Machine Learning for Software Analysis (MLSA)

Fabio Pinelli

fabio.pinelli@imtlucca.it (mailto:fabio.pinelli@imtlucca.it)
IMT School for Advanced Studies Lucca
2024/2025
October, 10 2024

This notebook

- How to organize a ML project and the steps you need to perform
- Regression
- Classification

... and some scikit-learn functionalities to authomatize the ML process.

Taxonomy of ML ¶

- Roughly speaking there are two main **categories** of ML tasks:

    1. **Supervised Learning**: Given a set of **labeled** examples, **predict** the labels of *new and unseen* examples
    2. **Unsupervised Learning**: Given a set of examples, **find structure** in the data (e.g., *clusters*, *subspaces*, *manifolds*)

Supervised learning

The main difference between **Regression** and **Classification** algorithms is:

- **Regression** algorithms are used to predict the *continuous values* such as price, salary, age, etc. and
- **Classification** algorithms are used to predict/classify the *discrete values* such as Male or Female, True or False, Spam or Not Spam, etc.

Check list for a Machine Learning project

1. Understand the problem.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
    - split on training and test
    - handling missing values
    - categorical values
    - scaling
    - etc.
5. Compare different models, choose the best and train it.
6. Fine-tune your model.
7. Present your solution.
    - stakeholder
    - paper
    - supervisor
    - etc.

Scikit API with more details

- Consinstency: All objects share a consistent and simple interface
  - Estimators:
    - Any object that can estimate some parameters based on a dataset is called an *estimator*
    - It uses the fit() and it takes only the dataset as parameters, the others are considered hyperparameters.
  - Transformers: Some estimators can transform the dataset.
    - transform() method with the dataset to transform as a parameter.
    - fit_transform() to perform both operation at once (more quick)
  - Predictors: estimators that make predictions
    - predict() method that takes a dataset of new instances and returns a dataset of corresponding predictions.
    - score() to measure the quality of the predictions
- Inspection --> two types of access to relevant information
  - All the estimator's **hyperparameters** are accessible directly via **public instance variables**
  - all the estimator's **learned** parameters are also accessible via public instance variables with an **underscore suffix**
- Nonproliferation of classes
  - Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes
- Composition: Create pipelines
- Sensible defaults: scikit provides reasonable default values for most parameters, making it easy to quickly create a working baseline system

Do you have some example of the different attribute types?

What about the object `kmeans` or `dbscan`

We start now to investigate a Regression problem, we will use some example to explore the data and understand better the problem. Then some concepts will be introduced to improve the quality of our machine learning pipeline

Understand the problem.

A regression problem

We load data of a dataset that contains the median house prices in California. So the prices can assume *continous* values, therefore we need to apply *Regression* methods/algorithms.

It is a quite old dataset, however it presents the basic properties needed to explain the different tasks that WE should take into account before delivering any results.

In order to understand the problem, we need to clarify some aspects that will influence the rest of the pipeline/project:

- **[Supervised/Unsupervised]:** Since we do have *labels* (prices) we can use a ***Supervised*** method.
- **[Classification/Regression]:** can assume *continous* values, therefore we need to apply ***Regression*** methods/algorithms
- **[Evalutation]:** How to evaluate the performance of our models, defining a measure of error/accuracy

Error

Since we are in regression problem there are some options out there:

- **RMSE** Root Mean Square Error
  - Also called *Euclidean Norm* or l_2 *norm* and indicated with ||.||_2
- **MAE** Mean Absolute Error
  - Also called Manhattan norm because it measures the distance between two points in a city if you can only travel along orthoganal city blocks. It is also called l_1 *norm* and indicated with ||.||_1
- **[Considerations]**
  - Generally, with a larger K in l_k The higher the norm index, the more it focuses on large values and neglects small ones. This is why the **RMSE is more sensitive to outliers** than the MAE.
  - when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

In [ ]:
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import pandas as pd
import os



# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

import seaborn as sns


# Where to save the figures
PROJECT_ROOT_DIR = "."
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images")
os.makedirs(IMAGES_PATH, exist_ok=True)

from google.colab import drive
drive.mount('/content/drive')

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)
```

```
Mounted at /content/drive
```

In [ ]:
```
housing = pd.read_csv('https://raw.githubusercontent.com/fpinell/mlsa/refs/heads/main/AA20252026/data/housing/housing.csv')
```

In [ ]:
```
# housing = pd.read_csv('/content/drive/Shareddrives/phd_hands_on/data/housing/housing.csv')
```

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

Out[ ]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	c
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	

In [ ]:
```
print('N. of rows: {}'.format(housing.shape[0]))
print('N. of columns: {}'.format(housing.shape[1]))
print('The columns contained in our first dataframe are:\n{}'.format( ',\n'.join('{}. {}'.format(i,c) for i,c
in enumerate(housing.columns)) ))
```

```
N. of rows: 20640
N. of columns: 10
The columns contained in our first dataframe are:
0. longitude,
1. latitude,
2. housing_median_age,
3. total_rooms,
4. total_bedrooms,
5. population,
6. households,
7. median_income,
8. median_house_value,
9. ocean_proximity
```

In [ ]:
```
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
```

```
<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
```

Two columns present some peculiarities?

- total_bedrooms
- ocean_proximity

Do you know these peculiarities?

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

Out[ ]:

ocean_proximity	count
<1H OCEAN	9136
INLAND	6551
NEAR OCEAN	2658
NEAR BAY	2290
ISLAND	5

dtype: int64

In [ ]:
```
housing.shape
```

Out[ ]:
```
(20640, 10)
```

In [ ]:
```
housing = housing[housing["ocean_proximity"]!='ISLAND'].copy()
```

In [ ]:
```
housing.reset_index(inplace=True,drop=True)
```

In [ ]:
```
housing.describe(include='all')
```

Out[ ]:

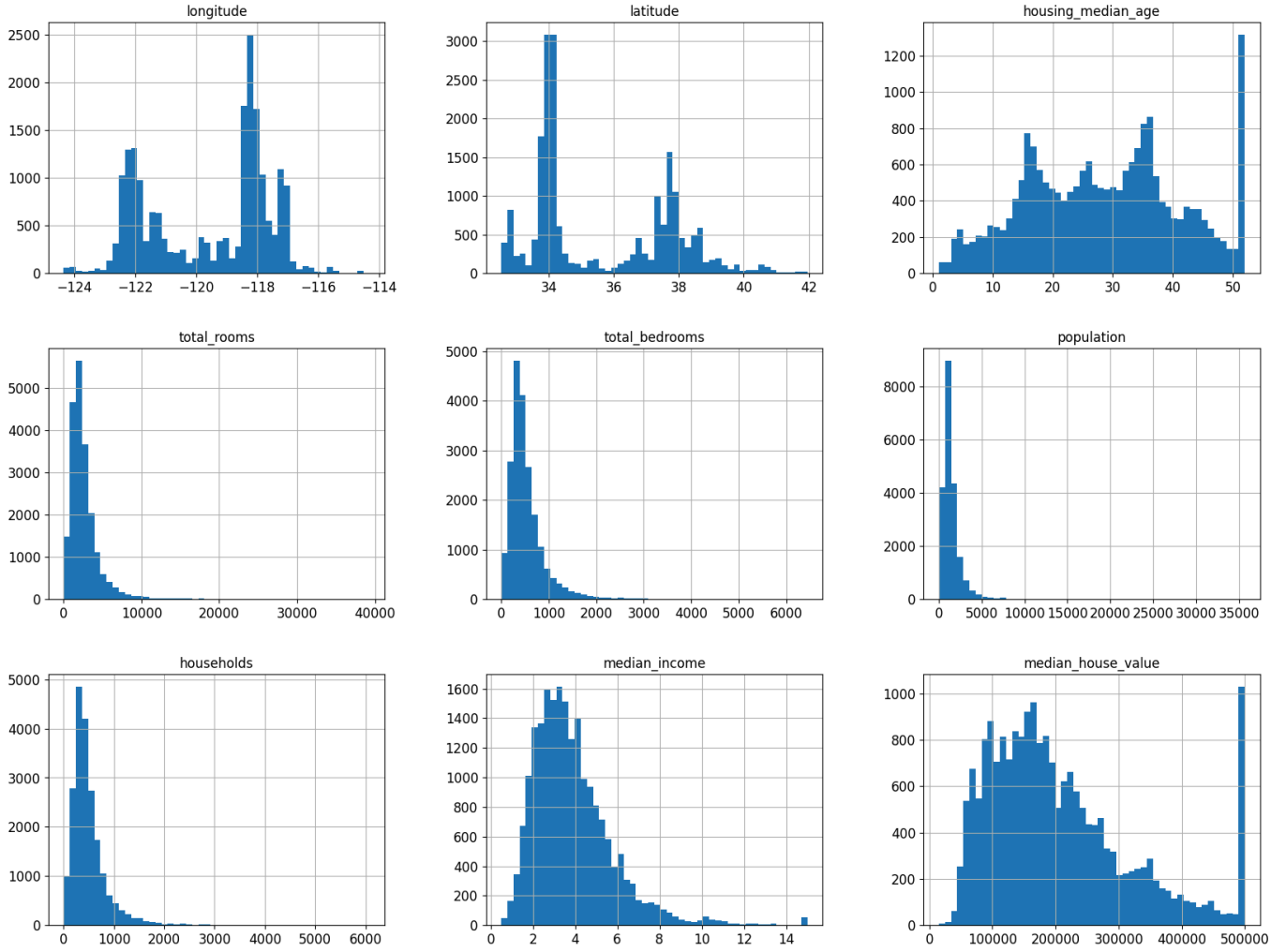
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	me
count	20635.000000	20635.000000	20635.000000	20635.000000	20428.000000	20635.000000	20635.000000	20635.000000	
unique	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
top	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
freq	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
mean	-119.569999	35.632412	28.636152	2636.020208	537.899305	1425.660286	499.593700	3.870944	
std	2.003685	2.135918	12.583924	2181.794772	421.425970	1132.530137	382.357072	1.899961	
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	
25%	-121.800000	33.930000	18.000000	1448.000000	296.000000	787.000000	280.000000	2.563100	
50%	-118.500000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.535200	
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743700	
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	

Explore the data and visualize

It helps to identify peculiarities in our datasets, and if we need to take into account some specificity that we highlight through some simple visualization plots.

In [ ]:
```
'''
We call hist() function defined for pandas dataframes.
It's a Pandas function that generates histograms for numerical features
'''

