Machine learning pipelines and an end to end project

[1]Church and Duncan Group,

[2]Information School, UC Berkeley

[3]School of Informatics, Computing and Engineering, Indiana University

EMAIL: James_DOT_Shanahan_AT_gmail_DOT_com

Reading Material

- **Machine learning pipelines in SKLearn**
  - Chapter 6, Python Machine Learning, Raschka and Mirjalili, 2017
  - Some simple pipelines with feature reduction (PCA) and selection (SelectKBest)
    - https://machinelearningmastery.com/automate-machine-learning-workflows-pipelines-python-scikit-learn/
    - http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

- **Reading material for significance testing**
  - Reference
    - https://towardsdatascience.com/statistical-significance-hypothesis-testing-the-normal-curve-and-p-values-93274fa32687
  - Notebook:
    - https://machinelearningmastery.com/use-statistical-significance-tests-interpret-machine-learning-results/

Homework Assignments

- **Potential HW Tasks Notes**

1. Try a Support Vector Machine regressor (`sklearn.svm.SVR`), with various hyperparameters such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best `SVR` predictor perform?
2. Try replacing `GridSearchCV` with `RandomizedSearchCV`.
3. Try adding a transformer in the preparation pipeline to select only the most important attributes.
4. Try creating a single pipeline that does the full data preparation plus the final prediction.
5. Automatically explore some preparation options using `GridSearchCV`.

Unit testing

Testing: not convinced yet?

Precision	Recall	F-score
0.34	0.31	0.30

```
f1 = fscore(p, r)
min_bound, max_bound = sorted([p, r])

assert min_bound <= f1 <= max_bound
```

Feature reduction (PCA) + Feature selectionSelectKBest

Pipeline 2: Feature Extraction and Modeling

Feature extraction is another procedure that is susceptible to data leakage.

Like data preparation, feature extraction procedures must be restricted to the data in your training dataset.

The pipeline provides a handy tool called the FeatureUnion which allows the results of multiple feature selection and extraction procedures to be combined into a larger dataset on which a model can be trained. Importantly, all the feature extraction and the feature union occurs within each fold of the cross validation procedure.

The example below demonstrates the pipeline defined with four steps:

1. Feature Extraction with Principal Component Analysis (3 features)
2. Feature Extraction with Statistical Selection (6 features)
3. Feature Union
4. Learn a Logistic Regression Model

https://machinelearningmastery.com/automate-machine-learning-workflows-pipelines-python-scikit-learn/

The pipeline is then evaluated using 10-fold cross validation.

```
# Create a pipeline that extracts features from the data then creates a model
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest
# load data
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-di
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# create feature union
features = []
features.append(('pca', PCA(n_components=3)))
features.append(('select_best', SelectKBest(k=6)))
feature_union = FeatureUnion(features)
# create pipeline
estimators = []
estimators.append(('feature_union', feature_union))
estimators.append(('logistic', LogisticRegression()))
model = Pipeline(estimators)
# evaluate pipeline
seed = 7
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

https://machinelearningmastery.com/automate-machine-learning-workflows-pipelines-python-scikit-learn/

Running the example provides a summary of accuracy of the pipeline on the dataset.

```
0.776042378674
```

Select features using the biggest coefficients from a linear SVM

