Chapter 2 – End-to-end Machine Learning project

Welcome to Machine Learning Housing Corp.! Your task is to predict median house values in Californian districts, given a number of features from these districts.

This notebook contains all the sample code and solutions to the exercices in chapter 2.

[Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb)](https://colab.research.google.com/github/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

In [1]:
```
# 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 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)

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

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    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)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Get the data

In [2]:
```
import os
import tarfile
import urllib

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

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

In [3]:
```
fetch_housing_data()
```

In [4]:
```
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)
```

In [5]:
```
housing = load_housing_data()
housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	

In [6]:
```
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
```

In [7]:
```
housing["ocean_proximity"].value_counts()
```

Out[7]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

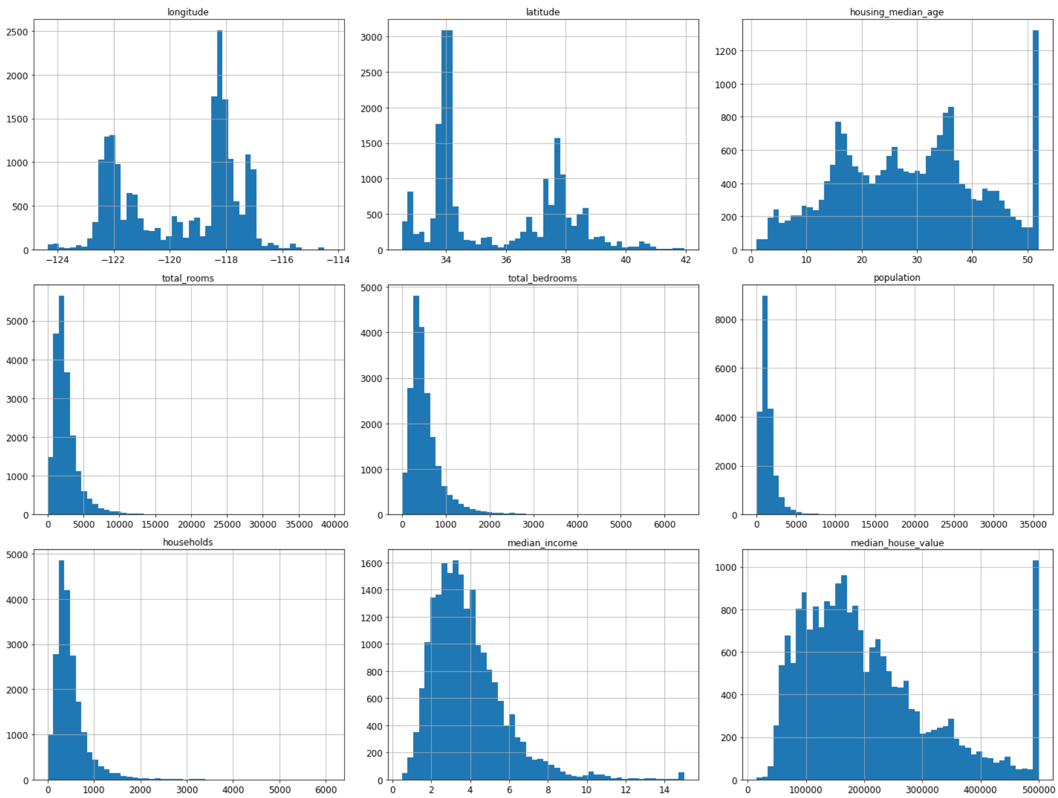
In [8]:
```
housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000

In [9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
```

```
Saving figure attribute_histogram_plots
```



In [10]:
```
# to make this notebook's output identical at every run
np.random.seed(42)
```

In [11]:
```
from sklearn.model_selection import train_test_split

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

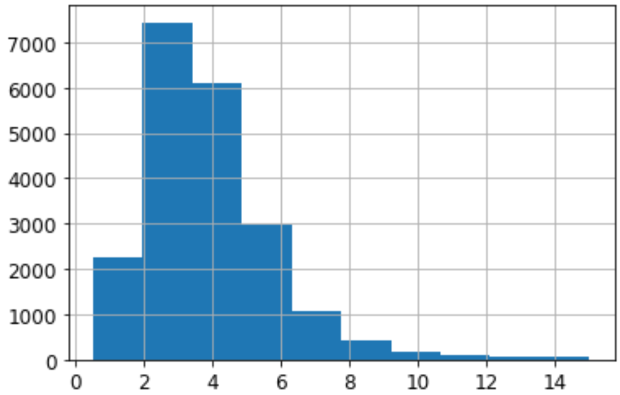
In [12]:
```
test_set.head()
```

Out[12]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
20046	-119.01	36.06	25.0	1505.0	NaN	1392.0	359.0	
3024	-119.46	35.14	30.0	2943.0	NaN	1565.0	584.0	
15663	-122.44	37.80	52.0	3830.0	NaN	1310.0	963.0	
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	
9814	-121.93	36.62	34.0	2351.0	NaN	1063.0	428.0	

In [12]:
```
housing["median_income"].hist()
```

Out[12]: <AxesSubplot:>



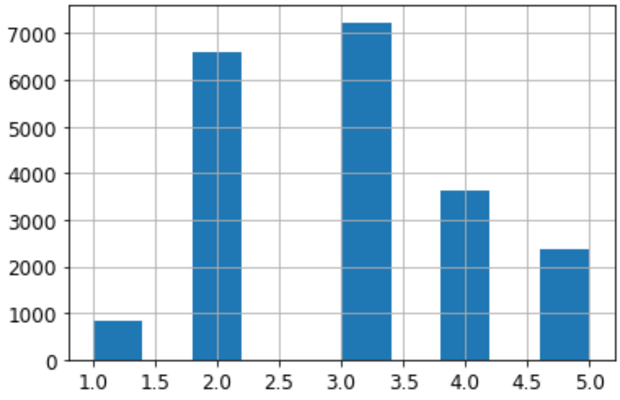
In [14]:
```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

In [15]:
```
housing["income_cat"].value_counts()
```

Out[15]:
```
3    7236
2    6581
4    3639
5    2362
1     822
Name: income_cat, dtype: int64
```

In [16]:
```
housing["income_cat"].hist()
```

Out[16]: <AxesSubplot:>



In [17]:
```
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 [18]:
```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

Out[18]:
```
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

In [19]:
```
housing["income_cat"].value_counts() / len(housing)
```

Out[19]:
```
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64
```

In [20]:
```
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_prop
s["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] / compare
_props["Overall"] - 100
```

In [21]:
```
compare_props
```

Out[21]:

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
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	0.114583	0.109496	-4.318374	0.127011

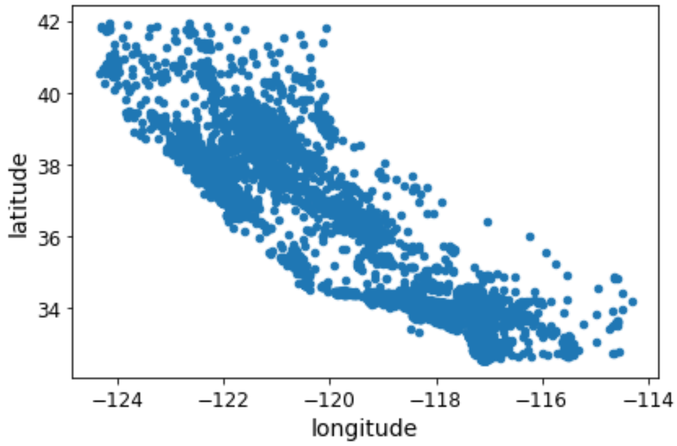
In [22]:
```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Discover and visualize the data to gain insights

In [23]:
```
housing = strat_train_set.copy()
```

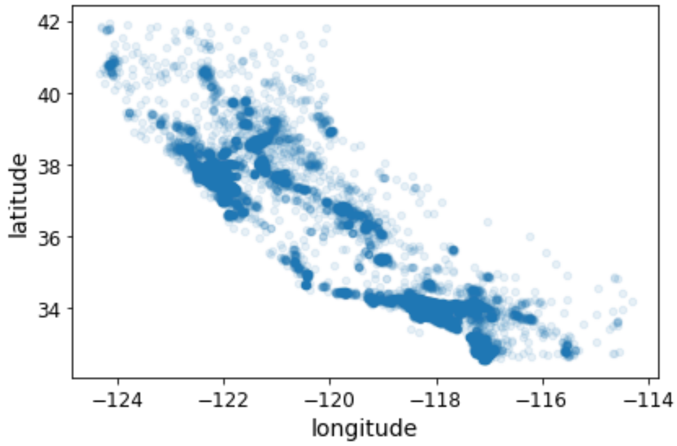
In [24]:
```
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

```
Saving figure bad_visualization_plot
```



In [25]:
```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

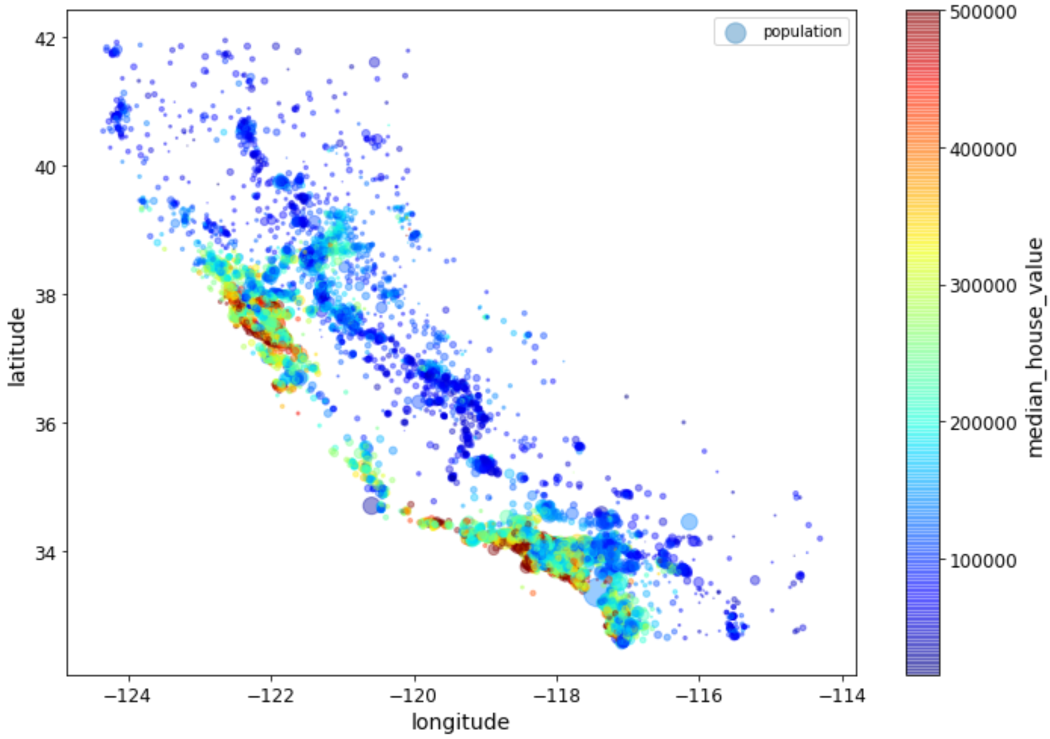
```
Saving figure better_visualization_plot
```



The argument `sharex=False` fixes a display bug (the x-axis values and legend were not displayed). This is a temporary fix (see: https://github.com/pandas-dev/pandas/issues/10611 (https://github.com/pandas-dev/pandas/issues/10611)). Thanks to Wilmer Arellano for pointing it out.

In [26]:
```
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")
```

```
Saving figure housing_prices_scatterplot
```



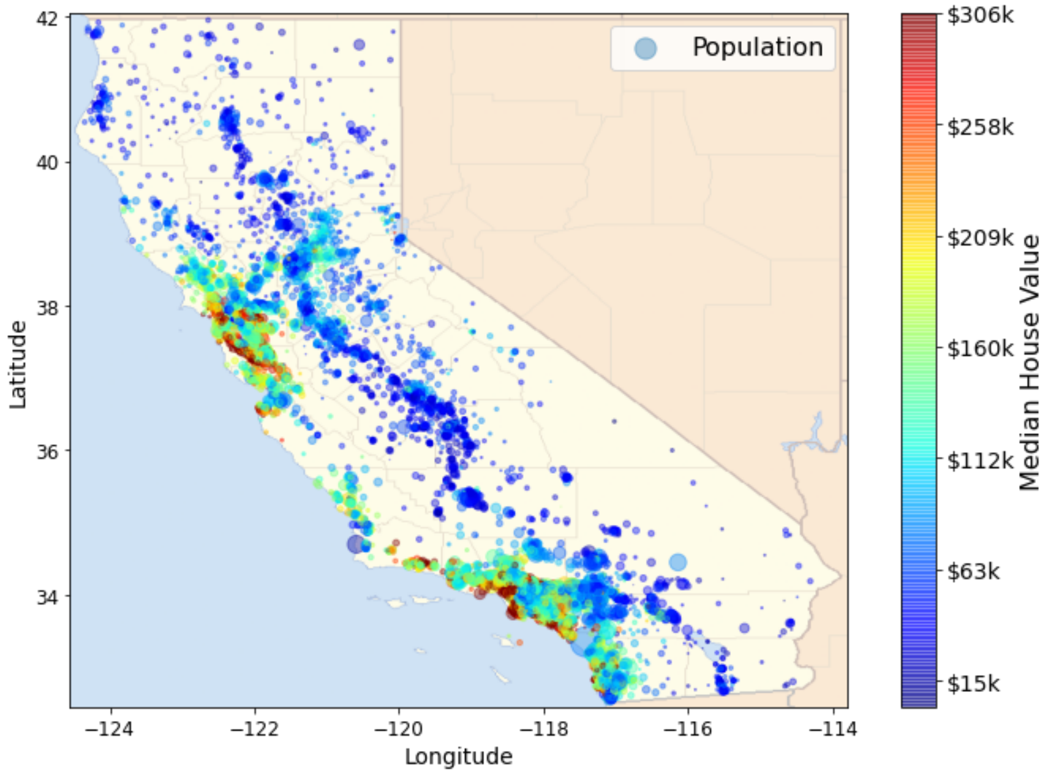
In [27]:
```
# Download the California image
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "end_to_end_project")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master
/"
filename = "california.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/end_to_end_project/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

```
Downloading california.png
```

Out[27]:
```
('.\\images\\end_to_end_project\\california.png',
 <http.client.HTTPMessage at 0x2703853d508>)
```

In [28]:
```
import matplotlib.image as mpimg
california_img=mpimg.imread(os.path.join(images_path, filename))
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,
7),
                       s=housing['population']/100, label="Population",
                       c="median_house_value", cmap=plt.get_cmap("jet"),
                       colorbar=False, alpha=0.4,
                      )
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.
5,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar()
cbar.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values], fonts
ize=14)
cbar.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

```
C:\Users\Tamy\Anaconda3\envs\tf\lib\site-packages\ipykernel_launcher.py:16: U
serWarning: FixedFormatter should only be used together with FixedLocator
  app.launch_new_instance()

Saving figure california_housing_prices_plot
```



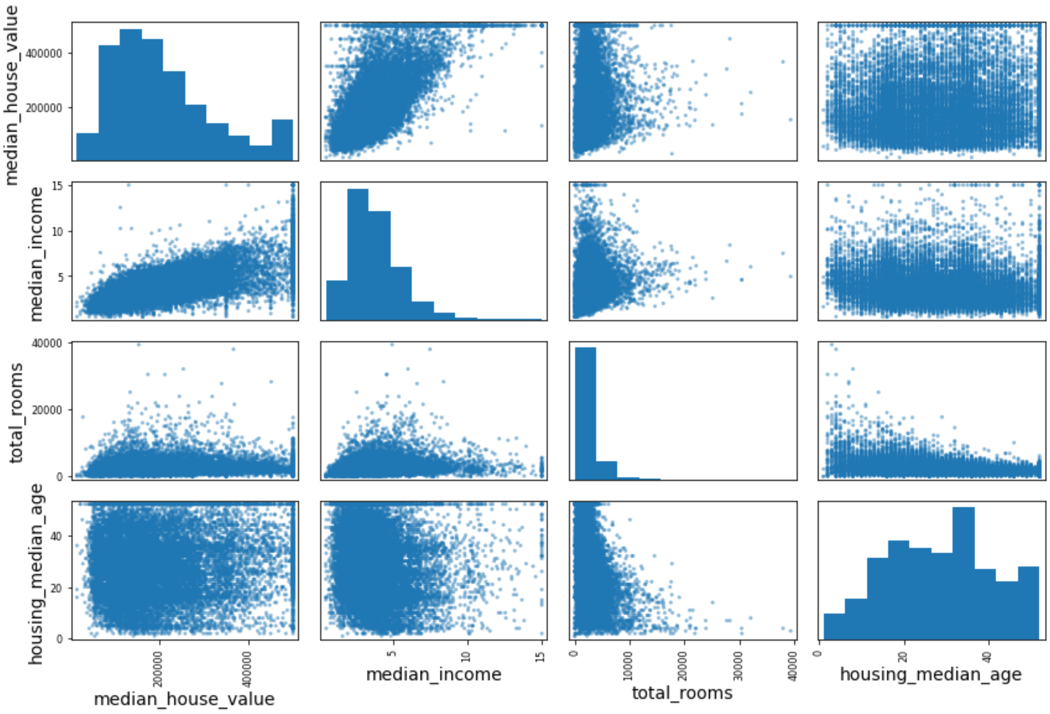
In [29]:
```
corr_matrix = housing.corr()
```

In [30]:
```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[30]:
```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

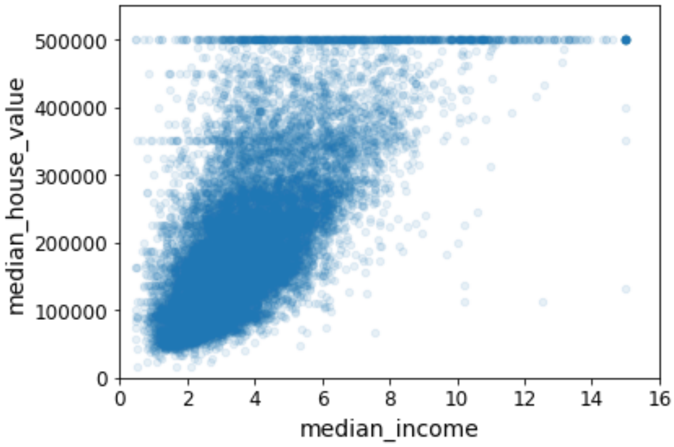
In [31]:
```
# from pandas.tools.plotting import scatter_matrix # For older versions of P
andas
from pandas.plotting import scatter_matrix

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

```
Saving figure scatter_matrix_plot
```



