Initialization

In [1]: %reload_ext autoreload
   %autoreload 2
   %matplotlib inline

In [2]: import warnings
   from sklearn.exceptions import DataConversionWarning
   warnings.filterwarnings(action='ignore', category=DataConversionWarning)
   warnings.filterwarnings(action='ignore', category=FutureWarning)

In [3]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt

In [4]: from sklearn.compose import ColumnTransformer
   from sklearn.pipeline import Pipeline
   from sklearn.impute import SimpleImputer
   from sklearn.preprocessing import OneHotEncoder
   from sklearn.preprocessing import StandardScaler
   from sklearn.decomposition import PCA
   from sklearn.linear_model import LinearRegression
   from sklearn.metrics import mean_squared_error
   from sklearn.metrics import mean_absolute_error
   from sklearn.model_selection import cross_val_score
   from sklearn.model_selection import ShuffleSplit
   from sklearn.model_selection import KFold

Building a Pipeline
We'll repeat the code from the end of the previous slide but with a difference

- We'll combine the `ColumnTransformer` and the `LinearRegression` into a Pipeline
  - This is very convenient: we only need to call `fit` and `predict`; we do not need to call `transform`. We'll explain how this works at the end of the lecture
  - It is very common — please do it this way!

```python
In [5]: # Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")

In [6]: # The features we want to select
numeric_features = ["flarea", "brms", "bthrms", "floors"]
nominal_features = ["type", "devment", "ber", "location"]

# Create the preprocessor
preprocessor = ColumnTransformer(["num", StandardScaler(), numeric_features],
                                   ["nom", OneHotEncoder(handle_unknown="ignore"), nominal_features],
                                   remainder="drop")

# Create a pipeline that combines the preprocessor with the estimator
pipeline = Pipeline(["pre", preprocessor],
                     ["est", LinearRegression()])

In [7]: # Get the target values
y = df["price"].values
```
How Good Is This Model?

- We've built an estimator by learning a model from a dataset
- We want to know how well it will do in practice, once we start to use it to make predictions
  - This is called error estimation
- Easy right?
  - The dataset comes with actual target values
  - We can ask the estimator to predict target values for each example in the dataset
  - So now we have actual and predicted values, we can compute the mean squared error

```
In [10]: y_predicted = pipeline.predict(df)
```
In [11]: mean_squared_error(y, y_predicted)
Out[11]: 4909.934051410393

• But, for at least two reasons, we don’t do this!
  ■ We might want to use a different performance measure than what we used as the loss function
  ■ We want to know how well the model generalizes to unseen data

Choosing a Different Performance Measure

• Often in machine learning, we use one measure during learning and another for evaluation
• Class exercise: We already saw this with $k$-means clustering. Explain!
• Our loss function (mean squared error or half of it!) was ideal for learning (why?) but may not be so good as a performance measure
  ■ We could use root mean squared error (RMSE):

\[
\sqrt{\frac{1}{m} \sum_{i=1}^{m} (h_{\beta}(x^{(i)}) - y^{(i)})^2}
\]

(i.e don’t halve the MSE, and take its square root: it’s the standard deviation of the errors in the predictions)

  ■ We could use mean absolute error (MAE):

\[
\frac{1}{m} \sum_{i=1}^{m} \text{abs}(h_{\beta}(x^{(i)}) - y^{(i)})
\]

In [12]: mean_absolute_error(y, y_predicted)
Out[12]: 47.70469066521621

Generalizing to Unseen Data

• The error on the training set is called the training error (also ‘resubstitution error’ and ‘in-sample error’)
• But we want to know how well we will perform in the future, on unseen data
  ■ The training error is not, in general a good indicator of performance on unseen data
  ■ It’s often too optimistic. Why?
• To predict future performance, we need to measure error on an independent dataset
  ■ A dataset that played no part in creating the estimator
  ■ This second dataset is called the test set
  ■ The error on the test set is called the test error (also ‘out-of-sample error’ and ‘extra-sample error’)
Holdout

- So we use the following method:
  - **Partition** our dataset at random into two:
    - training set (e.g. 80% of the full dataset)
    - test set (the rest of the full dataset)
  - Train the estimator on the training set
  - Test the model (evaluate the predictions) on the test set
- This method is called the **holdout** method, because the test set is withheld (held-out) during training
  - It is essential that the test set is not used in any way to create the estimator
  - *Don’t even look at it!*
  - ‘Cheating’ is called **leakage**
  - (Advanced: ‘Cheating’ is one cause of **overfitting**)
- Class exercise: Standardization, as we know, is about scaling the data. It requires calculation of the mean and standard deviation. When should the mean and standard deviation be calculated: (a) before splitting, on the entire dataset, or (b) after splitting, on just the training set? Why?