```
print('=' * 80)
print("LinearSVC with L1-based feature selection")
# The smaller C, the stronger the regularization.
# The more regularization, the more sparsity.
results.append(benchmark(Pipeline([
  ('feature_selection', SelectFromModel(LinearSVC(penalty="l1", dual=False,
                                                  tol=1e-3))),
  ('classification', LinearSVC(penalty="l2"))])))
```

If **threshold==**None and if the estimator has a parameter penalty set to l1, either explicitly or implicitly (e.g, Lasso), the threshold used is 1e-5.

class sklearn.feature_selection. **SelectFromModel** (*estimator, threshold=None, prefit=False, norm_order=1*) [source]

Meta-transformer for selecting features based on importance weights.

New in version 0.17.

Parameters: **estimator** : object

The base estimator from which the transformer is built. This can be both a fitted (if `prefit` is set to True) or a non-fitted estimator. The estimator must have either a `feature_importances_` or `coef_` attribute after fitting.

threshold : string, float, optional default None

The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median" (resp. "mean"), then the `threshold` value is the median (resp. the mean) of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. If None and if the estimator has a parameter penalty set to l1, either explicitly or implicitly (e.g, Lasso), the threshold used is 1e-5. Otherwise, "mean" is used by default.

http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

Machine learning pipelines and an end to end project

[1]Church and Duncan Group,

[2]Information School, UC Berkeley

[3]School of Informatics, Computing and Engineering, Indiana University

EMAIL: James_DOT_Shanahan_AT_gmail_DOT_com

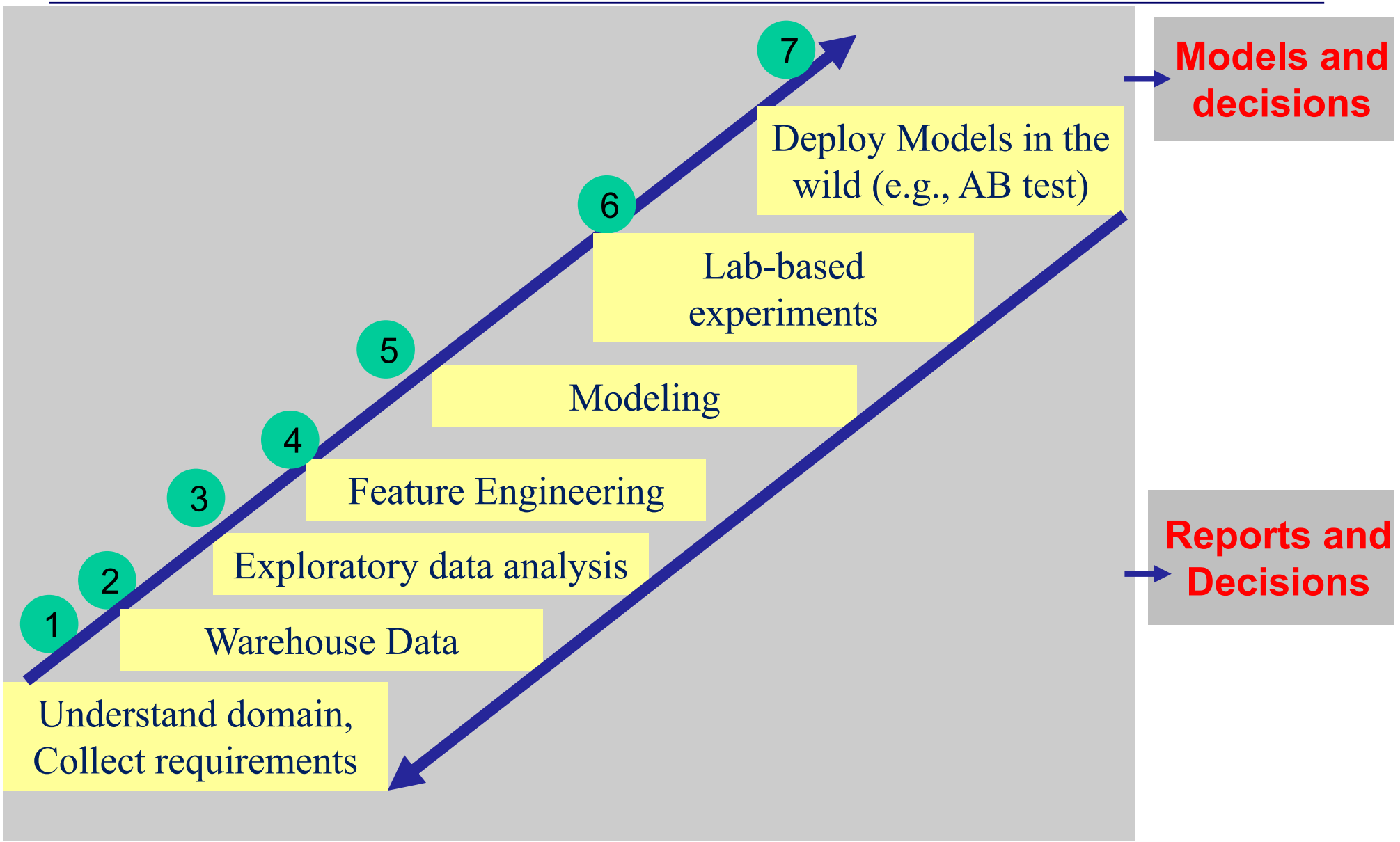
Outline

- **Introduction**
- **End to end ML project in SKLearn**
  - Problem definition and data
  - EDA
  - SKLearn Pipelines for prepping data
  - Full learning pipeline
  - Finetune model:
    - GridSearch versus Random
- **Which model is better? Significance Tests**
- **Summary**

Outline

- **Introduction**
- **End to end ML project in SKLearn**
  - Problem definition and data
  - EDA
  - SKLearn Pipelines for prepping data
  - Full learning pipeline
  - Finetune model:
    - GridSearch versus Random
- **Which model is better? Significance Tests**
- **Summary**

Typical Machine Learning Pipeline

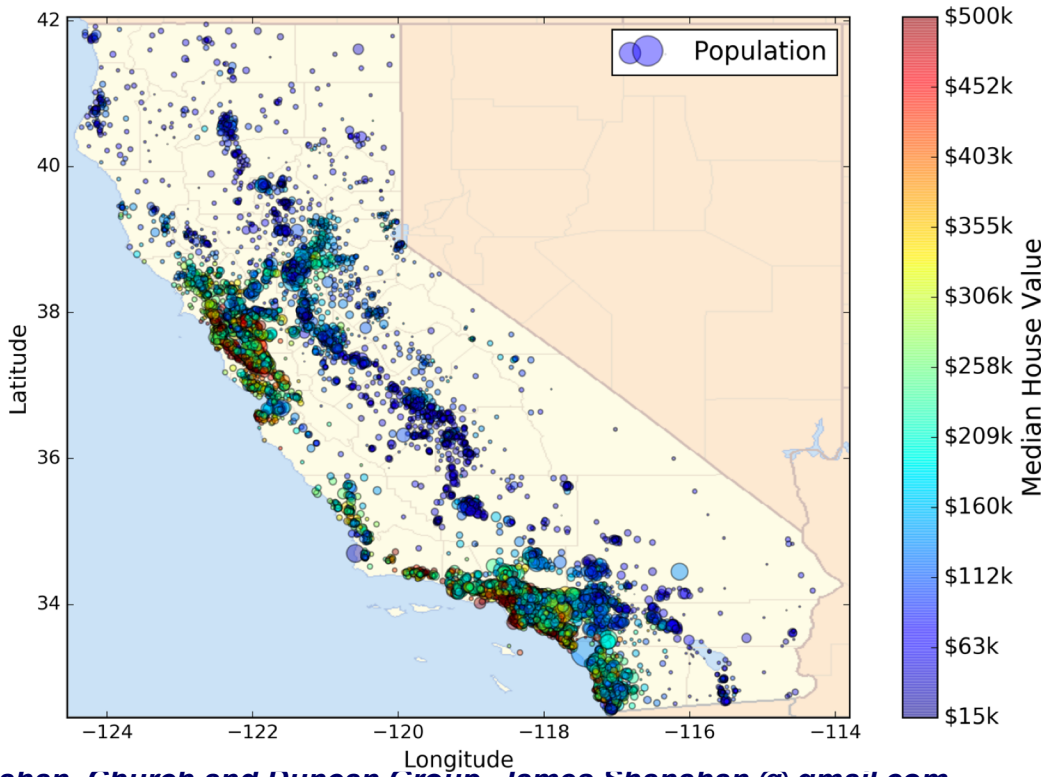


End to end project

- **An example project end to end, pretending to be a recently hired data scientist in a real estate company**
  - 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.

Task

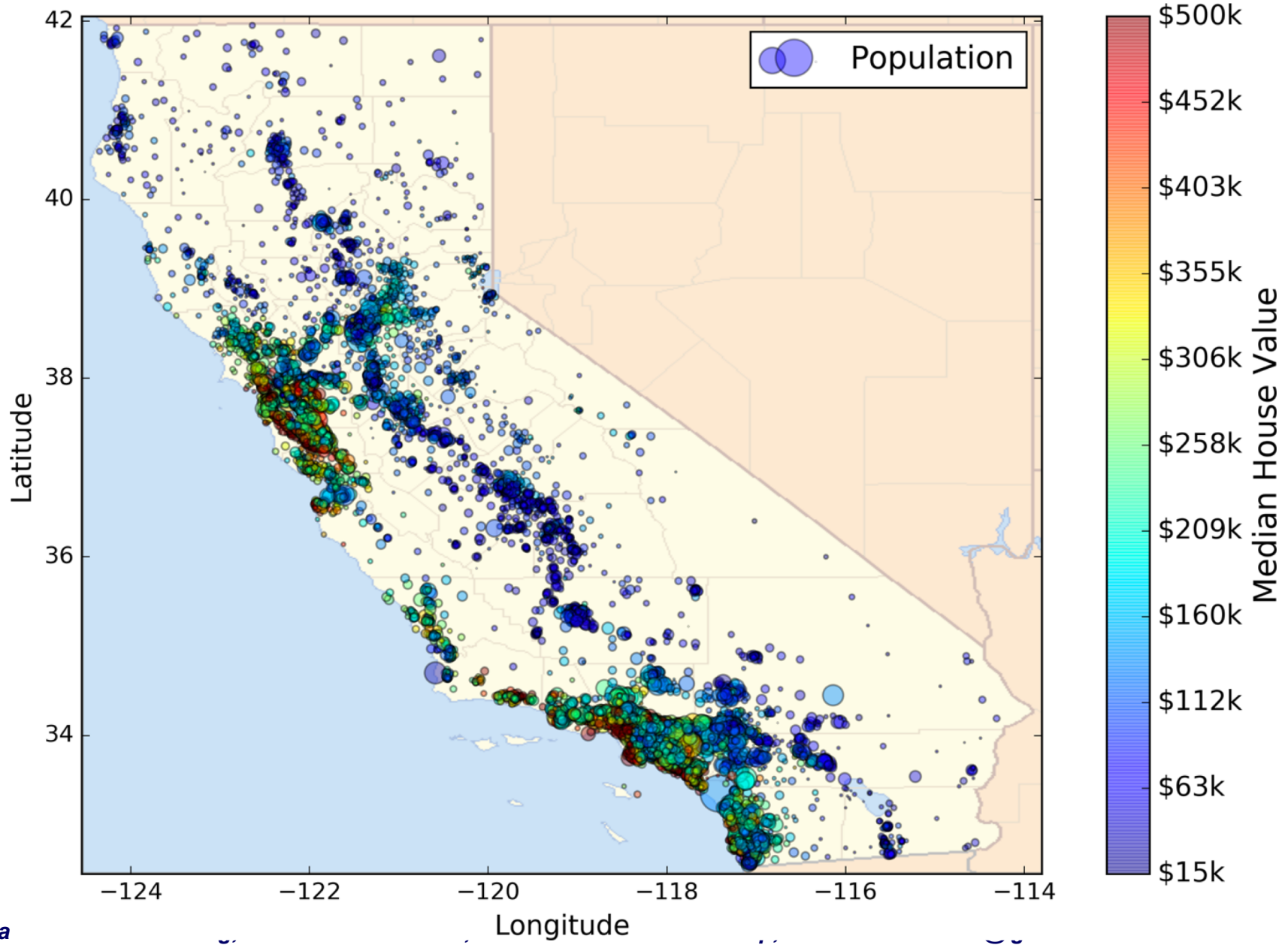
- The first task you are asked to perform is to build a model of housing prices in California using the California census data.
- Predict district's median housing price
- District housing prices are currently estimated manually by experts (15% error)
- Dataset



California Housing Prices dataset from the StatLib Repo

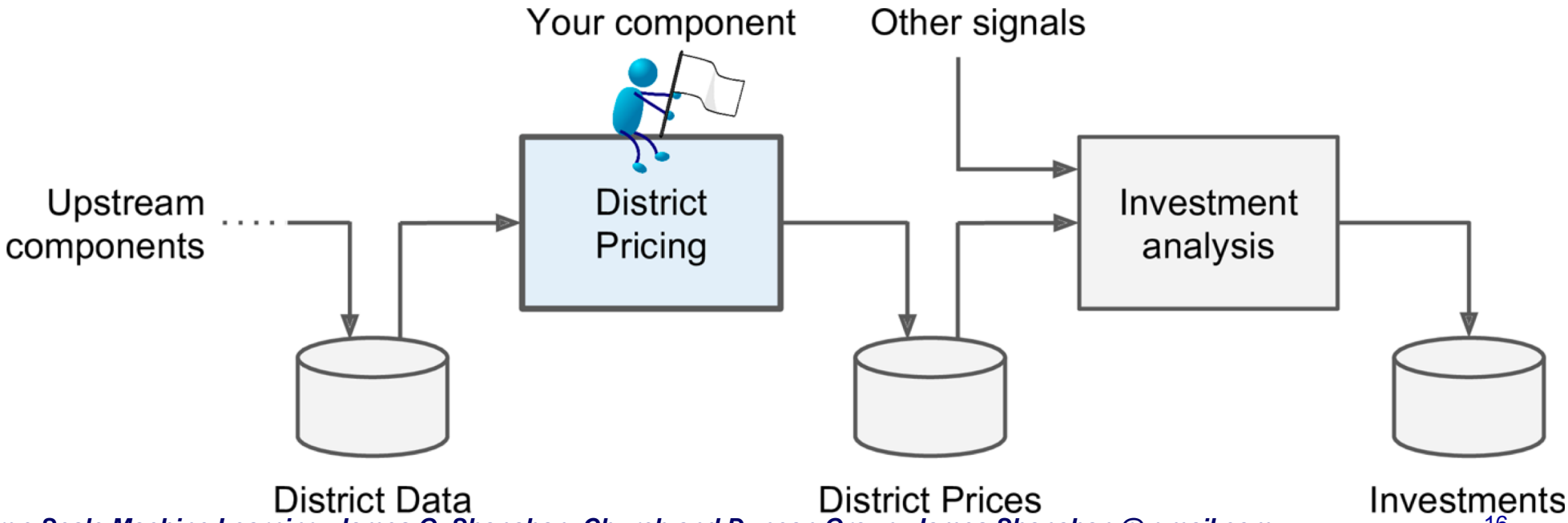
- **This dataset was based on data from the 1990 California census. It is not exactly recent!**
- **Data based Geo units (600-3,000 people)**
  - This data has metrics such as the population, median income, median housing price, and so on for each block group in California.
  - Block groups are the smallest 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 just call them “districts” for short.
- **Added a categorical attribute and removed a few features for teaching purposes.**

California Housing Prices dataset from the StatLib Repo



Frame the problem: Project: worth investing in a given area?

- **How does the company expect to use and benefit from this model?**
  - Your boss answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system along with many other signals. This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.



Select a Performance Measure

- **Frame as a regression problem so use these performance measures**
- **Root Mean Square Error (RMSE).**

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

- **Mean Absolute Error (also called the Average Absolute Deviation**

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

- **Mean absolute percentage error (MAPE)**

$$\text{MAPE}\ (\mathbf{X}, h) = \frac{100}{m}\sum_{i=1}^{m}\frac{\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|}{y^{(i)}}$$

Make a project

- **~/Project/CAHousing**
- **Download data**
  - https://www.dropbox.com/s/14measow06ptyt0/housing.tgz?dl=0
  - 400K compressed (small!)

Outline

- **Introduction**
- **End to end ML project in SKLearn**
  - Problem definition and data
  - EDA
  - SKLearn Pipelines for prepping data
  - Full learning pipeline
  - Finetune model:
    - GridSearch versus Random
- **Which model is better? Significance Tests**
- **Summary**

CAHousing: 20640, 10

- **Read data into a Pandas dataframe**

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

```
housing = load_housing_data()
housing.head()
```

Target

	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

CAHousing: 20640, 10

```
housing.info()
```

Notice that the `total_bedrooms` attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature. We will need to take care of this later.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude                 20640 non-null float64
latitude                  20640 non-null float64
housing_median_age        20640 non-null float64
total_rooms               20640 non-null float64
total_bedrooms            20433 non-null float64
population                20640 non-null float64
households                20640 non-null float64
median_income             20640 non-null float64
median_house_value        20640 non-null float64
ocean_proximity           20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Attributes: Numerical, plus one categorical