In [32]:
```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

```
Saving figure income_vs_house_value_scatterplot
```

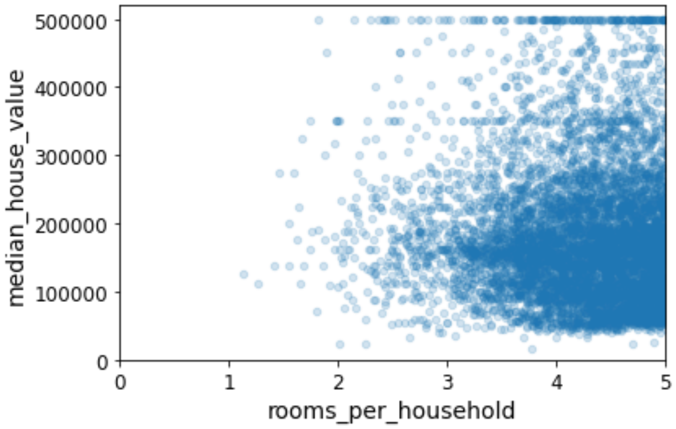


In [33]:
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

In [34]:
```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[34]:
```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

In [35]:
```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_valu
e",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



In [36]:
```
housing.describe()
```

Out[36]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	h
count	16512.000000	16512.000000	16512.000000	16512.000000	16354.000000	16512.000000	16
mean	-119.575834	35.639577	28.653101	2622.728319	534.973890	1419.790819	[illegible]
std	2.001860	2.138058	12.574726	2138.458419	412.699041	1115.686241	[illegible]
min	-124.350000	32.540000	1.000000	6.000000	2.000000	3.000000	
25%	-121.800000	33.940000	18.000000	1443.000000	295.000000	784.000000	[illegible]
50%	-118.510000	34.260000	29.000000	2119.500000	433.000000	1164.000000	[illegible]
75%	-118.010000	37.720000	37.000000	3141.000000	644.000000	1719.250000	[illegible]
max	-114.310000	41.950000	52.000000	39320.000000	6210.000000	35682.000000	5

Prepare the data for Machine Learning algorithms

In [37]:
```
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels f
or training set
housing_labels = strat_train_set["median_house_value"].copy()
```

In [38]:
```
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

Out[38]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
4629	-118.30	34.07	18.0	3759.0	NaN	3296.0	1462.0	
6068	-117.86	34.01	16.0	4632.0	NaN	3038.0	727.0	
17923	-121.97	37.35	30.0	1955.0	NaN	999.0	386.0	
13656	-117.30	34.05	6.0	2155.0	NaN	1039.0	391.0	
19252	-122.79	38.48	7.0	6837.0	NaN	3468.0	1405.0	

In [39]:
```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

Out[39]:

longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_

In [40]:
```
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

Out[40]:

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	oc
4629	-118.30	34.07	18.0	3759.0	3296.0	1462.0	2.2708	
6068	-117.86	34.01	16.0	4632.0	3038.0	727.0	5.1762	
17923	-121.97	37.35	30.0	1955.0	999.0	386.0	4.6328	
13656	-117.30	34.05	6.0	2155.0	1039.0	391.0	1.6675	
19252	-122.79	38.48	7.0	6837.0	3468.0	1405.0	3.1662	

In [41]:
```
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # opti
on 3
```

In [42]:
```
sample_incomplete_rows
```

Out[42]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0	1462.0	
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0	727.0	
17923	-121.97	37.35	30.0	1955.0	433.0	999.0	386.0	
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0	391.0	
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0	1405.0	

In [43]:
```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

Remove the text attribute because median can only be calculated on numerical attributes:

In [44]:
```
housing_num = housing.drop("ocean_proximity", axis=1)
# alternatively: housing_num = housing.select_dtypes(include=[np.number])
```

In [45]:
```
imputer.fit(housing_num)
```

Out[45]:
```
SimpleImputer(strategy='median')
```

In [46]:
```
imputer.statistics_
```

Out[46]:
```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

Check that this is the same as manually computing the median of each attribute:

In [47]:
```
housing_num.median().values
```

Out[47]:
```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

Transform the training set:

In [48]:
```
X = imputer.transform(housing_num)
```

In [49]:
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing.index)
```

In [50]:
```
housing_tr.loc[sample_incomplete_rows.index.values]
```

Out[50]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0	1462.0	
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0	727.0	
17923	-121.97	37.35	30.0	1955.0	433.0	999.0	386.0	
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0	391.0	
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0	1405.0	

In [51]:
```
imputer.strategy
```

Out[51]: 'median'

In [52]:
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

In [53]:
```
housing_tr.head()
```

Out[53]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	339.0	
18632	-121.93	37.05	14.0	679.0	108.0	306.0	113.0	
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	462.0	
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	353.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	1463.0	

Now let's preprocess the categorical input feature, `ocean_proximity` :

In [54]:
```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(10)
```

Out[54]:

	ocean_proximity
17606	<1H OCEAN
18632	<1H OCEAN
14650	NEAR OCEAN
3230	INLAND
3555	<1H OCEAN
19480	INLAND
8879	<1H OCEAN
13685	INLAND
4937	<1H OCEAN
4861	<1H OCEAN

In [55]:
```
from sklearn.preprocessing import OrdinalEncoder

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

Out[55]:
```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

In [56]:
```
ordinal_encoder.categories_
```

Out[56]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

In [57]:
```
from sklearn.preprocessing import OneHotEncoder

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

Out[57]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
	with 16512 stored elements in Compressed Sparse Row format>
```

By default, the `OneHotEncoder` class returns a sparse array, but we can convert it to a dense array if needed by calling the `toarray()` method:

In [58]:
```
housing_cat_1hot.toarray()
```

Out[58]:
```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

Alternatively, you can set `sparse=False` when creating the `OneHotEncoder` :

In [59]:
```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[59]:
```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

In [60]:
```
cat_encoder.categories_
```

Out[60]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

In [ ]:
```

```

Chapter 2 – End-to-end Machine Learning project

Welcome to Machine Learning Housing Corp.! Your task is to predict median house values in Californian districts, given a number of features from these districts.

This notebook contains all the sample code and solutions to the exercices in chapter 2.

[Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb)](https://colab.research.google.com/github/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

In [1]:

```
# 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 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)

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

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    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)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Get the data

In [2]:
```
import os
import tarfile
import urllib

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

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

In [3]:
```
fetch_housing_data()
```

In [4]:
```
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)
```

In [5]:
```
housing = load_housing_data()
housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	

In [6]:
```
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
```

In [7]:
```
housing["ocean_proximity"].value_counts()
```

Out[7]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

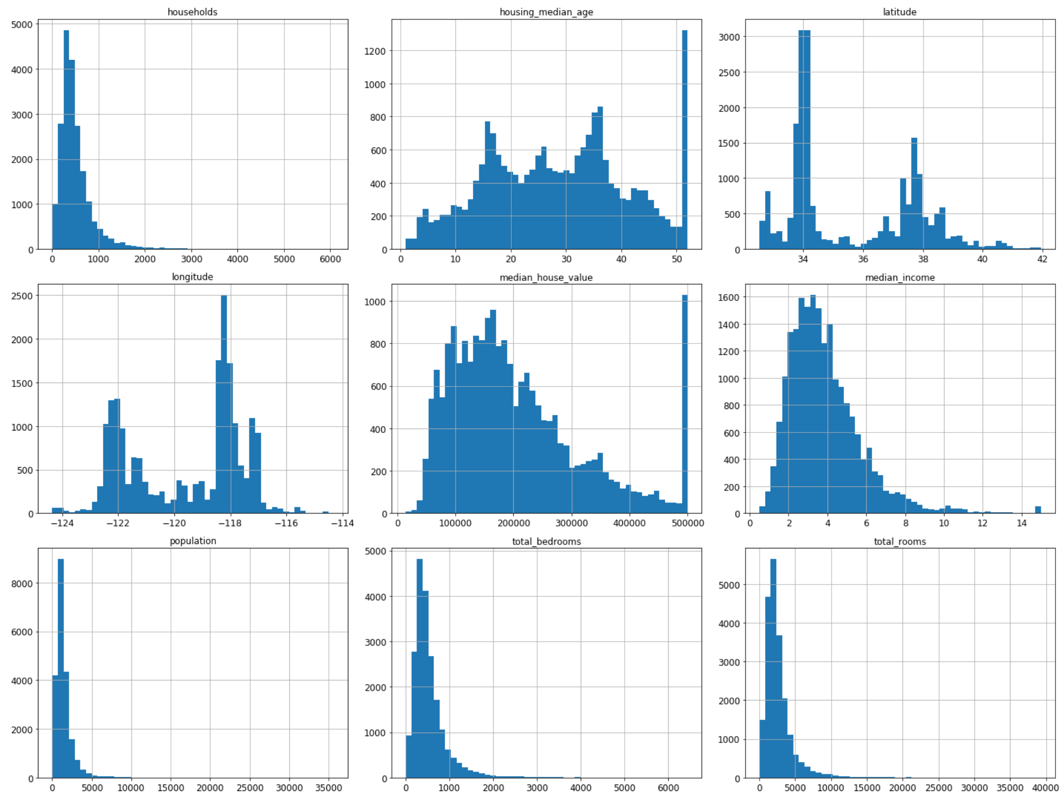
In [8]:
```
housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	h
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	4
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	3
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	2
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	4
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	(
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6(

In [9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
```

```
Saving figure attribute_histogram_plots
```



In [10]:
```
# to make this notebook's output identical at every run
np.random.seed(42)
```

In [11]:
```
from sklearn.model_selection import train_test_split

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

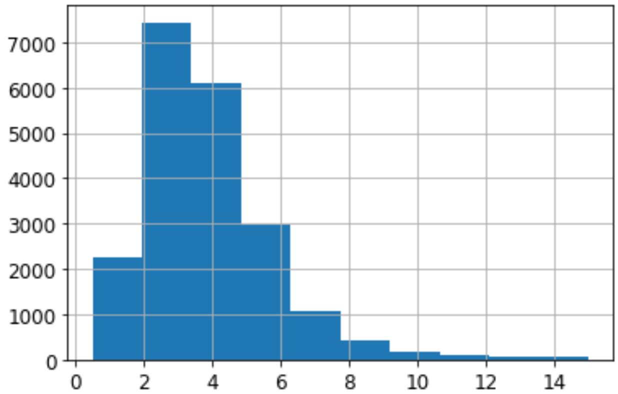
In [12]:
```
test_set.head()
```

Out[12]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
20046	-119.01	36.06	25.0	1505.0	NaN	1392.0	359.0	
3024	-119.46	35.14	30.0	2943.0	NaN	1565.0	584.0	
15663	-122.44	37.80	52.0	3830.0	NaN	1310.0	963.0	
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	
9814	-121.93	36.62	34.0	2351.0	NaN	1063.0	428.0	

In [13]:
```
housing["median_income"].hist()
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x1dd40d8bd08>



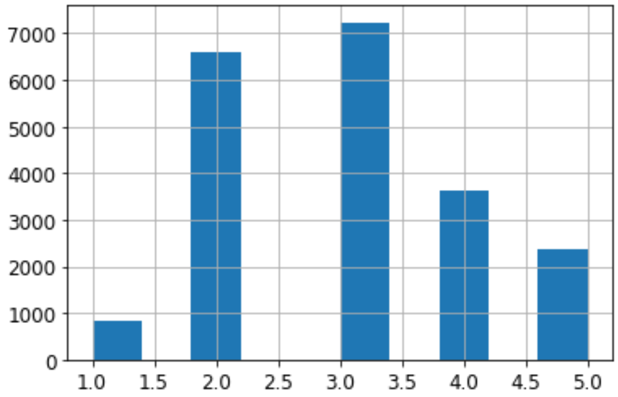
In [14]:
```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

In [15]:
```
housing["income_cat"].value_counts()
```

Out[15]:
```
3    7236
2    6581
4    3639
5    2362
1     822
Name: income_cat, dtype: int64
```

In [16]:
```
housing["income_cat"].hist()
```

Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x1dd40f56c48>



In [17]:
```
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 [18]:
```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

Out[18]:
```
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

In [19]:
```
housing["income_cat"].value_counts() / len(housing)
```

Out[19]:
```
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64
```

In [20]:
```
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_prop
s["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] / compare
_props["Overall"] - 100
```

In [21]:
```
compare_props
```

Out[21]:

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
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	0.114583	0.109496	-4.318374	0.127011

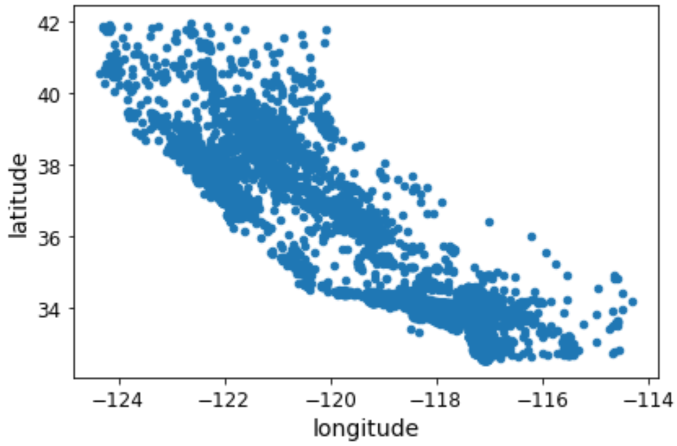
In [22]:
```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Discover and visualize the data to gain insights

In [23]:
```
housing = strat_train_set.copy()
```

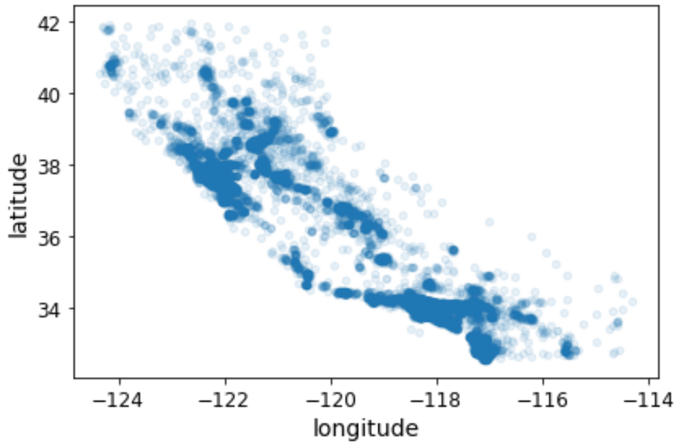
In [24]:
```
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

```
Saving figure bad_visualization_plot
```



In [25]:
```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

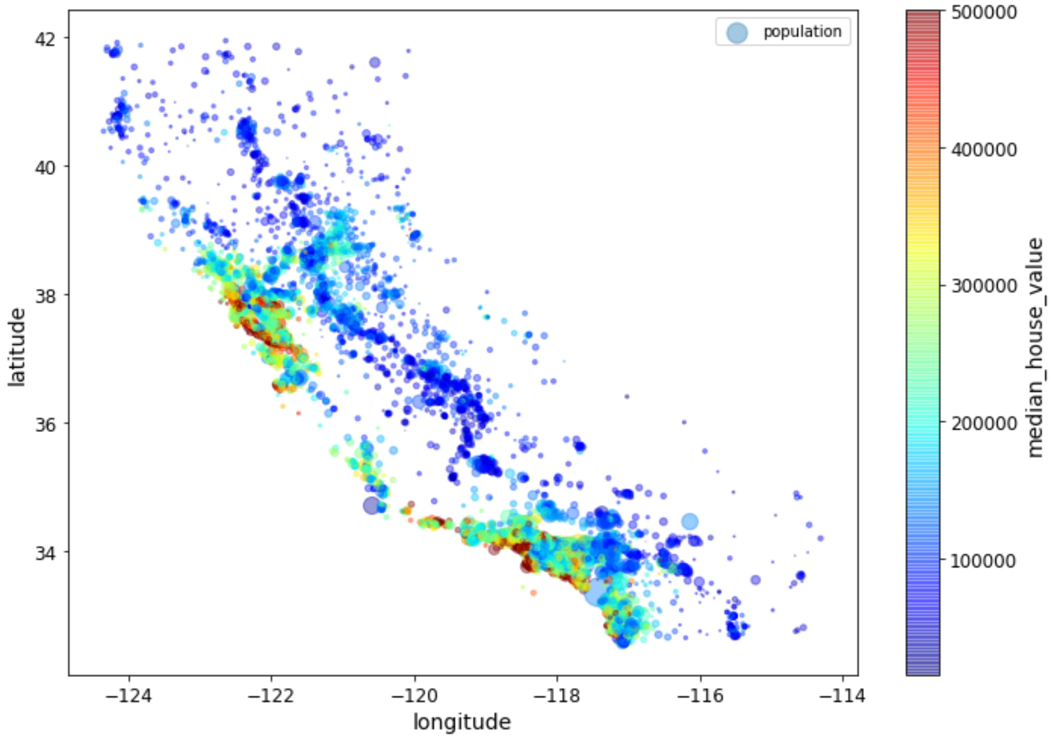
```
Saving figure better_visualization_plot
```



The argument `sharex=False` fixes a display bug (the x-axis values and legend were not displayed). This is a temporary fix (see: [https://github.com/pandas-dev/pandas/issues/10611 (https://github.com/pandas-dev/pandas/issues/10611)](https://github.com/pandas-dev/pandas/issues/10611)). Thanks to Wilmer Arellano for pointing it out.

