Data Visualization and Cleaning

CSCI-P556 Applied Machine Learning
Lecture 5

D.S. Williamson

Agenda and Learning Outcomes

Today's Topics

- **Topics**:
  - Finish data splitting and visualization
  - Data pre-processing
    - Attribute Removal and Imputation
    - Handling Categorical data
    - Features scaling (normalization)

Data Pre-processing

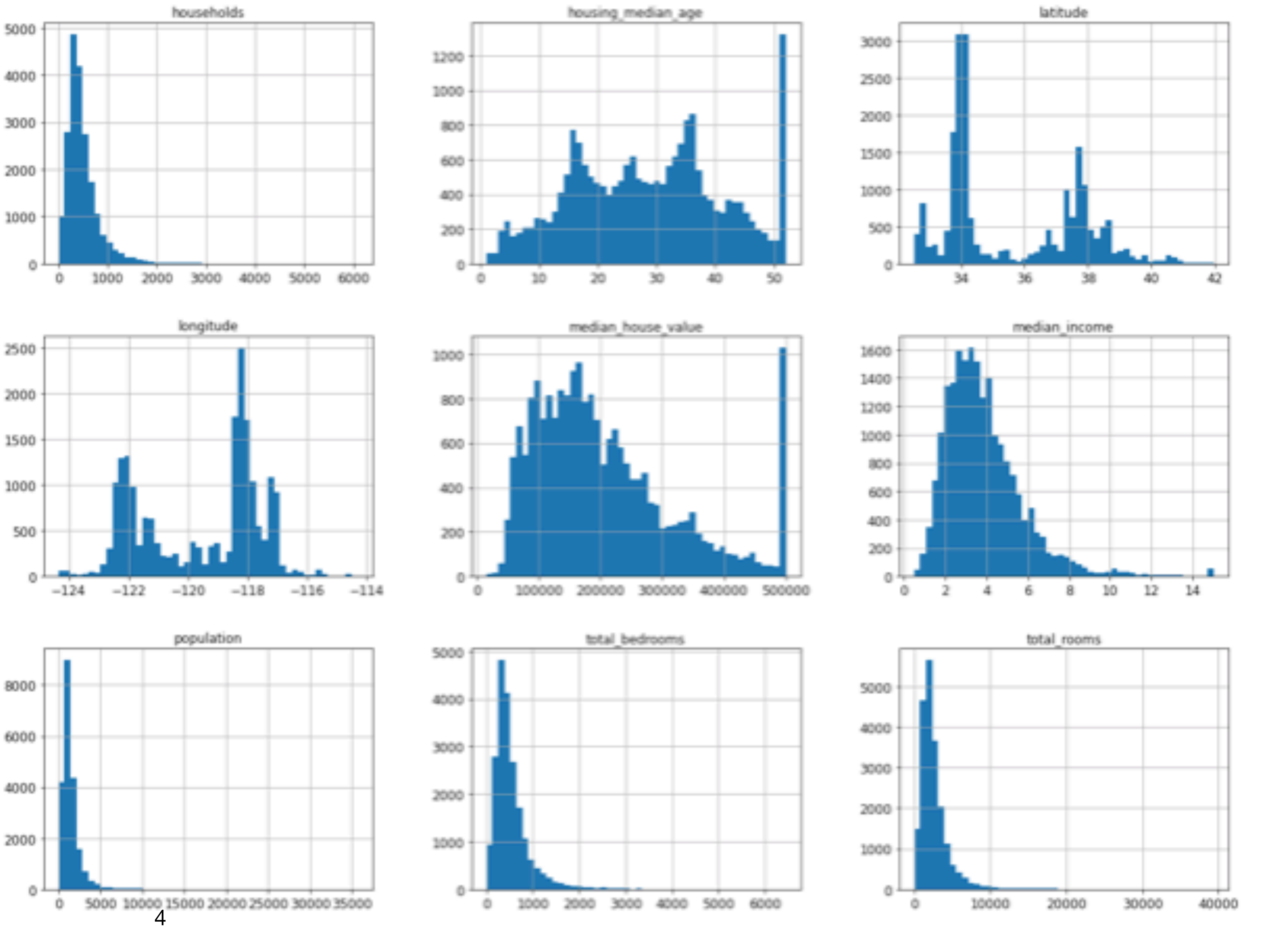
Now that we have data, what's next? An Example Case

- Suppose you are a Data Scientist at a Housing Corporation. Your boss wants you to build a prediction model of median housing prices in California using their census data
- **Data has info about**: population, median income, median housing prices, … for each block group or district in California.
- **How should this problem be framed**?
  - Supervised Learning, Unsupervised learning, Reinforcement Learning? Why?
  - Classification, Regression, Other? Why?
  - Batch vs. Online?

3. Analyze the Data - Group Activity

Look at the visual characteristics of the data

- Median age and house values were capped.
  - This may impact generalization
- Most attributes follow different “distributions”
- Four attributes have heavy tails.
  - May complicate ML
  - May need to be transformed



