4.
Prepare the Data for Machine Learning Algorithms

Data Preparation

- Always write functions for data preparation instead of doing it manually.
  - You will gradually build a library of transformation functions that you can reuse in future projects.

- Create a new copy of the dataset and separate the predictors and the labels:

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

Data Cleaning: Missing Values

- Most ML algorithms cannot work with missing features.
  - `total_bedrooms` attribute has some missing values.

```
sample_incomplete_rows =housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	househol
4629	-118.30	34.07	18.0	3759.0	NaN	3296.0	1462
6068	-117.86	34.01	16.0	4632.0	NaN	3038.0	727
17923	-121.97	37.35	30.0	1955.0	NaN	999.0	386
13656	-117.30	34.05	6.0	2155.0	NaN	1039.0	391
19252	-122.79	38.48	7.0	6837.0	NaN	3468.0	1405

Data Cleaning: Missing Values

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	householc
4629	-118.30	34.07	18.0	3759.0	NaN	3296.0	1462
6068	-117.86	34.01	16.0	4632.0	NaN	3038.0	727
17923	-121.97	37.35	30.0	1955.0	NaN	999.0	386

- We have 3 options:
  1. Get rid of the corresponding districts.
```
housing.dropna(subset=["total_bedrooms"])    # option 1
```
  2. Get rid of the whole attribute.
```
housing.drop("total_bedrooms", axis=1)       # option 2
```
  3. Set the values to some value (zero, the mean, the median, etc.).
```
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Data Cleaning: Missing Values

- Scikit-Learn provides a handy class to take care of missing values: `SimpleImputer`.
- Create a `SimpleImputer` instance, specifying that you want to replace each attribute's missing values with its median:

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

- Create a copy of the data without the text attribute `ocean_proximity`:

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

- Fit the imputer instance to the training data:

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

Data Cleaning: Missing Values

- The imputer has simply computed the median of each attribute and stored the result in its `statistics_` instance variable

```
imputer.statistics_
```

```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

- Transform the training set:

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

- The result is a plain NumPy array containing the transformed features. Put it back into a pandas DataFrame:

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

Data Cleaning: Missing Values

- Other strategies for `SimpleImputer`:
  - `(strategy="mean")`
  - `(strategy="most_frequent")`
  - `(strategy="constant", fill_value=…)`

- More powerful imputers in `sklearn.impute` package:
  - `KNNImputer` replaces each missing value with the mean of the k-nearest neighbors' values for that feature.
  - `IterativeImputer` trains a regression model per feature to predict the missing values based on all the other available features.

SCIKIT-LEARN Design

- Estimators: any object that can estimate some parameters based on a dataset.
  - The estimation itself is performed by the `fit()` method.
- Transformer: estimators that can also transform a dataset.
  - The transformation is performed by the `transform()` method.
- Predictors: estimators that, given a dataset, are capable of making predictions.
  - It has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions.
  - It has a `score()` method that measures the quality of the predictions, given a test set.

Handling Text and Categorical Attributes

- In this dataset, there is one categorical attribute: `ocean_proximity`

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(8)
```

	ocean_proximity
13096	NEAR BAY
14973	<1H OCEAN
3785	INLAND
14689	INLAND
20507	NEAR OCEAN
1286	INLAND
18078	<1H OCEAN
4396	NEAR BAY

Handling Text and Categorical Attributes

➢ `ocean_proximity` is categorical, but most ML algorithms prefer to work with numbers, so we convert these categories from text to numbers using `OrdinalEncoder` class:

```
from sklearn.preprocessing import OrdinalEncoder

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

```
housing_cat_encoded[:8]
```

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

```
ordinal_encoder.categories_
```

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

One-Hot Encoding

- *One-hot encoding*: create one binary attribute per category.
- Use `OneHotEncoder` class to convert categorical values into one-hot vectors:

```python
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

```python
housing_cat_1hot
```

```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
	with 16512 stored elements in Compressed Sparse Row format>
```

One-Hot Encoding

- The `OneHotEncoder` class returns a sparse array, but we can convert it to a dense array if needed by calling 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.]])
```

- You can also set `sparse=False` when creating the `OneHotEncoder`:

```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
```

One-Hot Encoding

- If a categorical attribute has a large number of categories, one-hot encoding results in a large number of input features.
  - This may slow down training and degrade performance.
- Solution 1: replace the categorical input with useful numerical features related to the categories.
  - Replace `ocean_proximity` feature with the distance to the ocean.
- Solution 2: in dealing with neural networks, replace each category with a learnable, low-dimensional vector called an *embedding*.
  - This is an example of *representation learning*.

Feature Scaling

- Most of ML algorithms do not perform well when the input numerical attributes have very different scales.

- Min-max scaling (aka *normalization*): subtract the min value and divide by the max minus the min.
  - ScikitLearn provides a transformer called `MinMaxScaler` for this.
  - It has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0–1.

