house-price-prediction-california

October 26, 2024

1 House Price Prediction California

1.1 Fetching Data

[10]:
```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request
```

[12]:
```
housing = pd.read_csv("datasets/housing/housing.csv")
```

Looking at the top five row in the data frame

[13]:
```
housing.head()
```

[13]:
```
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -122.23     37.88                41.0        880.0           129.0
1    -122.22     37.86                21.0       7099.0          1106.0
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0

   population  households  median_income  median_house_value ocean_proximity
0       322.0       126.0         8.3252            452600.0        NEAR BAY
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
```

Using info() able to find the Data type of the attribute, and Non-null count

[14]:
```
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
```

We can see there are some missing rows in total_bedrooms column missing rows in total_bedroom 207

Since ocean_proximity is categorycal value we can find how many distincts belong to each category

[15]:
```
housing ['ocean_proximity'].value_counts()
```

[15]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

describe() method shows a summary of the numerical columns in the dataframe

[16]:
```
housing.describe()
```

[16]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.800000     33.930000           18.000000   1447.750000
50%     -118.490000     34.260000           29.000000   2127.000000
75%     -118.010000     37.710000           37.000000   3148.000000
max     -114.310000     41.950000           52.000000  39320.000000

       total_bedrooms    population    households  median_income  \
count    20433.000000  20640.000000  20640.000000   20640.000000
mean       537.870553   1425.476744    499.539680       3.870671
std        421.385070   1132.462122    382.329753       1.899822
min          1.000000      3.000000      1.000000       0.499900
25%        296.000000    787.000000    280.000000       2.563400
50%        435.000000   1166.000000    409.000000       3.534800
75%        647.000000   1725.000000    605.000000       4.743250
max       6445.000000  35682.000000   6082.000000      15.000100
```

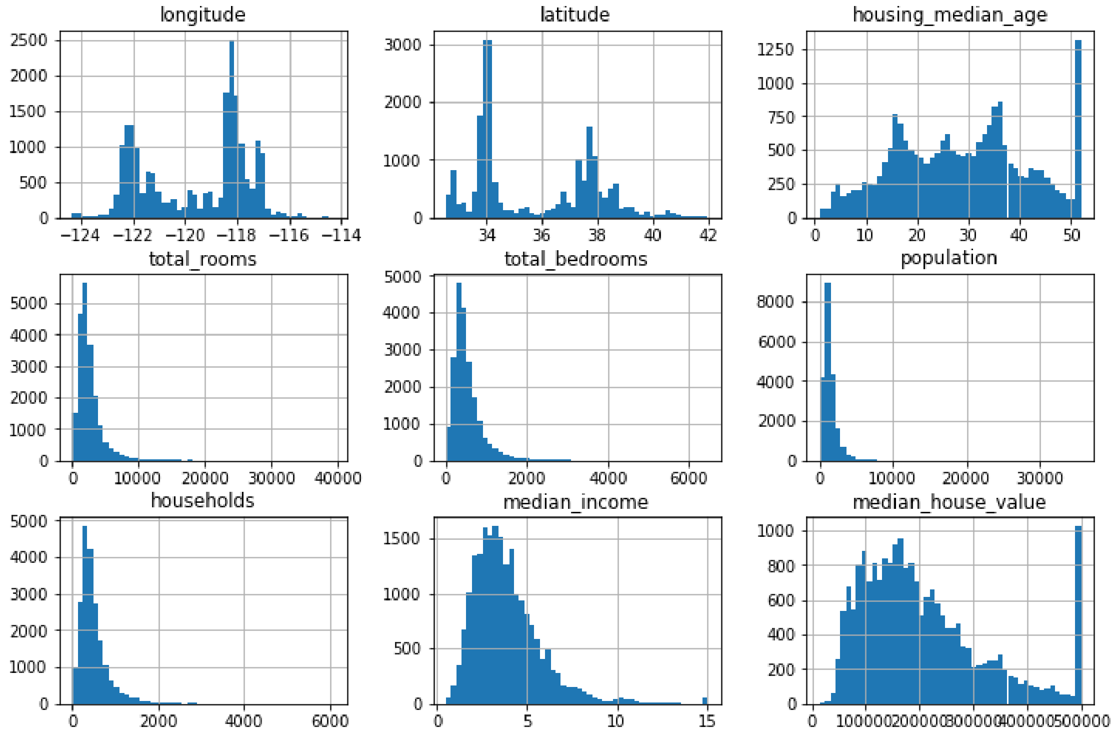
```
       median_house_value
count        20640.000000
mean        206855.816909
std         115395.615874
min          14999.000000
25%         119600.000000
50%         179700.000000
75%         264725.000000
max         500001.000000
```

Another way to feel the type of data we aare dealing with is to plot a histogram of each numberical attribute

Saving my plot Image Below code is used to save the image of the visual in high resolution, i will use this images in my portfolio side and Git hub page

[17]:
```
IMAGES_PATH = Path() / "images" / "end_to_end_project"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

[18]:
```
import matplotlib.pyplot as plt

plt.rc('font', size = 14)
plt.rc('axes', labelsize = 14, titlesize = 14)
plt.rc('legend', fontsize = 14)
plt.rc('xtick',labelsize = 10)
plt.rc('ytick', labelsize = 10)
plt.rc()
housing.hist(bins = 50,figsize = (12,8))
plt.show()
```



1) From this visual we can clearly see Income was Not in standard USD. It was capped, the highest median income is 15. which meas 1 defines $10,000.
2) Housing median also capped.
3) Many Histograms are Skewed right, This may make it a bit harder for some ML algorithm to detect patterns.

Temporoary Test data creation before moving in furture analysis 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.

And there are several way to create train and test data, below is one of the way

[19]:
```
import numpy as np

def shuffle_and_split_data(data, test_ratio):
    """
    This function gives split the given data into two dataframe(Train and test).
    Ratio of those data is defined by the used input.
    And it also give the output in shuffled manner using np.random
    """

    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
```

```
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

[20]:
```
train_set, test_set = shuffle_and_split_data(housing, 0.2)
print(len(train_set))
print(len(test_set))
```

```
16512
4128
```

The shuffle_and_split_data() function work's well, but that not perfect. Because when ever we rerun the cell it will generate different form of dataframe everytime.

To avoid that we can use seed() function. this helps us to gives the same shuffled dataframe everytime.

[21]:
```
np.random.seed(42)
```

Below is another possible implemantation

[22]:
```
from zlib import crc32

def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier))< test_ratio * 2**32

def split_data_with_id_hash(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: is_id_in_test_set(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

[23]:
```
# We are creating row_index column as a ID

housing_with_id = housing.reset_index()
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2,"index")
print(len(train_set))
print(len(test_set))
```

```
16512
4128
```

1.2 Scikit learn “train_test_split”

Sklearn provides a function to split datasets into multipel subsets in various ways. The simplest function is train_test_split(), which does pretty similar to shuffle_and_split_data() function, with a couple of additional features.

[24]:
```
from sklearn.model_selection import train_test_split
```

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

```
C:\Users\hi\anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarning: A
NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy
(detected version 1.26.4
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

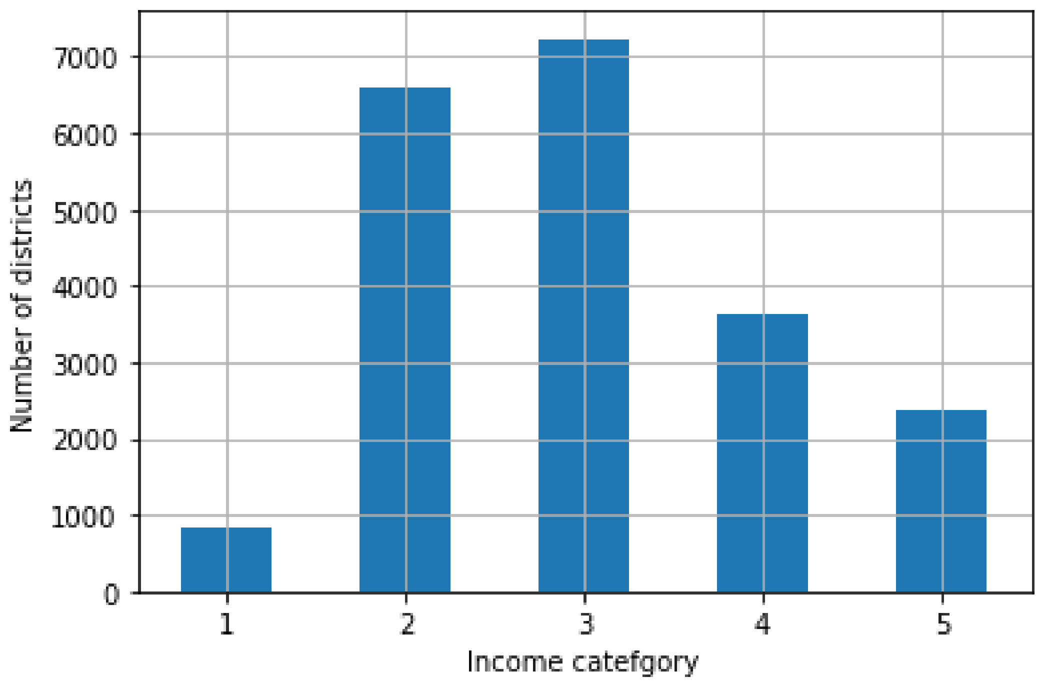
[25]:
```
test_set['total_bedrooms'].isnull().sum()
```

[25]: 44

1.2.1 fuction to create a income attribute with five categories(from 1 to 5)

[26]:
```
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])
```

[27]:
```
housing['income_cat'].value_counts().sort_index().plot.bar(rot = 0, grid = True)
plt.xlabel('Income catefgory')
plt.ylabel('Number of districts')
plt.show()
```



StratifiedShuffleSplit split() method gives the training and test indices, not the data itself. Having multiple splits can be usefull if we want to better estimate the performance of you model. the Stratified splits gives n different splits of the same dataset

[28]:
```
from sklearn.model_selection import StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits = 10 , test_size = 0.2, random_state␣
 ↪= 42)
strat_splits = []
for train_index, test_index in splitter.split(housing, housing['income_cat']):
    strat_train_set_n = housing.iloc[train_index]
    strat_test_set_n = housing.iloc[test_index]
    strat_splits.append([strat_test_set_n, strat_train_set_n])
```

[29]:
```
# for getting the first split from Stratified split

strat_train_set_n, strat_test_set_n = strat_splits[0]
print(len(strat_train_set_n))
print(len(strat_test_set_n))
```

