

tive than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

WARNING

When creating new combined features, make sure they are not too linearly correlated with existing features; *collinearity* can cause issues with some models, such as linear regression. In particular, avoid simple weighted sums of existing features.

This round of exploration does not have to be absolutely thorough; the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype. But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

Prepare the Data for Machine Learning Algorithms

It's time to prepare the data for your machine learning algorithms. Instead of doing this manually, you should write functions for this purpose, for several good reasons:

- This will allow you to reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset).
- You will gradually build a library of transformation functions that you can reuse in future projects.
- You can use these functions in your live system to transform the new data before feeding it to your algorithms.
- This will make it possible for you to easily try various transformations and see which combination of transformations works best.

But first, revert to a clean training set (by copying `strat_train_set` once again). You should also separate the predictors and the labels, since you don't necessarily

want to apply the same transformations to the predictors and the target values
(note that `drop()` creates a copy of the data and does not affect
`strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Clean the Data

Most machine learning algorithms cannot work with missing features, so you'll need to take care of these. For example, you noticed earlier that the `total_bedrooms` attribute has some missing values. You have three options to fix this:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the missing values to some value (zero, the mean, the median, etc.). This is called *imputation*.

You can accomplish these easily using the Pandas DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

```
housing.dropna(subset=["total_bedrooms"], inplace=True) # option 1

housing.drop("total_bedrooms", axis=1, inplace=True) # option 2

median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"] = housing["total_bedrooms"].fillna(median)
```

You decide to go for option 3 since it is the least destructive, but instead of the preceding code, you will use a handy Scikit-Learn class: `SimpleImputer`. The benefit is that it will store the median value of each feature: this will make it possible to impute missing values not only on the training set, but also on the validation set, the test set, and any new data fed to the model. To use it, first you need to create a

`SimpleImputer` instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.impute import SimpleImputer  
  
imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, you then need to create a copy of the data with only the numerical attributes (this will exclude the text attribute `ocean_proximity`):

```
housing_num = housing.select_dtypes(include=[np.number])
```

Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(housing_num)
```

The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance variable. Only the `total_bedrooms` attribute had missing values, but you cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the `imputer` to all the numerical attributes:

```
>>> imputer.statistics_  
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])  
>>> housing_num.median().values  
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
```

Now you can use this “trained” `imputer` to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

Missing values can also be replaced with the mean value (`strategy="mean"`), or with the most frequent value (`strategy="most_frequent"`), or with a constant value (`strategy="constant"`, `fill_value=...`). The last two strategies support non-numerical data.

TIP

There are also more powerful imputers available in the `sklearn.impute` package (both for numerical features only):

- `KNNImputer` replaces each missing value with the mean of the k -nearest neighbors' values for that feature. The distance is based on all the available features.
 - `IterativeImputer` trains a regression model per feature to predict the missing values based on all the other available features. It then trains the model again on the updated data, and repeats the process several times, improving the models and the replacement values at each iteration.
-

SCIKIT-LEARN DESIGN

Scikit-Learn's API is remarkably well designed. These are the [main design principles](#):⁹

Consistency

All objects share a consistent and simple interface:

Estimators

Any object that can estimate some parameters based on a dataset is called an *estimator* (e.g., a `SimpleImputer` is an estimator). The estimation itself is performed by the `fit()` method, and it takes a dataset as a parameter, or two for supervised learning algorithms—the second dataset contains the labels. Any other parameter needed to guide the estimation process is considered a hyperparameter (such as a `SimpleImputer`'s `strategy`), and it must be set as an instance variable (generally via a constructor parameter).

Transformers

Some estimators (such as a `SimpleImputer`) can also transform a dataset; these are called *transformers*. Once again, the API is simple: the transformation is performed by the `transform()` method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for a `SimpleImputer`. All transformers also have a convenience method called `fit_transform()`, which is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

Predictors

Finally, some estimators, given a dataset, are capable of making predictions; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: given a country's GDP per capita, it predicted life satisfaction. 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).¹⁰

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 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 you will see.

Sensible defaults

Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

Scikit-Learn transformers output NumPy arrays (or sometimes SciPy sparse matrices) even when they are fed Pandas DataFrames as input.¹¹ So, the output of `imputer.transform(housing_num)` is a NumPy array: `X` has neither column names nor index. Luckily, it's not too hard to wrap `X` in a DataFrame and recover the column names and index from `housing_num`:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,  
                           index=housing_num.index)
```

Handling Text and Categorical Attributes

So far we have only dealt with numerical attributes, but your data may also contain text attributes. In this dataset, there is just one: the `ocean_proximity` attribute. Let's look at its value for the first few instances:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(8)
   ocean_proximity
0      NEAR BAY
1    <1H OCEAN
2      INLAND
3      INLAND
4      NEAR OCEAN
5      INLAND
6    <1H OCEAN
7      NEAR BAY
```

It's not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute. Most machine learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's `OrdinalEncoder` class:

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

Here's what the first few encoded values in `housing_cat_encoded` look like:

```
>>> housing_cat_encoded[:8]
array([[3.],
       [0.],
       [1.],
       [1.],
```

```
[4.],  
[1.],  
[0.],  
[3.]])
```

You can get the list of categories using the `categories_` instance variable. It is a list containing a 1D array of categories for each categorical attribute (in this case, a list containing a single array since there is just one categorical attribute):