Splitting the data in Python

Scikit-Learn’s Solution: Training and Testing Sets

In [17]:
```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

In [18]:
```
test_set.head()
```

Out[18]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
20046	-119.01	36.06	25.0	1505.0	NaN	1392.0	359.0	1.6812
3024	-119.46	35.14	30.0	2943.0	NaN	1565.0	584.0	2.5313
15663	-122.44	37.80	52.0	3830.0	NaN	1310.0	963.0	3.4801
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	5.7376
9814	-121.93	36.62	34.0	2351.0	NaN	1063.0	428.0	3.7250

Data Spliting using Random Sampling

- Any problems with randomly splitting the da
    - Potential for Sampling Bias
    - Need training/testing data to be representative
- Instead, maintain “appropriate and representative” ratios of data in both sets. This is called ***stratified sampling***, since the data is divided into homogenous subgroups called strata where the right number of instances is sampled from each stratum (or subgroup)
    - Let’s see this through an example

A Famous Example of Sampling Bias

Perhaps the most famous example of sampling bias happened during the US presidential election in 1936, which pitted Landon against Roosevelt: the *Literary Digest* conducted a very large poll, sending mail to about 10 million people. It got 2.4 million answers, and predicted with high confidence that Landon would get 57% of the votes.

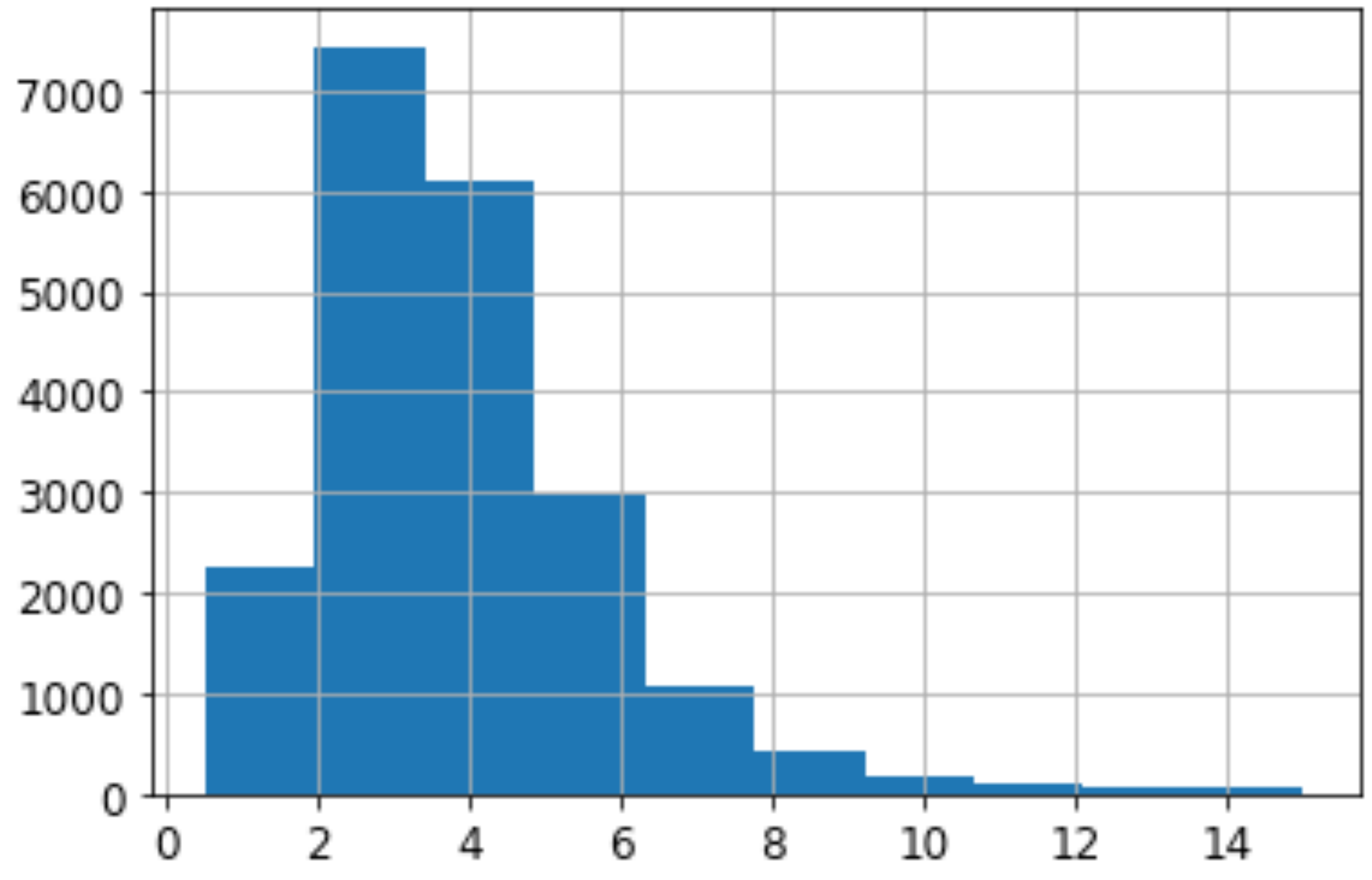
24 | Chapter 1: The Machine Learning Landscape

Stratified Sampling

Housing Example Continued

- Let’s look at the “median_income” attribute

```
In [19]: housing["median_income"].hist()

Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8ad8f104f0>
```

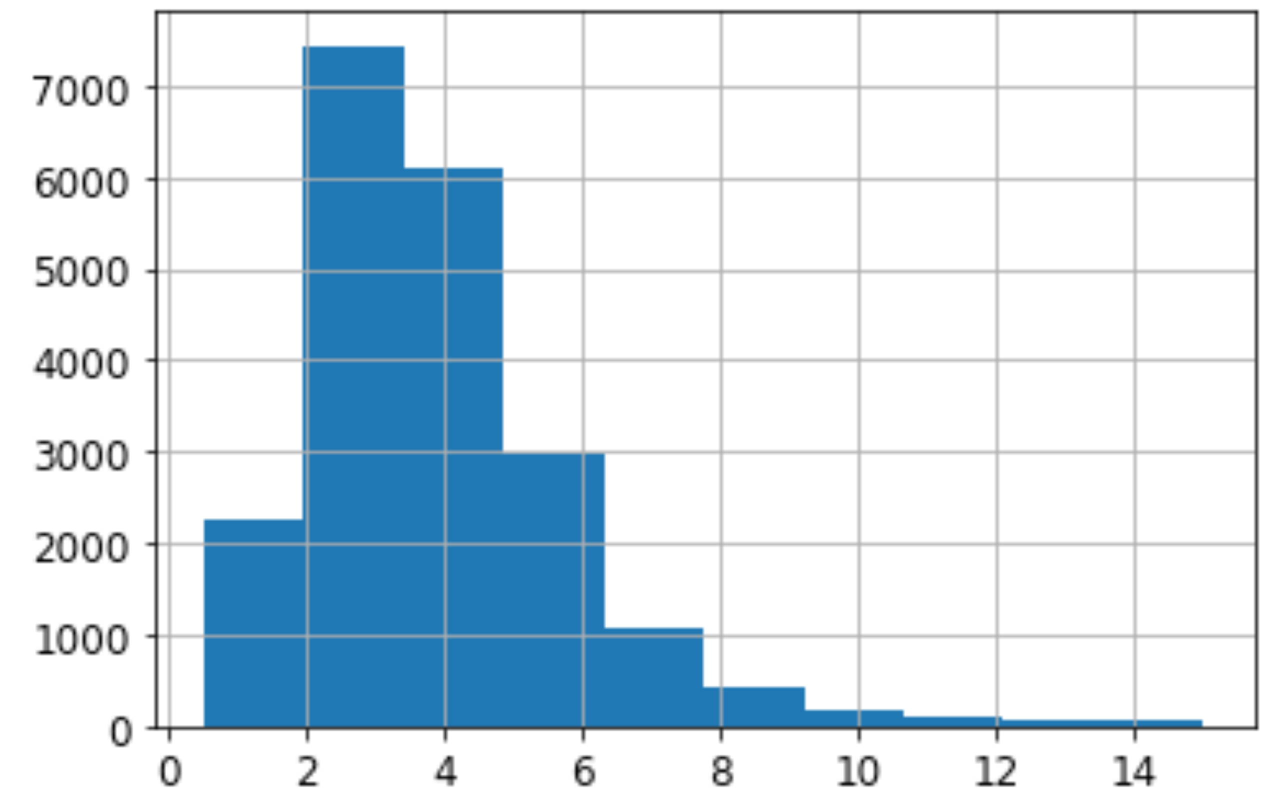


- Most data is between 2 and 5, but some goes beyond this
- Need instances from each stratum, or bias will occur

Stratified Sampling

Housing Example Continued

```
In [19]: housing["median_income"].hist()
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8ad8f104f0>
```



- We can: (1) Limit the number of strata and (2) Ensure each strata has enough examples (e.g. merge instances where income > 6 into one strata)

```
In [20]: 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])
```

Divide into Strata (using 1.5 spacing)

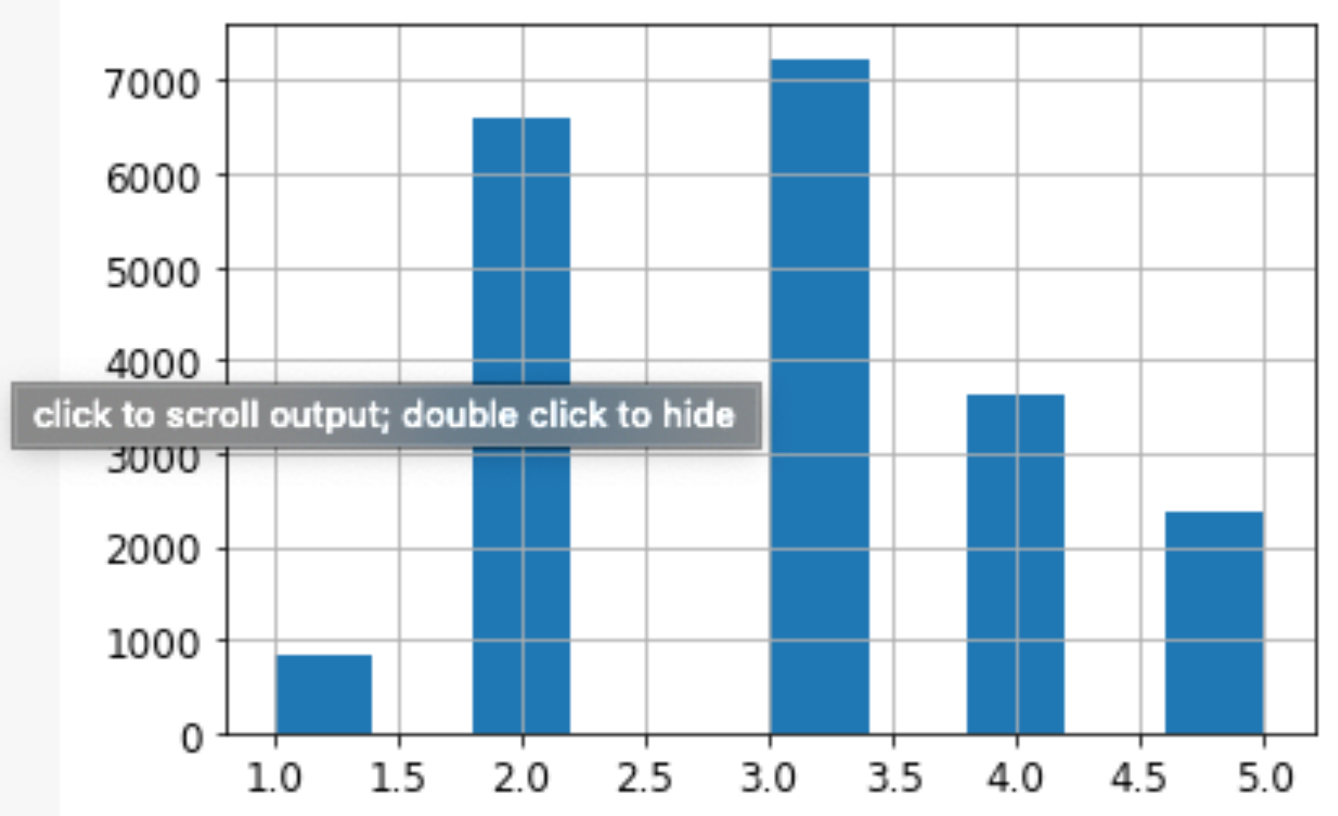
```
In [21]: housing["income_cat"].value_counts()

