# Sickit-Learn for Machine Learning

In this notebook we will work through an example project end to end using Sickit Learn library.

# About

In this chapter we’ll use the California Housing Prices dataset from the StatLib repository This dataset is based on data from the 1990 California census.

This data includes metrics such as the population, median income, and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typical).

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

Goal : Your boss answers that your model’s output (a prediction of a district’s median housing price) will be fed to another Machine Learning system , along with many other signals. This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.

First, you need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning? Is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques?

Let’s see: it is clearly a typical supervised learning task, since you are given labeled training examples.

It is also a typical regression task, since you are asked to predict a value. More specifically, this is a multiple regression problem, since the system will use multiple features to make a prediction.

It is also a univariate regression problem, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a multivariate regression problem.

Finally, there is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain batch learning should do just fine.

# Select a Performance Measure

## Performance Measures for our univariate regression problem

Typical performance measure for regression problems :

* *Root Mean Square Error (RMSE)*: it gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors : Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE should be more useful when large errors are particularly undesirable.
* RMSE is sensitive to outliers : If we make a single very bad prediction, taking the square will make the error even worse and it may skew the metric towards overestimating the model’s badness. Actually, it’s hard to realize if our model is good or not by looking at the absolute values of MSE or MSE : We would probably want to measure how much our model is better than the constant baseline : A model should at least perform better than the RMSE score constant baseline.
* RMSE has the benefit of penalizing large errors more so can be more appropriate in some cases, for example, if being off by 10 is more than twice as bad as being off by 5. But if being off by 10 is just twice as bad as being off by 5, then MAE is more appropriate.
* *Root Mean Square Log Error (RMSLE)*: It is an extension on root Mean Squared Error (RMSE) that is mainly used when predictions have large deviations
* RMSLE is preferable when :
  + targets having exponential growth, such as population counts, average sales of a commodity over a span of years etc
  + we care about percentage errors rather than the absolute value of errors : The reason we use log is because generally, you care not so much about missing by €10 but missing by 10%. So if it was €1000,000 item and you are €100,000 off or if it was a 10,000 item and you are €1,000 off — we would consider those equivalent scale issues.
  + There is a wide range in the target variables and we don’t want to penalize big differences when both the predicted and the actual are big numbers.
  + We want to penalize under estimates more than over estimates.
  + Let's imagine two cases of predictions,
  + Case-1: our model makes a prediction of 30 when the actual number is 40 Case-2: our model makes a prediction of 300 when the actual number is 400
  + With RMSE the second result is scored as 10 times more than the first result Conversely, with RMSLogE two results are scored the same. RMSLogE takes into account just the ratio of change Lets have a look at the below example
  + Case-3 : Prediction = 600, Actual = 1000 (the absolute difference is 400)
  + RMSE = 400, RMSLogE = 0.5108
  + Case-4 : Prediction = 1400, Actual = 1000 (the absolute difference is 400)
  + RMSE = 400, RMSLogE = 0.3365
  + When the differences are the same between actual and predicted in both cases. RMSE treated them equally, however RMSLogE penalized the under estimate more than over estimate (under estimated prediction score is higher than over estimated prediction score). Often, penalizing the under estimate more than over estimate is important for prediction of sales and inventory demands.
* *Mean Absolute Error (MAE)*: also called the average absolute deviation : MAE measures the average magnitude of the errors in a set of predictions, without considering their direction.
* R-Squared (R2) : proportional improvement in prediction of the regression model, compared to the mean model (model predicting all given samples as mean value) : - If we were exactly as effective as just predicting the mean, SSres/SStot = 1 and R² = 0 - If we were perfect (i.e. yi = fi for all cases), SSres/SStot = 0 and R² = 1 However, it does not take into consideration of overfitting problem.
  + Interpreted as the proportion of total variance that is explained by the model.
  + R² is the ratio between how good your model is (RMSE)vs. how good is the naïve mean model (RMSE).

## RMSE vs RMSLE vs MAE

See links below :

* RMSE vs MAE : <https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d>
* RMSLE metric and defining baseline : <https://www.kaggle.com/carlolepelaars/understanding-the-metric-rmsle/notebook>
* Model fot metrics : <https://www.kaggle.com/residentmario/model-fit-metrics>

## Scikit-learn implementation

### R-square :  
  
# sklean  
from sklearn.metrics import r2\_score  
  
# hand implemetation   
import numpy as np  
  
def r2\_score(y, y\_pred):  
 rss\_adj = np.sum((y - y\_pred)\*\*2)  
 n = len(y)  
 y\_bar\_adj = (1 / n) \* np.sum(y)  
 ess\_adj = np.sum((y - y\_bar\_adj)\*\*2)  
 return 1 - rss\_adj / ess\_adj  
  
r2\_score(y, y\_pred)  
  
  
### Root Mean Squared Error (RMSE)  
  
from sklearn.metrics import mean\_squared\_error  
mean\_squared\_error(y,y\_pred, squared = False)  
  
# hand implemetation   
import math  
def rmse(y, y\_pred):  
 return math.sqrt( ((y-y\_pred)\*\*2).mean() )  
  
root\_mean\_squared\_error(y, y\_pred)  
  
  
### Root Mean log Squared Error (RMLSE)  
  
from sklearn.metrics import mean\_squared\_log\_error  
mean\_squared\_error(y,y\_pred, squared = False)  
  
# or   
import numpy as np  
y = np.log(df.y)  
RMSLE = rmse(y,y\_pred)  
  
  
### Mean Absolute Error (MAE)   
from sklearn.metrics import mean\_absolute\_error  
  
# hand implemetation   
import numpy as np  
def mae(y,y\_pred):  
 return (np.abs(y-y\_pred)).mean()

---------------------------------------------------------------------------  
NameError Traceback (most recent call last)  
<ipython-input-1061-7e2a74a2dd02> in <module>  
 14 return 1 - rss\_adj / ess\_adj  
 15   
---> 16 r2\_score(y, y\_pred)  
 17   
 18   
  
NameError: name 'y\_pred' is not defined

# Download the Data

PATH = '/Users/rmbp/handson-ml2/datasets/'  
!ls {PATH}

housing inception jsb\_chorales lifesat titanic

import pandas as pd  
  
housing = pd.read\_csv(f'{PATH}/housing/housing.csv')  
housing.head()

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

## Automating the process of fetching and loading the data

import os  
import tarfile  
import urllib  
  
DOWNLOAD\_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"  
HOUSING\_PATH = os.path.join("/Users/rmbp/Desktop", "housing")  
HOUSING\_URL = DOWNLOAD\_ROOT + "datasets/housing/housing.tgz"  
def fetch\_housing\_data(housing\_url=HOUSING\_URL, housing\_path=HOUSING\_PATH):  
 os.makedirs(housing\_path, exist\_ok=True)  
 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()

fetch\_housing\_data(HOUSING\_URL,HOUSING\_PATH)

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)

load\_housing\_data().head()

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

## Take a Quick Look at the Data Structure

housing.head()

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

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

There are 20,640 instances in the dataset. Notice that the total\_bedrooms attribute has only 20,433 nonnull values, meaning that 207 districts are missing this feature. All attributes are numerical, except the ocean\_proximity field. Its type is object. Since we loaded this data from a CSV file, it must be a text attribute. : the values in the ocean\_proximity column were repetitive, which means that it is probably a categorical attribute.

housing['ocean\_proximity'].value\_counts()

<1H OCEAN 9136  
INLAND 6551  
NEAR OCEAN 2658  
NEAR BAY 2290  
ISLAND 5  
Name: ocean\_proximity, dtype: int64

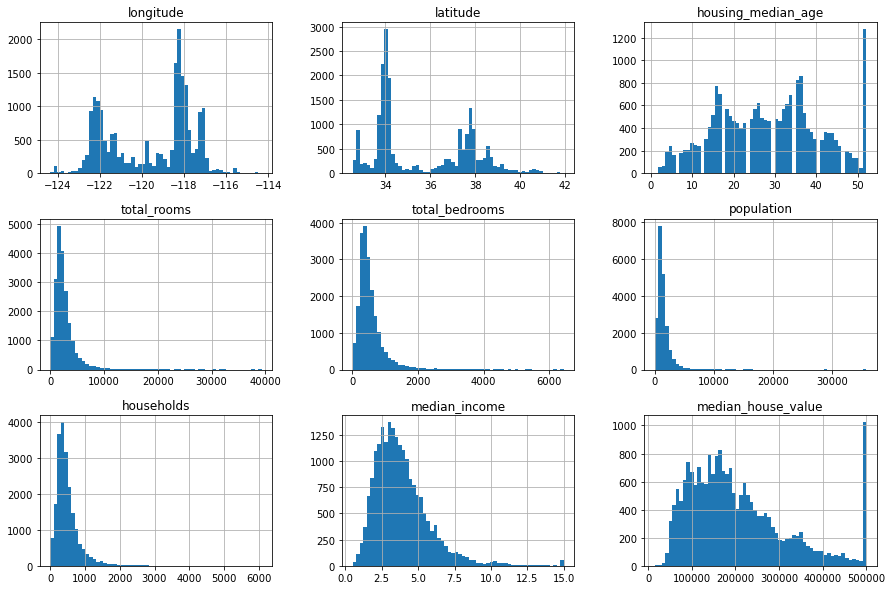
housing.describe(include='all').T

count unique top freq mean \  
longitude 20640.0 NaN NaN NaN -119.569704   
latitude 20640.0 NaN NaN NaN 35.631861   
housing\_median\_age 20640.0 NaN NaN NaN 28.639486   
total\_rooms 20640.0 NaN NaN NaN 2635.763081   
total\_bedrooms 20433.0 NaN NaN NaN 537.870553   
population 20640.0 NaN NaN NaN 1425.476744   
households 20640.0 NaN NaN NaN 499.53968   
median\_income 20640.0 NaN NaN NaN 3.870671   
median\_house\_value 20640.0 NaN NaN NaN 206855.816909   
ocean\_proximity 20640 5 <1H OCEAN 9136 NaN   
  
 std min 25% 50% 75% \  
longitude 2.003532 -124.35 -121.8 -118.49 -118.01   
latitude 2.135952 32.54 33.93 34.26 37.71   
housing\_median\_age 12.585558 1.0 18.0 29.0 37.0   
total\_rooms 2181.615252 2.0 1447.75 2127.0 3148.0   
total\_bedrooms 421.38507 1.0 296.0 435.0 647.0   
population 1132.462122 3.0 787.0 1166.0 1725.0   
households 382.329753 1.0 280.0 409.0 605.0   
median\_income 1.899822 0.4999 2.5634 3.5348 4.74325   
median\_house\_value 115395.615874 14999.0 119600.0 179700.0 264725.0   
ocean\_proximity NaN NaN NaN NaN NaN   
  
 max   
longitude -114.31   
latitude 41.95   
housing\_median\_age 52.0   
total\_rooms 39320.0   
total\_bedrooms 6445.0   
population 35682.0   
households 6082.0   
median\_income 15.0001   
median\_house\_value 500001.0   
ocean\_proximity NaN

hist() method on the whole dataset will plot a histogram for each numerical attribute

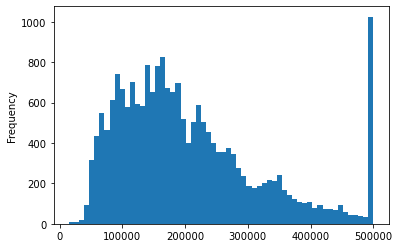
* A histogram is used for continuous data, where the bins represent ranges of data, counts the data points in each bin, and shows the bins on the x-axis and the counts on the y-axis. : <https://towardsdatascience.com/histograms-and-density-plots-in-python-f6bda88f5ac0>
* A bar chart is a plot of categorical variables.

