Data Preprocessing

APT 3025: APPLIED MACHINE LEARNING

Lecture Overview

- We inspect the data and plot visualisations to see where there may be problems
- We carry out the following transformations:
- Combine features
- Fill in gaps in the data
- Feature scaling
- Convert categorical features to numeric
- Then we look at transformation pipelines

Steps in a Machine Learning Project

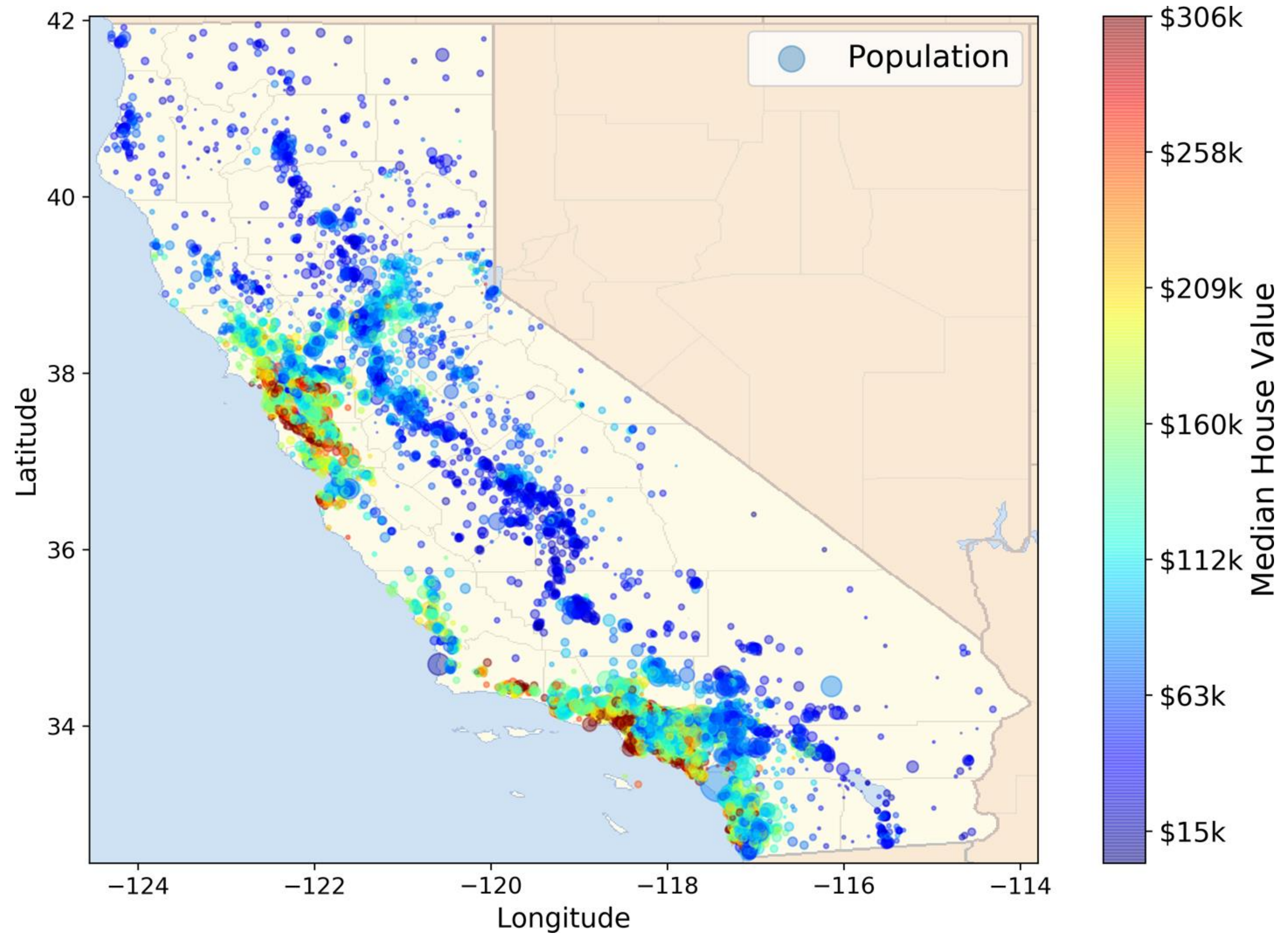
1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Sources of Real Data

- It’s a good idea to use real data when experimenting with machine learning algorithms. Here are some sources of open data:
- Popular open data repositories
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon’s AWS datasets
- Meta portals (they list open data repositories)
  - Data Portals
  - OpenDataMonitor
  - Quandl
- Other pages listing many popular open data repositories
  - Wikipedia’s list of Machine Learning datasets
  - Quora.com
  - The datasets subreddit

The California Housing Prices Dataset

- In this lecture we’ll use the California Housing Prices dataset.
- This dataset is based on data from the 1990 California census, but we will modify it slightly for our purposes.


Population
Latitude
Longitude
Median House Value
42
40
38
36
34
−124
−122
−120
−118
−116
−114
$306k
$258k
$209k
$160k
$112k
$63k
$15k

Look at the big picture

- This data includes metrics such as the population, median income, and median housing price for each block group in California.
- Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will call them “districts” for short.
- The model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

What kind of problems is this?

- This is a multiple regression problem, since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.).
- It is also a univariate regression problem, since we are only trying to predict a single value for each district.
- If we were trying to predict multiple values per district, it would be a multivariate regression problem.

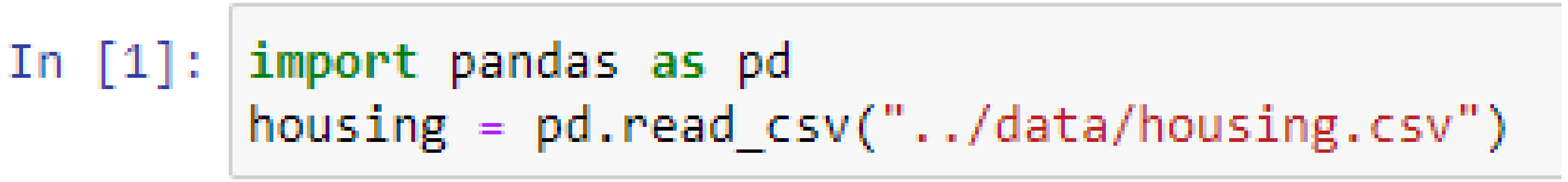
Select a Performance Measure

- Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (RMSE).
- It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors.

$$\mathrm{RMSE}\left(\mathbf{X}, h\right) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

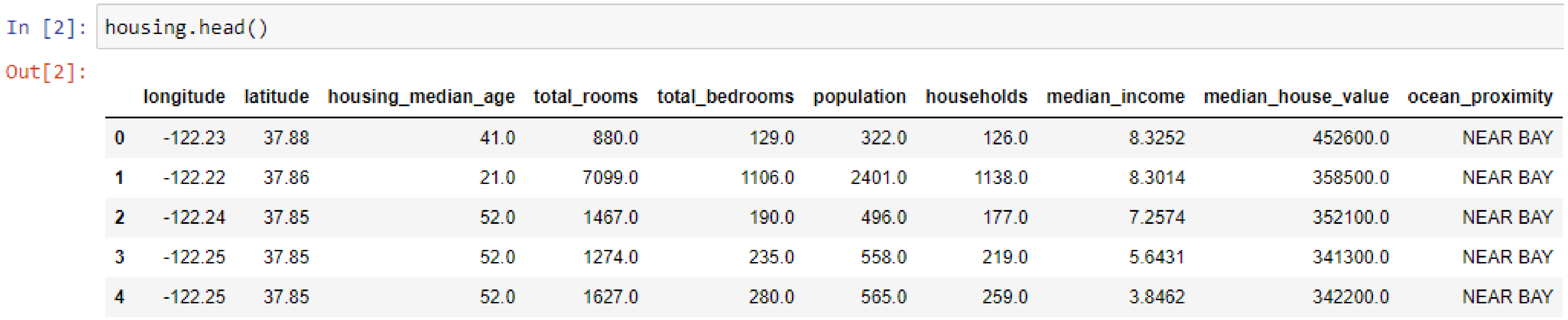
Get the Data

- Go to the link below to get the housing data. You will need to extract it, e.g. using 7-zip
  - https://github.com/ageron/handson-ml/blob/master/datasets/housing/housing.tgz
- Extract and put it in your data folder
- Load the data using pandas:



```
In [1]: import pandas as pd
        housing = pd.read_csv("../data/housing.csv")
```

Inspect the Data



In [2]:
```
housing.head()
```

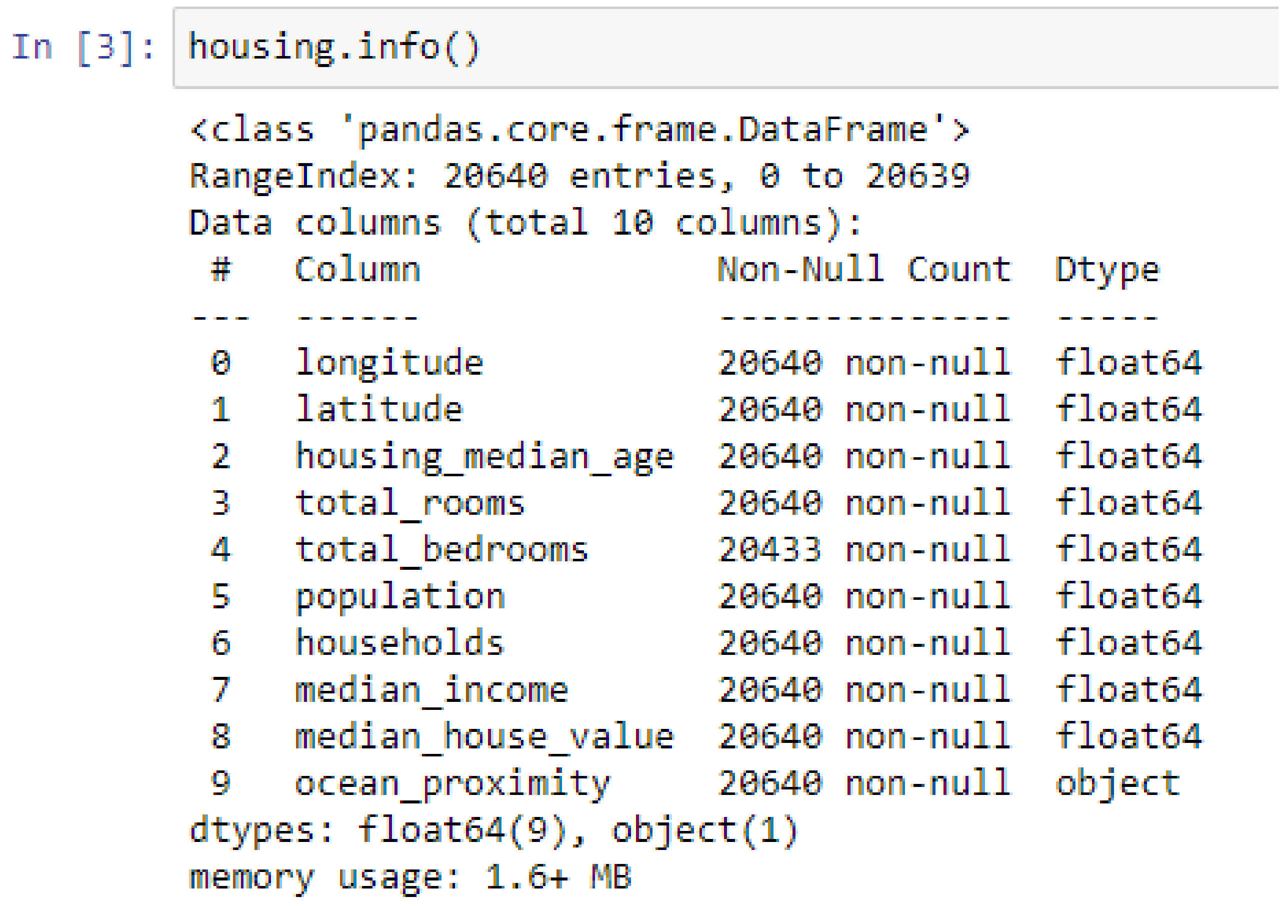
Out[2]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

- Each row represents one district, and there are ten attributes describing a district.