Out[21]: 3    7236
         2    6581
         4    3639
         5    2362
         1     822
         Name: income_cat, dtype: int64
```

Stratified Sampling

Housing Example Continued

- Finally performing stratified sampling

```
In [22]: housing["income_cat"].hist()
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8ad91709a0>
```




```
In [23]: from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

split.split(): Generate indices to split data into training and test sets

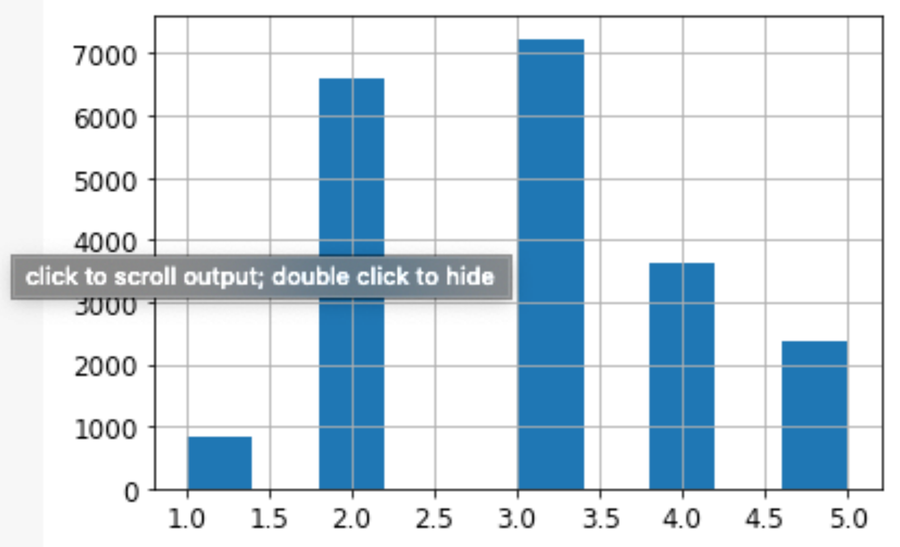
Specifies training data

Specifies variable/ attribute used for stratification

Stratified Sampling

Housing Example Continued

```
In [22]: housing["income_cat"].hist()
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8ad91709a0>
```



- Comparing data split for testing set, training set and original data

Percentages by income category match

```
In [24]: strat_test_set["income_cat"].value_counts() / len(strat_test_set)
Out[24]: 3    0.350533
         2    0.318798
         4    0.176357
         5    0.114583
         1    0.039729
         Name: income_cat, dtype: float64
```

```
In [26]: strat_train_set["income_cat"].value_counts() / len(strat_train_set)
Out[26]: 3    0.350594
         2    0.318859
         4    0.176296
         5    0.114402
         1    0.039850
         Name: income_cat, dtype: float64
```

```
In [25]: housing["income_cat"].value_counts() / len(housing)
Out[25]: 3    0.350581
         2    0.318847
         4    0.176308
         5    0.114438
         1    0.039826
         Name: income_cat, dtype: float64
```

Stratified Sampling

Housing Example Continued

- Remove stratified variable attribute “income_cat”, since we only used it to have representative data splits (we don’t really want to use it as an attribute)

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

```
In [44]: strat_train_set.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 16512 entries, 17606 to 15775
Data columns (total 11 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16354 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
 8   median_house_value  16512 non-null  float64
 9   ocean_proximity     16512 non-null  object
 10  income_cat          16512 non-null  category
dtypes: category(1), float64(9), object(1)
memory usage: 1.4+ MB
```

After Removal →

```
In [46]: strat_train_set.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 16512 entries, 17606 to 15775
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16354 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
 8   median_house_value  16512 non-null  float64
 9   ocean_proximity     16512 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.4+ MB
```

Example cont: Visualizing Training Data

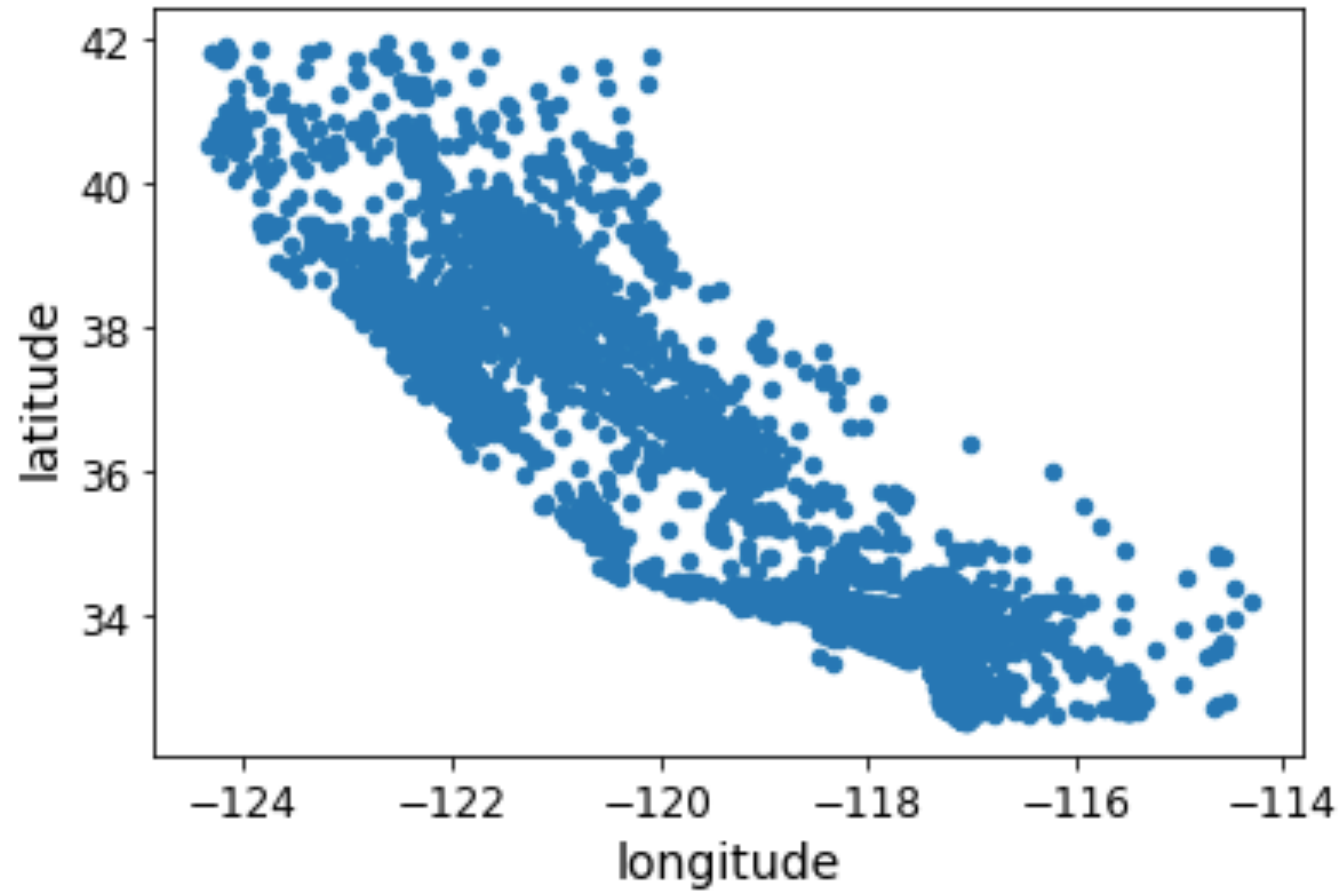
Look at training data (or subsets of it)