In [26]:
```
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")
```

```
Saving figure housing_prices_scatterplot
```



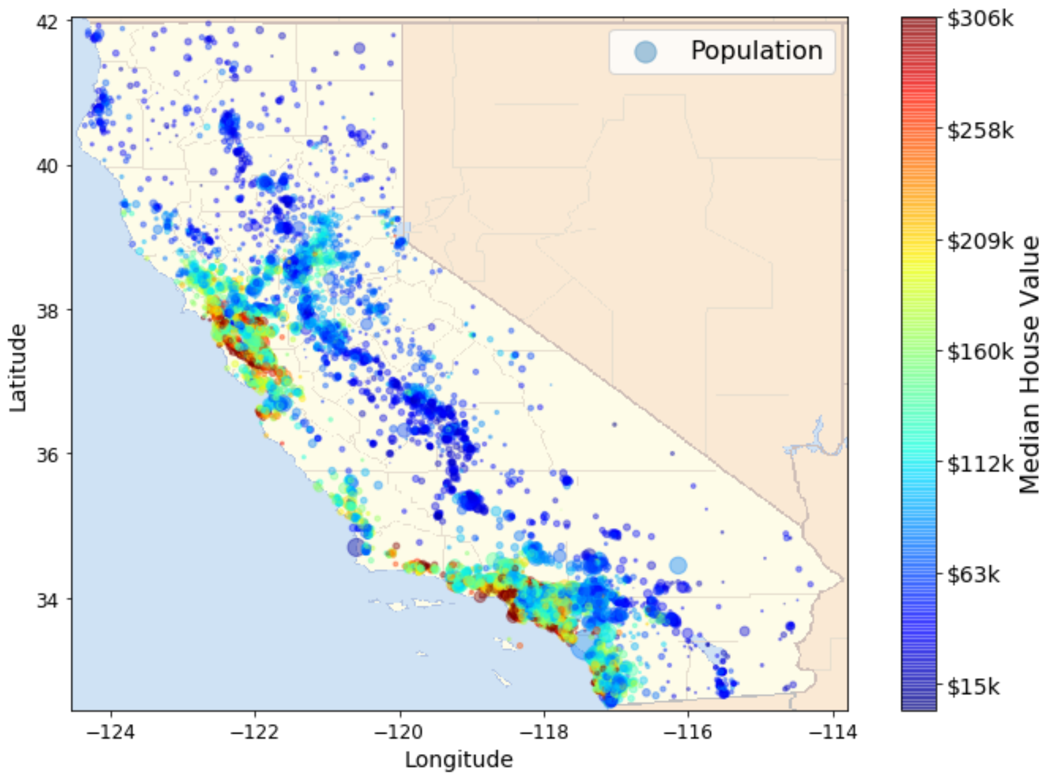
In [27]:
```
# Download the California image
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "end_to_end_project")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master
/"
filename = "california.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/end_to_end_project/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

```
Downloading california.png
```

Out[27]:
```
('.\\images\\end_to_end_project\\california.png',
 <http.client.HTTPMessage at 0x1dd41e1d408>)
```

In [28]:
```
import matplotlib.image as mpimg
california_img=mpimg.imread(os.path.join(images_path, filename))
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,
7),
                       s=housing['population']/100, label="Population",
                       c="median_house_value", cmap=plt.get_cmap("jet"),
                       colorbar=False, alpha=0.4,
                      )
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.
5,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar()
cbar.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values], fonts
ize=14)
cbar.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

```
Saving figure california_housing_prices_plot
```



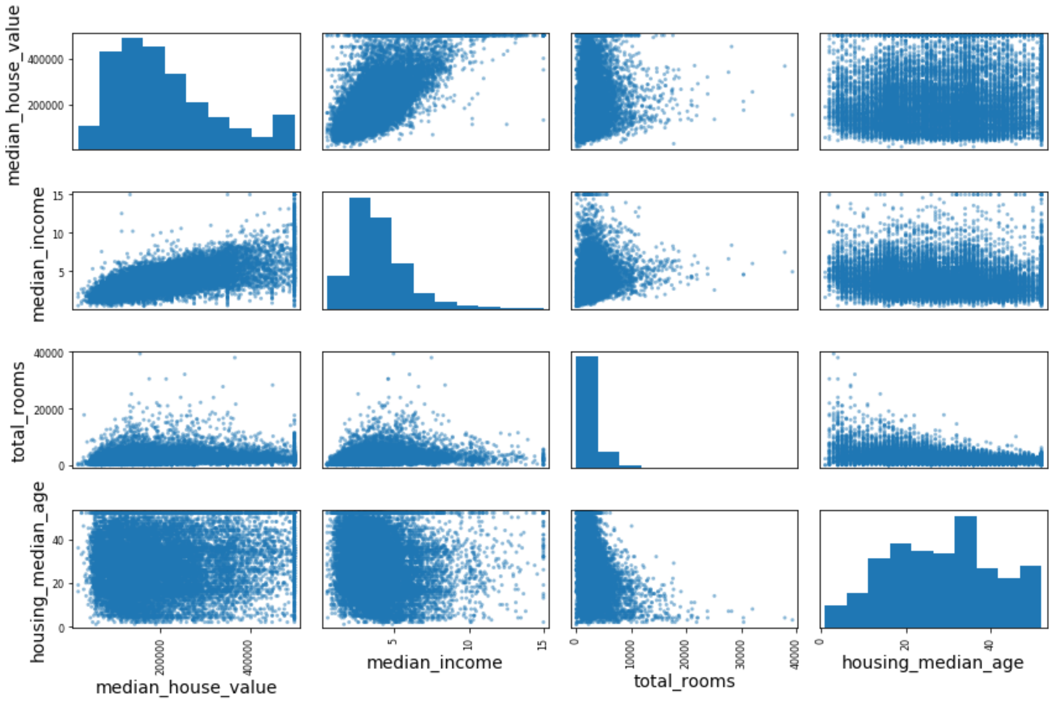
In [29]:
```
corr_matrix = housing.corr()
```

In [30]:
```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[30]:
```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

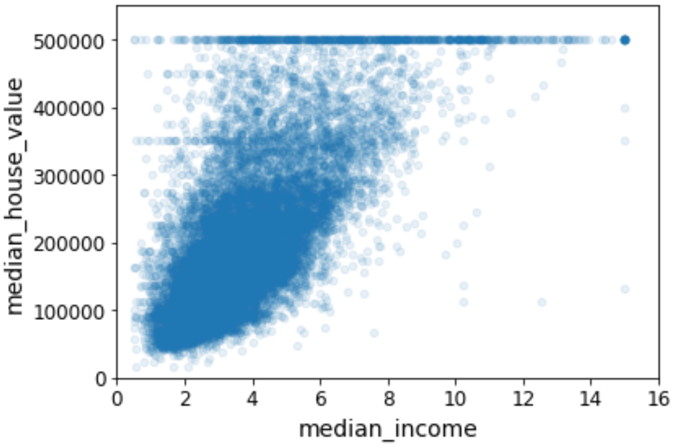
In [31]:
```
# from pandas.tools.plotting import scatter_matrix # For older versions of P
andas
from pandas.plotting import scatter_matrix

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

```
Saving figure scatter_matrix_plot
```



In [32]:
```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

```
Saving figure income_vs_house_value_scatterplot
```

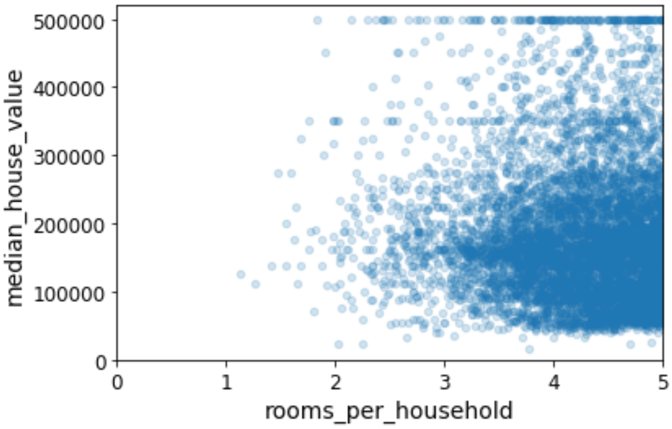


In [33]:
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["household
s"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_room
s"]
housing["population_per_household"]=housing["population"]/housing["household
s"]
```

In [34]:
```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[34]:
```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

In [35]:
```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_valu
e",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



In [36]:
```
housing.describe()
```

Out[36]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	h
count	16512.000000	16512.000000	16512.000000	16512.000000	16354.000000	16512.000000	16!
mean	-119.575834	35.639577	28.653101	2622.728319	534.973890	1419.790819	4
std	2.001860	2.138058	12.574726	2138.458419	412.699041	1115.686241	:
min	-124.350000	32.540000	1.000000	6.000000	2.000000	3.000000	
25%	-121.800000	33.940000	18.000000	1443.000000	295.000000	784.000000	:
50%	-118.510000	34.260000	29.000000	2119.500000	433.000000	1164.000000	4
75%	-118.010000	37.720000	37.000000	3141.000000	644.000000	1719.250000	(
max	-114.310000	41.950000	52.000000	39320.000000	6210.000000	35682.000000	5:

Prepare the data for Machine Learning algorithms

In [37]:
```
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels f
or training set
housing_labels = strat_train_set["median_house_value"].copy()
```

In [38]:
```
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

Out[38]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
4629	-118.30	34.07	18.0	3759.0	NaN	3296.0	1462.0	
6068	-117.86	34.01	16.0	4632.0	NaN	3038.0	727.0	
17923	-121.97	37.35	30.0	1955.0	NaN	999.0	386.0	
13656	-117.30	34.05	6.0	2155.0	NaN	1039.0	391.0	
19252	-122.79	38.48	7.0	6837.0	NaN	3468.0	1405.0	

In [39]:
```python
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

Out[39]:

longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_

In [40]:
```python
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

Out[40]:

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	oc
4629	-118.30	34.07	18.0	3759.0	3296.0	1462.0	2.2708	
6068	-117.86	34.01	16.0	4632.0	3038.0	727.0	5.1762	
17923	-121.97	37.35	30.0	1955.0	999.0	386.0	4.6328	
13656	-117.30	34.05	6.0	2155.0	1039.0	391.0	1.6675	
19252	-122.79	38.48	7.0	6837.0	3468.0	1405.0	3.1662	

In [41]:
```python
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # opti
on 3
```

In [42]:
```python
sample_incomplete_rows
```

Out[42]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0	1462.0	
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0	727.0	
17923	-121.97	37.35	30.0	1955.0	433.0	999.0	386.0	
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0	391.0	
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0	1405.0	

In [43]:
```python
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

Remove the text attribute because median can only be calculated on numerical attributes:

In [44]:
```python
housing_num = housing.drop("ocean_proximity", axis=1)
# alternatively: housing_num = housing.select_dtypes(include=[np.number])
```

In [45]:
```python
imputer.fit(housing_num)
```

Out[45]:
```
SimpleImputer(strategy='median')
```

In [46]:
```python
imputer.statistics_
```

Out[46]:
```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

Check that this is the same as manually computing the median of each attribute:

In [47]:
```python
housing_num.median().values
```

Out[47]:
```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

Transform the training set:

In [48]:
```
X = imputer.transform(housing_num)
```

In [49]:
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing.index)
```

In [50]:
```
housing_tr.loc[sample_incomplete_rows.index.values]
```

Out[50]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0	1462.0	
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0	727.0	
17923	-121.97	37.35	30.0	1955.0	433.0	999.0	386.0	
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0	391.0	
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0	1405.0	

In [51]:
```
imputer.strategy
```

Out[51]: 'median'

In [52]:
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

In [53]:
```
housing_tr.head()
```

Out[53]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	339.0	
18632	-121.93	37.05	14.0	679.0	108.0	306.0	113.0	
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	462.0	
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	353.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	1463.0	

Now let's preprocess the categorical input feature, `ocean_proximity` :

In [54]:
```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(10)
```

Out[54]:

	ocean_proximity
17606	<1H OCEAN
18632	<1H OCEAN
14650	NEAR OCEAN
3230	INLAND
3555	<1H OCEAN
19480	INLAND
8879	<1H OCEAN
13685	INLAND
4937	<1H OCEAN
4861	<1H OCEAN

In [55]:
```
from sklearn.preprocessing import OrdinalEncoder

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

Out[55]:
```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

In [56]:
```
ordinal_encoder.categories_
```

Out[56]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

In [57]:
```
from sklearn.preprocessing import OneHotEncoder

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

Out[57]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
	with 16512 stored elements in Compressed Sparse Row format>
```

By default, the `OneHotEncoder` class returns a sparse array, but we can convert it to a dense array if needed by calling the `toarray()` method:

In [58]:
```
housing_cat_1hot.toarray()
```

Out[58]:
```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

Alternatively, you can set `sparse=False` when creating the `OneHotEncoder`:

In [59]:
```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[59]:
```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

In [60]:
```
cat_encoder.categories_
```

Out[60]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Let's create a custom transformer to add extra attributes:

In [61]:
```
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
    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)
```

In [62]:
```
housing_extra_attribs = pd.DataFrame(
    housing_extra_attribs,
    columns=list(housing.columns)+["rooms_per_household", "population_per_ho
usehold"],
    index=housing.index)
housing_extra_attribs.head()
```

Out[62]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	me
17606	-121.89	37.29	38	1568	351	710	339	
18632	-121.93	37.05	14	679	108	306	113	
14650	-117.2	32.77	31	1952	471	936	462	
3230	-119.61	36.31	25	1847	371	1460	353	
3555	-118.59	34.23	17	6592	1525	4459	1463	

Now let's build a pipeline for preprocessing the numerical attributes:

In [63]:
```
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)
```

In [64]:
```
housing_num_tr
```

Out[64]:
```
array([[-1.15604281,  0.77194962,  0.74333089, ..., -0.31205452,
         -0.08649871,  0.15531753],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.21768338,
         -0.03353391, -0.83628902],
       [ 1.18684903, -1.34218285,  0.18664186, ..., -0.46531516,
         -0.09240499,  0.4222004 ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.3469342 ,
         -0.03055414, -0.52177644],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.02499488,
          0.06150916, -0.30340741],
       [-1.43579109,  0.99645926,  1.85670895, ..., -0.22852947,
         -0.09586294,  0.10180567]])
```

In [65]:
```
from sklearn.compose import ColumnTransformer

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 [66]:
```
housing_prepared
```

Out[66]:
```
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
         0.        ,  0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, ...,  0.        ,
         0.        ,  1.        ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.        ,
         0.        ,  0.        ],
       [-1.43579109,  0.99645926,  1.85670895, ...,  0.        ,
         1.        ,  0.        ]])
```

In [67]:
```
housing_prepared.shape
```

Out[67]:
```
(16512, 16)
```

Select and train a model

In [68]:
```
from sklearn.linear_model import LinearRegression

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

Out[68]:
```
LinearRegression()
```

In [69]:
```
# 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: [210644.60459286 317768.80697211 210956.43331178  59218.98886849
 189747.55849879]
```

Compare against the actual values:

In [70]:
```
print("Labels:", list(some_labels))
```

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

In [71]:
```
some_data_prepared
```

Out[71]:
```
array([[-1.15604281,  0.77194962,  0.74333089, -0.49323393, -0.44543821,
        -0.63621141, -0.42069842, -0.61493744, -0.31205452, -0.08649871,
         0.15531753,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , -0.90896655, -1.0369278 ,
        -0.99833135, -1.02222705,  1.33645936,  0.21768338, -0.03353391,
        -0.83628902,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, -0.31365989, -0.15334458,
        -0.43363936, -0.0933178 , -0.5320456 , -0.46531516, -0.09240499,
         0.4222004 ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ],
       [-0.01706767,  0.31357576, -0.29052016, -0.36276217, -0.39675594,
         0.03604096, -0.38343559, -1.04556555, -0.07966124,  0.08973561,
        -0.19645314,  0.        ,  1.        ,  0.        ,  0.        ,
         0.        ],
       [ 0.49247384, -0.65929936, -0.92673619,  1.85619316,  2.41221109,
         2.72415407,  2.57097492, -0.44143679, -0.35783383, -0.00419445,
         0.2699277 ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ]])
```

In [72]:
```
from sklearn.metrics import mean_squared_error

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

Out[72]:
```
68628.19819848923
```

In [73]:
```
from sklearn.metrics import mean_absolute_error

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

Out[73]:
```
49439.89599001897
```

Fine-tune your model

In [74]:
```
from sklearn.tree import DecisionTreeRegressor

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

Out[74]:
```
DecisionTreeRegressor(random_state=42)
```

In [75]:
```
from sklearn.ensemble import RandomForestRegressor

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

Out[75]:
```
RandomForestRegressor(random_state=42)
```

In [76]:
```
from sklearn.svm import SVR

svm_reg = SVR(kernel="linear")
svm_reg.fit(housing_prepared, housing_labels)
```

Out[76]:
```
SVR(kernel='linear')
```

In [77]:
```
from sklearn.model_selection import cross_val_score

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)
pd.Series(lin_rmse_scores).describe()
```

Out[77]:
```
count       10.000000
mean     69052.461363
std       2879.437224
min      64969.630564
25%      67136.363758
50%      68156.372635
75%      70982.369487
max      74739.570526
dtype: float64
```

In [78]:
```
tree_scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                              scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-tree_scores)
pd.Series(tree_rmse_scores).describe()
```

Out[78]:
```
count       10.000000
mean     71407.687660
std       2571.389745
min      66855.163639
25%      70265.554176
50%      70937.310637
75%      72132.351151
max      75585.141729
dtype: float64
```

Note: we specify `n_estimators=100` to be future-proof since the default value is going to change to 100 in Scikit-Learn 0.22 (for simplicity, this is not shown in the book).

In [79]:
```
forest_scores = cross_val_score(forest_reg, housing_prepared, housing_label
s,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
pd.Series(forest_rmse_scores).describe()
```

Out[79]:
```
count       10.000000
mean     50182.303100
std       2210.517524
min      47461.911582
25%      48803.201309
50%      49770.694467
75%      51751.217424
max      53490.106998
dtype: float64
```

In [80]:
```
svm_scores = cross_val_score(svm_reg, housing_prepared, housing_labels,
                             scoring="neg_mean_squared_error", cv=10)
svm_rmse_scores = np.sqrt(-svm_scores)
pd.Series(svm_rmse_scores).describe()
```

Out[80]:
```
count        10.000000
mean     111809.840096
std        2911.818591
min      105342.091420
25%      110655.068116
50%      112004.679161
75%      113667.942015
max      115675.832002
dtype: float64
```

In [81]:
```
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)
```

Out[81]:
```
GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             return_train_score=True, scoring='neg_mean_squared_error')
```

The best hyperparameter combination found:

In [82]:
```
grid_search.best_params_
```

Out[82]:
```
{'max_features': 8, 'n_estimators': 30}
```

In [83]:
```
grid_search.best_estimator_
```

Out[83]:
```
RandomForestRegressor(max_features=8, n_estimators=30, random_state=42)
```

Let's look at the score of each hyperparameter combination tested during the grid search:

In [84]:
```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
63669.11631261028 {'max_features': 2, 'n_estimators': 3}
55627.099719926795 {'max_features': 2, 'n_estimators': 10}
53384.57275149205 {'max_features': 2, 'n_estimators': 30}
60965.950449450494 {'max_features': 4, 'n_estimators': 3}
52741.04704299915 {'max_features': 4, 'n_estimators': 10}
50377.40461678399 {'max_features': 4, 'n_estimators': 30}
58663.93866579625 {'max_features': 6, 'n_estimators': 3}
52006.19873526564 {'max_features': 6, 'n_estimators': 10}
50146.51167415009 {'max_features': 6, 'n_estimators': 30}
57869.25276169646 {'max_features': 8, 'n_estimators': 3}
51711.127883959234 {'max_features': 8, 'n_estimators': 10}
49682.273345071546 {'max_features': 8, 'n_estimators': 30}
62895.06951262424 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.176157539405 {'bootstrap': False, 'max_features': 2, 'n_estimators': 1
0}
59470.40652318466 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52724.9822587892 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.5691951261 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.495668875716 {'bootstrap': False, 'max_features': 4, 'n_estimators': 1
0}
```

In [85]:
```
pd.DataFrame(grid_search.cv_results_)
```

Out[85]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_features	param_n_estimator
0	0.060681	0.001426	0.002983	0.000037	2	
1	0.194322	0.003589	0.008981	0.000028	2	1
2	0.583491	0.005095	0.025140	0.000773	2	3
3	0.093365	0.002416	0.003011	0.000023	4	
4	0.302422	0.002909	0.008978	0.000047	4	1
5	0.963829	0.043499	0.026337	0.000795	4	3
6	0.125465	0.002487	0.003011	0.000019	6	
7	0.422492	0.007103	0.009374	0.000476	6	1
8	1.309725	0.022151	0.025704	0.000436	6	3
9	0.175359	0.009939	0.003371	0.000525	8	
10	0.630333	0.043427	0.010558	0.000819	8	1
11	1.801205	0.017485	0.028310	0.000492	8	3
12	0.105512	0.001595	0.003991	0.000018	2	
13	0.378807	0.027508	0.013155	0.001940	2	1
14	0.169348	0.005897	0.004988	0.000631	3	

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_features	param_n_estimator
15	0.537342	0.051332	0.012968	0.001538	3	1
16	0.191867	0.014059	0.004980	0.000628	4	