Holdout in scikit-learn

```
In [13]: # Create the object that splits the data
    ss = ShuffleSplit(n_splits=1, train_size=0.8)

In [14]: # Run the pipeline
    cross_val_score(pipeline, df, y, scoring="neg_mean_absolute_error", cv=ss)

Out[14]: array([-61.42136874])
```

- This is the negative of the MAE — so that higher values (closer to zero) are better
- Compare this value to what we got earlier, when we were training and testing on the whole dataset
- Run it again: what do you notice?
Pipelines Explained

- We are finally in a better position to explain ColumnTransformer s and Pipeline s in scikit-learn
- A scikit-learn Pipeline contains a number of steps
- All the steps, with the possible exception of the last step, must be transformers:
  - They are used to transform the data, e.g. to scale it; to binarize it; to reduce its dimensions; ...
  - They have a method called fit, which computes any values needed to carry out the transformation
    - E.g. what does the fit method of StandardScaler compute?
    - E.g. what about PCA?
  - They have a method called transform, which uses the values computed by fit to modify whatever data is passed to it
    - E.g. what does the transform method of StandardScaler do?
    - What about PCA?
- The last step in a Pipeline can be an estimator:
  - They are used to build models from the data and make predictions (typically regression and classification)
  - They have a method called fit which learns the model from the data
    - E.g. what does the fit method of LinearRegression do?
  - They have a method called predict, which uses the model learned by the fit method to make predictions
- Pipeline s themselves have various methods including:
  - fit: this calls the fit method of the first step, then its transform method; then the fit method of the second step, then its transform method; and so on; and, eventually, if the last step is an estimator, it calls the fit method of the estimator
  - predict: this calls the transform method of the first step; then the transform method of the second step; and so on; and, eventually, if the last step is an estimator, it calls the predict method of the estimator

Hence it makes sense:
- to call the pipeline's fit method on the training set
- then to call the pipeline's predict method on the test set

Pipelines Explained, continued

- You pass your training set into a pipeline's fit method:

  ![Diagram of training set pipeline](image)

- Then, you pass your test data into a pipeline's predict method:

  ![Diagram of test data pipeline](image)
cross_val_score explained

In [15]:
# Create the object that splits the data
ss = ShuffleSplit(n_splits=1, train_size=0.8)

# Run the pipeline
cross_val_score(pipeline, df, y, scoring="neg_mean_absolute_error", cv=ss)

Out[15]: array([-55.03544912])

- cross_val_score takes in the full dataset and the target values
- It splits the dataset in a way determined by its cv parameter
- It calls the pipeline’s fit method on the training set
- It calls the pipeline’s predict method on the test set
- It compares the test set’s actual target values with the test set’s predictions using the scoring function

Pros and Cons of Holdout

- The advantage of holdout is:
  - The test error is independent of the training set
- The disadvantages of this method are:
  - Results can vary quite a lot across different runs
    - Informally, you might get lucky — or unlucky
      - I.e. in any one split, the data used for training or testing might not be representative
    - We are training on only a subset of the available dataset, perhaps as little as 50% of it
      - From so little data, we may learn a worse model and so our error measurement may be pessimistic
- In practice, we only use the holdout method when we have a very large dataset
- The size of the dataset mitigates the above problems
- When we have a smaller dataset, we use a resampling method:
  - The examples get re-used for training and testing

\(k\)-Fold Cross-Validation

- The most-used resampling method is \(k\)-fold cross-validation:
  - We randomly partition the data into \(k\) disjoint subsets of equal size
    - Each of the partitions is called a fold
      - Typically, \(k = 10\), so you have 10 folds
  - You take each fold in turn and use it as the test set, training the learner on the remaining folds
  - Clearly, you can do this \(k\) times, so that each fold gets ‘a turn’ at being the test set
    - By this method, each example is used exactly once for testing, and \(k - 1\) times for training
- In pseudocode:
  - partition the dataset \(D\) into \(k\) disjoint equal-sized subsets, \(T_1, T_2, \ldots, T_k\)
  - for \(i = 1\) to \(k\)
    - train on \(D \setminus T_i\)
    - make predictions for \(T_i\)
    - measure error (e.g. MAE)
  - report the mean of the errors
Pros and Cons of $k$-Fold Cross-Validation