- Copy training data before doing this, to avoid potential mistakes

```
In [47]: housing = strat_train_set.copy()
```

- The goal is to visualize the data to find informative patterns

```
housing.plot(kind="scatter", x="longitude", y="latitude")
#save_fig("bad_visualization_plot")
```



- Looks like California
- Other plots may provide more info

Example cont: Visualizing Training Data

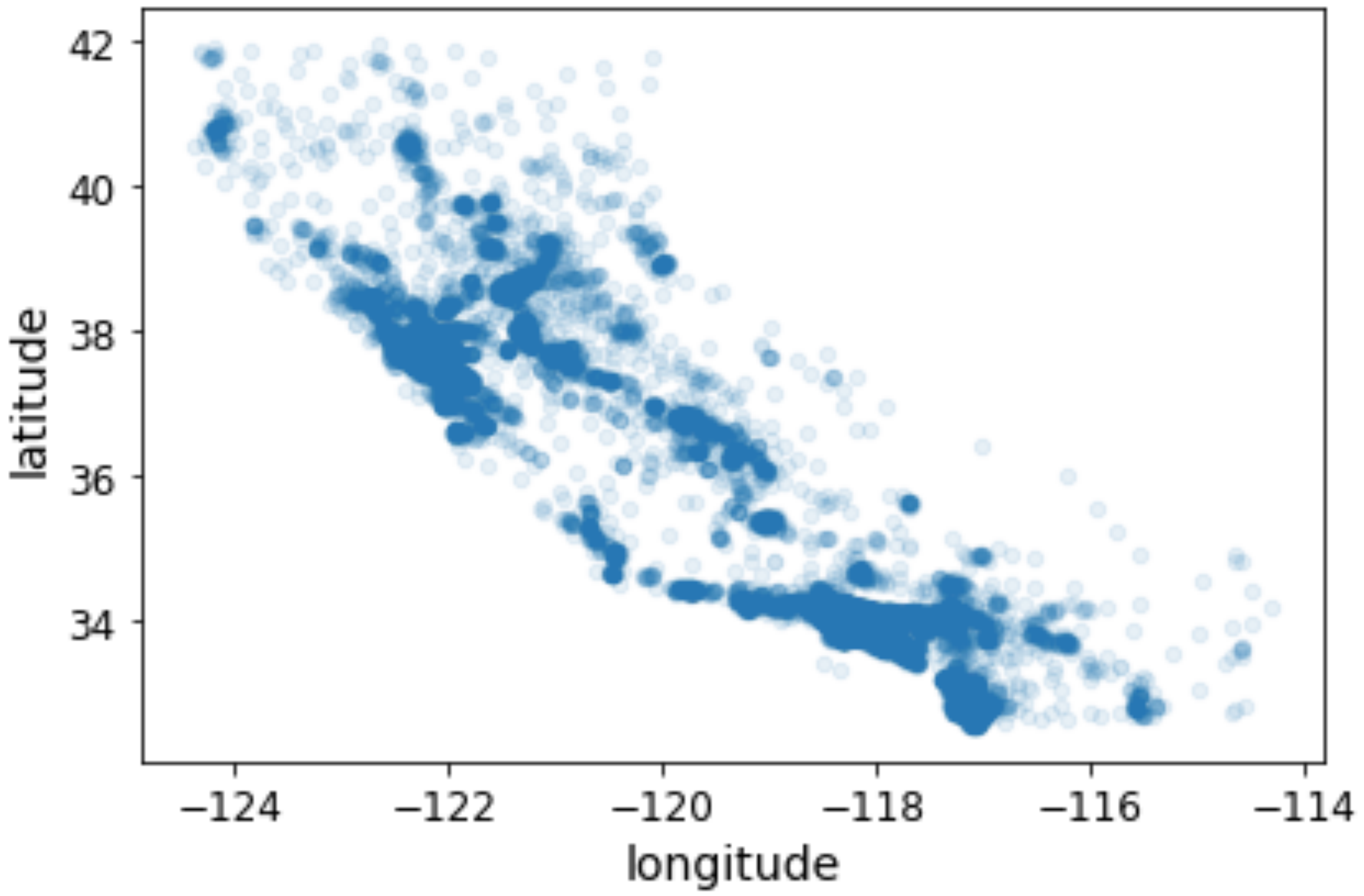
Look at training data (or subsets of it)