In [86]:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distri
bs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_e
rror', random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

Out[86]:

```
RandomizedSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
                   param_distributions={'max_features': <scipy.stats._distn_i
nfrastructure.rv_frozen object at 0x000001DD43194548>,
                                        'n_estimators': <scipy.stats._distn_i
nfrastructure.rv_frozen object at 0x000001DD43194B48>},
                   random_state=42, scoring='neg_mean_squared_error')
```

In [87]:

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

```
49150.70756927707 {'max_features': 7, 'n_estimators': 180}
51389.889203389284 {'max_features': 5, 'n_estimators': 15}
50796.155224308866 {'max_features': 3, 'n_estimators': 72}
50835.13360315349 {'max_features': 5, 'n_estimators': 21}
49280.9449827171 {'max_features': 7, 'n_estimators': 122}
50774.90662363929 {'max_features': 3, 'n_estimators': 75}
50682.78888164288 {'max_features': 3, 'n_estimators': 88}
49608.99608105296 {'max_features': 5, 'n_estimators': 100}
50473.61930350219 {'max_features': 3, 'n_estimators': 150}
64429.84143294435 {'max_features': 5, 'n_estimators': 2}
```

In [88]:

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

Out[88]:

```
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```

In [89]:
```
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)
```

Out[89]:
```
[(0.36615898061813423, 'median_income'),
 (0.16478099356159054, 'INLAND'),
 (0.10879295677551575, 'pop_per_hhold'),
 (0.07334423551601243, 'longitude'),
 (0.06290907048262032, 'latitude'),
 (0.056419179181954014, 'rooms_per_hhold'),
 (0.053351077347675815, 'bedrooms_per_room'),
 (0.04114379847872964, 'housing_median_age'),
 (0.014874280890402769, 'population'),
 (0.014672685420543239, 'total_rooms'),
 (0.014257599323407808, 'households'),
 (0.014106483453584104, 'total_bedrooms'),
 (0.010311488326303788, '<1H OCEAN'),
 (0.0028564746373201584, 'NEAR OCEAN'),
 (0.0019604155994780706, 'NEAR BAY'),
 (6.0280386727366e-05, 'ISLAND')]
```

In [90]:
```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)

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

In [91]:
```
final_rmse
```

Out[91]:
```
47730.22690385927
```

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

In [92]:
```
from scipy import stats

confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

Out[92]:
```
array([45685.10470776, 49691.25001878])
```

Congratulations! You already know quite a lot about Machine Learning. :)

In [ ]:
```

```

Chapter 8 – Dimensionality Reduction

This notebook contains all the sample code and solutions to the exercises in chapter 8.

[Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/08_dimensionality_reduction.ipynb)](https://colab.research.google.com/github/ageron/handson-ml2/blob/master/08_dimensionality_reduction.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

In [1]:
```
# 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 os

# to make this notebook's output stable across runs
np.random.seed(42)

# 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)

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

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    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)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Projection methods

Build 3D dataset:

In [2]:
```
np.random.seed(4)
m = 60
w1, w2 = 0.1, 0.3
noise = 0.1

angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
X = np.empty((m, 3))
X[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(m) / 2
X[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
X[:, 2] = X[:, 0] * w1 + X[:, 1] * w2 + noise * np.random.randn(m)
```

PCA using Scikit-Learn

With Scikit-Learn, PCA is really trivial. It even takes care of mean centering for you:

In [3]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

In [4]:
```
X2D[:5]
```

Out[4]:
```
array([[ 1.26203346,  0.42067648],
       [-0.08001485, -0.35272239],
       [ 1.17545763,  0.36085729],
       [ 0.89305601, -0.30862856],
       [ 0.73016287, -0.25404049]])
```

Notice that running PCA multiple times on slightly different datasets may result in different results. In general the only difference is that some axes may be flipped.

Recover the 3D points projected on the plane (PCA 2D subspace).

In [5]:
```
X3D_inv = pca.inverse_transform(X2D)
```

Of course, there was some loss of information during the projection step, so the recovered 3D points are not exactly equal to the original 3D points:

In [6]:
```
np.allclose(X3D_inv, X)
```

Out[6]:
```
False
```

We can compute the reconstruction error:

In [7]:
```
np.mean(np.sum(np.square(X3D_inv - X), axis=1))
```

Out[7]:
```
0.010170337792848549
```

The PCA object gives access to the principal components that it computed:

In [8]:
```
pca.components_
```

Out[8]:
```
array([[-0.93636116, -0.29854881, -0.18465208],
       [ 0.34027485, -0.90119108, -0.2684542 ]])
```

Notice how the axes are flipped.

Now let's look at the explained variance ratio:

In [9]:
```
pca.explained_variance_ratio_
```

Out[9]:
```
array([0.84248607, 0.14631839])
```

The first dimension explains 84.2% of the variance, while the second explains 14.6%.

By projecting down to 2D, we lost about 1.1% of the variance:

In [10]:
```
1 - pca.explained_variance_ratio_.sum()
```

Out[10]:
```
0.011195535570688975
```

Next, let's generate some nice figures! :)

Utility class to draw 3D arrows (copied from [http://stackoverflow.com/questions/11140163 (http://stackoverflow.com/questions/11140163)](http://stackoverflow.com/questions/11140163))

In [11]:
```
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d

class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)
```

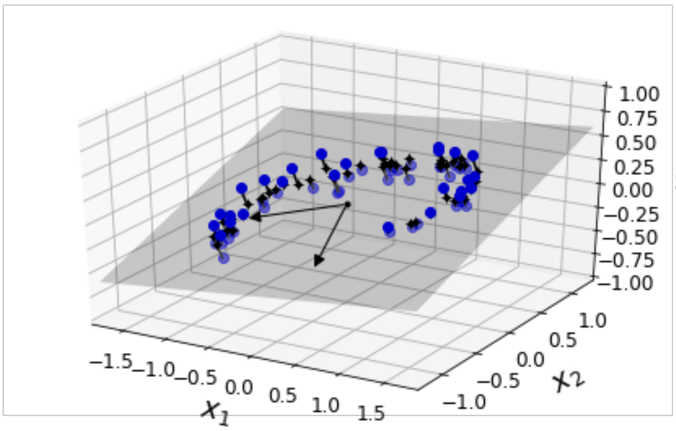
Express the plane as a function of x and y.

In [12]:
```
axes = [-1.8, 1.8, -1.3, 1.3, -1.0, 1.0]

x1s = np.linspace(axes[0], axes[1], 10)
x2s = np.linspace(axes[2], axes[3], 10)
x1, x2 = np.meshgrid(x1s, x2s)

C = pca.components_
R = C.T.dot(C)
z = (R[0, 2] * x1 + R[1, 2] * x2) / (1 - R[2, 2])
```

Plot the 3D dataset, the plane and the projections on that plane.

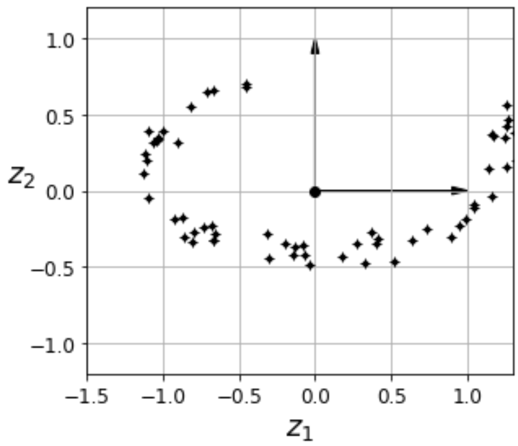
In [13]:
```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(6, 3.8))
ax = fig.add_subplot(111, projection='3d')

X3D_above = X[X[:, 2] > X3D_inv[:, 2]]
X3D_below = X[X[:, 2] <= X3D_inv[:, 2]]

ax.plot(X3D_below[:, 0], X3D_below[:, 1], X3D_below[:, 2], "bo", alpha=0.5)

ax.plot_surface(x1, x2, z, alpha=0.2, color="k")
np.linalg.norm(C, axis=0)
ax.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]], mutation_scale
=15, lw=1, arrowstyle="-|>", color="k"))
ax.add_artist(Arrow3D([0, C[1, 0]],[0, C[1, 1]],[0, C[1, 2]], mutation_scale
=15, lw=1, arrowstyle="-|>", color="k"))
ax.plot([0], [0], [0], "k.")

for i in range(m):
    if X[i, 2] > X3D_inv[i, 2]:
        ax.plot([X[i][0], X3D_inv[i][0]], [X[i][1], X3D_inv[i][1]], [X
[i][2], X3D_inv[i][2]], "k-")
    else:
        ax.plot([X[i][0], X3D_inv[i][0]], [X[i][1], X3D_inv[i][1]], [X
[i][2], X3D_inv[i][2]], "k-", color="#505050")

ax.plot(X3D_inv[:, 0], X3D_inv[:, 1], X3D_inv[:, 2], "k+")
ax.plot(X3D_inv[:, 0], X3D_inv[:, 1], X3D_inv[:, 2], "k.")
ax.plot(X3D_above[:, 0], X3D_above[:, 1], X3D_above[:, 2], "bo")
ax.set_xlabel("$x_1$", fontsize=18, labelpad=10)
ax.set_ylabel("$x_2$", fontsize=18, labelpad=10)
ax.set_zlabel("$x_3$", fontsize=18, labelpad=10)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

# Note: If you are using Matplotlib 3.0.0, it has a bug and does not
# display 3D graphs properly.
# See https://github.com/matplotlib/matplotlib/issues/12239
# You should upgrade to a later version. If you cannot, then you can
# use the following workaround before displaying each 3D graph:
# for spine in ax.spines.values():
#     spine.set_visible(False)

save_fig("dataset_3d_plot")
plt.show()
```

```
Saving figure dataset_3d_plot
```



In [14]:
```
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')

ax.plot(X2D[:, 0], X2D[:, 1], "k+")
ax.plot(X2D[:, 0], X2D[:, 1], "k.")
ax.plot([0], [0], "ko")
ax.arrow(0, 0, 0, 1, head_width=0.05, length_includes_head=True, head_length
=0.1, fc='k', ec='k')
ax.arrow(0, 0, 1, 0, head_width=0.05, length_includes_head=True, head_length
=0.1, fc='k', ec='k')
ax.set_xlabel("$z_1$", fontsize=18)
ax.set_ylabel("$z_2$", fontsize=18, rotation=0)
ax.axis([-1.5, 1.3, -1.2, 1.2])
ax.grid(True)
save_fig("dataset_2d_plot")
```

```
Saving figure dataset_2d_plot
```



PCA Example : MNIST compression

In [15]:
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
mnist.target = mnist.target.astype(np.uint8)
```

In [16]:
```
from sklearn.model_selection import train_test_split

X = mnist["data"]
y = mnist["target"]

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

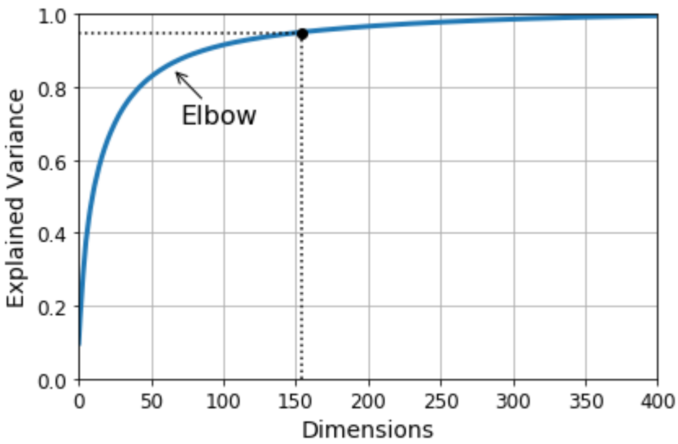
In [17]:
```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

In [18]:
```
d
```

Out[18]:
```
154
```

In [19]:
```
plt.figure(figsize=(6,4))
plt.plot(cumsum, linewidth=3)
plt.axis([0, 400, 0, 1])
plt.xlabel("Dimensions")
plt.ylabel("Explained Variance")
plt.plot([d, d], [0, 0.95], "k:")
plt.plot([0, d], [0.95, 0.95], "k:")
plt.plot(d, 0.95, "ko")
plt.annotate("Elbow", xy=(65, 0.85), xytext=(70, 0.7),
             arrowprops=dict(arrowstyle="->"), fontsize=16)
plt.grid(True)
save_fig("explained_variance_plot")
plt.show()
```

```
Saving figure explained_variance_plot
```



In [20]:
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

In [21]:
```
pca.n_components_
```

Out[21]:
```
154
```

In [22]:
```
np.sum(pca.explained_variance_ratio_)
```

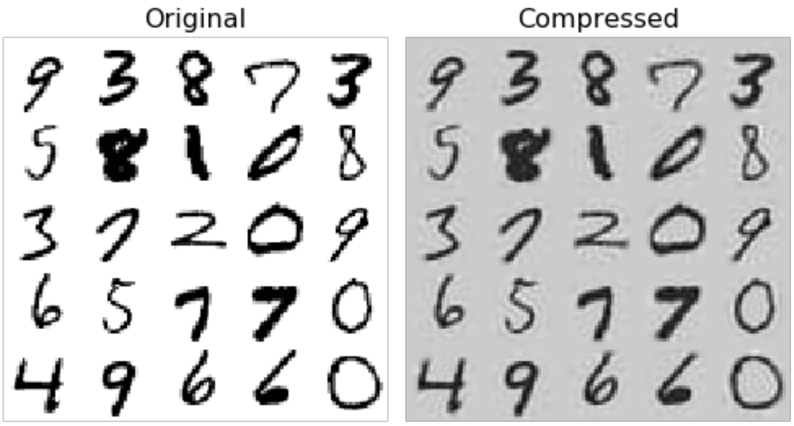
Out[22]:
```
0.9503684424557437
```

In [23]:
```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

In [24]:
```
def plot_digits(instances, images_per_row=5, **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 [25]:
```
plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(X_train[::2100])
plt.title("Original", fontsize=16)
plt.subplot(122)
plot_digits(X_recovered[::2100])
plt.title("Compressed", fontsize=16)

save_fig("mnist_compression_plot")
```

```
Saving figure mnist_compression_plot
```



In [26]:
```
X_reduced_pca = X_reduced
```

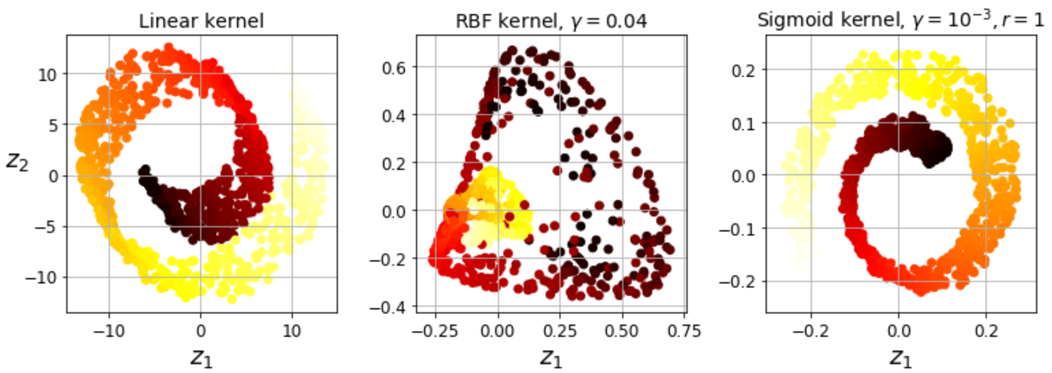
Kernel PCA

In [27]:
```
from sklearn.datasets import make_swiss_roll
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
```

In [28]:
```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

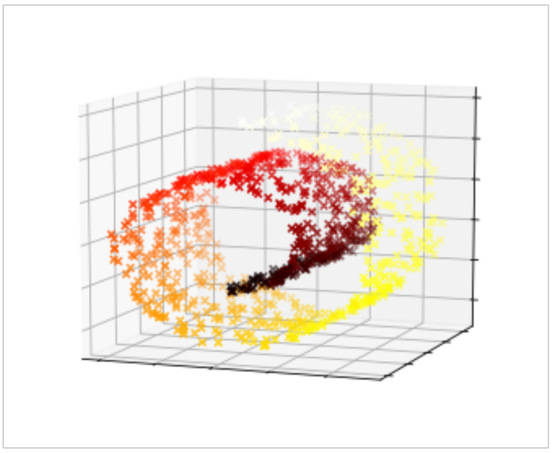
In [29]:
```
from sklearn.decomposition import KernelPCA

lin_pca = KernelPCA(n_components = 2, kernel="linear", fit_inverse_transform
=True)
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433, fit_invers
e_transform=True)
sig_pca = KernelPCA(n_components = 2, kernel="sigmoid", gamma=0.001, coef0=
1, fit_inverse_transform=True)

y = t > 6.9

plt.figure(figsize=(11, 4))
for subplot, pca, title in ((131, lin_pca, "Linear kernel"), (132, rbf_pca,
"RBF kernel, $\gamma=0.04$"), (133, sig_pca, "Sigmoid kernel, $\gamma=10^{-
3}, r=1$")):
    X_reduced = pca.fit_transform(X)
    if subplot == 132:
        X_reduced_rbf = X_reduced

    plt.subplot(subplot)
    #plt.plot(X_reduced[y, 0], X_reduced[y, 1], "gs")
    #plt.plot(X_reduced[~y, 0], X_reduced[~y, 1], "y^")
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)

save_fig("kernel_pca_plot")
plt.show()
```

```
Saving figure kernel_pca_plot
```

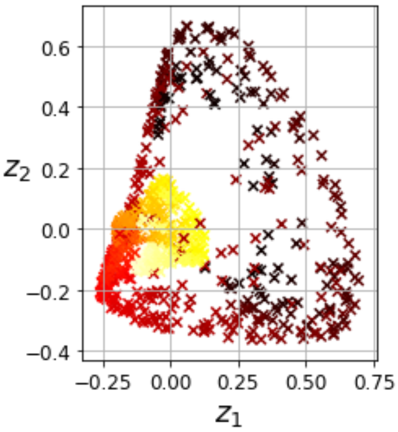


In [30]:
```
plt.figure(figsize=(6, 5))

X_inverse = rbf_pca.inverse_transform(X_reduced_rbf)

ax = plt.subplot(111, projection='3d')
ax.view_init(10, -70)
ax.scatter(X_inverse[:, 0], X_inverse[:, 1], X_inverse[:, 2], c=t, cmap=plt.
cm.hot, marker="x")
ax.set_xlabel("")
ax.set_ylabel("")
ax.set_zlabel("")
ax.set_xticklabels([])
ax.set_yticklabels([])
ax.set_zticklabels([])

save_fig("preimage_plot", tight_layout=False)
plt.show()
```

```
Saving figure preimage_plot
```



In [31]:
```
X_reduced = rbf_pca.fit_transform(X)

plt.figure(figsize=(11, 4))
plt.subplot(132)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot, marker="
x")
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
plt.grid(True)
```