housing.hist(bins=50, figsize=(20,15))
plt.show()
```



Some comments on the data

1. The variable *median income* doesn't seem to be expressed in US dollars.
    - They are capped:
        - to 15 (15000) for higher median income
        - 0.5 for lower median income
    - The numbers represent tens of thousands of dollars: 3$-->3000$
2. The housing **median age** and the **median house** value are also capped
3. All the features present almost a different scale w.r.t. the others.
4. Many histograms are tail-heavy. The machine learning algorithms can have problems to detect this not-that-frequent patterns.

We discussed some other functions to easily and quickly visualize the data of our dataframes.

- Correlation Matrix
- Seaborn pairplots
- KDE

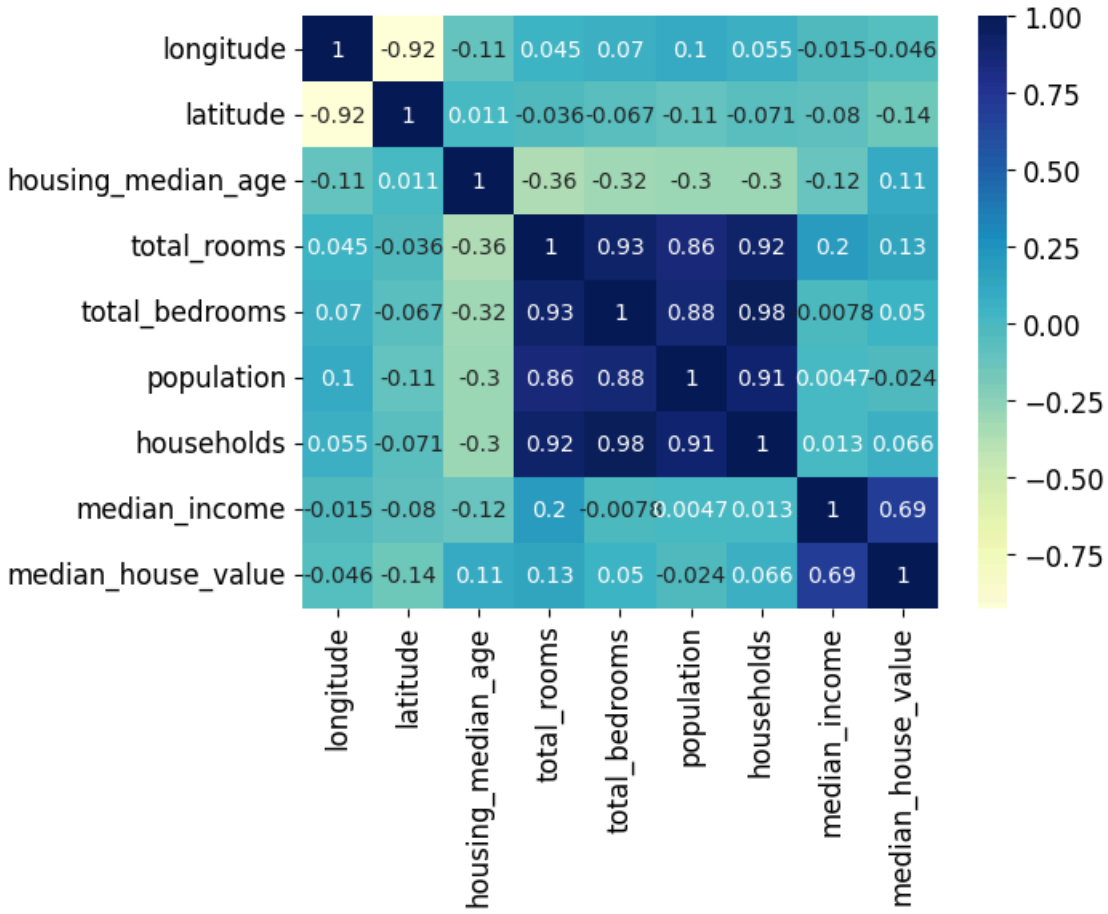
In [ ]:

```
# tipically we group by the classes when we deal with a classification task
# In any case, we could see if there are any interesting pattern emerging, or differences in the distribution
s.
plot=True
if plot:
  _ = sns.pairplot(housing[:5000],hue='ocean_proximity')
```

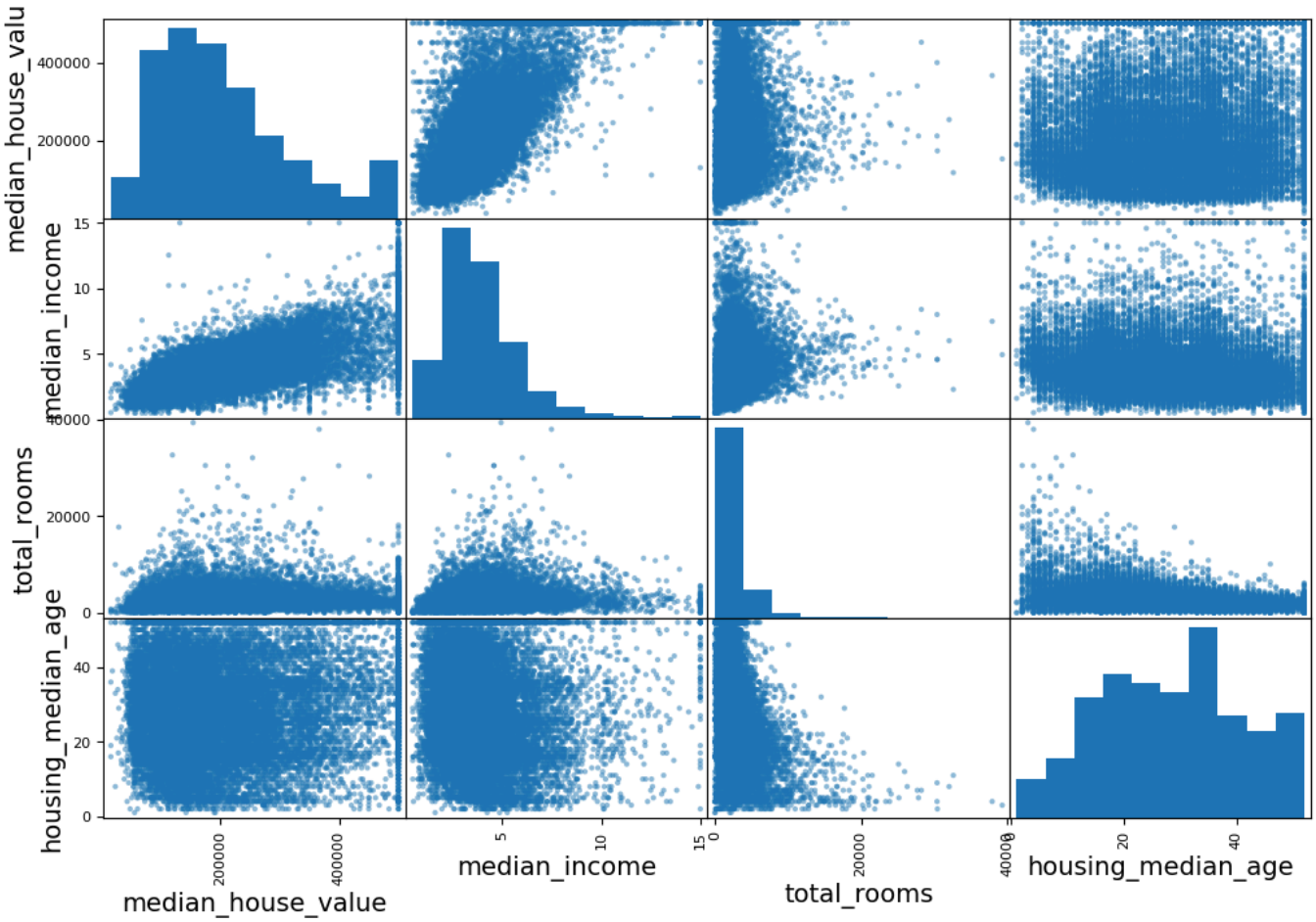
In [ ]:

```
'''
Another check that we could do is to verify if features have strong correlation among them or with the target variable
'''

corr = housing.corr('pearson',numeric_only=True)
_ = sns.heatmap(corr, annot=True,cmap="YlGnBu")
```



In [ ]:
```
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
_ = scatter_matrix(housing[attributes], figsize=(12, 8))
# save_fig("scatter_matrix_plot")
```



In [ ]:
```
corr
```

Out[ ]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	m
longitude	1.000000	-0.924676	-0.108394	0.044642	0.069653	0.099881	0.055400	-0.015090	
latitude	-0.924676	1.000000	0.011462	-0.036231	-0.067066	-0.108978	-0.071199	-0.079977	
housing_median_age	-0.108394	0.011462	1.000000	-0.361268	-0.320486	-0.296172	-0.302863	-0.118949	
total_rooms	0.044642	-0.036231	-0.361268	1.000000	0.930382	0.857117	0.918480	0.197991	
total_bedrooms	0.069653	-0.067066	-0.320486	0.930382	1.000000	0.877758	0.979740	-0.007767	
population	0.099881	-0.108978	-0.296172	0.857117	0.877758	1.000000	0.907213	0.004737	
households	0.055400	-0.071199	-0.302863	0.918480	0.979740	0.907213	1.000000	0.012950	
median_income	-0.015090	-0.079977	-0.118949	0.197991	-0.007767	0.004737	0.012950	1.000000	
median_house_value	-0.046208	-0.143837	0.105272	0.134373	0.049792	-0.024421	0.066069	0.688563	

In [ ]:
```
'''
We can check which features have a greater correlation with the target variable
'''
corr["median_house_value"].sort_values(ascending=False)
```

Out[ ]:

	median_house_value
median_house_value	1.000000
median_income	0.688563
total_rooms	0.134373
housing_median_age	0.105272
households	0.066069
total_bedrooms	0.049792
population	-0.024421
longitude	-0.046208
latitude	-0.143837

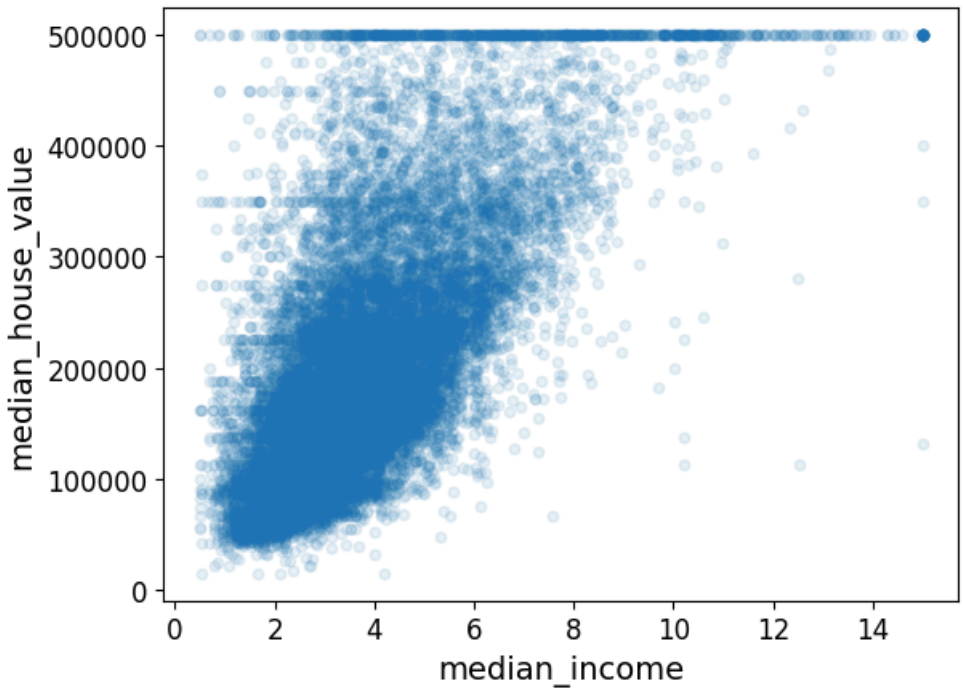
dtype: float64

In [ ]:
```
'''
Looking at the previous results, the __median_income__ is the most correlated.
Let's focus just with this plot now

We can notice the capped value at 500000$ and other unexpected horizontal lines.
These might indicate some odd patterns in the data.
A datascientist should evaluate if to keep these rows in the pipeline or discard them.

'''

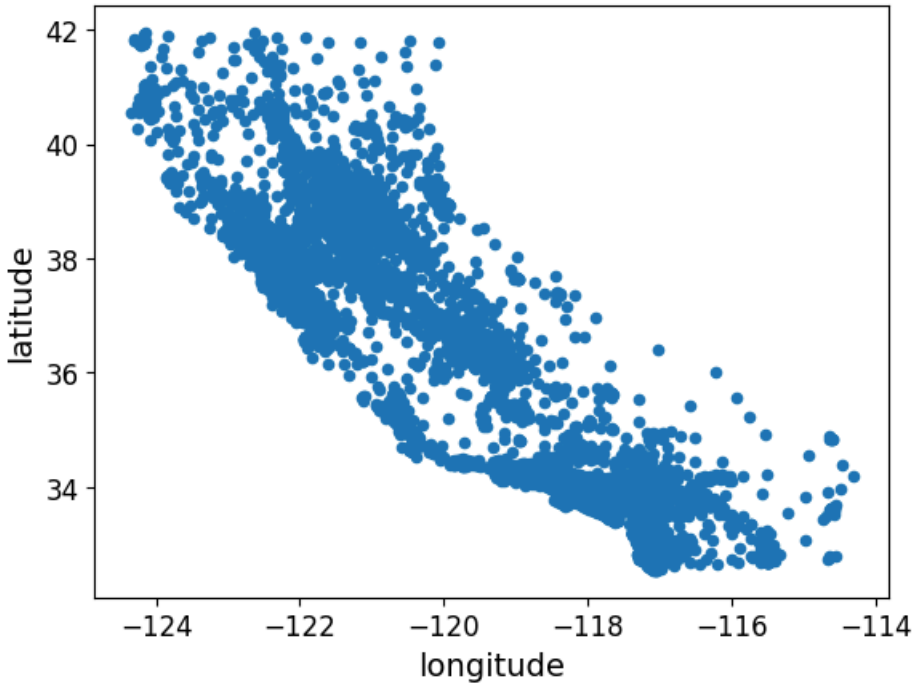
_ = housing.plot(kind='scatter',x='median_income',y='median_house_value',alpha=.1)
```



In [ ]:
```
'''
First plot
'''

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

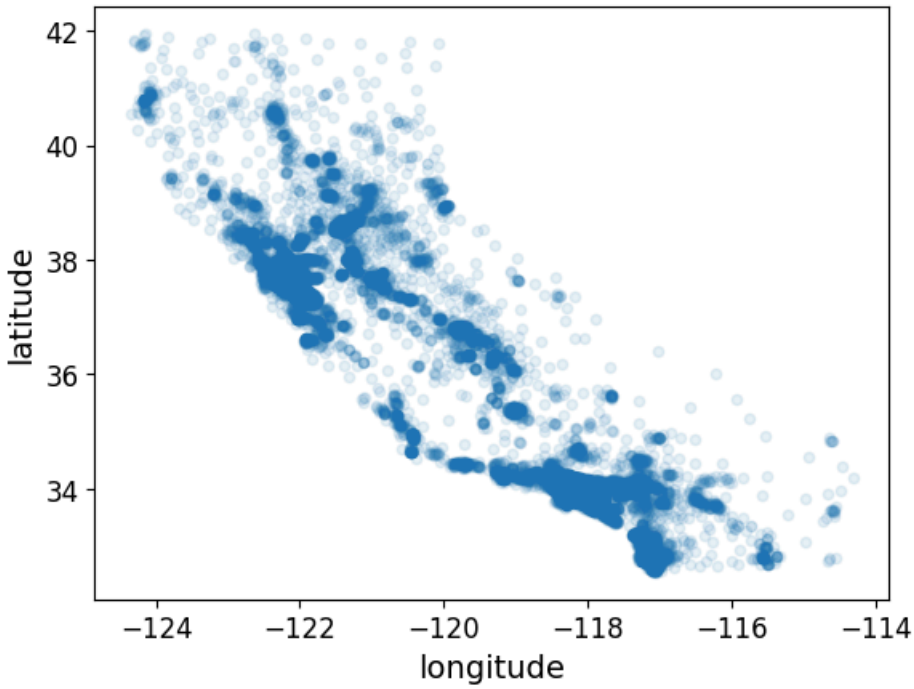
Out[ ]: <Axes: xlabel='longitude', ylabel='latitude'>



In [ ]:
```
'''
Changing the value of the alpha parameter is possible now to spot some interesting patterns,
i.e. the high-density areas, namely the Bay Area and around Los Angeles and San Diego,
plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno.
'''

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

Out[ ]: <Axes: xlabel='longitude', ylabel='latitude'>



In [ ]:

In [ ]:
```
'''
Let's check the median house prices

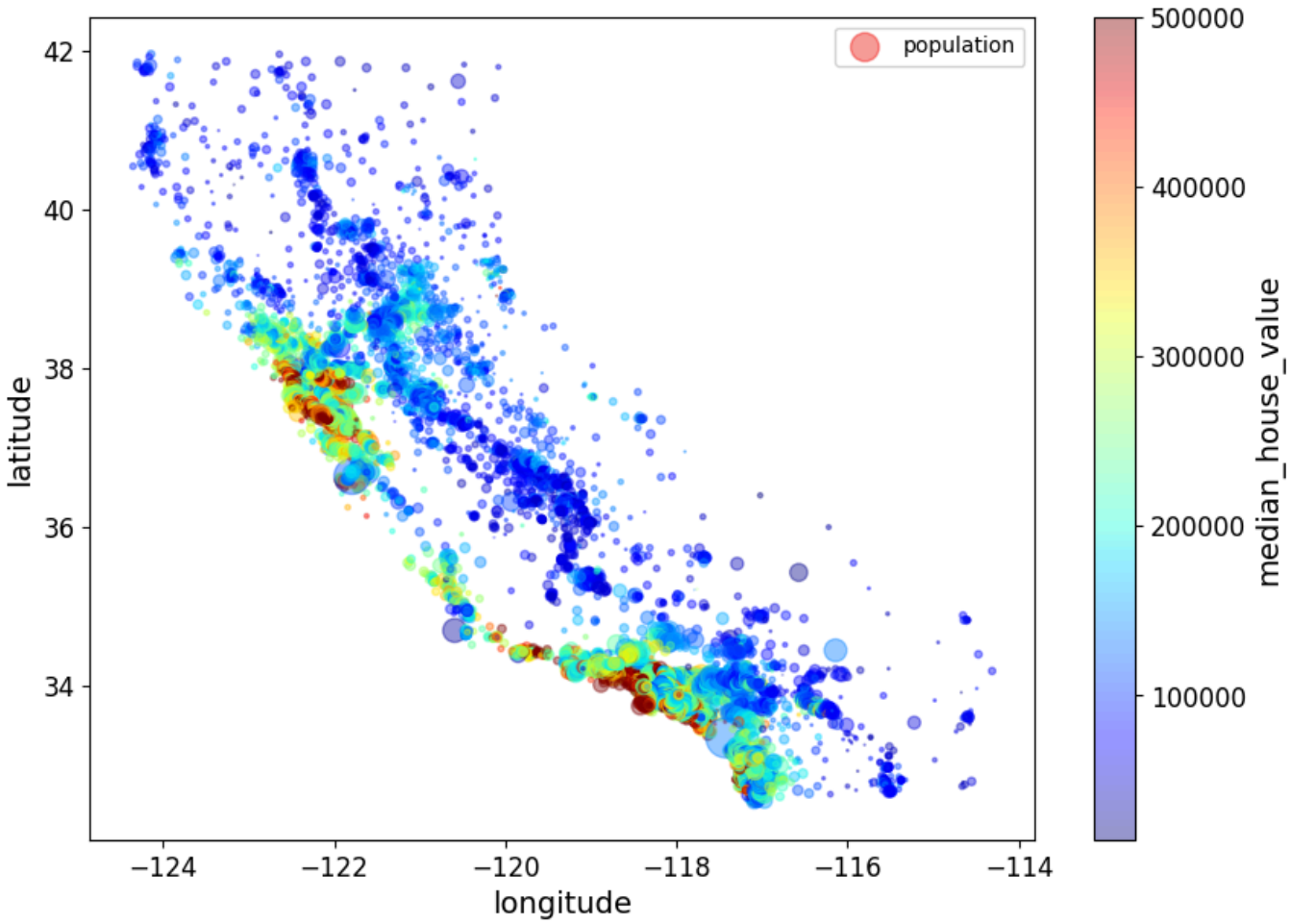
The radius of each circle represents the district's population (option s that stands for size),
and the color represents the price (option c that stands for color). Colors range from blue
(low values) to red (high prices):