Inspect the data

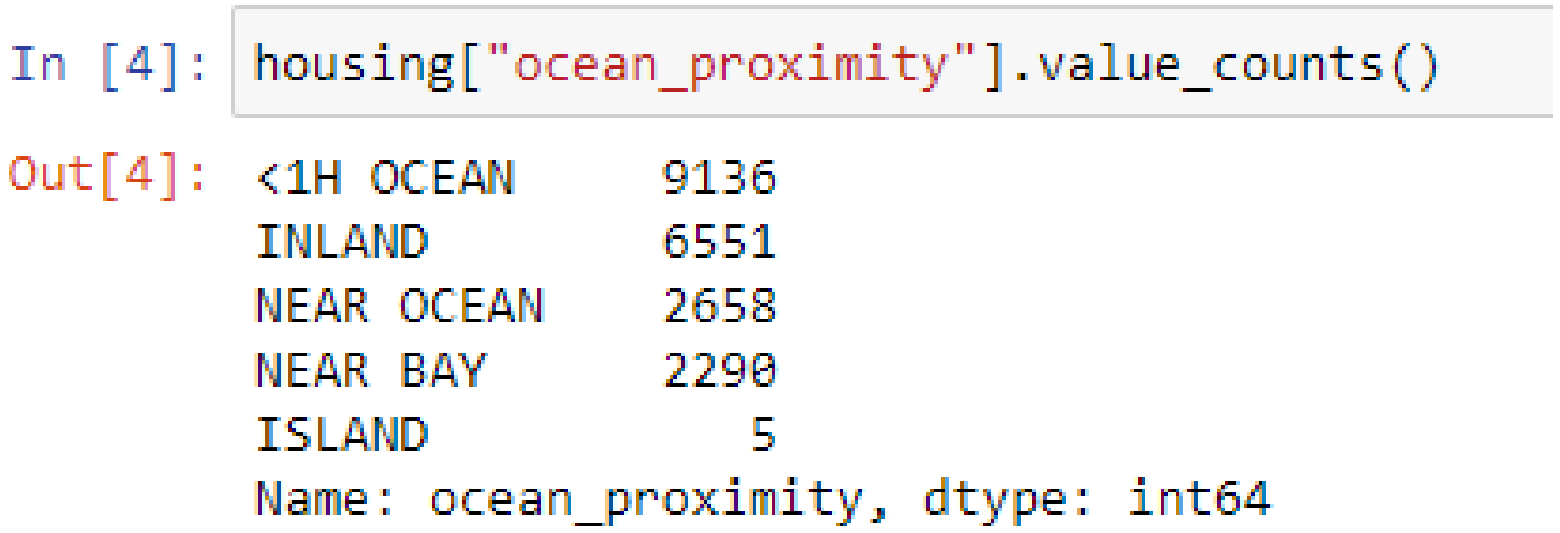
- The info() method is useful to get a quick description of the data, in particular the total number of rows, each attribute’s type, and the number of nonnull values



```
In [3]: housing.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Inspect the Data

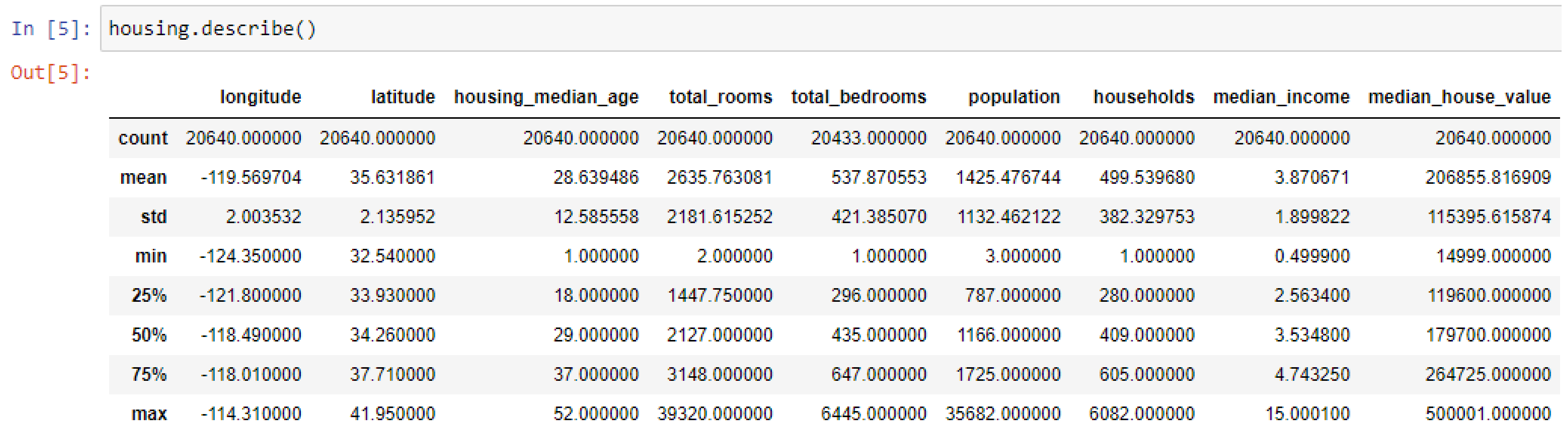
- You can find out what categories exist under ocean_proximity, and how many districts belong to each category by using the value_counts() method:



```
In [4]: housing["ocean_proximity"].value_counts()

Out[4]: <1H OCEAN     9136
        INLAND        6551
        NEAR OCEAN    2658
        NEAR BAY      2290
        ISLAND           5
        Name: ocean_proximity, dtype: int64
```

Inspect the Data

- The describe() method shows a summary of the numerical attributes.



```
In [5]: housing.describe()
```

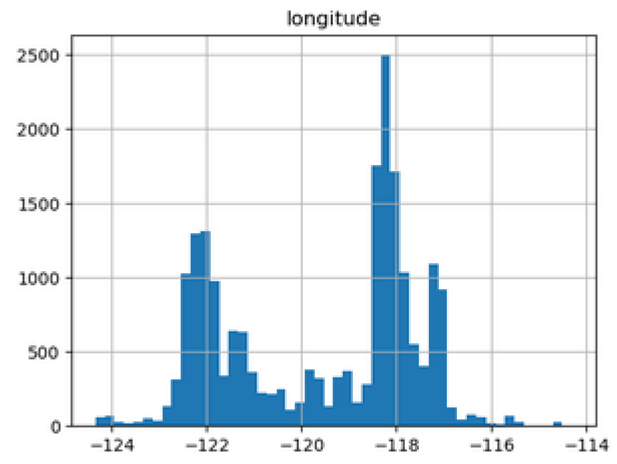
Out[5]:

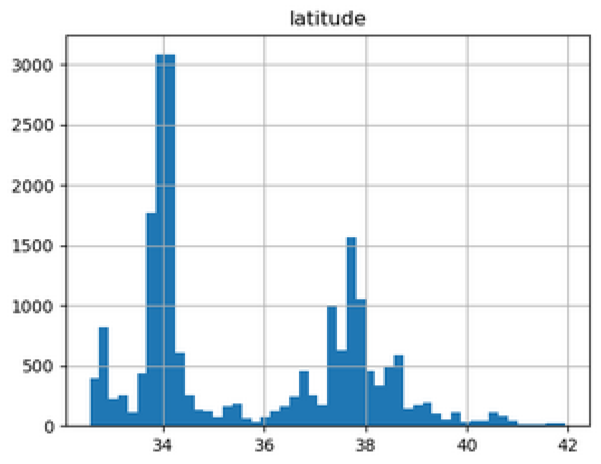
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

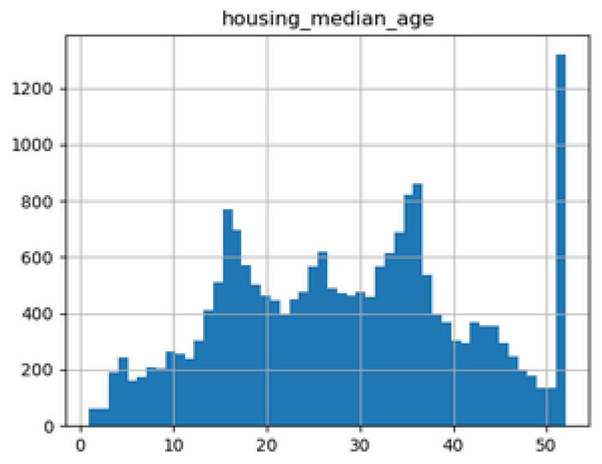
Inspect the Data

- Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. Here we show 3 histograms, but there are 9, one for each numeric attribute.

```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20, 15))
plt.show()
```







Research the Data

- Not all information about the data is available in the dataset. You may need to research the things that are unclear. For example an annual household income of USD 8 does not make sense.
- After doing some research you learn that the data has been scaled and capped at ~15 for higher median incomes, and at ~0.5 for lower median incomes.
- The numbers represent roughly tens of thousands of dollars (e.g., 3 actually means about $30,000).

Research the Data

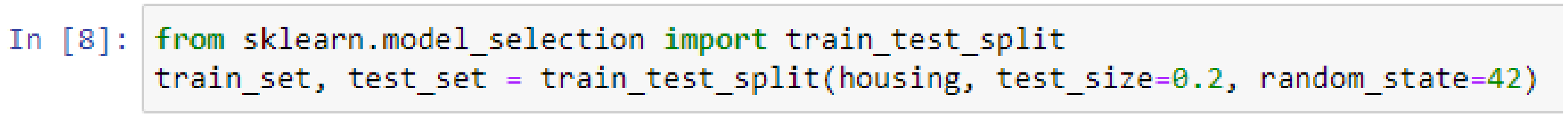
- The housing median age and the median house value were also capped.
- The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit.
- If the project requires precise predictions even beyond $500,000, then you have two options:
    - Collect proper labels for the districts whose labels were capped.
    - Remove those districts from the training and sets

Inspecting the Data

- Notice that the attributes have very different scales. This can be a problem for machine learning and is dealt with using feature scaling.
- Many histograms are tail-heavy: they extend much farther to the right of the median than to the left.
- This may make it a bit harder for some Machine Learning algorithms to detect patterns.
- We could try to transform these distributions to be more bell-shaped, but this is beyond the scope of this course.

Separate out the test set early

- If you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model.
- When you estimate the generalization error using the test set, your estimate will be too optimistic, and you will launch a system that will not perform as well as expected.
- This is called data snooping bias.



```
In [8]: from sklearn.model_selection import train_test_split
        train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Avoiding Sampling Bias

- Purely random sampling is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias.
- When a survey company decides to call 1,000 people to ask them a few questions, they don’t just pick 1,000 people randomly in a phone book.
- They try to ensure that these 1,000 people are representative of the whole population.

Stratified Sampling

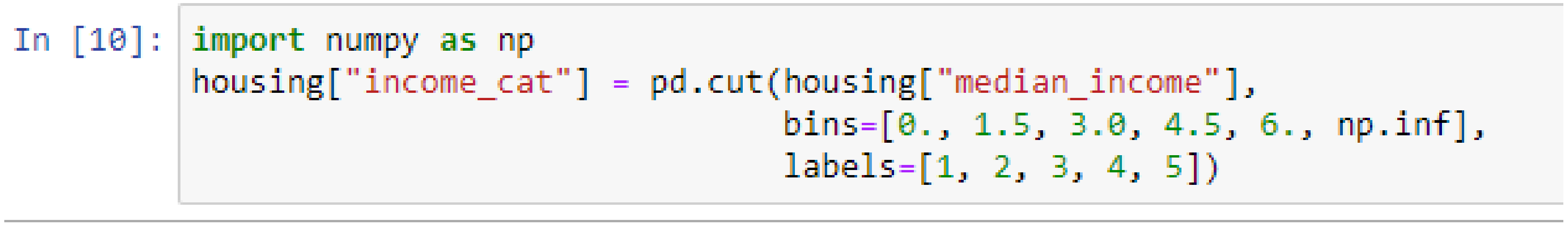
- This is called stratified sampling: the population is divided into homogeneous subgroups called strata, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population.

Stratified Sampling

- Suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices.
- You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset.
- Since the median income is a continuous numerical attribute, you first need to create an income category attribute.

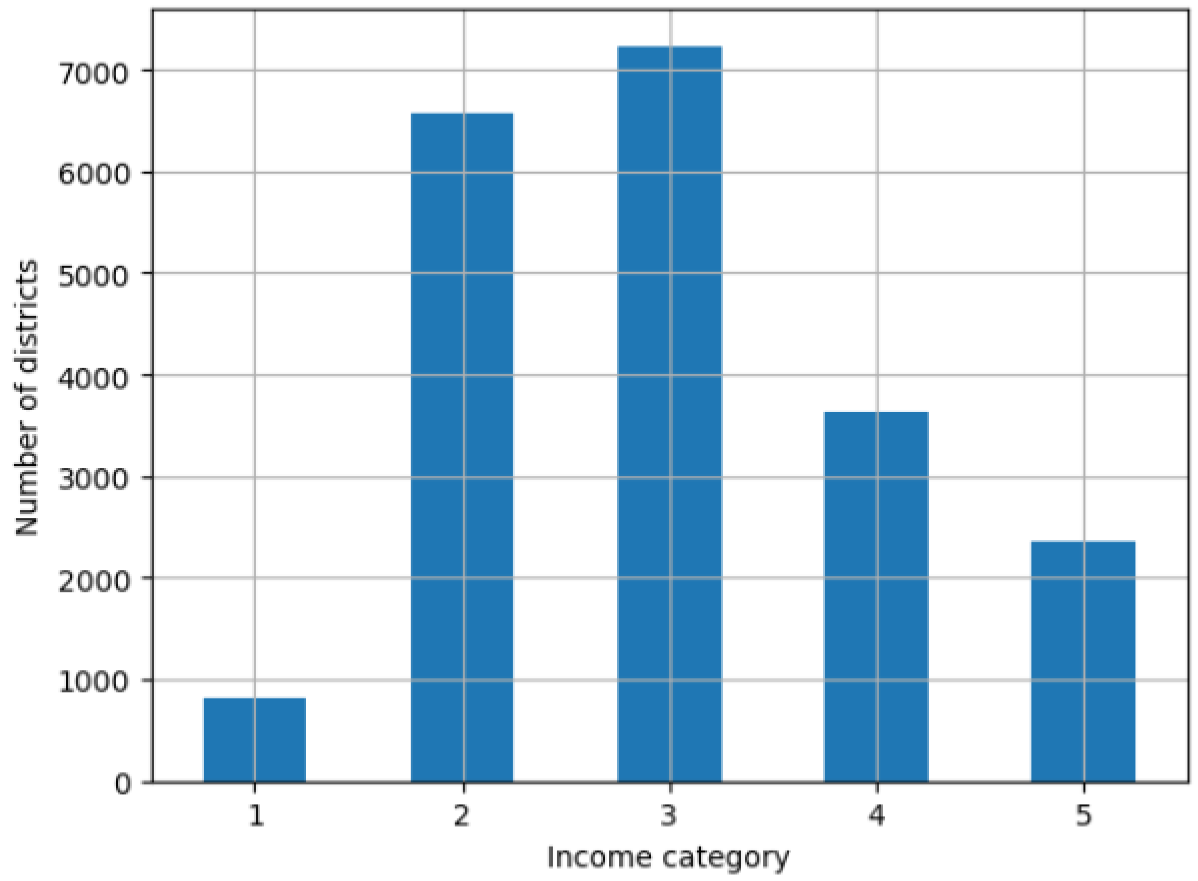
Stratified Sampling

- The following code uses the pd.cut() function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than $15,000), category 2 from 1.5 to 3, and so on:



```
In [10]: import numpy as np
         housing["income_cat"] = pd.cut(housing["median_income"],
                                        bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                        labels=[1, 2, 3, 4, 5])
```

```
housing['income_cat'].value_counts().sort_index().plot.bar(rot=0, grid=True)
plt.xlabel('Income category')
plt.ylabel('Number of districts')
plt.show()
```