```
4128
16512
```

another shorter way to get a single split using train_test_split()

[30]:
```
strat_train_set, strat_test_set = train_test_split(housing,
                                                   test_size = 0.2,
                                                   ␣
 ↪stratify=housing['income_cat'],
                                                   random_state=42)
```

let see the income category propotion in the dataset

[31]:
```
strat_test_set['income_cat'].value_counts()/len(strat_test_set)
```

[31]:
```
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: income_cat, dtype: float64
```

1.2.2 Comparing the data with each other

[32]:
```
def income_cat_proportions(data):
    return data['income_cat'].value_counts()/len(data)
```

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

compare_props = pd.DataFrame({
    'Overall %': income_cat_proportions(housing),
    'Stratified %': income_cat_proportions(strat_test_set),
    'Random %': income_cat_proportions(test_set)
}).sort_index()
compare_props.index_name = 'Income Category'
compare_props['Strat. Error %'] = (compare_props['Stratified %'] /
                                   compare_props['Overall %']-1)
compare_props['Rand. Error %'] = (compare_props['Random %']/
                                  compare_props['Overall %']-1)
(compare_props*100).round(2)
```

[32]:

	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

Removing “Income_cat” columns

[33]:
```
for set_ in (strat_train_set, strat_test_set):
    set_.drop('income_cat', axis = 1, inplace = True)
```

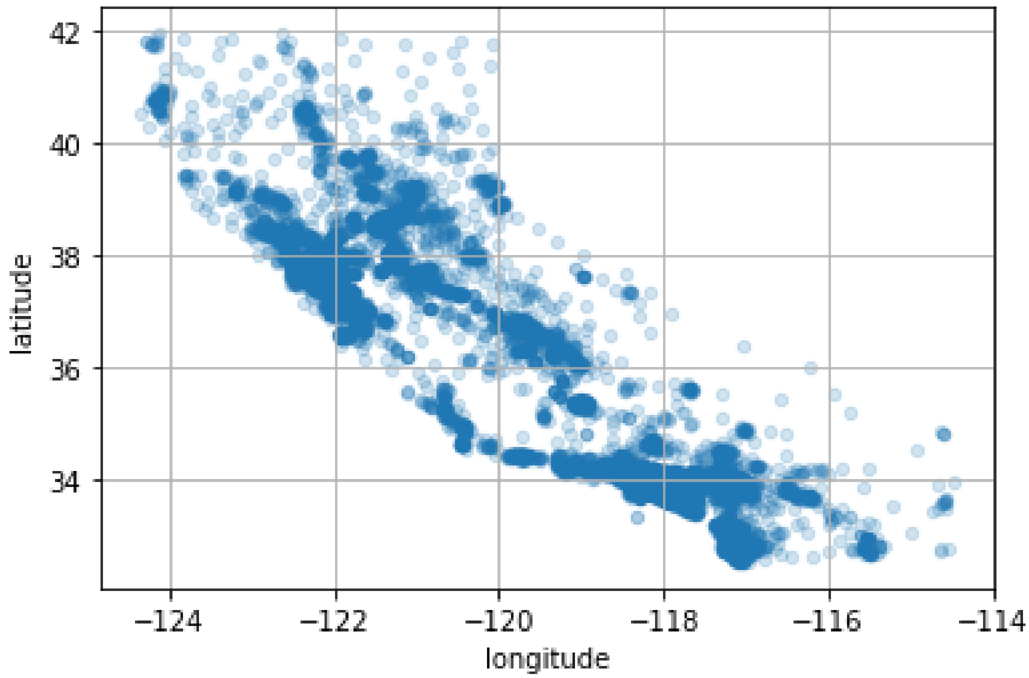
2 Exploratory Data Analysis(EDA)

For EDA we are using train data set

[34]:
```
housing = strat_train_set.copy()
```

2.1 Visualizing Geographical data

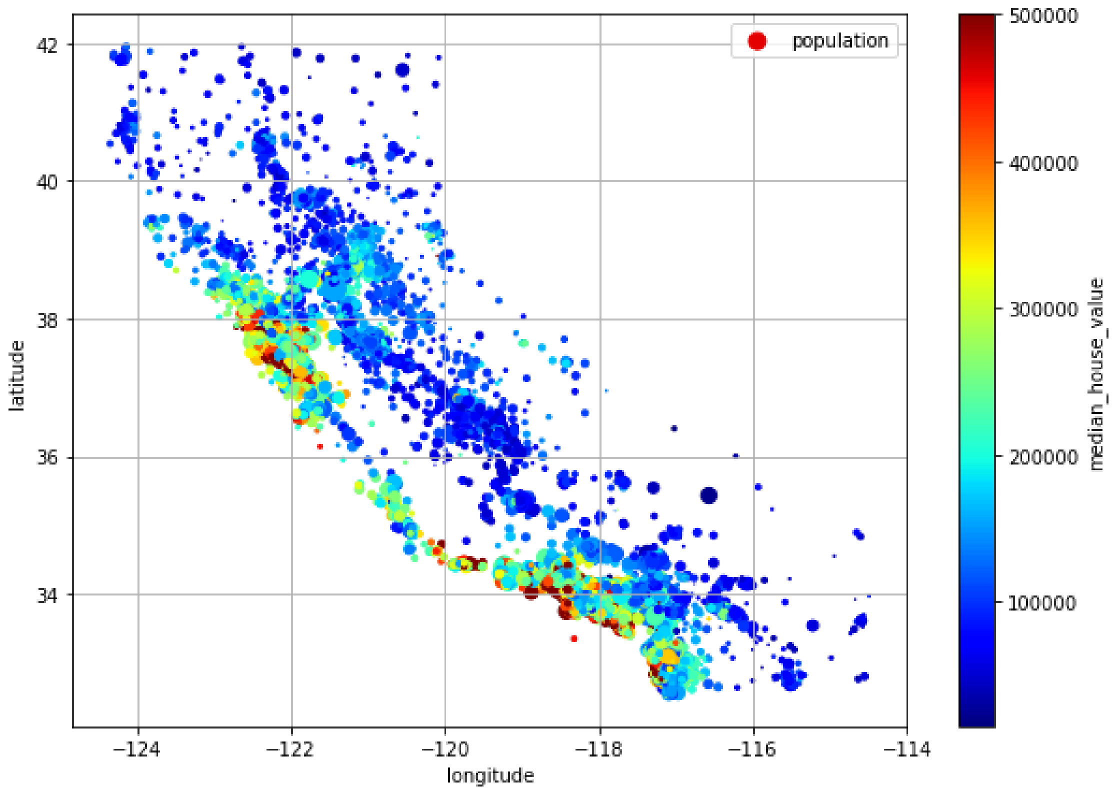
[35]:
```
housing.plot(kind ='scatter', x = 'longitude',y='latitude',grid = True, alpha =
  ↪0.2)
plt.show()
```



By the above visual we can clearly see the high-area namely the Bay Area and around Los Angeles and San Diego and Long line of fairly high-density areas in the central Valley.

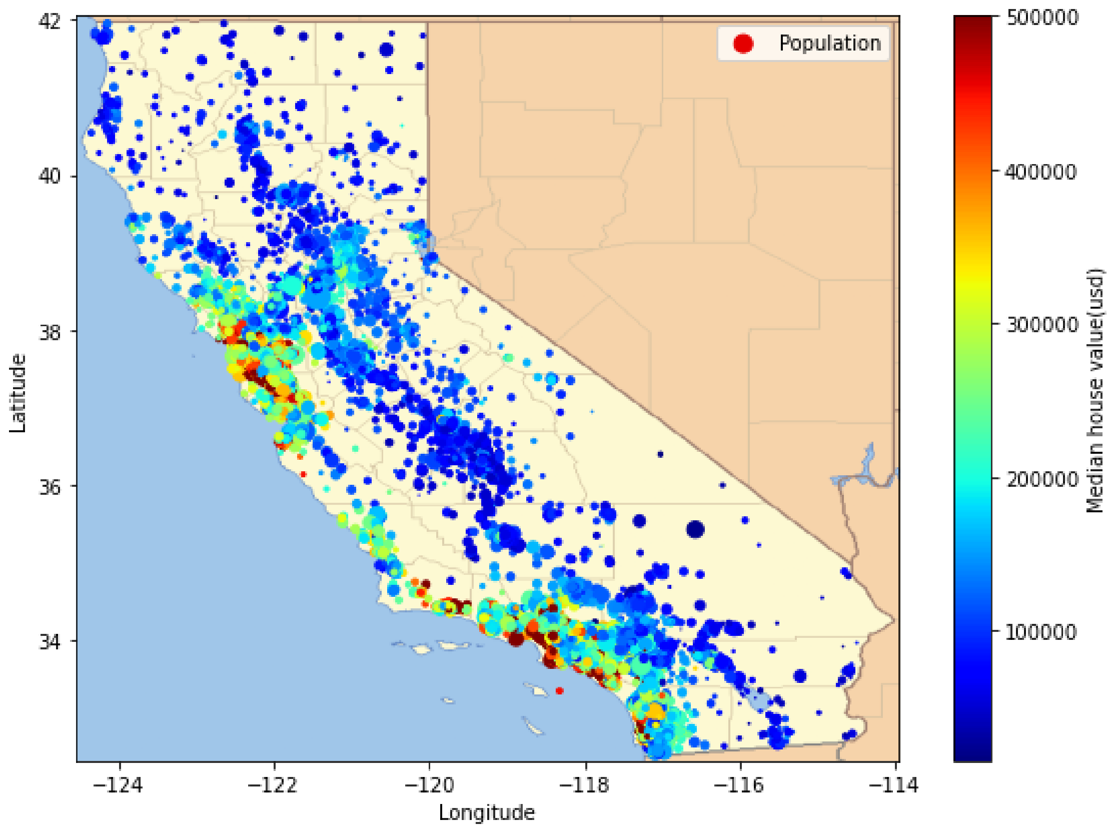
2.1.1 Housing price as per location and population %