The location has a high influence on the prices as well as the population density

'''

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

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f2a3fa81700>
```



Data preparation

Split in training and test.

Scikit-learn provides to the developers a series of function that can be used to split the original dataset in training and test.

- ***Training*** is the set used to train our model together with the relative pipeline
- ***Test*** is the dataset that shouldn't be touched until the very end of the ML process. Everything needs to be "learned" from the training set (e.g. imputing value for an inputer)

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

'''
1. random_state to get the same dataset split in different runs.
2. test_size the proportion of the dataset that we want to preserve for test, the rest is for training.
3. This is a purely random --> can introduce a significant sampling bias
'''

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

In [ ]:
```
type(train_set)
```

Out[ ]:
```
pandas.core.frame.DataFrame
def __init__(data=None, index: Axes | None=None, columns: Axes | None=None, dtype: Dtype | None=Non
e, copy: bool | None=None) -> None

Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns).
Arithmetic operations align on both row and column labels. Can be
thought of as a dict-like container for Series objects. The primary
```

We can imagine that the **median income** is a really important feature to predict the median houses pricing. Thus:

- We want to avoid a sample bias between training and test.
- Ensure that the test set is representative of the various categories of incomes in the whole dataset.

---> ***stratified sampling***: the population is divided into homogeneous subgroups called strata, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population

In order to use the median income as a categorical value we need to create a new column using a special function of pandas... do you remember?

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```
'''
pd.cut creates bins and assigns each row to a specific bin given the intervals provided to the function.
It's used to maintain the distribution of the feature in both the training set and the test set
'''

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])
```

In [ ]:
```
'''
Check the frequency of the new values
'''

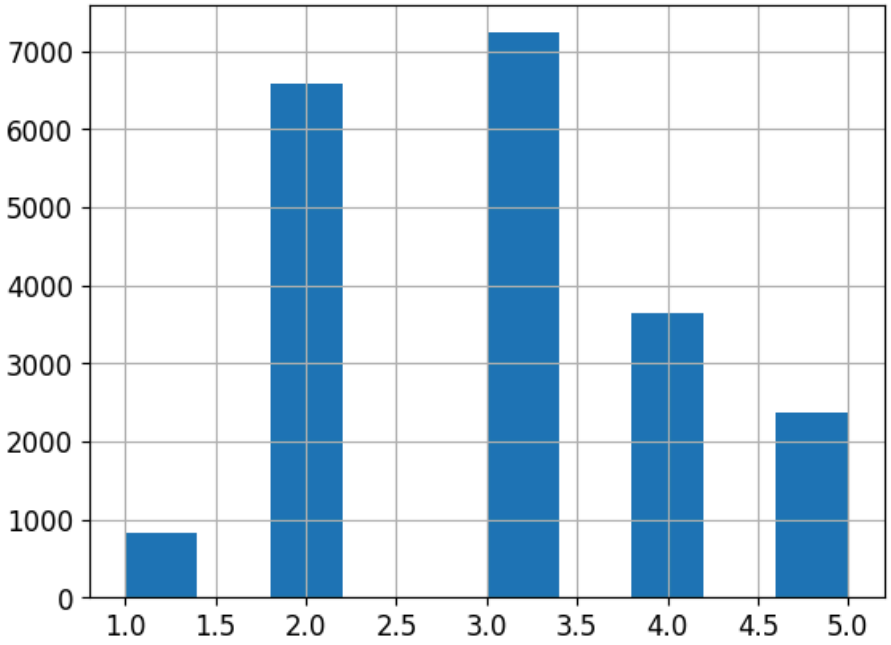
housing["income_cat"].value_counts()
```

Out[ ]:

income_cat	count
3	7235
2	6577
4	3639
5	2362
1	822

dtype: int64

In [ ]:
```
'''
and here the relative plot
'''
_ = housing["income_cat"].hist()
```



In [ ]:
```
'''
Now that we have a column with discrete values, it is easier to preserve the distributions between training an
d test set.
Scikit provides StratifiedShuffleSplit

Provides train/test indices to split data in train/test sets.

This cross-validation object is a merge of StratifiedKFold and
ShuffleSplit, which returns stratified randomized folds
'''

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]
```

In [ ]:
```
'''
Let's check if the proportion of income cat is mantained between test and training
'''
print(strat_test_set["income_cat"].value_counts() / len(strat_test_set))
print(strat_train_set["income_cat"].value_counts() / len(strat_train_set))
```

```
income_cat
3    0.350618
2    0.318633
4    0.176399
5    0.114369
1    0.039981
Name: count, dtype: float64
income_cat
3    0.350618
2    0.318755
4    0.176339
5    0.114490
1    0.039799
Name: count, dtype: float64
```

In [ ]:
```
print(strat_test_set["income_cat"].value_counts() )
print(strat_train_set["income_cat"].value_counts())
```

```
income_cat
3    1447
2    1315
4     728
5     472
1     165
Name: count, dtype: int64
income_cat
3    5788
2    5262
4    2911
5    1890
1     657
Name: count, dtype: int64
```

In [ ]:
```
strat_train_set.shape
```

Out[ ]:
```
(16508, 11)
```

In [ ]:
```
strat_test_set.shape
```

Out[ ]:
```
(4127, 11)
```

In [ ]:
```
'''
We can compare the results with the pure random split
'''

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
```

Out[ ]:

income_cat	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039835	0.039981	0.041919	5.231144	0.364964
2	0.318730	0.318633	0.321299	0.805839	-0.030409
3	0.350618	0.350618	0.359341	2.487906	0.000000
4	0.176351	0.176399	0.165980	-5.880736	0.027480
5	0.114466	0.114369	0.111461	-2.624894	-0.084674

In [ ]:
```
'''
And finally we can drop the created column
'''

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

Feature engineering

In [ ]:
```
'''
Another strategy to enrich the dataset under analysis is to create new features
defined as a combination of the existing ones.

For example, in this case, the number of rooms per district is not really informative,
if you don't normalize this number by the number of households.

Similarly, the total number of bedrooms by itself is not very useful: you probably
want to compare it to the number of rooms.

The population per household also
seems like an interesting attribute combination to look at

'''

# we create new columns in the dataframe using the pandas syntax and we combine
# two columns by arithmetic operations.

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"]
```

In [ ]:
```
'''
And now we can check if we found some interesting correlation with the target variable and the new ones.
'''

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

Out[ ]:

	median_house_value
median_house_value	1.000000
median_income	0.688563
rooms_per_household	0.151968
total_rooms	0.134373
housing_median_age	0.105272
households	0.066069
total_bedrooms	0.049792
population_per_household	-0.023719
population	-0.024421
longitude	-0.046208
latitude	-0.143837
bedrooms_per_room	-0.256397

dtype: float64

1. The **bedrooms_per_room** feature is much more correlated with the median_house_value than the total number of rooms or bedrooms. It seems that houses with a lower bedroom/room ration tend to be more expensive
2. Also the **rooms_per_household** is more informative than the total number of rooms in a district.

In [ ]:
```
'''
From now on we will use the strat_train_set as DataFrame
'''

housing = strat_train_set.drop("median_house_value", axis=1) # drop labels for training set
housing_labels = strat_train_set["median_house_value"].copy() # our labels
```

In [ ]:
```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 16508 entries, 20120 to 19283
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           16508 non-null  float64
 1   latitude            16508 non-null  float64
 2   housing_median_age  16508 non-null  float64
 3   total_rooms         16508 non-null  float64
 4   total_bedrooms      16346 non-null  float64
 5   population          16508 non-null  float64
 6   households          16508 non-null  float64
 7   median_income       16508 non-null  float64
 8   ocean_proximity     16508 non-null  object 
dtypes: float64(8), object(1)
memory usage: 1.3+ MB
```

Data cleaning

Missing values

• Get rid of the corresponding districts.

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

• Get rid of the whole attribute.

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

• Set the values to some value (zero, the mean, the median, etc.).