#This tells Jupyter to set up Matplotlib so it uses Jupyter’s own backend.  
%matplotlib inline   
import matplotlib.pyplot as plt  
housing.hist(bins=60, figsize=(15,10))  
plt.show()



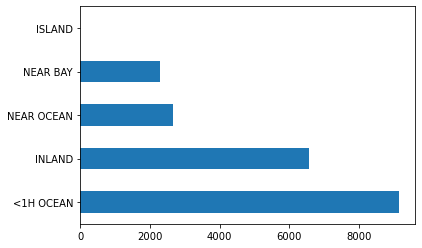
# plot 'median\_house\_value'  
housing['median\_house\_value'].plot(kind='hist', bins= 60)

<AxesSubplot:ylabel='Frequency'>



housing['ocean\_proximity'].value\_counts().plot(kind= 'barh')

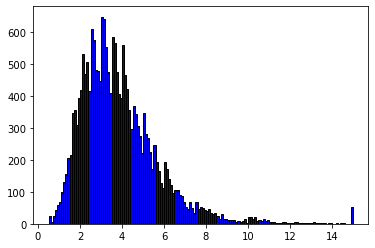
<AxesSubplot:>



pd.DataFrame(housing['median\_income'].describe()).T

count mean std min 25% 50% 75% \  
median\_income 20640.0 3.870671 1.899822 0.4999 2.5634 3.5348 4.74325   
  
 max   
median\_income 15.0001

n, bins, patches = plt.hist(housing.median\_income, bins = int((15.000100 - 0.499900)/0.1),edgecolor = 'black'  
 ,color = 'blue')  
# bins = int((15.000100 - 0.499900)/0.1) : We choose the number of bins with an interval lenght of 100€

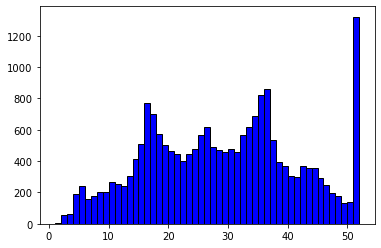


pd.DataFrame(housing['housing\_median\_age'].describe()).T

count mean std min 25% 50% 75% max  
housing\_median\_age 20640.0 28.639486 12.585558 1.0 18.0 29.0 37.0 52.0

n, bins, patches = plt.hist(housing.housing\_median\_age, bins = int((52.000000 - 1.000000)/1)  
 , color = 'blue'  
 , edgecolor = 'black')  
bins

array([ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12., 13.,  
 14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24., 25., 26.,  
 27., 28., 29., 30., 31., 32., 33., 34., 35., 36., 37., 38., 39.,  
 40., 41., 42., 43., 44., 45., 46., 47., 48., 49., 50., 51., 52.])

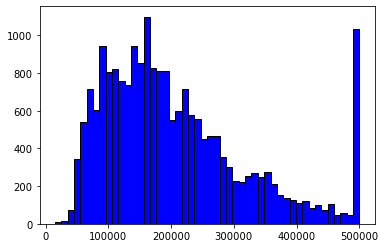


# Target  
pd.DataFrame(housing['median\_house\_value'].describe()).T

count mean std min 25% \  
median\_house\_value 20640.0 206855.816909 115395.615874 14999.0 119600.0   
  
 50% 75% max   
median\_house\_value 179700.0 264725.0 500001.0

n, bins, patches = plt.hist(housing.median\_house\_value , bins = int((500001.000000 - 14999.000000)/10000)  
 , color = 'blue'  
 ,edgecolor = 'black')  
bins

array([ 14999. , 25103.20833333, 35207.41666667, 45311.625 ,  
 55415.83333333, 65520.04166667, 75624.25 , 85728.45833333,  
 95832.66666667, 105936.875 , 116041.08333333, 126145.29166667,  
 136249.5 , 146353.70833333, 156457.91666667, 166562.125 ,  
 176666.33333333, 186770.54166667, 196874.75 , 206978.95833333,  
 217083.16666667, 227187.375 , 237291.58333333, 247395.79166667,  
 257500. , 267604.20833333, 277708.41666667, 287812.625 ,  
 297916.83333333, 308021.04166667, 318125.25 , 328229.45833333,  
 338333.66666667, 348437.875 , 358542.08333333, 368646.29166667,  
 378750.5 , 388854.70833333, 398958.91666667, 409063.125 ,  
 419167.33333333, 429271.54166667, 439375.75 , 449479.95833333,  
 459584.16666667, 469688.375 , 479792.58333333, 489896.79166667,  
 500001. ])



From the figure below, we can see how the data was computed :

* We can see that median\_income was scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. The numbers represent roughly tens of thousands of dollars.
* The housing median age and the median house value were also capped. The latter may be a serious problem since it is our target attribute. In this caseour Machine Learning algorithms may learn that prices never go beyond that limit (€500,000). We need to check with our team to see if this is a problem or not. If the team needs precise predictions even beyond €500,000, then you have two options:
  + Collect proper labels for the districts whose labels were capped.
  + Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond €500,000).

We can also see that :

* These attributes have very different scales.
* Many histograms are tail-heavy : they extend much farther to the right of the median than to the left.

# Create a Test Set

We ahve only taken a quick glance at the data : numeric / categorical features, missing values, scale of attributes, distribution, how values are computed, distribution of the target variable. It's enough. Why ?

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

## Train and test set stability

Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset

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]

train\_set, test\_set = split\_train\_test(housing,0.2)  
print(len(train\_set))  
print(len(test\_set))

16512  
4128

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set!

* Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid.

# first run test set  
split\_train\_test(housing,0.2)[1].head(5)

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
12003 -117.57 33.90 7.0 3797.0 850.0   
13304 -117.63 34.09 19.0 3490.0 816.0   
19037 -121.99 38.36 35.0 2728.0 451.0   
9871 -121.82 36.61 24.0 2437.0 438.0   
16526 -121.20 37.80 37.0 311.0 61.0   
  
 population households median\_income median\_house\_value \  
12003 2369.0 720.0 3.5525 137600.0   
13304 2818.0 688.0 2.8977 126200.0   
19037 1290.0 452.0 3.2768 117600.0   
9871 1430.0 444.0 3.8015 169100.0   
16526 171.0 54.0 4.0972 101800.0   
  
 ocean\_proximity   
12003 INLAND   
13304 INLAND   
19037 INLAND   
9871 <1H OCEAN   
16526 INLAND

# second run test set  
split\_train\_test(housing,0.2)[1].head(5)

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
5709 -118.23 34.21 32.0 1464.0 406.0   
16381 -121.30 38.02 4.0 1515.0 384.0   
16458 -121.30 38.13 26.0 2256.0 360.0   
8613 -118.37 33.87 23.0 1829.0 331.0   
2738 -115.56 32.78 35.0 1185.0 202.0   
  
 population households median\_income median\_house\_value \  
5709 693.0 380.0 2.5463 200000.0   
16381 491.0 348.0 2.8523 87500.0   
16458 937.0 372.0 5.0528 153700.0   
8613 891.0 356.0 6.5755 359900.0   
2738 615.0 191.0 4.6154 86200.0   
  
 ocean\_proximity   
5709 <1H OCEAN   
16381 INLAND   
16458 INLAND   
8613 <1H OCEAN   
2738 INLAND

Solution :

* One solution is to save the test set on the first run and then load it in subsequent runs.
* Another option is to set the random number generator’s seed (e.g., with np.ran dom.seed(42))14 before calling np.random.permutation() so that it always generates the same shuffled indices :

import numpy as np  
  
def split\_train\_test(data, test\_ratio):  
 np.random.seed(1997)  
 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]

# first run test set  
split\_train\_test(housing,0.2)[1].head(5)

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
9009 -118.60 34.07 16.0 319.0 59.0   
17779 -121.83 37.38 15.0 4430.0 992.0   
20209 -119.21 34.28 27.0 2219.0 312.0   
3170 -119.69 36.41 38.0 1016.0 202.0   
2200 -119.85 36.83 15.0 2563.0 335.0   
  
 population households median\_income median\_house\_value \  
9009 149.0 64.0 4.6250 433300.0   
17779 3278.0 1018.0 4.5533 209900.0   
20209 937.0 315.0 5.7601 281100.0   
3170 540.0 187.0 2.2885 75000.0   
2200 1080.0 356.0 6.7181 160300.0   
  
 ocean\_proximity   
9009 <1H OCEAN   
17779 <1H OCEAN   
20209 NEAR OCEAN   
3170 INLAND   
2200 INLAND

# second run test set  
split\_train\_test(housing,0.2)[1].head(5)

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
9009 -118.60 34.07 16.0 319.0 59.0   
17779 -121.83 37.38 15.0 4430.0 992.0   
20209 -119.21 34.28 27.0 2219.0 312.0   
3170 -119.69 36.41 38.0 1016.0 202.0   
2200 -119.85 36.83 15.0 2563.0 335.0   
  
 population households median\_income median\_house\_value \  
9009 149.0 64.0 4.6250 433300.0   
17779 3278.0 1018.0 4.5533 209900.0   
20209 937.0 315.0 5.7601 281100.0   
3170 540.0 187.0 2.2885 75000.0   
2200 1080.0 356.0 6.7181 160300.0   
  
 ocean\_proximity   
9009 <1H OCEAN   
17779 <1H OCEAN   
20209 NEAR OCEAN   
3170 INLAND   
2200 INLAND

But both these solutions will break the next time you fetch an updated dataset.

If the dataset is updated, we want to ensure that the test set will remain consistent across multiple runs, even if you refresh the dataset : The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set.

To have a stable train/test split even after updating the dataset, a common solution is to use each instance’s identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier). For example, we could compute a hash of each instance’s identifier and put that instance in the test set if the hash is lower than or equal to 20% of the maximum hash value.

from zlib import crc32  
  
def test\_set\_check(identifier, test\_ratio):  
 return crc32(np.int64(identifier)) & 0xffffffff < test\_ratio \* 2\*\*32  
  
# crc32(np.int64(identifier)) = create a hash from a given value  
# crc32(np.int64(identifier)) & 0xffffffff = make sure the hash value does not exceed 2^32 (or 4294967296).  
# crc32(np.int64(identifier)) & 0xffffffff < test\_ratio \* 2\*\*32.   
# crc32(np.int64(identifier)) & 0xffffffff < test\_ratio \* 2\*\*32  
# This line returns True or False. Let test\_ratio be 0.2.   
# Then, any hash value less than 0.2 \* 4294967296 returns True and will be   
# added to the test set; otherwise, it returns False and will be added to the training set. \*/

def split\_train\_test\_by\_id(data, test\_ratio, id\_column):  
 ids = data[id\_column] # compute a hash of each instance’s identifier  
 in\_test\_set = ids.apply(lambda id\_: test\_set\_check(id\_, test\_ratio)) # if hash is lower than or equal to 20% of the maximum hash value  
 return data.loc[~in\_test\_set], data.loc[in\_test\_set]

Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

housing\_with\_id = housing.reset\_index() # adds an `index` column  
train\_set, test\_set = split\_train\_test\_by\_id(housing\_with\_id, 0.2, "index")

If we use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted. If this is not possible, then we can try to use the most stable features to build a unique identifier.