In [32]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression(solver="lbfgs"))
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Out[32]:
```
GridSearchCV(cv=3,
             estimator=Pipeline(steps=[('kpca', KernelPCA(n_components=2)),
                                       ('log_reg', LogisticRegression())]),
             param_grid=[{'kpca__gamma': array([0.03      , 0.03222222, 0.034
44444, 0.03666667, 0.03888889,
       0.04111111, 0.04333333, 0.04555556, 0.04777778, 0.05      ]),
                          'kpca__kernel': ['rbf', 'sigmoid']}])
```

In [33]:
```
print(grid_search.best_params_)
```

```
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

In [34]:
```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```

In [35]:
```
from sklearn.metrics import mean_squared_error

mean_squared_error(X, X_preimage)
```

Out[35]:
```
9.733964708814549e-27
```

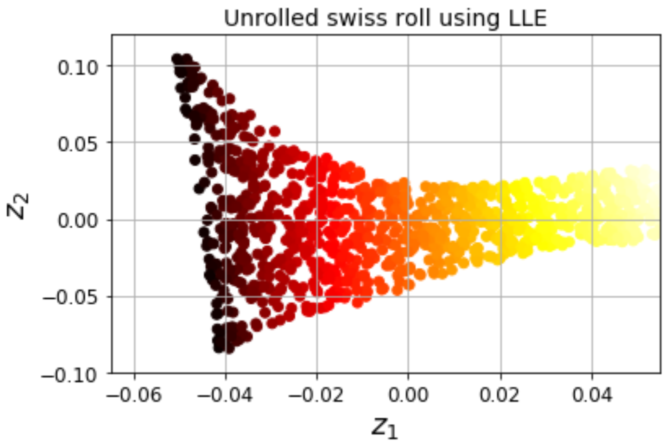
LLE

In [36]:
```
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=41)
```

In [37]:
```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=4
2)
X_reduced = lle.fit_transform(X)
```

In [38]:
```
plt.title("Unrolled swiss roll using LLE", fontsize=14)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18)
plt.axis([-0.065, 0.055, -0.1, 0.12])
plt.grid(True)

save_fig("lle_unrolling_plot")
plt.show()
```

```
Saving figure lle_unrolling_plot
```



MDS, Isomap and t-SNE

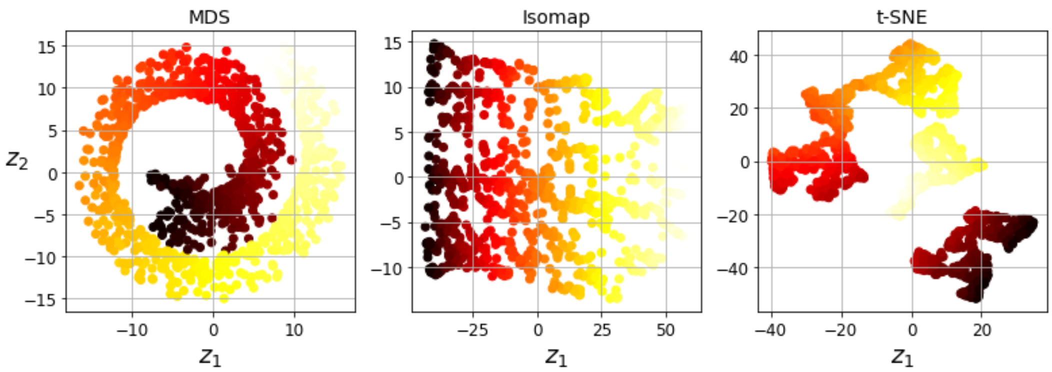
In [39]:
```
from sklearn.manifold import MDS

mds = MDS(n_components=2, random_state=42)
X_reduced_mds = mds.fit_transform(X)
```

In [40]:
```
from sklearn.manifold import Isomap

isomap = Isomap(n_components=2)
X_reduced_isomap = isomap.fit_transform(X)
```

In [41]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
X_reduced_tsne = tsne.fit_transform(X)
```

In [42]:
```
titles = ["MDS", "Isomap", "t-SNE"]

plt.figure(figsize=(11,4))

for subplot, title, X_reduced in zip((131, 132, 133), titles,
                                     (X_reduced_mds, X_reduced_isomap, X_red
uced_tsne)):
    plt.subplot(subplot)
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)

save_fig("other_dim_reduction_plot")
plt.show()
```

```
Saving figure other_dim_reduction_plot
```



Chapter 9 – Unsupervised Learning

This notebook contains all the sample code in chapter 9.

Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/09_unsupervised_learning.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

In [1]:
```
# 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 os

# to make this notebook's output stable across runs
np.random.seed(42)

# 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)

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

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    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)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Clustering

Introduction – Classification *vs* Clustering

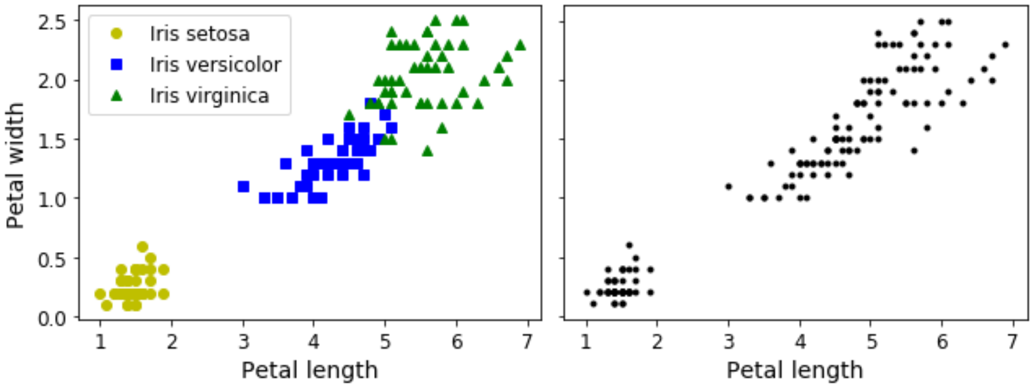
In [2]:
```
from sklearn.datasets import load_iris
```

In [3]:
```
data = load_iris()
X = data.data
y = data.target
data.target_names
```

Out[3]:
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

In [4]:
```
plt.figure(figsize=(9, 3.5))

plt.subplot(121)
plt.plot(X[y==0, 2], X[y==0, 3], "yo", label="Iris setosa")
plt.plot(X[y==1, 2], X[y==1, 3], "bs", label="Iris versicolor")
plt.plot(X[y==2, 2], X[y==2, 3], "g^", label="Iris virginica")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(fontsize=12)

plt.subplot(122)
plt.scatter(X[:, 2], X[:, 3], c="k", marker=".")
plt.xlabel("Petal length", fontsize=14)
plt.tick_params(labelleft=False)

save_fig("classification_vs_clustering_plot")
plt.show()
```

```
Saving figure classification_vs_clustering_plot
```

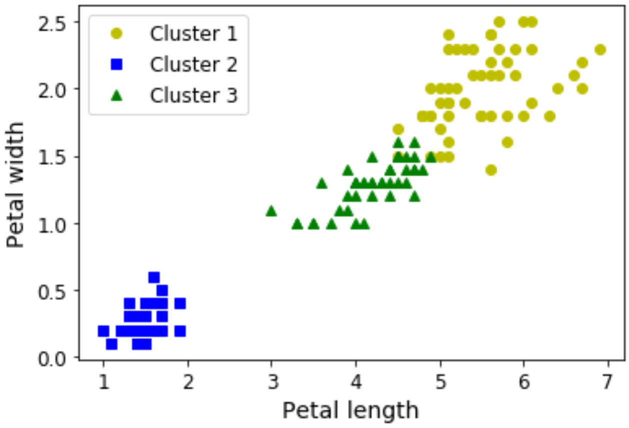


A Gaussian mixture model (explained below) can actually separate these clusters pretty well (using all 4 features: petal length & width, and sepal length & width).

In [5]:
```
from sklearn.mixture import GaussianMixture
```

In [6]:
```
y_pred = GaussianMixture(n_components=3, random_state=42).fit(X).predict(X)
mapping = np.array([2, 0, 1])
y_pred = np.array([mapping[cluster_id] for cluster_id in y_pred])
```

In [7]:
```
plt.plot(X[y_pred==0, 2], X[y_pred==0, 3], "yo", label="Cluster 1")
plt.plot(X[y_pred==1, 2], X[y_pred==1, 3], "bs", label="Cluster 2")
plt.plot(X[y_pred==2, 2], X[y_pred==2, 3], "g^", label="Cluster 3")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=12)
plt.show()
```



K-Means

Let's start by generating some blobs:

In [8]:
```
from sklearn.datasets import make_blobs
```

In [9]:
```
blob_centers = np.array(
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
     [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
```

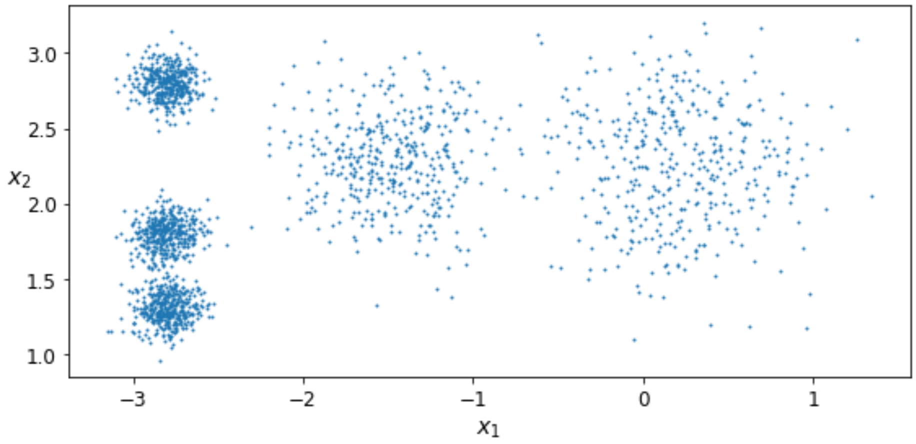
In [10]:
```
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```

Now let's plot them:

In [11]:
```
def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$", fontsize=14)
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

In [12]:
```
plt.figure(figsize=(8, 4))
plot_clusters(X)
save_fig("blobs_plot")
plt.show()
```

```
Saving figure blobs_plot
```



Fit and Predict

Let's train a K-Means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

In [13]:
```
from sklearn.cluster import KMeans
```

In [14]:
```
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Each instance was assigned to one of the 5 clusters:

In [15]:
```
y_pred
```

Out[15]:
```
array([0, 4, 1, ..., 2, 1, 4])
```

And the following 5 *centroids* (i.e., cluster centers) were estimated:

In [16]:
```
kmeans.cluster_centers_
```

Out[16]:
```
array([[-2.80037642,  1.30082566],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80389616,  1.80117999]])
```

Note that the `KMeans` instance preserves the labels of the instances it was trained on. Somewhat confusingly, in this context, the *label* of an instance is the index of the cluster that instance gets assigned to:

In [17]:
```
kmeans.labels_
```

Out[17]:
```
array([0, 4, 1, ..., 2, 1, 4])
```

Of course, we can predict the labels of new instances:

In [18]:
```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```

Out[18]:
```
array([1, 1, 2, 2])
```

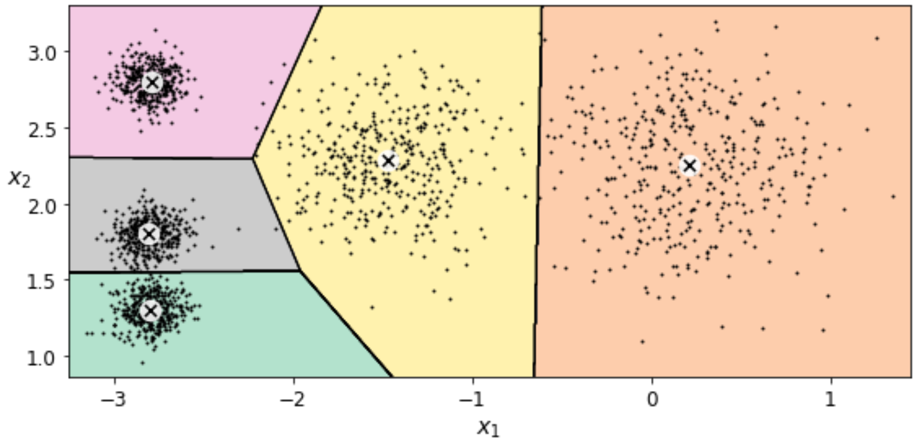
Decision Boundaries

Let's plot the model's decision boundaries. This gives us a *Voronoi diagram*:

In [19]:
```
def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k
'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=30, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=50, linewidths=50,
                color=cross_color, zorder=11, alpha=1)

def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=T
rue,
                             show_xlabels=True, show_ylabels=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                cmap="Pastel2")
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                linewidths=1, colors='k')
    plot_data(X)
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)

    if show_xlabels:
        plt.xlabel("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

In [20]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
save_fig("voronoi_plot")
plt.show()
```

```
Saving figure voronoi_plot
```



Not bad! Some of the instances near the edges were probably assigned to the wrong cluster, but overall it looks pretty good.

Hard Clustering *vs* Soft Clustering

Rather than arbitrarily choosing the closest cluster for each instance, which is called *hard clustering*, it might be better measure the distance of each instance to all 5 centroids. This is what the `transform()` method does:

In [21]:
```
kmeans.transform(X_new)
```

Out[21]:
```
array([[2.88633901, 0.32995317, 2.9042344 , 1.49439034, 2.81093633],
       [5.84236351, 2.80290755, 5.84739223, 4.4759332 , 5.80730058],
       [1.71086031, 3.29399768, 0.29040966, 1.69136631, 1.21475352],
       [1.21567622, 3.21806371, 0.36159148, 1.54808703, 0.72581411]])
```

You can verify that this is indeed the Euclidian distance between each instance and each centroid:

In [22]:
```
np.linalg.norm(np.tile(X_new, (1, k)).reshape(-1, k, 2) - kmeans.cluster_cen
ters_, axis=2)
```

Out[22]:
```
array([[2.88633901, 0.32995317, 2.9042344 , 1.49439034, 2.81093633],
       [5.84236351, 2.80290755, 5.84739223, 4.4759332 , 5.80730058],
       [1.71086031, 3.29399768, 0.29040966, 1.69136631, 1.21475352],
       [1.21567622, 3.21806371, 0.36159148, 1.54808703, 0.72581411]])
```

K-Means Algorithm

The K-Means algorithm is one of the fastest clustering algorithms, but also one of the simplest:

- First initialize k centroids randomly: k distinct instances are chosen randomly from the dataset and the centroids are placed at their locations.
- Repeat until convergence (i.e., until the centroids stop moving):
  - Assign each instance to the closest centroid.
  - Update the centroids to be the mean of the instances that are assigned to them.

The `KMeans` class applies an optimized algorithm by default. To get the original K-Means algorithm (for educational purposes only), you must set `init="random"` , `n_init=1` and `algorithm="full"` . These hyperparameters will be explained below.

Let's run the K-Means algorithm for 1, 2 and 3 iterations, to see how the centroids move around:

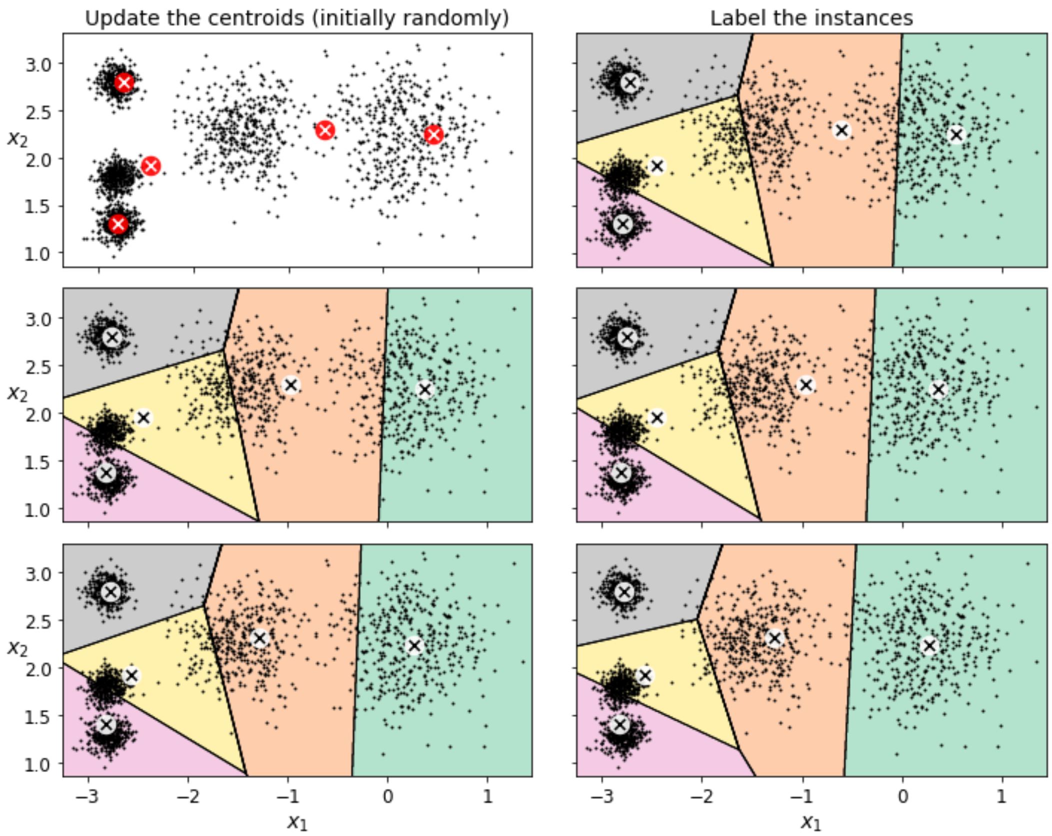
In [23]:
```
kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=1, random_state=1)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=2, random_state=1)
kmeans_iter3 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=3, random_state=1)
kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)
```

Out[23]:
```
KMeans(algorithm='full', init='random', max_iter=3, n_clusters=5, n_init=1,
       random_state=1)
```

And let's plot this:

In [24]:
```
plt.figure(figsize=(10, 8))

plt.subplot(321)
plot_data(X)
plot_centroids(kmeans_iter1.cluster_centers_, circle_color='r', cross_color=
'w')
plt.ylabel("$x_2$", fontsize=14, rotation=0)
plt.tick_params(labelbottom=False)
plt.title("Update the centroids (initially randomly)", fontsize=14)

plt.subplot(322)
plot_decision_boundaries(kmeans_iter1, X, show_xlabels=False, show_ylabels=F
alse)
plt.title("Label the instances", fontsize=14)

plt.subplot(323)
plot_decision_boundaries(kmeans_iter1, X, show_centroids=False, show_xlabels
=False)
plot_centroids(kmeans_iter2.cluster_centers_)

plt.subplot(324)
plot_decision_boundaries(kmeans_iter2, X, show_xlabels=False, show_ylabels=F
alse)

plt.subplot(325)
plot_decision_boundaries(kmeans_iter2, X, show_centroids=False)
plot_centroids(kmeans_iter3.cluster_centers_)

plt.subplot(326)
plot_decision_boundaries(kmeans_iter3, X, show_ylabels=False)

save_fig("kmeans_algorithm_plot")
plt.show()
```

```
Saving figure kmeans_algorithm_plot
```



K-Means Variability

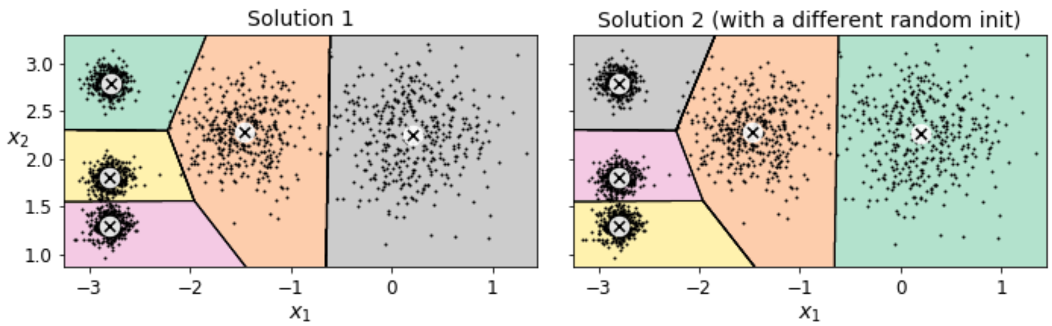
In the original K-Means algorithm, the centroids are just initialized randomly, and the algorithm simply runs a single iteration to gradually improve the centroids, as we saw above.