- `median = housing["total_bedrooms"].median()`
- `housing["total_bedrooms"].fillna(median, inplace=True)`
- compute the median value on the training set, and use that value to replace missing values in the test set when you want to evaluate your system

For this last option scikit provides a simple class `SimpleImputer`

--> This allows us to perform all the *transformations* of our dataset only using Scikit-learn

In [ ]:
```
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

Out[ ]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
20120	-118.80	34.41	45.0	1610.0	NaN	1148.0	347.0	2.7000	<1H OCEAN
5665	-118.29	33.73	30.0	3161.0	NaN	1865.0	771.0	2.7139	NEAR OCEAN
5990	-117.73	34.10	37.0	3457.0	NaN	1344.0	530.0	5.8891	INLAND
5654	-118.30	33.73	42.0	1731.0	NaN	866.0	403.0	2.7451	NEAR OCEAN
4743	-118.36	34.05	42.0	1372.0	NaN	674.0	271.0	2.8793	<1H OCEAN

In [ ]:
```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

Out[ ]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity

In [ ]:
```
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

Out[ ]:

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	ocean_proximity
20120	-118.80	34.41	45.0	1610.0	1148.0	347.0	2.7000	<1H OCEAN
5665	-118.29	33.73	30.0	3161.0	1865.0	771.0	2.7139	NEAR OCEAN
5990	-117.73	34.10	37.0	3457.0	1344.0	530.0	5.8891	INLAND
5654	-118.30	33.73	42.0	1731.0	866.0	403.0	2.7451	NEAR OCEAN
4743	-118.36	34.05	42.0	1372.0	674.0	271.0	2.8793	<1H OCEAN

In [ ]:
```
median = housing["total_bedrooms"].median()
print("The median of total_bedrooms is {}".format(median))
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
sample_incomplete_rows
```

```
The median of total_bedrooms is 435.0

/tmp/ipython-input-1809300428.py:3: FutureWarning: A value is trying to be set on a copy of a DataFrame or Ser
ies through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on
which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=Tru
e)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.
```

sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3

Out[ ]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
20120	-118.80	34.41	45.0	1610.0	435.0	1148.0	347.0	2.7000	<1H OCEAN
5665	-118.29	33.73	30.0	3161.0	435.0	1865.0	771.0	2.7139	NEAR OCEAN
5990	-117.73	34.10	37.0	3457.0	435.0	1344.0	530.0	5.8891	INLAND
5654	-118.30	33.73	42.0	1731.0	435.0	866.0	403.0	2.7451	NEAR OCEAN
4743	-118.36	34.05	42.0	1372.0	435.0	674.0	271.0	2.8793	<1H OCEAN

In [ ]:
```
'''
Use the scikit class SimpleImputer.

First, we create an instance of that type of object using a median as the strategy

'''

from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

In [ ]:
```
imputer.statistics_
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipython-input-1710085974.py in <cell line: 0>()
----> 1 imputer.statistics_

AttributeError: 'SimpleImputer' object has no attribute 'statistics_'
```

In [ ]:
```
'''
Use the scikit class SimpleImputer.

First, we create an instance of that type of object using a median as the strategy

'''

from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")

# we drop the non numerical column/feature/attribute
housing_num = housing.drop("ocean_proximity", axis=1)

# we fit the imputer on the sub-set just created.
imputer.fit(housing_num)
```

Out[ ]:

SimpleImputer (https://scikit-learn.org/1.6/modules/generated/sklearn.impute.SimpleImputer.html)

SimpleImputer(strategy='median')

In [ ]:
```
imputer.statistics_
```

Out[ ]:
```
array([-118.51   ,   34.26   ,   29.     , 2131.     ,  435.     ,
       1170.     ,  410.     ,    3.54275])
```

In [ ]:
```
'''
The Imputer has simply computed the median of each attribute and stored the result
in its statistics_ instance variable.

Only the total_bedrooms attribute had missing values, but we cannot be sure that
there won't be any missing values in new data after
the system goes live, so it is safer to apply the imputer to all the numerical attributes
'''
imputer.statistics_
print([ '{} {}'.format(c,m)  for c,m in zip(housing_num.columns,imputer.statistics_)])
```

```
['longitude -118.51', 'latitude 34.26', 'housing_median_age 29.0', 'total_rooms 2131.0', 'total_bedrooms 435.
0', 'population 1170.0', 'households 410.0', 'median_income 3.54275']
```

In [ ]:
```
'''
Now we can transform our dataset
'''
X = imputer.transform(housing_num)
print(type(X))

# we create a new dataframe since the result of the transform operation is a numpy array.
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

```
<class 'numpy.ndarray'>
```

In [ ]:
```
housing_tr.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16508 entries, 0 to 16507
Data columns (total 8 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           16508 non-null  float64
 1   latitude            16508 non-null  float64
 2   housing_median_age  16508 non-null  float64
 3   total_rooms         16508 non-null  float64
 4   total_bedrooms      16508 non-null  float64
 5   population          16508 non-null  float64
 6   households          16508 non-null  float64
 7   median_income       16508 non-null  float64
dtypes: float64(8)
memory usage: 1.0 MB
```

Categorical values

Most Machine Learning algorithms prefer to work with numbers anyway, so let's convert these categories from text to numbers.

There are different strategies and it depends the type of data we are handling.

OrdinalEncoder when the order of the values has a meening, e.g. bad, good, very good, excellent.

```
from sklearn.preprocessing import OrdinalEncoder
```

OneHotEncoder when the values do not present a meaningful order, e.g. provinces.

```
from sklearn.preprocessing import OneHotEncoder
```

In [ ]:
```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(10)

from sklearn.preprocessing import OrdinalEncoder

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

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

In [ ]:
```
ordinal_encoder.categories_
```

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

1. What happen if I call only `.fit()` ?
2. how is the object `ordinal_encoder` modified?
3. Does something happen to the input dataframe?

If we only .fit() then the dataframe doesn't change, but only create the ordinal_encoder

In [ ]:
```
from sklearn.preprocessing import OneHotEncoder

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

Out[ ]:
```
<Compressed Sparse Row sparse matrix of dtype 'float64'
        with 16508 stored elements and shape (16508, 4)>
```

In [ ]:
```
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[ ]:
```
<Compressed Sparse Row sparse matrix of dtype 'float64'
        with 16508 stored elements and shape (16508, 4)>
```

Given the flexibility of scikit, and the API structure, it is possible to define Custom Transformers.

All we need is to create a class with three methods:

1. fit() --> return self
2. transform()
3. fit_transform()

In our case, we can use a custom transformer for the creation of the new attributes defined as a combination of attributes.

In [ ]:
```
'''
TransformerMixin already provides fit_transform for free.

Adding BaseEstimator we get two extra methods get_params() and set_params()
These can be useful for automatic hyperparameter tuning

add_bedrooms_per_room is the only hyperparameter, it allows us to understand if this computed attribute is use
ful to increase the quality of our model.
'''



from sklearn.base import BaseEstimator, TransformerMixin

# column index
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 ## this is the only hyperparameter that we would li
ke to tune.
    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)
```

Feature scaling

- **Min-max** scaling (many people call this **normalization**) is quite simple: values are shifted and rescaled so that they end up ranging from 0 to 1 -
  - `MinMaxScaler()`
- **Standardization** is quite different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms
  - `StandardScaler()`

  Notice that it is important to fit the scalers to the **training data only**, not to the full dataset (including the test set). Only then can you use them to transform the training set and the test set (and new data).

Using feature scaling we can obtain values in different columns (features) that can be comparatated.

So, until now, we saw: how to treat missing values a categorical values a how feature engineering (es. feature scaling) works.

In training part -> we use fit_transform()

In testing part -> we don't fit or transform but we just predict()

Pipeline

- Pipeline

  ```
  class sklearn.pipeline.Pipeline(steps, *, memory=None, verbose=False)
  ```

  - Sequentially apply a list of transforms and a final estimator. -Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods.
  - The final estimator only needs to implement fit.
  - The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.
  - For this, it enables setting *parameters* of the various *steps* using their names and the parameter name separated by a '__'.

In [ ]:
```
housing.shape
```

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

In [ ]:
```
'''
The Pipeline is built using a list of (key, value) pairs, where the key
is a string containing the name you want to give
this step and value is an estimator object:
'''

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

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

In [ ]:
```
housing_prepared[:5]
```

In [ ]:
```
type(housing_prepared)
```

In [ ]:
```
'''
The estimators of a pipeline are stored as a list in the steps attribute,
but can be accessed by index or name by indexing (with [idx]) the Pipeline:
'''

print('which is the first step of the pipeline? {}'.format(num_pipeline.steps[0]))
print('which is the first step of the pipeline? {}'.format(num_pipeline[0]))
print('which is the first step of the pipeline? {}'.format(num_pipeline['std_scaler']))

'''
tab completion when using the named_step attribute of the pipeline
'''
num_pipeline.named_steps.std_scaler
```

In [ ]:
```
'''
A sub-pipeline can also be extracted using the slicing notation commonly used
for Python Sequences such as lists or strings (although only a step of 1 is permitted).
This is convenient for performing only some of the transformations (or their inverse):
'''

print(num_pipeline[0:2])
```

In [ ]:
```
'''

Parameters of the estimators in the pipeline can be accessed using the <estimator>__<parameter> syntax
'''

num_pipeline.set_params(attribs_adder__add_bedrooms_per_room=True)

'''
This is particularly important for doing grid search, we will see later

'''
```

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

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

In [ ]:
```
# 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))
```

In [ ]:
```
print("Labels:", list(some_labels))
```

In [ ]:
```
from sklearn.metrics import mean_squared_error

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
print(lin_mse)
```

In [ ]:
```
from sklearn.metrics import mean_absolute_error

