Housing

November 28, 2024

0.0.1 Importing data

[2]:
```python
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

0.0.2 Some exploratory data analysis

[3]:
```python
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)
```

[4]:
```python
fetch_housing_data()
housing = load_housing_data()
housing.head()
```

[4]:
```
   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
```

[5]:
```python
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
```

[6]:
```python
housing["ocean_proximity"].value_counts()
```

[6]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

[7]:
```python
housing.describe()
```

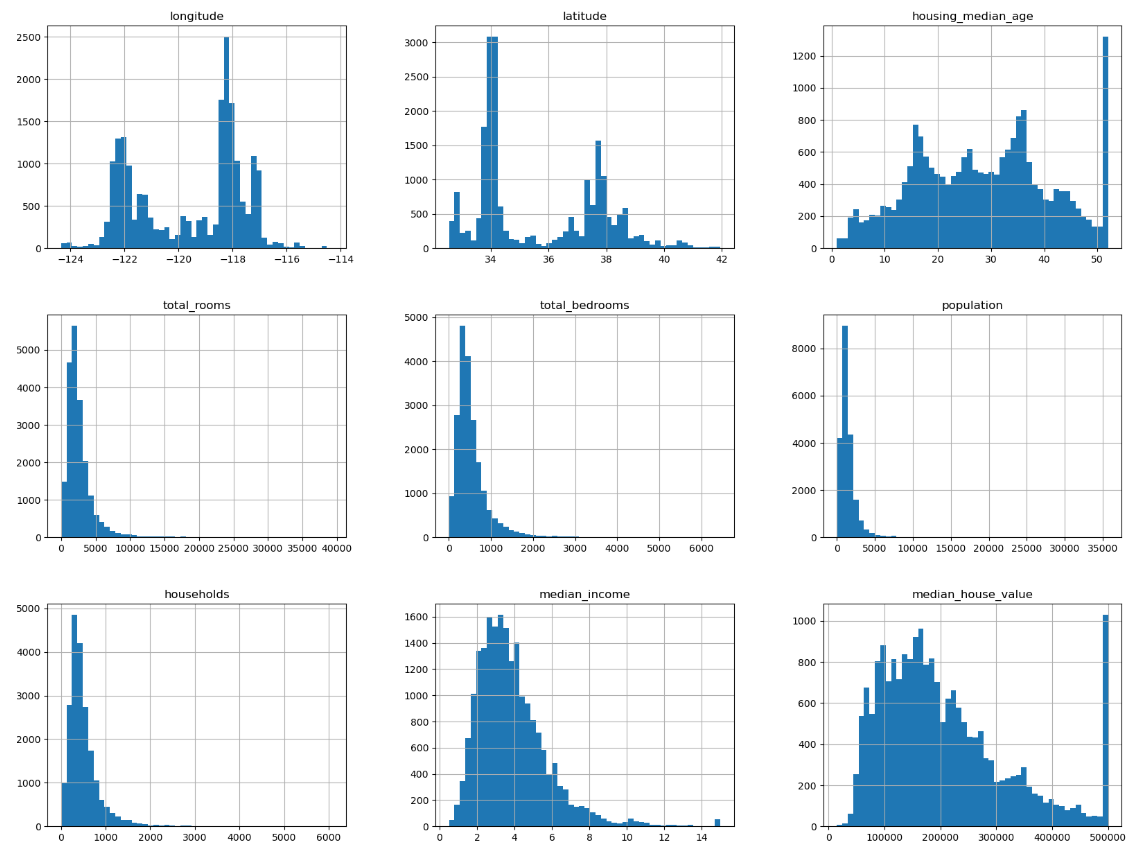
[7]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     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
```

[8]:

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



0.0.3 Training-Testing Split

[9]:
```python
import numpy as np

def split_train_test(data, test_ratio):
    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]
```

[10]:
```python
train_set, test_set = split_train_test(housing, 0.2)
len(train_set)
```

