Machine Learning

|Custom Transformers

Prof. Dr. Mostafa Elhosseini

https://youtube.com/drmelhosseini

Agenda

- Custom Transformers

Custom Transformers

- Although Scikit-Learn provides many useful transformers, you will **need to write your own** for tasks such as **custom transformations**, cleanup operations, or combining specific attributes.
- For transformations that **don’t require any training**, you can just write a function that takes a NumPy array as input and outputs the transformed array

Custom Transformers

- To create simple transformers:

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

- Quick, simple transformations that don't require fitting to the data.
- **Stateless** Transformation: Since it's stateless, the fit method doesn't actually do anything other than return the transformer itself
- The **inverse_func** argument is **optional**. It lets you specify an inverse transform function, e.g., if you plan to use your transformer in a TransformedTargetRegressor.

Custom Transformers

- Your transformation function can take **hyperparameters** as additional **arguments**:

```
rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

- There's no inverse function for the RBF kernel, since there are always two values at a given distance from a fixed point (except at distance 0)

```
array([[2.81118530e-13],
       [8.20849986e-02],
       [6.70320046e-01],
       ...,
       [9.55316054e-22],
       [6.70320046e-01],
       [3.03539138e-04]])
```

Custom Transformers

- rbf_kernel() does not treat the features separately. If you pass it an array with two features, it will measure the 2D distance (Euclidean) to measure similarity
- Add a feature that will measure the geographic similarity between each district and San Francisco

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

```
sf_simil
```

```
array([[0.999927  ],
       [0.05258419],
       [0.94864161],
       ...,
       [0.00388525],
       [0.05038518],
       [0.99868067]])
```

Custom Transformers

- Custom transformers are also **useful to combine features**. For example, here's a FunctionTransformer that computes the ratio between the input features 0 and 1:

```
ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
```

```
array([[0.5 ],
       [0.75]])
```

Custom Transformers

- FunctionTransformer is very handy, but what if you would like your transformer to be **trainable**, learning some parameters in the **fit()** method and using them later in the **transform()** method.
- For this, you **need to write a custom class**.
- Scikit-Learn relies on **duck typing**, so this class does not have to inherit from any particular base class. All it needs is **three methods**:
  - **fit()** (which must return self),
  - **transform()**, and
  - **fit_transform()**.

Custom Transformers

- **Duck typing** is a concept used in dynamic programming languages like Python, where an object's suitability for use in a particular context is determined by the **presence of certain methods and properties**, rather than the object’s type itself being of a specific class.
- The term comes from the phrase "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck," which **emphasizes actions and behaviors over explicit types**.

Custom Transformers

- In the context of Scikit-Learn, duck typing means that as long as a class **implements certain methods** expected by Scikit-Learn's routines, **it can be used as part of Scikit-Learn workflows**, such as **pipelines** and **model fitting processes**.
- Specifically, for a class to be used as a transformer within Scikit-Learn, it needs to implement the following methods:
  - fit(self, X, y=None)
  - transform(self, X)
  - fit_transform(self, X, y=None)

Custom Transformers

- You can get **fit_transform()** for free by simply adding **TransformerMixin** as a base class: the default implementation will just call fit() and then transform().
- If you add BaseEstimator as a base class (and avoid using *args and **kwargs in your constructor), you will also **get two extra methods**: **get_params()** and **set_params()**. These will be useful for **automatic hyperparameter tuning**.

Custom Transformers

- For example, here's a custom transformer that acts much like the StandardScaler:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True):  # no *args or **kwargs!
        self.with_mean = with_mean

    def fit(self, X, y=None):  # y is required even though we don't use it
        X = check_array(X)  # checks that X is an array with finite float values
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1]  # every estimator stores this in fit()
        return self  # always return self!

    def transform(self, X):
        check_is_fitted(self)  # looks for learned attributes (with trailing _)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

Custom Transformers

Here are a few things to note:

- The sklearn.utils.validation package contains several functions we can use to validate the inputs. For simplicity, we will skip such tests in the rest of this book, but production code should have them.
- Scikit-Learn pipelines require the fit() method to have two arguments X and y, which is why we need the y=None argument even though we don’t use y.

Custom Transformers

Here are a few things to note:

- All Scikit-Learn estimators set n_features_in_ in the fit() method, and they ensure that the data passed to transform() or predict() has this number of features.
- The fit() method must return self.
- This implementation is not 100% complete: all estimators should set feature_names_in_ in the fit() method when they are passed a DataFrame. Moreover, all transformers should provide a get_feature_names_out() method, as well as an inverse_transform() method when their transformation can be reversed. See the last exercise at the end of this chapter for more details.

Custom Transformers

- A custom transformer can (and often does) use other estimators in its implementation. For example, the following code demonstrates custom transformer that uses a KMeans clusterer in the fit() method to identify the main clusters in the training data, and then uses rbf_kernel() in the transform() method to measure how similar each sample is to each cluster center:

Custom Transformers

```python
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, n_init=10,
                              random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```

Tip

- You can check whether your custom estimator respects Scikit-Learn's API by passing an instance to check_estimator() from the sklearn.utils.estimator_checks package.
- For the full API, check out https://scikit-learn.org/stable/developers.

Custom Transformers

- **k-means** is a clustering algorithm that locates clusters in the data. How many it searches for is controlled by the **n_clusters** hyperparameter.
- After training, **the cluster centers are available via the** cluster_centers_ attribute.
- The **fit()** method of KMeans supports an optional argument **sample_weight**, which lets the user specify the **relative weights of the samples**.
- k-means is a **stochastic** algorithm, meaning that it relies on randomness to locate the clusters, so if you want **reproducible** results, you must **set the random_state parameter**.