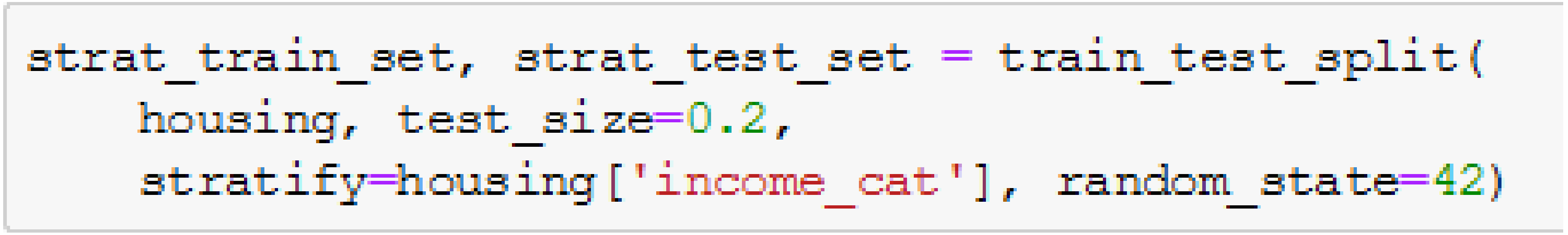


Stratified Sampling

- This histogram plot shows that most median income values are clustered around 1.5 to 6 (i.e., $15,000–$60,000), but some median incomes go far beyond 6.
- It is important to have a sufficient number of instances in your dataset for each stratum.
- This means that you should not have too many strata, and each stratum should be large enough.

Stratified Sampling

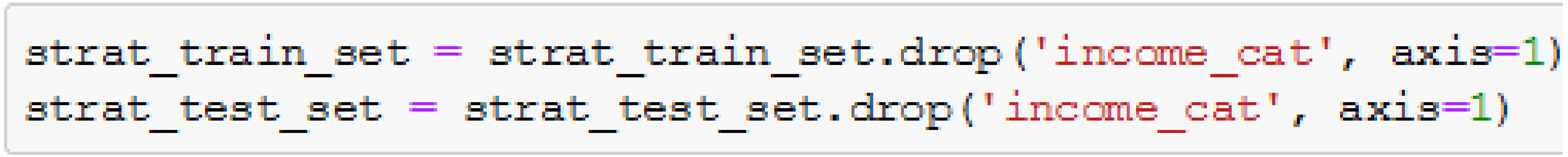
- Now you are ready to do stratified sampling based on the income category. For this we can use Scikit-Learn’s train_test_split, which we imported earlier from sklearn.model_selection.



```
strat_train_set, strat_test_set = train_test_split(
    housing, test_size=0.2,
    stratify=housing['income_cat'], random_state=42)
```

Remove income category field

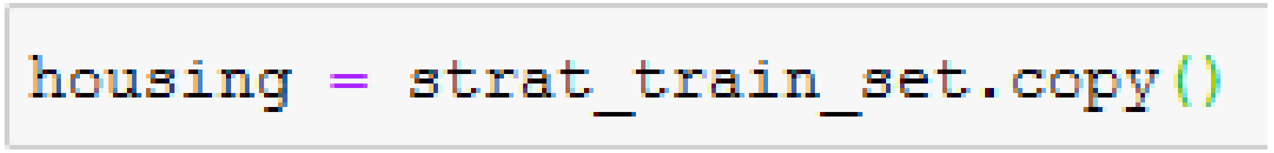
- Now you should remove the income_cat attribute so the data is back to its original state:



```
strat_train_set = strat_train_set.drop('income_cat', axis=1)
strat_test_set = strat_test_set.drop('income_cat', axis=1)
```

Visualize the Data

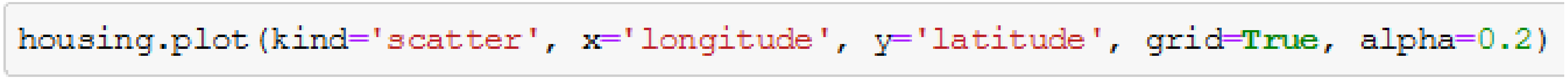
- First, make sure you have put the test set aside and you are only exploring the training set.
- Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast.
- In our case, the set is quite small, so you can just work directly on the full set. Let’s create a copy so that you can play with it without harming the training set:



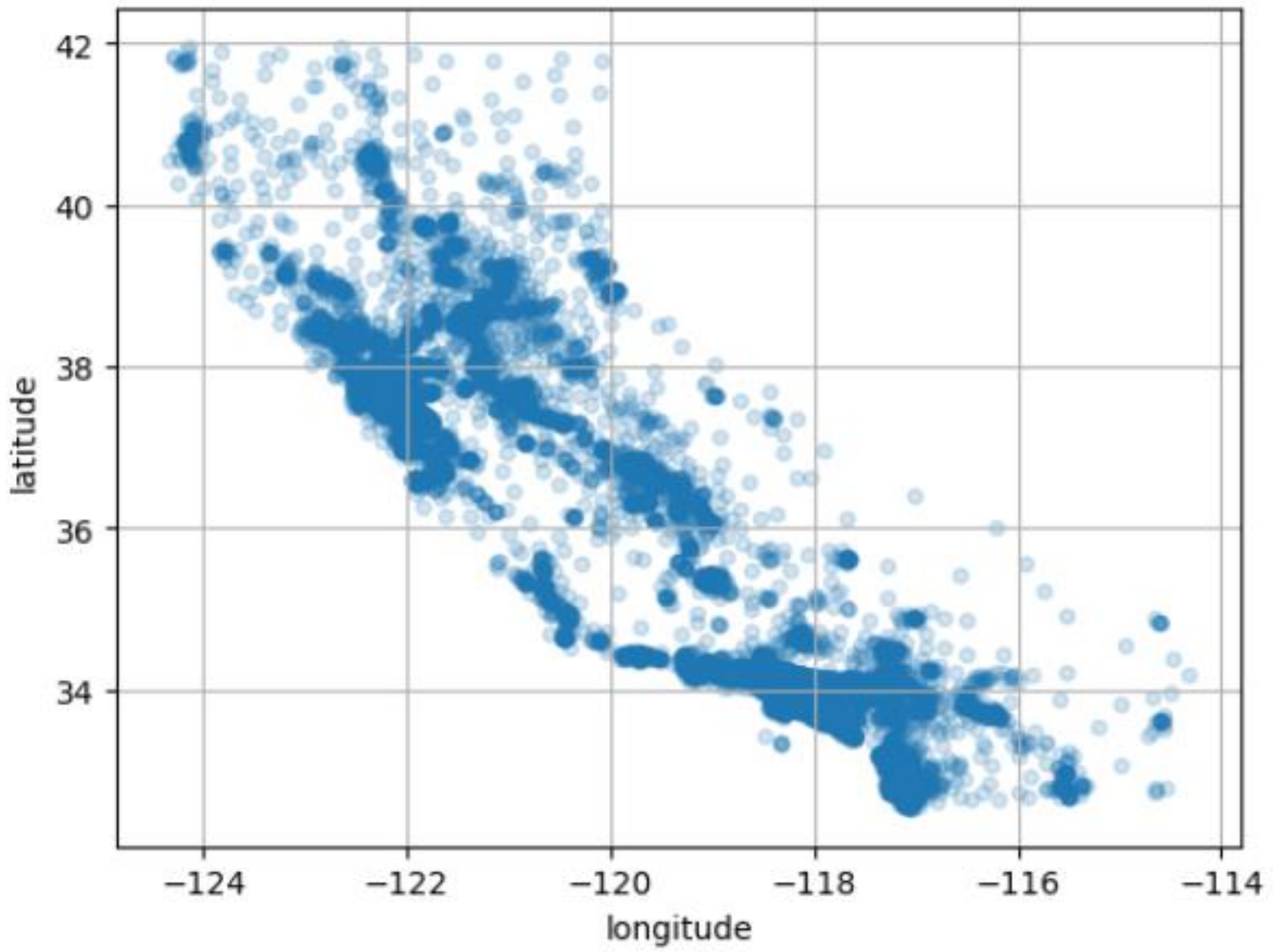
```
housing = strat_train_set.copy()
```

Visualize the Data

- Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data.
- Setting the alpha option to 0.2 makes it easier to visualize the places where there is a high density of data points

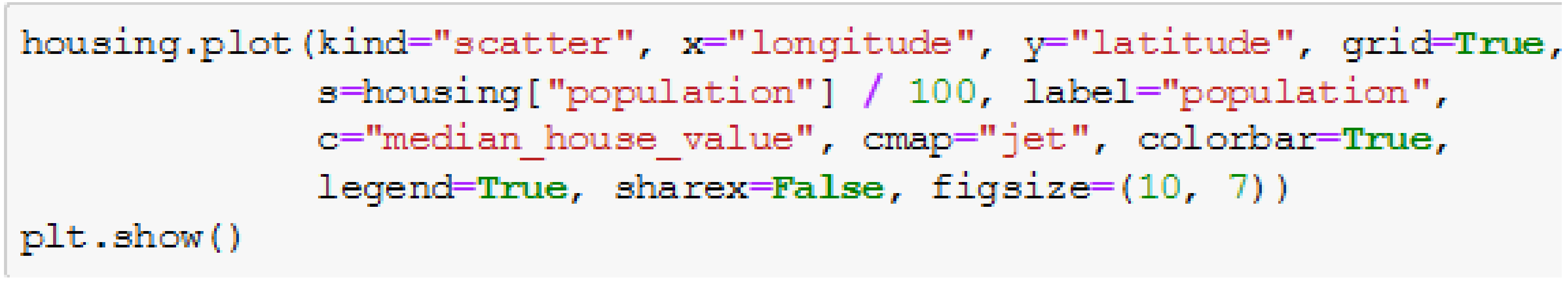


```
housing.plot(kind='scatter', x='longitude', y='latitude', grid=True, alpha=0.2)
```