[36]:
```
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()
```



Above image tells us Housing prices are very much related to the location, and population. Houses are close to the Ocean having higher cost

```
[38]: filename = "california.png"
housing_renamed = housing.rename(columns = {'latitude':'Latitude',
                                            'longitude':'Longitude',
                                            'population':'Population',
                                            'median_house_value':'Median house␣
 ↪value(usd)'})
housing_renamed.plot(kind='scatter', x = 'Longitude',y = 'Latitude',
                     s = housing_renamed['Population']/100, label =␣
 ↪'Population',
                     c = 'Median house value(usd)', cmap = 'jet', colorbar =␣
 ↪True,
                     legend = True, sharex = False, figsize = (10,7))
california_img = plt.imread(IMAGES_PATH/filename)
axis = -124.55, -113.95, 32.45, 42.05
plt.axis(axis)
plt.imshow(california_img, extent = axis)
plt.show()
```



3 Correlations

[39]:
```
corr_matrix = housing.corr()
```

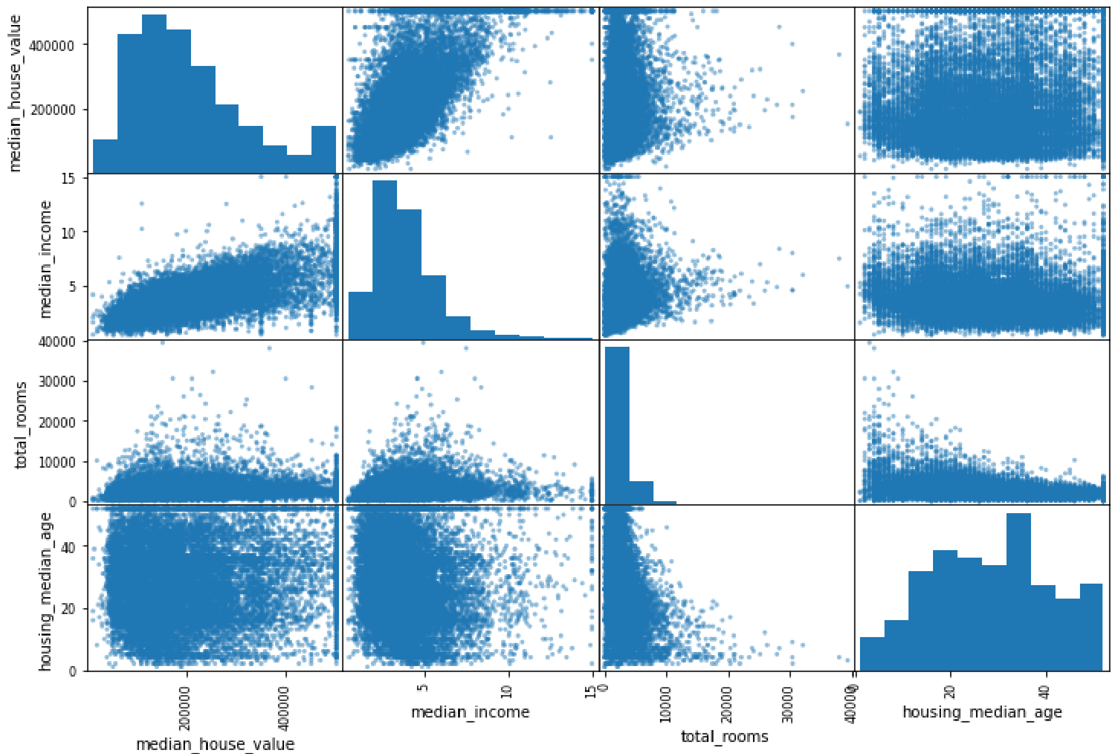
Looking for how much each arrtribute correlates with Median house value

[40]:
```
corr_matrix['median_house_value'].sort_values(ascending = False)
```

[40]:
```
median_house_value    1.000000
median_income         0.688380
total_rooms           0.137455
housing_median_age    0.102175
households            0.071426
total_bedrooms        0.054635
population           -0.020153
longitude            -0.050859
latitude             -0.139584
Name: median_house_value, dtype: float64
```

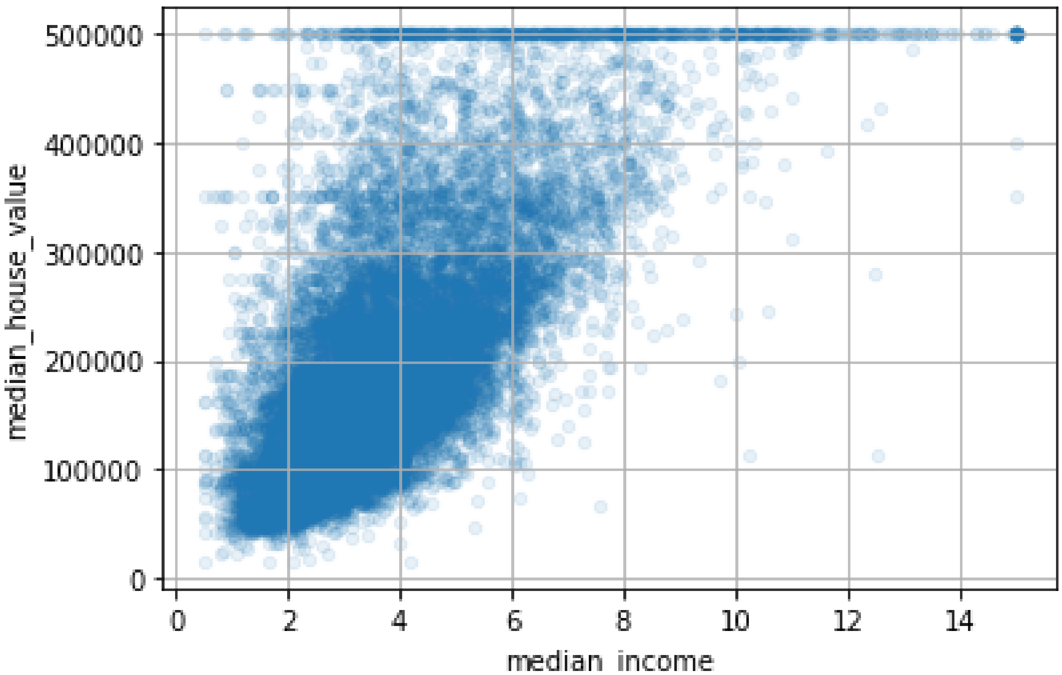
Since we are having 11 numerical column, for visual manner we will get lot of plot(121). To avoid that we are focusing on few attributes that seem most correlated

[41]:
```
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()
```



Correlation between Median_income and Median_house_value

[42]:
```
housing.plot(kind = 'scatter', x = 'median_income', y = 'median_house_value',
             alpha = 0.1, grid = True)
plt.show()
```



Above plot reveals a few thing 1) First correlatioon is quite strong 2) You can clearly see upward trend 3) capped price are clearly visible as a horizontal line at 500,000. 4) And some other Straight lines in 450,000, 350,000, 280,000 few more below We can remove those to prevent occur algorithm performace

3.0.1 Feature Engineering and Experimenting their combination

creating a new meaning full column and check how they are correlated

[43]:
```
housing['rooms_per_house'] = housing['total_rooms'] / housing['households']
housing['bedroom_ratio'] = housing['total_bedrooms'] / housing['total_rooms']
housing['people_per_house'] = housing['population'] / housing['households']
```

[44]:
```
corr_matrix = housing.corr()
corr_matrix['median_house_value'].sort_values(ascending = False)
```

[44]:
```
median_house_value    1.000000
median_income         0.688380
rooms_per_house       0.143663
total_rooms           0.137455
housing_median_age    0.102175
households            0.071426
total_bedrooms        0.054635
population           -0.020153
people_per_house     -0.038224
```

```
longitude            -0.050859
latitude             -0.139584
bedroom_ratio        -0.256397
Name: median_house_value, dtype: float64
```

