Workflow

This lecture tries to bring the previous lectures together by showing a complete workflow for a simple example. Specifically, we will exemplify the following topics in the case of the CorkB dataset:

- Data preparation;
- Error estimation (in the case where there are hyperparameters);
- Model selection; and
- Deployment.

One new thing to watch out for is the use of scikit-learn pipelines, which we will use in order to impute means and modes and to standardize data, without leakage.

Obviously, you start by reading in your data:

```python
In [29]: df = pd.read_csv("dataset-corkB.csv")
```

Data Preparation
Next, you should do such things as the following:

- You should get to know your data by: displaying the column names; displaying the shape; displaying the data types; running the pandas `describe` function; plotting graphs; and so on. We won't show the Python for that again here.
- If your dataset uses strings such as ‘?’, ‘N/A’, ‘UNK’, etc. to represent missing values, you might consider replacing these strings by the float NaN. This doesn't apply to the CorkB dataset.
- You should detect and remove anomalies — for the CorkB dataset, we will remove examples with anomalous-looking `flarea` values.
- You should consider what to do about examples that have features whose values are missing. We discussed the options previously. This might result in deleting examples, or even deleting whole features. We'll exemplify that here by deleting examples from the CorkB dataset whose price is missing.
- You should change nominal-valued features into numeric-values features, e.g. using binary encoding or one-hot encoding.
- If you want to scale values using min-max scaling, then now is the time to do it (assuming the min and max are provided by your domain expert). (But if you want to scale values using standardization, now is not the time to do it.) We will not do any min-max scaling on the CorkB dataset.

What you should not do at this stage is anything that involves computing values from the dataset as a whole, such as computing means or modes to impute missing values, or computing means and standard deviations to standardize values. Remember, doing these things on the whole dataset results in leakage.

```python
In [30]:
# Remove anomalies from flarea
   df = df[(df['flarea'].isnull()) | ((df['flarea'] > 10) & (df['flarea'] < 1000))]
   df.reset_index(drop=True, inplace=True)
# Remove examples with missing prices
   df.dropna(subset=['price'], inplace=True)
   df.reset_index(drop=True, inplace=True)
# Encode nominal-valued features
# - devment is easy because it is binary-valued
   df.replace({'devment': {'SecondHand': 0, 'New': 1}}, inplace=True)
# - for type we use one-hot encoding
   one_hot = pd.get_dummies(df['type'], 'type', '_')
   df.drop('type', axis=1, inplace=True)
   df = pd.concat([df, one_hot], axis=1)
# - similarly for ber
   one_hot = pd.get_dummies(df['ber'], 'ber', '_')
   df.drop('ber', axis=1, inplace=True)
   df = pd.concat([df, one_hot], axis=1)
# - and similarly for location
   one_hot = pd.get_dummies(df['location'], 'location', '_')
   df.drop('location', axis=1, inplace=True)
   df = pd.concat([df, one_hot], axis=1)
```
We must remember that we still have some missing values to deal with, and we would like to standardize the data too.

It turns out that there will be a problem when it comes to the missing values. scikit-learn's class for doing this is somewhat brute force. It will either replace NaNs by the mean or by the mode or by the median. What it will not allow us to do is to replace one feature's missing values by the mean, and another's by the mode. Yet, this is exactly what we would like to be able to do. E.g. we want to replace NaNs for `flarea` by the mean floor area; and we want to replace NaNs for `devment` by the mode. Why?

Furthermore, it seems that Imputer does not play nicely with scikit-learn pipelines. The input to a pipeline is checked to see whether it contains NaNs and, if it does, you get an error — even though the pipeline might contain an Imputer whose job it is to get rid of the NaNs.

My hack to make this work is, to replace NaNs as this stage by some other values. Where we want to use the mean, replace the NaNs by one value; where we want to use the mode, replace the NaNs by some other value. These values must be special values, different from all other values in the dataset. For `devment`, we will replace NaNs by -2 and then for all other NaNs, we will replace by -1. Why will this do what we want?

```python
In [31]: df['devment'].fillna(-2, inplace=True)
   df.fillna(-1, inplace=True)
```