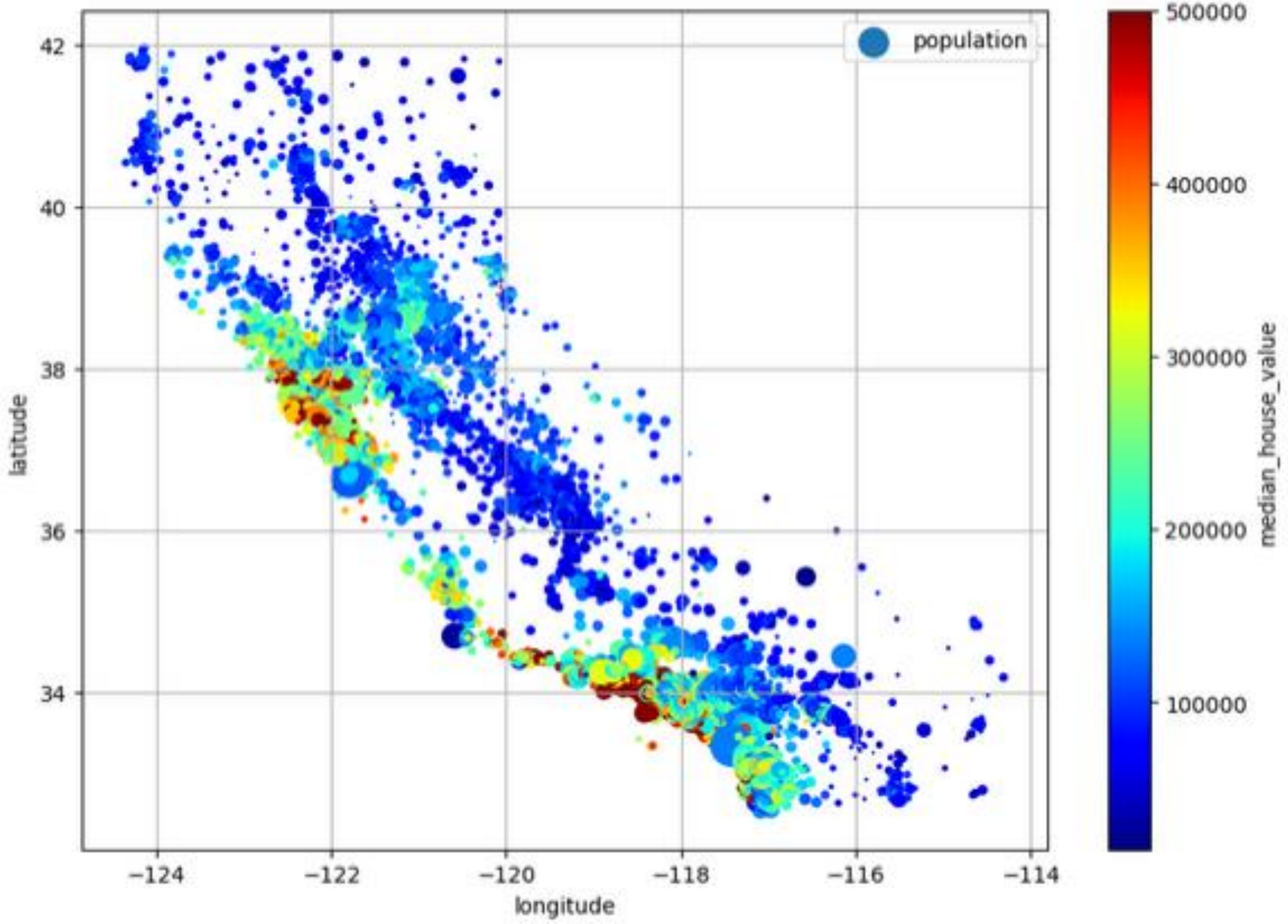

42
40
38
36
34
latitude
−124
−122
−120
−118
−116
−114
longitude

Visualizing

- Now let’s look at the housing prices next.
- The radius of each circle represents the district’s population (option s), and the color represents the price (option c).
- We will use a predefined color map (option cmap) called jet, which ranges from blue (low values) to red (high prices).
- This image tells you that the housing prices are very much related to the location.



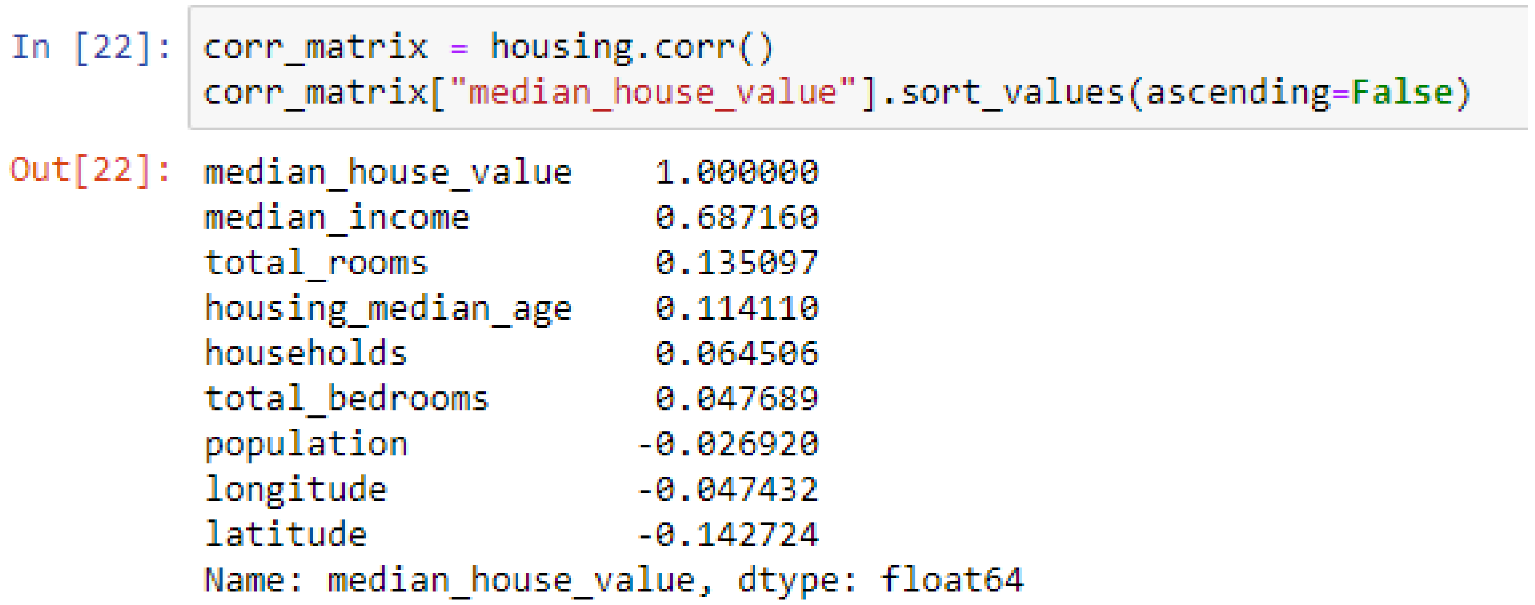
```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))
plt.show()
```



population
latitude
longitude
median_house_value
42
40
38
36
34
−124
−122
−120
−118
−116
−114
500000
400000
300000
200000
100000

Examine Correlations

- Since the dataset is not too large, you can easily compute the standard correlation coefficient (also called Pearson's r) between every pair of attributes using the corr() method:



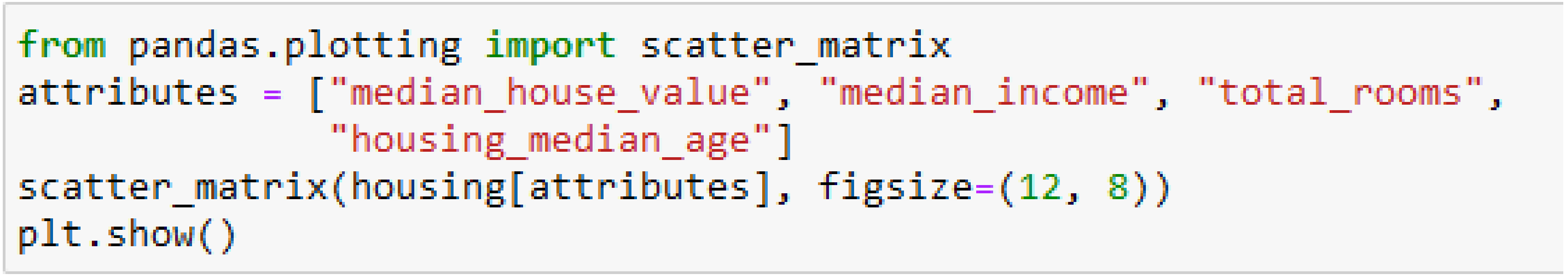
```
In [22]: corr_matrix = housing.corr()
         corr_matrix["median_house_value"].sort_values(ascending=False)

Out[22]: median_house_value    1.000000
         median_income         0.687160
         total_rooms           0.135097
         housing_median_age    0.114110
         households            0.064506
         total_bedrooms        0.047689
         population           -0.026920
         longitude            -0.047432
         latitude             -0.142724
         Name: median_house_value, dtype: float64
```

Examine Correlations

- The correlation coefficient ranges from –1 to 1.
- When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up.
- When the coefficient is close to –1, it means that there is a strong negative correlation; you can see a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when you go north).

Examining Correlations with Scatter Plots

- Another way to check for correlation between attributes is to use the pandas scatter_matrix() function, which plots every numerical attribute against every other numerical attribute.
- Since there are now 9 numerical attributes, you would get 81 plots, which would not fit on a page—so let's just focus on a few promising attributes that seem most correlated with the median housing value

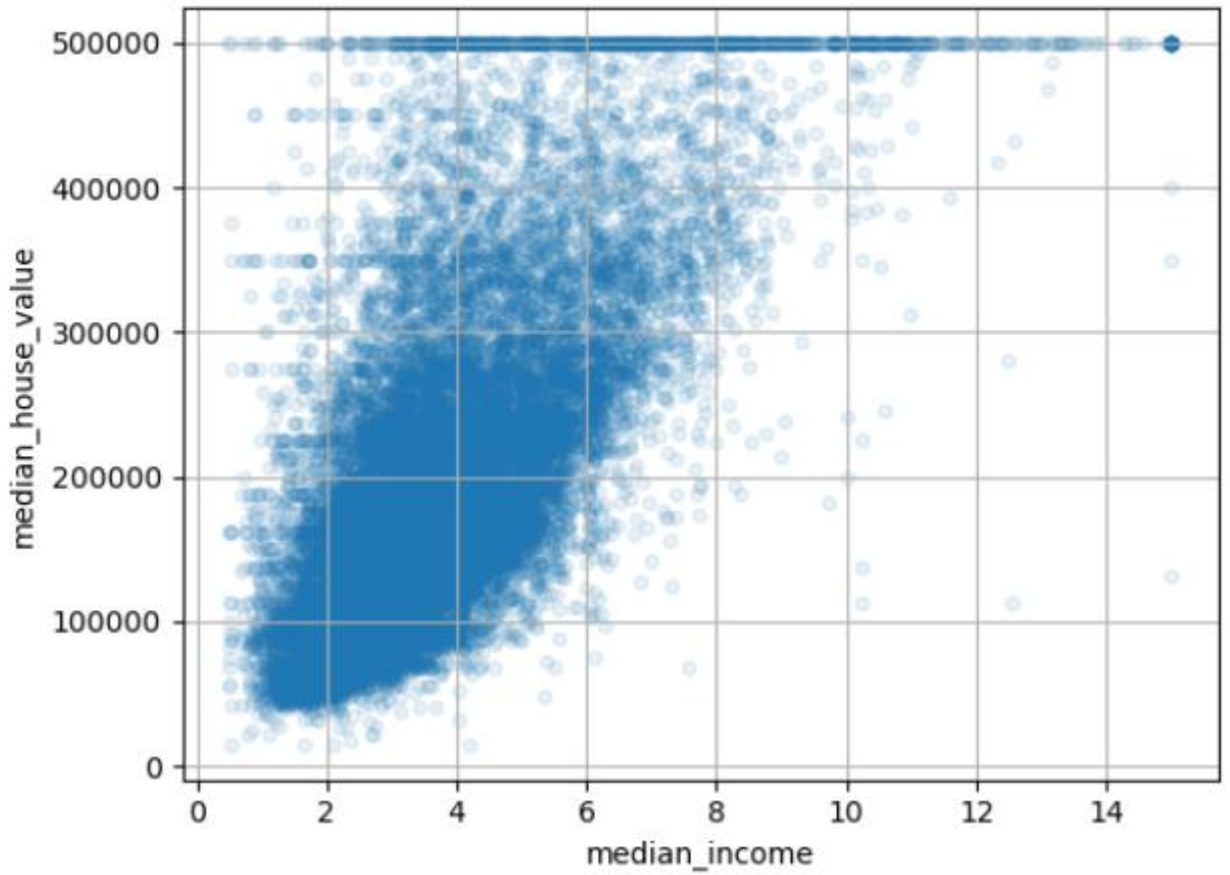


```python
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```

Median Income

- Median income seems to have the strongest correlation with our target attribute so let’s zoom in on it.

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1, grid=True)
plt.show()
```



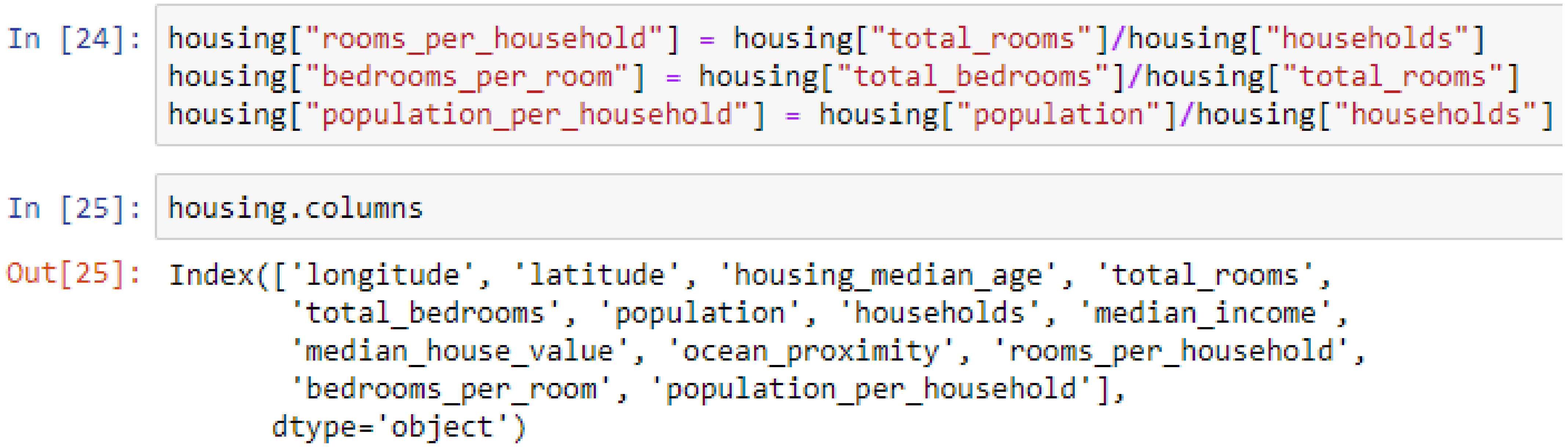
What the plot reveals

- This plot reveals that the correlation is indeed very strong; you can clearly see the upward trend, and the points are not too dispersed.
- Second, the price cap that we mentioned earlier is clearly visible as a horizontal line at $500,000.
- But this plot reveals other less obvious straight lines: a horizontal line around $450,000, another around $350,000, perhaps one around $280,000, and a few more below that.
- You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