- Copy training data before doing this, to avoid potential mistakes

```
In [47]: housing = strat_train_set.copy()
```

- The goal is to visualize the data to find informative patterns

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```



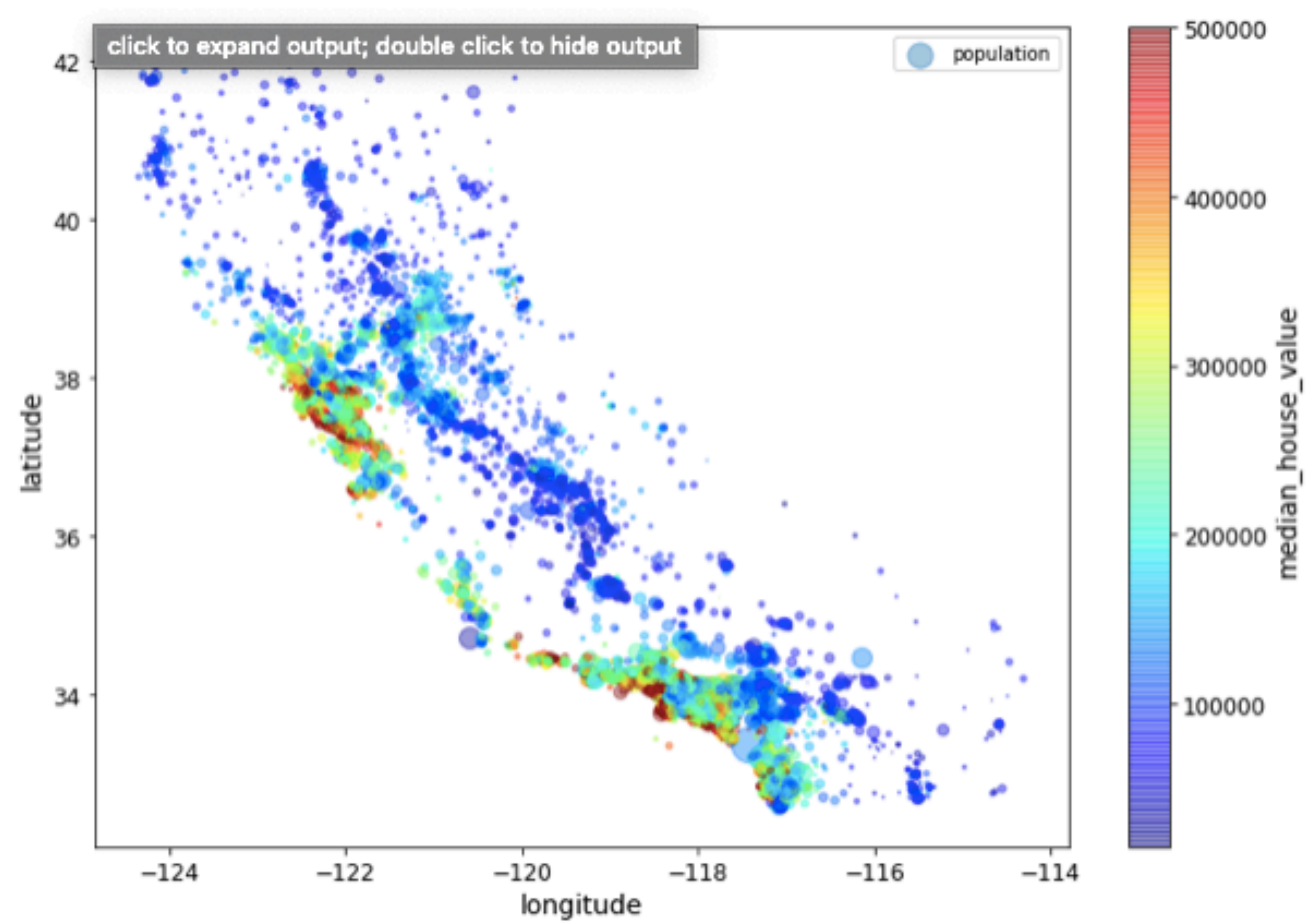
Can better see where data is more densely located

Visualizing House Prices of Training Data

Look at training data (or subsets of it)

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
    sharex=False)
plt.legend()
```

- Size of circle radius represents population (option s)
- Price is represented by color (option c)
- **What does the image say about the housing prices?**



Analyze Relations Between Attributes

Correlation

- ***<u>Pearson's Correlation coefficient</u>*** is a standard approach to determine if two attributes (or sets of data) are ***<u>linearly</u>*** related (i.e. y = mx + b)

- Let's compute the correlation coefficient between two data sets ***r*** and ***d***, where ***r*** and ***d*** are *N*-dimensional vectors

$$\mu_r = \frac{1}{N}\sum_{I=1}^{N} r_i$$

$$\mu_d = \frac{1}{N}\sum_{I=1}^{N} d_i$$

Average values of ***r*** and ***d***, respectively

$$\rho = \frac{\sum_{I=1}^{N}(r_i - \mu_r)(d_i - \mu_d)}{\sqrt{\sum_{I=1}^{N}(r_i - \mu_r)^2}\sqrt{\sum_{I=1}^{N}(d_i - \mu_d)^2}}$$

Correlation has values between -1 and 1

Correlation Coefficient: Data Plots and Correlation

Interpretation of Correlation Coefficient

- ***<u>Close to -1</u>*** -> strong negative correlation between pairs
- ***<u>Close to 1</u>*** -> strong positive correlation
- ***<u>Close to 0</u>*** -> There is no linear correlation

PCC between pairs of attributes

corr() method computes PCC in Python

- Python enables the computation of correlation across each pair of attributes