```python
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

Feature Scaling

- Standardization: subtract the mean value, and divide by the standard deviation. The resulting distribution has zero mean and unit variance.
  - Scikit-Learn provides a transformer called `StandardScaler` for this.

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

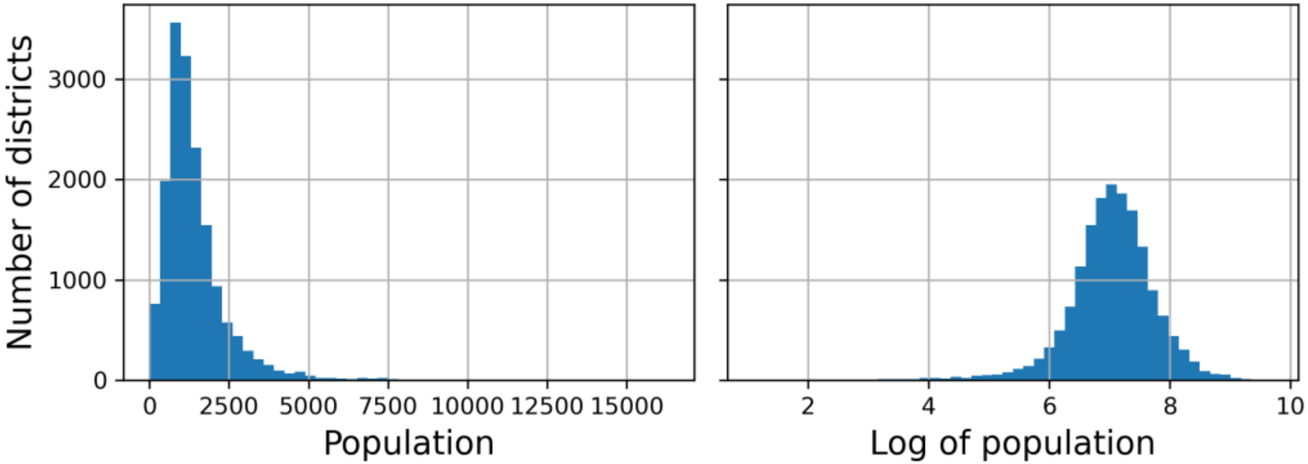
- Standardization does not restrict values to a specific range and it is much less affected by outliers.
- To scale a sparse matrix without converting it to a dense matrix, you can use a `StandardScaler` with its `with_mean` hyperparameter set to False.
  - only divides the data by the standard deviation, without subtracting the mean.

Feature Scaling

- 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: the validation set, the test set, and new data.
- When a feature's distribution has a *heavy tail*, both min-max scaling and standardization will squash most values into a small range.
  - Machine learning models generally don't like this at all.
  - Before you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the distribution more symmetrical.

Transformation before Scaling

- If the feature has a really long and heavy tail, such as a *power law distribution*, then replacing the feature with its logarithm may help.



Inverse Transform

- In addition to the input features, the target values may also need to be transformed.
  - The ML model will now predict the *transformed* target value.
- Most of Scikit-Learn's transformers have an `inverse_transform()` method, to compute the inverse of their transformations easily.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

Custom Transformers

- Write your own transformers for tasks such as custom cleanup operations or combining specific attributes.
- Scikit-Learn relies on duck typing (not inheritance), all you need to do is create a class and implement three methods: `fit()` (returning self), `transform()`, and `fit_transform()`.
- You can get `fit_transform` by simply adding `TransformerMixin` as a base class.
- If you add `BaseEstimator` as a base class you will also get two extra methods (`get_params()` and `set_params()`)

Custom Transformers

```python
from sklearn.base import BaseEstimator, TransformerMixin

# column index
rooms_ix, bedrooms_ix, population_ix, households_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):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

Transformation Pipelines

- There are many data transformation steps that need to be executed in the right order. Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations.

```python
from sklearn.pipeline import Pipeline
num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

- If you don't want to name the transformers, you can use the `make_pipeline()` function instead:

```python
from sklearn.pipeline import make_pipeline
num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

Transformation Pipelines

- It would be convenient to have a single transformer capable of handling all columns, applying the appropriate transformations to each column. For this, you can use a `ColumnTransformer`.

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```