3.1 Preparing Data for Machine Learning

[45]:
```python
# Reasigning the data set

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

3.1.1 Data Cleaning

In the beginng proces itself we found total_bedrooms having null values. So we decided to replace the missing value with median of the same column

[46]:
```python
median = housing['total_bedrooms'].median()
housing["total_bedrooms"].fillna(median, inplace = True)
```

But in future if we try to refresh the code with new data, there is possibility some other columns can get missing value. At the time we should create a new set of code to replace fillna. To avoid that we can use Imputer function

[47]:
```python
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy = 'median')
```

Since Median can be done with numerical value,we need to get only numerical column from the datafram

[48]:
```python
housing_num = housing.select_dtypes(include=[np.number])
imputer.fit(housing_num)
```

[48]:
```
SimpleImputer(strategy='median')
```

[49]:
```python
imputer.statistics_
```

[49]:
```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

[50]:
```python
housing_num.median().values
```

[50]:
```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

above two code block explains both imputer and median of housing_num are same

[51]:
```
X = imputer.transform(housing_num)
housing_tr = pd.DataFrame(X, columns = housing_num.columns,
                          index = housing_num.index)
housing_tr
```

[51]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
13096    -122.42     37.80                52.0       3321.0          1115.0
14973    -118.38     34.14                40.0       1965.0           354.0
3785     -121.98     38.36                33.0       1083.0           217.0
14689    -117.11     33.75                17.0       4174.0           851.0
20507    -118.15     33.77                36.0       4366.0          1211.0
…           …        …                  …            …              …
14207    -118.40     33.86                41.0       2237.0           597.0
13105    -119.31     36.32                23.0       2945.0           592.0
19301    -117.06     32.59                13.0       3920.0           775.0
19121    -118.40     34.06                37.0       3781.0           873.0
19888    -122.41     37.66                44.0        431.0           195.0

       population  households  median_income
13096      1576.0      1034.0         2.0987
14973       666.0       357.0         6.0876
3785        562.0       203.0         2.4330
14689      1845.0       780.0         2.2618
20507      1912.0      1172.0         3.5292
…             …          …             …
14207       938.0       523.0         4.7105
13105      1419.0       532.0         2.5733
19301      2814.0       760.0         4.0616
19121      1725.0       838.0         4.1455
19888       682.0       212.0         3.2833

[16512 rows x 8 columns]
```

3.1.2 Handling Text and Categorical columns

[52]:
```
housing_cat = housing[['ocean_proximity']]
housing_cat.head(10)
```

[52]:
```
      ocean_proximity
13096        NEAR BAY
14973       <1H OCEAN
3785           INLAND
14689          INLAND
20507      NEAR OCEAN
1286           INLAND
18078       <1H OCEAN
4396         NEAR BAY
```

```
18031       <1H OCEAN
6753        <1H OCEAN
```

Converting the categorical column to numerical column using OrdinalEncoder

[53]:
```python
from sklearn.preprocessing import OrdinalEncoder

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

[53]:
```
array([[3.],
       [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.],
       [0.],
       [0.]])
```

[54]:
```python
# for viewing what are the category in ordinal_encoder

ordinal_encoder.categories_
```

[54]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Converting the categorical column to numerical column using OneHotEncoder

[55]:
```python
from sklearn.preprocessing import OneHotEncoder

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

[56]:
```python
housing_cat_1hot.toarray()
```

[56]:
```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       …,
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.]])
```

3.2 Feature Scaling and Transformation

Implementing MinMaxScaler

[57]:
```python
from sklearn.preprocessing import MinMaxScaler

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

[57]:
```
array([[-0.60851927,  0.11702128,  1.        , …, -0.80701754,
        -0.61433638, -0.7794789 ],
       [ 0.21095335, -0.66170213,  0.52941176, …, -0.91866029,
        -0.86708979, -0.22929339],
       [-0.51926978,  0.23617021,  0.25490196, …, -0.93141946,
        -0.92458466, -0.73336919],
       …,
       [ 0.47870183, -0.99148936, -0.52941176, …, -0.65513434,
        -0.71663244, -0.50873781],
       [ 0.20689655, -0.6787234 ,  0.41176471, …, -0.78873758,
        -0.68751167, -0.49716556],
       [-0.60649087,  0.08723404,  0.68627451, …, -0.91669734,
        -0.92122457, -0.61608805]])
```

Implementing StandardScalar

[58]:
```python
from sklearn.preprocessing import StandardScaler

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

[58]:
```
array([[-1.42303652,  1.0136059 ,  1.86111875, …,  0.13746004,
         1.39481249, -0.93649149],
       [ 0.59639445, -0.702103  ,  0.90762971, …, -0.69377062,
        -0.37348471,  1.17194198],
       [-1.2030985 ,  1.27611874,  0.35142777, …, -0.78876841,
        -0.77572662, -0.75978881],
       …,
       [ 1.25620853, -1.42870103, -1.23772062, …,  1.26829911,
         0.67913534,  0.1010487 ],
       [ 0.58639727, -0.73960483,  0.66925745, …,  0.27356264,
         0.88286825,  0.14539615],
       [-1.41803793,  0.94797769,  1.22545939, …, -0.67915557,
        -0.75221898, -0.31034135]])
```

inverse_transform() For example, if the target distribution have a heavy tail, we may choose to replace the target with logarithm. But if we do, the regression model will now predict log of the median house value, not the median house value. For that we have inverse_transform() method. Below code explains how to scale the labels using a StandardScalar, then train a simple Linear Regression on the resulting scaled labels and use it to make predictions on some new data, Which we transform back to the original scale using the trained scaler's inverse_transform() method

[59]:
```
from sklearn.linear_model import LinearRegression

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

model = LinearRegression()
model.fit(housing[['median_income']], scaled_labels)
some_new_data = housing[['median_income']].iloc[:5] # temp data frame

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

[59]:
```
array([[131997.15275877],
       [299359.35844434],
       [146023.37185694],
       [138840.33653057],
       [192016.61557639]])
```

[60]:
```
scaled_predictions
```

[60]:
```
array([[-0.64466228],
       [ 0.80674175],
       [-0.52302364],
       [-0.5853166 ],
       [-0.12415952]])
```

Above code are works good, but there is another simpler version TransformedTargetRegressor

[61]:
```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer = StandardScaler())
model.fit(housing[['median_income']],housing_labels)
predictions = model.predict(some_new_data)
predictions
```

[61]:
```
array([131997.15275877, 299359.35844434, 146023.37185694, 138840.33653057,
       192016.61557639])
```

3.3 Custom Transformers

Below Custom Transformers function show how we can create our own function to use in future

[62]:
```python
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

[63]:
```python
from sklearn.metrics.pairwise import rbf_kernel

rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
age_simil_35
```

[63]:
```
array([[2.81118530e-13],
       [8.20849986e-02],
       [6.70320046e-01],
       …,
       [9.55316054e-22],
       [6.70320046e-01],
       [3.03539138e-04]])
```

If we pass it and arry with two features, it will measure the 2D distance(Eclidean) to some similarity. Below code will measure how the geograplic similarity betwen each district and Sanfrancisco.

[64]:
```python
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args = dict(Y=[sf_coords], gamma = 0.1))
sf_simil = sf_transformer.transform(housing[['latitude','longitude']])
sf_simil
```

[64]:
```
array([[0.999927  ],
       [0.05258419],
       [0.94864161],
       …,
       [0.00388525],
       [0.05038518],
       [0.99868067]])
```

Custom Transformer for Fit and Transform

Below Custom Transformer that uses a KMmeans clusterer in the fit() method to identify th emain clusters in the training data, and then uses rbf_kernel() in the transform() megthod to measure how similar each sample is to each cluster center

[65]:
```python
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted
```

[66]:
```python
from sklearn.cluster import KMeans
```

```python
class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, n_init=10,
                              random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```

Calling the createde Tansformer

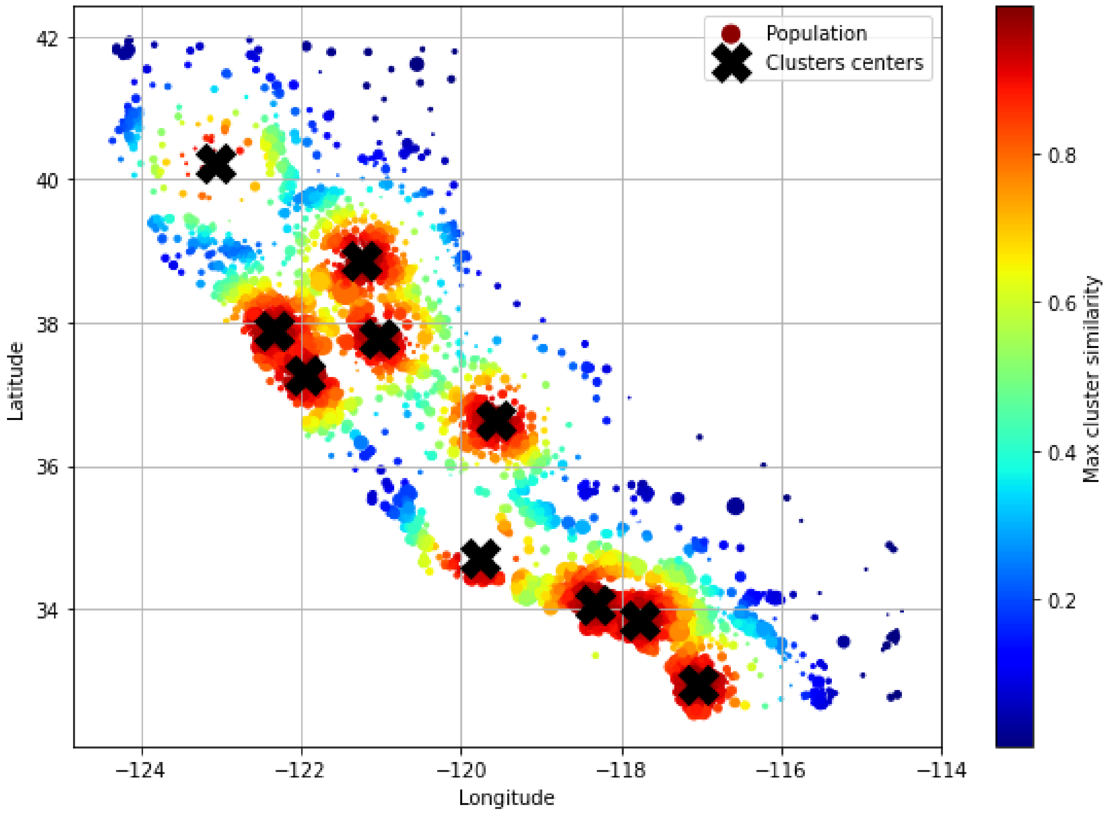
[67]:
```python
cluster_simil = ClusterSimilarity(n_clusters = 10, gamma = 1., random_state =␣
 ↪42)
similiraties = cluster_simil.fit_transform(housing[['latitude','longitude']],
                                           sample_weight = housing_labels)
similiraties[:3].round(2)
```

[67]:
```
array([[0.  , 0.14, 0.  , 0.  , 0.  , 0.08, 0.  , 0.99, 0.  , 0.6 ],
       [0.63, 0.  , 0.99, 0.  , 0.  , 0.  , 0.04, 0.  , 0.11, 0.  ],
       [0.  , 0.29, 0.  , 0.  , 0.01, 0.44, 0.  , 0.7 , 0.  , 0.3 ]])
```

Guassian RBF similarity to the nearest cluster center

[68]:
```python
housing_renamed = housing.rename(columns={
    "latitude": "Latitude", "longitude": "Longitude",
    "population": "Population",
    "median_house_value": "Median house value ( s )"})
housing_renamed['Max cluster similarity'] = similiraties.max(axis = 1)