```
housing["ocean_proximity"].value_counts()
```

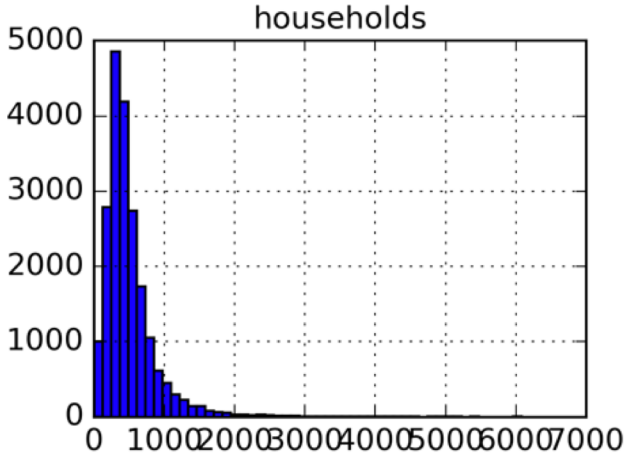
```
<1H OCEAN      9136
INLAND         6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND            5
Name: ocean_proximity, dtype: int64
```

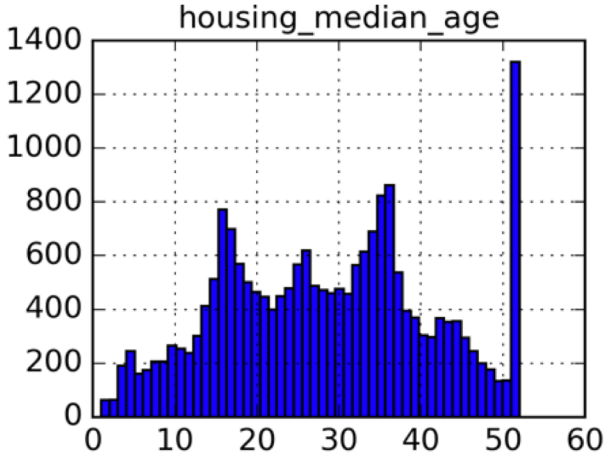
Summary of each numerical attribute

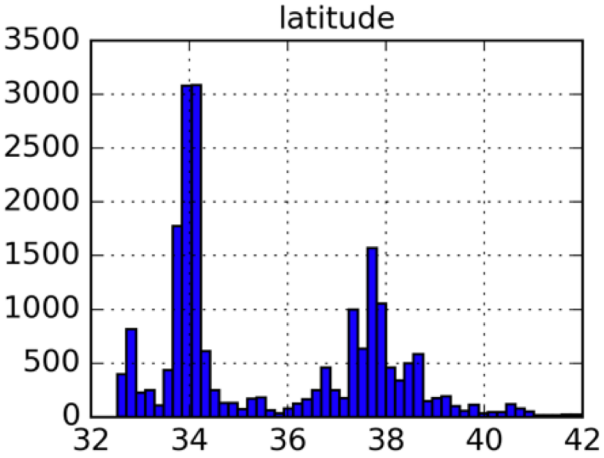
```
In [8]: housing.describe()
```

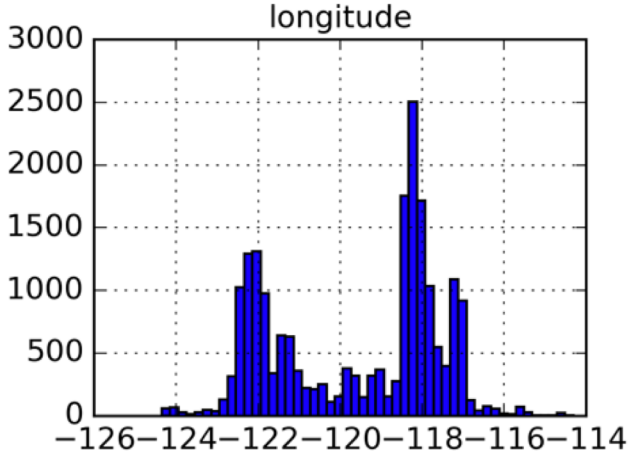
Out[8]:

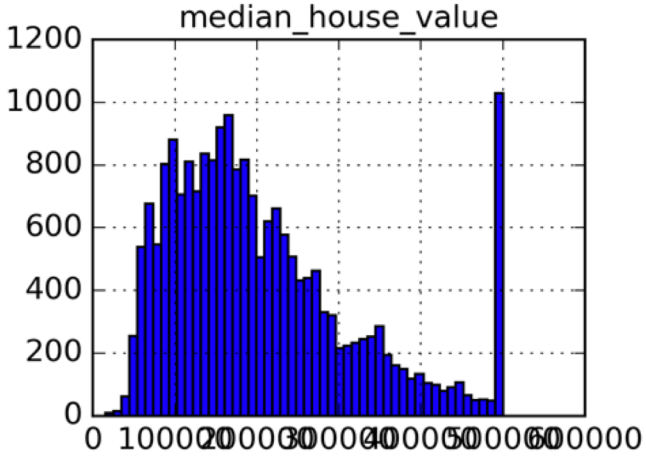
	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.38507(
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000(
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000(
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000(
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000(

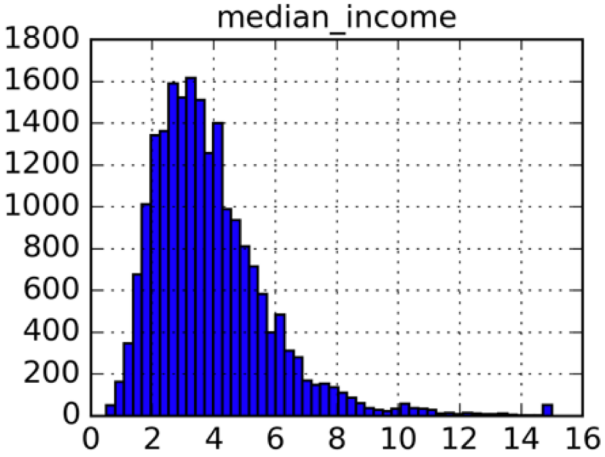

households
5000
4000
3000
2000
1000
0
0 1000 2000 3000 4000 5000 6000 7000

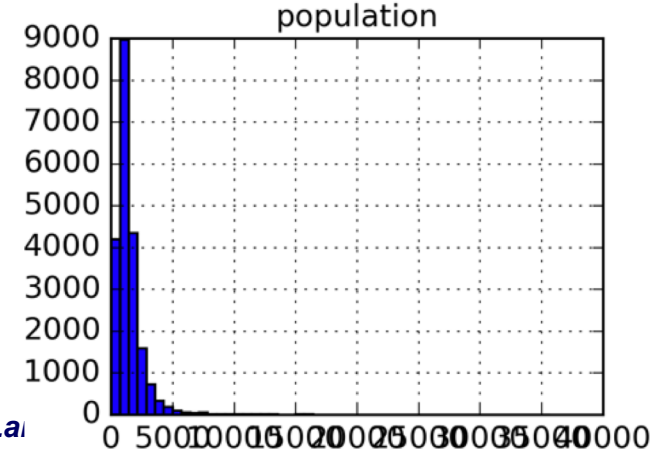

housing_median_age
1400
1200
1000
800
600
400
200
0
0 10 20 30 40 50 60

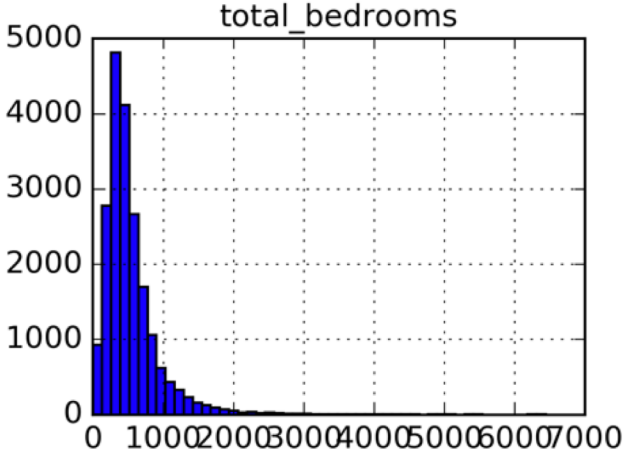

latitude
3500
3000
2500
2000
1500
1000
500
0
32 34 36 38 40 42

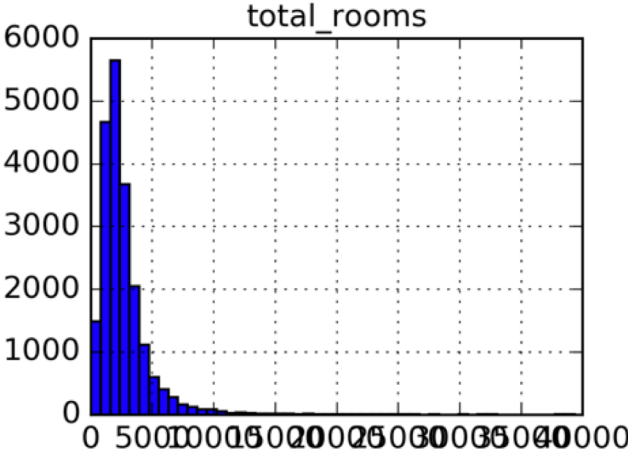

longitude
3000
2500
2000
1500
1000
500
0
−126 −124 −122 −120 −118 −116 −114

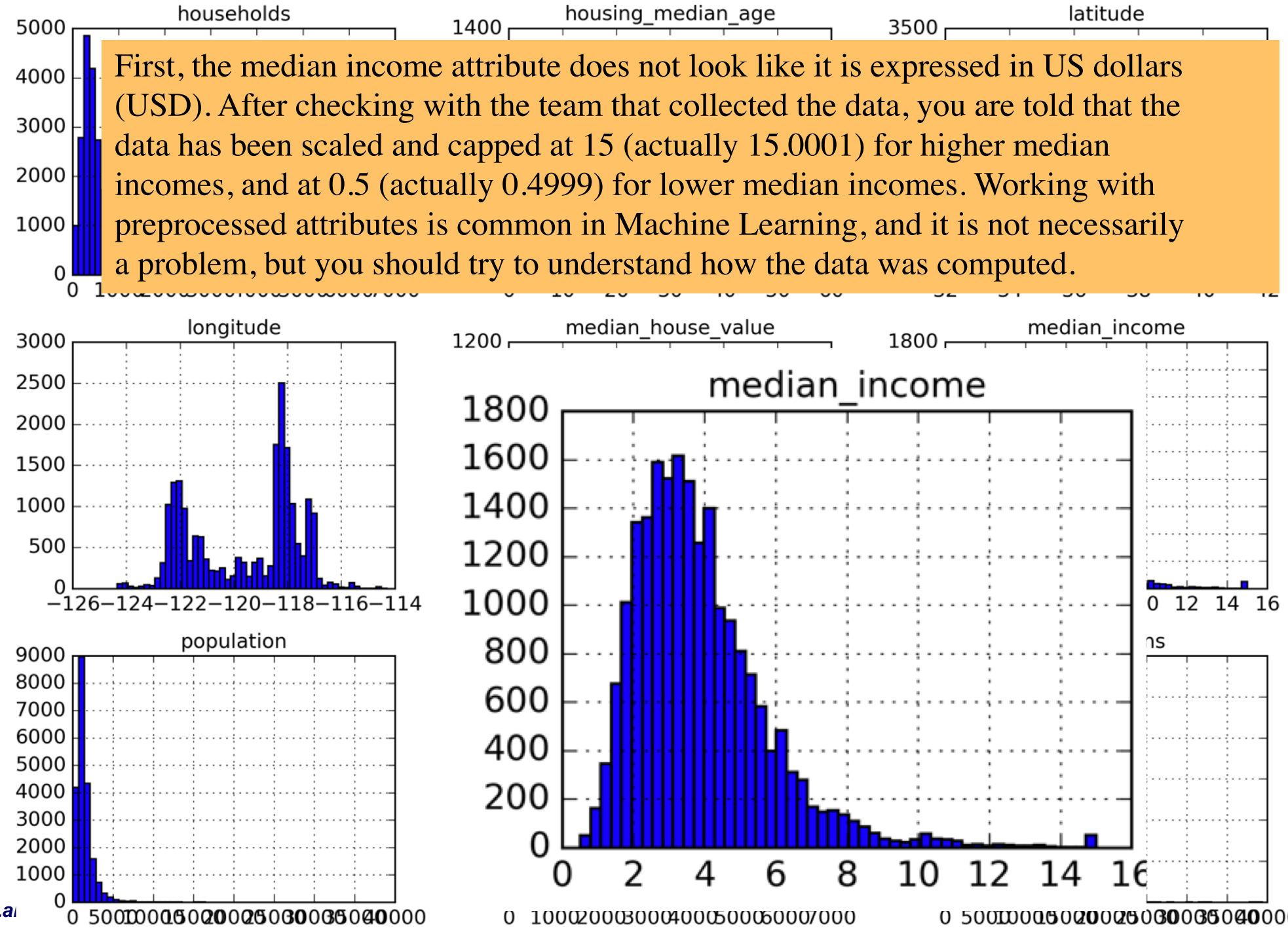

median_house_value
1200
1000
800
600
400
200
0
0 100000 200000 300000 400000 500000 600000


median_income
1800
1600
1400
1200
1000
800
600
400
200
0
0 2 4 6 8 10 12 14 16

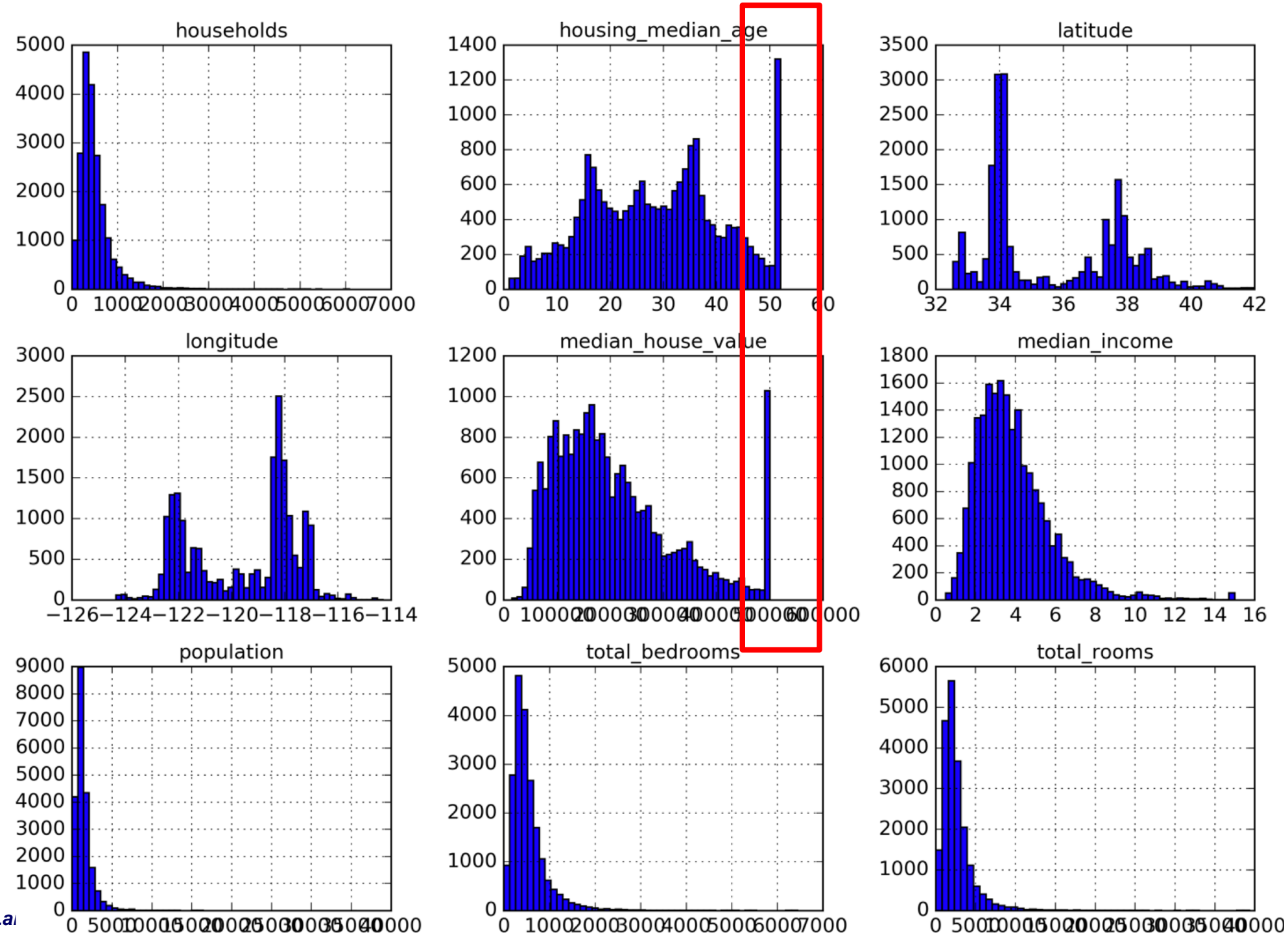

population
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0 5000 10000 15000 20000 25000 30000 35000 40000


total_bedrooms
5000
4000
3000
2000
1000
0
0 1000 2000 3000 4000 5000 6000 7000


total_rooms
6000
5000
4000