However, one major problem with this approach is that if you run K-Means multiple times (or with different random seeds), it can converge to very different solutions, as you can see below:

In [25]:
```
def plot_clusterer_comparison(clusterer1, clusterer2, X, title1=None, title2
=None):
    clusterer1.fit(X)
    clusterer2.fit(X)

    plt.figure(figsize=(10, 3.2))

    plt.subplot(121)
    plot_decision_boundaries(clusterer1, X)
    if title1:
        plt.title(title1, fontsize=14)

    plt.subplot(122)
    plot_decision_boundaries(clusterer2, X, show_ylabels=False)
    if title2:
        plt.title(title2, fontsize=14)
```

In [26]:
```
kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=11)
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=19)

plot_clusterer_comparison(kmeans_rnd_init1, kmeans_rnd_init2, X,
                           "Solution 1", "Solution 2 (with a different random
init)")

save_fig("kmeans_variability_plot")
plt.show()
```

```
Saving figure kmeans_variability_plot
```



Inertia

To select the best model, we will need a way to evaluate a K-Mean model's performance. Unfortunately, clustering is an unsupervised task, so we do not have the targets. But at least we can measure the distance between each instance and its centroid. This is the idea behind the *inertia* metric:

In [27]:
```
kmeans.inertia_
```
Out[27]:
```
211.5985372581683
```

As you can easily verify, inertia is the sum of the squared distances between each training instance and its closest centroid:

In [28]:
```
X_dist = kmeans.transform(X)
np.sum(X_dist[np.arange(len(X_dist)), kmeans.labels_]**2)
```
Out[28]:
```
211.5985372581687
```

The `score()` method returns the negative inertia. Why negative? Well, it is because a predictor's `score()` method must always respect the "*great is better*" rule.

In [29]:
```
kmeans.score(X)
```
Out[29]:
```
-211.59853725816828
```

Multiple Initializations

So one approach to solve the variability issue is to simply run the K-Means algorithm multiple times with different random initializations, and select the solution that minimizes the inertia. For example, here are the inertias of the two "bad" models shown in the previous figure:

In [30]:
```
kmeans_rnd_init1.inertia_
```
Out[30]:
```
211.60832621558367
```

In [31]:
```
kmeans_rnd_init2.inertia_
```
Out[31]:
```
211.62301821329766
```

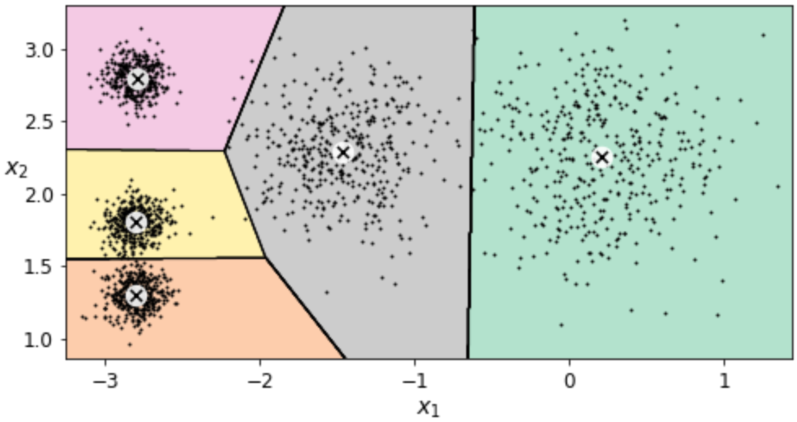
As you can see, they have a higher inertia than the first "good" model we trained, which means they are probably worse.

When you set the `n_init` hyperparameter, Scikit-Learn runs the original algorithm `n_init` times, and selects the solution that minimizes the inertia. By default, Scikit-Learn sets `n_init=10`.

In [32]:
```
kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random", n_init=10,
                              algorithm="full", random_state=11)
kmeans_rnd_10_inits.fit(X)
```
Out[32]:
```
KMeans(algorithm='full', init='random', n_clusters=5, random_state=11)
```

As you can see, we end up with the initial model, which is certainly the optimal K-Means solution (at least in terms of inertia, and assuming $k = 5$).

In [33]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans_rnd_10_inits, X)
plt.show()
```



K-Means++

Instead of initializing the centroids entirely randomly, it is preferable to initialize them using the following algorithm, proposed in a [2006 paper (https://goo.gl/eNUPw6)](https://goo.gl/eNUPw6) by David Arthur and Sergei Vassilvitskii:

- Take one centroid c_1, chosen uniformly at random from the dataset.
- Take a new center c_i, choosing an instance $\mathbf{x}_i$ with probability: $D(\mathbf{x}_i)^2$ / $\sum\limits_{j=1}^{m} D(\mathbf{x}_j)^2$ where $D(\mathbf{x}_i)$ is the distance between the instance $\mathbf{x}_i$ and the closest centroid that was already chosen. This probability distribution ensures that instances that are further away from already chosen centroids are much more likely be selected as centroids.
- Repeat the previous step until all k centroids have been chosen.

The rest of the K-Means++ algorithm is just regular K-Means. With this initialization, the K-Means algorithm is much less likely to converge to a suboptimal solution, so it is possible to reduce `n_init` considerably. Most of the time, this largely compensates for the additional complexity of the initialization process.

To set the initialization to K-Means++, simply set `init="k-means++"` (this is actually the default):

In [34]:
```
KMeans()
```

Out[34]: KMeans()

In [35]:
```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
kmeans.fit(X)
kmeans.inertia_
```

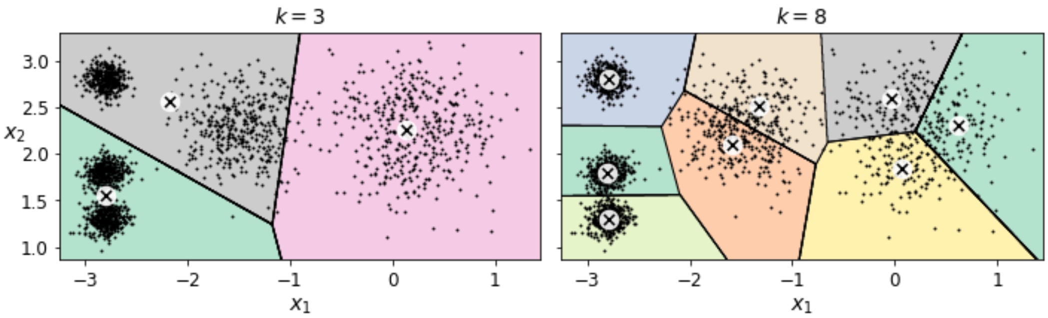
Out[35]: 211.62337889822362

Finding the optimal number of clusters

What if the number of clusters was set to a lower or greater value than 5?

In [36]:
```
kmeans_k3 = KMeans(n_clusters=3, random_state=42)
kmeans_k8 = KMeans(n_clusters=8, random_state=42)

plot_clusterer_comparison(kmeans_k3, kmeans_k8, X, "$k=3$", "$k=8$")
save_fig("bad_n_clusters_plot")
plt.show()
```

Saving figure bad_n_clusters_plot



Ouch, these two models don't look great. What about their inertias?

In [37]:
```
kmeans_k3.inertia_
```

Out[37]: 653.2223267580945

In [38]:
```
kmeans_k8.inertia_
```

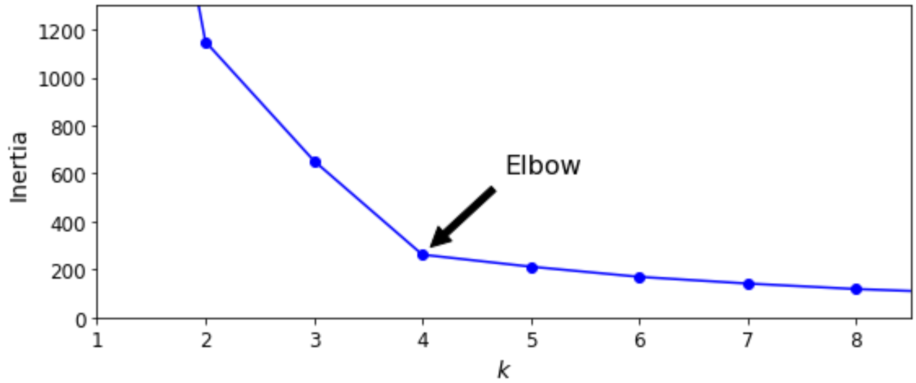
Out[38]: 118.44108623570082

No, we cannot simply take the value of k that minimizes the inertia, since it keeps getting lower as we increase k. Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. However, we can plot the inertia as a function of k and analyze the resulting curve:

In [39]:
```
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]
```

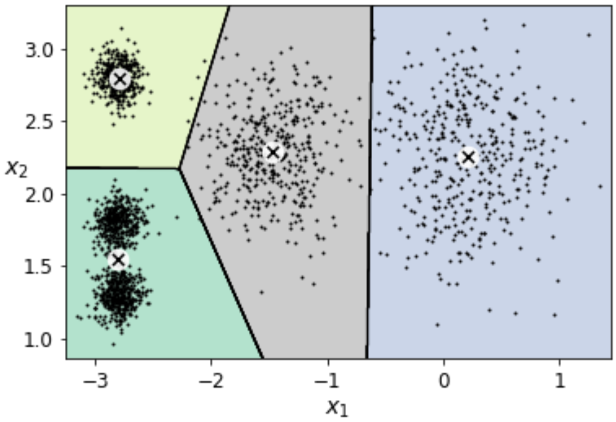
In [40]:
```
plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(4, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.axis([1, 8.5, 0, 1300])
save_fig("inertia_vs_k_plot")
plt.show()
```

```
Saving figure inertia_vs_k_plot
```



As you can see, there is an elbow at $k = 4$, which means that less clusters than that would be bad, and more clusters would not help much and might cut clusters in half. So $k = 4$ is a pretty good choice. Of course in this example it is not perfect since it means that the two blobs in the lower left will be considered as just a single cluster, but it's a pretty good clustering nonetheless.

In [41]:
```
plot_decision_boundaries(kmeans_per_k[4-1], X)
plt.show()
```



Another approach is to look at the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to $(b - a) / \max(a, b)$ where a is the mean distance to the other instances in the same cluster (it is the *mean intra-cluster distance*), and b is the *mean nearest-cluster distance*, that is the mean distance to the instances of the next closest cluster (defined as the one that minimizes b, excluding the instance's own cluster). The silhouette coefficient can vary between -1 and +1: a coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

Let's plot the silhouette score as a function of k:

In [42]:
```
from sklearn.metrics import silhouette_score
```

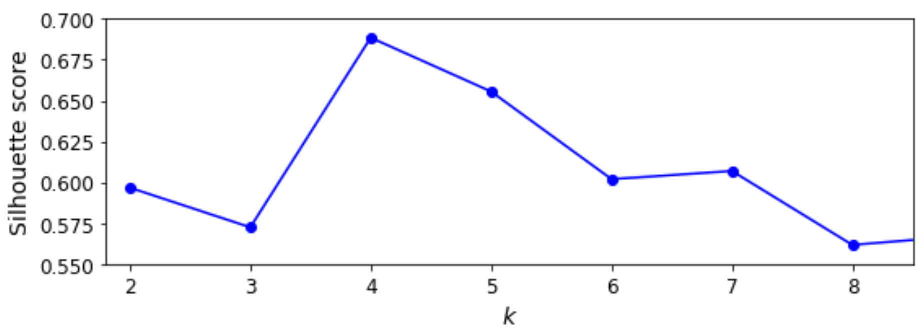
In [43]:
```
silhouette_score(X, kmeans.labels_)
```

Out[43]:
```
0.655517642572828
```

In [44]:
```
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]
```

In [45]:
```
plt.figure(figsize=(8, 3))
plt.plot(range(2, 10), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.axis([1.8, 8.5, 0.55, 0.7])
save_fig("silhouette_score_vs_k_plot")
plt.show()
```

```
Saving figure silhouette_score_vs_k_plot
```

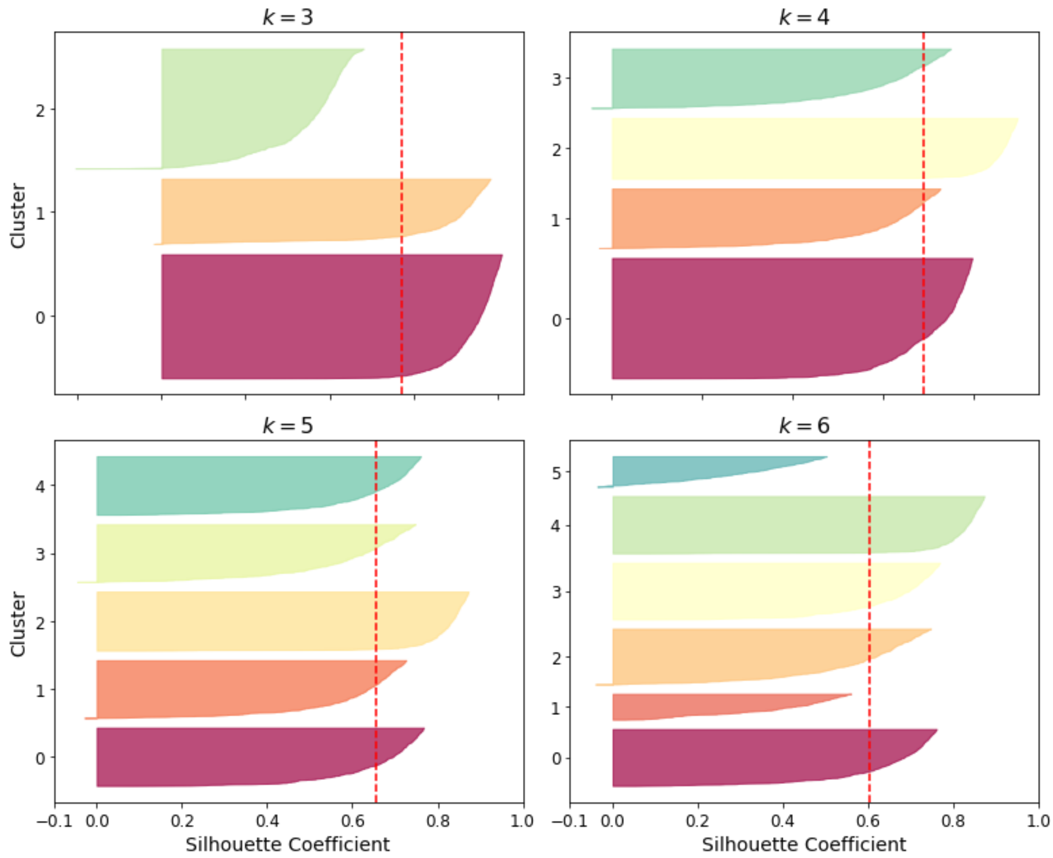


As you can see, this visualization is much richer than the previous one: in particular, although it confirms that $k = 4$ is a very good choice, but it also underlines the fact that $k = 5$ is quite good as well.

An even more informative visualization is given when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient. This is called a *silhouette diagram*:

In [46]:
```
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter

plt.figure(figsize=(11, 9))

for k in (3, 4, 5, 6):
    plt.subplot(2, 2, k - 2)

    y_pred = kmeans_per_k[k - 1].labels_
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
        coeffs.sort()

        color = mpl.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (3, 5):
        plt.ylabel("Cluster")

    if k in (5, 6):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)

save_fig("silhouette_analysis_plot")
plt.show()
```

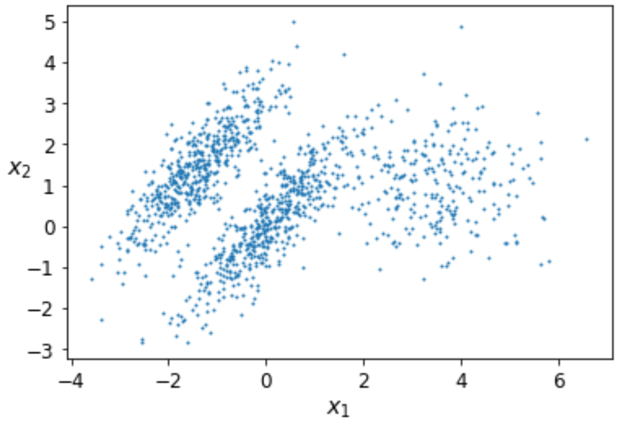
```
Saving figure silhouette_analysis_plot
```



Limits of K-Means

In [47]:
```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=
42)
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]
```

In [48]:
```
plot_clusters(X)
```

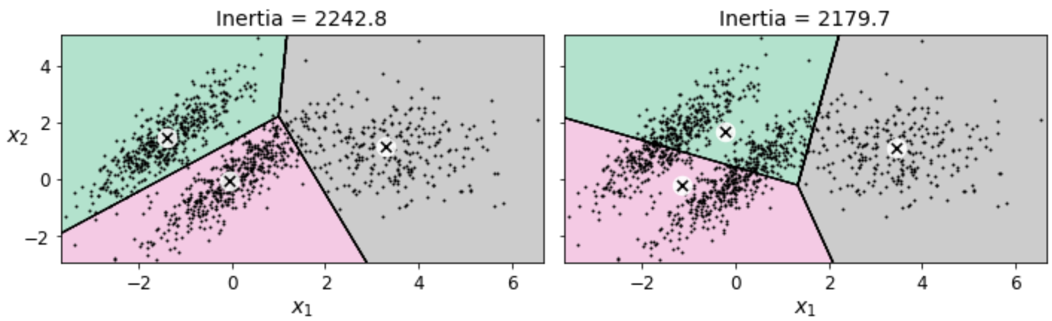


In [49]:
```
kmeans_good = KMeans(n_clusters=3, init=np.array([[-1.5, 2.5], [0.5, 0], [4,
0]]), n_init=1, random_state=42)
kmeans_bad = KMeans(n_clusters=3, random_state=42)
kmeans_good.fit(X)
kmeans_bad.fit(X)
```

Out[49]:
```
KMeans(n_clusters=3, random_state=42)
```

In [50]:
```
plt.figure(figsize=(10, 3.2))

plt.subplot(121)
plot_decision_boundaries(kmeans_good, X)
plt.title("Inertia = {:.1f}".format(kmeans_good.inertia_), fontsize=14)

plt.subplot(122)
plot_decision_boundaries(kmeans_bad, X, show_ylabels=False)
plt.title("Inertia = {:.1f}".format(kmeans_bad.inertia_), fontsize=14)

save_fig("bad_kmeans_plot")
plt.show()
```

```
Saving figure bad_kmeans_plot
```



Gaussian Mixtures

In [51]:
```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=
42)
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]
```

Let's train a Gaussian mixture model on the previous dataset:

In [52]:
```
from sklearn.mixture import GaussianMixture
```

In [53]:
```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Out[53]:
```
GaussianMixture(n_components=3, n_init=10, random_state=42)
```

Let's look at the parameters that the EM algorithm estimated:

In [54]:
```
gm.weights_
```

Out[54]:
```
array([0.39054348, 0.2093669 , 0.40008962])
```

In [55]:
```
gm.means_
```

Out[55]:
```
array([[ 0.05224874,  0.07631976],
       [ 3.40196611,  1.05838748],
       [-1.40754214,  1.42716873]])
```

In [56]:
```
gm.covariances_
```

Out[56]:
```
array([[[ 0.6890309 ,  0.79717058],
        [ 0.79717058,  1.21367348]],

       [[ 1.14296668, -0.03114176],
        [-0.03114176,  0.9545003 ]],

       [[ 0.63496849,  0.7298512 ],
        [ 0.7298512 ,  1.16112807]]])
```

Did the algorithm actually converge?