Finally, extract the data:

```python
In [32]: X = df[['flarea', 'bdrms', 'bthrms', 'floors', 'devment',
             'type_Apartment', 'type_Detached', 'type_Semi-detached',
             'type_Terraced',
             'ber_B1', 'ber_B2', 'ber_B3', 'ber_C1', 'ber_C2', 'ber_C3',
             'ber_D1', 'ber_D2', 'ber_E1', 'ber_E2', 'ber_F', 'ber_G',
             'location_Ballinlough', 'location_Ballintemple', 'location_Ballyphehane',
             'location_Ballyvolane',
             'location_Banduff', 'location_Bishopstown', 'location_Blackpool',
             'location_Blackrock', 'location_Carrigrohane',
             'location_CityCentre', 'location_Cloghroe', 'location_Donnybrook',
             'location_Douglas', 'location_DublinPike',
             'location_Farranree', 'location_Fota', 'location_Glanmire',
             'location_Glasheen', 'location_Grange',
             'location_Gurranabraher', 'location_Inniscarra', 'location_Mayfield',
             'location_ModelFarmRoad',
             'location_Montenotte', 'location_Ovens', 'location_PassageWest',
             'location_Rochestown', 'location_Silversprings',
             'location_StLukes', 'location_SundaysWell', 'location_TheLough',
             'location_Togher', 'location_TurnersCross',
             'location_VictoriaCross', 'location_Waterfall', 'location_WesternRoad', 'location_Wilton']].values
   y = df['price'].values
```

**Estimators**
You should now decide which estimators you would like to compare. For the CorkB dataset, let's compare OLS linear regression, lasso regression, ridge regression, and distance-weighted kNN.

We create these estimators. As we know, these estimators will be trained on the training examples (using scikit-learn's fit method) and tested on the test examples (using scikit-learn's predict method).

But, as part of the training, we want to compute means and modes from the training examples only. Then we must impute these means and modes for the missing values in the training examples and in the test examples. Fortunately, this is what scikit-learn's Imputer class is designed to do. It has a fit method, which computes means or modes from the training data; then it has a transform method, which can be invoked either on the training examples or the test examples, to replace the missing values by the means or modes computed from the training examples.

scikit-learn's StandardScaler class is similar. It has a fit method, which you can use to compute means and standard deviations from the training examples. Then it has a transform method, which can be invoked either on the training examples or the test examples, in order to standardize the values using the means and standard deviations computed from the training examples.

We will use scikit-learn's Pipeline class, which allows us to stick together a sequence of transformations (such as an Imputer and a StandardScaler) plus, optionally, a final estimator.

When we invoke the Pipeline's fit method on the training examples, it runs the fit methods and transform methods of each transformation in the Pipeline, and it runs the fit method of the estimator in the Pipeline.

When we invoke the Pipeline's predict method on the test examples, it runs the transform methods of each transformation in the Pipeline, and it runs the predict method of the estimator in the Pipeline.
Questions:

- Suppose we call `ols.fit(X_train, y_train)` on some training data. How many methods will this invoke?
- Suppose we then call `ols.predict(X_test)` on some test data. How many methods will this invoke?
- Why are we doing OLS differently from Lasso and Ridge above, and why is kNN different again?

Error Estimation
Model Selection

We have a clear winner: Lasso. So now we must find the best value for its hyperparameter, $\lambda$:

In [35]:

```
lassocv.fit(X, y)

lassocv.get_params()]['estimator'].alpha_
```

Out[35]: 5.0750946535713855

If the winner had been kNN, then we would need to find the best value for $k$:

In [36]:

```
gs = GridSearchCV(knn, knn_hyperparameters, scoring = 'mean_squared_error', cv = 10)
gs.fit(X, y)
gs.best_params_
```

Out[36]: {'estimator__n_neighbors': 2}

What would you have done if OLS had been the best model?
Deployment

So now it is decided: we are going to use Lasso with $\lambda = 5.075$. It makes sense to build this system from all the data that we have available:

```python
In [37]: from sklearn.linear_model import Lasso

final_model = Pipeline([('impute_means', Imputer(missing_values='NaN', strategy='mean')),
                         ('impute_modes', Imputer(missing_values=-1, strategy='most_frequent')),
                         ('standardize', StandardScaler()),
                         ('estimator', Lasso(alpha = 5.075))])

final_model.fit(X, y)
```

```
Out[37]: Pipeline(steps=[('impute_means', Imputer(axis=0, copy=True, missing_values='NaN', strategy='mean', verbose=0)), ('impute_modes', Imputer(axis=0, copy=True, missing_values=-1, strategy='most_frequent', verbose=0)), ('standardize', StandardScaler(copy=True, with_mean=True, with_std=True)), ('estimator', Lasso(alpha=5.075, copy_X=True, fit_intercept=True, max_iter=1000, normalize=False, positive=False, precompute='auto', tol=0.0001, warm_start=False))])
```

Now whenever we want a prediction for a house price, this is the system that we will use. For example, a property that has a floor area of 114 square metres, 3 bedrooms, 2 bathrooms, 2 floors, is a second-hand development, is semi-detached, has a BER of C1, and is situated in Ballinlough:

```python
In [38]: final_model.predict([114, 3, 2, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
Out[38]: array([ 232.42375259])
```