housing_renamed.plot(kind = 'scatter', x = 'Longitude',y = 'Latitude', grid =␣
 ↪True,
                     s = housing_renamed['Population']/100, label =␣
 ↪'Population',
                     c = 'Max cluster similarity',cmap = 'jet', colorbar = True,
                     legend = True, sharex = False, figsize = (10,7))
plt.plot(cluster_simil.kmeans_.cluster_centers_[:,1],
         cluster_simil.kmeans_.cluster_centers_[:,0],
         linestyle= '', color = 'black', marker = 'X', markersize = 20,
```

```
        label = 'Clusters centers')
plt.legend(loc = 'upper right')
plt.show()
```



3.4 Tranformation Pipelines

There are many data tranformation steps that need to be executed in the right order. So Pipelines class to help with such sequences of tranformations. Below is the small pipeline for numerical attributes, which all first impute then scale the input features.

[69]:
```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ('impute', SimpleImputer(strategy = 'median')),
    ('standardize', StandardScaler())
])
```

If we don't want to name the tranformers, we can use make_pipeline() function instead.It takes transformera as positional arguments and create a Pipeline

[70]:
```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy ='median'),␣
↪StandardScaler())
```

Note: if multiple transformer have the same name, an index is appended to their names(eg 'temp-1', 'temp-2')

fit(): Applies fit_transform() to each transformer in sequence, using the output of one as input to the next. The final estimator's fit() is called with the last transformation's output.

Pipeline behavior mirrors its final step:

If the final step is a transformer (e.g., StandardScaler), the pipeline acts as a transformer and offers transform().

If the final step is a predictor (e.g., a classifier), the pipeline acts as a predictor and offers predict().

transform(): (For pipelines ending in transformers) Sequentially applies all transformations to the input data.

predict(): (For pipelines ending in predictors) Sequentially applies all transformations and feeds the result to the final predictor's predict() method.

[71]:
```
# lets call the fit_transform() and look at the ouput of first two rows

housing_num_prepared = num_pipeline.fit_transform(housing_num)
housing_num_prepared[:2].round(2)
```

[71]:
```
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

[72]:
```
def monkey_patch_get_signature_names_out():
    """Monkey patch some classes which did not handle get_feature_names_out()
       correctly in Scikit-Learn 1.0.*."""
    from inspect import Signature, signature, Parameter
    import pandas as pd
    from sklearn.impute import SimpleImputer
    from sklearn.pipeline import make_pipeline, Pipeline
    from sklearn.preprocessing import FunctionTransformer, StandardScaler

    default_get_feature_names_out = StandardScaler.get_feature_names_out

    if not hasattr(SimpleImputer, "get_feature_names_out"):
        print("Monkey-patching SimpleImputer.get_feature_names_out()")
        SimpleImputer.get_feature_names_out = default_get_feature_names_out

    if not hasattr(FunctionTransformer, "get_feature_names_out"):
        print("Monkey-patching FunctionTransformer.get_feature_names_out()")
        orig_init = FunctionTransformer.__init__
```

```
        orig_sig = signature(orig_init)

        def __init__(*args, feature_names_out=None, **kwargs):
            orig_sig.bind(*args, **kwargs)
            orig_init(*args, **kwargs)
            args[0].feature_names_out = feature_names_out

        __init__.__signature__ = Signature(
            list(signature(orig_init).parameters.values()) + [
                Parameter("feature_names_out", Parameter.KEYWORD_ONLY)])

        def get_feature_names_out(self, names=None):
            if callable(self.feature_names_out):
                return self.feature_names_out(self, names)
            assert self.feature_names_out == "one-to-one"
            return default_get_feature_names_out(self, names)

        FunctionTransformer.__init__ = __init__
        FunctionTransformer.get_feature_names_out = get_feature_names_out

monkey_patch_get_signature_names_out()
```

```
Monkey-patching SimpleImputer.get_feature_names_out()
Monkey-patching FunctionTransformer.get_feature_names_out()
```

If we want to DataFrame, we can use the get_feature_names_out()

[73]:
```
df_housing_num_prepared = pd.DataFrame(housing_num_prepared,
                                       columns=num_pipeline.
 ↪get_feature_names_out(),
                                      index = housing_num.index)
df_housing_num_prepared
```

[73]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
13096  -1.423037  1.013606            1.861119     0.311912        1.368167
14973   0.596394 -0.702103            0.907630    -0.308620       -0.435925
3785   -1.203098  1.276119            0.351428    -0.712240       -0.760709
14689   1.231216 -0.884924           -0.919891     0.702262        0.742306
20507   0.711362 -0.875549            0.589800     0.790125        1.595753
…           …         …                  …           …             …
14207   0.586397 -0.833359            0.987087    -0.184147        0.140152
13105   0.131525  0.319822           -0.443146     0.139847        0.128298
19301   1.256209 -1.428701           -1.237721     0.586026        0.562134
19121   0.586397 -0.739605            0.669257     0.522417        0.794461
19888  -1.418038  0.947978            1.225459    -1.010608       -0.812864

       population  households  median_income
13096    0.137460    1.394812      -0.936491
```

```
14973  -0.693771  -0.373485       1.171942
3785   -0.788768  -0.775727      -0.759789
14689   0.383175   0.731375      -0.850281
20507   0.444376   1.755263      -0.180365
…           …         …          …
14207  -0.445315   0.060101       0.444041
13105  -0.005950   0.083608      -0.685630
19301   1.268299   0.679135       0.101049
19121   0.273563   0.882868       0.145396
19888  -0.679156  -0.752219      -0.310341

[16512 rows x 8 columns]
```

A single Transformer is capable of handling all columns(numerical and categorical), For this we can use ColumnTransformer.This will apply num_pipeline to the numerical attributes and cat_pipeline to the categorical attribute

[74]:
```
from sklearn.compose import ColumnTransformer

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

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

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

[74]:
```
ColumnTransformer(transformers=[('num',
                                 Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
                                                 ('standardscaler',
                                                  StandardScaler())]),
                                 ['longitude', 'latitude', 'housing_median_age',
                                  'total_rooms', 'total_bedrooms', 'population',
                                  'households', 'median_income']),
                                ('cat',
                                 Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='most_frequent')),
                                                 ('onehotencoder',
OneHotEncoder(handle_unknown='ignore'))]),
                                 ['ocean_proximity'])])
```

Listing all the column names is not very convenient, Sklean provides make_columns_selector()

function that returns a selector function you can use to automatically select all the features of a given type, such as numerical or categorical. You can pass this selector function to the Column-Transformer instead of column names or indices Moreover, if we don't care about the naming the transformers, we can use make_columns_transformer() example from the above code except the trasformers are automatically named as "pipeline-1" and "pipeline-2" instead of "num" and "cat".