[10]: 16512

[11]:
```python
len(test_set)
```

[11]: 4128

0.0.4 Making sure split stays constant by hashing row index

[12]:
```
from zlib import crc32
def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32
def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]

housing_with_id = housing.reset_index()
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

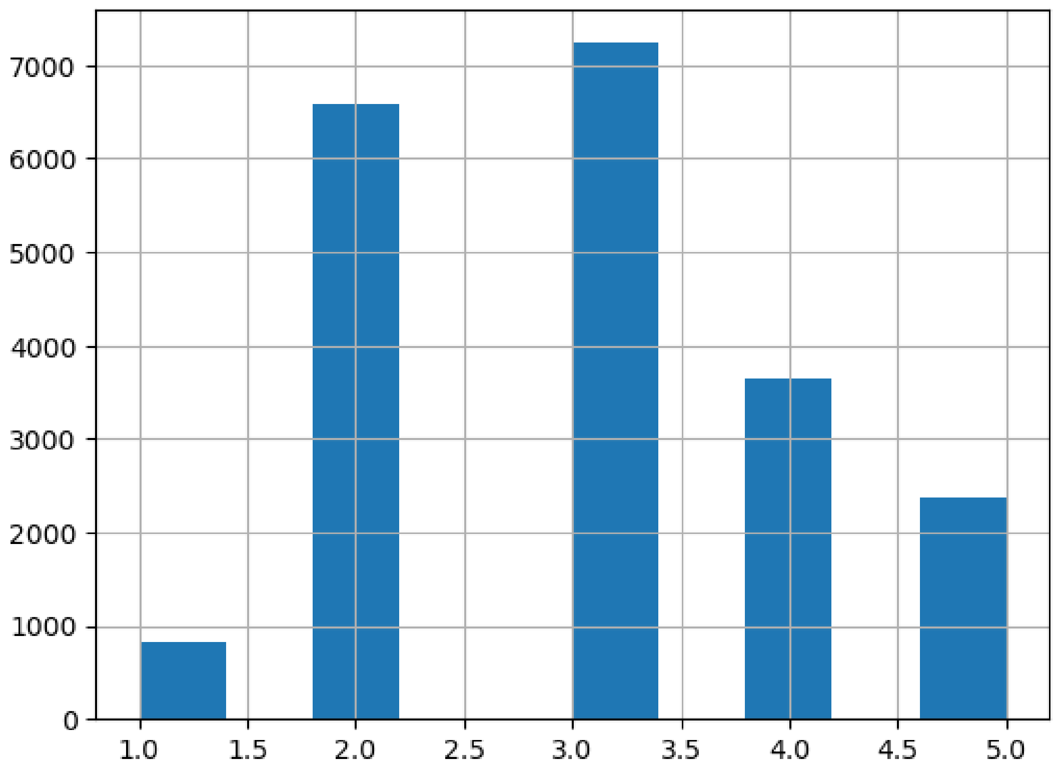
0.0.5 Can use sklearn for built in random split

[13]:
```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

0.0.6 Discretizing median income

[14]:
```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
housing["income_cat"].hist()
```

[14]:
```
<AxesSubplot: >
```



0.0.7

[15]:
```
# Splitting test and training in a stratified manner. i.e. to the proportion of␣
↪median income category groups

from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

[16]:
```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

[16]:
```
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: income_cat, dtype: float64
```

0.0.8

[17]:
```
for set_ in (strat_train_set, strat_test_set): # Dropping income category
    set_.drop("income_cat", axis=1, inplace=True)
```