3000
2000
1000
0
0 5000 10000 15000 20000 25000 30000 35000 40000

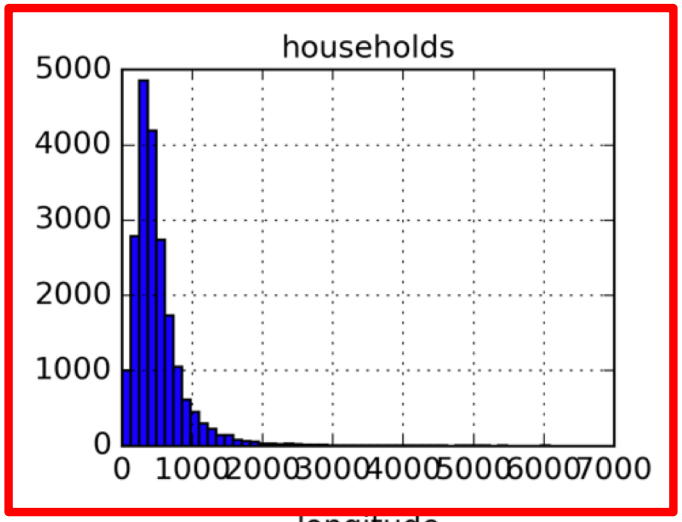


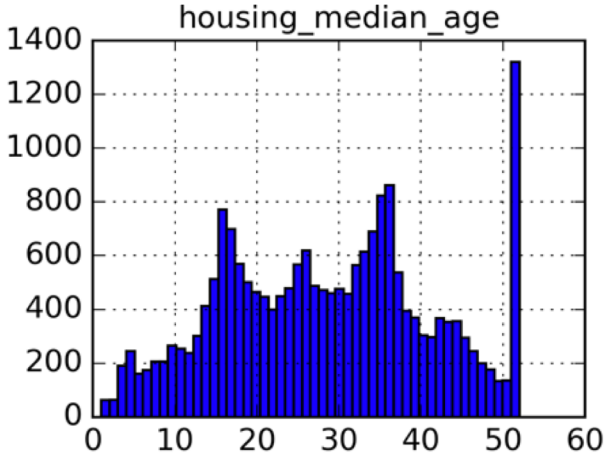
First, the median income attribute does not look like it is expressed in US dollars (USD). After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually 15.0001) for higher median incomes, and at 0.5 (actually 0.4999) for lower median incomes. Working with preprocessed attributes is common in Machine Learning, and it is not necessarily a problem, but you should try to understand how the data was computed.

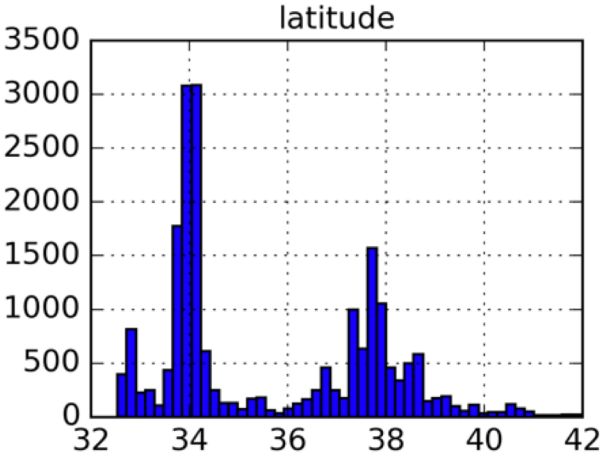

households
housing_median_age
latitude
longitude
median_house_value
median_income
population
total_bedrooms
total_rooms

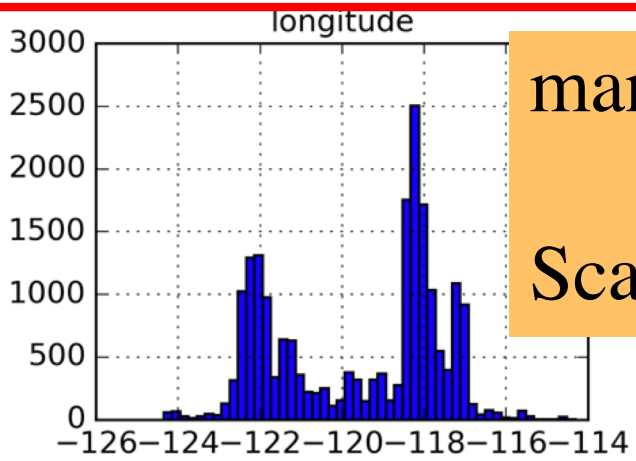
housing median age AND the median house value

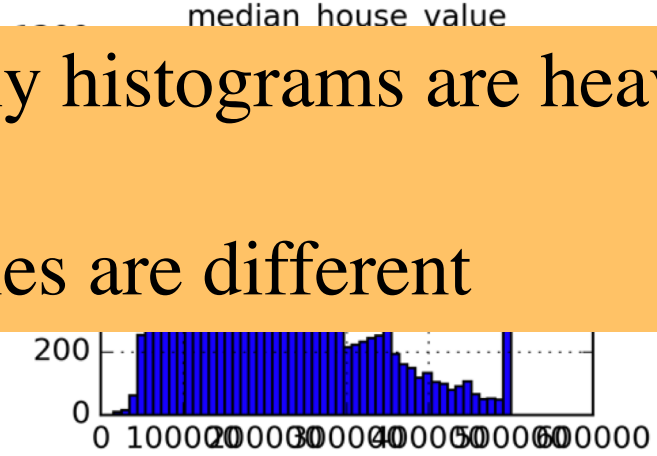
- **The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit.**
- **You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond $500,000, then you have mainly two options:**
  - a. Collect proper labels for the districts whose labels were capped.
  - b. Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond $500,000).

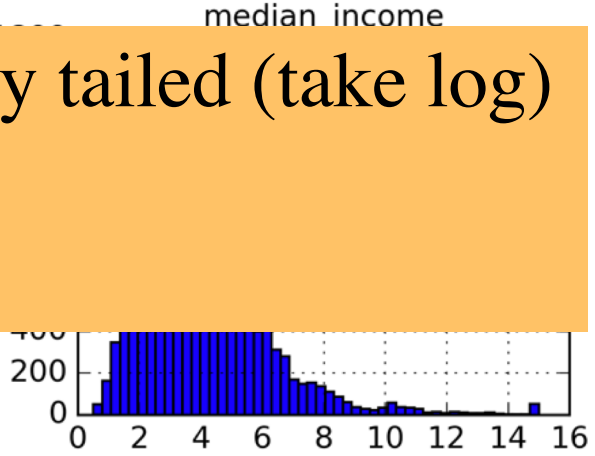






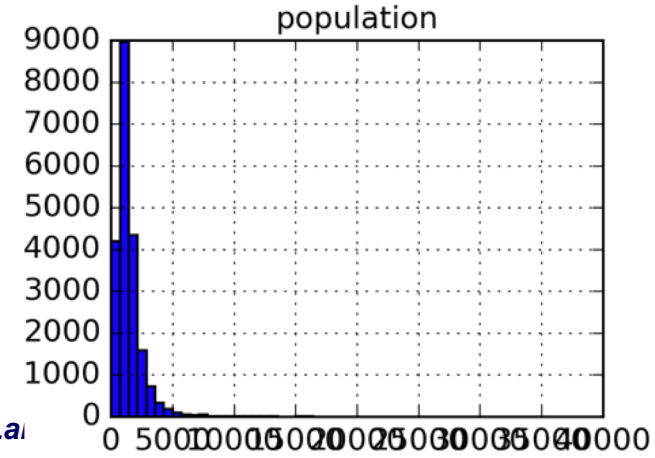


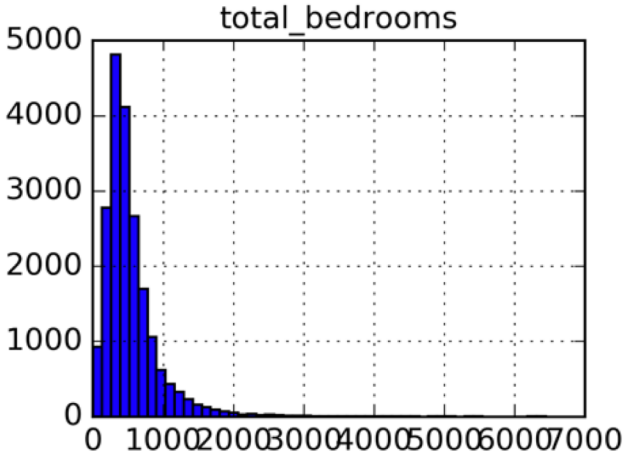


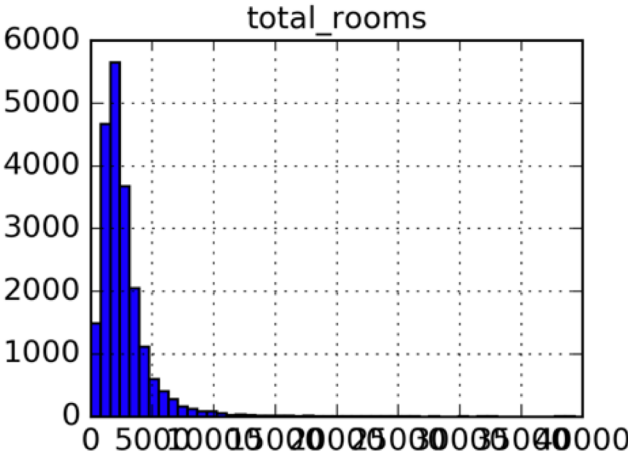


many histograms are heavy tailed (take log)

Scales are different







Create a test set

- So far we have considered purely random sampling methods.
- This 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.
- Do stratified sampling based on a particular dimension

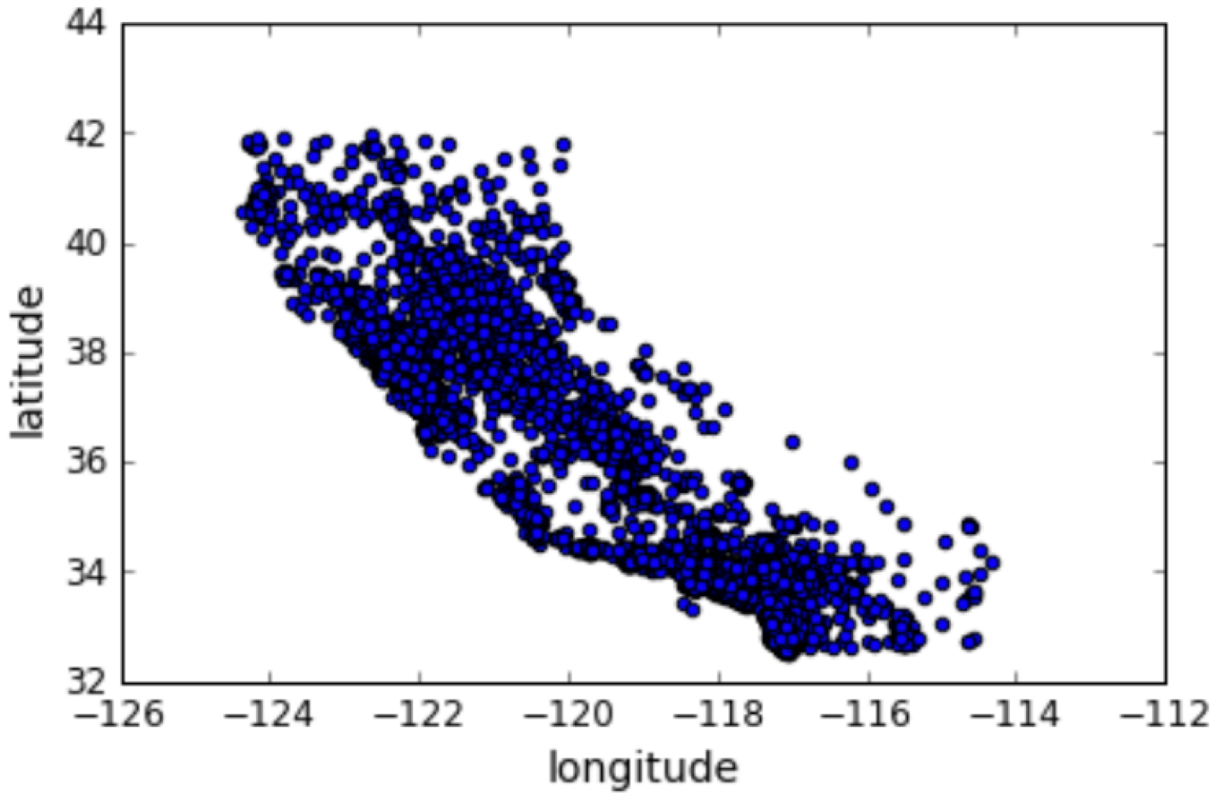
```
from sklearn.model_selection import train_test_split

