Refactoring Your Code to Use Pipelines

Introduction

In this lesson, you will learn how to use the core features of scikit-learn pipelines to refactor existing machine learning preprocessing code into a portable pipeline format.

Objectives

You will be able to:

- Recall the benefits of using pipelines
- Describe the difference between a `Pipeline`, a `FeatureUnion`, and a `ColumnTransformer` in scikit-learn
- Iteratively refactor existing preprocessing code into a pipeline

Pipelines in the Data Science Process

If my code already works, why do I need a pipeline?

As we covered previously, pipelines are a great way to organize your code in a DRY (don't repeat yourself) fashion. It also allows you to perform cross validation (including `GridSearchCV`) in a way that avoids leakage, because you are performing all preprocessing steps separately. Finally, it's helpful if you want to deploy your code, since it means that you only need to pickle the overall pipeline, rather than pickling the fitted model as well as all of the fitted preprocessing transformers.

Then why not just write a pipeline from the start?

Pipelines are designed for efficiency rather than readability, so they can become very confusing very quickly if something goes wrong. (All of the data is in NumPy arrays, not pandas dataframes, so there are no column labels by default.)

Therefore it's a good idea to write most of your code without pipelines at first, then refactor it. Eventually if you are very confident with pipelines you can save time by writing them from the start, but it's okay if you stick with the refactoring strategy!

Code without Pipelines

Let's say we have the following (very-simple) dataset:

In [1]:
```
import pandas as pd

example_data = pd.DataFrame([
    {"category": "A", "number": 7, "target": 1},
    {"category": "A", "number": 8, "target": 1},
    {"category": "B", "number": 9, "target": 0},
    {"category": "B", "number": 7, "target": 1},
    {"category": "C", "number": 4, "target": 0}
])

example_X = example_data.drop("target", axis=1)
example_y = example_data["target"]

example_X
```

Out[1]:

	category	number
0	A	7
1	A	8
2	B	9
3	B	7
4	C	4

Preprocessing Steps without Pipelines

These steps should be a review of preprocessing steps you have learned previously.

One-Hot Encoding Categorical Data

If we just tried to apply a `StandardScaler` then a `LogisticRegression` to this dataset, we would get a `ValueError` because the values in `category` are not yet numeric.

So, let's use a `OneHotEncoder` to convert the `category` column into multiple dummy columns representing each of the categories present:

In [2]:
```
from sklearn.preprocessing import OneHotEncoder

# Make a transformer
ohe = OneHotEncoder(categories="auto", handle_unknown="ignore", sparse=False)

# Create transformed dataframe
category_encoded = ohe.fit_transform(example_X[["category"]])
category_encoded = pd.DataFrame(
    category_encoded,
    columns=ohe.categories_[0],
    index=example_X.index
)

# Replace categorical data with encoded data
example_X.drop("category", axis=1, inplace=True)
example_X = pd.concat([category_encoded, example_X], axis=1)

# Visually inspect dataframe
example_X
```

Out[2]:

	A	B	C	number
0	1.0	0.0	0.0	7
1	1.0	0.0	0.0	8
2	0.0	1.0	0.0	9
3	0.0	1.0	0.0	7
4	0.0	0.0	1.0	4

Feature Engineering

Let's say for the sake of example that we wanted to add a new feature called `number_odd`, which is `1` when the value of `number` is odd and `0` when the value of `number` is even. (It's not clear why this would be useful, but you can imagine a more realistic example, e.g. a boolean flag related to a purchase threshold that triggers free shipping.)

We don't want to remove `number` and replace it with `number_odd`, we want an entire new feature `number_odd` to be added.

Let's make a custom transformer for this purpose. Specifically, we'll use a `FunctionTransformer` (documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html)). As you might have guessed from the name, a `FunctionTransformer` takes in a function as an argument (similar to the `.apply` dataframe method) and uses that function to transform the data. Unlike just using `.apply`, this transformer has the typical `.fit_transform` interface and can be used just like any other transformer (including being used in a pipeline).