Experimenting with Attribute Combinations

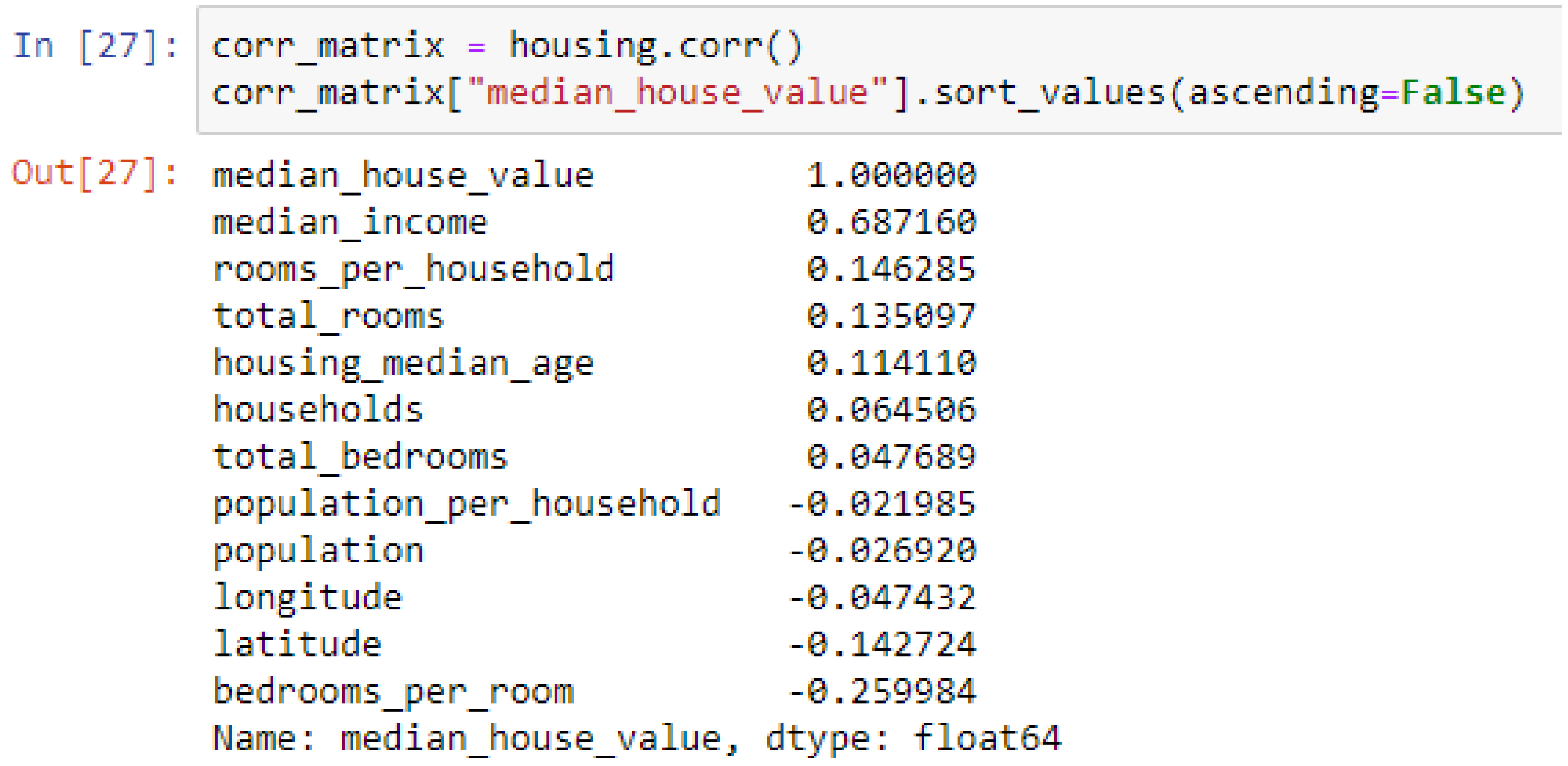
- Some attributes have little value until they are combined with others.
- For example, the total number of rooms in a district is not very useful if you don’t know how many households there are. What you really want is the number of rooms per household.
- Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms.
- And the population per household also seems like an interesting attribute combination to look at.

Experimenting with Attribute Combinations



```
In [24]: housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
         housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
         housing["population_per_household"] = housing["population"]/housing["households"]

In [25]: housing.columns

Out[25]: Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
                'total_bedrooms', 'population', 'households', 'median_income',
                'median_house_value', 'ocean_proximity', 'rooms_per_household',
                'bedrooms_per_room', 'population_per_household'],
               dtype='object')
```

Experimenting with Attribute Combinations



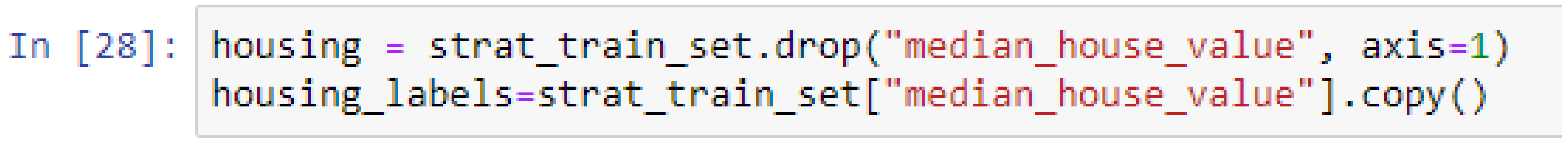
```
In [27]: corr_matrix = housing.corr()
         corr_matrix["median_house_value"].sort_values(ascending=False)

Out[27]: median_house_value          1.000000
         median_income               0.687160
         rooms_per_household         0.146285
         total_rooms                 0.135097
         housing_median_age          0.114110
         households                  0.064506
         total_bedrooms              0.047689
         population_per_household   -0.021985
         population                 -0.026920
         longitude                  -0.047432
         latitude                   -0.142724
         bedrooms_per_room          -0.259984
         Name: median_house_value, dtype: float64
```

Comparing Correlations

- From the correlation matrix, we see that some of the combined attributes offer new insights.
- We see that the new rooms per household attribute has a stronger positive correlation with the target than rooms alone or bedrooms alone.
- We also see that bedrooms per room has a significant negative correlation with the target. That is, a higher bedrooms to rooms ratio suggests a lower house value.

Prepare the Data for Machine Learning

- Let’s revert to a clean training set (by copying strat_train_set once again; we’re abandoning the combined attributes).
- Let’s also separate the predictors and the labels, since we don’t necessarily want to apply the same transformations to the predictors and the target values:



```
In [28]: housing = strat_train_set.drop("median_house_value", axis=1)
         housing_labels=strat_train_set["median_house_value"].copy()
```

Cleaning the Data

- We saw earlier that the total_bedrooms attribute has some missing values, so let’s fix this.
- You have three options:
  - Get rid of the corresponding districts.
  - Get rid of the whole attribute.
  - Set the values to some value (zero, the mean, the median, etc.).

Cleaning the Data

- You can accomplish these easily using DataFrame’s dropna(), drop(), and fillna() methods.

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

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

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

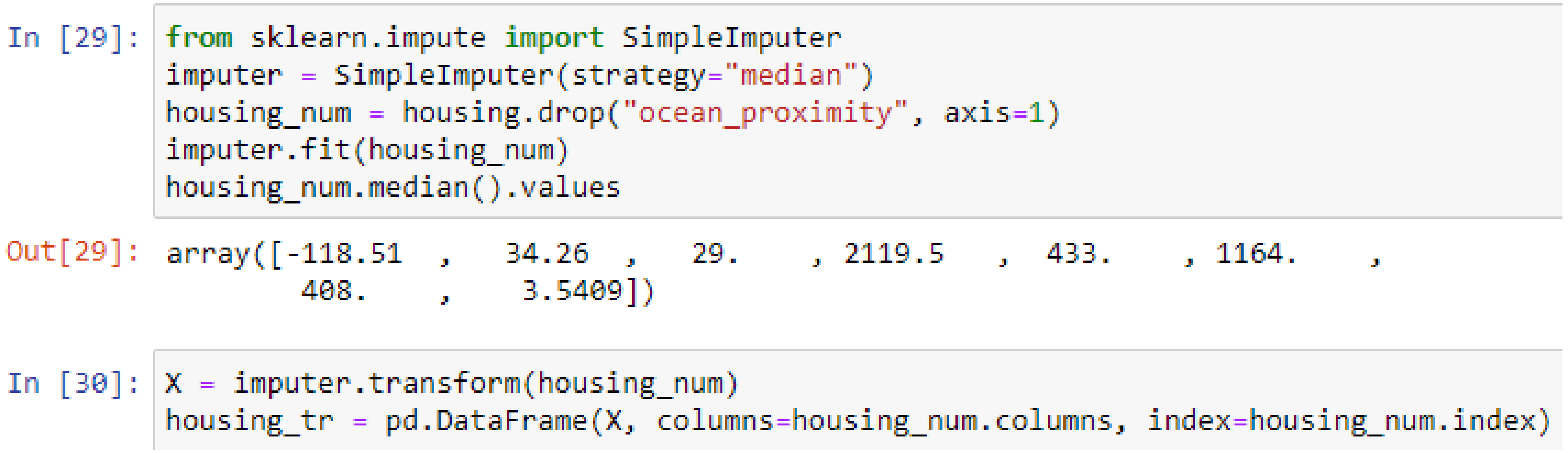
Cleaning the Data

- Let’s go for option 3 since it is the least destructive, but instead of the code above, we 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.

Cleaning the Data using Scikit-Learn

- 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:
- Since the median can only be computed on numerical attributes, you need to create a copy of the data without the text attribute ocean_proximity.

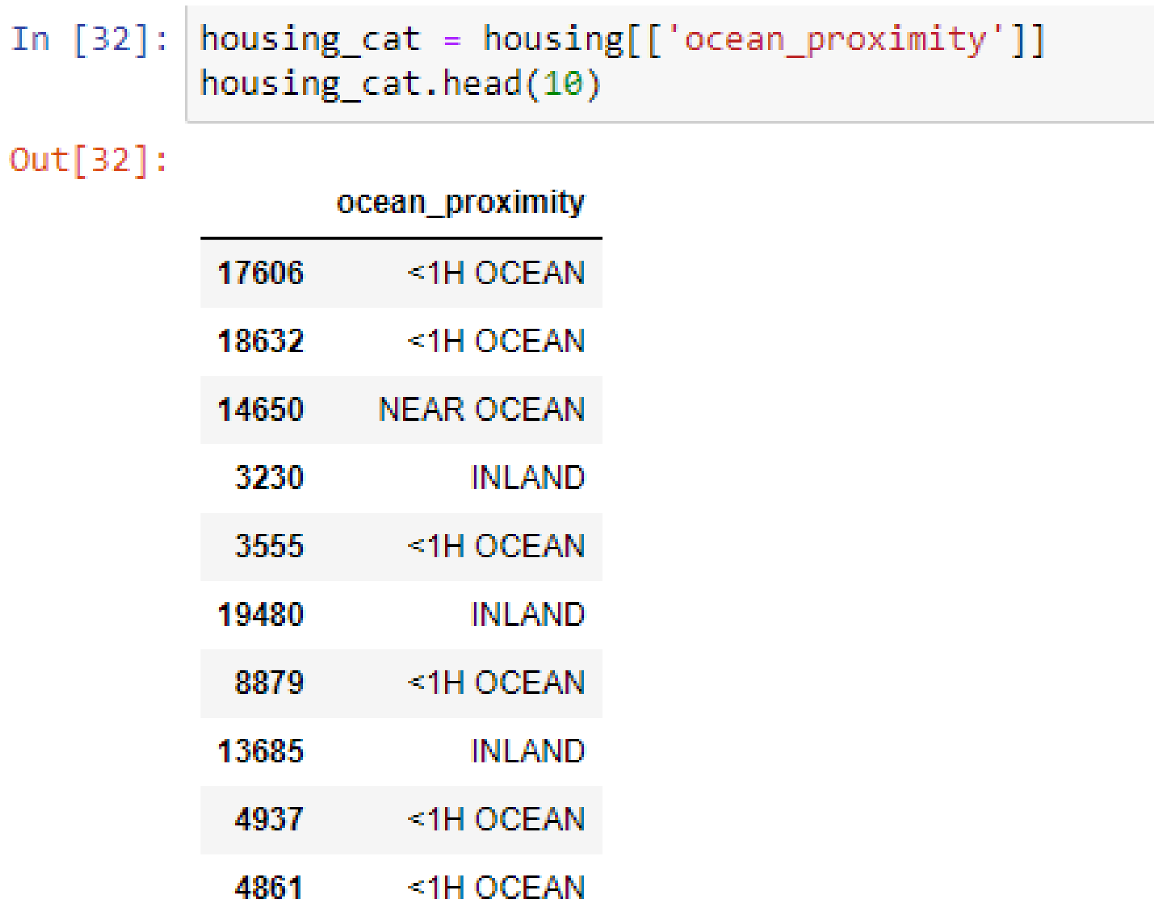
Cleaning the Data using Scikit-Learn



```
In [29]: from sklearn.impute import SimpleImputer
         imputer = SimpleImputer(strategy="median")
         housing_num = housing.drop("ocean_proximity", axis=1)
         imputer.fit(housing_num)
         housing_num.median().values

Out[29]: array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
                 408.    ,    3.5409])

In [30]: X = imputer.transform(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 now let’s look at text attributes.
- In this dataset, there is just one: the ocean_proximity attribute. Let’s look at its value for the first 10 instances

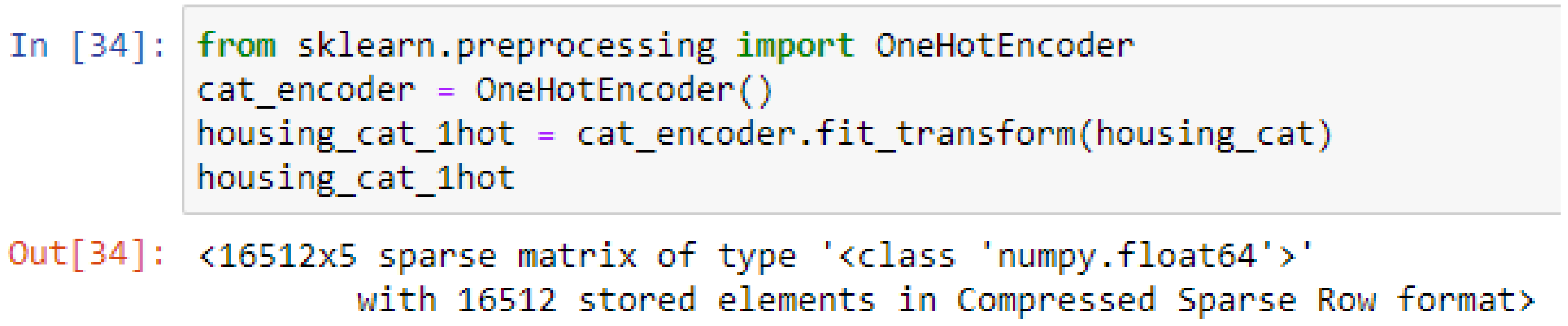