Custom Transformers

- As you can see, despite the complexity of the task, the code is fairly straightforward. Now let's use this custom transformer:

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]],
                                           sample_weight=housing_labels)
```

- This code creates a ClusterSimilarity transformer, setting the number of clusters to 10. Then it calls fit_transform() with the latitude and longitude of every district in the training set, weighting each district by its median house value. The transformer uses k-means to locate the clusters, then measures the Gaussian RBF similarity between each district and all 10 cluster centers. The result is a matrix with one row per district, and one column per cluster

Custom Transformers

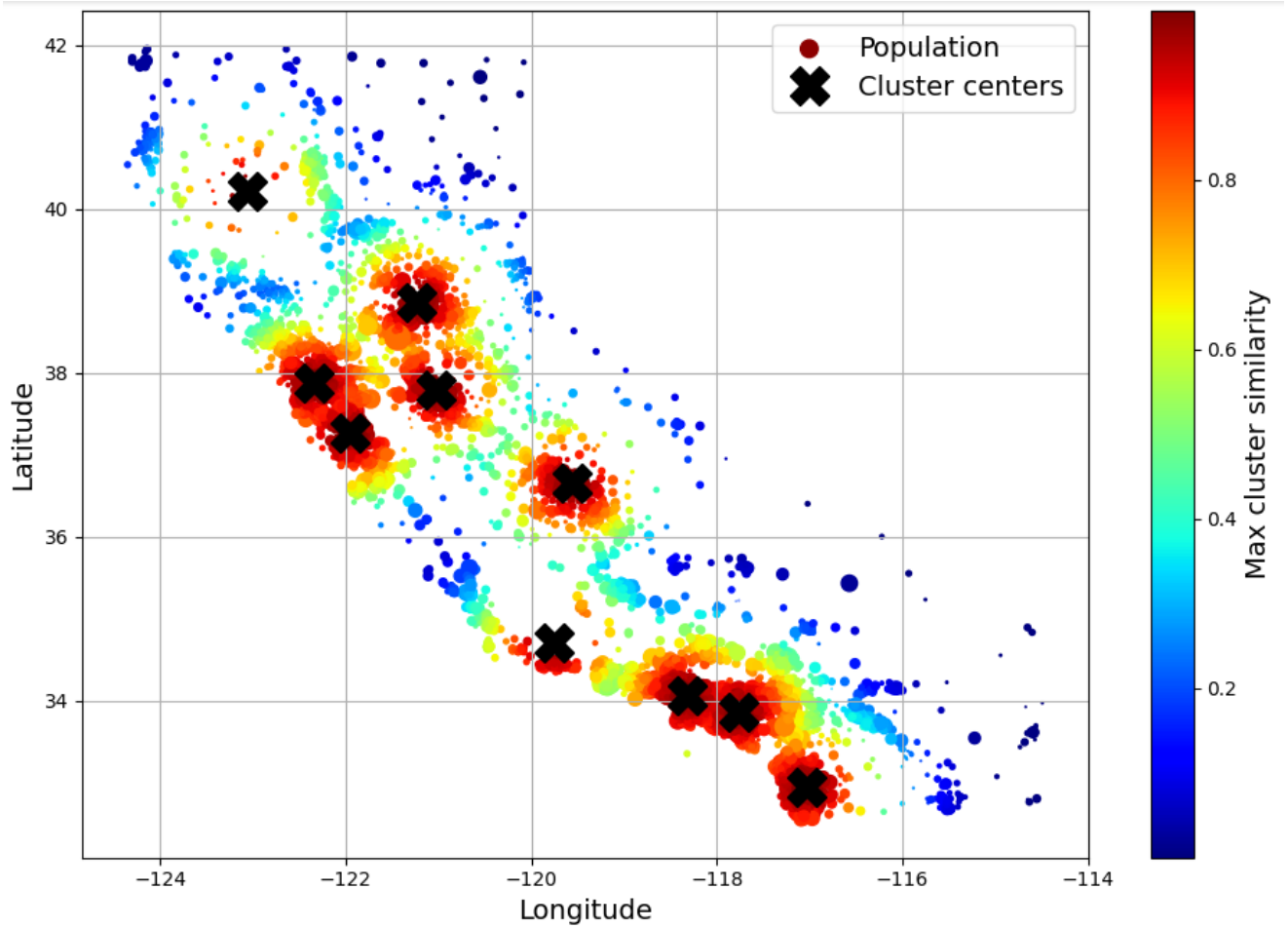
- look at the first three rows, rounding to two decimal places:

```
similarities[:3].round(2)
```

```
array([[0.  , 0.14, 0.  , 0.  , 0.  , 0.08, 0.  , 0.99, 0.  , 0.6 ],
       [0.63, 0.  , 0.99, 0.  , 0.  , 0.  , 0.04, 0.  , 0.11, 0.  ],
       [0.  , 0.29, 0.  , 0.  , 0.01, 0.44, 0.  , 0.7 , 0.  , 0.3 ]])
```

Custom Transformers

- Figure below shows the 10 cluster centers found by k-means. The districts are colored according to their geographic similarity to their closest cluster center. As you can see, most clusters are located in highly populated and expensive areas.



Next Lecture

- Transformation pipelines: **systematic way to assemble** several data *preprocessing and transformation steps* into a single, unified process.
- These pipelines streamline the workflow, **ensuring** that data transformations are **applied** consistently **during both the model training and the prediction** phases.
- They also help in **maintaining clean code and enhance reproducibility**.