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

Visualize the data: location of sample

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

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

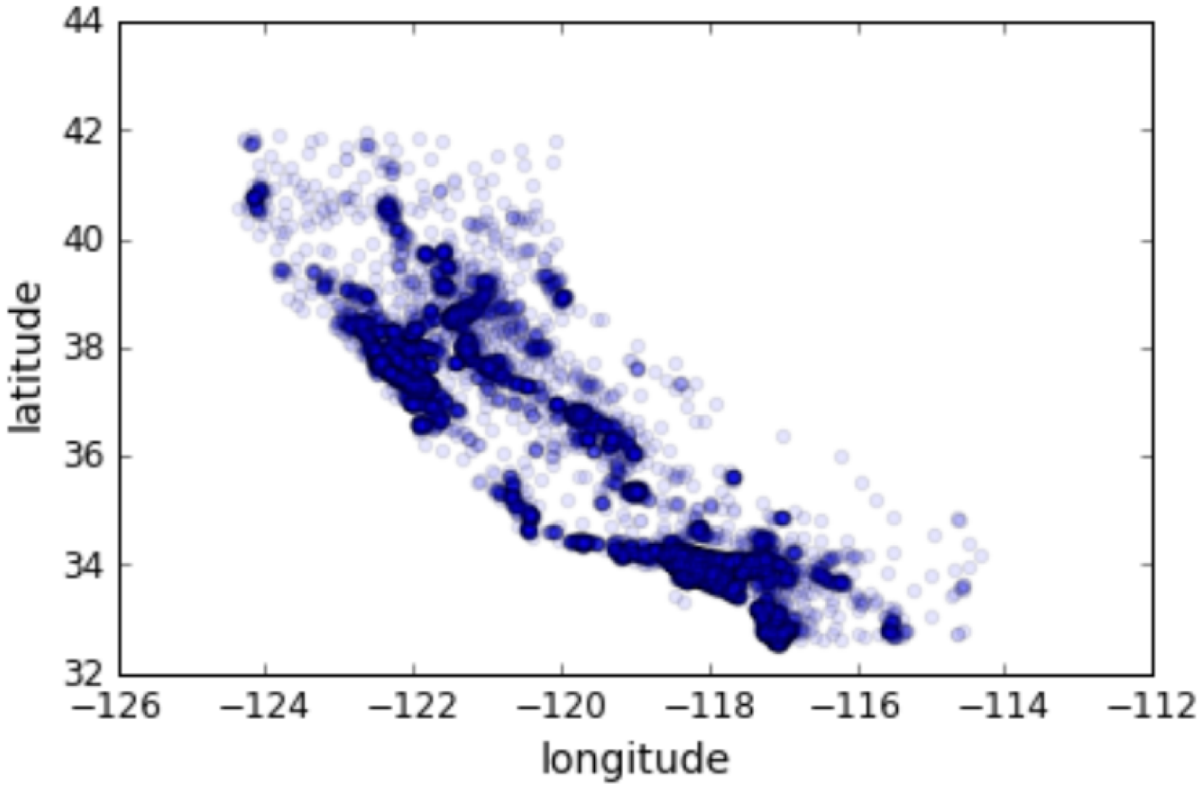
```
Saving figure bad_visualization_plot
```



location of sample: see higher density easier

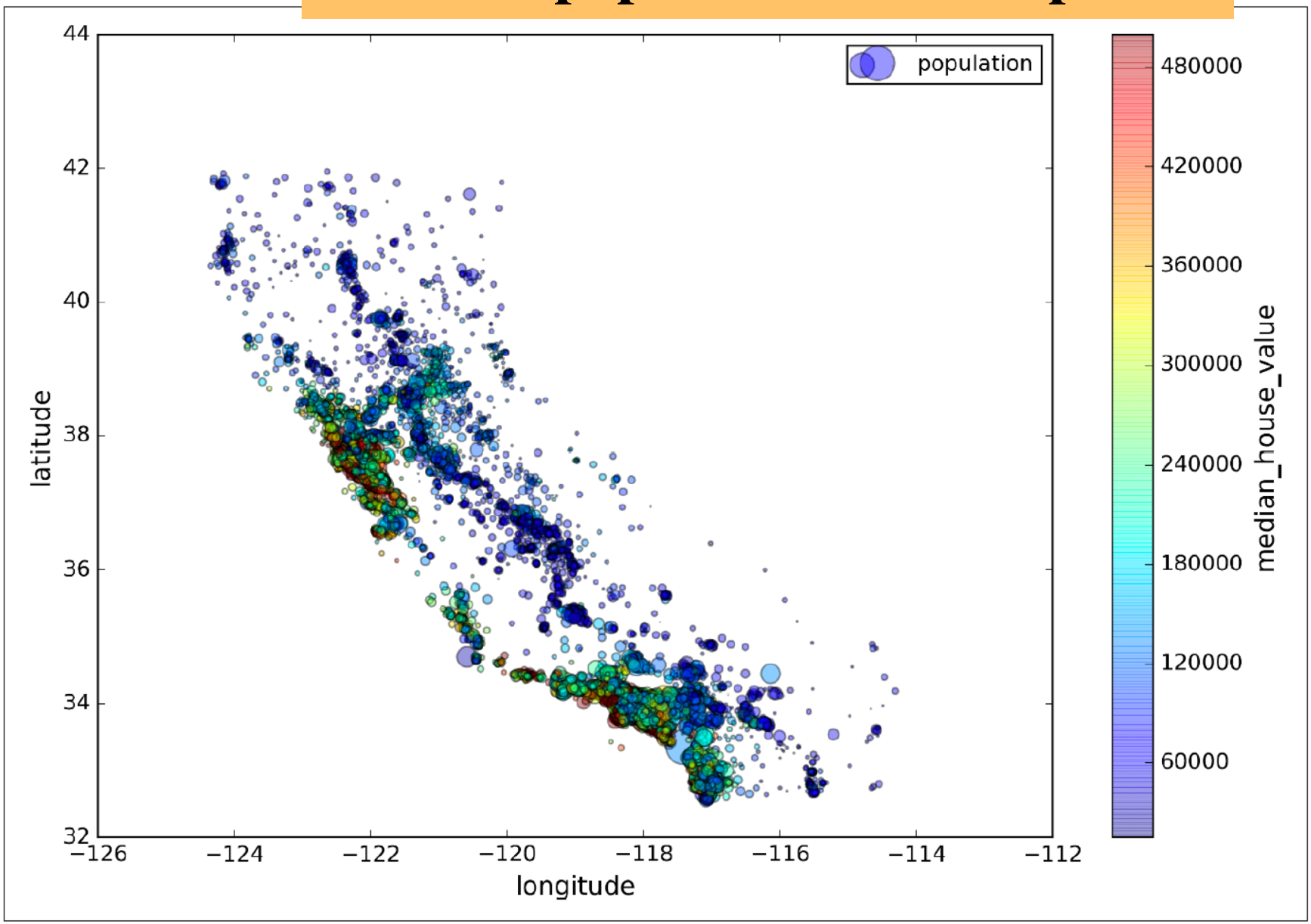
```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