For example, a district’s latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so:

housing\_with\_id["id"] = housing["longitude"] \* 1000 + housing["latitude"]  
train\_set, test\_set = split\_train\_test\_by\_id(housing\_with\_id, 0.2, "id")

See explanation of this method in : <https://ichi.pro/fr/ameliorez-la-repartition-des-tests-de-train-avec-la-fonction-de-hachage-267796356735483> and <https://datascience.stackexchange.com/questions/51348/splitting-train-test-sets-by-an-identifier>

## Train / Test split using Sckit-learn

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is train\_test\_split(), which does pretty much the same thing as the function split\_train\_test(), with a couple of additional features :

* First, there is a random\_state parameter random\_state that allows you to set the random generator seed and a test size test\_size.
* Second, we can pass it multiple datasets with an identical number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

from sklearn.model\_selection import train\_test\_split  
  
train\_set, test\_set = train\_test\_split(housing, test\_size=0.2, random\_state = 1997)

# Sampling bias in Test set

Using train\_test\_splitmethod, we using purely random sampling methods to generate our test set. This is generally fine if our dataset is large enough (especially relative to the number of attributes), but if it is not, we run the risk of introducing a significant sampling bias.

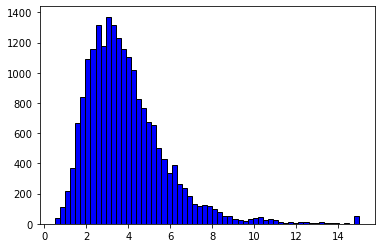
If an attribute (continues or categorical) is important (after discussing with experts for exemple) : We may want to ensure that the test set is representative of the various categories of that variable in the whole dataset.

Suppose that the median income is a very important attribute to predict median housing prices.

housing['median\_income'].describe()

count 20640.000000  
mean 3.870671  
std 1.899822  
min 0.499900  
25% 2.563400  
50% 3.534800  
75% 4.743250  
max 15.000100  
Name: median\_income, dtype: float64

plt.hist(housing['median\_income']  
 #, bins = int( (housing['median\_income'].max() - housing['median\_income'].min()) / 0.5)  
 , bins = 60  
 , color = 'blue'  
 ,edgecolor = 'black'  
 )  
plt.show()

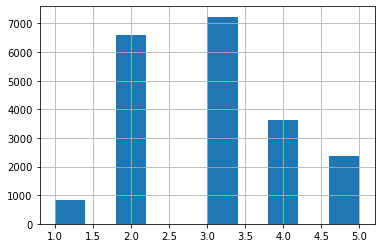


housing["income\_cat"] = pd.cut(housing["median\_income"],  
bins=[0., 1.5, 3.0, 4.5, 6., np.inf],  
labels=[1, 2, 3, 4, 5])

housing['income\_cat'] = 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()

<AxesSubplot:>



housing["income\_cat"].value\_counts() / len(housing)

3 0.350581  
2 0.318847  
4 0.176308  
5 0.114438  
1 0.039826  
Name: income\_cat, dtype: float64

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]

strat\_train\_set["income\_cat"].value\_counts() / len(strat\_train\_set)

3 0.350594  
2 0.318859  
4 0.176296  
5 0.114462  
1 0.039789  
Name: income\_cat, dtype: float64

strat\_test\_set["income\_cat"].value\_counts() / len(strat\_test\_set)

3 0.350533  
2 0.318798  
4 0.176357  
5 0.114341  
1 0.039971  
Name: income\_cat, dtype: float64

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["Rand. %error"] = 100 \* compare\_props["Random"] / compare\_props["Overall"] - 100  
compare\_props["Strat. %error"] = 100 \* compare\_props["Stratified"] / compare\_props["Overall"] - 100

compare\_props

Overall Stratified Random Rand. %error Strat. %error  
1 0.039826 0.039971 0.040213 0.973236 0.364964  
2 0.318847 0.318798 0.324370 1.732260 -0.015195  
3 0.350581 0.350533 0.358527 2.266446 -0.013820  
4 0.176308 0.176357 0.167393 -5.056334 0.027480  
5 0.114438 0.114341 0.109496 -4.318374 -0.084674

* Further analysis later

for set\_ in (strat\_train\_set, strat\_test\_set):  
 set\_.drop("income\_cat", axis=1, inplace=True)

# Discover and Visualize the Data to Gain Insights

First, we make sure that we have put the test set aside and we are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small, so we can just work directly on the full set.

* Let’s create a copy so that you can play with it without harming the training set:

housing = strat\_train\_set.copy()

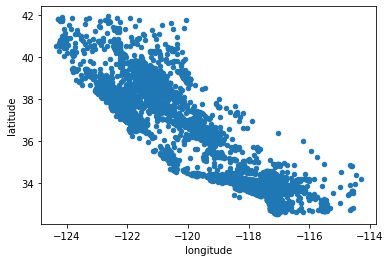
housing.head(5)

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
12655 -121.46 38.52 29.0 3873.0 797.0   
15502 -117.23 33.09 7.0 5320.0 855.0   
2908 -119.04 35.37 44.0 1618.0 310.0   
14053 -117.13 32.75 24.0 1877.0 519.0   
20496 -118.70 34.28 27.0 3536.0 646.0   
  
 population households median\_income median\_house\_value \  
12655 2237.0 706.0 2.1736 72100.0   
15502 2015.0 768.0 6.3373 279600.0   
2908 667.0 300.0 2.8750 82700.0   
14053 898.0 483.0 2.2264 112500.0   
20496 1837.0 580.0 4.4964 238300.0   
  
 ocean\_proximity income\_cat   
12655 INLAND 2   
15502 NEAR OCEAN 5   
2908 INLAND 2   
14053 NEAR OCEAN 2   
20496 <1H OCEAN 3

* Since we have geographic information (lon / lat), let's create a scatterplot of all districts to visualize the data : doc of a scatterplot parameter <https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html>

housing.plot(kind = 'scatter', x = 'longitude', y = 'latitude')

<AxesSubplot:xlabel='longitude', ylabel='latitude'>



Scatter plots work well for hundreds of observations but overplotting becomes an issue once the number of observations gets into tens of thousands.

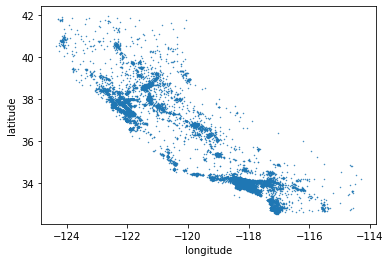
We can see that in some areas, there are vast numbers of dots, so it is hard to see any particular pattern.

Simple options to address overplotting :

* reducing the point size : usisng the s parameter - This parameter indicates the marker size.
* alpha blending : using alpha parameter This option indicates the blending value, between 0 (transparent) and 1 (opaque).

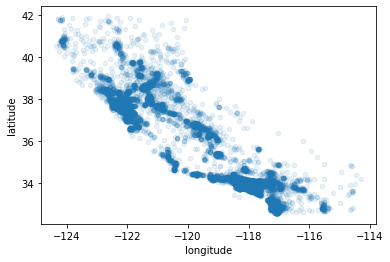
housing.plot(kind = 'scatter' ,x = 'longitude' ,y = 'latitude' , s= 0.2)

<AxesSubplot:xlabel='longitude', ylabel='latitude'>



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

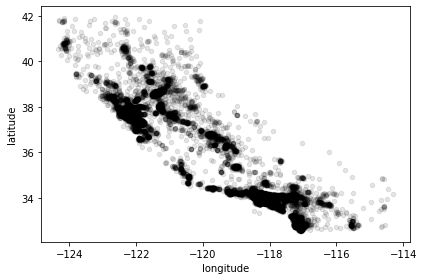
<AxesSubplot:xlabel='longitude', ylabel='latitude'>



We can get the names of the cities in the map and conclude which have the highest density - Many article covers this subject - we will do it later

# To save a picture in our folder project :  
IMAGES\_PATH = "/Users/rmbp/Desktop/housing"  
  
def save\_fig(fig\_id, tight\_layout=True, fig\_extension="png", resolution=300):  
 path = os.path.join(IMAGES\_PATH, fig\_id + "." + fig\_extension)  
 print("Saving figure", fig\_id)  
 if tight\_layout:  
 plt.tight\_layout()  
 plt.savefig(path, format=fig\_extension, dpi=resolution)  
   
housing.plot(kind = 'scatter' ,x = 'longitude' ,y = 'latitude' , alpha= 0.1, c='black')  
save\_fig("better\_visualization\_plot")

Saving figure better\_visualization\_plot

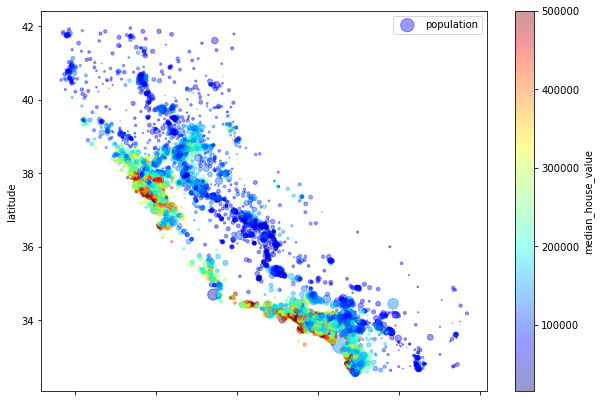


We can see the houses price crossing with the population on the map below :

* the parameter s re presenting the radius of each circle will represents the `district’s population``
* the paramter c representing the color will represents the price.
* We will use a predefined color map (option cmap) called jet, which ranges from blue (low values) to red (high prices):

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()

<matplotlib.legend.Legend at 0x12f4d1650>



* This image tells us that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density.
* A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers. See later.. Check this blog : <https://dev.to/travelleroncode/analyzing-a-dataset-with-unsupervised-learning-31ld>
* The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.