0.0.9 Some more exploratory analysis

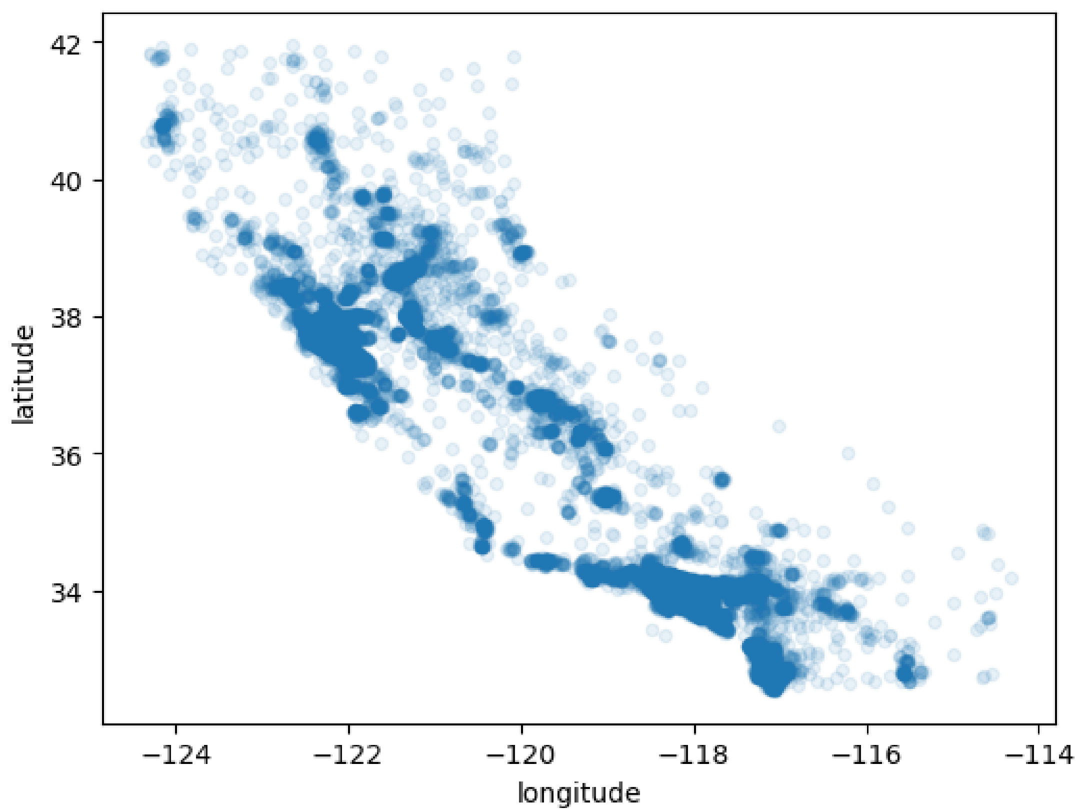
[18]:
```
# Copying training set to do some more exploratory analysis

housing = strat_train_set.copy()
```

[19]:
```
# Spacial visualisation of data

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

[19]: <AxesSubplot: xlabel='longitude', ylabel='latitude'>

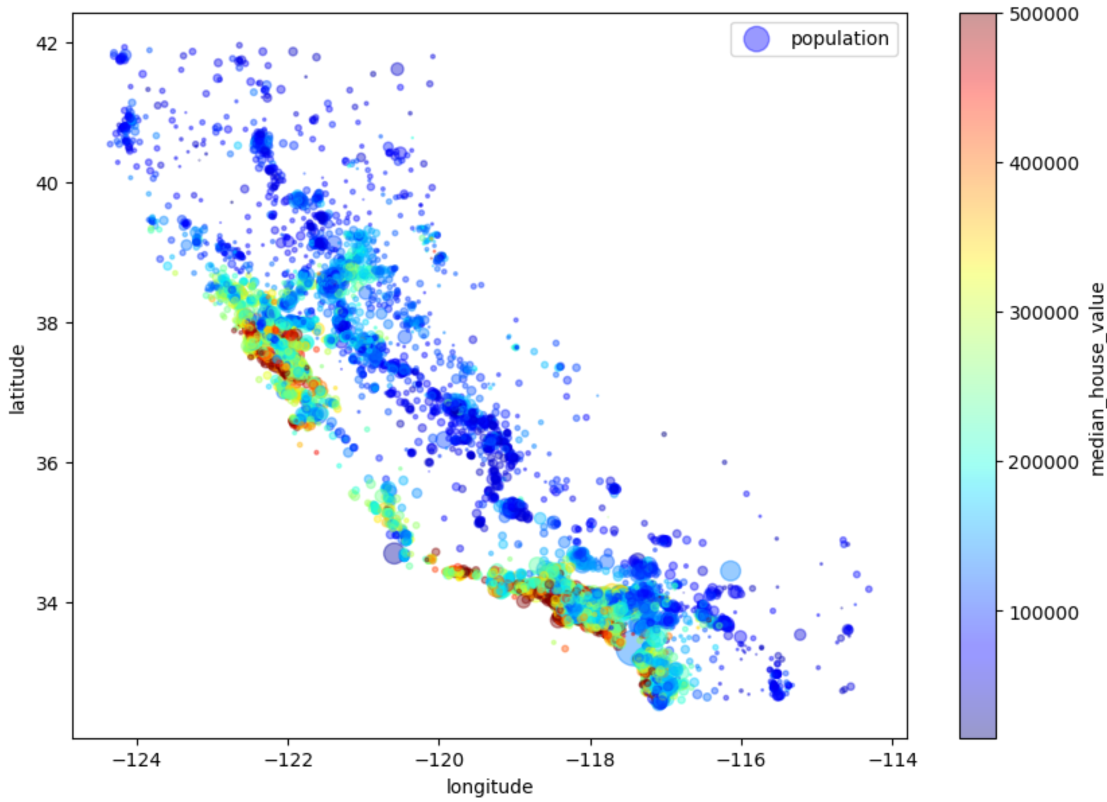


[20]:
```
# Adding more information to the visualization, size = population, colour =␣
↪housing price

