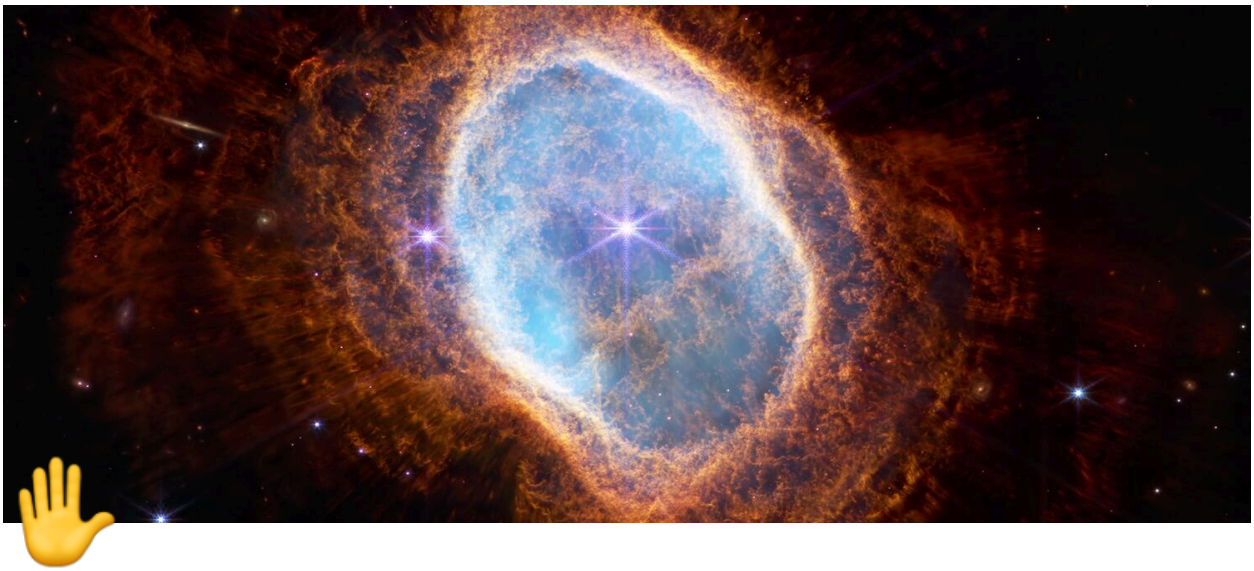

Chapter 4: Data Preprocessing

Created	@August 24, 2025 5:12 PM
Class	ML with Pytorch and Scikit Learn

Welcome back to “I already know this, but just gonna write to again to not forget”

Dealing with Missing Data

```
import pandas as pd
from io import StringIO

csv_data = \
    '''A,B,C,D
    1.0,2.0,3.0,4.0
    5.0,6.0,,8.0
    10.0,11.0,12.0,'''
df = pd.read_csv(StringIO(csv_data))

#fill all mising values with sum
df.isnull().sum()
```

```
#drop examples with missing values
df.dropna(axis=1) #drop colums with even 1 nan value
df.dropna(thresh=3) #specify to drop at what threshold ie. if 3 missing values then drop
df.dropna(subset=['C']) #drop from specific column

#Use imputer to just fill values
from sklearn.impute import SimpleImputer
import numpy as np

#Fills empty spaces with mean of other values
imr = SimpleImputer(missing_values=np.nan, strategy='mean')
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
imputed_data

#EVEN BETTER JUST USE NUMPY FUNCTION:
df.fillna(df.mean())
```

TRANSFORMER API IN SCIKIT-LEARN

It's an API in scikit-learn used for data manipulation and transformation:

It has 2 main functions: fit and transform, fit learns the training data and transform method transforms that data:

