In this tutorial, you will learn how to use **pipelines** to clean up your modeling code.

Introduction

Pipelines are a simple way to keep your data preprocessing and modeling code organized. Specifically, a pipeline bundles preprocessing and modeling steps so you can use the whole bundle as if it were a single step.

Many data scientists hack together models without pipelines, but pipelines have some important benefits. Those include:

1. **Cleaner Code:** Accounting for data at each step of preprocessing can get messy. With a pipeline, you won't need to manually keep track of your training and validation data at each step.
2. **Fewer Bugs:** There are fewer opportunities to misapply a step or forget a preprocessing step.
3. **Easier to Productionize:** It can be surprisingly hard to transition a model from a prototype to something deployable at scale. We won't go into the many related concerns here, but pipelines can help.
4. **More Options for Model Validation:** You will see an example in the next tutorial, which covers cross-validation.

Example

As in the previous tutorial, we will work with the Melbourne Housing dataset.

We won't focus on the data loading step. Instead, you can imagine you are at a point where you already have the training and validation data in `X_train`, `X_valid`, `y_train`, and `y_valid`.

Hide

In [1]:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data
data = pd.read_csv('../input/melbourne-housing-snapshot/melb_data.csv')

# Separate target from predictors
y = data.Price
X = data.drop(['Price'], axis=1)

# Divide data into training and validation subsets
X_train_full, X_valid_full, y_train, y_valid = train_test_split(X, y, train_size=
0.8, test_size=0.2,
                                                                random_state=0)

# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbi
trary)
categorical_cols = [cname for cname in X_train_full.columns if X_train_full[cname
].nunique() < 10 and
                        X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].
dtype in ['int64', 'float64']]

# Keep selected columns only
my_cols = categorical_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

We take a peek at the training data with the `head()` method below. Notice that the data contains both categorical data and columns with missing values. With a pipeline, it's easy to deal with both!

In [2]:

```
X_train.head()
```

Out[2]:

	Type	Method	Regionname	Rooms	Distance	Postcode	Bedroom2	Bathroom	Car	La
12167	u	S	Southern Metropolitan	1	5.0	3182.0	1.0	1.0	1.0	0.
6524	h	SA	Western Metropolitan	2	8.0	3016.0	2.0	2.0	1.0	19
8413	h	S	Western Metropolitan	3	12.6	3020.0	3.0	1.0	1.0	55
2919	u	SP	Northern Metropolitan	3	13.0	3046.0	3.0	1.0	1.0	26
6043	h	S	Western Metropolitan	3	13.3	3020.0	3.0	1.0	2.0	67

We construct the full pipeline in three steps.

Step 1: Define Preprocessing Steps

Similar to how a pipeline bundles together preprocessing and modeling steps, we use the `ColumnTransformer` class to bundle together different preprocessing steps. The code below:

- imputes missing values in ***numerical*** data, and
- imputes missing values and applies a one-hot encoding to ***categorical*** data.

In [3]:

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy='constant')

# Preprocessing for categorical data
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

Step 2: Define the Model

Next, we define a random forest model with the familiar `RandomForestRegressor` class.

In [4]:

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=0)
```

Step 3: Create and Evaluate the Pipeline

Finally, we use the `Pipeline` class to define a pipeline that bundles the preprocessing and modeling steps. There are a few important things to notice:

- With the pipeline, we preprocess the training data and fit the model in a single line of code. (*In contrast, without a pipeline, we have to do imputation, one-hot encoding, and model training in separate steps. This becomes especially messy if we have to deal with both numerical and categorical variables!*)
- With the pipeline, we supply the unprocessed features in `X_valid` to the `predict()` command, and the pipeline automatically preprocesses the features before generating predictions. (*However, without a pipeline, we have to remember to preprocess the validation data before making predictions.*)

In [5]:

```
from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model)
                             ])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

```
MAE: 160679.18917034855
```

Conclusion

Pipelines are valuable for cleaning up machine learning code and avoiding errors, and are especially useful for workflows with sophisticated data preprocessing.

Your Turn

Use a pipeline in the **next exercise** to use advanced data preprocessing techniques and improve your predictions!