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()
```

[20]: <matplotlib.legend.Legend at 0x7effe3445250>



[21]:
```
# Now we find correlation between each attribute and median housing price

corr_matrix = housing.corr(numeric_only=True)
corr_matrix["median_house_value"].sort_values(ascending=False)
```

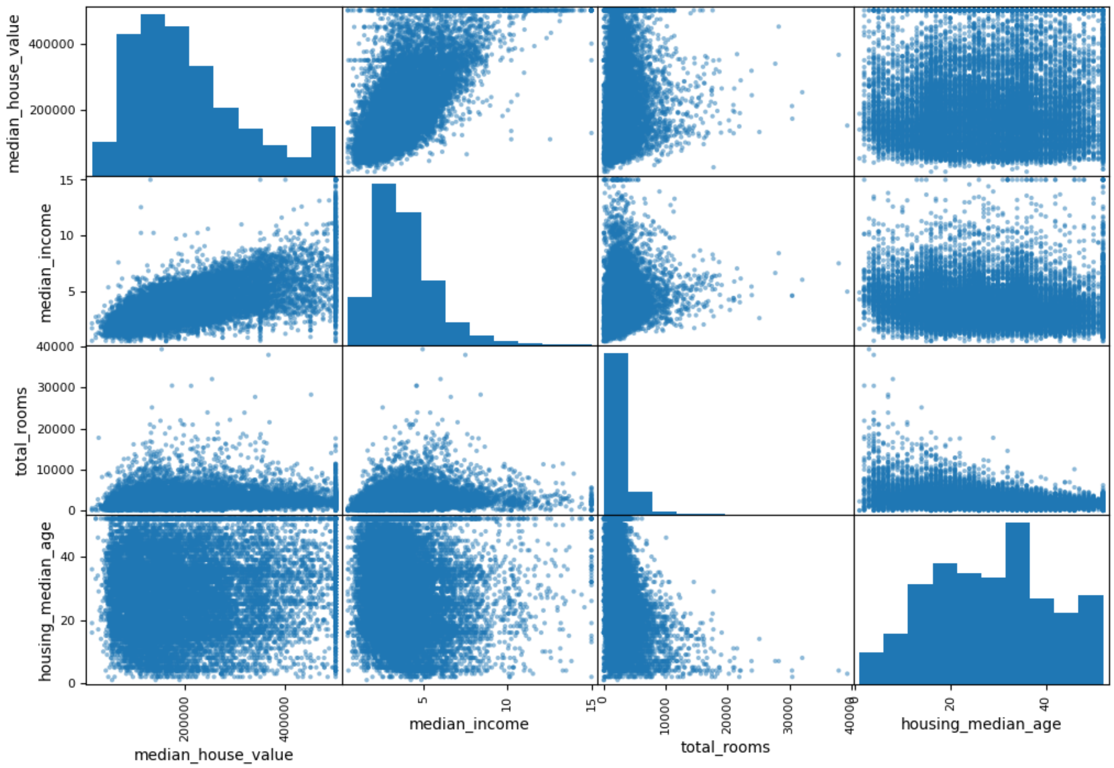
[21]:
```
median_house_value    1.000000
median_income         0.687151
total_rooms           0.135140
housing_median_age    0.114146
households            0.064590
total_bedrooms        0.047781
population           -0.026882
longitude            -0.047466
latitude             -0.142673
```

```
Name: median_house_value, dtype: float64
```

[22]:
```python
# Visual of correlation

