Data Science 1 - Home Assignment 3

Author: Márton Nagy

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

prng = np.random.RandomState(20250317)

url_data_on_github = 'https://raw.githubusercontent.com/divenyijanos/ceu-ml/refs/heads/2025/data/real_estate/real_estate.csv'
real_estate_data = pd.read_csv(url_data_on_github)
real_estate_sample = real_estate_data.sample(frac=0.2, random_state=prng)

outcome = real_estate_sample['house_price_of_unit_area']
features = real_estate_sample.drop('house_price_of_unit_area', axis=1)

X_train, X_test, y_train, y_test = train_test_split(features, outcome, test_size=0.3, random_state=prng)

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
```

Task 1

Description: . 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 [2]:
```
def calculateWeightedMAE(y_true, y_pred):
    weight = 1/4
    errors = y_pred - y_true
    # for positive errors, the weight is one-quarter, for negative errors, the weight is three-quarters
    # thus negative errors are three times as important as positive errors
```

```
    loss = np.where(errors > 0, weight * np.abs(errors), (1 - weight) * np.abs(errors))
    return np.mean(loss)
```

Answer: As per the business description in the assignment, both over and underpredictions have a risk for the business. If the price is underpredicted, sellers will list their homes for lower prices than the true value, thus losing out on some profit. If the price is overpredicted, sellers may not be able to find any buyers - but they can react to this issue by lowering the listed price, and ultimately finding a buyer at the fair market price. Thus, I believe underpredictions are more risky for the business (as they cannot be reacted to). Also, the risk grows linearly as a function of the error in both directions (as the cost is equal to the error). Therefore, I believe the most appropriate loss function is a weighted mean absolute error, with higher weights for negative errors.

Task 2

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

In [3]:

```
benchmark = y_train.mean()

class ResultCollector:
    def __init__(self):
        self.results = {}

    def add_model(self, name, train_error, test_error):
        """Add or update a model's results."""
        self.results[name] = {
            'Train WMAE': train_error,
            'Test WMAE': test_error
        }
        return self.get_table()

    def get_table(self, style=True):
        """Get the results table with optional styling."""
        df = pd.DataFrame(self.results).T
        if style:
            return df.style.format("{:.5f}").background_gradient(cmap='RdYlGn_r', axis=None)
        return df

results = ResultCollector()
```

```
results.add_model('Benchmark', calculateWeightedMAE(y_train, np.full(y_train.shape, benchmark)),
                  calculateWeightedMAE(y_test, np.full(y_test.shape, benchmark)))
```

Out[3]:

	Train WMAE	Test WMAE
Benchmark	5.25606	4.31097

Answer: My simple benchmark model is just the mean of the train set. These results are not informative on their own (and there is no direct way to interpret them neither), but they will be a good baseline to compare the later models to. If we manage to achieve smaller WMAE values than the simple mean, than our models may have some business value to them. Note that the WMAE is smaller on the test set than on the training data, meaning that our model fits the test set better than what it was trained on - but this may be just by chance bacause of the random way we have constructed the sets.

Task 3

Description: 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 [4]:

```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

simple_ols_pipe = Pipeline([
    ('select_cols', ColumnTransformer([('keep', 'passthrough', ['house_age'])])),
    ('regressor', LinearRegression())
])

simple_ols_pipe.fit(X_train, y_train)

results.add_model('Simple OLS', calculateWeightedMAE(y_train, simple_ols_pipe.predict(X_train)),
                  calculateWeightedMAE(y_test, simple_ols_pipe.predict(X_test)))
results.get_table()
```

Out[4]:

	Train WMAE	Test WMAE
Benchmark	5.25606	4.31097
Simple OLS	5.09657	4.57101

Answer: I chose to train a model using the age of the house, as I thought this may be an important feature in determining the price. Interestingly though, even if this model performs better on the train set than the benchmark, it fits the test set worse, so there is not much added value in this feature if we want to do good predictions - thus I would not launch my app using this simple model. In addition, the train set WMAE is still higher than the test set, so there is a huge room for improvement.

Task 4

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

In [5]:
```
multi_ols_pipe = Pipeline([
    ('select_cols', ColumnTransformer([('keep', 'passthrough',
                                        ['house_age', 'distance_to_the_nearest_MRT_station',
                                         'number_of_convenience_stores', 'latitude', 'longitude'])])),
    ('regressor', LinearRegression())
])