In [57]:
```
gm.converged_
```

Out[57]:
```
True
```

Yes, good. How many iterations did it take?

In [58]:
```
gm.n_iter_
```

Out[58]:
```
4
```

You can now use the model to predict which cluster each instance belongs to (hard clustering) or the probabilities that it came from each cluster. For this, just use `predict()` method or the `predict_proba()` method:

In [59]:
```
gm.predict(X)
```

Out[59]:
```
array([0, 0, 2, ..., 1, 1, 1], dtype=int64)
```

In [60]:
```
gm.predict_proba(X)
```

Out[60]:
```
array([[9.77227791e-01, 2.27715290e-02, 6.79898914e-07],
       [9.83288385e-01, 1.60345103e-02, 6.77104389e-04],
       [7.51824662e-05, 1.90251273e-06, 9.99922915e-01],
       ...,
       [4.35053542e-07, 9.99999565e-01, 2.17938894e-26],
       [5.27837047e-16, 1.00000000e+00, 1.50679490e-41],
       [2.32355608e-15, 1.00000000e+00, 8.21915701e-41]])
```

This is a generative model, so you can sample new instances from it (and get their labels):

In [61]:
```
X_new, y_new = gm.sample(6)
X_new
```

Out[61]:
```
array([[-0.8690223 , -0.32680051],
       [ 0.29945755,  0.2841852 ],
       [ 1.85027284,  2.06556913],
       [ 3.98260019,  1.50041446],
       [ 3.82006355,  0.53143606],
       [-1.04015332,  0.7864941 ]])
```

In [62]:
```
y_new
```

Out[62]:
```
array([0, 0, 1, 1, 1, 2])
```

Notice that they are sampled sequentially from each cluster.

You can also estimate the log of the *probability density function* (PDF) at any location using the `score_samples()` method:

In [63]:
```
gm.score_samples(X)
```

Out[63]:
```
array([-2.60674489, -3.57074133, -3.33007348, ..., -3.51379355,
       -4.39643283, -3.8055665 ])
```

Let's check that the PDF integrates to 1 over the whole space. We just take a large square around the clusters, and chop it into a grid of tiny squares, then we compute the approximate probability that the instances will be generated in each tiny square (by multiplying the PDF at one corner of the tiny square by the area of the square), and finally summing all these probabilities). The result is very close to 1:

In [64]:
```
resolution = 100
grid = np.arange(-10, 10, 1 / resolution)
xx, yy = np.meshgrid(grid, grid)
X_full = np.vstack([xx.ravel(), yy.ravel()]).T

pdf = np.exp(gm.score_samples(X_full))
pdf_probas = pdf * (1 / resolution) ** 2
pdf_probas.sum()
```

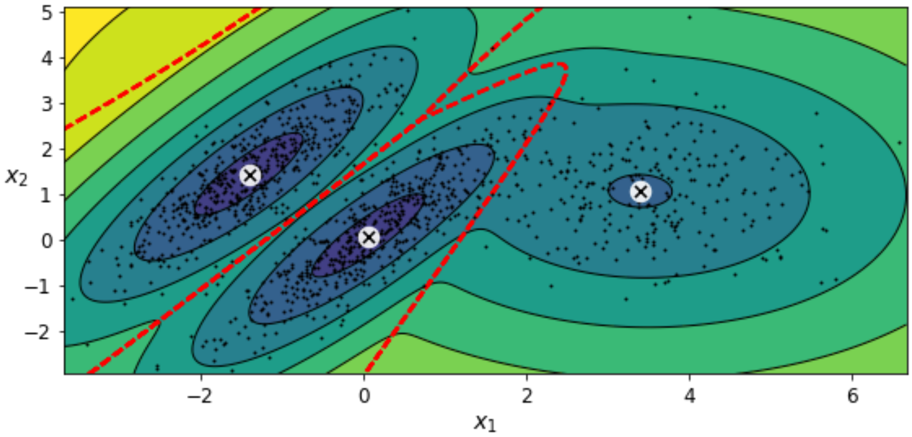
Out[64]:
```
0.9999999999271592
```

Now let's plot the resulting decision boundaries (dashed lines) and density contours:

In [65]:
```
from matplotlib.colors import LogNorm

def plot_gaussian_mixture(clusterer, X, resolution=1000, show_ylabels=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = -clusterer.score_samples(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z,
                 norm=LogNorm(vmin=1.0, vmax=30.0),
                 levels=np.logspace(0, 2, 12))
    plt.contour(xx, yy, Z,
                norm=LogNorm(vmin=1.0, vmax=30.0),
                levels=np.logspace(0, 2, 12),
                linewidths=1, colors='k')

    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z,
                linewidths=2, colors='r', linestyles='dashed')

    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)
    plot_centroids(clusterer.means_, clusterer.weights_)

    plt.xlabel("$x_1$", fontsize=14)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

In [66]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)

save_fig("gaussian_mixtures_plot")
plt.show()
```

Saving figure gaussian_mixtures_plot



You can impose constraints on the covariance matrices that the algorithm looks for by setting the `covariance_type` hyperparameter:

- `"full"` (default): no constraint, all clusters can take on any ellipsoidal shape of any size.
- `"tied"` : all clusters must have the same shape, which can be any ellipsoid (i.e., they all share the same covariance matrix).
- `"spherical"` : all clusters must be spherical, but they can have different diameters (i.e., different variances).
- `"diag"` : clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the axes (i.e., the covariance matrices must be diagonal).

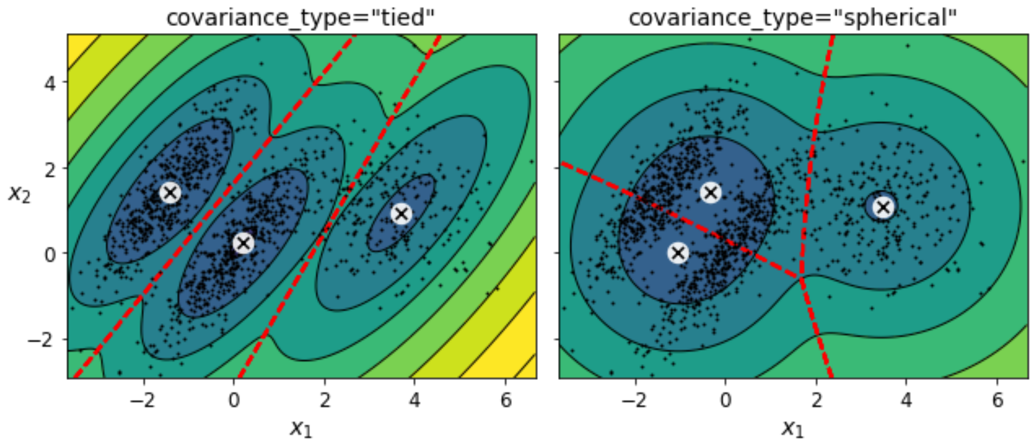
In [67]:
```
gm_full = GaussianMixture(n_components=3, n_init=10, covariance_type="full", random_state=42)
gm_tied = GaussianMixture(n_components=3, n_init=10, covariance_type="tied", random_state=42)
gm_spherical = GaussianMixture(n_components=3, n_init=10, covariance_type="spherical", random_state=42)
gm_diag = GaussianMixture(n_components=3, n_init=10, covariance_type="diag", random_state=42)
gm_full.fit(X)
gm_tied.fit(X)
gm_spherical.fit(X)
gm_diag.fit(X)
```

Out[67]:
```
GaussianMixture(covariance_type='diag', n_components=3, n_init=10,
                random_state=42)
```

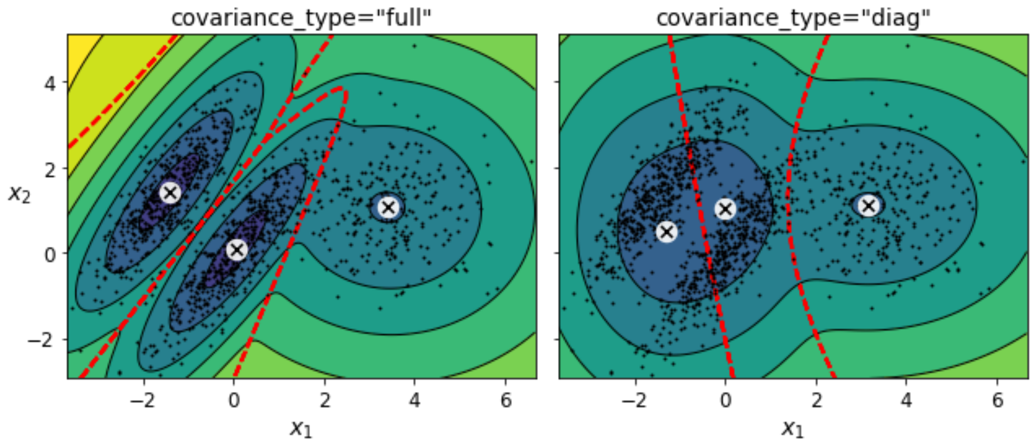
In [68]:
```
def compare_gaussian_mixtures(gm1, gm2, X):
    plt.figure(figsize=(9, 4))

    plt.subplot(121)
    plot_gaussian_mixture(gm1, X)
    plt.title('covariance_type="{}"'.format(gm1.covariance_type), fontsize=14)

    plt.subplot(122)
    plot_gaussian_mixture(gm2, X, show_ylabels=False)
    plt.title('covariance_type="{}"'.format(gm2.covariance_type), fontsize=14)
```

In [69]:
```
compare_gaussian_mixtures(gm_tied, gm_spherical, X)

save_fig("covariance_type_plot")
plt.show()
```

```
Saving figure covariance_type_plot
```



In [70]:
```
compare_gaussian_mixtures(gm_full, gm_diag, X)
plt.tight_layout()
plt.show()
```



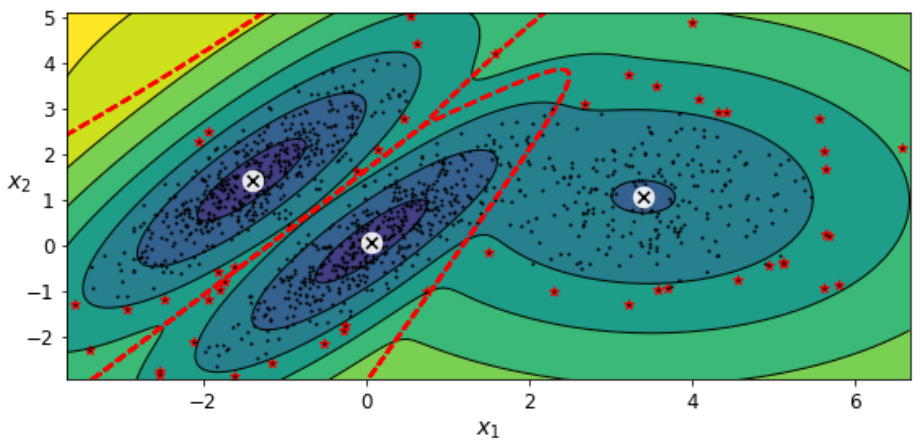
Anomaly Detection using Gaussian Mixtures

Gaussian Mixtures can be used for *anomaly detection*: instances located in low-density regions can be considered anomalies. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well-known. Say it is equal to 4%, then you can set the density threshold to be the value that results in having 4% of the instances located in areas below that threshold density:

In [71]:
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

In [72]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)
plt.scatter(anomalies[:, 0], anomalies[:, 1], color='r', marker='*')
plt.ylim(top=5.1)

save_fig("mixture_anomaly_detection_plot")
plt.show()
```

```
Saving figure mixture_anomaly_detection_plot
```



Model selection

We cannot use the inertia or the silhouette score because they both assume that the clusters are spherical. Instead, we can try to find the model that minimizes a theoretical information criterion such as the Bayesian Information Criterion (BIC) or the Akaike Information Criterion (AIC):

$$BIC = \log(m)p - 2\log(\hat{L})$$

$$AIC = 2p - 2\log(\hat{L})$$

- m is the number of instances.
- p is the number of parameters learned by the model.
- $\hat{L}$ is the maximized value of the likelihood function of the model. This is the conditional probability of the observed data $\mathbf{X}$, given the model and its optimized parameters.

Both BIC and AIC penalize models that have more parameters to learn (e.g., more clusters), and reward models that fit the data well (i.e., models that give a high likelihood to the observed data).

In [73]:
```
gm.bic(X)
```

Out[73]: 8189.662685850679

In [74]:
```
gm.aic(X)
```

Out[74]: 8102.437405735641

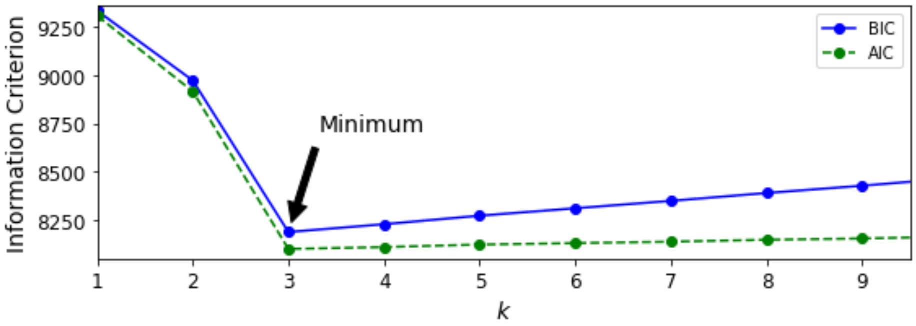
Let's train Gaussian Mixture models with various values of k and measure their BIC:

In [75]:
```
gms_per_k = [GaussianMixture(n_components=k, n_init=10, random_state=42).fit(X)
             for k in range(1, 11)]
```

In [76]:
```
bics = [model.bic(X) for model in gms_per_k]
aics = [model.aic(X) for model in gms_per_k]
```

In [77]:
```
plt.figure(figsize=(8, 3))
plt.plot(range(1, 11), bics, "bo-", label="BIC")
plt.plot(range(1, 11), aics, "go--", label="AIC")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Information Criterion", fontsize=14)
plt.axis([1, 9.5, np.min(aics) - 50, np.max(aics) + 50])
plt.annotate('Minimum',
             xy=(3, bics[2]),
             xytext=(0.35, 0.6),
             textcoords='figure fraction',
             fontsize=14,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.legend()
save_fig("aic_bic_vs_k_plot")
plt.show()
```

Saving figure aic_bic_vs_k_plot



Let's search for best combination of values for both the number of clusters and the `covariance_type` hyperparameter:

In [78]:
```
min_bic = np.infty

for k in range(1, 11):
    for covariance_type in ("full", "tied", "spherical", "diag"):
        bic = GaussianMixture(n_components=k, n_init=10,
                              covariance_type=covariance_type,
                              random_state=42).fit(X).bic(X)
        if bic < min_bic:
            min_bic = bic
            best_k = k
            best_covariance_type = covariance_type
```

In [79]:
```
best_k
```

Out[79]: 3

In [80]:
```
best_covariance_type
```

Out[80]: 'full'

Note : Rather than manually searching for the optimal number of clusters, it is possible to use instead the `BayesianGaussianMixture` class which is capable of giving weights equal (or close) to zero to unnecessary clusters. Just set the number of components to a value that you believe is greater than the optimal number of clusters, and the algorithm will eliminate the unnecessary clusters automatically.

In [ ]:

Chapter 10 – Introduction to Artificial Neural Networks with Keras

This notebook contains all the sample code and solutions to the exercises in chapter 10.

[Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/10_neural_nets_with_keras.ipynb)](https://colab.research.google.com/github/ageron/handson-ml2/blob/master/10_neural_nets_with_keras.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20 and TensorFlow ≥2.0.

In [1]:
```
# 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"

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass

# TensorFlow ≥2.0 is required
import tensorflow as tf
assert tf.__version__ >= "2.0"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# 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)

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

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    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)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Perceptrons

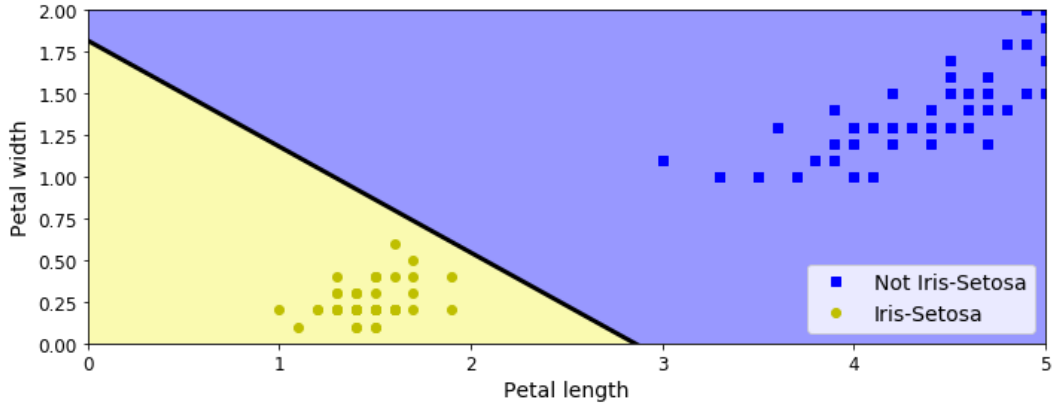
Note: we set `max_iter` and `tol` explicitly to avoid warnings about the fact that their default value will change in future versions of Scikit-Learn.