In [3]:
```
from sklearn.preprocessing import FunctionTransformer

def is_odd(data):
    """
    Helper function that returns 1 if odd, 0 if even
    """
    return data % 2

# Instantiate transformer
func_transformer = FunctionTransformer(is_odd)

# Create transformed column
number_odd = func_transformer.fit_transform(example_X["number"])

# Add engineered column
example_X["number_odd"] = number_odd
example_X
```

Out[3]:

	A	B	C	number	number_odd
0	1.0	0.0	0.0	7	1
1	1.0	0.0	0.0	8	0
2	0.0	1.0	0.0	9	1
3	0.0	1.0	0.0	7	1
4	0.0	0.0	1.0	4	0

Scaling

Then let's say we want to scale all of the features after the previous steps have been taken:

In [4]:
```
from sklearn.preprocessing import StandardScaler

# Instantiate transformer
scaler = StandardScaler()

# Create transformed dataset
data_scaled = scaler.fit_transform(example_X)

# Replace dataset with transformed one
example_X = pd.DataFrame(
    data_scaled,
    columns=example_X.columns,
    index=example_X.index
)
example_X
```

Out[4]:

	A	B	C	number	number_odd
0	1.224745	-0.816497	-0.5	0.000000	0.816497
1	1.224745	-0.816497	-0.5	0.597614	-1.224745
2	-0.816497	1.224745	-0.5	1.195229	0.816497
3	-0.816497	1.224745	-0.5	0.000000	0.816497
4	-0.816497	-0.816497	2.0	-1.792843	-1.224745

Bringing It All Together

Here is the full preprocessing example without a pipeline:

In [5]:
```
def preprocess_data_without_pipeline(X):

    transformers = []

    ### Encoding categorical data ###

    # Make a transformer
    ohe = OneHotEncoder(categories="auto", handle_unknown="ignore", sparse=False)

    # Create transformed dataframe
    category_encoded = ohe.fit_transform(X[["category"]])
    category_encoded = pd.DataFrame(
        category_encoded,
        columns=ohe.categories_[0],
        index=X.index
    )
    transformers.append(ohe)

    # Replace categorical data with encoded data
    X.drop("category", axis=1, inplace=True)
    X = pd.concat([category_encoded, X], axis=1)

    ### Feature engineering ###

    def is_odd(data):
        """
        Helper function that returns 1 if odd, 0 if even
        """
        return data % 2

    # Instantiate transformer
    func_transformer = FunctionTransformer(is_odd)

    # Create transformed column
    number_odd = func_transformer.fit_transform(X["number"])
    transformers.append(func_transformer)

    # Add engineered column
    X["number_odd"] = number_odd

    ### Scaling ###

    # Instantiate transformer
    scaler = StandardScaler()

    # Create transformed dataset
    data_scaled = scaler.fit_transform(X)
    transformers.append(scaler)

    # Replace dataset with transformed one
    X = pd.DataFrame(
        data_scaled,
        columns=X.columns,
        index=X.index
    )

    return X, transformers

# Reset value of example_X
example_X = example_data.drop("target", axis=1)
# Test out our function
result, transformers = preprocess_data_without_pipeline(example_X)
result
```

Out[5]:

	A	B	C	number	number_odd
0	1.224745	-0.816497	-0.5	0.000000	0.816497
1	1.224745	-0.816497	-0.5	0.597614	-1.224745
2	-0.816497	1.224745	-0.5	1.195229	0.816497
3	-0.816497	1.224745	-0.5	0.000000	0.816497
4	-0.816497	-0.816497	2.0	-1.792843	-1.224745

Now let's rewrite that with pipeline logic!

Pieces of a Pipeline

`Pipeline` Class

In a previous lesson, we introduced the most fundamental part of pipelines: the `Pipeline` class. This class is useful if you want to perform the same steps on every single column in your dataset. A simple example of just using a `Pipeline` would be:

```
pipe = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("model", LogisticRegression())
])
```

However, many interesting datasets contain a mixture of kinds of data (e.g. numeric and categorical data), which means you often do not want to perform the same steps on every column. For example, one-hot encoding is useful for converting categorical data into a format that is usable in ML models, but one-hot encoding numeric data is a bad idea. You also usually want to apply different feature engineering processes to different features.

In order to apply different data cleaning and feature engineering steps to different columns, we'll use the `FeatureUnion` and `ColumnTransformer` classes.

`ColumnTransformer` Class

The core idea of a `ColumnTransformer` is that you can **apply different preprocessing steps to different columns of the dataset**.