[75]:
```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
(num_pipeline, make_column_selector(dtype_include=np.number)),
(cat_pipeline, make_column_selector(dtype_include=object))
)
```

Now we can apply our housing data set to our transformer

[76]:
```
housing_prepared = preprocessing.fit_transform(housing)
```

[77]:
```
housing_prepared_fr = pd.DataFrame(
    housing_prepared,
    columns=preprocessing.get_feature_names_out(),
    index=housing.index)
housing_prepared_fr.head(2)
```

[77]:
```
       pipeline-1__longitude  pipeline-1__latitude  \
13096              -1.423037              1.013606
14973               0.596394             -0.702103

       pipeline-1__housing_median_age  pipeline-1__total_rooms  \
13096                        1.861119                 0.311912
14973                        0.907630                -0.308620

       pipeline-1__total_bedrooms  pipeline-1__population  \
13096                    1.368167                0.137460
14973                   -0.435925               -0.693771

       pipeline-1__households  pipeline-1__median_income  \
13096                1.394812                  -0.936491
14973               -0.373485                   1.171942

       pipeline-2__ocean_proximity_<1H OCEAN  \
13096                                    0.0
14973                                    1.0

       pipeline-2__ocean_proximity_INLAND  pipeline-2__ocean_proximity_ISLAND  \
13096                                 0.0                                 0.0
14973                                 0.0                                 0.0

       pipeline-2__ocean_proximity_NEAR BAY  \
```

```
13096                                   1.0
14973                                   0.0

       pipeline-2__ocean_proximity_NEAR OCEAN
13096                                     0.0
14973                                     0.0
```

Comining all the process

[78]:
```
def column_ratio(X):
    return X[:,[0]]/X[:,[1]]

def ratio_name(function_transformer, feature_names_in):
    return['ratio'] # Gives feature names

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy = 'median'),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler()
    )

log_pipeline = make_pipeline(
    SimpleImputer(strategy = 'median'),
    FunctionTransformer(np.log, feature_names_out='one-to-one'),
    StandardScaler()
)

cluster_simil = ClusterSimilarity(n_clusters = 10, gamma = 1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy='median'),
                                     StandardScaler())

preprocessing = ColumnTransformer([
    ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
    ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
    ("people_per_house", ratio_pipeline(), ["population", "households"]),
    ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
    "households", "median_income"]),
    ("geo", cluster_simil, ["latitude", "longitude"]),
    ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
],remainder = default_num_pipeline )# for remaining column - housing_median_age
```

[79]:
```
housing_prepared = preprocessing.fit_transform(housing)
housing_prepared.shape
```

[79]: (16512, 24)

[80]:
```
preprocessing.get_feature_names_out()
```

[80]:
```
array(['bedrooms__ratio', 'rooms_per_house__ratio',
       'people_per_house__ratio', 'log__total_bedrooms',
       'log__total_rooms', 'log__population', 'log__households',
       'log__median_income', 'geo__Cluster 0 similarity',
       'geo__Cluster 1 similarity', 'geo__Cluster 2 similarity',
       'geo__Cluster 3 similarity', 'geo__Cluster 4 similarity',
       'geo__Cluster 5 similarity', 'geo__Cluster 6 similarity',
       'geo__Cluster 7 similarity', 'geo__Cluster 8 similarity',
       'geo__Cluster 9 similarity', 'cat__ocean_proximity_<1H OCEAN',
       'cat__ocean_proximity_INLAND', 'cat__ocean_proximity_ISLAND',
       'cat__ocean_proximity_NEAR BAY', 'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```

3.5 Select and Train a Model

Train and Evaluagte on the Training set

[81]:
```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
```

[81]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(remainder=Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
                                                             ('standardscaler',
StandardScaler())]),
                                   transformers=[('bedrooms',
Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
('functiontransformer',
FunctionTransformer(feature_names_out=<function ratio_name at 0x000…
                                                  'median_income']),
                                                ('geo',
ClusterSimilarity(random_state=42),
                                                 ['latitude', 'longitude']),
                                                ('cat',
Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='most_frequent')),
('onehotencoder',
OneHotEncoder(handle_unknown='ignore'))]),
<sklearn.compose._column_transformer.make_column_selector object at
0x000001DF2E3DDC40>)])),
                ('linearregression', LinearRegression())])
```

Looking at the first five predictions and comparing them to the labels

[82]:
```python
housing_predictions = lin_reg.predict(housing)
housing_predictions[:5].round(-2) # -2 rounded to the nearest hundred
```

[82]:
```
array([243700., 372400., 128800.,  94400., 328300.])
```

[83]:
```python
housing_labels.iloc[:5].values
```

[83]:
```
array([458300., 483800., 101700.,  96100., 361800.])
```

by the above prediction we can we there is huge difference in the first prediction

for error calculation we can use RMSE

[84]:
```python
from sklearn.metrics import mean_squared_error
lin_rmse = mean_squared_error(housing_labels, housing_predictions,
                              squared = False)
```

[85]:
```python
lin_rmse
```

[85]:
```
68687.89176590036
```

Not a great score, and it shows the model is underfit. so we can approach powerful model

Decision Tree Regressor

[86]:
```python
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```

[86]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(remainder=Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
                                                              ('standardscaler',
StandardScaler())]),
                                   transformers=[('bedrooms',
Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
('functiontransformer',
FunctionTransformer(feature_names_out=<function ratio_name at 0x000…
ClusterSimilarity(random_state=42),
                                                  ['latitude', 'longitude']),
                                                 ('cat',
Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='most_frequent')),
('onehotencoder',
OneHotEncoder(handle_unknown='ignore'))]),
<sklearn.compose._column_transformer.make_column_selector object at
0x000001DF2E3DDC40>)])),
```

```
                    ('decisiontreeregressor',
                     DecisionTreeRegressor(random_state=42))])
```

[87]:
```
housing_predictions = tree_reg.predict(housing)
tree_rmse = mean_squared_error(housing_labels, housing_predictions,
                               squared = False)
tree_rmse
```

[87]:
```
0.0
```

This 0 error shows the of Overfit the data, for solving the we need to use part of the training set for training and part of it for model validation

3.6 Evaluation using Cross - Validation

The following code randomly splits the training set into 10 nonoverlapping subsets called folds

Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the RMSE. It's a negative value, so you need to switch the sign of the output to get the RMSE scores.

[88]:
```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring = 'neg_root_mean_squared_error', cv = 10)
```

[89]:
```
tree_rmses
```

[89]:
```
array([66506.70121103, 67097.20497743, 66144.65155743, 65191.02780781,
       64727.51994473, 70094.77824639, 67685.73110573, 68411.33556978,
       69293.5932016 , 63649.53649274])
```

[90]:
```
pd.Series(tree_rmses).describe()
```

[90]:
```
count       10.000000
mean     66880.208011
std       2049.481815
min      63649.536493
25%      65429.433745
50%      66801.953094
75%      68229.934454
max      70094.778246
dtype: float64
```

This Decision tree using cross validation also gives poor result

Random Forest Regressor

[91]:
```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,␣
↪RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                scoring = 'neg_root_mean_squared_error', cv = 10)
```

[92]:
```
forest_rmses
```

[92]:
```
array([46336.33927043, 47340.03146064, 45458.11252725, 46887.47214675,
       46032.78935423, 46955.59525055, 46979.59745817, 49227.03060992,
       47778.38173655, 47309.76160899])
```

[93]:
```
pd.Series(forest_rmses).describe()
```

[93]:
```
count       10.000000
mean     47030.511142
std       1029.358881
min      45458.112527
25%      46474.122490
50%      46967.596354
75%      47332.463998
max      49227.030610
dtype: float64
```

Above number much better than other model

3.7 Model Fine-Tune

Grid Search

[94]:
```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ('preprocessing', preprocessing),
    ('random_forest', RandomForestRegressor(random_state=42))
])

param_grid = [
    {'preprocessing__geo__n_clusters': [5, 8, 10],
    'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo__n_clusters': [10, 15],
    'random_forest__max_features': [6, 8, 10]},
]

grid_search = GridSearchCV(full_pipeline, param_grid, cv = 3,
                           scoring = 'neg_root_mean_squared_error')
```

```python
grid_search.fit(housing, housing_labels)
```

[94]:
```
GridSearchCV(cv=3,
             estimator=Pipeline(steps=[('preprocessing',
ColumnTransformer(remainder=Pipeline(steps=[('simpleimputer',
     SimpleImputer(strategy='median')),
    ('standardscaler',
     StandardScaler())]),
transformers=[('bedrooms',
Pipeline(steps=[('simpleimputer',
          SimpleImputer(strategy='median')),
         ('functiontransformer',
          FunctionTransformer(feature_names_out=<f…
<sklearn.compose._column_transformer.make_column_selector object at
0x000001DF2E3DDC40>)])),
                                       ('random_forest',
RandomForestRegressor(random_state=42))]),
             param_grid=[{'preprocessing__geo__n_clusters': [5, 8, 10],
                          'random_forest__max_features': [4, 6, 8]},
                         {'preprocessing__geo__n_clusters': [10, 15],
                          'random_forest__max_features': [6, 8, 10]}],
             scoring='neg_root_mean_squared_error')
```

To see the full list of hyperparameters available for tuning by looking at full_pipeline.get_params().keys()

[95]:
```python
print(str(full_pipeline.get_params().keys())[:1000]+"...")
```

```
dict_keys(['memory', 'steps', 'verbose', 'preprocessing', 'random_forest',
'preprocessing__n_jobs', 'preprocessing__remainder__memory',
'preprocessing__remainder__steps', 'preprocessing__remainder__verbose',
'preprocessing__remainder__simpleimputer',
'preprocessing__remainder__standardscaler',
'preprocessing__remainder__simpleimputer__add_indicator',
'preprocessing__remainder__simpleimputer__copy',
'preprocessing__remainder__simpleimputer__fill_value',
'preprocessing__remainder__simpleimputer__missing_values',
'preprocessing__remainder__simpleimputer__strategy',
'preprocessing__remainder__simpleimputer__verbose',
'preprocessing__remainder__standardscaler__copy',
'preprocessing__remainder__standardscaler__with_mean',
'preprocessing__remainder__standardscaler__with_std',
'preprocessing__remainder', 'preprocessing__sparse_threshold',
'preprocessing__transformer_weights', 'preprocessing__transformers',
'preprocessing__verbose', 'preprocessing__verbose_feature_names_out',
'preprocessing__be…
```