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

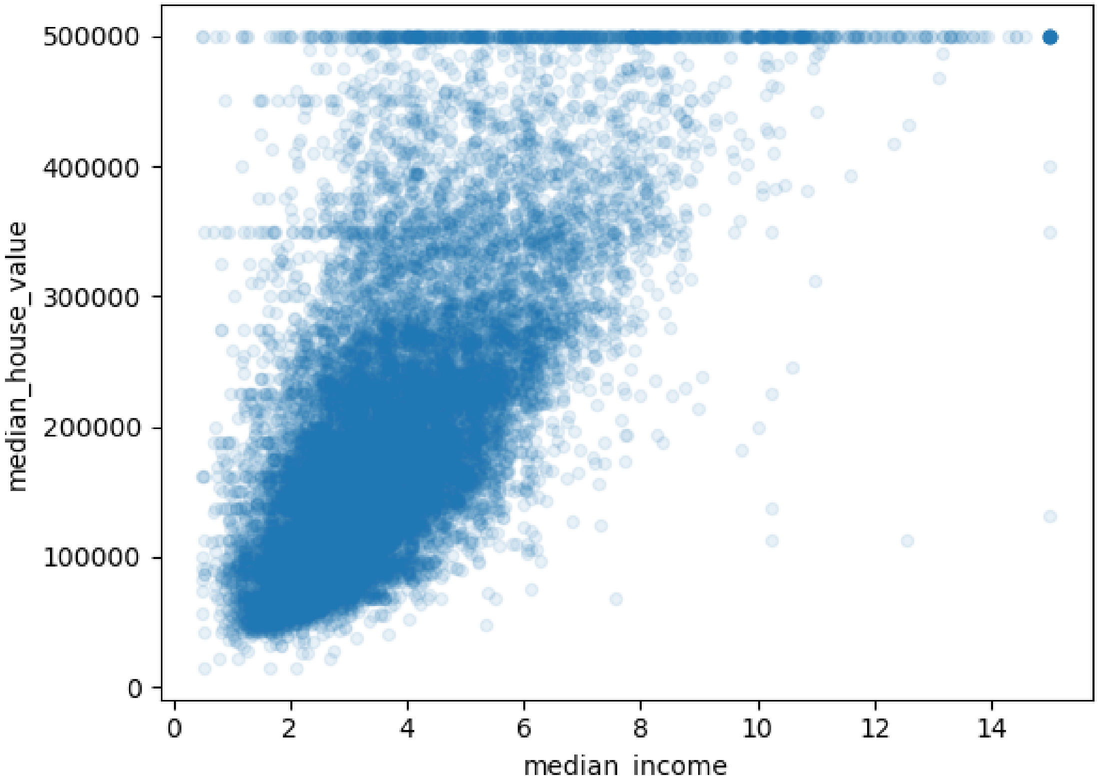
[22]:
```
array([[<AxesSubplot: xlabel='median_house_value', ylabel='median_house_value'>,
        <AxesSubplot: xlabel='median_income', ylabel='median_house_value'>,
        <AxesSubplot: xlabel='total_rooms', ylabel='median_house_value'>,
        <AxesSubplot: xlabel='housing_median_age',
ylabel='median_house_value'>],
       [<AxesSubplot: xlabel='median_house_value', ylabel='median_income'>,
        <AxesSubplot: xlabel='median_income', ylabel='median_income'>,
        <AxesSubplot: xlabel='total_rooms', ylabel='median_income'>,
        <AxesSubplot: xlabel='housing_median_age', ylabel='median_income'>],
       [<AxesSubplot: xlabel='median_house_value', ylabel='total_rooms'>,
        <AxesSubplot: xlabel='median_income', ylabel='total_rooms'>,
        <AxesSubplot: xlabel='total_rooms', ylabel='total_rooms'>,
        <AxesSubplot: xlabel='housing_median_age', ylabel='total_rooms'>],
       [<AxesSubplot: xlabel='median_house_value', ylabel='housing_median_age'>,
        <AxesSubplot: xlabel='median_income', ylabel='housing_median_age'>,
        <AxesSubplot: xlabel='total_rooms', ylabel='housing_median_age'>,
        <AxesSubplot: xlabel='housing_median_age',
ylabel='housing_median_age'>]],
      dtype=object)
```



[23]:
```
# Since we know median income is an important attribute we want a closer look

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

[23]: <AxesSubplot: xlabel='median_income', ylabel='median_house_value'>



[24]:
```
# Creating some attributes of interest