```
In [32]: housing_cat = housing[['ocean_proximity']]
         housing_cat.head(10)

Out[32]:
```

	ocean_proximity
17606	<1H OCEAN
18632	<1H OCEAN
14650	NEAR OCEAN
3230	INLAND
3555	<1H OCEAN
19480	INLAND
8879	<1H OCEAN
13685	INLAND
4937	<1H OCEAN
4861	<1H OCEAN

Converting Categories to Numbers

- 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 OneHotEncoder class.
- One hot encoding creates binary attribute for each category. The category that is true for a given instance will be 1 and all the others 0.



```
In [34]: from sklearn.preprocessing import OneHotEncoder
         cat_encoder = OneHotEncoder()
         housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
         housing_cat_1hot

Out[34]: <16512x5 sparse matrix of type '<class 'numpy.float64'>'
                 with 16512 stored elements in Compressed Sparse Row format>
```

One hot encoding produces a sparse matrix

- Notice that the output is a sparse matrix.
- This is very useful when you have categorical attributes with thousands of categories.
- After one-hot encoding, we get a matrix with thousands of rows full of 0s except for a single 1 per row.
- Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the nonzero elements and their location.

Feature Scaling

- One of the most important transformations you need to apply to your data is feature scaling.
- Machine Learning algorithms generally 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 2 to 39,320, while the median incomes only range from 0 to 15.
- There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

Min-Max Scaling

- Min-max scaling (or normalization) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1.
- We do this by subtracting the min value and dividing by the max minus the min. Scikit-Learn provides a transformer called MinMaxScaler for this.

Min-Max Scaling

- Example of min-max scaling:

```
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)
```

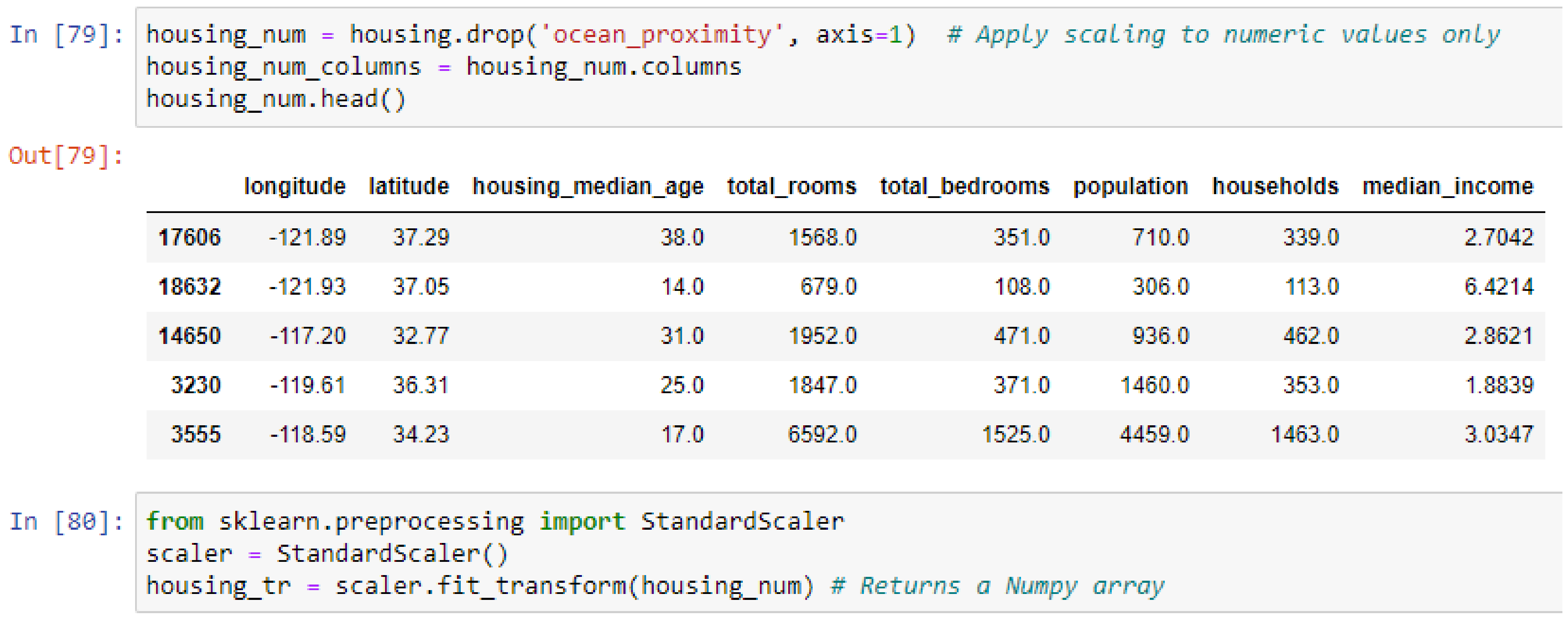
Standardization

- Standardization is different: first it subtracts the mean value (so standardized values have a zero mean), then it divides the result by the standard deviation (so standardized values have a standard deviation equal to 1).
- Unlike min-max scaling, standardization does not bound values to a specific range.

Standardization

- However, standardization is much less affected by outliers.
- For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected.
- Scikit-Learn provides a transformer called StandardScaler for standardization.

Standardization



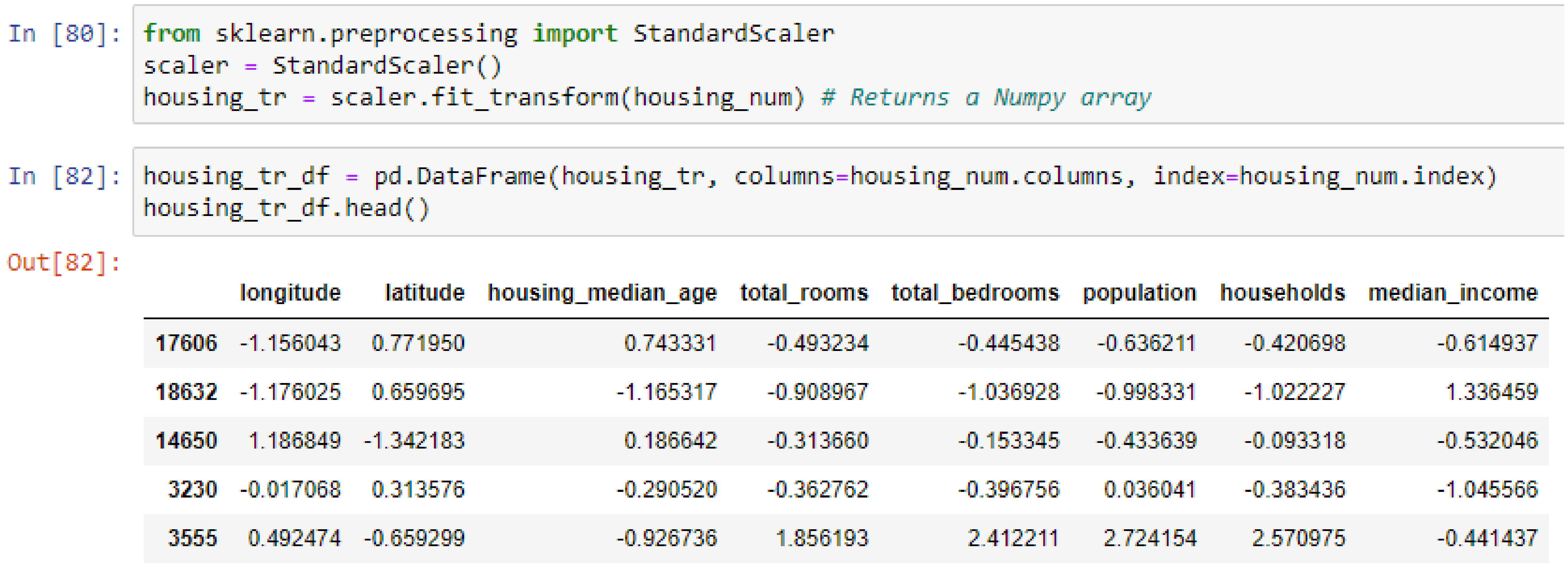
In [79]:
```
housing_num = housing.drop('ocean_proximity', axis=1)  # Apply scaling to numeric values only
housing_num_columns = housing_num.columns
housing_num.head()
```

Out[79]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	339.0	2.7042
18632	-121.93	37.05	14.0	679.0	108.0	306.0	113.0	6.4214
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	462.0	2.8621
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	353.0	1.8839
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	1463.0	3.0347

In [80]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
housing_tr = scaler.fit_transform(housing_num) # Returns a Numpy array
```

Standardization



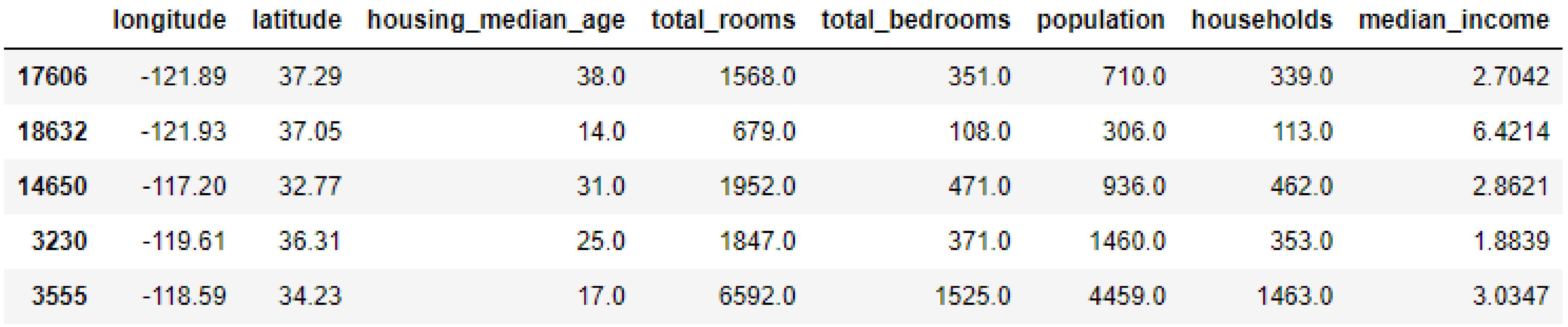
```
In [80]: from sklearn.preprocessing import StandardScaler
         scaler = StandardScaler()
         housing_tr = scaler.fit_transform(housing_num) # Returns a Numpy array
```

```
In [82]: housing_tr_df = pd.DataFrame(housing_tr, columns=housing_num.columns, index=housing_num.index)
         housing_tr_df.head()
```