```
corr_matrix = housing.corr()
```

- We can now check to see how each attribute (linearly) correlates with the median house value (e.g. our label)

```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
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
```

What does it tell you about the features? Are any attributes more important than others? Less important?

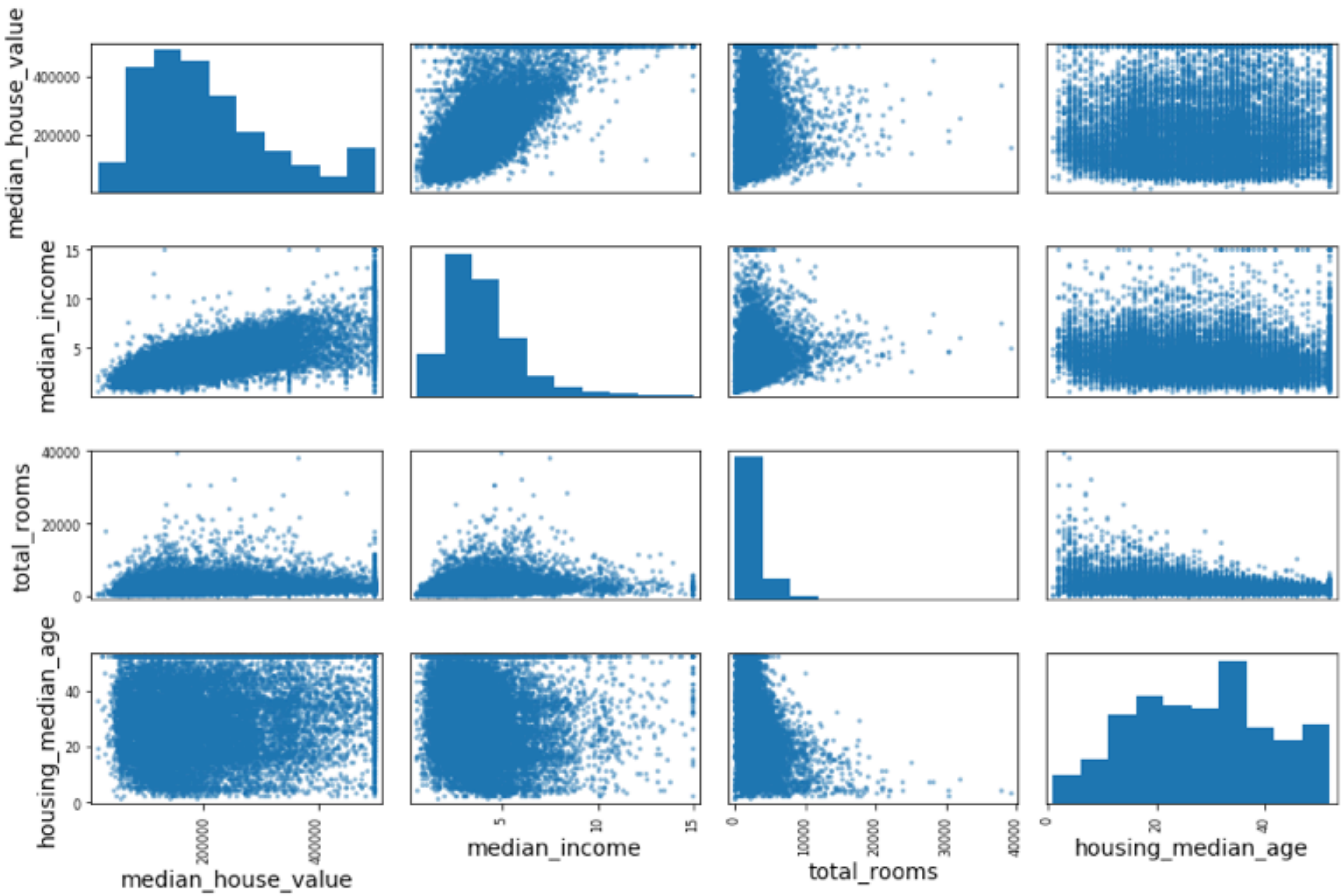
PCC between pairs of attributes

We can also generate scatter plots to show this

```
# from pandas.tools.plotting import scatter_matrix # For older versi
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
#save_fig("scatter_matrix_plot")
```

- We can now check to see how each attribute (linearly) correlates with the median house value (e.g. our label)

- <u>Is it clear which attribute may be most helpful in predicting median house value? Why or Why not?</u>



Data Cleaning

- Often times the received data is unclean and needs to be modified before it can be given to a machine learning algorithm
- The process of generated “good quality” data is known as ***data cleaning.*** It involves
  - Removing and/or imputing missing values
  - Getting categorical data into the proper format
  - Selecting relevant features
- Luckily, Python has built-in functionality to help with this

Housing Example cont.

Data Cleaning

- First, let's separate the data (e.g., input, feature) from the labels using Panda's drop() method for a DataFrame object

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

Make a copy of the median house values

```
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16354 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
 8   ocean_proximity     16512 non-null  object
dtypes: float64(8), object(1)
memory usage: 1.3+ MB
```

`DataFrame.drop`(*labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise'*)

Drop specified labels from rows or columns. [source]

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters:

labels : *single label or list-like*
Index or column labels to drop.

axis : *{0 or 'index', 1 or 'columns'}, default 0*
Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index : *single label or list-like*
Alternative to specifying axis (`labels, axis=0` is equivalent to `index=labels`).

columns : *single label or list-like*
Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

level : *int or level name, optional*
For MultiIndex, level from which the labels will be removed.

inplace : *bool, default False*
If False, return a copy. Otherwise, do operation inplace and return None.

errors : *{'ignore', 'raise'}, default 'raise'*
If 'ignore', suppress error and only existing labels are dropped.

Returns:

DataFrame or None
DataFrame without the removed index or column labels or None if `inplace=True`.

Recall: 3. Analyze the Data

Look at information

- Use info() to get information about the data, including formats of attributes/ labels

```
In [6]: 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
```

Number of data samples (e.g. entries)

Format of each attribute

total_bedrooms has null values (e.g. missing data)

?

Data Cleaning

Complete Removal of Attributes

- Total_bedrooms is missing data. Use DataFrame's drop() methods to remove the attribute

```
housing = housing.drop("total_bedrooms",axis=1)
```

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16512 entries, 17606 to 15775
Data columns (total 8 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   population          16512 non-null  float64
 5   households          16512 non-null  float64
 6   median_income       16512 non-null  float64
 7   ocean_proximity     16512 non-null  object
dtypes: float64(7), object(1)
memory usage: 1.1+ MB
```

Why should we or should we not completely remove an attribute?