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"]
```

[25]:
```
# Finding the new correlation coefficients

corr_matrix = housing.corr(numeric_only=True)
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[25]:
```
median_house_value          1.000000
median_income               0.687151
rooms_per_household         0.146255
total_rooms                 0.135140
housing_median_age          0.114146
households                  0.064590
total_bedrooms              0.047781
population_per_household   -0.021991
population                 -0.026882
longitude                  -0.047466
```

```
latitude                    -0.142673
bedrooms_per_room           -0.259952
Name: median_house_value, dtype: float64
```

0.0.10 Preparing data for ML

[26]:
```
# Refreshing the training data set

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

[27]:
```
# Replacing null values with median

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
housing_num = housing.drop("ocean_proximity", axis=1) # dropping so only␣
↪numeric values left
imputer.fit(housing_num)
imputer.statistics_
```

[27]:
```
array([-118.51     ,   34.26     ,   29.      , 2119.      ,  433.      ,
       1164.      ,  408.      ,    3.54155])
```

[28]:
```
X = imputer.transform(housing_num)
housing_tr = pd.DataFrame(X, columns=housing_num.columns) # converting np to␣
↪pandas df
```

[29]:
```
# Transforming categorical attribute to numerical via inclusion vector(one hot)

from sklearn.preprocessing import OneHotEncoder

housing_cat = housing[["ocean_proximity"]]
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

[29]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

[35]:
```
# Writing a custom class for a 'Custom Transformer' for combined attributes␣
↪discussed in the attributes of
# interest section.

from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6
```

```
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # bedrooms per room␣
 ↪attribute default to true
        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[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

[36]:
```
# Simple transformation pipeline for numerical data

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")), # First getting rid of null␣
 ↪values or NaN values
    ('attribs_adder', CombinedAttributesAdder()), # Creating attributes of␣
 ↪interest
    ('std_scaler', StandardScaler()), # Scaling the attributes by␣
 ↪standardization.
])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

[37]:
```
# We want work with numerical and categorical data at the same time in the␣
 ↪pipeline

from sklearn.compose import ColumnTransformer
num_attribs = list(housing_num) # turning the columns into a list so we can␣
 ↪transform numerical data
cat_attribs = ["ocean_proximity"] # only one categorical column so a list of 1
full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs), # Calling previous num pipeline␣
 ↪pipeline for numerical data
    ("cat", OneHotEncoder(), cat_attribs), # Using the one hot encoder method␣
 ↪to transform categorical data
])
```

```
housing_prepared = full_pipeline.fit_transform(housing) # If output is mixed␣
↪sparse and dense matrices then final
                                                        # output is checked for␣
↪ratio of zero-nonzero values.
                                                        # default threshold for␣
↪returning a sparse matrix is 0.3
```

0.0.11 Fitting a model onto cleaned data

[38]:
```
# Fitting a Linear Regression Model

from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression() # Specifying model
lin_reg.fit(housing_prepared, housing_labels) # housing_prepared = X,␣
↪housing_labels = median price = y
```

[38]:
```
LinearRegression()
```

[39]:
```
# Trying to predict some values

some_data = housing.iloc[:5] # getting some predictor values
some_labels = housing_labels.iloc[:5] # getting the sample median housing values
some_data_prepared = full_pipeline.transform(some_data) # transforming␣
↪predictors from our pipeline
print("Predictions:", lin_reg.predict(some_data_prepared)) # predicting median␣
↪housing price and printing them
```

```
Predictions: [ 85657.90192014 305492.60737488 152056.46122456 186095.70946094
 244550.67966089]
```

[40]:
```
# Comparing predicted to sample

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

```
Labels: [72100.0, 279600.0, 82700.0, 112500.0, 238300.0]
```

[41]:
```
# Getting a more quantitative look by finding the standard deviation (root MSE␣
↪= roor variance)

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
```

[41]:
```
68627.87390018745
```

This standard deviation is very high and suggests undefitting. To provide a better approximation we can try more combined attributes or transform the data to reduce variance. This data however as we can see from the exploratory analysis is not really linear and in some cases not homoschedastic. This means it violates some base assumptions for the linear regression model.Transformations may help a lot to reduce heteroschedasticity (e.g. root transform). But honestly it doesn't look like this data will play well with linear regression models.

We are going to try a decision tree instead.

[42]:
```
from sklearn.tree import DecisionTreeRegressor