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

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

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

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

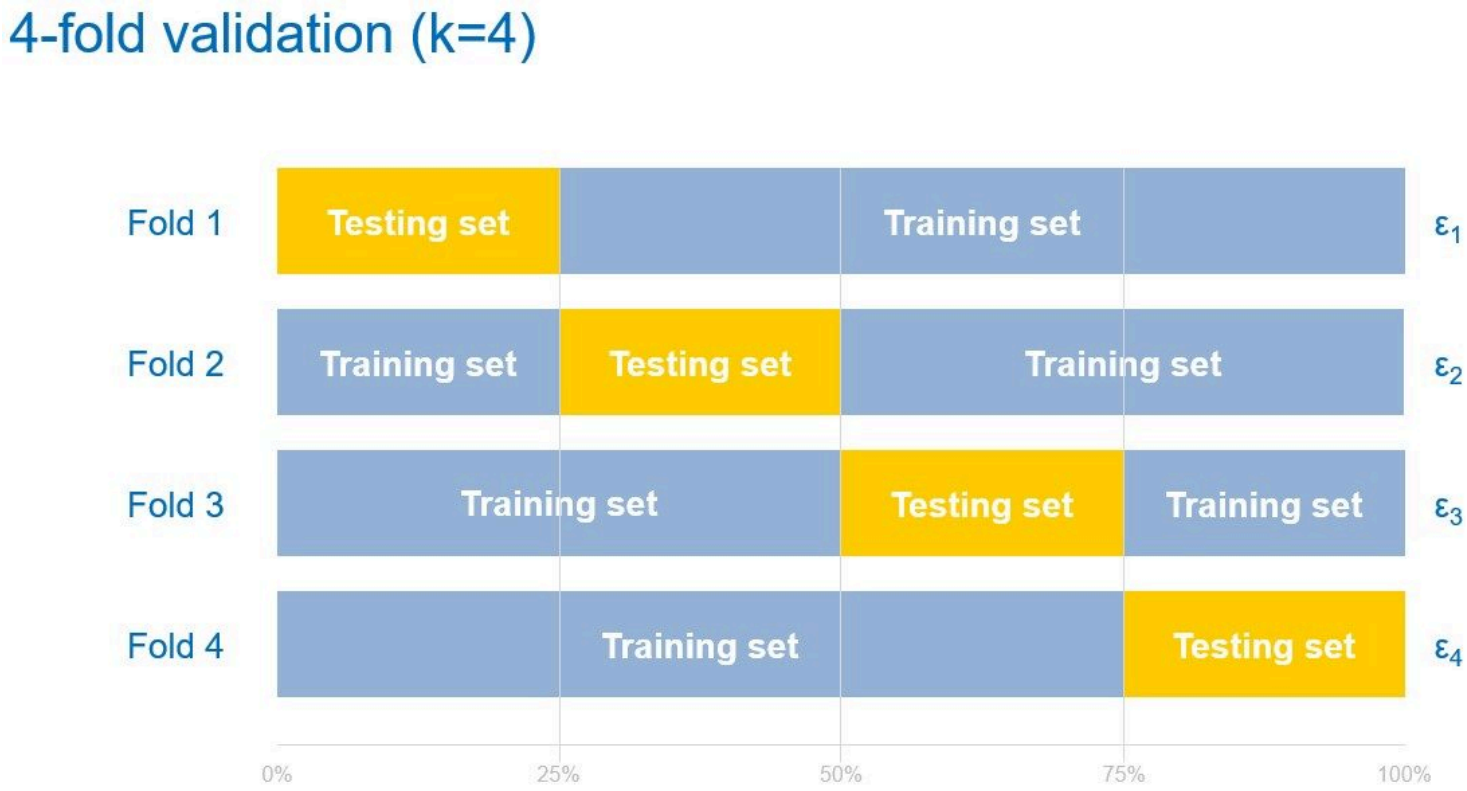
Check list for a Machine Learning project

1. Understand the problem.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
    - split on training and test
    - handling missing values
    - categorical values
    - scaling
    - etc.
5. Compare different models, choose the best and train it.
6. Fine-tune your model.
7. Present your solution.
    - stakeholder
    - paper
    - supervisor
    - etc.

GridSearchCV

- ```
  class sklearn.model_selection.GridSearchCV(estimator, param_grid, *,
  scoring=None, n_jobs=None, refit=True, cv=None, verbose=0,
  pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False)
  ```

- Exhaustive search over specified parameter values for an estimator.
- Important members are fit, predict.
- GridSearchCV implements a “fit” and a “score” method.
- It also implements “score_samples”, “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.
- The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.
- CV --> CrossValidation


  ```

In [ ]:
```
'''
cross_val_score 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 con
taining the 10 evaluation scores
'''
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

In [ ]:
```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

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

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

# Random Forest = Ensemble of Decision Trees

forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

In [ ]:
```
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

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

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

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

param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

In [ ]:
```
pd.DataFrame(grid_search.cv_results_).sort_values('rank_test_score')
```

GridSearch and Pipeline can be combined.

The estimator can be a parameter as well of the grid. It is possible to access the different parameters of each estimator as we saw before.

In [ ]:
```
pipeline = Pipeline(steps=[('preprocessor',full_pipeline),('regressor',RandomForestRegressor(random_state=4
2))])
```

In [ ]:
```
params =[{'preprocessor': [full_pipeline],
        'regressor': [LinearRegression()],
        'regressor__fit_intercept':[True]},
        {'preprocessor': [full_pipeline],
          'regressor': [RandomForestRegressor(random_state=42)],
         'regressor__n_estimators':[10,20],
         'regressor__min_samples_split': [5,10]}
         ]
grid_search = GridSearchCV(pipeline, params, cv=3,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing, housing_labels)
```

In [ ]:
```
grid_search.error_score
```

In [ ]:
```
grid_search.best_params_
```

In [ ]:
```
grid_search.best_estimator_
```

In [ ]:
```
cvres = pd.DataFrame(grid_search.cv_results_)
print(cvres.columns)
cvres['sqrt'] = np.sqrt(-cvres['mean_test_score'])
cvres.sort_values(by='sqrt',ascending=True)
```

In [ ]:
```
grid_search.best_estimator_
```

In [ ]:
```
type(grid_search.best_estimator_['regressor'])
```

In [ ]:
```
'''
Once we get the best estimator we can check -- at least for some models -- the feature importance.
Such as which are the features that are __more used__ to make the predictions

Firstly, we get the feature importance of our best regressor
Then, we associate each value to the relative name gathered from the pipeline
Finally, we print them.
'''

feature_importances = grid_search.best_estimator_['regressor'].feature_importances_
feature_importances

extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

Classification

Now we prove how good we are in a classification tasks.

Given a set of drawn digits we want to predict if the image is a 5 or not.

This is a problem of BINARY CLASSIFICATION

We will study which are the quality measures that we need to take into account to improve the perfomance of our model.

- Confusion matrix
- Precision vs Recall
- F1
- ROC and AUC

In [ ]:
```
'''
We gathered the MNIST Dataset from the dataset available directly on scikit
'''

from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.keys()
```

Out[ ]:
```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'ur
l'])
```

In [ ]:
```
'''
Let's explore some property of this dataset.
'''

X, y = mnist["data"], mnist["target"]
X.shape
```

Out[ ]:
```
(70000, 784)
```

In [ ]:
```
'''
And how it looks like
'''

import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)
plt.axis("off")

# save_fig("some_digit_plot")
plt.show()
```

In [ ]:
```python
X[0]
```

Out[ ]:
```
array([  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   3,  18,  18,  18,
       126, 136, 175,  26, 166, 255, 247, 127,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170, 253,
       253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253,
       253, 253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  18, 219, 253,
       253, 253, 253, 253, 198, 182, 247, 241,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
        80, 156, 107, 253, 253, 205,  11,   0,  43, 154,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,  14,   1, 154, 253,  90,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0, 139, 253, 190,   2,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190, 253,  70,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
       241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,  81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,  45, 186, 253, 253, 150,  27,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,  16,  93, 252, 253, 187,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 249,
       253, 249,  64,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  46, 130,
       183, 253, 253, 207,   2,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39, 148,
       229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114,
       221, 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  23,  66,
       213, 253, 253, 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  18, 171,
       219, 253, 253, 253, 253, 195,  80,   9,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  55, 172,
       226, 253, 253, 253, 253, 244, 133,  11,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
       136, 253, 253, 253, 212, 135, 132,  16,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0])
```

In [ ]:
```python
'''
y contains the label for each image
'''

y[0]
```

Out[ ]:
```
'5'
```

In [ ]:
```python
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.binary,
               interpolation="nearest")
    plt.axis("off")
```

In [ ]:
```
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

In [ ]:
```
import numpy as np
```

In [ ]:
```
'''
Some more images, just to have an idea of the content
'''
plt.figure(figsize=(9,9))
example_images = X[:100]
plot_digits(example_images, images_per_row=10)
# save_fig("more_digits_plot")
plt.show()
```

In [ ]:
```
'''
In this case, we don't use any scikit function to create training and test.
We simply take the first 60000 as training and the remaining as test.
'''

X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

In [ ]:
```
'''
As I said before, just to start and to understand some metrics tipically used
We build a binary class: it's a 5 or not.

Therefore, we will train a model to predict if a given image is a 5 or not.
'''

y_train_5 = (y_train == '5')
y_test_5 = (y_test == '5')
```

In [ ]:
```
y_train_5
```

Out[ ]:
```
array([ True, False, False, ...,  True, False, False])
```

Select the best evaluation metrics!

- Accuracy?
- Some others?

In [ ]:
```
'''
We try a Stochastic Gradient Descent algorithm
'''

from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Out[ ]:
```
SGDClassifier
SGDClassifier(random_state=42)
```

class sklearn.linear_model.SGDClassifier(loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001,
shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5, early_stopping=False,
validation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False, average=False)[source]¶

In [ ]:
```
sgd_clf.predict([some_digit])
```

Out[ ]:
```
array([ True])
```

In [ ]:
```
len(y_train_5)-y_train_5.sum()
```

Out[ ]:
```
np.int64(54579)
```

In [ ]:
```
'''
Let's try a cross validation, as we did it before.

- Notice that we use as scoring metrics the ACCURACY
'''

from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Out[ ]:
```
array([0.95035, 0.96035, 0.9604 ])
```

In [ ]:
```
'''
Our classifier seems working pretty good.

And if define a classifier that predicts always false?