Data Cleaning

Removal of Instances with Missing Attribute values

- Use DataFrame’s dropna() method to remove data instances with missing values (up to 207 districts with null values)

```
housing = housing.dropna(subset=["total_bedrooms"])
```

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16354 entries, 17606 to 15775
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16354 non-null  float64
 1   latitude            16354 non-null  float64
 2   housing_median_age  16354 non-null  float64
 3   total_rooms         16354 non-null  float64
 4   total_bedrooms      16354 non-null  float64
 5   population          16354 non-null  float64
 6   households          16354 non-null  float64
 7   median_income       16354 non-null  float64
 8   ocean_proximity     16354 non-null  object
dtypes: float64(8), object(1)
memory usage: 1.2+ MB
```

Is this a better idea?

Data Cleaning

Impute Missing Values

Data Imputation is a heavily researched area. Machine Learning techniques are now often used to do it.

- Replace missing values with an alternative value. Often statistical value is used (e.g. median, mean,…)

```
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median,inplace=True)
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16512 entries, 17606 to 15775
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16512 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
 8   ocean_proximity     16512 non-null  object
dtypes: float64(8), object(1)
memory usage: 1.9+ MB
```

- Imputation can also be done with Scikit-Learn (see textbook)

Number of data samples (e.g. entries)

total_bedrooms has no null values

Handling Categorical Data

Converting to Numerical Values

- Data often contains non-numerical attributes. Machine Learning, however, requires numerical values in order to learn. Hence, must modify categorical attributes.

```
In [7]: housing["ocean_proximity"].value_counts()

Out[7]: <1H OCEAN     9136
        INLAND        6551
        NEAR OCEAN    2658
        NEAR BAY      2290
        ISLAND           5
        Name: ocean_proximity, dtype: int64
```

- Two categorical data types:
  - **Ordinal**: values can be sorted or ordered (e.g. shirt size: XL > L > M).
  - **Nominal**: text values without a order (e.g. shirt color: red, blue, black,…)

Handling Categorical Data

Converting to Numerical Values

- We can transform the values using Scikit-Learn's OrdinalEncoder, which assigns a numeric value to each class

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

	ocean_proximity	
17606	<1H OCEAN	0
18632	<1H OCEAN	
14650	NEAR OCEAN	4
3230	INLAND	1
3555	<1H OCEAN	
19480	INLAND	
8879	<1H OCEAN	
13685	INLAND	
4937	<1H OCEAN	
4861	<1H OCEAN	

```
try:
    from sklearn.preprocessing import OrdinalEncoder
except ImportError:
    from future_encoders import OrdinalEncoder # Scikit-Learn < 0.20
```

```
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

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

Category	Value
<1H OCEAN	0
INLAND	1
ISLAND	2
NEAR BAY	3
NEAR OCEAN	4

This gives Ordinal values, but ordering/similarities are not needed

Handling Categorical Data

One-hot Encoding

- To fix this, create one binary attribute per category (e.g. a binary vector), where only one non-zero value exists, based on the category

	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

Category	Vector Value
<1H OCEAN	0
INLAND	0
ISLAND	0
NEAR BAY	0
NEAR OCEAN	1

Category	Vector Value
<1H OCEAN	1
INLAND	0
ISLAND	0
NEAR BAY	0
NEAR OCEAN	0

- This is called a ***<u>one-hot encoding</u>***, because only one value will be 1 (hot), which the others are 0 (cold).
- Avoids issues with ordering and similarity

Handling Categorical Data

One-hot Encoding

- One-hot encoding can be accomplished with Scikit-Learn using OneHotEncoder

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

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

- Create instance of encoder
- Apply encoding to categorical data

```
cat_encoder.categories_
```

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

- Shows what position in vector implies (e.g. which category)

Feature Scaling

Two approaches

- Machine learning algorithms do not perform well when the features/attributes have very different numerical scales
    - Total rooms varies from 2 to 39320
    - Median ages varies from 1 to 52
- ***Feature scaling***, modify the range of values while maintaining relative information, is needed. Two common approaches:
    - Min-max scaling (aka normalization)
    - Standardization

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Feature Scaling

Min-max scaling (or normalization)

- Min-max scaling (or normalization) involves:
    - Computing the min and max values of the attribute/feature
    - Subtract the min value from each instance of this attribute
    - Divide the result by the difference between the max and min values.
- Results in attributes/features that range from 0 to 1.

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
housing_rooms = housing[["total_rooms"]]
housing_rooms_scaled = scaler.fit_transform(housing_rooms)
print("Min: ", min(housing_rooms_scaled), "Max: ", max(housing_rooms_scaled))
```

```
Min:  [0.] Max:  [1.]
```

Feature Scaling

Standardization

- Steps for standardizing features:
    - Compute mean (or average) and standard deviation of feature/attribute
    - Subtract the mean value from each instance of this attribute
    - Divide the result by the standard deviation.
- Resulting attributes/features are zero mean and unit variance, but not bound to specific range.
- See StandardScaler in Scikit-Learn for a built-in function for accomplishing this.

Next Class

Evaluation and Metrics