## Looking for correlations

If we want to explore our data it is good to compute correlation between numeric variable : Spearman S and Pearon P. W can compute them both since the relation between the Spearman (S) and Pearson (P) correlations will give some good information :

* Briefly, S is computed on ranks and so depicts monotonic relationships while P is on true values and depicts linear relationships.
* We the corr method : By default, method = 'Pearson'

s = {}  
for x in range(1,100):  
 s[x] = math.exp(x)  
s = pd.DataFrame(s.items())

s.corr('pearson')

0 1  
0 1.000000 0.253274  
1 0.253274 1.000000

s.corr('spearman')

0 1  
0 1.0 1.0  
1 1.0 1.0

This is because 𝑦 increases monotonically with 𝑥 so the Spearman correlation is perfect, but not linearly, so the Pearson correlation is imperfect.

Doing both is interesting because if we have S > P, that means that we have a correlation that is monotonic but not linear. Since it is good to have linearity in statistics (it is easier) we can try to apply a transformation on 𝑦(such a log).

corr\_matrix = housing.corr()

corr\_matrix

longitude latitude housing\_median\_age total\_rooms \  
longitude 1.000000 -0.924478 -0.105823 0.048909   
latitude -0.924478 1.000000 0.005737 -0.039245   
housing\_median\_age -0.105823 0.005737 1.000000 -0.364535   
total\_rooms 0.048909 -0.039245 -0.364535 1.000000   
total\_bedrooms 0.076686 -0.072550 -0.325101 0.929391   
population 0.108071 -0.115290 -0.298737 0.855103   
households 0.063146 -0.077765 -0.306473 0.918396   
median\_income -0.019615 -0.075146 -0.111315 0.200133   
median\_house\_value -0.047466 -0.142673 0.114146 0.135140   
  
 total\_bedrooms population households median\_income \  
longitude 0.076686 0.108071 0.063146 -0.019615   
latitude -0.072550 -0.115290 -0.077765 -0.075146   
housing\_median\_age -0.325101 -0.298737 -0.306473 -0.111315   
total\_rooms 0.929391 0.855103 0.918396 0.200133   
total\_bedrooms 1.000000 0.876324 0.980167 -0.009643   
population 0.876324 1.000000 0.904639 0.002421   
households 0.980167 0.904639 1.000000 0.010869   
median\_income -0.009643 0.002421 0.010869 1.000000   
median\_house\_value 0.047781 -0.026882 0.064590 0.687151   
  
 median\_house\_value   
longitude -0.047466   
latitude -0.142673   
housing\_median\_age 0.114146   
total\_rooms 0.135140   
total\_bedrooms 0.047781   
population -0.026882   
households 0.064590   
median\_income 0.687151   
median\_house\_value 1.000000

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

corr\_matrix['median\_house\_value'].sort\_values(ascending = False)

median\_house\_value 1.000000  
median\_income 0.687151  
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

corr\_matrix = housing.corr('spearman')

corr\_matrix['median\_house\_value'].sort\_values(ascending = False)

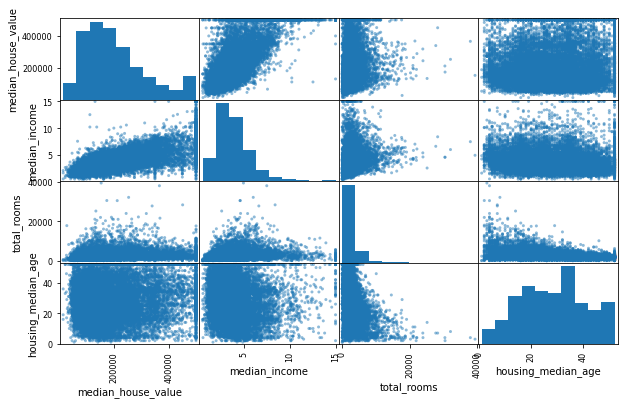
median\_house\_value 1.000000  
median\_income 0.675714  
total\_rooms 0.204476  
households 0.110722  
total\_bedrooms 0.084284  
housing\_median\_age 0.083301  
population 0.001309  
longitude -0.071562  
latitude -0.162283  
Name: median\_house\_value, dtype: float64

Another way to check for correlation between attributes is to use the pandas scatter\_matrix() function, which plots every numerical attribute against every other numerical attribute. ( if we have 11 attribiute, we will plot 11\*\*2 plots )

From the pearson coefficient below, we focus on a few promising attributes that seem most correlated with the median housing value :

from pandas.plotting import scatter\_matrix  
  
attributes = ['median\_house\_value', 'median\_income', 'total\_rooms', 'housing\_median\_age' ]  
  
scatter\_matrix(housing[attributes], figsize=(10,6), )

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)

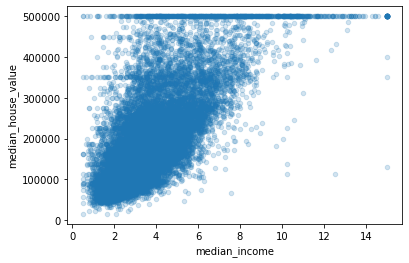


The main diagonal (top left to bottom right) would be full of straight lines if pandas plotted each variable against itself, which would not be very useful. So instead pandas displays a histogram of each attribute. The diagonal option in scatter\_matrix pick between 'kde' and 'hist' for either Kernel Density Estimation or Histogram plot in the diagonal.

The most promising attribute to predict the median house value is the median income( Pearson and Spearman correlation coefficient = 0.67 ), so let’s zoom in on their correlation scatterplot :

housing.plot( kind = 'scatter'  
 ,x = 'median\_income'  
 ,y = 'median\_house\_value'  
 ,alpha = 0.2)

<AxesSubplot:xlabel='median\_income', ylabel='median\_house\_value'>



This plot reveals a few things :

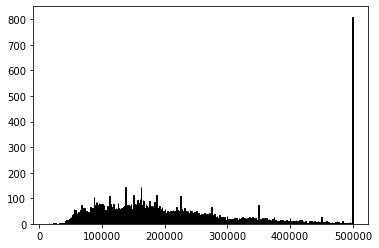
* First, the correlation is indeed very strong; we can clearly see the upward trend, and the points are not too dispersed.
* Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000. But this plot reveals other less obvious straight lines: a horizontal line around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

housing['median\_house\_value'].describe()

count 16512.000000  
mean 207005.322372  
std 115701.297250  
min 14999.000000  
25% 119800.000000  
50% 179500.000000  
75% 263900.000000  
max 500001.000000  
Name: median\_house\_value, dtype: float64

# Data picks in the target variable  
plt.hist(housing['median\_house\_value']  
 #, bins = int( (housing['median\_income'].max() - housing['median\_income'].min()) / 0.5)  
 , bins = int ((500001.000000 - 14999.000000)/1000)  
 , color = 'blue'  
 ,edgecolor = 'black'  
 )