multi_ols_pipe.fit(X_train, y_train)

results.add_model('Multivariate OLS',
                  calculateWeightedMAE(y_train, multi_ols_pipe.predict(X_train)),
                  calculateWeightedMAE(y_test, multi_ols_pipe.predict(X_test)))
results.get_table()
```

Out[5]:

	Train WMAE	Test WMAE
Benchmark	5.25606	4.31097
Simple OLS	5.09657	4.57101
Multivariate OLS	3.23100	2.98373

Answer: Including all possible features in the dataset (without any feature engineering) improved the model's performance significantly. The improvement is present for both the training and the test set, indicating that the model has actually learned some new patterns. Also, the train and test set WMAE metrics are now closer to each other, which indicates that there is less room for improvement from this model (but there is still some, so we are not yet ready for deployment).

Task 5

Description: Try to make your model (even) better using the following approaches:

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

In [6]:
```
from sklearn.preprocessing import StandardScaler, PolynomialFeatures, OneHotEncoder
from sklearn.feature_selection import VarianceThreshold
from sklearn.linear_model import LassoCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import make_scorer
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBRegressor
from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')
```

In [7]:
```
# Function to calculate distance to city center
def haversine(lat1, lon1, lat2=25.0330, lon2=121.5654):
    """
    Calculate the great-circle distance between two points (lat1, lon1) and (lat2, lon2)
    using the Haversine formula.

    Parameters:
    - lat1, lon1: Coordinates of the given point.
    - lat2, lon2: Coordinates of New Taipei City center (default).

    Returns:
```

```
    - Distance in kilometers.
    """
    R = 6371
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat / 2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon / 2)**2
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))
    distance = R * c

    return distance

# Function to convert float year to datetime
def float_to_datetime(float_year):
    """
    Convert a float year (e.g., 2013.250) to a datetime object.

    Parameters:
    - float_year (float): Year in decimal format.

    Returns:
    - datetime: Corresponding datetime object.
    """
    year = int(float_year)
    remainder = float_year - year

    base_date = datetime(year, 1, 1)
    days_in_year = (datetime(year + 1, 1, 1) - base_date).days

    exact_date = base_date + timedelta(days=remainder * days_in_year)

    return exact_date

# Applying the functions to the dataset
X_train['distance_to_center'] = X_train.apply(lambda row: haversine(row['latitude'], row['longitude']), axis=1)
X_test['distance_to_center'] = X_test.apply(lambda row: haversine(row['latitude'], row['longitude']), axis=1)
X_train['transaction_date'] = X_train['transaction_date'].apply(float_to_datetime)
X_test['transaction_date'] = X_test['transaction_date'].apply(float_to_datetime)

X_train['year'] = X_train['transaction_date'].dt.year
X_test['year'] = X_test['transaction_date'].dt.year
```

```python
X_train['month'] = X_train['transaction_date'].dt.month
X_test['month'] = X_test['transaction_date'].dt.month

# Setting numerical and categorical features
# Latitude and longitude are still included, as they may hold other information as well
# than distance to the city center (e.g., neighborhood).
num_features = ['house_age', 'distance_to_the_nearest_MRT_station',
                'number_of_convenience_stores', 'distance_to_center', 'latitude', 'longitude']
cat_features = ['year', 'month']

# Feature engineered OLS
fe_ols_pipe = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', 'passthrough', num_features),
        ('cat', OneHotEncoder(drop='first'), cat_features)
    ])),
    ("2_degree_poly", PolynomialFeatures(degree=2, include_bias=False)),
    ("drop_zero_variance", VarianceThreshold()),
    ('regressor', LinearRegression())
])

fe_ols_pipe.fit(X_train, y_train)

results.add_model('Feature Engineered OLS', calculateWeightedMAE(y_train, fe_ols_pipe.predict(X_train)),
                  calculateWeightedMAE(y_test, fe_ols_pipe.predict(X_test)))