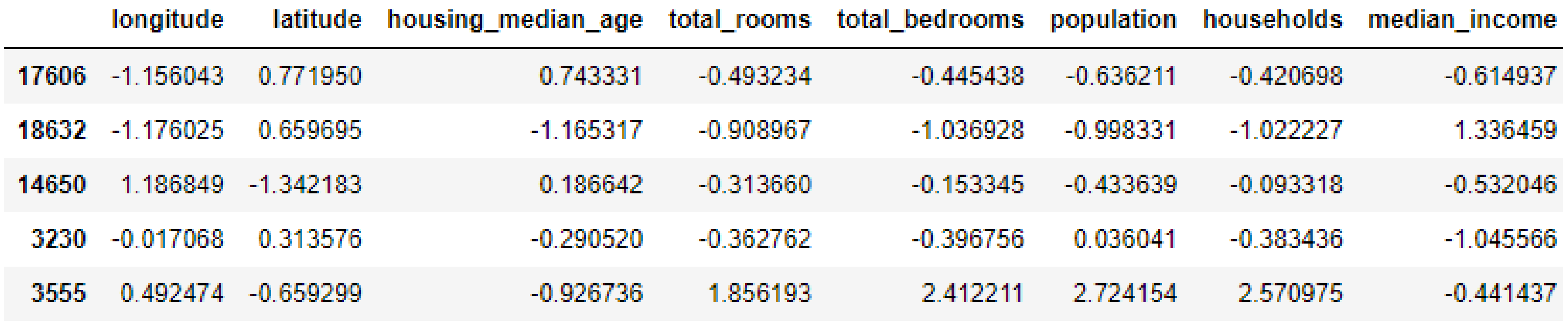
Out[82]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
17606	-1.156043	0.771950	0.743331	-0.493234	-0.445438	-0.636211	-0.420698	-0.614937
18632	-1.176025	0.659695	-1.165317	-0.908967	-1.036928	-0.998331	-1.022227	1.336459
14650	1.186849	-1.342183	0.186642	-0.313660	-0.153345	-0.433639	-0.093318	-0.532046
3230	-0.017068	0.313576	-0.290520	-0.362762	-0.396756	0.036041	-0.383436	-1.045566
3555	0.492474	-0.659299	-0.926736	1.856193	2.412211	2.724154	2.570975	-0.441437

Standardization (before and after)

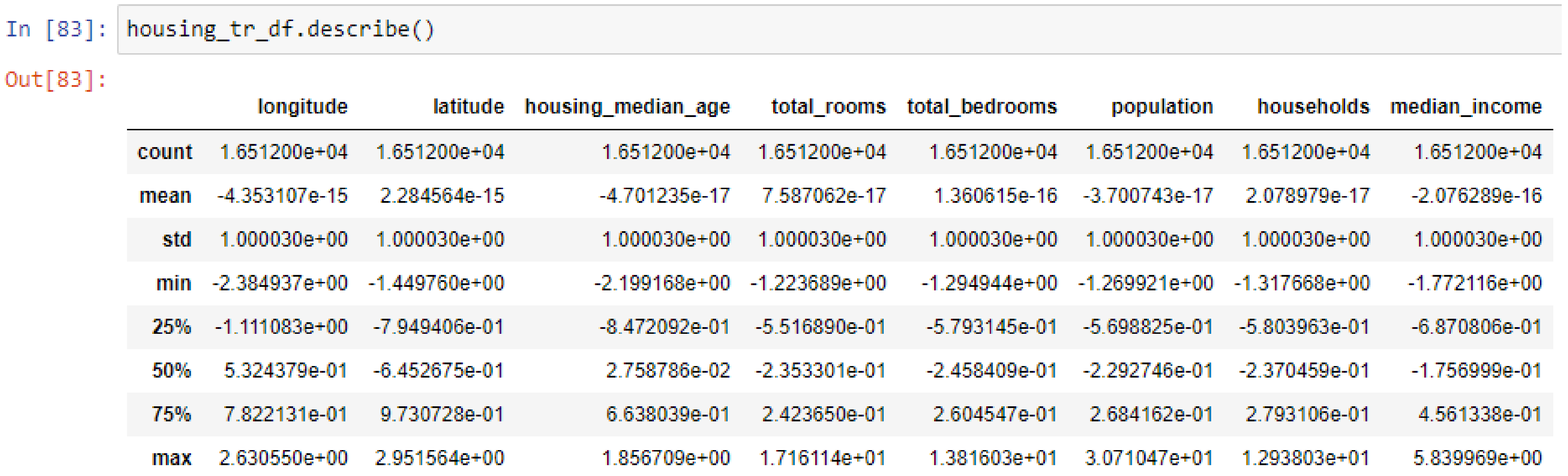


	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	339.0	2.7042
18632	-121.93	37.05	14.0	679.0	108.0	306.0	113.0	6.4214
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	462.0	2.8621
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	353.0	1.8839
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	1463.0	3.0347



	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
17606	-1.156043	0.771950	0.743331	-0.493234	-0.445438	-0.636211	-0.420698	-0.614937
18632	-1.176025	0.659695	-1.165317	-0.908967	-1.036928	-0.998331	-1.022227	1.336459
14650	1.186849	-1.342183	0.186642	-0.313660	-0.153345	-0.433639	-0.093318	-0.532046
3230	-0.017068	0.313576	-0.290520	-0.362762	-0.396756	0.036041	-0.383436	-1.045566
3555	0.492474	-0.659299	-0.926736	1.856193	2.412211	2.724154	2.570975	-0.441437

Standardization



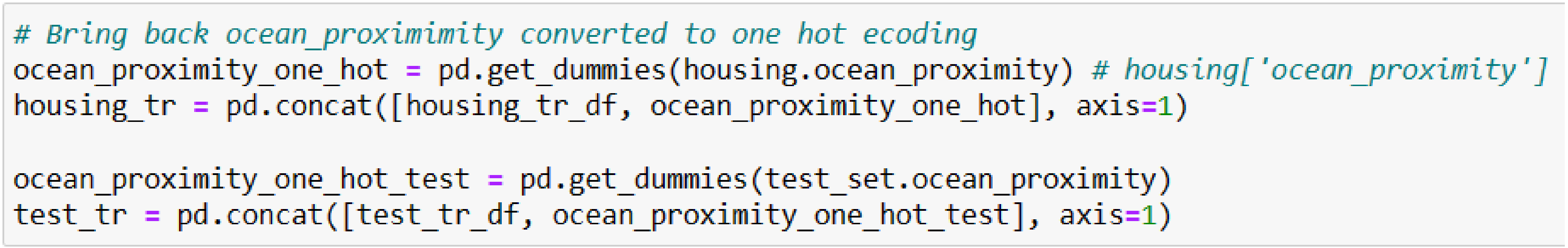
```
In [83]: housing_tr_df.describe()
```

Out[83]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
count	1.651200e+04	1.651200e+04	1.651200e+04	1.651200e+04	1.651200e+04	1.651200e+04	1.651200e+04	1.651200e+04
mean	-4.353107e-15	2.284564e-15	-4.701235e-17	7.587062e-17	1.360615e-16	-3.700743e-17	2.078979e-17	-2.076289e-16
std	1.000030e+00	1.000030e+00	1.000030e+00	1.000030e+00	1.000030e+00	1.000030e+00	1.000030e+00	1.000030e+00
min	-2.384937e+00	-1.449760e+00	-2.199168e+00	-1.223689e+00	-1.294944e+00	-1.269921e+00	-1.317668e+00	-1.772116e+00
25%	-1.111083e+00	-7.949406e-01	-8.472092e-01	-5.516890e-01	-5.793145e-01	-5.698825e-01	-5.803963e-01	-6.870806e-01
50%	5.324379e-01	-6.452675e-01	2.758786e-02	-2.353301e-01	-2.458409e-01	-2.292746e-01	-2.370459e-01	-1.756999e-01
75%	7.822131e-01	9.730728e-01	6.638039e-01	2.423650e-01	2.604547e-01	2.684162e-01	2.793106e-01	4.561338e-01
max	2.630550e+00	2.951564e+00	1.856709e+00	1.716114e+01	1.381603e+01	3.071047e+01	1.293803e+01	5.839969e+00

One-hot encoded ocean proximity

- The ocean_proximity feature was one_hot encoded separately
- We need to attach it to the train and test sets
- We will repeat the encoding using the easier get_dummies method of pandas and then attach.



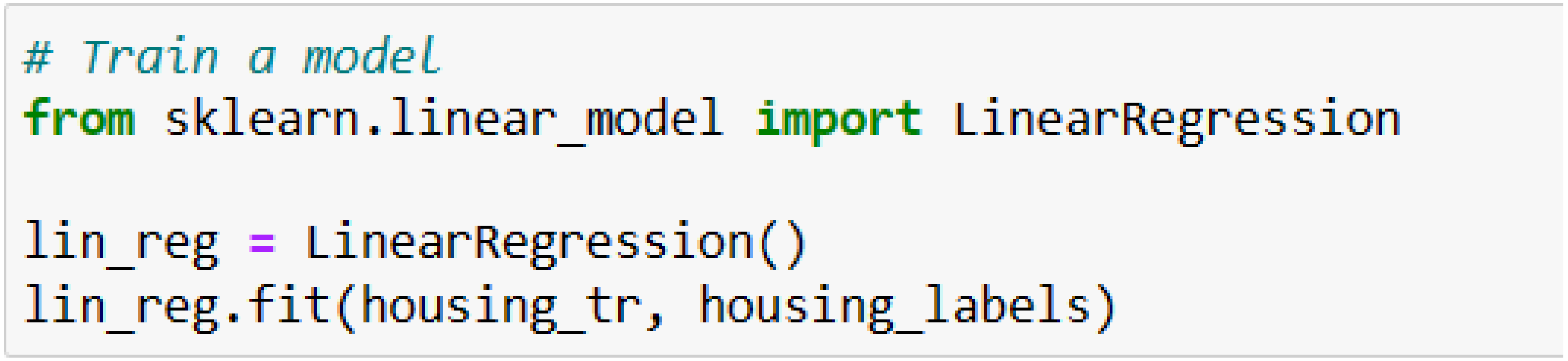
```
# Bring back ocean_proximimity converted to one hot ecoding
ocean_proximity_one_hot = pd.get_dummies(housing.ocean_proximity) # housing['ocean_proximity']
housing_tr = pd.concat([housing_tr_df, ocean_proximity_one_hot], axis=1)

ocean_proximity_one_hot_test = pd.get_dummies(test_set.ocean_proximity)
test_tr = pd.concat([test_tr_df, ocean_proximity_one_hot_test], axis=1)
```

Training a model

- At this stage, we are ready to train a model
- We can use k-fold cross validation to evaluate multiple machine learning algorithms on this pre-processed dataset, e.g.,
  - Linear regression
  - KNN
  - Decision tree regression
- Then we pick the one that gives us the best performance and use it to train a model.
- To evaluate the model on the test data, we must subject the test data to the same transformations we put the training data through.

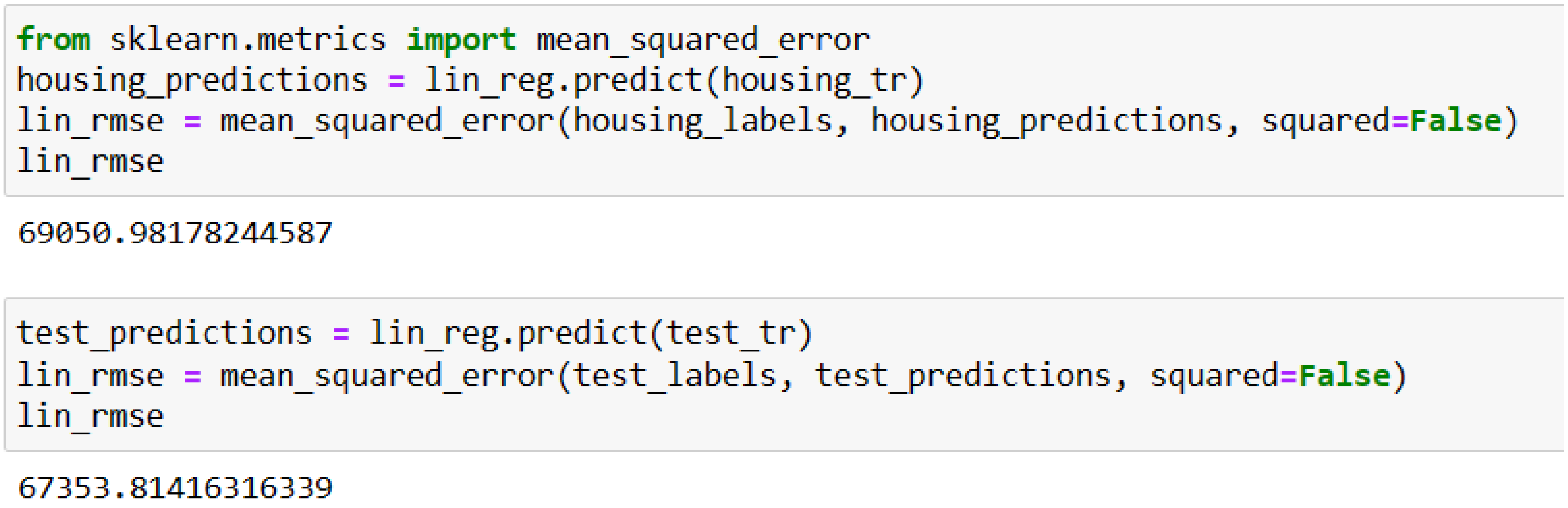
Training a Linear Regression Model

- Let’s train a linear regression model.



```python
# Train a model
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_tr, housing_labels)
```