Looking at the preprocessing steps above, we only want to apply the `OneHotEncoder` to the `category` column, so this is a good use case for a `ColumnTransformer`:

In [6]:
```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Reset value of example_X
example_X = example_data.drop("target", axis=1)

# Create a column transformer
col_transformer = ColumnTransformer(transformers=[
    ("ohe", OneHotEncoder(categories="auto", handle_unknown="ignore"), ["category"])
], remainder="passthrough")

# Create a pipeline containing the single column transformer
pipe = Pipeline(steps=[
    ("col_transformer", col_transformer)
])

# Use the pipeline to fit and transform the data
transformed_data = pipe.fit_transform(example_X)
transformed_data
```

Out[6]:
```
array([[1., 0., 0., 7.],
       [1., 0., 0., 8.],
       [0., 1., 0., 9.],
       [0., 1., 0., 7.],
       [0., 0., 1., 4.]])
```

The pipeline returns a NumPy array, but we can convert it back into a dataframe for readability if we want to:

In [7]:
```
import numpy as np

# Extract the category labels from the OHE within the pipeline
encoder = col_transformer.named_transformers_["ohe"]
category_labels = encoder.categories_[0]

# Make a dataframe with the relevant columns
pd.DataFrame(transformed_data, columns=np.append(category_labels, "number"))
```

Out[7]:

	A	B	C	number
0	1.0	0.0	0.0	7.0
1	1.0	0.0	0.0	8.0
2	0.0	1.0	0.0	9.0
3	0.0	1.0	0.0	7.0
4	0.0	0.0	1.0	4.0

Interpreting the `ColumnTransformer` Example

Let's go back and look at each of those steps more closely.

First, creating a column transformer. Here is what that code looked like above:

```
# Create a column transformer
col_transformer = ColumnTransformer(transformers=[
    ("ohe", OneHotEncoder(categories="auto", handle_unknown="ignore"), ["category"])
], remainder="passthrough")
```

Here is the same code, spread out so we can add more comments explaining what's happening:

```
# Create a column transformer
col_transformer = ColumnTransformer(
    # ColumnTransformer takes a list of "transformers", each of which
    # is represented by a three-tuple (not just a transformer object)
    transformers=[
        # Each tuple has three parts
        (
            # (1) This is a string representing the name. It is there
            # for readability and so you can extract information from
            # the pipeline later. scikit-learn doesn't actually care
            # what the name is.
            "ohe",
            # (2) This is the actual transformer
            OneHotEncoder(categories="auto", handle_unknown="ignore"),
            # (3) This is the list of columns that the transformer should
            # apply to. In this case, there is only one column, but it
            # still needs to be in a list
            ["category"]
        )
        # If we wanted to perform multiple different transformations
        # on different columns, we would add more tuples here
    ],
    # By default, any column that is not specified in the list of
    # transformer tuples will be dropped, but we can indicate that we
    # want them to stay as-is if we set remainder="passthrough"
    remainder="passthrough"
)
```

Next, putting the column transformer into a pipeline. Here is that original code:

```
# Create a pipeline containing the single column transformer
pipe = Pipeline(steps=[
    ("col_transformer", col_transformer)
])
```

And again, here it is with more comments:

```
# Create a pipeline containing the single column transformer
pipe = Pipeline(
    # Pipeline takes a list of "steps", each of which is
    # represented by a two-tuple (not just a transformer or
    # estimator object)
    steps=[
        # Each tuple has two parts
        (
            # (1) This is name of the step. Again, this is for
            # readability and retrieving steps later, so just
            # choose a name that makes sense to you
            "col_transformer",
            # (2) This is the actual transformer or estimator.
            # Note that a transformer can be a simple one like
            # StandardScaler, or a composite one like a
            # ColumnTransformer (shown here), a FeatureUnion,
            # or another Pipeline.
            # Typically the last step will be an estimator
            # (i.e. a model that makes predictions)
            col_transformer
        )
    ]
)
```

`FeatureUnion` Class

A `FeatureUnion` **concatenates together the results of multiple different transformers**. While `Pipeline` and a `ColumnTransformer` are usually enough to perform basic *data cleaning* forms of preprocessing, it's also helpful to be able to use a `FeatureUnion` for *feature engineering* forms of preprocessing.