(array([ 3., 0., 0., 0., 0., 0., 0., 3., 0., 0., 0.,  
 2., 1., 1., 0., 1., 0., 4., 1., 3., 1., 2.,  
 4., 2., 3., 4., 5., 10., 13., 11., 14., 11., 19.,  
 17., 21., 27., 24., 36., 33., 39., 59., 30., 53., 50.,  
 38., 43., 39., 45., 45., 35., 54., 48., 74., 59., 62.,  
 54., 47., 61., 51., 39., 51., 41., 33., 47., 37., 39.,  
 64., 43., 54., 59., 63., 55., 106., 73., 55., 86., 53.,  
 84., 74., 73., 81., 70., 77., 71., 64., 79., 59., 45.,  
 73., 50., 55., 49., 55., 68., 65., 62., 72., 110., 78.,  
 54., 56., 66., 55., 79., 53., 57., 55., 60., 59., 43.,  
 83., 61., 54., 45., 59., 52., 61., 51., 55., 65., 59.,  
 68., 144., 51., 75., 72., 65., 75., 74., 64., 68., 79.,  
 57., 52., 38., 111., 76., 64., 67., 77., 63., 94., 75.,  
 83., 76., 96., 74., 145., 73., 74., 92., 88., 55., 61.,  
 61., 79., 68., 52., 64., 53., 91., 50., 57., 63., 68.,  
 52., 69., 59., 84., 74., 65., 70., 114., 44., 56., 63.,  
 70., 69., 57., 51., 64., 52., 37., 49., 40., 59., 41.,  
 35., 32., 52., 47., 45., 34., 47., 51., 41., 37., 51.,  
 53., 50., 51., 44., 49., 66., 44., 55., 50., 49., 46.,  
 38., 107., 56., 50., 48., 48., 52., 60., 51., 44., 52.,  
 40., 41., 49., 44., 43., 53., 49., 27., 51., 39., 43.,  
 30., 47., 37., 22., 50., 33., 36., 42., 43., 35., 20.,  
 34., 40., 29., 29., 37., 44., 41., 39., 38., 40., 40.,  
 39., 35., 30., 43., 34., 34., 65., 21., 33., 25., 29.,  
 38., 22., 23., 26., 30., 20., 22., 24., 36., 22., 25.,  
 28., 26., 26., 21., 24., 26., 16., 15., 9., 31., 12.,  
 17., 19., 19., 18., 17., 18., 14., 15., 19., 11., 22.,  
 14., 18., 21., 23., 15., 9., 24., 16., 17., 18., 23.,  
 20., 28., 12., 12., 21., 11., 22., 17., 22., 19., 22.,  
 19., 25., 21., 15., 14., 20., 25., 22., 20., 18., 22.,  
 22., 16., 13., 22., 75., 14., 19., 19., 19., 15., 22.,  
 13., 18., 16., 21., 16., 19., 24., 11., 13., 16., 17.,  
 13., 11., 15., 4., 18., 9., 8., 26., 8., 14., 6.,  
 8., 12., 12., 11., 10., 12., 14., 5., 13., 16., 7.,  
 7., 11., 10., 12., 14., 15., 9., 11., 10., 10., 22.,  
 4., 2., 12., 2., 10., 12., 11., 2., 4., 14., 9.,  
 10., 10., 5., 13., 5., 13., 8., 13., 7., 9., 3.,  
 8., 8., 12., 5., 5., 5., 5., 2., 11., 7., 6.,  
 9., 11., 7., 7., 7., 9., 7., 7., 6., 7., 8.,  
 9., 6., 7., 5., 2., 28., 6., 5., 7., 6., 5.,  
 3., 6., 10., 6., 6., 1., 6., 4., 4., 3., 3.,  
 6., 5., 3., 5., 3., 4., 6., 3., 10., 1., 2.,  
 8., 4., 1., 3., 1., 3., 10., 7., 2., 4., 4.,  
 3., 3., 4., 3., 4., 4., 2., 6., 2., 2., 5.,  
 810.]),  
 array([ 14999. , 15999.00412371, 16999.00824742, 17999.01237113,  
 18999.01649485, 19999.02061856, 20999.02474227, 21999.02886598,  
 22999.03298969, 23999.0371134 , 24999.04123711, 25999.04536082,  
 26999.04948454, 27999.05360825, 28999.05773196, 29999.06185567,  
 30999.06597938, 31999.07010309, 32999.0742268 , 33999.07835052,  
 34999.08247423, 35999.08659794, 36999.09072165, 37999.09484536,  
 38999.09896907, 39999.10309278, 40999.10721649, 41999.11134021,  
 42999.11546392, 43999.11958763, 44999.12371134, 45999.12783505,  
 46999.13195876, 47999.13608247, 48999.14020619, 49999.1443299 ,  
 50999.14845361, 51999.15257732, 52999.15670103, 53999.16082474,  
 54999.16494845, 55999.16907216, 56999.17319588, 57999.17731959,  
 58999.1814433 , 59999.18556701, 60999.18969072, 61999.19381443,  
 62999.19793814, 63999.20206186, 64999.20618557, 65999.21030928,  
 66999.21443299, 67999.2185567 , 68999.22268041, 69999.22680412,  
 70999.23092784, 71999.23505155, 72999.23917526, 73999.24329897,  
 74999.24742268, 75999.25154639, 76999.2556701 , 77999.25979381,  
 78999.26391753, 79999.26804124, 80999.27216495, 81999.27628866,  
 82999.28041237, 83999.28453608, 84999.28865979, 85999.29278351,  
 86999.29690722, 87999.30103093, 88999.30515464, 89999.30927835,  
 90999.31340206, 91999.31752577, 92999.32164948, 93999.3257732 ,  
 94999.32989691, 95999.33402062, 96999.33814433, 97999.34226804,  
 98999.34639175, 99999.35051546, 100999.35463918, 101999.35876289,  
 102999.3628866 , 103999.36701031, 104999.37113402, 105999.37525773,  
 106999.37938144, 107999.38350515, 108999.38762887, 109999.39175258,  
 110999.39587629, 111999.4 , 112999.40412371, 113999.40824742,  
 114999.41237113, 115999.41649485, 116999.42061856, 117999.42474227,  
 118999.42886598, 119999.43298969, 120999.4371134 , 121999.44123711,  
 122999.44536082, 123999.44948454, 124999.45360825, 125999.45773196,  
 126999.46185567, 127999.46597938, 128999.47010309, 129999.4742268 ,  
 130999.47835052, 131999.48247423, 132999.48659794, 133999.49072165,  
 134999.49484536, 135999.49896907, 136999.50309278, 137999.50721649,  
 138999.51134021, 139999.51546392, 140999.51958763, 141999.52371134,  
 142999.52783505, 143999.53195876, 144999.53608247, 145999.54020619,  
 146999.5443299 , 147999.54845361, 148999.55257732, 149999.55670103,  
 150999.56082474, 151999.56494845, 152999.56907216, 153999.57319588,  
 154999.57731959, 155999.5814433 , 156999.58556701, 157999.58969072,  
 158999.59381443, 159999.59793814, 160999.60206186, 161999.60618557,  
 162999.61030928, 163999.61443299, 164999.6185567 , 165999.62268041,  
 166999.62680412, 167999.63092784, 168999.63505155, 169999.63917526,  
 170999.64329897, 171999.64742268, 172999.65154639, 173999.6556701 ,  
 174999.65979381, 175999.66391753, 176999.66804124, 177999.67216495,  
 178999.67628866, 179999.68041237, 180999.68453608, 181999.68865979,  
 182999.69278351, 183999.69690722, 184999.70103093, 185999.70515464,  
 186999.70927835, 187999.71340206, 188999.71752577, 189999.72164948,  
 190999.7257732 , 191999.72989691, 192999.73402062, 193999.73814433,  
 194999.74226804, 195999.74639175, 196999.75051546, 197999.75463918,  
 198999.75876289, 199999.7628866 , 200999.76701031, 201999.77113402,  
 202999.77525773, 203999.77938144, 204999.78350515, 205999.78762887,  
 206999.79175258, 207999.79587629, 208999.8 , 209999.80412371,  
 210999.80824742, 211999.81237113, 212999.81649485, 213999.82061856,  
 214999.82474227, 215999.82886598, 216999.83298969, 217999.8371134 ,  
 218999.84123711, 219999.84536082, 220999.84948454, 221999.85360825,  
 222999.85773196, 223999.86185567, 224999.86597938, 225999.87010309,  
 226999.8742268 , 227999.87835052, 228999.88247423, 229999.88659794,  
 230999.89072165, 231999.89484536, 232999.89896907, 233999.90309278,  
 234999.90721649, 235999.91134021, 236999.91546392, 237999.91958763,  
 238999.92371134, 239999.92783505, 240999.93195876, 241999.93608247,  
 242999.94020619, 243999.9443299 , 244999.94845361, 245999.95257732,  
 246999.95670103, 247999.96082474, 248999.96494845, 249999.96907216,  
 250999.97319588, 251999.97731959, 252999.9814433 , 253999.98556701,  
 254999.98969072, 255999.99381443, 256999.99793814, 258000.00206186,  
 259000.00618557, 260000.01030928, 261000.01443299, 262000.0185567 ,  
 263000.02268041, 264000.02680412, 265000.03092784, 266000.03505155,  
 267000.03917526, 268000.04329897, 269000.04742268, 270000.05154639,  
 271000.0556701 , 272000.05979381, 273000.06391753, 274000.06804124,  
 275000.07216495, 276000.07628866, 277000.08041237, 278000.08453608,  
 279000.08865979, 280000.09278351, 281000.09690722, 282000.10103093,  
 283000.10515464, 284000.10927835, 285000.11340206, 286000.11752577,  
 287000.12164948, 288000.1257732 , 289000.12989691, 290000.13402062,  
 291000.13814433, 292000.14226804, 293000.14639175, 294000.15051546,  
 295000.15463918, 296000.15876289, 297000.1628866 , 298000.16701031,  
 299000.17113402, 300000.17525773, 301000.17938144, 302000.18350515,  
 303000.18762887, 304000.19175258, 305000.19587629, 306000.2 ,  
 307000.20412371, 308000.20824742, 309000.21237113, 310000.21649485,  
 311000.22061856, 312000.22474227, 313000.22886598, 314000.23298969,  
 315000.2371134 , 316000.24123711, 317000.24536082, 318000.24948454,  
 319000.25360825, 320000.25773196, 321000.26185567, 322000.26597938,  
 323000.27010309, 324000.2742268 , 325000.27835052, 326000.28247423,  
 327000.28659794, 328000.29072165, 329000.29484536, 330000.29896907,  
 331000.30309278, 332000.30721649, 333000.31134021, 334000.31546392,  
 335000.31958763, 336000.32371134, 337000.32783505, 338000.33195876,  
 339000.33608247, 340000.34020619, 341000.3443299 , 342000.34845361,  
 343000.35257732, 344000.35670103, 345000.36082474, 346000.36494845,  
 347000.36907216, 348000.37319588, 349000.37731959, 350000.3814433 ,  
 351000.38556701, 352000.38969072, 353000.39381443, 354000.39793814,  
 355000.40206186, 356000.40618557, 357000.41030928, 358000.41443299,  
 359000.4185567 , 360000.42268041, 361000.42680412, 362000.43092784,  
 363000.43505155, 364000.43917526, 365000.44329897, 366000.44742268,  
 367000.45154639, 368000.4556701 , 369000.45979381, 370000.46391753,  
 371000.46804124, 372000.47216495, 373000.47628866, 374000.48041237,  
 375000.48453608, 376000.48865979, 377000.49278351, 378000.49690722,  
 379000.50103093, 380000.50515464, 381000.50927835, 382000.51340206,  
 383000.51752577, 384000.52164948, 385000.5257732 , 386000.52989691,  
 387000.53402062, 388000.53814433, 389000.54226804, 390000.54639175,  
 391000.55051546, 392000.55463918, 393000.55876289, 394000.5628866 ,  
 395000.56701031, 396000.57113402, 397000.57525773, 398000.57938144,  
 399000.58350515, 400000.58762887, 401000.59175258, 402000.59587629,  
 403000.6 , 404000.60412371, 405000.60824742, 406000.61237113,  
 407000.61649485, 408000.62061856, 409000.62474227, 410000.62886598,  
 411000.63298969, 412000.6371134 , 413000.64123711, 414000.64536082,  
 415000.64948454, 416000.65360825, 417000.65773196, 418000.66185567,  
 419000.66597938, 420000.67010309, 421000.6742268 , 422000.67835052,  
 423000.68247423, 424000.68659794, 425000.69072165, 426000.69484536,  
 427000.69896907, 428000.70309278, 429000.70721649, 430000.71134021,  
 431000.71546392, 432000.71958763, 433000.72371134, 434000.72783505,  
 435000.73195876, 436000.73608247, 437000.74020619, 438000.7443299 ,  
 439000.74845361, 440000.75257732, 441000.75670103, 442000.76082474,  
 443000.76494845, 444000.76907216, 445000.77319588, 446000.77731959,  
 447000.7814433 , 448000.78556701, 449000.78969072, 450000.79381443,  
 451000.79793814, 452000.80206186, 453000.80618557, 454000.81030928,  
 455000.81443299, 456000.8185567 , 457000.82268041, 458000.82680412,  
 459000.83092784, 460000.83505155, 461000.83917526, 462000.84329897,  
 463000.84742268, 464000.85154639, 465000.8556701 , 466000.85979381,  
 467000.86391753, 468000.86804124, 469000.87216495, 470000.87628866,  
 471000.88041237, 472000.88453608, 473000.88865979, 474000.89278351,  
 475000.89690722, 476000.90103093, 477000.90515464, 478000.90927835,  
 479000.91340206, 480000.91752577, 481000.92164948, 482000.9257732 ,  
 483000.92989691, 484000.93402062, 485000.93814433, 486000.94226804,  
 487000.94639175, 488000.95051546, 489000.95463918, 490000.95876289,  
 491000.9628866 , 492000.96701031, 493000.97113402, 494000.97525773,  
 495000.97938144, 496000.98350515, 497000.98762887, 498000.99175258,  
 499000.99587629, 500001. ]),  
 <BarContainer object of 485 artists>)



Check docs on how to detect picks :

* Finding peaks in the histograms of the variables : <https://www.kaggle.com/simongrest/finding-peaks-in-the-histograms-of-the-variables>
* Peak-finding algorithm for Python/SciPy : <https://stackoverflow.com/questions/1713335/peak-finding-algorithm-for-python-scipy>

## Experimenting with Attribute Combinations

* We identified a few data quirks that we may want to clean up before feeding the data to a Machine Learning algorithm,
* We found interesting correlations between attributes, in particular with the target attribute.
* We also noticed that some attributes have a tail-heavy distribution, so you may want to transform them (e.g., by computing their logarithm).
* One last thing we may want to do before preparing the data for Machine Learning algorithms is to try out various attribute combinations : For example, the total number of rooms in a district is not very useful if we don’t know how many households there are. What we really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. Let’s create these new attributes:

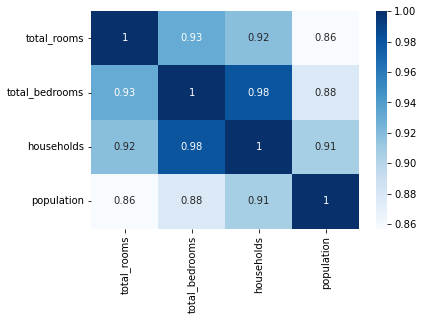
# We can see that some attributes are very linked to each others  
housing[['total\_rooms','total\_bedrooms','households','population' ]].corr()

total\_rooms total\_bedrooms households population  
total\_rooms 1.000000 0.930380 0.918484 0.857126  
total\_bedrooms 0.930380 1.000000 0.979728 0.877747  
households 0.918484 0.979728 1.000000 0.907222  
population 0.857126 0.877747 0.907222 1.000000