```
Saving figure better_visualization_plot
```



```
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,
)
plt.legend()
```

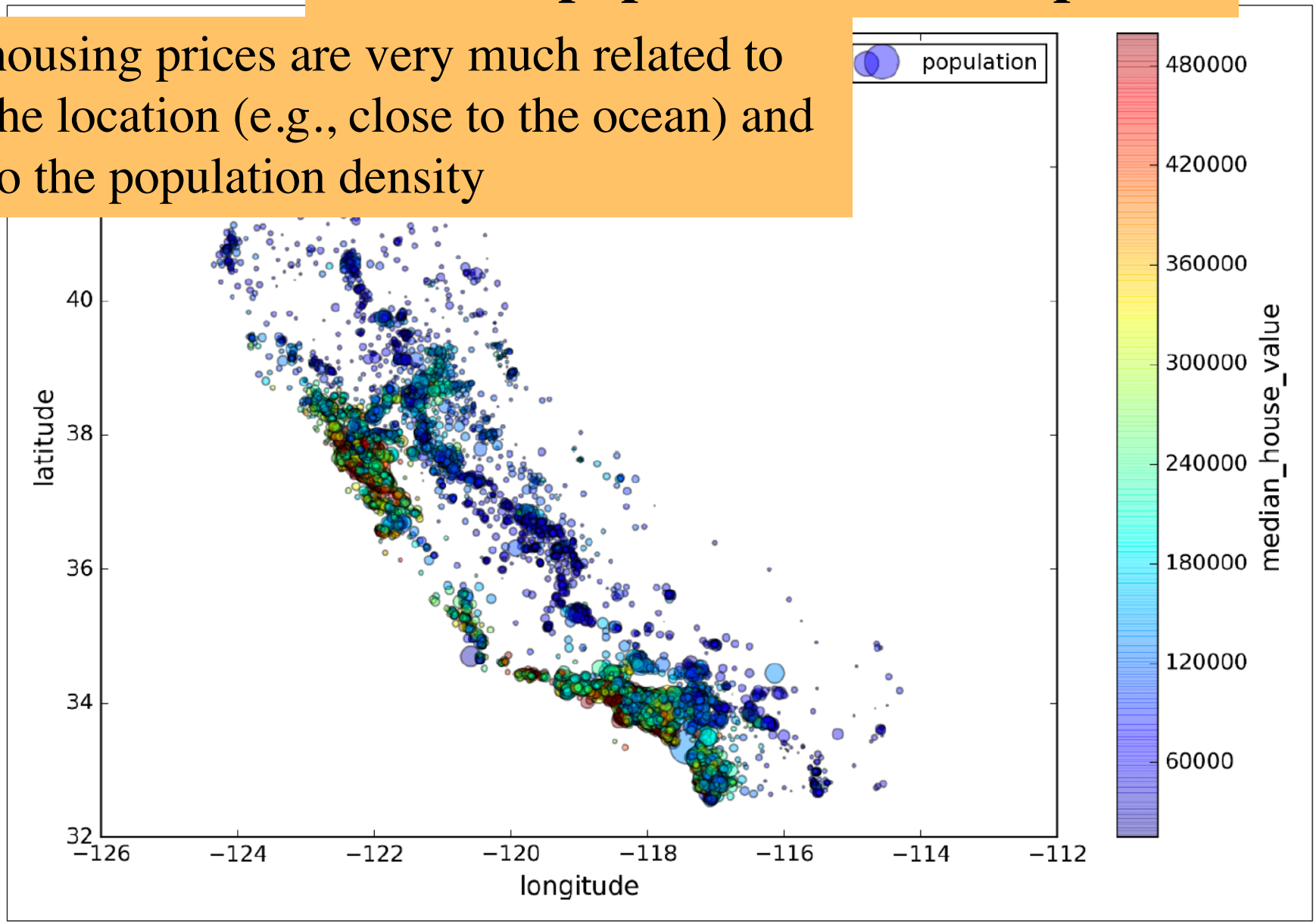
district's population versus price



```
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,
)
plt.legend()
```

district's population versus price

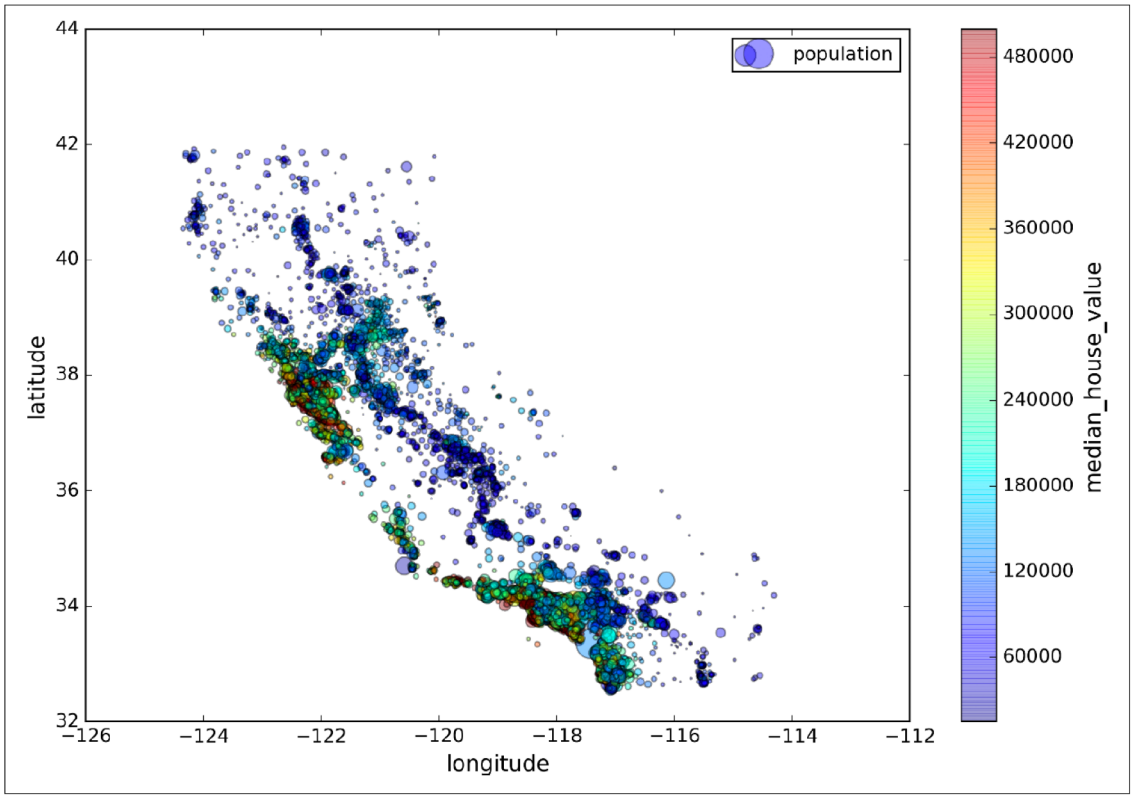
housing prices are very much related to the location (e.g., close to the ocean) and to the population density



district's population versus price

- 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