Let's use a `FeatureUnion` to add on the `number_odd` feature from before. Because we only want this transformation to apply to the `number` column, we need to wrap it in a `ColumnTransformer` again. Let's call this new one `feature_eng` to indicate what it is doing:

In [8]:
```
# Create a ColumnTransformer for feature engineering
feature_eng = ColumnTransformer(transformers=[
    ("add_number_odd", FunctionTransformer(is_odd), ["number"])
], remainder="drop")
```

Let's also rename the other `ColumnTransformer` to `original_features_encoded` to make it clearer what it is responsible for:

In [9]:
```
# Create a ColumnTransformer to encode categorical data
# and keep numeric data as-is
original_features_encoded = ColumnTransformer(transformers=[
    ("ohe", OneHotEncoder(categories="auto", handle_unknown="ignore"), ["category"])
], remainder="passthrough")
```

Now we can combine those two into a `FeatureUnion`:

In [10]:
```
from sklearn.pipeline import FeatureUnion

feature_union = FeatureUnion(transformer_list=[
    ("encoded_features", original_features_encoded),
    ("engineered_features", feature_eng)
])
```

And put that `FeatureUnion` into a `Pipeline`:

In [11]:
```
# Create a pipeline containing union of encoded
# original features and engineered features
pipe = Pipeline(steps=[
    ("feature_union", feature_union)
])

# Use the pipeline to fit and transform the data
transformed_data = pipe.fit_transform(example_X)
transformed_data
```

Out[11]:
```
array([[1., 0., 0., 7., 1.],
       [1., 0., 0., 8., 0.],
       [0., 1., 0., 9., 1.],
       [0., 1., 0., 7., 1.],
       [0., 0., 1., 4., 0.]])
```

Again, here it is as a more-readable dataframe:

In [12]:
```
# Extract the category labels from the OHE within the pipeline
encoder = original_features_encoded.named_transformers_["ohe"]
category_labels = encoder.categories_[0]

# Make a dataframe with the relevant columns
all_cols = list(category_labels) + ["number", "number_odd"]
pd.DataFrame(transformed_data, columns=all_cols)
```

Out[12]:

	A	B	C	number	number_odd
0	1.0	0.0	0.0	7.0	1.0
1	1.0	0.0	0.0	8.0	0.0
2	0.0	1.0	0.0	9.0	1.0
3	0.0	1.0	0.0	7.0	1.0
4	0.0	0.0	1.0	4.0	0.0

Interpreting the `FeatureUnion` Example

Once more, here was the code used to create the `FeatureUnion`:

```
feature_union = FeatureUnion(transformer_list=[
    ("encoded_features", original_features_encoded),
    ("engineered_features", feature_eng)
])
```

And here it is spread out with more comments:

```
feature_union = FeatureUnion(
    # FeatureUnion takes a "transformer_list" containing
    # two-tuples (not just transformers)
    transformer_list=[
        # Each tuple contains two elements
        (
            # (1) Name of the feature. If you make this "drop",
            # the transformer will be ignored
            "encoded_features",
            # (2) The actual transformer (in this case, a
            # ColumnTransformer). This one will produce the
            # numeric features as-is and the categorical
            # features one-hot encoded
            original_features_encoded
        ),
        # Here is another tuple
        (
            # (1) Name of the feature
            "engineered_features",
            # (2) The actual transformer (again, a
            # ColumnTransformer). This one will produce just
            # the flag of whether the number is even or odd
            feature_eng
        )
    ]
)
```

Adding Final Steps to Pipeline

If we want to scale all of the features at the end, this doesn't require any additional `ColumnTransformer` or `FeatureUnion` objects, it just means we need to add another step in our `Pipeline` like this:

In [13]:
```
# Create a pipeline containing union of encoded
# original features and engineered features, then
# all features scaled
pipe = Pipeline(steps=[
    ("feature_union", feature_union),
    ("scale", StandardScaler())
])

# Use the pipeline to fit and transform the data
transformed_data = pipe.fit_transform(example_X)
transformed_data
```

Out[13]:
```
array([[ 1.22474487, -0.81649658, -0.5       ,  0.        ,  0.81649658],
       [ 1.22474487, -0.81649658, -0.5       ,  0.5976143 , -1.22474487],
       [-0.81649658,  1.22474487, -0.5       ,  1.19522861,  0.81649658],
       [-0.81649658,  1.22474487, -0.5       ,  0.        ,  0.81649658],
       [-0.81649658, -0.81649658,  2.        , -1.79284291, -1.22474487]])
```