To highlight the matrix correlation, we can use heatmap from seaborn:

import seaborn as sns  
cor= housing[['total\_rooms','total\_bedrooms','households','population' ]].corr()  
sns.heatmap(cor, cmap='Blues', annot= True)

<AxesSubplot:>



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"] # nbre of person per houshold

housing["bedrooms\_per\_room"].describe()

count 20433.000000  
mean 0.213039  
std 0.057983  
min 0.100000  
25% 0.175427  
50% 0.203162  
75% 0.239821  
max 1.000000  
Name: bedrooms\_per\_room, dtype: float64

# on average, we have 21 bedrooms for 100 rooms  
from fractions import Fraction  
z = Fraction(0.21).limit\_denominator()  
z

Fraction(21, 100)

corr\_matrix = housing.corr()  
corr\_matrix['median\_house\_value'].sort\_values(ascending = False)

median\_house\_value 1.000000  
median\_income 0.688075  
rooms\_per\_household 0.151948  
total\_rooms 0.134153  
housing\_median\_age 0.105623  
households 0.065843  
total\_bedrooms 0.049686  
population\_per\_household -0.023737  
population -0.024650  
longitude -0.045967  
latitude -0.144160  
bedrooms\_per\_room -0.255880  
Name: median\_house\_value, dtype: float64

The new bedrooms\_per\_room attribute is much more correlated (0.25)with the median house value than the total number of rooms(0.13) or bedrooms (0.04) :

* Apparently houses with a lower bedroom/room ratio tend to be more expensive.
* The number of rooms per household is also more informative than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

# Prepare the Data for Machine Learning Algorithms

* let’s revert to a clean training set (by copying strat\_train\_set once again).
* Let’s also separate the predictors and the labels, since we don’t necessarily want to apply the same transformations to the predictors and the target values.

# drop() creates a copy of the data and does not affect strat\_train\_set  
housing = strat\_train\_set.drop("median\_house\_value", axis=1)   
housing\_labels = strat\_train\_set["median\_house\_value"].copy()

housing = strat\_train\_set.drop('median\_house\_value', axis = 1)  
housing\_labels = strat\_train\_set['median\_house\_value'].copy()

## Data Cleaning

Formissing values (like for total\_bedrooms), we have three options:

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

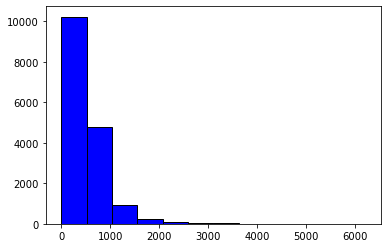
We can accomplish these easily using DataFrame’s dropna(), drop(), and fillna()

housing['total\_bedrooms'].describe()

count 16354.000000  
mean 534.914639  
std 412.665649  
min 2.000000  
25% 295.000000  
50% 433.000000  
75% 644.000000  
max 6210.000000  
Name: total\_bedrooms, dtype: float64

# in bar chart, NanN's are filled with 0's  
plt.hist(housing['total\_bedrooms']  
 , bins = int ((6210.000000 - 2.000000 )/500)  
 , color = 'blue'  
 , edgecolor = 'black'   
 )

(array([1.0221e+04, 4.7670e+03, 9.1400e+02, 2.6100e+02, 9.9000e+01,  
 5.1000e+01, 1.7000e+01, 1.1000e+01, 7.0000e+00, 3.0000e+00,  
 2.0000e+00, 1.0000e+00]),  
 array([2.00000000e+00, 5.19333333e+02, 1.03666667e+03, 1.55400000e+03,  
 2.07133333e+03, 2.58866667e+03, 3.10600000e+03, 3.62333333e+03,  
 4.14066667e+03, 4.65800000e+03, 5.17533333e+03, 5.69266667e+03,  
 6.21000000e+03]),  
 <BarContainer object of 12 artists>)



# to count the number of Nan's  
housing['total\_bedrooms'].isna().sum()

158

housing.isna().sum()

longitude 0  
latitude 0  
housing\_median\_age 0  
total\_rooms 0  
total\_bedrooms 158  
population 0  
households 0  
median\_income 0  
ocean\_proximity 0  
income\_cat 0  
dtype: int64

# option 1 : Get rid of the corresponding districts  
housing.dropna(subset = ['total\_bedrooms']) # drop from 16512 to 16354 using len()  
  
# option 2 : Get rid of the whole attribute  
housing.drop('total\_bedrooms', axis = 1)  
  
# option 3 : Set the values to some value (zero, the mean, the median, etc.)  
median = housing['total\_bedrooms'].median()  
housing['total\_bedrooms'].fillna(median, inplace = True)

longitude latitude housing\_median\_age total\_rooms population \  
12655 -121.46 38.52 29.0 3873.0 2237.0   
15502 -117.23 33.09 7.0 5320.0 2015.0   
2908 -119.04 35.37 44.0 1618.0 667.0   
14053 -117.13 32.75 24.0 1877.0 898.0   
20496 -118.70 34.28 27.0 3536.0 1837.0   
... ... ... ... ... ...   
15174 -117.07 33.03 14.0 6665.0 2026.0   
12661 -121.42 38.51 15.0 7901.0 4769.0   
19263 -122.72 38.44 48.0 707.0 458.0   
19140 -122.70 38.31 14.0 3155.0 1208.0   
19773 -122.14 39.97 27.0 1079.0 625.0   
  
 households median\_income ocean\_proximity income\_cat   
12655 706.0 2.1736 INLAND 2   
15502 768.0 6.3373 NEAR OCEAN 5   
2908 300.0 2.8750 INLAND 2   
14053 483.0 2.2264 NEAR OCEAN 2   
20496 580.0 4.4964 <1H OCEAN 3   
... ... ... ... ...   
15174 1001.0 5.0900 <1H OCEAN 4   
12661 1418.0 2.8139 INLAND 2   
19263 172.0 3.1797 <1H OCEAN 3   
19140 501.0 4.1964 <1H OCEAN 3   
19773 197.0 3.1319 INLAND 3   
  
[16512 rows x 9 columns]

For 'option' 3 : fill missings with some value, the median for example, we should :

* Compute the median value on the training and use it to fill the missing values in the training set.
* Save the median value that you have computed.
* Using later for to replace missing values in the test set when we want to evaluate our system
* Using it once the system goes live to replace missing values in new data.

Scikit-Learn provides a handy class to take care of missing values: SimpleImputer.

from sklearn.impute import SimpleImputer

First, you need to create a SimpleImputer instance, specifying that we want to replace each numeric attribute’s missing values with the median of that attribute :

imputer = SimpleImputer(strategy = 'median')

housing.dtypes

longitude float64  
latitude float64  
housing\_median\_age float64  
total\_rooms float64  
total\_bedrooms float64  
population float64  
households float64  
median\_income float64  
ocean\_proximity object  
dtype: object

# We drop the categorical variables since the median can only be computed on numerical attributes  
housing\_num = housing.drop(['ocean\_proximity'], axis = 1)

Now you can fit the imputer instance to the training data using the fit() method

imputer.fit(housing\_num)

SimpleImputer(strategy='median')

The imputer has simply computed the median of each attribute and stored the result in its statistics\_ instance variable. We apply the imputer to all the numerical attributes :

imputer.statistics\_

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

housing\_num.columns

Index(['longitude', 'latitude', 'housing\_median\_age', 'total\_rooms',  
 'total\_bedrooms', 'population', 'households', 'median\_income'],  
 dtype='object')

# equivalant to imputer.statistics\_ : we have the median for each numeric variable  
housing\_num.median().values

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

Now we can use this trained imputer to transform the training set by replacing missing values with the learned medians:

# The result is a plain NumPy array containing the transformed features.   
X = imputer.transform(housing\_num)  
  
  
#If you want to put it back into a pandas DataFrame, it’s simple:  
housing\_tr = pd.DataFrame(X, columns=housing\_num.columns,  
 index=housing\_num.index)

Alternative method to fit() and transform() method is using directy fit\_transform() method

X = imputer.fit\_transform(housing\_num)  
  
housing\_tr = pd.DataFrame( X, columns = housing\_num.columns  
 , index = housing\_num.index)  
housing\_tr

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
12655 -121.46 38.52 29.0 3873.0 797.0   
15502 -117.23 33.09 7.0 5320.0 855.0   
2908 -119.04 35.37 44.0 1618.0 310.0   
14053 -117.13 32.75 24.0 1877.0 519.0   
20496 -118.70 34.28 27.0 3536.0 646.0   
... ... ... ... ... ...   
15174 -117.07 33.03 14.0 6665.0 1231.0   
12661 -121.42 38.51 15.0 7901.0 1422.0   
19263 -122.72 38.44 48.0 707.0 166.0   
19140 -122.70 38.31 14.0 3155.0 580.0   
19773 -122.14 39.97 27.0 1079.0 222.0   
  
 population households median\_income   
12655 2237.0 706.0 2.1736   
15502 2015.0 768.0 6.3373   
2908 667.0 300.0 2.8750   
14053 898.0 483.0 2.2264   
20496 1837.0 580.0 4.4964   
... ... ... ...   
15174 2026.0 1001.0 5.0900   
12661 4769.0 1418.0 2.8139   
19263 458.0 172.0 3.1797   
19140 1208.0 501.0 4.1964   
19773 625.0 197.0 3.1319   
  
[16512 rows x 8 columns]

## Handling Text and Categorical Attributes

So far we have only dealt with numerical attributes, but now let’s look at text attributes. In this dataset, there is just one: the ocean\_proximity attribute. Let’s look at its value for the first 10 instances:

housing\_cat = housing[['ocean\_proximity']]  
housing\_cat.head(10)  
housing\_cat.nunique()

ocean\_proximity 5  
dtype: int64

housing['ocean\_proximity'].unique()

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

It’s not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute. Most Machine Learning algorithms prefer to work with numbers, so let’s convert these categories from text to numbers. For this, we can use Scikit-Learn’s OrdinalEncoder class :

from sklearn.preprocessing import OrdinalEncoder

ordinal\_encoder = OrdinalEncoder()  
housing\_cat\_encoded = ordinal\_encoder.fit\_transform(housing\_cat) # categorical dataframe  
housing\_cat\_encoded[:10]

array([[1.],  
 [4.],  
 [1.],  
 [4.],  
 [0.],  
 [3.],  
 [0.],  
 [0.],  
 [0.],  
 [0.]])

housing\_cat\_encoded.shape

(16512, 1)

# we can get the list of categories using the categories\_ instance variable. It is a list containing a 1D array   
 #of categories for each categorical attribute  
   
ordinal\_encoder.categories\_

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

np.unique(housing\_cat\_encoded)

array([0., 1., 2., 3., 4.])