```
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,
)
plt.legend()
```



Looking for 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:

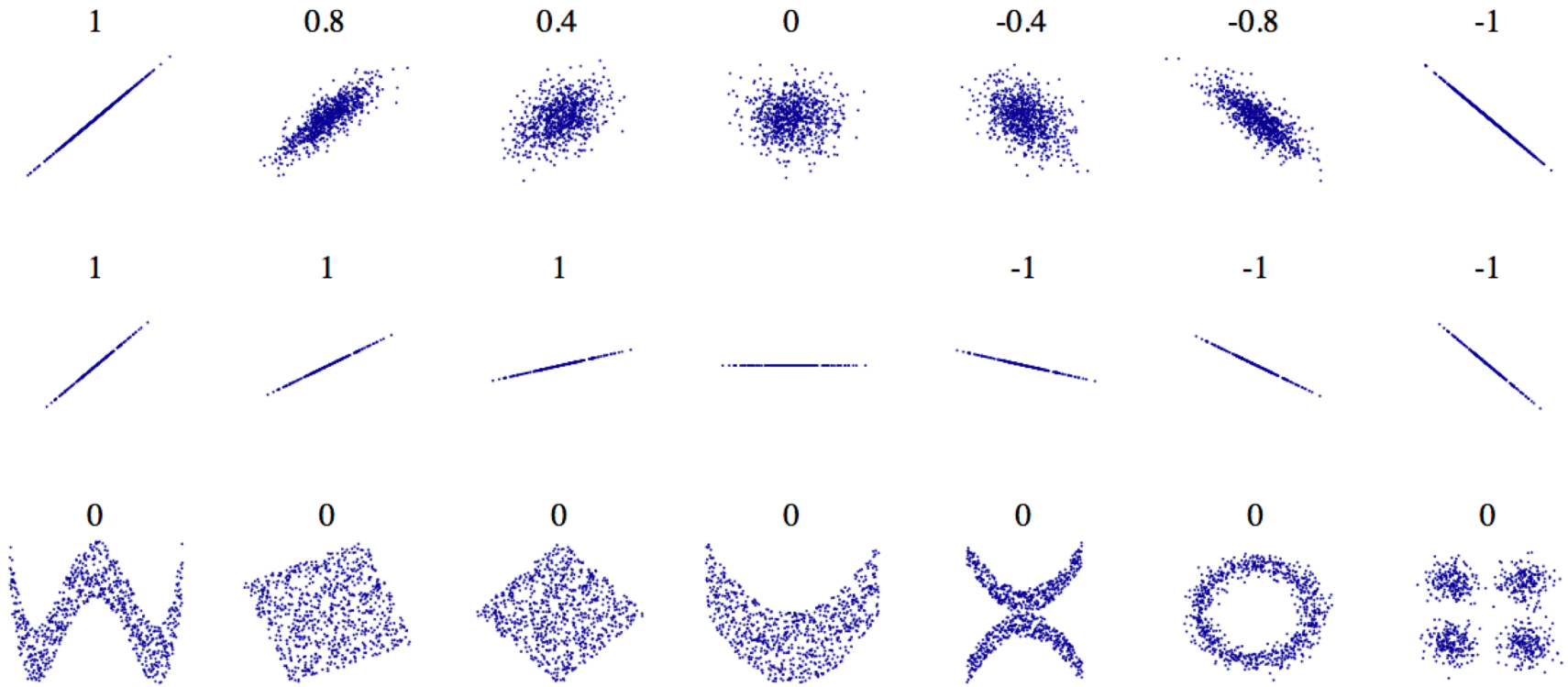
```
corr_matrix = housing.corr()
```

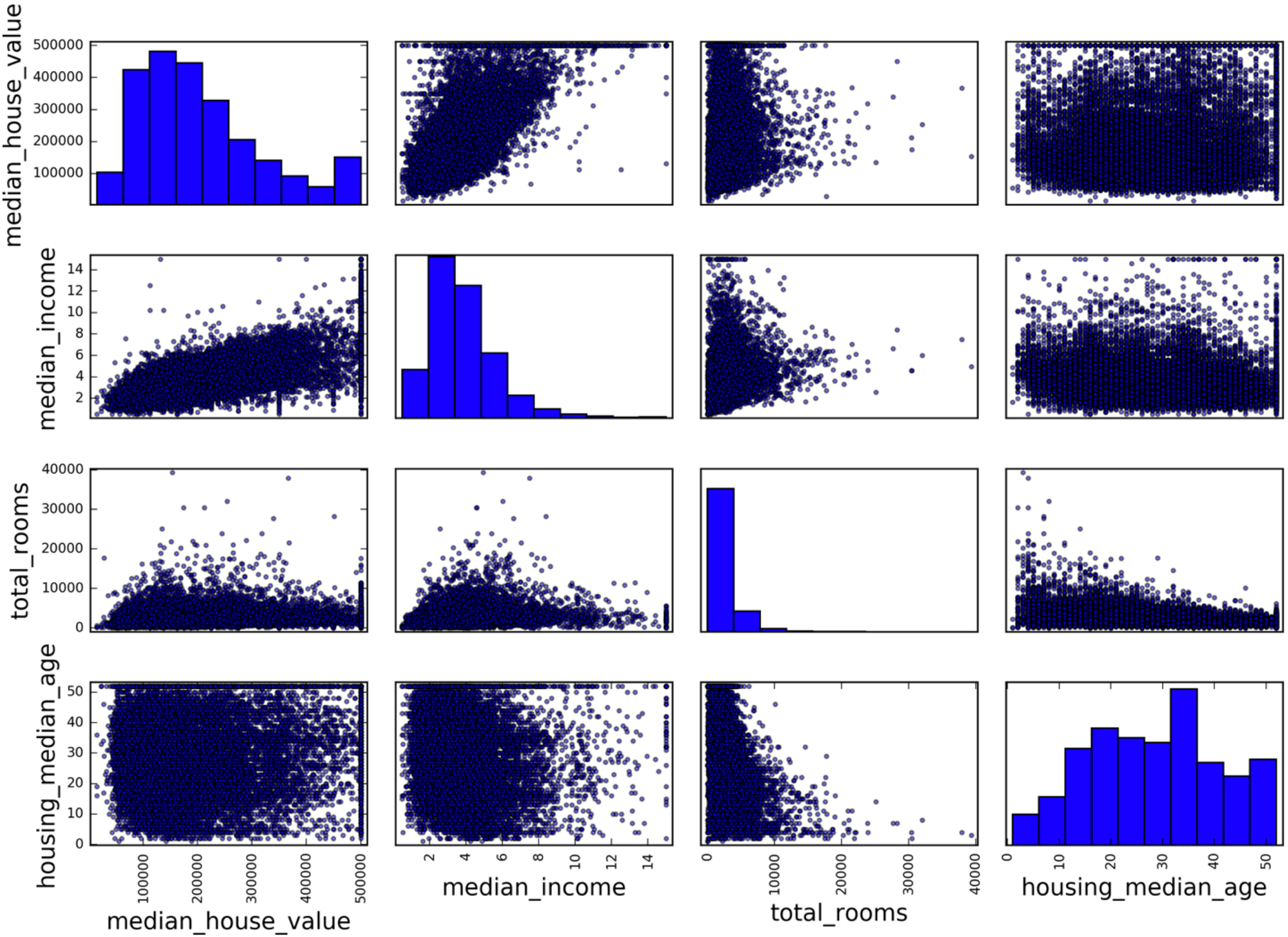
Now let's look at how much each attribute correlates with the median house value:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

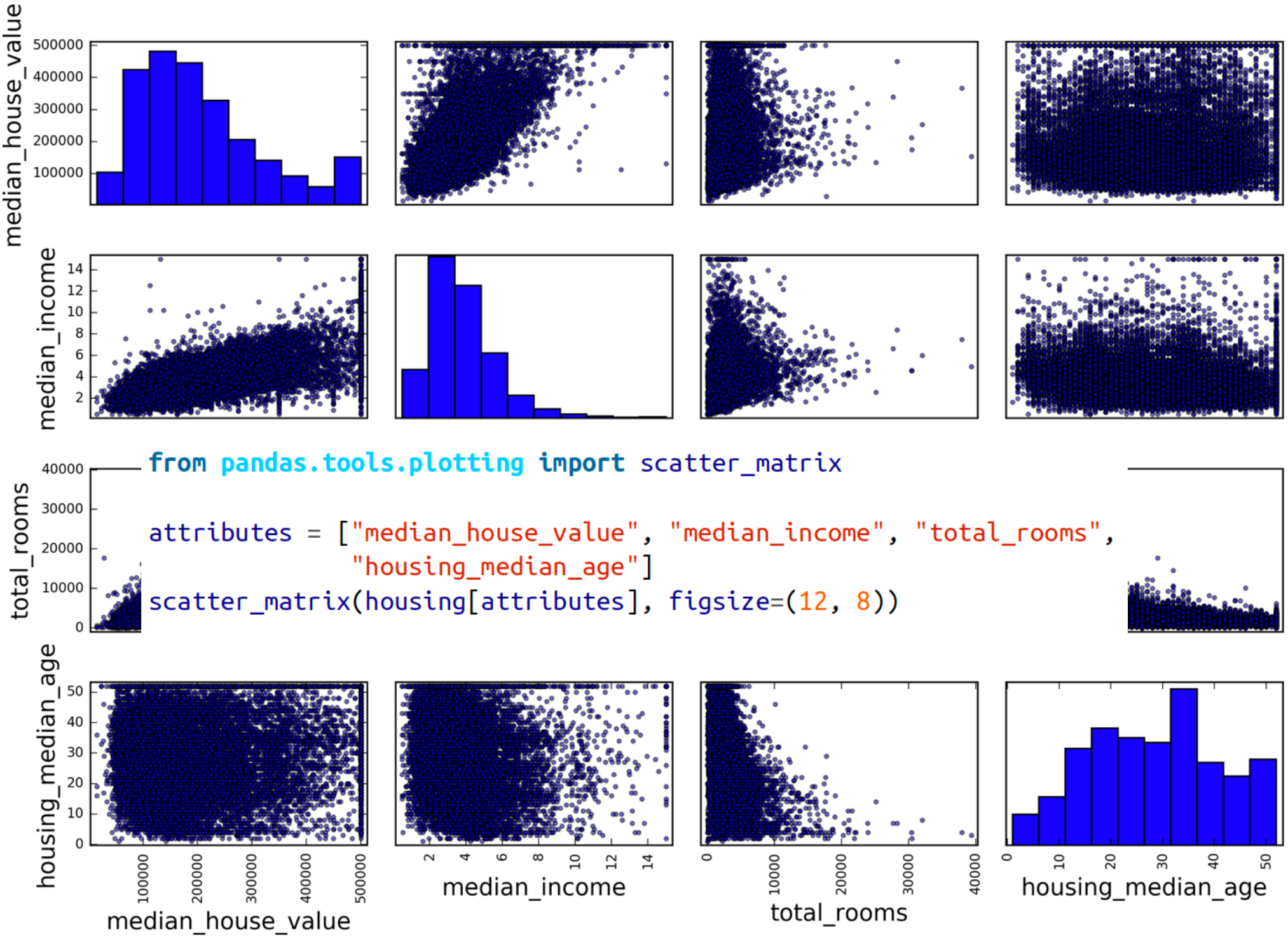
Standard correlation

- Note how all the plots of the bottom row have a correlation coefficient equal to zero despite the fact that their axes are clearly not independent: these are examples of nonlinear relationships.
- Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

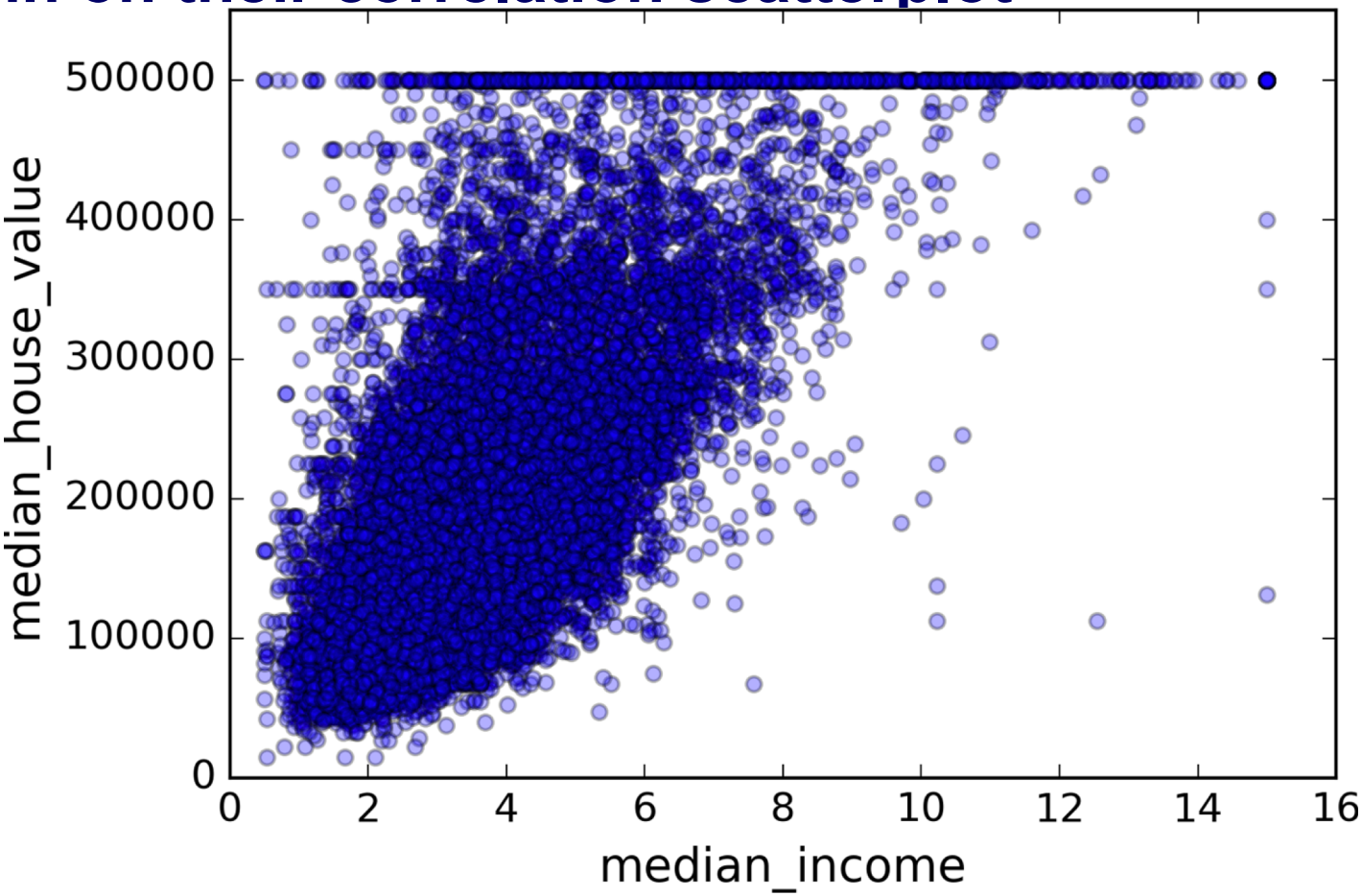



median_house_value
median_income
total_rooms
housing_median_age
500000
400000
300000
200000
100000
14
12
10
8
6
4
2
40000
30000
20000
10000
0
50
40
30
20
10
0
100000
200000
300000
400000
500000
2
4
6
8
10
12
14
0
10000
20000
30000
40000
0
10
20
30
40
50