# Feature engineered LASSO
fe_lasso_pipe = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), num_features),
        ('cat', OneHotEncoder(drop='first'), cat_features)
    ])),
    ("2_degree_poly", PolynomialFeatures(degree=2, include_bias=False)),
    ("drop_zero_variance", VarianceThreshold()),
    ('regressor', LassoCV(random_state=prng, cv=5, max_iter=10000))
])

fe_lasso_pipe.fit(X_train, y_train)

results.add_model('Feature Engineered LASSO', calculateWeightedMAE(y_train, fe_lasso_pipe.predict(X_train)),
                  calculateWeightedMAE(y_test, fe_lasso_pipe.predict(X_test)))
```

```
# Basic Random Forest
rf_pipe = Pipeline([
    ('select_cols', ColumnTransformer([('keep', 'passthrough', num_features+cat_features)])),
    ('regressor', RandomForestRegressor(random_state=prng))
])

rf_pipe.fit(X_train, y_train)

results.add_model('Basic Random Forest', calculateWeightedMAE(y_train, rf_pipe.predict(X_train)),
                    calculateWeightedMAE(y_test, rf_pipe.predict(X_test)))

# Random Forest with CV
rf_cv = RandomizedSearchCV(
    estimator=rf_pipe,
    param_distributions={
        'regressor__n_estimators': [10, 50, 100, 200, 300],
        'regressor__max_depth': [5, 10, 15, 20, None],
        'regressor__min_samples_split': [2, 5, 10],
        'regressor__min_samples_leaf': [1, 2, 4]
    },
    n_iter=250,
    scoring=make_scorer(calculateWeightedMAE, greater_is_better=False),
    n_jobs=-1,
    cv=5,
    refit=True,
    random_state=prng
)

rf_cv.fit(X_train, y_train)

results.add_model('Random Forest with CV', calculateWeightedMAE(y_train, rf_cv.predict(X_train)),
                  calculateWeightedMAE(y_test, rf_cv.predict(X_test)))

# Casting categorical features to category type
X_train[cat_features] = X_train[cat_features].astype('category')
X_test[cat_features] = X_test[cat_features].astype('category')

# Basic XGBoost
xgb_pipe = Pipeline([
    ('select_cols', ColumnTransformer([('keep', 'passthrough', num_features+cat_features)])),
```

```python
    ('regressor', XGBRegressor(random_state=prng, enable_categorical=True))
])

xgb_pipe.fit(X_train, y_train)

results.add_model('Basic XGBoost', calculateWeightedMAE(y_train, xgb_pipe.predict(X_train)),
                  calculateWeightedMAE(y_test, xgb_pipe.predict(X_test)))
results.get_table()

# XGBoost with CV
cv_xgb = RandomizedSearchCV(
    estimator=xgb_pipe,
    param_distributions={
        'regressor__n_estimators': [10, 50, 100, 200, 300],
        'regressor__max_depth': [5, 10, 15, 20],
        'regressor__learning_rate': [0.01, 0.1, 0.3, 0.5],
        'regressor__subsample': [0.5, 0.75, 1],
        'regressor__colsample_bytree': [0.5, 0.75, 1],
        'regressor__reg_alpha': [0, 0.1, 0.5, 1],
        'regressor__reg_lambda': [0, 0.1, 0.5, 1]
    },
    n_iter=250,
    scoring=make_scorer(calculateWeightedMAE, greater_is_better=False),
    n_jobs=-1,
    cv=5,
    refit=True,
    random_state=prng
)

cv_xgb.fit(X_train, y_train)