One issue with this representation is that ML algorithms ( OrdinaEncoder) will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad,” “average,” “good,” and “excellent”) : ordinal encoding for categorical variables that have a natural rank ordering but it is obviously not the case for the ocean\_proximity column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1).

To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is “<1H OCEAN” (and 0 otherwise), another attribute equal to 1 when the category is “INLAND” (and 0 otherwise), and so on. This is called one-hot encoding. The new attributes are sometimes called dummy attributes. Scikit-Learn provides a OneHotEncoder class to convert categorical values into one-hot vectors

See this blogpost : <https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/>

from sklearn.preprocessing import OneHotEncoder  
cat\_encoder = OneHotEncoder()  
housing\_cat\_1hot = cat\_encoder.fit\_transform(housing\_cat)  
housing\_cat\_1hot

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

Notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After onehot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. we can use it mostly like a normal 2D array,but if we really want to convert it to a (dense) NumPy array, we call the toarray() method:

housing\_cat\_1hot.toarray()

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

housing\_cat.head(3)

ocean\_proximity  
12655 INLAND  
15502 NEAR OCEAN  
2908 INLAND

# to get the list of categories  
cat\_encoder.categories\_

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

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance.

If this happens, we may want to replace the categorical input with useful numerical features related to the categories: for example, we could replace the ocean\_proximity feature with the distance to the ocean (similarly, a country code could be replaced with the country’s population and GDP per capita). Alternatively, we could replace each category with a learnable, low-dimensional vector called an embedding.

Each category’s representation would be learned during training. This is an example of representation learning.

## Custom Transformers

retun back for more details ? see blogpost : <https://towardsdatascience.com/pipelines-custom-transformers-in-scikit-learn-the-step-by-step-guide-with-python-code-4a7d9b068156>

and this : <https://github.com/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb>

housing.values[: ,4:]

array([[797.0, 2237.0, 706.0, 2.1736, 'INLAND', 2],  
 [855.0, 2015.0, 768.0, 6.3373, 'NEAR OCEAN', 5],  
 [310.0, 667.0, 300.0, 2.875, 'INLAND', 2],  
 ...,  
 [166.0, 458.0, 172.0, 3.1797, '<1H OCEAN', 3],  
 [580.0, 1208.0, 501.0, 4.1964, '<1H OCEAN', 3],  
 [222.0, 625.0, 197.0, 3.1319, 'INLAND', 3]], dtype=object)

housing.head(3)

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \  
12655 -121.46 38.52 29.0 3873.0 797.0   
15502 -117.23 33.09 7.0 5320.0 855.0   
2908 -119.04 35.37 44.0 1618.0 310.0   
  
 population households median\_income ocean\_proximity income\_cat   
12655 2237.0 706.0 2.1736 INLAND 2   
15502 2015.0 768.0 6.3373 NEAR OCEAN 5   
2908 667.0 300.0 2.8750 INLAND 2

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): # no \*args or \*\*kargs  
 self.add\_bedrooms\_per\_room = add\_bedrooms\_per\_room  
   
 def fit(self, X, y=None):  
 return self # nothing else to do  
   
 def transform(self, X):  
 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)

housing\_extra\_attribs

array([[-121.46, 38.52, 29.0, ..., 2, 5.485835694050992,  
 3.168555240793201],  
 [-117.23, 33.09, 7.0, ..., 5, 6.927083333333333,  
 2.6236979166666665],  
 [-119.04, 35.37, 44.0, ..., 2, 5.3933333333333335,  
 2.223333333333333],  
 ...,  
 [-122.72, 38.44, 48.0, ..., 3, 4.1104651162790695,  
 2.6627906976744184],  
 [-122.7, 38.31, 14.0, ..., 3, 6.297405189620759,  
 2.411177644710579],  
 [-122.14, 39.97, 27.0, ..., 3, 5.477157360406092,  
 3.1725888324873095]], dtype=object)

## Feature Scaling

One of the most important transformations you need to apply to your data is feature scaling. With few exceptions, Machine Learning algorithms don’t perform well when the input numerical attributes have very different scales . This is the case for the housing data: total\_rooms ranges from about 6 to 39320, while median\_income only range from 0 to 15 :

Note that scaling the target values is generally not required.

housing\_num.describe().T

count mean std min 25% \  
longitude 16512.0 -119.575635 2.001828 -124.3500 -121.80000   
latitude 16512.0 35.639314 2.137963 32.5400 33.94000   
housing\_median\_age 16512.0 28.653404 12.574819 1.0000 18.00000   
total\_rooms 16512.0 2622.539789 2138.417080 6.0000 1443.00000   
total\_bedrooms 16512.0 533.939438 410.806260 2.0000 296.00000   
population 16512.0 1419.687379 1115.663036 3.0000 784.00000   
households 16512.0 497.011810 375.696156 2.0000 279.00000   
median\_income 16512.0 3.875884 1.904931 0.4999 2.56695   
  
 50% 75% max   
longitude -118.51000 -118.010000 -114.3100   
latitude 34.26000 37.720000 41.9500   
housing\_median\_age 29.00000 37.000000 52.0000   
total\_rooms 2119.00000 3141.000000 39320.0000   
total\_bedrooms 433.00000 641.000000 6210.0000   
population 1164.00000 1719.000000 35682.0000   
households 408.00000 602.000000 5358.0000   
median\_income 3.54155 4.745325 15.0001

There are two common ways to get all attributes to have the same scale:

* min-max scaling : ranging from 0 to 1. Scikit-Learn provides a transformer called MinMaxScaler for this. It has a feature\_range hyperparameter that lets you change the range if, for some reason, you don’t want 0–1 : <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>
* Standardization : returns has unit variance and unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1).However, standardization is much less affected by outliers. Scikit-Learn provides a transformer called StandardScaler for standardization : <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

# Transformation Pipelines

As we can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations.

Here is a small pipeline for the numerical attributes:

from sklearn.pipeline import Pipeline  
from sklearn.impute import SimpleImputer  
from sklearn.preprocessing import StandardScaler,MinMaxScaler  
# plus the class add attribure that we created

The Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps. All but the last estimator must be transformers (i.e., they must have a fit\_transform() method).

# This pipeline is for numeric pipeline, we call is num\_pipeline  
num\_pipeline = Pipeline([  
 ('imputer', SimpleImputer(strategy = 'median')),  
 #('attribs\_adder', CombinedAttributesAdder()),  
 ('std\_scaler', StandardScaler())  
   
   
])

housing\_num\_tr = num\_pipeline.fit\_transform(housing\_num)

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. In version 0.20, Scikit-Learn introduced the ColumnTransformer for this purpose, and the good news is that it works great with pandas DataFrames. Let’s use it to apply all the transformations to the housing data:

from sklearn.compose import ColumnTransformer

num\_attribs = list(housing\_num)  
cat\_attribs = ['ocean\_proximity']

---------------------------------------------------------------------------  
NameError Traceback (most recent call last)  
<ipython-input-2-344d101417c2> in <module>  
----> 1 num\_attribs = list(housing\_num)  
 2 cat\_attribs = ['ocean\_proximity']  
  
NameError: name 'housing\_num' is not defined

full\_pipeline = ColumnTransformer([  
 ('num', num\_pipeline, num\_attribs),  
 ('cat', OneHotEncoder(), cat\_attribs)  
]  
)

housing\_prepared = full\_pipeline.fit\_transform(housing)

housing\_prepared

array([[-0.94135046, 1.34743822, 0.02756357, ..., 0. ,  
 0. , 0. ],  
 [ 1.17178212, -1.19243966, -1.72201763, ..., 0. ,  
 0. , 1. ],  
 [ 0.26758118, -0.1259716 , 1.22045984, ..., 0. ,  
 0. , 0. ],  
 ...,  
 [-1.5707942 , 1.31001828, 1.53856552, ..., 0. ,  
 0. , 0. ],  
 [-1.56080303, 1.2492109 , -1.1653327 , ..., 0. ,  
 0. , 0. ],  
 [-1.28105026, 2.02567448, -0.13148926, ..., 0. ,  
 0. , 0. ]])

housing\_prepared[0].shape

(13,)

housing\_prepared.shape

(16512, 13)

to read carefully for later :

First we import the ColumnTransformer class, next we get the list of numerical column

names and the list of categorical column names, and then we construct a Colum nTransformer. The constructor requires a list of tuples, where each tuple contains a name,22 a transformer, and a list of names (or indices) of columns that the transformer should be applied to. In this example, we specify that the numerical columns should be transformed using the num\_pipeline that we defined earlier, and the categorical columns should be transformed using a OneHotEncoder. Finally, we apply this ColumnTransformer to the housing data: it applies each transformer to the appropriate columns and concatenates the outputs along the second axis (the transformers must return the same number of rows). Note that the OneHotEncoder returns a sparse matrix, while the num\_pipeline returns a dense matrix. When there is such a mix of sparse and dense matrices, the Colum nTransformer estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, sparse\_threshold=0.3). In this example, it returns a dense matrix. And that’s it! We have a preprocessing pipeline that takes the full housing data and applies the appropriate transformations to each column. Instead of using a transformer, you can specify the string "drop" if you want the columns to be dropped, or you can specify "pass through" if you want the columns to be left untouched. By default, the remaining columns (i.e., the ones that were not listed) will be dropped, but you can set the remainder hyperparameter to any transformer (or to "passthrough") if you want these columns to be handled differently. If you are using Scikit-Learn 0.19 or earlier, you can use a third-party library such as sklearn-pandas, or you can roll out your own custom transformer to get the same functionality as the ColumnTransformer. Alternatively, you can use the FeatureUnion class, which can apply different transformers and concatenate their outputs. But you cannot specify different columns for each transformer; they all apply to the whole data. It is possible to work around this limitation using a custom transformer for column selection (see the Jupyter notebook for an example).

see link : <https://github.com/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb>

# Select and Train a Model

* we framed the problem
* we got the data and explored it
* we sampled a training set and a test set
* we wrote transformation pipelines to clean up and prepare your data for Machine Learning algorithms automatically.

