multiple linear regression with new features	5.618501	7.580872
4	LASSO	6.096429	7.842076
5	Random Forest	3.366758	8.583653

In [222…

```python
# model 6 gradient boosting
from sklearn.ensemble import GradientBoostingRegressor

# Define the pipeline steps
pipeline_steps = [
    ('scaler', StandardScaler()),
    ('gb', GradientBoostingRegressor(random_state=prng))
]

# Create the pipeline
gb_pipeline = Pipeline(pipeline_steps)

# Define the parameter grid
param_grid = {
    'gb__n_estimators': np.linspace(10, 400, 5).astype(int),
    'gb__max_features': [0.05, 0.2, 0.25, 0.5, 1.0],
    'gb__max_depth': np.linspace(5, 50, 5).astype(int)
}

# Create the grid search object
grid_search = GridSearchCV(estimator=gb_pipeline, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)

# Fit the grid search
grid_search.fit(X_train, y_train)

# Best parameters
best_params_gb = grid_search.best_params_
print(f"Best parameters: {best_params_gb}")

# Update the pipeline with the best parameters
gb_pipeline = Pipeline([
```

```
    ('scaler', StandardScaler()),
    ('gb', GradientBoostingRegressor(
        n_estimators=best_params_gb['gb__n_estimators'],
        max_features=best_params_gb['gb__max_features'],
        max_depth=best_params_gb['gb__max_depth'],
        random_state=prng
    ))
])

# Fit the final model
gb_pipeline.fit(X_train, y_train)

# calculate the RMSE
rmse_gb_train = rmse(y_train, gb_pipeline.predict(X_train))
rmse_gb_test = rmse(y_test, gb_pipeline.predict(X_test))

# put everything into the results dataframe
results_df = pd.concat([results_df, pd.DataFrame({'Model': ['Gradient Boosting'],'Train': [rmse_gb_train], 'Test': [rmse_gb_test]}
                          ignore_index=True)
results_df
```

```
Best parameters: {'gb__max_depth': 5, 'gb__max_features': 0.05, 'gb__n_estimators': 107}
```

Out[222]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198
2	Multiple linear regression	7.535398	6.241648
3	Multiple linear regression with new features	5.618501	7.580872
4	LASSO	6.096429	7.842076
5	Random Forest	3.366758	8.583653
6	Gradient Boosting	0.044068	6.556802

The Random Forest model shows exceptional training performance with an RMSE of 3.37, yet experiences a substantial increase in the test RMSE to 8.58, indicating potential overfitting. On the other hand, the Gradient Boosting model demonstrates an outstandingly low training RMSE of 0.044, suggesting an almost perfect fit, but its test RMSE of 6.56, while significantly better than Random Forest's, hints at overfitting as well, maybe to a lesser degree. Among all models at this point, the Multiple linear regression indicates the lowest RMSE on the test data.

Question 6

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

Answer 6

Considering the results, launching the web app could be considered using the Gradient Boosting model, which shows the best balance between training and test performance among the developed models. However, there are signs of overfitting, so we should be cautious and maybe go with the model #2(Multiple linear regression). The model is also better suited for interpretation.

To further improve prediction performance and before finalizing the launch, we can consider the following options: 1) Implement additional feature engineering techniques and explore alternative feature combinations to capture more relevant patterns without increasing complexity unnecessarily. 2) Use higher limits for parameter values when finding the best parameters for complex models. 3) Collect and use more data, to enhance the model's learning and its generalization capability.

Question 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. (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?

In [223…

```
# Preparing the full training set
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}")

X_full_train = real_estate_full.drop(columns=["house_price_of_unit_area"])
y_full_train = real_estate_full["house_price_of_unit_area"]

# Multiple Linear Regression with new features on full training set
model_full = LinearRegression()
model_full.fit(X_full_train, y_full_train)
rmse_full = rmse(y_test, model_full.predict(X_test))
rmse_full_train = rmse(y_full_train, model_full.predict(X_full_train))

# Random Forest with full training set
```

```python
rf_pipeline_full = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestRegressor(
        n_estimators=best_params_rf['rf__n_estimators'],
        max_features=best_params_rf['rf__max_features'],
        max_depth=best_params_rf['rf__max_depth'],
        random_state=prng
    ))
])
rf_pipeline_full.fit(X_full_train, y_full_train)
rmse_rf_full_train = rmse(y_full_train, rf_pipeline_full.predict(X_full_train))
rmse_rf_full_test = rmse(y_test, rf_pipeline_full.predict(X_test))

# Gradient Boosting with full training set
gb_pipeline_full = Pipeline([
    ('scaler', StandardScaler()),
    ('gb', GradientBoostingRegressor(
        n_estimators=best_params_gb['gb__n_estimators'],
        max_features=best_params_gb['gb__max_features'],
        max_depth=best_params_gb['gb__max_depth'],
        random_state=prng
    ))
])
gb_pipeline_full.fit(X_full_train, y_full_train)
rmse_gb_full_train = rmse(y_full_train, gb_pipeline_full.predict(X_full_train))
rmse_gb_full_test = rmse(y_test, gb_pipeline_full.predict(X_test))

# Updating the results DataFrame
new_results = pd.DataFrame({
    'Model': [
        'Multiple linear regression with new features (full)',
        'Random Forest (full)',
        'Gradient Boosting (full)'
    ],
    'Train': [rmse_full_train, rmse_rf_full_train, rmse_gb_full_train],
    'Test': [rmse_full, rmse_rf_full_test, rmse_gb_full_test]
})
results_df = pd.concat([results_df, new_results], ignore_index=True)
results_df
```

```
Size of the full training set: (389, 17)
```

Out[223]:

	Model	Train	Test
0	Benchmark	12.283463	12.031567
1	Simple linear regression	8.381904	6.866198
2	Multiple linear regression	7.535398	6.241648
3	Multiple linear regression with new features	5.618501	7.580872
4	LASSO	6.096429	7.842076
5	Random Forest	3.366758	8.583653
6	Gradient Boosting	0.044068	6.556802
7	Multiple linear regression with new features (...	7.651385	6.940852
8	Random Forest (full)	5.059651	5.908197
9	Gradient Boosting (full)	1.943382	6.379522

After retraining, the Random Forest model demonstrates notable improvement with significant reduction in test RMSE (from 8.6 to 5.9). That indicates enhanced generalization when trained on the full dataset. This suggests a substantial increase in predictive accuracy and a better balance between learning from the training data and generalizing to unseen data.

On the other hand, while the Multiple Linear Regression and Gradient Boosting models show changes in RMSE, neither achieves the same results as Random Forest. Therefore, we can consider to launch the web application using Random Forest model.