that is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

— *Predictors*. Finally, some estimators are capable of making predictions given a dataset; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: it predicted life satisfaction given a country's GDP per capita. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions given a test set (and the corresponding labels in the case of supervised learning algorithms).[17]

- **Inspection**. All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are also accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).
- **Nonproliferation of classes**. Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.
- **Composition**. Existing building blocks are reused as much as possible. For example, it is easy to create a `Pipeline` estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.
- **Sensible defaults**. Scikit-Learn provides reasonable default values for most parameters, making it easy to create a baseline working system quickly.

Handling Text and Categorical Attributes

Earlier we left out the categorical attribute `ocean_proximity` because it is a text attribute so we cannot compute its median. Most Machine Learning algorithms prefer to work with numbers anyway, so let's convert these text labels to numbers.

Scikit-Learn provides a transformer for this task called `LabelEncoder`:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([1, 1, 4, ..., 1, 0, 3])
```

17 Some predictors also provide methods to measure the confidence of their predictions.

This is better: now we can use this numerical data in any ML algorithm. You can look at the mapping that this encoder has learned using the `classes_` attribute ("<1H OCEAN" is mapped to 0, "INLAND" is mapped to 1, etc.):

```
>>> print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. Obviously this is not the case (for example, categories 0 and 4 are more similar than categories 0 and 1). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise), another attribute equal to 1 when the category is "INLAND" (and 0 otherwise), and so on. This is called *one-hot encoding*, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold).

Scikit-Learn provides a `OneHotEncoder` encoder to convert integer categorical values into one-hot vectors. Let's encode the categories as one-hot vectors. Note that `fit_transform()` expects a 2D array, but `housing_cat_encoded` is a 1D array, so we need to reshape it:[18]

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16513x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16513 stored elements in Compressed Sparse Row format>
```

Notice that the output is a SciPy *sparse matrix*, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding we get a matrix with thousands of columns, and the matrix is full of zeros except for one 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. You can use it mostly like a normal 2D array,[19] but if you really want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()
array([[ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
```

18 NumPy's `reshape()` function allows one dimension to be –1, which means "unspecified": the value is inferred from the length of the array and the remaining dimensions.

19 See SciPy's documentation for more details.

```
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

We can apply both transformations (from text categories to integer categories, then from integer categories to one-hot vectors) in one shot using the `LabelBinarizer` class:

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> encoder = LabelBinarizer()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
array([[0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]])
```

Note that this returns a dense NumPy array by default. You can get a sparse matrix instead by passing `sparse_output=True` to the `LabelBinarizer` constructor.

Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), all you need is to create a class and implement three methods: `fit()` (returning `self`), `transform()`, and `fit_transform()`. You can get the last one for free by simply adding `TransformerMixin` as a base class. Also, if you add `BaseEstimator` as a base class (and avoid `*args` and `**kargs` in your constructor) you will get two extra methods (`get_params()` and `set_params()`) that will be useful for automatic hyperparameter tuning. For example, here is a small transformer class that adds the combined attributes we discussed earlier:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
```