## Training and Evaluating on the Training Set

Let’s first train a Linear Regression model

housing\_prepared.shape

(16512, 13)

from sklearn.linear\_model import LinearRegression  
  
lin\_reg = LinearRegression()  
lin\_reg.fit(housing\_prepared, housing\_labels)

LinearRegression()

Let’s try it out on a few instances from the training set:

# let's try the full preprocessing pipeline on a few training instances  
some\_data = housing.iloc[:5]  
some\_labels = housing\_labels.iloc[:5]  
some\_data\_prepared = full\_pipeline.transform(some\_data)  
  
print("Predictions:", lin\_reg.predict(some\_data\_prepared))

Predictions: [ 88983.14806384 305351.35385026 153334.71183453 184302.55162102  
 246840.18988841]

print("Labels:", list(some\_labels))

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

some\_data\_prepared.shape

(5, 13)

* Some remarks :
  + If we encotered missing values are in the test set ? ( OneHotEncode() has a paramete : handle\_unknown = 'ignore')
  + Indexing and selection data : if we want to modifiy a certain column in the dataframe, we should not proceed in this way : df['ocean\_proxemity][0]= np.nan but rather copy the dataset first and then df.loc[0,'ocean\_proxemity']=np.nan. Read : <https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy>

More on sklearn pipelines : <https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html>

* The iloc indexer for Pandas Dataframe is used for integer-location based indexing / selection by position.
* The Pandas loc indexer can be used with DataFrames for two different use cases:
  + Selecting rows by label/index
  + Selecting rows with a boolean / conditional lookup

Read : <https://www.shanelynn.ie/pandas-iloc-loc-select-rows-and-columns-dataframe/>

Let’s measure this regression model’s RMSE on the whole training set using Scikit-Learn’s mean\_squared\_error() function :

from sklearn.metrics import mean\_squared\_error

#  
housing\_predictions = lin\_reg.predict(housing\_prepared)  
housing\_predictions[:4]

array([ 88983.14806384, 305351.35385026, 153334.71183453, 184302.55162102])

housing\_labels

12655 72100.0  
15502 279600.0  
2908 82700.0  
14053 112500.0  
20496 238300.0  
 ...   
15174 268500.0  
12661 90400.0  
19263 140400.0  
19140 258100.0  
19773 62700.0  
Name: median\_house\_value, Length: 16512, dtype: float64

#RMSE  
lin\_rmse = mean\_squared\_error(housing\_predictions,housing\_labels, squared= False)  
lin\_rmse

69050.56219504567

Most districts’ median\_housing\_values range between $120,000 and $265,000, so a typical prediction error of $69,050 is not very satisfying:

housing\_labels.describe()

count 16512.000000  
mean 207005.322372  
std 115701.297250  
min 14999.000000  
25% 119800.000000  
50% 179500.000000  
75% 263900.000000  
max 500001.000000  
Name: median\_house\_value, dtype: float64

This is an example of a model underfitting the training data : When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough. As we saw in the previous chapter, the main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model.

* Let’s train a DecisionTreeRegressor. This is a powerful model, capable of finding complex nonlinear relationships in the data

from sklearn.tree import DecisionTreeRegressor  
tree\_reg = DecisionTreeRegressor()  
tree\_reg.fit(housing\_prepared, housing\_labels)

DecisionTreeRegressor()

#Now that the model is trained, let’s evaluate it on the training set:  
housing\_predictions = tree\_reg.predict(housing\_prepared)  
tree\_rmse = mean\_squared\_error(housing\_labels, housing\_predictions, squared=False)  
tree\_rmse

0.0

As we saw earlier, we don’t want to touch the test set until we are ready to launch a model we are confident about, so we need to use part of the training set for training and part of it for model validation.

# Better Evaluation Using Cross-Validation

* One way to evaluate the Decision Tree model would be to use the train\_test\_split() function to split the training set into a smaller training set and a validation set, then train our models against the smaller training set and evaluate them against the validation set.
* A great alternative is to use Scikit-Learn’s K-fold cross-validation feature : The following code randomly splits the training set into 10 distinct subsets called folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

from sklearn.model\_selection import cross\_val\_score

# to get the 'scoring' options, use ssorted(sklearn.metrics.SCORERS.keys())  
  
scores = - cross\_val\_score(tree\_reg, housing\_prepared, housing\_labels, scoring='neg\_root\_mean\_squared\_error', cv=10)  
scores

array([71270.15951523, 68888.32011559, 64997.85188763, 69263.03318422,  
 68197.14503697, 68963.98885461, 73536.17215975, 69183.4936482 ,  
 66243.08004208, 71783.50940468])

* to get the 'scoring' options, use ssorted(sklearn.metrics.SCORERS.keys())
* 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 MSE (i.e., a negative value)

# display the results   
def display\_scores(scores):  
 print('Scores :',scores)  
 print('Mean :',scores.mean())  
 print('Standard deviation :',scores.std())

display\_scores(scores)

Scores : [71270.15951523 68888.32011559 64997.85188763 69263.03318422  
 68197.14503697 68963.98885461 73536.17215975 69183.4936482  
 66243.08004208 71783.50940468]  
Mean : 69232.67538489516  
Standard deviation : 2394.0765898258674

* Cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation). The Decision Tree has a score of approximately 69,232, generally ±2,394. We would not have this information if we just used one validation set.
* Let’s compute the same scores for the Linear Regression model just to be sure:

lin\_scores = - cross\_val\_score(lin\_reg,housing\_prepared, housing\_labels, scoring='neg\_root\_mean\_squared\_error', cv = 10)  
lin\_scores

array([72229.03469752, 65318.2240289 , 67706.39604745, 69368.53738998,  
 66767.61061621, 73003.75273869, 70522.24414582, 69440.77896541,  
 66930.32945876, 70756.31946074])

display\_scores(lin\_scores)

Scores : [72229.03469752 65318.2240289 67706.39604745 69368.53738998  
 66767.61061621 73003.75273869 70522.24414582 69440.77896541  
 66930.32945876 70756.31946074]  
Mean : 69204.32275494766  
Standard deviation : 2372.07079105592

The Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

Let’s try one last model now: the RandomForestRegressor : Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models is called Ensemble Learning, and it is often a great way to push ML algorithms even further.

from sklearn.ensemble import RandomForestRegressor  
forest\_reg = RandomForestRegressor()  
forest\_reg.fit(housing\_prepared, housing\_labels)

RandomForestRegressor()

forest\_reg\_scores = - cross\_val\_score(forest\_reg,housing\_prepared, housing\_labels, scoring='neg\_root\_mean\_squared\_error', cv = 10)  
forest\_reg\_scores

array([50311.08798022, 49043.69572163, 46081.95238283, 50467.56214907,  
 47657.84153211, 49419.96274189, 51772.42197545, 49030.57976501,  
 47498.17482111, 53167.33074077])

display\_scores(forest\_reg\_scores)

Scores : [50311.08798022 49043.69572163 46081.95238283 50467.56214907  
 47657.84153211 49419.96274189 51772.42197545 49030.57976501  
 47498.17482111 53167.33074077]  
Mean : 49445.06098101039  
Standard deviation : 1992.3842490271882

forest\_predictions = forest\_reg.predict(housing\_prepared)  
forst\_rmse = mean\_squared\_error(housing\_labels, forest\_predictions, squared=False)  
forst\_rmse

18266.74368085342

Note that the score on the training set(18,266) is still much lower than on the validation sets(49,445 +/-1992.38), meaning that the model is still overfitting the training set.

* We should save every model we experiment with so that er can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well. This will allow you to easily compare scores across model types, and compare the types of errors they make.
* We can easily save Scikit-Learn models by using Python’s pickle module or by using the joblib library, which is more efficient at serializing large NumPy arrays (you can install this library using pip):

pip install joblib

Requirement already satisfied: joblib in /Users/rmbp/opt/anaconda3/lib/python3.7/site-packages (1.1.0)  
Note: you may need to restart the kernel to use updated packages.

import joblib

# random forest :  
  
joblib.dump(forest\_reg,'rmd\_forest.pkl') # saving the model as pkl file and named 'rmd\_forest.pkl  
model\_reload = joblib.load('rmd\_forest.pkl') # loading the model  
rmd\_forest\_prediction = model\_reload.predict(housing\_prepared) # saving the predictions  
rmd\_forest\_rmse = mean\_squared\_error(rmd\_forest\_prediction, housing\_labels) # saving the rmse on the train test to check overfit  
rmd\_forest\_cross\_validation = -cross\_val\_score(model\_reload, housing\_prepared, housing\_labels,  
 scoring='neg\_root\_mean\_squared\_error', cv = 10) # rmse on validation to check overfit  
display\_scores(rmd\_forest\_cross\_validation) # cross validation score

Scores : [50992.51555592 49288.84220573 46237.11091931 50248.85062075  
 47806.3116179 49272.797347 51801.0468531 48800.83283468  
 47540.31917616 53091.63650718]  
Mean : 49508.0263637728  
Standard deviation : 1972.8867918884973

# Fine-Tune Our Model

## Grid Search

Using Scikit-Learn’s GridSearchCV, All we need to do is tell it which hyperparameters we want it to experiment with and what valuesto try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values :

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\_root\_mean\_squared\_error',  
 return\_train\_score = True)  
grid\_search.fit(housing\_prepared, housing\_labels)

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\_root\_mean\_squared\_error')

?GridSearchCV

* This param\_grid tells Scikit-Learn to first evaluate all 3 × 4 = 12 combinations of n\_estimators and max\_features hyperparameter values specified in the first dict. -Then try all 2 × 3 = 6 combinations of hyperparameter values in the second dict, but this time with the bootstrap hyperparameter set to False instead of True
* The grid search will explore 12 + 6 = 18 combinations of `RandomForestRegressor hyperparameter values``
* And it will train each model 5 times (cv=5).In other words, all in all, there will be 18 × 5 = 90 rounds of training.

We can get the best combination of parameters like this :

grid\_search.best\_params\_  
  
#we should probably try searching again with higher values; the score may continue to improve.

{'max\_features': 8, 'n\_estimators': 30}

grid\_search.best\_estimator\_

RandomForestRegressor(max\_features=8, n\_estimators=30)

cvres = grid\_search.cv\_results\_  
for mean\_score, params in zip(cvres["mean\_test\_score"], cvres["params"]):  
 print(-mean\_score, params)

64530.85647414619 {'max\_features': 2, 'n\_estimators': 3}  
55067.77832337284 {'max\_features': 2, 'n\_estimators': 10}  
52781.7167175866 {'max\_features': 2, 'n\_estimators': 30}  
60327.066875895776 {'max\_features': 4, 'n\_estimators': 3}  
52586.95798629394 {'max\_features': 4, 'n\_estimators': 10}  
50346.63948997191 {'max\_features': 4, 'n\_estimators': 30}  
58398.87657548992 {'max\_features': 6, 'n\_estimators': 3}  
52075.368300249116 {'max\_features': 6, 'n\_estimators': 10}  
50093.621910952315 {'max\_features': 6, 'n\_estimators': 30}  
58628.430409285626 {'max\_features': 8, 'n\_estimators': 3}  
51936.797178963665 {'max\_features': 8, 'n\_estimators': 10}  
50089.501357132904 {'max\_features': 8, 'n\_estimators': 30}  
62303.627420601595 {'bootstrap': False, 'max\_features': 2, 'n\_estimators': 3}  
54183.89357722118 {'bootstrap': False, 'max\_features': 2, 'n\_estimators': 10}  
60004.47703668058 {'bootstrap': False, 'max\_features': 3, 'n\_estimators': 3}  
52603.81684475204 {'bootstrap': False, 'max\_features': 3, 'n\_estimators': 10}  
58002.985249363366 {'bootstrap': False, 'max\_features': 4, 'n\_estimators': 3}  
52121.979634950054 {'bootstrap': False, 'max\_features': 4, 'n\_estimators': 10}

Rq: Don’t forget that we can treat some of the data preparation steps as hyperparameters. The grid search will automatically find out whether or not to add a feature you were not sure about. Ex : using the add\_bedrooms\_per\_room hyperparameter of your CombinedAttributesAdder transformer).