- **Pros:**
  - The test errors of the folds are independent — because examples are included in only one test set
  - Better use is made of the dataset: for $k = 10$, for example, we train using 9/10 of the dataset

- **Cons:**
  - While the test sets are independent of each other, the training sets are not:
    - They will overlap with each other to some degree
    - (This effect of this will be less, of course, for larger datasets)
  - The number of folds is constrained by the size of the dataset and the desire to have folds of at least 30 examples
  - It can be costly to train the learning algorithm $k$ times
  - There may still be some variability in the results due to ‘lucky’/’unlucky’ splits

$k$-Fold Cross Validation in scikit-learn

```python
In [16]: # Create the object that splits the data
kf = KFold(n_splits=10)

# Run the pipeline
np.mean(cross_val_score(pipeline, df, y, scoring="neg_mean_absolute_error", cv=kf))

Out[16]: -56.9894439116823
```

- But $k$-fold cross-validation is so common, there’s a shorthand:

```python
In [17]: np.mean(cross_val_score(pipeline, df, y, scoring="neg_mean_absolute_error", cv=10))

Out[17]: -56.9894439116823
```

- Be warned, however, this does not shuffle the dataset before splitting it into folds
  - Why might that be a problem?
  - If you use the longer form, you can ask it to shuffle: kf = KFold(n_splits=10, shuffle=True)
  - If you use the shorthand, you should probably shuffle the DataFrame just after reading it in from the CSV file (see example below)
Final Remarks

- In the past, students have tried holdout and $k$-fold as if they were in competition with each other. This betrays a misunderstanding. You do not try them both and see which one gives the lower error. You pick one of them — the one that makes most sense for your data — and use it.
- There are many resampling methods other than $k$-Fold Cross-Validation:
  - Repeated $k$-Fold Cross-Validation, Leave-One-Out-Cross-Validation, ...
- So you've used one of the above methods and found the test error of your estimator.
  - This is supposed to give you an idea of how your estimator will perform in practice.
  - What if you are dissatisfied with the test error? It seems too high.
    - It is tempting to tweak your learning algorithm or try different algorithms to try to bring down the test error.
    - This is wrong! It is leakage again: you will be using knowledge of the test set to develop the estimator and is likely to result in an optimistic view of the ultimate performance of the estimator on unseen data.
    - Ideally, error estimation on the test set is the last thing you do.
- Finally, suppose you have used one of the above methods to estimate the error of your regressor. You are ready to release your regressor on the world. At this point, you can train it on all the examples in your dataset, so as to maximize the use of the data.

A Little Case Study in scikit-learn

```
In [18]: # Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")

In [19]: # Shuffle
df = df.take(np.random.permutation(len(df)))

In [20]: # The features we want to select
    numeric_features = ["flarea", "bdrms", "bthrms", "floors"]
    nominal_features = ["type", "devment", "ber", "location"]

    # Create a preprocessor - without PCA
    preprocessor_no_PCA = ColumnTransformer([
        ("num", StandardScaler(), numeric_features),
        ("nom", OneHotEncoder(handle_unknown="ignore"), nominal_features)],
        remainder="drop")

    # Create a pipeline that combines the preprocessor with the estimator
    pipeline_no_PCA = Pipeline([
        ("pre", preprocessor_no_PCA),
        ("est", LinearRegression())])

    # Create a preprocessor - with PCA
    preprocessor_with_PCA = ColumnTransformer([
        ("num", PCA(n_components=0.9), numeric_features),
        ("nom", OneHotEncoder(handle_unknown="ignore"), nominal_features)],
        remainder="drop")

    # Create a pipeline that combines the preprocessor with the estimator
    pipeline_with_PCA = Pipeline([
        ("pre", preprocessor_with_PCA),
        ("est", LinearRegression())])

In [21]: # Get the target values
    y = df["price"].values
```
# Run the no-PCA pipeline
```
np.mean(cross_val_score(pipeline_no_PCA, df, y, scoring="neg_mean_absolute_error", cv=10))
```

Out[22]: -57.64174989265905

# Run the pipeline with PCA
```
np.mean(cross_val_score(pipeline_with_PCA, df, y, scoring="neg_mean_absolute_error", cv=10))
```

Out[23]: -58.027251972001714

- It appears that not using PCA is better
- However, if we had more time, we'd study **statistical significance testing**, which would tell us whether the difference between the two is likely to be due to chance
- I haven't done such a test here, but assuming that it says that this difference *is* due to chance, then the difference in these scores would not be a basis for preferring one version over the other