results.add_model('XGBoost with CV', calculateWeightedMAE(y_train, cv_xgb.predict(X_train)),
                  calculateWeightedMAE(y_test, cv_xgb.predict(X_test)))
```

Out[7]:

	Train WMAE	Test WMAE
Benchmark	5.25606	4.31097
Simple OLS	5.09657	4.57101
Multivariate OLS	3.23100	2.98373
Feature Engineered OLS	0.00000	27.37603
Feature Engineered LASSO	2.18461	2.40623
Basic Random Forest	1.29234	2.03972
Random Forest with CV	1.45064	2.08621
Basic XGBoost	0.00040	2.35427
XGBoost with CV	1.25716	2.28868

Answer: I tried to improve my predictions using the following steps:

- adding the distance to the city center (but still including the coordinates, as they may contain other relevant information as well, e.g. the neighbourhood, or proximity to certain points of interest);
- adding dummy variables for the year and month of the transaction (as prices may fluctuate by time);
- adding squared features and interactions.

Having these, I trained a feature engineered OLS model, a feature engineered LASSO, a Random Forest and an XGBoost model (both with and without cross-validating the hyperparameters). It turns out that the feature engineered OLS model performs perfectly on the training data, but very bad on the test set, indicating a clear overfitting issue. The other models are more balanced in this sense. The best model (based on the test set WMAE) seems to be the Random Forest with deafault parameters, followed by the cross-validated XGBoost and LASSO. Thus, I will retrain the basic Random Forest and the CV XGBoost models on the full dataset, together with the less flexible multivariate OLS model.

Task 6

Description: 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 [8]:
```
real_estate_full = real_estate_data.loc[~real_estate_data.index.isin(X_test.index)]
X_full = real_estate_full.drop('house_price_of_unit_area', axis=1)
y_full = real_estate_full['house_price_of_unit_area']

X_full['distance_to_center'] = X_full.apply(lambda row: haversine(row['latitude'], row['longitude']), axis=1)
X_full['transaction_date'] = X_full['transaction_date'].apply(float_to_datetime)

X_full['year'] = X_full['transaction_date'].dt.year
X_full['month'] = X_full['transaction_date'].dt.month

print(f"Size of the full training set: {len(X_full)}")
```

```
Size of the full training set: 389
```

In [9]:
```
X_full[cat_features] = X_full[cat_features].astype(int)
X_test[cat_features] = X_test[cat_features].astype(int)

multi_ols_pipe.fit(X_full, y_full)

results.add_model('Multivariate OLS (large N)',
                  calculateWeightedMAE(y_full, multi_ols_pipe.predict(X_full)),
                  calculateWeightedMAE(y_test, multi_ols_pipe.predict(X_test)))

rf_pipe.fit(X_full, y_full)

results.add_model('Basic Random Forest (large N)', calculateWeightedMAE(y_full, rf_pipe.predict(X_full)),
                  calculateWeightedMAE(y_test, rf_pipe.predict(X_test)))

X_full[cat_features] = X_full[cat_features].astype('category')
X_test[cat_features] = X_test[cat_features].astype('category')

xgb_full = Pipeline([
    ('select_cols', ColumnTransformer([('keep', 'passthrough', num_features+cat_features)])),
    ('regressor', XGBRegressor(random_state=prng, enable_categorical=True,
                               **{k.replace("regressor__", ""): v for k, v in cv_xgb.best_params_.items()}))
]).fit(X_full, y_full)
```

```python
results.add_model('XGBoost with CV (large N)', calculateWeightedMAE(y_full, xgb_full.predict(X_full)),
                  calculateWeightedMAE(y_test, xgb_full.predict(X_test)))
```

Out[9]:

	Train WMAE	Test WMAE
Benchmark	5.25606	4.31097
Simple OLS	5.09657	4.57101
Multivariate OLS	3.23100	2.98373
Feature Engineered OLS	0.00000	27.37603
Feature Engineered LASSO	2.18461	2.40623
Basic Random Forest	1.29234	2.03972
Random Forest with CV	1.45064	2.08621
Basic XGBoost	0.00040	2.35427
XGBoost with CV	1.25716	2.28868
Multivariate OLS (large N)	3.11071	2.90679
Basic Random Forest (large N)	0.90373	2.06431
XGBoost with CV (large N)	1.19489	2.31514

Answer: The training set performance improved for all three models with the addition of extra observations. Interestingly, having more data only improved (slightly) the test-set performance for the multivariate OLS model, the Random Forest and XGBoost performance remained more or less the same. This may indicate that these flexible models could already learn the main patterns on the smaller dataset, and there were no additional patterns to learn in the extra observations. As the test set performance could not be improved further, I believe these models are close to the best we can achieve, thus they are ready to be used for launching the app. For this, I would opt for the Basic Random Forest model, as it still has the lowest WMAE value on the larger sample. Before deployment, I would of course retrain this model using all available data (that is, the full `real_estate_data`).