'''

from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

In [ ]:
```
'''
The model accuracy is pretty high also for this simple classifier, greater than 90%

Probably the accuracy is not the right metrics to use.
'''

never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Out[ ]:
```
array([0.91125, 0.90855, 0.90915])
```

In [ ]:
```
'''
We can check how our classifier predicts all the instances w.r.t. the real label

This is represented as matrix called Confusion Matrix.

Each row represents an actual class
Each column represents the predicted class
_____________
|     |     |
| TP  | FP  |
|_____|_____|
|     |     |
| FN  | TN  |
|_____|_____|

The diagonal contains the correctly labelled instances

'''
```

In [ ]:
```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

confusion_matrix(y_train_5, y_train_pred)
```

Out[ ]:
```
array([[53892,   687],
       [ 1891,  3530]])
```

In [ ]:
```
y_train_perfect_predictions = y_train_5  # pretend we reached perfection
confusion_matrix(y_train_5, y_train_perfect_predictions)
```

Out[ ]:
```
array([[54579,     0],
       [    0,  5421]])
```

Precision and Recall

- Precision
  - accuracy of the positive predictions
  - $TP/(TP+FP)$
  - `from sklearn.metrics import precision_score`
- Recall
  - ratio of positive instances that are correctly detected by the classifier
  - $TP/(TP+FN)$
  - also called *sensitivity*
  - `from sklearn.metrics import recall_score`
- F1
  - The harmonic mean between Precision and Recall
  - $2 * (Precision * Recall)/(Precision + Recall)$
  - `from sklearn.metrics import f1_score`

In [ ]:
```
from sklearn.metrics import precision_score, recall_score

print('Precision score: {}'.format(precision_score(y_train_5, y_train_pred)))

cm = confusion_matrix(y_train_5, y_train_pred)
print('Precision score: {}'.format(cm[1, 1] / (cm[0, 1] + cm[1, 1])))
```

```
Precision score: 0.8370879772350012
Precision score: 0.8370879772350012
```

In [ ]:
```
print('Recall score: {}'.format(recall_score(y_train_5, y_train_pred)))
print('Recall score: {}'.format(cm[1, 1] / (cm[1, 0] + cm[1, 1])))
```

```
Recall score: 0.6511713705958311
Recall score: 0.6511713705958311
```

In [ ]:
```
from sklearn.metrics import f1_score

print('F1 score: {}'.format(f1_score(y_train_5, y_train_pred)))
print('F1 score: {}'.format(cm[1, 1] / (cm[1, 1] + (cm[1, 0] + cm[0, 1]) / 2)))
```

```
F1 score: 0.7325171197343847
F1 score: 0.7325171197343847
```

In [ ]:
```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

Out[ ]:
```
array([2164.22030239])
```

In [ ]:
```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
```

In [ ]:
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

In [ ]:
```
y_scores.shape
```

Out[ ]:
```
(60000,)
```

In [ ]:
```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

There are cases in which for the final goal of the project is better to have a higher precision, while in others it is better to increase the recall.

In order to find a good balance between them, we will explore some techniques that can be used in your future projects:

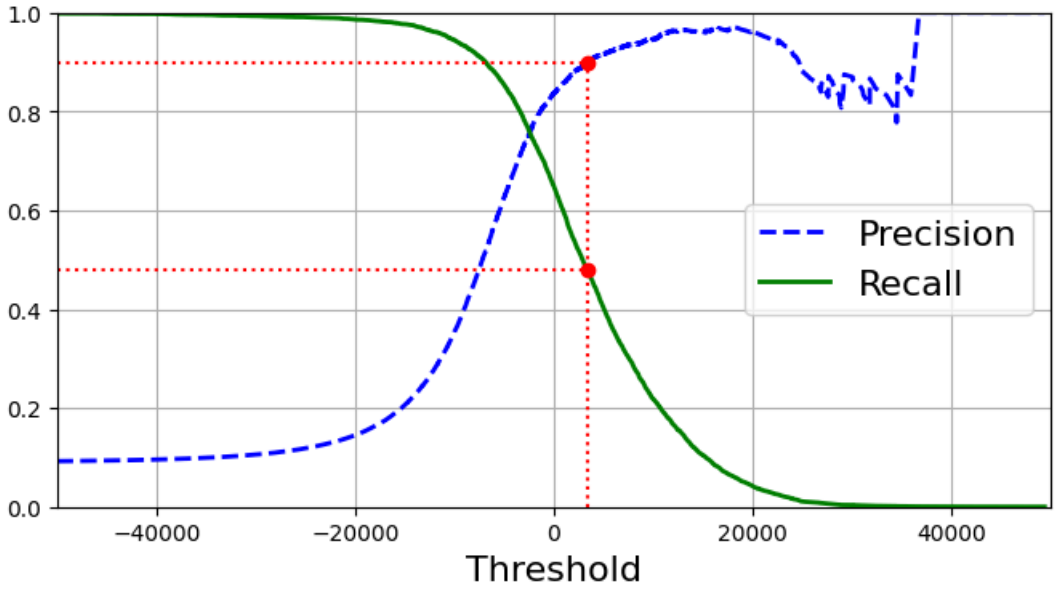
- Precision/Recall curve
- ROC and AUC

They both use the fact that scikit classifiers compute a score based on a *decision function* (or predict_proba), and if that score is greater than a threshold, it assigns the instance to the positive class, or else it assigns it to the negative class.

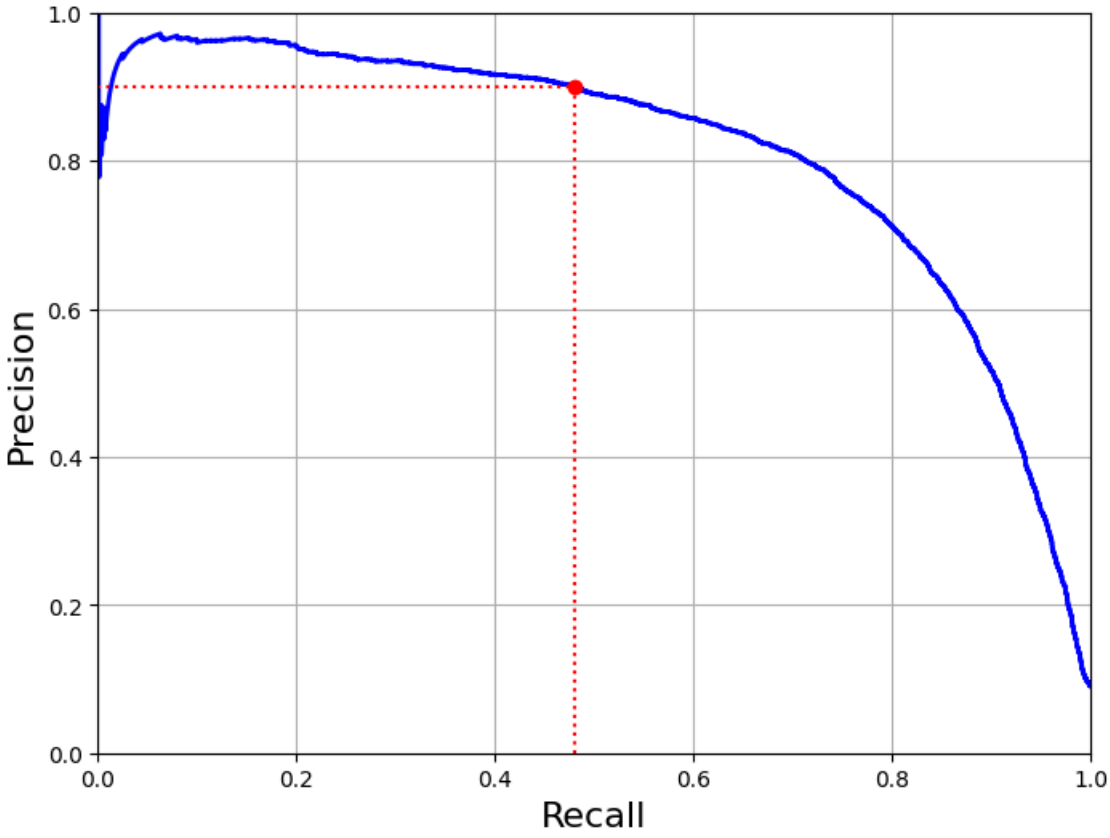
- Precision/Recall curve
  - Varying the threshold use to classify an instance, it plots for each value the relative precision and recall, so that we can identify the right threshold to have a certain precision (recall).
- ROC
  - true positive rate (another name for recall) against the false positive rate (FPR) . The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the true negative rate, which is the ratio of negative instances that are correctly classified as negative. The TNR is also called *specificity*.
  - Also called sensitivity vs specificity
- To compare two classifiers we use the Area Under the Curve (AUC) A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.

In [ ]:

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.legend(loc="center right", fontsize=16)
    plt.xlabel("Threshold", fontsize=16)
    plt.grid(True)
    plt.axis([-50000, 50000, 0, 1])



recall_90_precision = recalls[np.argmax(precisions >= 0.90)]
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]


plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [recall_90_precision, recall_90_precision], "r:")
plt.plot([threshold_90_precision], [0.9], "ro")
plt.plot([threshold_90_precision], [recall_90_precision], "ro")
#save_fig("precision_recall_vs_threshold_plot")
plt.show()
```



In [ ]:
```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.plot([recall_90_precision, recall_90_precision], [0., 0.9], "r:")
plt.plot([0.0, recall_90_precision], [0.9, 0.9], "r:")
plt.plot([recall_90_precision], [0.9], "ro")
# save_fig("precision_vs_recall_plot")
plt.show()
```



In [ ]:
```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

In [ ]:
```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