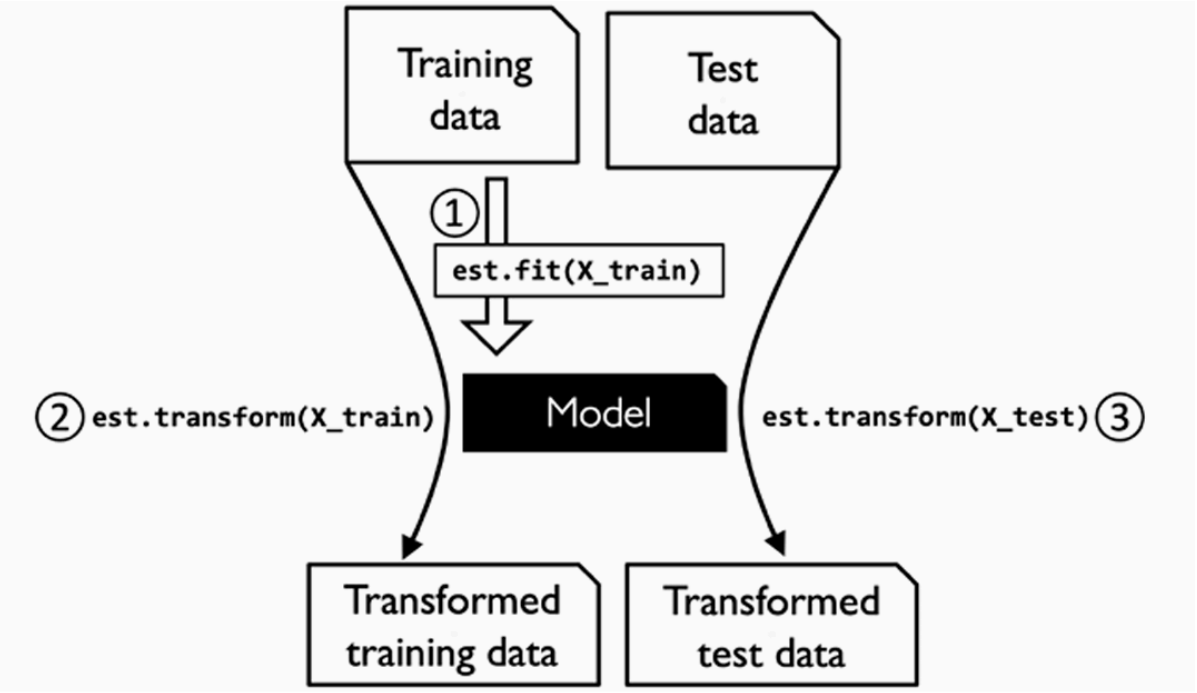

Figure 4.2: Using the scikit-learn API for data transformation

The classifiers we used before are just estimators of another API in scikit-learn (basically the same thing). They just have the predict method which we use to guess new values after learning from the training dataset

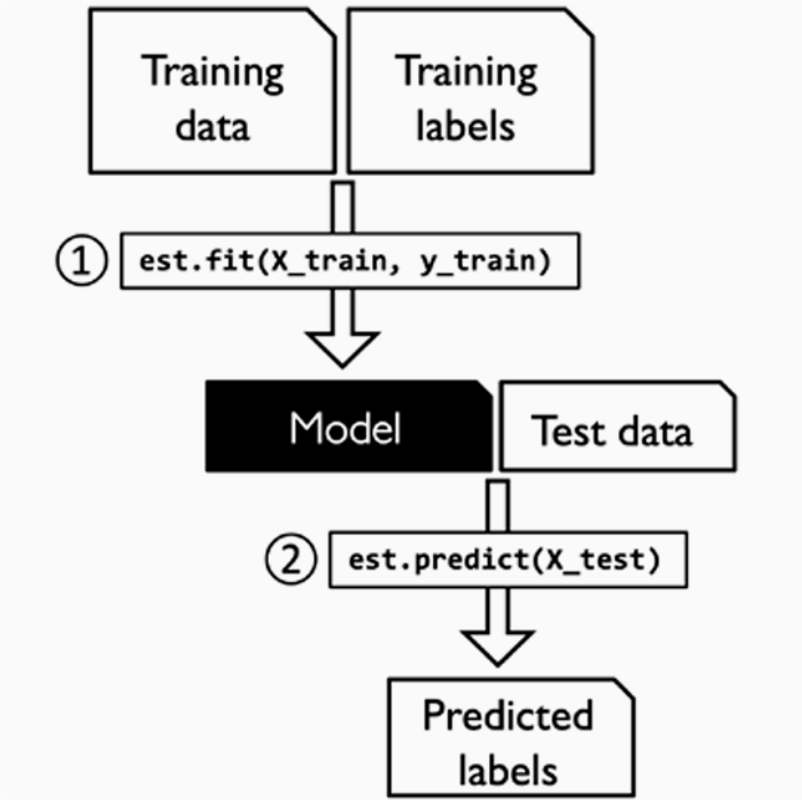

Figure 4.3: Using the scikit-learn API for predictive models such as classifiers

Handling Categorical Data

There are 2 types of categorical data: Nominal and Ordinal

Nominal: categories that **have names/labels only**, with **no inherent ranking**.