Evaluating the model



```
from sklearn.metrics import mean_squared_error
housing_predictions = lin_reg.predict(housing_tr)
lin_rmse = mean_squared_error(housing_labels, housing_predictions, squared=False)
lin_rmse
```

```
69050.98178244587
```

```
test_predictions = lin_reg.predict(test_tr)
lin_rmse = mean_squared_error(test_labels, test_predictions, squared=False)
lin_rmse
```

```
67353.81416316339
```

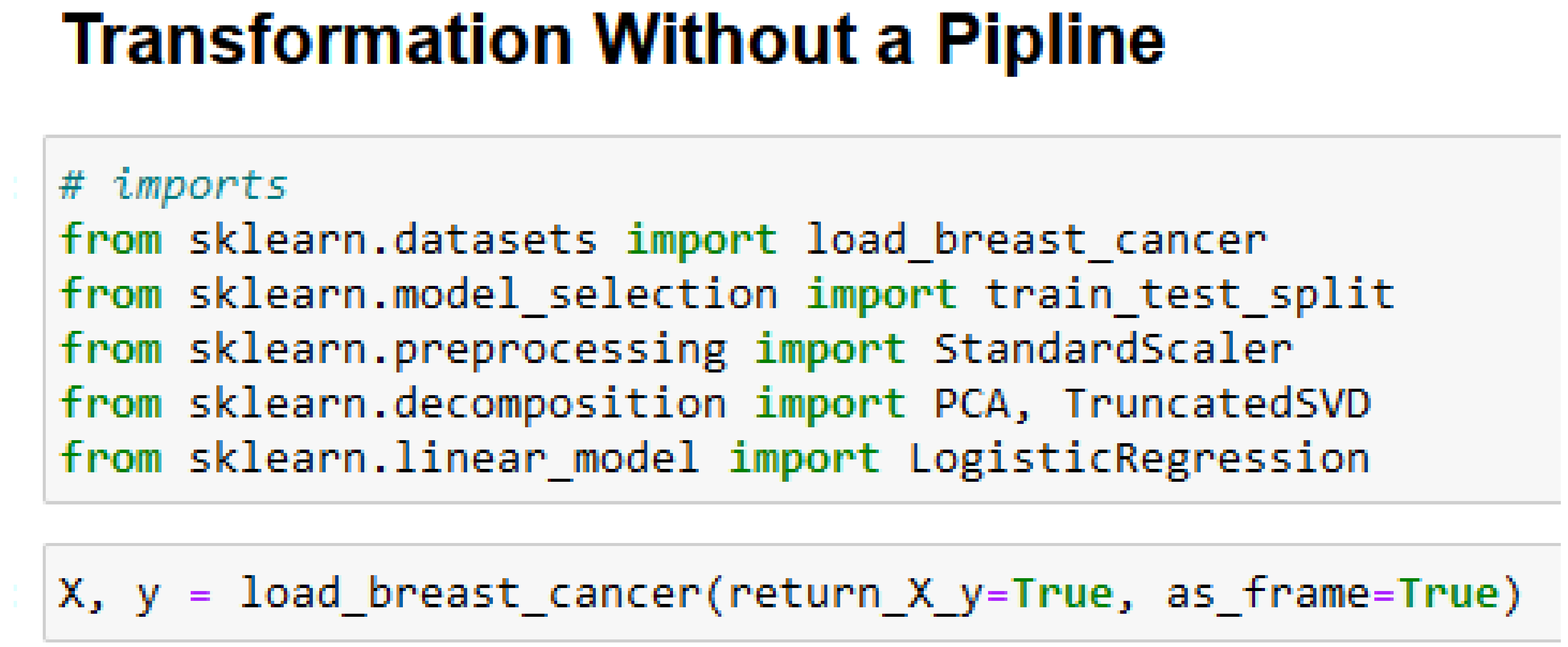
Pipelines

- As you can see, data preprocessing can involve many steps.
- These steps are applied to the training set only.
- We need to a mechanism that reliably applies the exact same steps to any dataset that comes into the picture later on, including the test set and new instances to make predictions on.
- The best approach to ensure the same steps are followed is to use a pipeline.

Pipelines

- Our steps in the California Housing Prices example are:
  - Imputing missing values
  - Converting categorical data to numeric
  - Scaling data
  - Training a model
- A pipeline is an assembly of steps whose aim is to automate the machine learning workflow.

Example

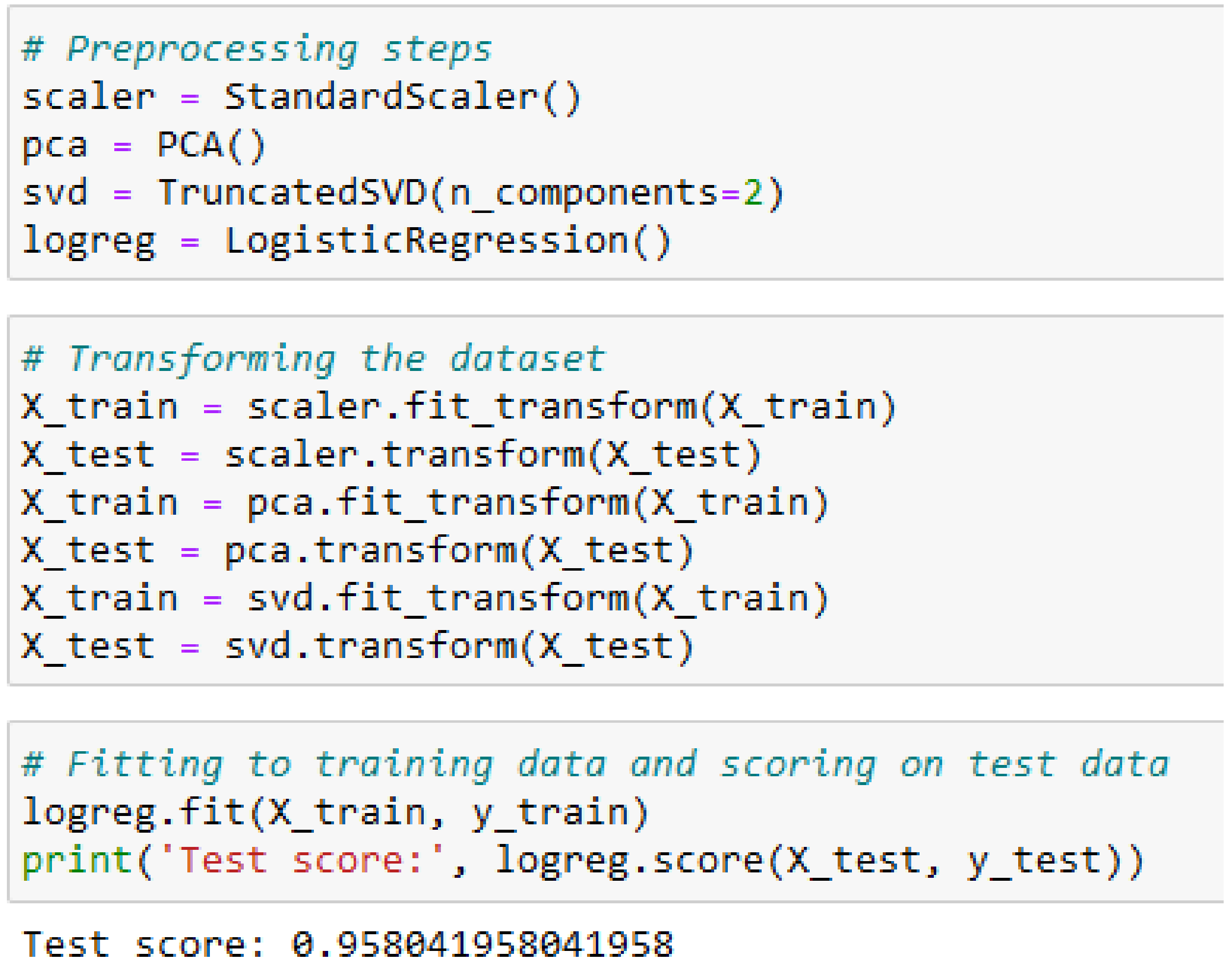


Transformation Without a Pipline

```python
# imports
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.linear_model import LogisticRegression
```

```python
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
```

Transformation without pipeline



```python
# Preprocessing steps
scaler = StandardScaler()
pca = PCA()
svd = TruncatedSVD(n_components=2)
logreg = LogisticRegression()
```

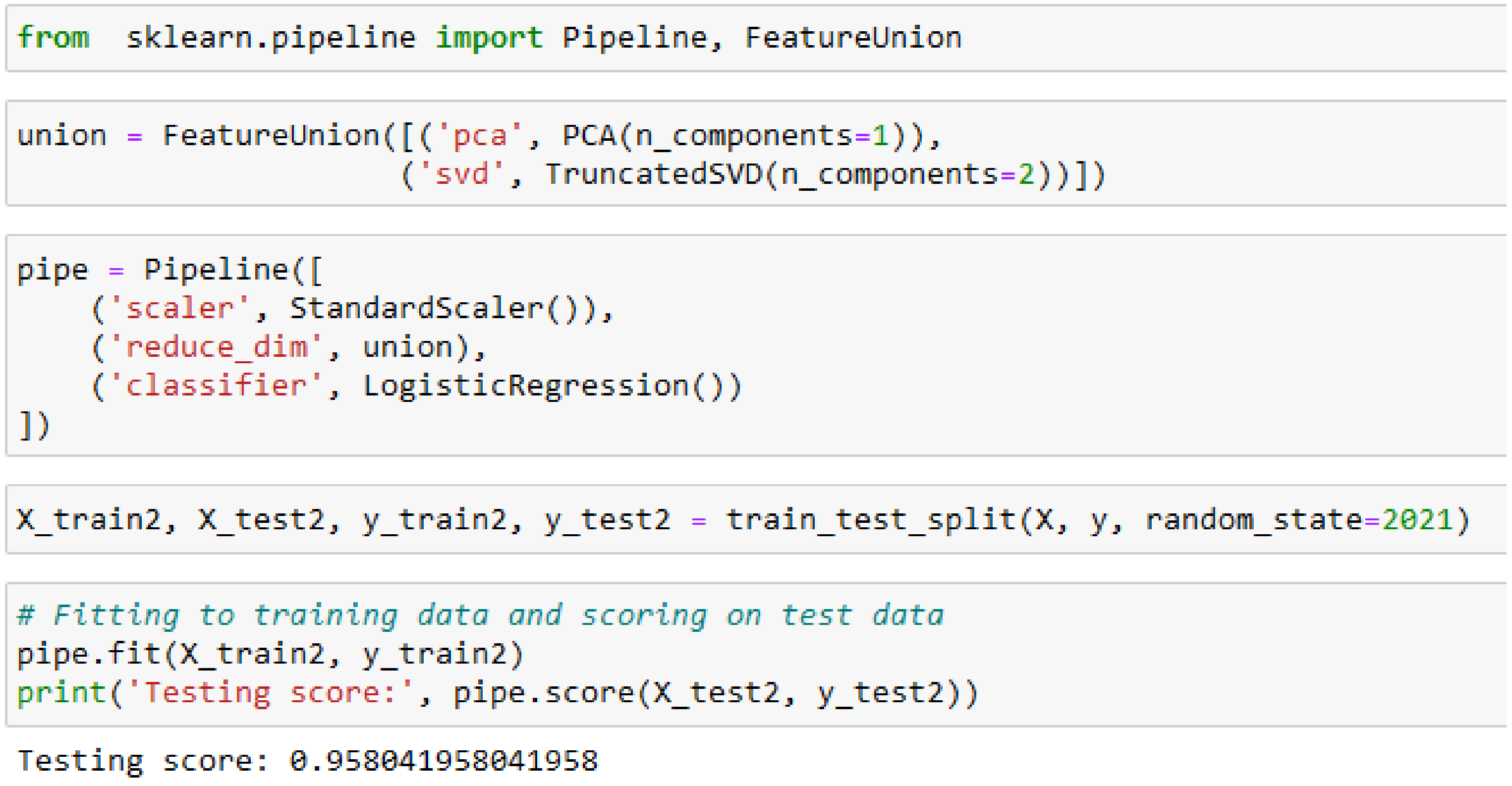
```python
# Transforming the dataset
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
X_train = svd.fit_transform(X_train)
X_test = svd.transform(X_test)
```

```python
# Fitting to training data and scoring on test data
logreg.fit(X_train, y_train)
print('Test score:', logreg.score(X_test, y_test))
```

```
Test score: 0.958041958041958
```

Transformation with pipeline

- Assume all previous imports are in place



```
from  sklearn.pipeline import Pipeline, FeatureUnion
```

```
union = FeatureUnion([('pca', PCA(n_components=1)),
                     ('svd', TruncatedSVD(n_components=2))])
```

```
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('reduce_dim', union),
    ('classifier', LogisticRegression())
])
```

```
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, random_state=2021)
```

```
# Fitting to training data and scoring on test data
pipe.fit(X_train2, y_train2)
print('Testing score:', pipe.score(X_test2, y_test2))
```

```
Testing score: 0.958041958041958
```

Advantages of using a pipeline

- Clean code
- Easy to reproduce
- Removes redundancy
- Establishes a standard workflow
- Easy to debug
- Supports code re-use

Reference

- The content in these slides was taken from
  - *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition by Aurélien Géron, O'Reilly Media, Inc., 2022
- This book is highly recommended for anyone aiming to go into data science as a profession.