```
>>> ordinal_encoder.categories_  
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],  
      dtype=object)]
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad”, “average”, “good”, and “excellent”), but it is obviously not the case for the `ocean_proximity` column (for example, categories 0 and 4 are clearly 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). The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors:

```
from sklearn.preprocessing import OneHotEncoder  
  
cat_encoder = OneHotEncoder()  
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

By default, the output of a `OneHotEncoder` is a SciPy *sparse matrix*, instead of a NumPy array:

```
>>> housing_cat_1hot  
<Compressed Sparse Row sparse matrix of dtype 'float64'  
with 16512 stored elements and shape (16512, 5)>
```

A sparse matrix is a very efficient representation for matrices that contain mostly zeros. Indeed, internally it only stores the nonzero values and their positions. When a categorical attribute has hundreds or thousands of categories, one-hot encoding it results in a very large matrix full of 0s except for a single 1 per row. In this case, a sparse matrix is exactly what you need: it will save plenty of memory and speed up computations. You can use a sparse matrix mostly like a normal 2D array,¹² but if you want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()  
array([[0., 0., 0., 1., 0.],  
       [1., 0., 0., 0., 0.],  
       [0., 1., 0., 0., 0.],  
       ...,  
       [0., 0., 0., 0., 1.],  
       [1., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 1.]], shape=(16512, 5))
```

Alternatively, you can set `sparse_output=False` when creating the `OneHotEncoder`, in which case the `transform()` method will return a regular (dense) NumPy array directly:

```
cat_encoder = OneHotEncoder(sparse_output=False)  
housing_cat_1hot = cat_encoder.fit_transform(housing_cat) # now a dense array
```

As with the `OrdinalEncoder`, you can get the list of categories using the encoder's `categories_` instance variable:

```
>>> cat_encoder.categories_  
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
```

```
dtype=object)]
```

Pandas has a function called `get_dummies()`, which also converts each categorical feature into a one-hot representation, with one binary feature per category:

```
>>> df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]})  
>>> pd.get_dummies(df_test)  
ocean_proximity_INLAND  ocean_proximity_NEAR      BAY  
0                      True                  False  
1                      False                 True
```

It looks nice and simple, so why not use it instead of `OneHotEncoder`? Well, the advantage of `OneHotEncoder` is that it remembers which categories it was trained on. This is very important because once your model is in production, it should be fed exactly the same features as during training: no more, no less. Look what our trained `cat_encoder` outputs when we make it transform the same `df_test` (using `transform()`, not `fit_transform()`):

```
>>> cat_encoder.transform(df_test)  
array([[0., 1., 0., 0., 0.],  
       [0., 0., 0., 1., 0.]])
```

See the difference? `get_dummies()` saw only two categories, so it output two columns, whereas `OneHotEncoder` output one column per learned category, in the right order. Moreover, if you feed `get_dummies()` a DataFrame containing an unknown category (e.g., "`<2H OCEAN`"), it will happily generate a column for it:

```
>>> df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]})  
>>> pd.get_dummies(df_test_unknown)  
ocean_proximity_<2H OCEAN  ocean_proximity_ISLAND  
0                      True                  False  
1                      False                 True
```

But `OneHotEncoder` is smarter: it will detect the unknown category and raise an exception. If you prefer, you can set the `handle_unknown` hyperparameter to "ignore", in which case it will just represent the unknown category with zeros:

```
>>> cat_encoder.handle_unknown = "ignore"
>>> cat_encoder.transform(df_test_unknown)
array([[0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

TIP

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the `ocean_proximity` feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you can use one of the encoders provided by the `category_encoders` package on [GitHub](#). Or, when dealing with neural networks, you can replace each category with a learnable, low-dimensional vector called an *embedding* (see [Chapter 14](#)). This is an example of *representation learning* (we will see more examples in [Chapter 18](#)).

When you fit any Scikit-Learn estimator using a DataFrame, the estimator stores the column names in the `feature_names_in_` attribute. Scikit-Learn then ensures that any DataFrame fed to this estimator after that (e.g., to `transform()` or `predict()`) has the same column names. Transformers also provide a `get_feature_names_out()` method that you can use to build a DataFrame around the transformer's output:

```
>>> cat_encoder.feature_names_in_
array(['ocean_proximity'], dtype=object)
>>> cat_encoder.get_feature_names_out()
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',
       'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
```

```
'ocean_proximity_NEAR OCEAN'], dtype=object)
>>> df_output = pd.DataFrame(cat_encoder.transform(df_test_unknown),
...                           columns=cat_encoder.get_feature_names_out(),
...                           index=df_test_unknown.index)
...
...
```

This feature helps avoid column mismatches, and it's also quite useful when debugging.

Feature Scaling and Transformation

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, machine learning algorithms don't perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Without any scaling, most models will be biased toward ignoring the median income and focusing more on the number of rooms.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

WARNING

As with all estimators, it is important to fit the scalers to the training data only: never use `fit()` or `fit_transform()` for anything else than the training set. Once you have a trained scaler, you can then use it to `transform()` any other set, including the validation set, the test set, and new data. Note that while the training set values will always be scaled to the specified range, if new data contains outliers, these may end up scaled outside the range. If you want to avoid this, just set the `clip` hyperparameter to `True`.

Min-max scaling (many people call this *normalization*) is the simplest: for each attribute, the values are shifted and rescaled so that they end up ranging from 0 to 1. This is performed by subtracting the min value from all values, and dividing the results by the difference between the min and the max. Scikit-Learn provides