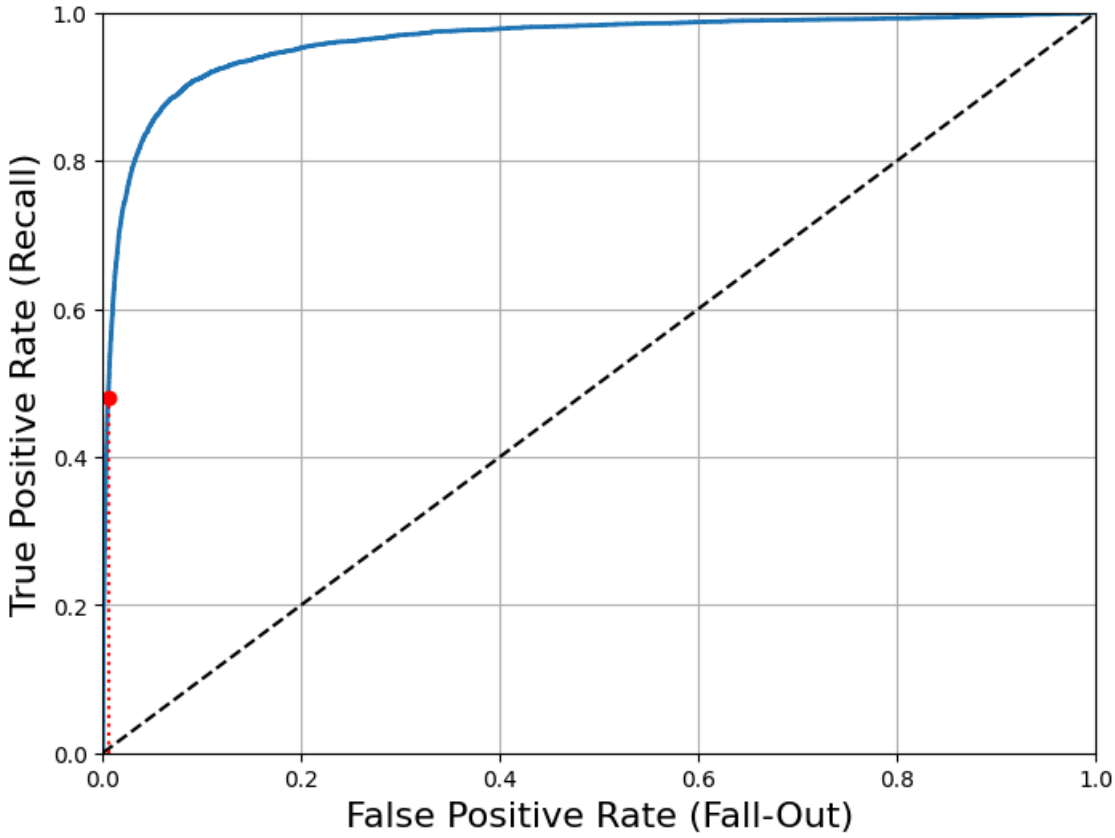
In [ ]:
```
precision_score(y_train_5, y_train_pred_90)
```

Out[ ]:
```
0.9000345901072293
```

In [ ]:
```
recall_score(y_train_5, y_train_pred_90)
```

Out[ ]:
```
0.4799852425751706
```

In [ ]:
```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # dashed diagonal
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=16)
    plt.ylabel('True Positive Rate (Recall)', fontsize=16)
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)
fpr_90 = fpr[np.argmax(tpr >= recall_90_precision)]
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
plt.plot([fpr_90], [recall_90_precision], "ro")
#save_fig("roc_curve_plot")
plt.show()
```



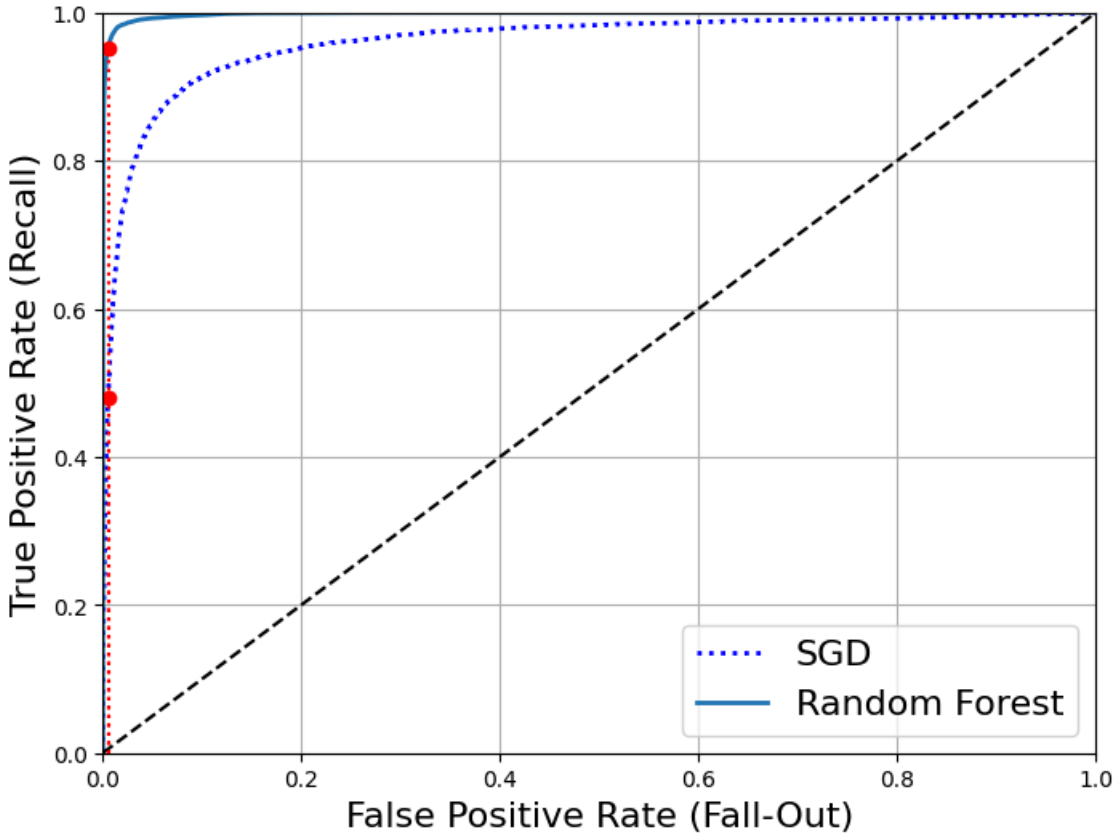
In [ ]:
```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_train_5, y_scores)
```

Out[ ]:
```
np.float64(0.9604938554008616)
```

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

In [ ]:
```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

In [ ]:
```
recall_for_forest = tpr_forest[np.argmax(fpr_forest >= fpr_90)]

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
plt.plot([fpr_90], [recall_90_precision], "ro")
plt.plot([fpr_90, fpr_90], [0., recall_for_forest], "r:")
plt.plot([fpr_90], [recall_for_forest], "ro")
plt.grid(True)
plt.legend(loc="lower right", fontsize=16)
#save_fig("roc_curve_comparison_plot")
plt.show()
```



In [ ]:
```
!pip install xgboost
```

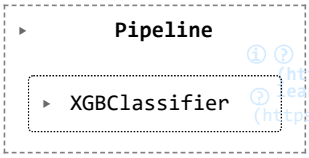
```
Requirement already satisfied: xgboost in /usr/local/lib/python3.12/dist-packages (3.0.5)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from xgboost) (2.0.2)
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.12/dist-packages (from xgboost) (2.2
7.3)
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packages (from xgboost) (1.16.2)
```

In [ ]:
```
from sklearn.ensemble import GradientBoostingClassifier,RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler

pipeline = Pipeline(steps=[('classifier',RandomForestClassifier())])

params = [
        {
         'classifier': [GradientBoostingClassifier(random_state=42)],
         'classifier__max_depth':[10],
         'classifier__n_estimators': [10,20],
         'classifier__n_iter_no_change':[10],
         'classifier__learning_rate':[0.1,1]
        },
    {
          'classifier': [XGBClassifier(random_state=42)],
         'classifier__n_estimators': [10,20],
          'classifier__learning_rate':[0.1,1]
        },
          # {
          #  'classifier': [RandomForestClassifier(random_state=42)],
          #  'classifier__n_estimators':[10,20],
          #  'classifier__min_samples_split':[4],
          # }
]

grid_search = GridSearchCV(pipeline, params, verbose=10, n_jobs=3,scoring='roc_auc',cv=3).fit(X_train[:1000],
y_train_5[:1000])
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
```

In [ ]:
```
grid_search.best_estimator_
```

Out[ ]:



In [ ]:
```
import pandas as pd

cv = pd.DataFrame(grid_search.cv_results_)
```

In [ ]:
```
cv
```

Out[ ]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_classifier	param_classifier__learning_rate
0	1.949004	0.248246	0.007219	0.001849	GradientBoostingClassifier(random_state=42)	0.1
1	2.167941	0.160729	0.005066	0.002092	GradientBoostingClassifier(random_state=42)	0.1
2	1.294060	0.218776	0.005729	0.001764	GradientBoostingClassifier(random_state=42)	1.0
3	1.667534	0.646836	0.004730	0.001603	GradientBoostingClassifier(random_state=42)	1.0
4	0.439263	0.030983	0.007955	0.002358	XGBClassifier(base_score=None, booster=None, c...	0.1
5	0.690340	0.116133	0.009007	0.003312	XGBClassifier(base_score=None, booster=None, c...	0.1
6	0.273603	0.065269	0.007739	0.002502	XGBClassifier(base_score=None, booster=None, c...	1.0
7	0.264268	0.056385	0.006666	0.003224	XGBClassifier(base_score=None, booster=None, c...	1.0

Let's try our checklist on a new dataset

Titanic -- well known dataset available from Kaggle

In [ ]:
```
training = pd.read_csv('/content/drive/Shareddrives/phd_hands_on/data/titanic/train.csv')
```

In [ ]:
```
test = pd.read_csv('/content/drive/Shareddrives/phd_hands_on/data/titanic/test.csv')
```

In [ ]:
```
training.head()
```