Ordinal: categories that **have a logical order/ranking**, but **differences between ranks are not uniform or meaningful**.

Using Label Encoder for Encoding the data:

```
from sklearn.preprocessing import LabelEncoder
class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
```

```
y
#Use inverse to revert the values back
class_le.inverse_transform(y)
```

If we stop at this point and feed the array to our classifier, we will make one of the most common
mistakes in dealing with categorical data. Can you spot the problem?

Although the color values don’t come in any particular order, common classification models, such as the ones covered in the previous chapters, will now assume that green is larger than blue, and red is larger than green.

Although this assumption is incorrect, a classifier could still produce useful results. However, those results would not be optimal.

A common workaround for this problem is to use a technique called one-hot encoding. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature column.

Here, we would convert the color feature into three new features: blue, green, and red. Binary values
can then be used to indicate the particular color of an example; for example, a blue example can be
encoded as blue=1, green=0, red=0.

```
from sklearn.preprocessing import OneHotEncoder
X = df[['color', 'size', 'price']].values
color_ohe = OneHotEncoder()
color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

An even more convenient way to create those dummy features via one-hot encoding is to use the

get_dummies method implemented in pandas. Applied to a DataFrame, the get_dummies method will only convert string columns and leave all other columns unchanged

```
pd.get_dummies(df[['price', 'color', 'size']])
    price  size  color_blue  color_green  color_red
0   10.1    1          0           1          0
1   13.5    2          0           0          1
2   15.3    3          1           0          0
```

Partitioning a dataset into separate training and test datasets

Use Sklearn's train test spit method:

```
from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.3,
...                      random_state=0,
...                      stratify=y)
```

Bringing features onto the same scale

Decision trees and random forests are two of the very few machine learning algorithms where we don't need to worry about feature scaling. Those algorithms are scale-invariant.

However, the majority of machine learning and optimization algorithms behave much better if features are on the same scale

Min Max Scaler is a very common and effective feature scaling method:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

Though, Standardization can be more practical for many machine learning algorithms, especially for optimization algorithms such as gradient descent.

The reason is that many linear models, such as the logistic regression and SVM from Chapter 3, initialize the weights to 0 or small random values close to 0

It goes:

$$z^{(i)} = \frac{x^{(i)} - \mu}{\sigma}$$

where:

- x(i) → the i-th data point
- μ → mean of the dataset
- σ → standard deviation

```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

Similarly, if the machine learning algorithm applied to this dataset is prone to overfitting, RobustScaler can be a good choice.

L1 and L2 regularization as penalties against model complexity

What is Regularization?

- A technique used to **reduce overfitting** in machine learning models.
- It adds a **penalty term** to the loss function based on the size of model weights.

L1 Regularization (Lasso)

- **Adds absolute values of weights** to the loss function.
- Encourages **sparsity**: pushes some weights exactly to **0** → performs **feature selection**.
- Useful when you think **only a few features are important**.

$$J(w) = \text{Loss}(w) + \lambda \sum_{j=1}^{n} |w_j|$$

✅ **Pros:**

- Can remove irrelevant features automatically.
- Good for **high-dimensional data** (e.g., text, genomics).

❌ **Cons:**

- Less stable when features are highly correlated.
- Optimization is a bit harder (because of absolute values).

```
from sklearn.linear_model import LogisticRegression
LogisticRegression(penalty='l1',
                   solver='liblinear',
                   multi_class='ovr')
```

L2 Regularization (Ridge)

- **Adds squared weights** to the loss function.
- Encourages weights to be **small**, but rarely zero.
- Keeps all features but **reduces their influence**.

$$J(w) = \text{Loss}(w) + \lambda \sum_{j=1}^{n} w_j^2$$

✅ **Pros:**

- Works well when **all features are useful**.
- More stable for **correlated features**.
- Easier optimization (smooth quadratic).

❌ **Cons:**

- Does not eliminate features → no feature selection.

Sequential feature selection algorithms

An alternative way to reduce the complexity of the model and avoid overfitting is dimensionality reduction via feature selection, which is especially useful for unregularized models.

There are two main categories of dimensionality reduction techniques: feature selection and feature extraction.