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

[42]:
```
DecisionTreeRegressor()
```

[43]:
```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

[43]:
```
0.0
```

This standard deviation is a little suspicious, the decision tree is probably overfit onto the data. We want to further split the training data into training and validation data so as not to look at test data until we are ready to launch the model. We can do it manually, or use k-fold cross validation.

[44]:
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
    scoring="neg_mean_squared_error", cv=10) # Ok so I understand that the␣
 ↪cross-validation expects a utility
                                             # instead of a cost function so we␣
 ↪invert the sign but I'm not
                                             # totally sure how this neg-MSE is␣
 ↪calculated. I'm sure it'll come up
                                             # later in the book. Otherwise␣
 ↪I'll google it.
tree_rmse_scores = np.sqrt(-scores) # for now I'm going to black-box it and␣
 ↪assume it is some negative variance.
```

[45]:
```
def display_scores(scores): # The creation of a function I guess is useful if␣
 ↪we need to keep displaying scores
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())
display_scores(tree_rmse_scores)
```

```
Scores: [72477.76865825 70330.20733195 68329.67315153 72789.7139895
 71288.28735686 75971.36944257 70581.28965744 73934.71844595
```

```
 67145.98796546 72356.18718466]
Mean: 71520.5203184169
Standard deviation: 2460.384054031556
```

We do see that the decision trees are quite badly overfitting, performing worse than the linear regression.

[46]:
```python
# Computing k-fold for linear regression for a fair comparison

lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
    scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

```
Scores: [71762.76364394 64114.99166359 67771.17124356 68635.19072082
 66846.14089488 72528.03725385 73997.08050233 68802.33629334
 66443.28836884 70139.79923956]
Mean: 69104.07998247063
Standard deviation: 2880.328209818062
```

And yes, the decision tree on average performs worse than the linear regression model. Although we have slightly lower standard deviaton in the decision tree. We try a random forest.

[50]:
```python
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)

housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
print(forest_rmse)
```

```
18715.906994628574
```

[52]:
```python
# k-fold

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,␣
 ↪scoring = "neg_mean_squared_error",
                                cv = 10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
Scores: [51296.1090341  49070.91558851 46512.19200197 52085.86483388
 47561.49613468 51780.09717729 52693.9054573  49726.93595841
 48460.13272188 53729.18846826]
Mean: 50291.6837376268
Standard deviation: 2256.82774468693
```

The difference between the 2 previous tests still indicate some overfitting but the forest is performing much better than the tree.

[ ]:
```
# Some code to save and load models

from sklearn.externals import joblib
joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

0.0.12 Model Tuning

[53]:
```
# Using grid search to try out some hyperparameters

from sklearn.model_selection import GridSearchCV
param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring='neg_mean_squared_error',
    return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

[53]:
```
GridSearchCV(cv=5, estimator=RandomForestRegressor(),
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             return_train_score=True, scoring='neg_mean_squared_error')
```

[54]:
```
# As the var name states, the best parameters

grid_search.best_params_
```

[54]:
```
{'max_features': 6, 'n_estimators': 30}
```

[55]:
```
# Performance of all parameters

cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
        print(np.sqrt(-mean_score), params)
```

```
63735.469487444585 {'max_features': 2, 'n_estimators': 3}
55403.85542910621 {'max_features': 2, 'n_estimators': 10}
52568.90965169572 {'max_features': 2, 'n_estimators': 30}
59619.50829563403 {'max_features': 4, 'n_estimators': 3}
53154.31424921435 {'max_features': 4, 'n_estimators': 10}
50580.67477136508 {'max_features': 4, 'n_estimators': 30}
58645.4199637894 {'max_features': 6, 'n_estimators': 3}
```

```
51914.09798546415 {'max_features': 6, 'n_estimators': 10}
49803.88699897764 {'max_features': 6, 'n_estimators': 30}
58411.49408514365 {'max_features': 8, 'n_estimators': 3}
52000.73912815425 {'max_features': 8, 'n_estimators': 10}
50040.87085970944 {'max_features': 8, 'n_estimators': 30}
62645.0377674515 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54651.240016868906 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59776.552213217445 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52591.804841512276 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58432.273468394305 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51589.55362199886 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