[96]:
```
grid_search.best_params_
```

[96]:
```
{'preprocessing__geo__n_clusters': 15, 'random_forest__max_features': 6}
```

[97]:
```
cv_res = pd.DataFrame(grid_search.cv_results_)
cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)

cv_res = cv_res[["param_preprocessing__geo__n_clusters",
                 "param_random_forest__max_features", "split0_test_score",
                 "split1_test_score", "split2_test_score", "mean_test_score"]]
score_cols = ["split0", "split1", "split2", "mean_test_rmse"]
cv_res.columns = ["n_clusters", "max_features"] + score_cols
cv_res[score_cols] = -cv_res[score_cols].round().astype(np.int64)

cv_res.head()
```

[97]:

	n_clusters	max_features	split0	split1	split2	mean_test_rmse
12	15	6	43427	43919	44754	44033
13	15	8	44131	44075	45037	44415
14	15	10	44323	44286	45305	44638
7	10	6	44679	44655	45617	44984
9	10	6	44679	44655	45617	44984

3.7.1 Randomized Search

[99]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {'preprocessing__geo__n_clusters': randint(low=3, high=50),
                  'random_forest__max_features': randint(low=2, high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distribs, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

[99]:
```
RandomizedSearchCV(cv=3,
                   estimator=Pipeline(steps=[('preprocessing',
ColumnTransformer(remainder=Pipeline(steps=[('simpleimputer',
           SimpleImputer(strategy='median')),
          ('standardscaler',
           StandardScaler())]),
transformers=[('bedrooms',
Pipeline(steps=[('simpleimputer',
                SimpleImputer(strategy='median')),
               ('functiontransformer',
```

```
                FunctionTransformer(feature_names_…
<sklearn.compose._column_transformer.make_column_selector object at
0x000001DF2E3DDC40>)])),
                                             ('random_forest',
RandomForestRegressor(random_state=42))]),
                   param_distributions={'preprocessing__geo__n_clusters':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x000001DF46377FA0>,
                                        'random_forest__max_features':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x000001DF46380190>},
                   random_state=42, scoring='neg_root_mean_squared_error')
```

Displaying the result for Random Search result

[102]:
```
cv_res = pd.DataFrame(rnd_search.cv_results_)
cv_res.sort_values(by = 'mean_test_score', ascending=False, inplace = True)
cv_res = cv_res[['param_preprocessing__geo__n_clusters',
                'param_random_forest__max_features', 'split0_test_score',
                'split1_test_score','split2_test_score','mean_test_score']]
cv_res.columns = ['n_clusters','max_features']+score_cols
cv_res[score_cols] = -cv_res[score_cols].round().astype(np.int64)
cv_res.head()
```

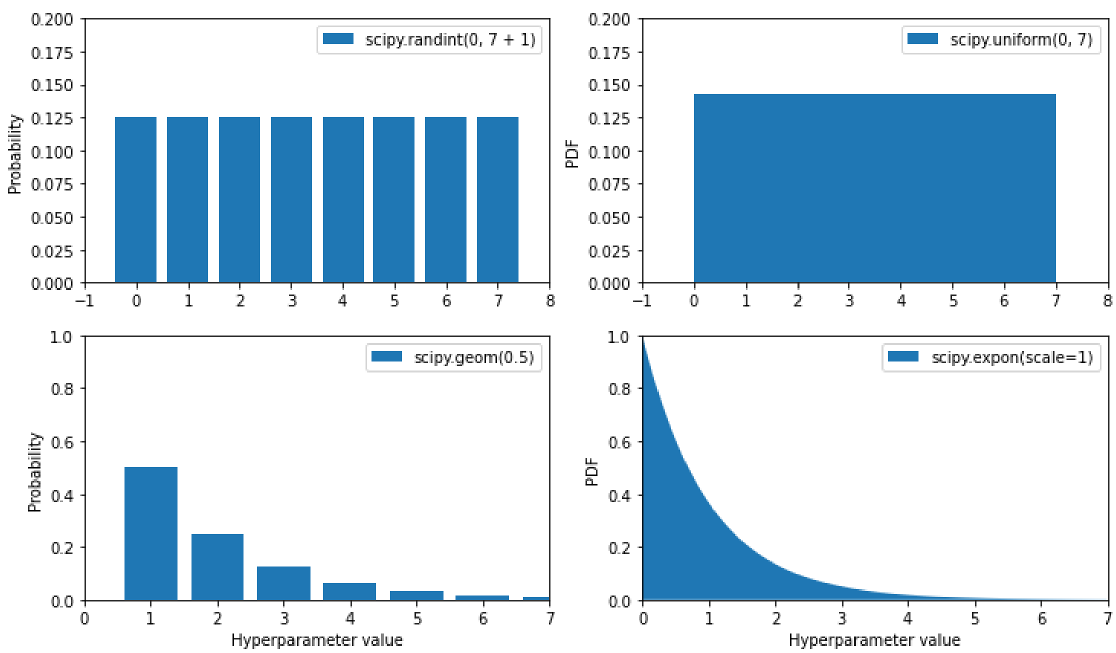
[102]:

	n_clusters	max_features	split0	split1	split2	mean_test_rmse
1	45	9	41280	42150	42668	42033
8	32	7	41678	42542	43132	42451
0	41	16	42264	42959	43305	42843
5	42	4	41752	43094	43788	42878
2	23	8	42208	42996	43807	43004

Plot a few distrubutions you can use in randomized search

[103]:
```
from scipy.stats import randint, uniform, geom, expon

xs1 = np.arange(0, 7 + 1)
randint_distrib = randint(0, 7 + 1).pmf(xs1)

xs2 = np.linspace(0, 7, 500)
uniform_distrib = uniform(0, 7).pdf(xs2)

xs3 = np.arange(0, 7 + 1)
geom_distrib = geom(0.5).pmf(xs3)

xs4 = np.linspace(0, 7, 500)
expon_distrib = expon(scale=1).pdf(xs4)

plt.figure(figsize=(12, 7))

plt.subplot(2, 2, 1)
```

```
plt.bar(xs1, randint_distrib, label="scipy.randint(0, 7 + 1)")
plt.ylabel("Probability")
plt.legend()
plt.axis([-1, 8, 0, 0.2])

plt.subplot(2, 2, 2)
plt.fill_between(xs2, uniform_distrib, label="scipy.uniform(0, 7)")
plt.ylabel("PDF")
plt.legend()
plt.axis([-1, 8, 0, 0.2])

plt.subplot(2, 2, 3)
plt.bar(xs3, geom_distrib, label="scipy.geom(0.5)")
plt.xlabel("Hyperparameter value")
plt.ylabel("Probability")
plt.legend()
plt.axis([0, 7, 0, 1])

plt.subplot(2, 2, 4)
plt.fill_between(xs4, expon_distrib, label="scipy.expon(scale=1)")
plt.xlabel("Hyperparameter value")
plt.ylabel("PDF")
plt.legend()
plt.axis([0, 7, 0, 1])

plt.show()
```

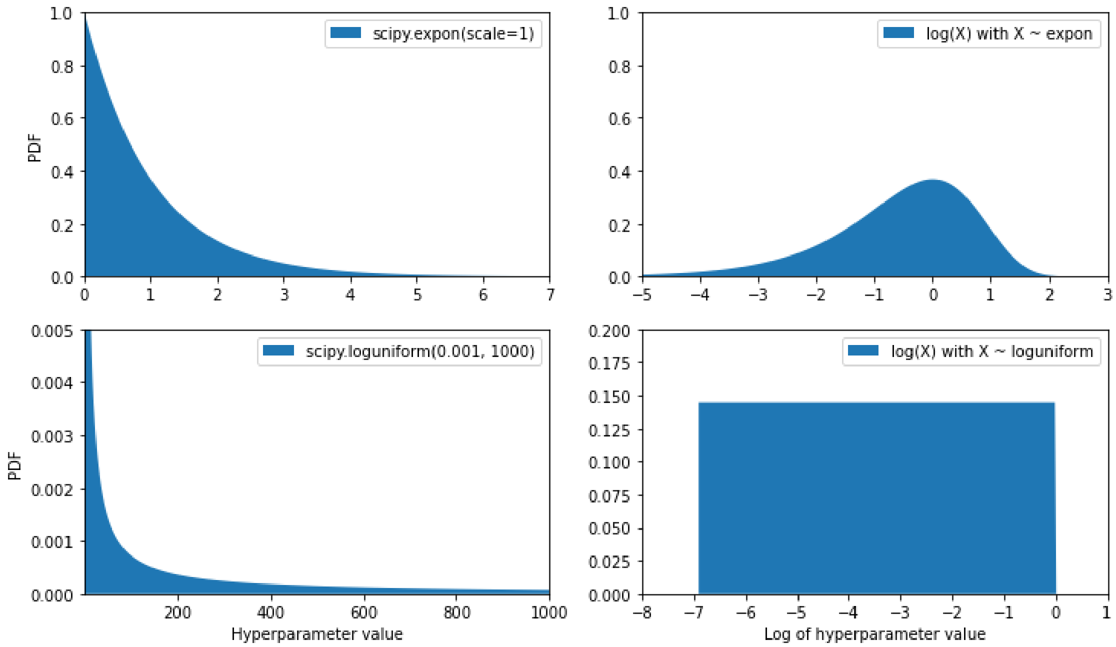


Here are the PDF for expon() and loguniform() (left column), as well as the PDF of log(X) (right column). The right column shows the distribution of hyperparameter scales. You can see that expon() favors hyperparameters with roughly the desired scale, with a longer tail towards the smaller scales. But loguniform() does not favor any scale, they are all equally likely