Additionally, if we want to add an estimator (model) as the last step, we can do it like this:

In [14]:
```
from sklearn.linear_model import LogisticRegression

# Create a pipeline containing union of encoded
# original features and engineered features, then
# all features scaled, then feed into a model
pipe = Pipeline(steps=[
    ("feature_union", feature_union),
    ("scale", StandardScaler()),
    ("model", LogisticRegression())
])

# Use the pipeline to fit the model and score it
pipe.fit(example_X, example_y)
pipe.score(example_X, example_y)
```

Out[14]:
```
0.8
```

Complete Refactored Pipeline Example

Below is the complete pipeline (without the estimator), which produces the same output as the original full preprocessing example:

In [15]:
```
def preprocess_data_with_pipeline(X):

    ### Encoding categorical data ###
    original_features_encoded = ColumnTransformer(transformers=[
        ("ohe", OneHotEncoder(categories="auto", handle_unknown="ignore"), ["category"])
    ], remainder="passthrough")

    ### Feature engineering ###
    def is_odd(data):
        """
        Helper function that returns 1 if odd, 0 if even
        """
        return data % 2

    feature_eng = ColumnTransformer(transformers=[
        ("add_number_odd", FunctionTransformer(is_odd), ["number"])
    ], remainder="drop")

    ### Combine encoded and engineered features ###
    feature_union = FeatureUnion(transformer_list=[
        ("encoded_features", original_features_encoded),
        ("engineered_features", feature_eng)
    ])

    ### Pipeline (including scaling) ###
    pipe = Pipeline(steps=[
        ("feature_union", feature_union),
        ("scale", StandardScaler())
    ])

    transformed_data = pipe.fit_transform(X)

    ### Re-apply labels (optional step for readability) ###
    encoder = original_features_encoded.named_transformers_["ohe"]
    category_labels = encoder.categories_[0]
    all_cols = list(category_labels) + ["number", "number_odd"]
    return pd.DataFrame(transformed_data, columns=all_cols, index=X.index), pipe

# Reset value of example_X
example_X = example_data.drop("target", axis=1)
# Test out our new function
result, pipe = preprocess_data_with_pipeline(example_X)
result
```

Out[15]:

	A	B	C	number	number_odd
0	1.224745	-0.816497	-0.5	0.000000	0.816497
1	1.224745	-0.816497	-0.5	0.597614	-1.224745
2	-0.816497	1.224745	-0.5	1.195229	0.816497
3	-0.816497	1.224745	-0.5	0.000000	0.816497
4	-0.816497	-0.816497	2.0	-1.792843	-1.224745

Just to confirm, this produces the same result as the previous function:

In [16]:
```
# Reset value of example_X
example_X = example_data.drop("target", axis=1)
# Compare result to old function
result, transformers = preprocess_data_without_pipeline(example_X)
result
```

Out[16]:

	A	B	C	number	number_odd
0	1.224745	-0.816497	-0.5	0.000000	0.816497
1	1.224745	-0.816497	-0.5	0.597614	-1.224745
2	-0.816497	1.224745	-0.5	1.195229	0.816497
3	-0.816497	1.224745	-0.5	0.000000	0.816497
4	-0.816497	-0.816497	2.0	-1.792843	-1.224745

We achieved the same thing in fewer lines of code, better prevention of leakage, and the ability to pickle the whole process!

Note that in both cases we returned the object or objects used for preprocessing so they could be used on test data. Without a pipeline, we would need to apply each of the transformers in `transformers`. With a pipeline, we would just need to use `pipe.transform` on test data.

Summary

In this lesson, you learned how to make more-sophisticated pipelines using `ColumnTransformer` and `FeatureUnion` objects in addition to `Pipeline` s. We started with a preprocessing example that used sckit-learn code without pipelines, and rewrote it to use pipelines. Along the way we used `ColumnTransformer` to conditionally preprocess certain columns while leaving others alone, and `FeatureUnion` to combine engineered features with preprocessed versions of the original data. Now you should have a clearer idea of how pipelines can be used for non-trivial preprocessing tasks.