```
from pandas.tools.plotting import scatter_matrix

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



- **The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot**



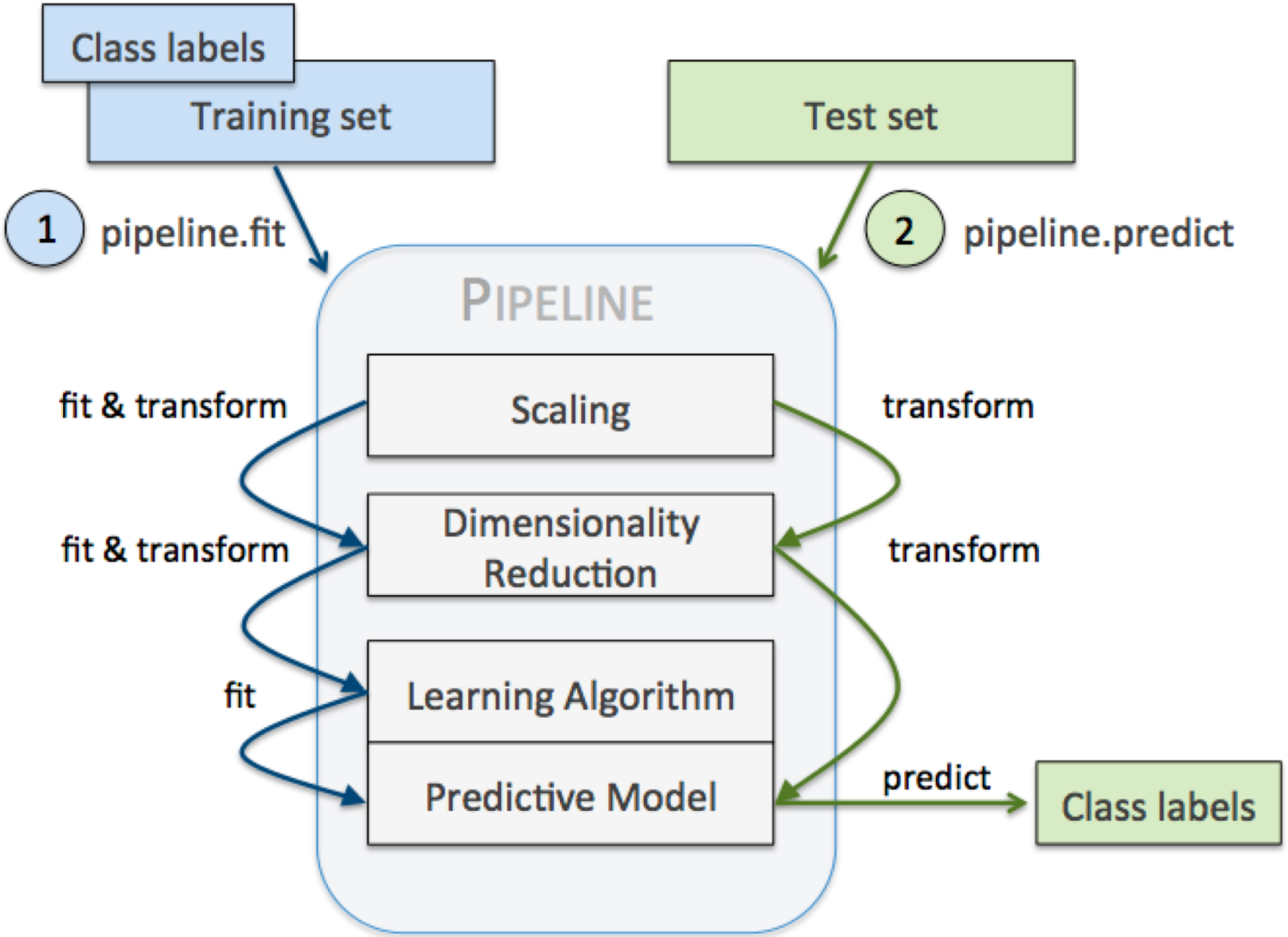
Outline

- **Introduction**
- **End to end ML project in SKLearn**
  - Problem definition and data
  - EDA
  - SKLearn Pipelines for prepping data
  - Full learning pipeline
  - Finetune model:
    - GridSearch versus Random
- **Which model is better? Significance Tests**
- **Summary**

Machine learning pipelines

- **A sequence of data processing components is called a data pipeline .**
- **Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.**
- **The interface between components is simply the data store.**
  - Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store, and then some time later the next component in the pipeline pulls this data and spits out its own output, and so on.
  - Each component is fairly self-contained: the interface between components is simply the data store.

Sample SKLearn Pipeline: APIs for fit, predict



FeatureUnion pipeline

- This post was very handy in finding the FeatureUnion pipeline that can help you break down your preprocessing and recombine it again. So you can combine two pipelines:
  - one that extracts categorical variables and applies one hot encoding, and then scales the outputs to -1 / 1
  - one that extracts quantitative variables, imputes missing values with each variable's mean and then applies mean scaling into one and have a little pipeline that is ready to preprocess your data.
  - This can then be further combined in a larger pipeline with other steps like PCA followed by your classifier.

http://nbviewer.jupyter.org/github/krosaen/ml-study/blob/master/kaggle/forest-cover-type-prediction2/forest-cover-type-prediction2.ipynb#Tuning-3-models'-hyperparameters

But first let's revert to a clean training set (by copying `strat_train_set` once again), and let's separate the predictors and the labels since we 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()
```

Prep data for ML using pipelines

- Get rid of the corresponding districts.
- Get rid of the whole attribute.
- Set the values to some value (zero, the mean, the median, etc.).

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

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

If you choose option 3, you should compute the median value on the training set, and use it to fill the missing values in the training set, but also don't forget to save the median value that you have computed

- **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., an `imputer` is an estimator). The estimation itself is performed by the `fit()` method, and it takes only 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 an `imputer`'s `strategy`), and it must be set as an instance variable (generally via a constructor parameter).
  - *Transformers*. Some estimators (such as an `imputer`) can also transform a dataset; these are called *transformers*. Once again, the API is quite 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 an `imputer`. All transformers also have a convenience method called `fit_transform()` that is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).
  - *Predictors*. Finally, some estimators are capable of making predictions given a dataset; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: it predicted life satisfaction given a country's GDP per capita. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions given a test set (and the corresponding labels in the case of supervised learning algorithms).[16]

Create new features: population_per_household
population / household

[62]:
```
from sklearn.base import BaseEstimator, TransformerMixin

# column index
rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

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

```
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"]
```

One last thing you may want to do before actually preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are.

rooms_per_household =
total number of rooms /households

[63]:
```
housing_extra_attribs = pd.DataFrame(housing_extra_attribs, columns=list(housing.columns)+["rooms_per_household", "pop
housing_extra_attribs.head()
```

[63]:

ude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity	rooms_per_household	population_per_household
.29	38	1568	351	710	339	2.7042	<1H OCEAN	4.62537	2.0944
.05	14	679	108	306	113	6.4214	<1H OCEAN	6.00885	2.70796
.77	31	1952	471	936	462	2.8621	NEAR OCEAN	4.22511	2.02597
.31	25	1847	371	1460	353	1.8839	INLAND	5.23229	4.13598
.23	17	6592	1525	4459	1463	3.0347	<1H OCEAN	4.50581	3.04785

Transform pipelines

```
1  from sklearn.pipeline import Pipeline
2  from sklearn.preprocessing import StandardScaler
3
4  num_pipeline = Pipeline([
5          ('imputer', Imputer(strategy="median")),
6          ('attribs_adder', CombinedAttributesAdder()),
7          ('std_scaler', StandardScaler()),
8      ])
9
10 housing_num_tr = num_pipeline.fit_transform(housing_num)
```

```
housing_num_tr
```

```
array([[-1.15604281,  0.77194962,  0.74333089, ..., -0.31205452,
        -0.08649871,  0.15531753],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.21768338,
        -0.03353391, -0.83628902],
       [ 1.18684903, -1.34218285,  0.18664186, ..., -0.46531516,
        -0.09240499,  0.4222004 ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.3469342 ,
        -0.03055414, -0.52177644],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.02499488,
         0.06150916, -0.30340741],
       [-1.43579109,  0.99645926,  1.85670895, ..., -0.22852947,
        -0.09586294,  0.10180567]])
```

Custom transformer

- **feed a Pandas DataFrame directly into our pipeline instead of having to first manually extract the numerical columns into a NumPy array.**

```
1  from sklearn.base import BaseEstimator, TransformerMixin
2
3  # Create a class to select numerical or categorical columns
4  # since Scikit-Learn doesn't handle DataFrames yet
5  class DataFrameSelector(BaseEstimator, TransformerMixin):
6      def __init__(self, attribute_names):
7          self.attribute_names = attribute_names
8      def fit(self, X, y=None):
9          return self
10     def transform(self, X):
11         return X[self.attribute_names].values
```

Split data attributes and transform

```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
```

Label_binarizer: OHE ocean_proximity

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

Transformation Pipeline as one

Seq of transformations

In [67]:
```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
```

In [68]:
```
from sklearn.pipeline import FeatureUnion

full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
```

In [69]:
```
housing_prepared = full_pipeline.fit_transform(housing)
housing_prepared
```

Out[69]:
```
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
         0.        ,  0.        ],
```

Train dataset (16,512 rows)

```
In [69]:  1 housing_prepared = full_pipeline.fit_transform(housing)
          2 housing_prepared
```

```
Out[69]: array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
                  0.        ,  0.        ],
                [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
                  0.        ,  0.        ],
                [ 1.18684903, -1.34218285,  0.18664186, ...,  0.        ,
                  0.        ,  1.        ],
                ...,
                [ 1.58648943, -0.72478134, -1.56295222, ...,  0.        ,
                  0.        ,  0.        ],
                [ 0.78221312, -0.85106801,  0.18664186, ...,  0.        ,
                  0.        ,  0.        ],
                [-1.43579109,  0.99645926,  1.85670895, ...,  0.        ,
                  1.        ,  0.        ]])
```

```
In [70]: housing_prepared.shape
```

```
Out[70]: (16512, 16)
```

Outline

- **Introduction**
- **End to end ML project in SKLearn**
  - Problem definition and data
  - EDA
  - SKLearn Pipelines for prepping data
  - Full learning pipeline
  - Finetune model:
    - GridSearch versus Random
- **Which model is better? Significance Tests**
- **Summary**

Train LR Model; eval on 5 training exs.

```
from sklearn.linear_model import LinearRegression

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

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
# let's try the full pipeline on a few training instances
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)

print("Predictions:", lin_reg.predict(some_data_prepared))
```

```
Predictions: [ 210644.60459286  317768.80697211  210956.43331178   59218.98886849
  189747.55849879]
```

Compare against the actual values:

```
print("Labels:", list(some_labels))
```

```
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Evaluate on training set

```
from sklearn.metrics import mean_squared_error

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

```
68628.198198489219
```

```
from sklearn.metrics import mean_absolute_error

lin_mae = mean_absolute_error(housing_labels, housing_predictions)
lin_mae
```

```
49439.895990018973
```

DT is perfect on train set: WHAT?!

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)
```

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_split=1e-07,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, presort=False, random_state=42,
           splitter='best')
```

```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

```
0.0
```

Decision Tree doesn't look as good

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
```

```
1 from sklearn.model_selection import cross_val_score
2
3 scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
4                          scoring="neg_mean_squared_error", cv=10)
5 tree_rmse_scores = np.sqrt(-scores)
```

```
1 def display_scores(scores):
2     print("Scores:", scores)
3     print("Mean:", scores.mean())
4     print("Standard deviation:", scores.std())
5
6 display_scores(tree_rmse_scores)
```

```
Scores: [ 70232.0136482   66828.46839892  72444.08721003  70761.50186201
  71125.52697653  75581.29319857  70169.59286164  70055.37863456
  75370.49116773  71222.39081244]
Mean: 71379.0744771
Standard deviation: 2458.31882043
```