Showing the difference between expon and loguniform

[104]:
```
from scipy.stats import loguniform

xs1 = np.linspace(0, 7, 500)
expon_distrib = expon(scale=1).pdf(xs1)

log_xs2 = np.linspace(-5, 3, 500)
log_expon_distrib = np.exp(log_xs2 - np.exp(log_xs2))

xs3 = np.linspace(0.001, 1000, 500)
loguniform_distrib = loguniform(0.001, 1000).pdf(xs3)

log_xs4 = np.linspace(np.log(0.001), np.log(1000), 500)
log_loguniform_distrib = uniform(np.log(0.001), np.log(1000)).pdf(log_xs4)

plt.figure(figsize=(12, 7))

plt.subplot(2, 2, 1)
plt.fill_between(xs1, expon_distrib,
                 label="scipy.expon(scale=1)")
plt.ylabel("PDF")
plt.legend()
plt.axis([0, 7, 0, 1])

plt.subplot(2, 2, 2)
plt.fill_between(log_xs2, log_expon_distrib,
                 label="log(X) with X ~ expon")
plt.legend()
plt.axis([-5, 3, 0, 1])

plt.subplot(2, 2, 3)
plt.fill_between(xs3, loguniform_distrib,
                 label="scipy.loguniform(0.001, 1000)")
plt.xlabel("Hyperparameter value")
plt.ylabel("PDF")
plt.legend()
plt.axis([0.001, 1000, 0, 0.005])

plt.subplot(2, 2, 4)
plt.fill_between(log_xs4, log_loguniform_distrib,
                 label="log(X) with X ~ loguniform")
plt.xlabel("Log of hyperparameter value")
```

```
plt.legend()
plt.axis([-8, 1, 0, 0.2])

plt.show()
```



3.8 Analyzing best models and their errors

[105]:
```
final_model = rnd_search.best_estimator_
feature_importances = final_model['random_forest'].feature_importances_
feature_importances.round(2)
```

[105]:
```
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, 0.04, 0.01, 0.  ,
       0.01, 0.01, 0.01, 0.01, 0.01, 0.  , 0.01, 0.01, 0.01, 0.  , 0.01,
       0.01, 0.01, 0.01, 0.01, 0.  , 0.  , 0.02, 0.01, 0.01, 0.01, 0.02,
       0.01, 0.  , 0.02, 0.03, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01,
       0.01, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.  , 0.07,
       0.  , 0.  , 0.  , 0.01])
```

[107]:
```
sorted(zip(feature_importances,
           final_model['preprocessing'].get_feature_names_out()),
           reverse = True)
```

[107]:
```
[(0.18694559869103852, 'log__median_income'),
 (0.0748194905715524, 'cat__ocean_proximity_INLAND'),
 (0.06926417748515576, 'bedrooms__ratio'),
```

```
(0.05446998753775219, 'rooms_per_house__ratio'),
(0.05262301809680712, 'people_per_house__ratio'),
(0.03819415873915732, 'geo__Cluster 0 similarity'),
(0.02879263999929514, 'geo__Cluster 28 similarity'),
(0.023530192521380392, 'geo__Cluster 24 similarity'),
(0.020544786346378206, 'geo__Cluster 27 similarity'),
(0.019873052631077512, 'geo__Cluster 43 similarity'),
(0.018597511022930273, 'geo__Cluster 34 similarity'),
(0.017409085415656868, 'geo__Cluster 37 similarity'),
(0.015546519677632162, 'geo__Cluster 20 similarity'),
(0.014230331127504292, 'geo__Cluster 17 similarity'),
(0.0141032216204026, 'geo__Cluster 39 similarity'),
(0.014065768027447325, 'geo__Cluster 9 similarity'),
(0.01354220782825315, 'geo__Cluster 4 similarity'),
(0.013489636258229071, 'geo__Cluster 3 similarity'),
(0.013383196263838682, 'geo__Cluster 38 similarity'),
(0.012240533790212824, 'geo__Cluster 31 similarity'),
(0.012089046542256785, 'geo__Cluster 7 similarity'),
(0.01152326329703204, 'geo__Cluster 23 similarity'),
(0.011397459905603558, 'geo__Cluster 40 similarity'),
(0.011282340924816446, 'geo__Cluster 36 similarity'),
(0.01104139770781063, 'remainder__housing_median_age'),
(0.010671123191312804, 'geo__Cluster 44 similarity'),
(0.010296376177202627, 'geo__Cluster 5 similarity'),
(0.010184798445004483, 'geo__Cluster 42 similarity'),
(0.010121853542225083, 'geo__Cluster 11 similarity'),
(0.009795219101117579, 'geo__Cluster 35 similarity'),
(0.00952581084310724, 'geo__Cluster 10 similarity'),
(0.009433209165984825, 'geo__Cluster 13 similarity'),
(0.00915075361116215, 'geo__Cluster 1 similarity'),
(0.009021485619463173, 'geo__Cluster 30 similarity'),
(0.00894936224917583, 'geo__Cluster 41 similarity'),
(0.008901832702357514, 'geo__Cluster 25 similarity'),
(0.008897504713401587, 'geo__Cluster 29 similarity'),
(0.0086846298524955, 'geo__Cluster 21 similarity'),
(0.008061104590483955, 'geo__Cluster 15 similarity'),
(0.00786048176566994, 'geo__Cluster 16 similarity'),
(0.007793633130749198, 'geo__Cluster 22 similarity'),
(0.007501766442066527, 'log__total_rooms'),
(0.00720241119382413, 'geo__Cluster 32 similarity'),
(0.0069471565989956165, 'log__population'),
(0.006800076770899128, 'log__households'),
(0.006736105364684462, 'log__total_bedrooms'),
(0.006315268213499131, 'geo__Cluster 33 similarity'),
(0.005796398579893261, 'geo__Cluster 14 similarity'),
(0.005234954623294958, 'geo__Cluster 6 similarity'),
(0.0045514083468621595, 'geo__Cluster 12 similarity'),
```

```
 (0.004546042080216035, 'geo__Cluster 18 similarity'),
 (0.004314514641115754, 'geo__Cluster 2 similarity'),
 (0.003953528110719969, 'geo__Cluster 19 similarity'),
 (0.003297404747742136, 'geo__Cluster 26 similarity'),
 (0.0028945347429088692, 'cat__ocean_proximity_<1H OCEAN'),
 (0.0016978863168109132, 'cat__ocean_proximity_NEAR OCEAN'),
 (0.0016391131530559377, 'geo__Cluster 8 similarity'),
 (0.00015061247730531555, 'cat__ocean_proximity_NEAR BAY'),
 (7.301686597099842e-05, 'cat__ocean_proximity_ISLAND')]
```

3.8.1 Evaluating the Test Set

[110]:
```
X_test = strat_test_set.drop('median_house_value', axis = 1)
y_test = strat_test_set['median_house_value'].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions, squared = False)
print(final_rmse)
```

```
41424.40026462184
```

We can compute a 95% confidence interval for the test RMSE

[112]:
```
from scipy import stats
confidence = 0.95
squared_errors = (final_predictions - y_test)**2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc = squared_errors.mean(),
                         scale = stats.sem(squared_errors)))
```

[112]:
```
array([39275.40861216, 43467.27680583])
```

Showing how to compute a confidence interval for the RMSE

[113]:
```
m = len(squared_errors)
mean = squared_errors.mean()
tscore = stats.t.ppf((1+confidence)/2, df = m -1)
tmargin = tscore*squared_errors.std(ddof=1)/np.sqrt(m)
np.sqrt(mean - tmargin), np.sqrt(mean+tmargin)
```

[113]:
```
(39275.40861216077, 43467.27680583419)
```

Alternatively, we can use Z-score rather than t-score. Since the test is too small, it won't make a huge difference.

[115]:
```
zscore = stats.norm.ppf((1+confidence)/2)
zmargin = zscore * squared_errors.std(ddof = 1)/np.sqrt(m)
```

```
np.sqrt(mean - zmargin), np.sqrt(mean+zmargin)
```

[115]:
```
(39276.05610140007, 43466.69174996963)
```

3.9 Launch, Monitor and Maintain our System

[116]:
```
import joblib

joblib.dump(final_model, 'my_california_housing_model.pkl')
```

[116]:
```
['my_california_housing_model.pkl']
```

[122]:
```
final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = housing.iloc[:5]  # pretend these are new districts
predictions = final_model_reloaded.predict(new_data)
```

[124]:
```
new_data
```

[124]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
13096    -122.42     37.80                52.0       3321.0          1115.0
14973    -118.38     34.14                40.0       1965.0           354.0
3785     -121.98     38.36                33.0       1083.0           217.0
14689    -117.11     33.75                17.0       4174.0           851.0
20507    -118.15     33.77                36.0       4366.0          1211.0

       population  households  median_income ocean_proximity
13096      1576.0      1034.0         2.0987        NEAR BAY
14973       666.0       357.0         6.0876       <1H OCEAN
3785        562.0       203.0         2.4330          INLAND
14689      1845.0       780.0         2.2618          INLAND
20507      1912.0      1172.0         3.5292      NEAR OCEAN
```

[125]:
```
housing_labels.iloc[:5]
```

[125]:
```
13096    458300.0
14973    483800.0
3785     101700.0
14689     96100.0
20507    361800.0
Name: median_house_value, dtype: float64
```

[123]:
```
predictions
```

[123]:
```
array([442737.15, 457566.06, 105965.  ,  98462.  , 332992.01])
```