[58]:
```
# As the book suggests I will try higher parameters since our best parameters␣
↪are the maximum values we have allowed

param_grid = [
    {'n_estimators': [30, 100, 300], 'max_features': [8, 10, 12]},
    {'bootstrap': [False], 'n_estimators': [3, 10, 30], 'max_features': [2, 3,␣
↪4, 5]},
]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring='neg_mean_squared_error',
    return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

[58]:
```
GridSearchCV(cv=5, estimator=RandomForestRegressor(),
             param_grid=[{'max_features': [8, 10, 12],
                          'n_estimators': [30, 100, 300]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4, 5],
                          'n_estimators': [3, 10, 30]}],
             return_train_score=True, scoring='neg_mean_squared_error')
```

[59]:
```
# Printing the best parameters

grid_search.best_params_
```

[59]:
```
{'max_features': 8, 'n_estimators': 300}
```

[62]:
```
len(housing_prepared[0])
```

[62]:
```
16
```

We maxed out once again at the n_estimators parameter but max_features stayed at 8 even though we have created attributes.

[63]:
```
# Finding the relative importance of attributes for the models, not exactly␣
↪sure how this is calculated but I
```

```
# know what it means at least, again if it doesn't come up later in the book I
↪will google it. Looks like regression
# coefficients so I'm going to assume that thats what it is for now.

feature_importances = grid_search.best_estimator_.feature_importances_
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

[63]:
```
[(0.36471059138871365, 'median_income'),
 (0.16306207877851275, 'INLAND'),
 (0.11328480109052602, 'pop_per_hhold'),
 (0.0681366415438688, 'longitude'),
 (0.06281043324400132, 'bedrooms_per_room'),
 (0.06185649717557481, 'latitude'),
 (0.050812762384354705, 'rooms_per_hhold'),
 (0.043216190260155565, 'housing_median_age'),
 (0.01556925339060555, 'total_rooms'),
 (0.015140036559351002, 'population'),
 (0.014716439192159838, 'total_bedrooms'),
 (0.014468583342902476, 'households'),
 (0.006208807500706387, '<1H OCEAN'),
 (0.0034569141169565514, 'NEAR OCEAN'),
 (0.002466396860320856, 'NEAR BAY'),
 (8.357317128976957e-05, 'ISLAND')]
```

0.0.13 Model Evaluation on Test Data

[66]:
```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test) # Note we call transform
↪instead of fit transform

final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
final_rmse
```

[66]: 47216.496720307274

[67]:
```
# We create a 95% CI to evaluate the model performance
from scipy import stats
confidence = 0.95
```

```
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1, # Confidence and␣
 ↪DoF
    loc=squared_errors.mean(), # Estimate of mean of squared errors
    scale=stats.sem(squared_errors))) # Standard error of squared errors
```

[67]: array([45226.19357216, 49126.23067213])

0.0.14 Results and Reflection

Results The result is that our random forest model performed the best with hyper-parameters: max_features = 8 and n_estimators = 300. The latter parameter will probably affact the model for the better as it increases.

We also found that median income is the best single predictor of the median home price of an area with the Inland category of ocean proximity being the next best at half the importance score. We could, based off this knowledge, reduce the parameters to the best performing ones and see how it affects our prediction since the less parameters we have the better the computational complexity of the program.

Our 95% confidence interval [45226.19, 49126.23] indicates that our mean squared error will lie in the interval in 95 of 100 CI intervals calculated like we did with independent samples.

Reflection I found that this project was very accessible given that I took a course in Linear Models. There are still some things that are blurry like the way that the importance of features were calculated and what exactly decision trees and random forests are. I know for sure that decision trees and random forests will be discussed later in the textbook, I will probably have to check out how the importance of features were calculated elsewhere.

I really enjoyed this project, as someone who has been programming and learning about statistics its nice to know that ML is quite accessible at this level for someone like me. This project opened my eyes to some of the methods used in ML and how my intuitions from statistics can help me understand the inner workings of machine learning. Things like splitting for training and testing seemed obvious and in line with what I learned in statistics. The created attributes remind me of interaction terms in linear regression and with regression I know which assumptions were violated and why the linear regression didn't really work.

I look forward to finishing the textbook and becomming a useful machine learning engineer.