In [2]:
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)
```

Out[2]:
```
Perceptron(random_state=42)
```

In [3]:
```
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linew
idth=3)
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

save_fig("perceptron_iris_plot")
plt.show()
```

```
Saving figure perceptron_iris_plot
```

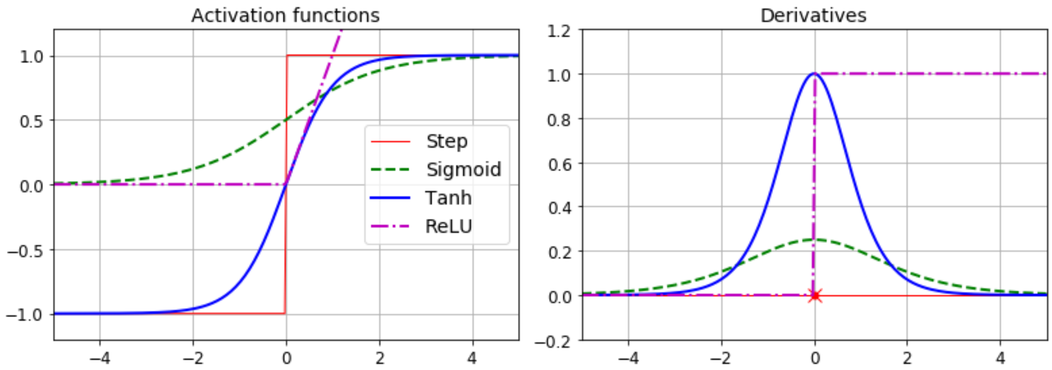


Activation functions

In [4]:
```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def relu(z):
    return np.maximum(0, z)

def derivative(f, z, eps=0.000001):
    return (f(z + eps) - f(z - eps))/(2 * eps)
```

In [5]:
```
z = np.linspace(-5, 5, 200)

plt.figure(figsize=(11,4))

plt.subplot(121)
plt.plot(z, np.sign(z), "r-", linewidth=1, label="Step")
plt.plot(z, sigmoid(z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, np.tanh(z), "b-", linewidth=2, label="Tanh")
plt.plot(z, relu(z), "m-.", linewidth=2, label="ReLU")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.title("Activation functions", fontsize=14)
plt.axis([-5, 5, -1.2, 1.2])

plt.subplot(122)
plt.plot(z, derivative(np.sign, z), "r-", linewidth=1, label="Step")
plt.plot(0, 0, "ro", markersize=5)
plt.plot(0, 0, "rx", markersize=10)
plt.plot(z, derivative(sigmoid, z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, derivative(np.tanh, z), "b-", linewidth=2, label="Tanh")
plt.plot(z, derivative(relu, z), "m-.", linewidth=2, label="ReLU")
plt.grid(True)
#plt.legend(loc="center right", fontsize=14)
plt.title("Derivatives", fontsize=14)
plt.axis([-5, 5, -0.2, 1.2])

save_fig("activation_functions_plot")
plt.show()
```

```
Saving figure activation_functions_plot
```



Building an Image Classifier

First let's import TensorFlow and Keras.

In [6]:
```
import tensorflow as tf
from tensorflow import keras
```

In [7]:
```
tf.__version__
```
Out[7]:
```
'2.1.0'
```

In [8]:
```
keras.__version__
```
Out[8]:
```
'2.2.4-tf'
```

Let's start by loading the fashion MNIST dataset. Keras has a number of functions to load popular datasets in `keras.datasets`. The dataset is already split for you between a training set and a test set, but it can be useful to split the training set further to have a validation set:

In [9]:
```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

The training set contains 60,000 grayscale images, each 28x28 pixels:

In [10]:
```
X_train_full.shape
```
Out[10]:
```
(60000, 28, 28)
```

Each pixel intensity is represented as a byte (0 to 255):

In [11]:
```
X_train_full.dtype
```
Out[11]:
```
dtype('uint8')
```

Let's split the full training set into a validation set and a (smaller) training set. We also scale the pixel intensities down to the 0-1 range and convert them to floats, by dividing by 255.

In [12]:
```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

You can plot an image using Matplotlib's `imshow()` function, with a `'binary'` color map:

In [13]:
```
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```

The labels are the class IDs (represented as uint8), from 0 to 9:

In [14]:
```
y_train
```

Out[14]:
```
array([4, 0, 7, ..., 3, 0, 5], dtype=uint8)
```

Here are the corresponding class names:

In [15]:
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

So the first image in the training set is a coat:

In [16]:
```
class_names[y_train[0]]
```

Out[16]:
```
'Coat'
```

The validation set contains 5,000 images, and the test set contains 10,000 images:

In [17]:
```
X_valid.shape
```

Out[17]:
```
(5000, 28, 28)
```

In [18]:
```
X_test.shape
```

Out[18]:
```
(10000, 28, 28)
```

Let's take a look at a sample of the images in the dataset:

In [19]:
```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_plot', tight_layout=False)
plt.show()
```

```
Saving figure fashion_mnist_plot
```

In [20]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

In [21]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

In [22]:
```
model.layers
```

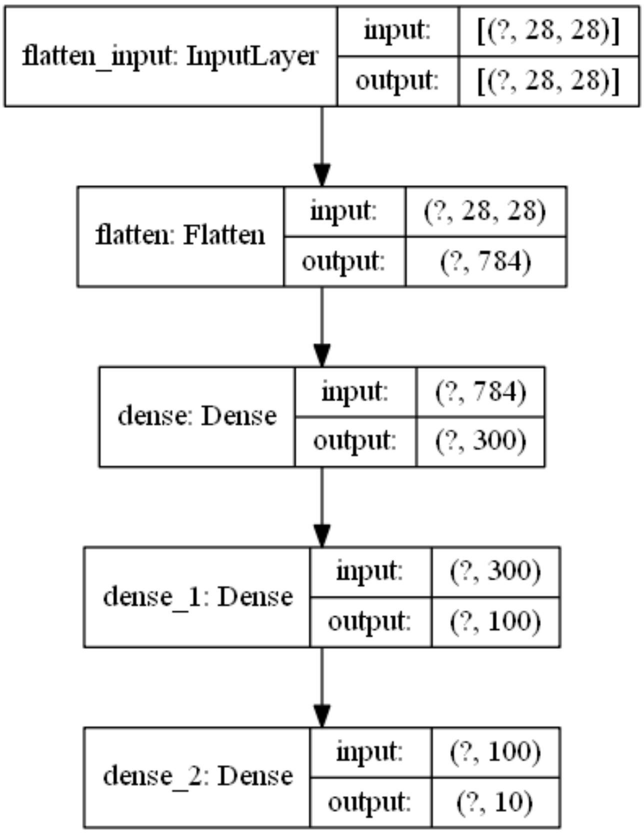
Out[22]:
```
[<tensorflow.python.keras.layers.core.Flatten at 0x22eb0840d48>,
 <tensorflow.python.keras.layers.core.Dense at 0x22eb084b748>,
 <tensorflow.python.keras.layers.core.Dense at 0x22eafe39808>,
 <tensorflow.python.keras.layers.core.Dense at 0x22eafe34148>]
```

In [23]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten (Flatten)            (None, 784)               0         
_________________________________________________________________
dense (Dense)                (None, 300)               235500    
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100     
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010      
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [24]:
```
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```

Out[24]:



In [25]:
```
hidden1 = model.layers[1]
hidden1.name
```

Out[25]: 'dense'

In [26]:
```
model.get_layer(hidden1.name) is hidden1
```

Out[26]: True

In [27]:
```
weights, biases = hidden1.get_weights()
```

In [28]:
```
weights
```

Out[28]:
```
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       [ 0.00476504, -0.03105379, -0.0586676 , ...,  0.00602964,
        -0.02763776, -0.04165364],
       [-0.06189284, -0.06901957,  0.07102345, ..., -0.04238207,
         0.07121518, -0.07331658],
       ...,
       [-0.03048757,  0.02155137, -0.05400612, ..., -0.00113463,
         0.00228987,  0.05581069],
       [ 0.07061854, -0.06960931,  0.07038955, ..., -0.00384101,
         0.00034875,  0.02878492],
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
```

In [29]:
```
weights.shape
```

Out[29]:
```
(784, 300)
```

In [30]:
```
biases.shape
```

Out[30]:
```
(300,)
```

In [31]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

This is equivalent to:

```
model.compile(loss=keras.losses.sparse_categorical_crossentropy,
              optimizer=keras.optimizers.SGD(),
              metrics=[keras.metrics.sparse_categorical_accuracy])
```

In [32]:
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [==============================] - 4s 65us/sample - loss: 0.7226
- accuracy: 0.7642 - val_loss: 0.5075 - val_accuracy: 0.8314
Epoch 2/30
55000/55000 [==============================] - 3s 57us/sample - loss: 0.4843
- accuracy: 0.8321 - val_loss: 0.4538 - val_accuracy: 0.8486
Epoch 3/30
55000/55000 [==============================] - 3s 52us/sample - loss: 0.4413
- accuracy: 0.8465 - val_loss: 0.4385 - val_accuracy: 0.8490
Epoch 4/30
55000/55000 [==============================] - 3s 51us/sample - loss: 0.4128
- accuracy: 0.8549 - val_loss: 0.4163 - val_accuracy: 0.8562
Epoch 5/30
55000/55000 [==============================] - 3s 54us/sample - loss: 0.3926
- accuracy: 0.8617 - val_loss: 0.3817 - val_accuracy: 0.8636
Epoch 6/30
55000/55000 [==============================] - 3s 52us/sample - loss: 0.3770
- accuracy: 0.8667 - val_loss: 0.3729 - val_accuracy: 0.8680
Epoch 7/30
55000/55000 [==============================] - 3s 51us/sample - loss: 0.3626
- accuracy: 0.8733 - val_loss: 0.3699 - val_accuracy: 0.8710
Epoch 8/30
55000/55000 [==============================] - 3s 51us/sample - loss: 0.3517
- accuracy: 0.8749 - val_loss: 0.3666 - val_accuracy: 0.8696
Epoch 9/30
55000/55000 [==============================] - 3s 51us/sample - loss: 0.3420
- accuracy: 0.8772 - val_loss: 0.3438 - val_accuracy: 0.8786
Epoch 10/30
55000/55000 [==============================] - 3s 51us/sample - loss: 0.3326
- accuracy: 0.8814 - val_loss: 0.3511 - val_accuracy: 0.8794
Epoch 11/30
55000/55000 [==============================] - 3s 61us/sample - loss: 0.3241
- accuracy: 0.8835 - val_loss: 0.3357 - val_accuracy: 0.8816
Epoch 12/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.3159
- accuracy: 0.8871 - val_loss: 0.3310 - val_accuracy: 0.8846
Epoch 13/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.3073
- accuracy: 0.8903 - val_loss: 0.3325 - val_accuracy: 0.8826
Epoch 14/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.3017
- accuracy: 0.8920 - val_loss: 0.3243 - val_accuracy: 0.8880
Epoch 15/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.2953
- accuracy: 0.8937 - val_loss: 0.3173 - val_accuracy: 0.8892
Epoch 16/30
55000/55000 [==============================] - 3s 62us/sample - loss: 0.2898
- accuracy: 0.8966 - val_loss: 0.3251 - val_accuracy: 0.8888
Epoch 17/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.2833
- accuracy: 0.8987 - val_loss: 0.3178 - val_accuracy: 0.8920
Epoch 18/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.2781
- accuracy: 0.8999 - val_loss: 0.3102 - val_accuracy: 0.8908
Epoch 19/30
55000/55000 [==============================] - 3s 60us/sample - loss: 0.2729
- accuracy: 0.9021 - val_loss: 0.3205 - val_accuracy: 0.8836
Epoch 20/30
55000/55000 [==============================] - 3s 57us/sample - loss: 0.2680
- accuracy: 0.9044 - val_loss: 0.3219 - val_accuracy: 0.8852
Epoch 21/30
55000/55000 [==============================] - 3s 61us/sample - loss: 0.2635
- accuracy: 0.9041 - val_loss: 0.3007 - val_accuracy: 0.8944
Epoch 22/30
55000/55000 [==============================] - 3s 58us/sample - loss: 0.2575
- accuracy: 0.9076 - val_loss: 0.3103 - val_accuracy: 0.8878
Epoch 23/30
```

```
55000/55000 [==============================] - 3s 60us/sample - loss: 0.2538
- accuracy: 0.9084 - val_loss: 0.3001 - val_accuracy: 0.8910
Epoch 24/30
55000/55000 [==============================] - 3s 60us/sample - loss: 0.2492
- accuracy: 0.9103 - val_loss: 0.3096 - val_accuracy: 0.8870
Epoch 25/30
55000/55000 [==============================] - 3s 61us/sample - loss: 0.2452
- accuracy: 0.9121 - val_loss: 0.3114 - val_accuracy: 0.8892
Epoch 26/30
55000/55000 [==============================] - 4s 69us/sample - loss: 0.2408
- accuracy: 0.9145 - val_loss: 0.3263 - val_accuracy: 0.8850
Epoch 27/30
55000/55000 [==============================] - 3s 59us/sample - loss: 0.2367
- accuracy: 0.9151 - val_loss: 0.3117 - val_accuracy: 0.8852
Epoch 28/30
55000/55000 [==============================] - 4s 64us/sample - loss: 0.2323
- accuracy: 0.9177 - val_loss: 0.2921 - val_accuracy: 0.8950
Epoch 29/30
55000/55000 [==============================] - 4s 67us/sample - loss: 0.2288
- accuracy: 0.9190 - val_loss: 0.2970 - val_accuracy: 0.8920
Epoch 30/30
55000/55000 [==============================] - 4s 71us/sample - loss: 0.2256
- accuracy: 0.9192 - val_loss: 0.3016 - val_accuracy: 0.8902
```

In [33]:
```
history.params
```

Out[33]:
```
{'batch_size': 32,
 'epochs': 30,
 'steps': 1719,
 'samples': 55000,
 'verbose': 0,
 'do_validation': True,
 'metrics': ['loss', 'accuracy', 'val_loss', 'val_accuracy']}
```

In [34]:
```
print(history.epoch)
```

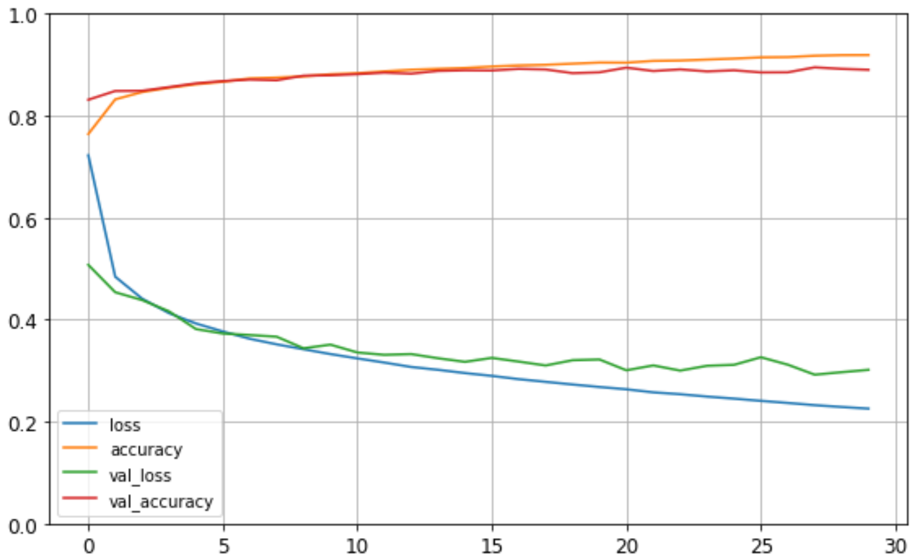
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
1, 22, 23, 24, 25, 26, 27, 28, 29]
```

In [35]:
```
history.history.keys()
```

Out[35]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [36]:
```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
save_fig("keras_learning_curves_plot")
plt.show()
```

Saving figure keras_learning_curves_plot



In [37]:
```
model.evaluate(X_test, y_test)
```

```
10000/10000 [==============================] - 0s 34us/sample - loss: 0.3348
- accuracy: 0.8788
```

Out[37]: `[0.3347936288356781, 0.8788]`

In [38]:
```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

Out[38]:
```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.98],
       [0.  , 0.  , 0.99, 0.  , 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

In [39]:
```
y_pred = model.predict_classes(X_new)
y_pred
```

Out[39]: `array([9, 2, 1], dtype=int64)`

In [40]:
```
np.array(class_names)[y_pred]
```

Out[40]: `array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')`

In [41]:
```
y_new = y_test[:3]
y_new
```

Out[41]: `array([9, 2, 1], dtype=uint8)`

In [42]:
```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_images_plot', tight_layout=False)
plt.show()
```

```
Saving figure fashion_mnist_images_plot
```

Regression MLP

Let's load, split and scale the California housing dataset (the original one, not the modified one as in chapter 2):

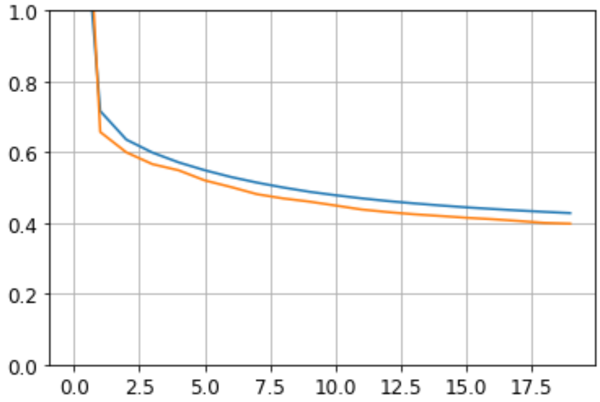
In [43]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full, y_train_full, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

In [44]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

In [45]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape
[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1
e-3))
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y
_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

```
Train on 11610 samples, validate on 3870 samples
Epoch 1/20
11610/11610 [==============================] - 1s 65us/sample - loss: 1.6205
- val_loss: 2.0374
Epoch 2/20
11610/11610 [==============================] - 0s 40us/sample - loss: 0.7162
- val_loss: 0.6571
Epoch 3/20
11610/11610 [==============================] - 0s 41us/sample - loss: 0.6356
- val_loss: 0.5996
Epoch 4/20
11610/11610 [==============================] - 0s 40us/sample - loss: 0.5989
- val_loss: 0.5662
Epoch 5/20
11610/11610 [==============================] - 0s 40us/sample - loss: 0.5713
- val_loss: 0.5489
Epoch 6/20
11610/11610 [==============================] - 0s 40us/sample - loss: 0.5491
- val_loss: 0.5204
Epoch 7/20
11610/11610 [==============================] - 0s 41us/sample - loss: 0.5301
- val_loss: 0.5018
Epoch 8/20
11610/11610 [==============================] - 0s 41us/sample - loss: 0.5142
- val_loss: 0.4815
Epoch 9/20
11610/11610 [==============================] - 0s 40us/sample - loss: 0.5004
- val_loss: 0.4695
Epoch 10/20
11610/11610 [==============================] - 0s 41us/sample - loss: 0.4883
- val_loss: 0.4605
Epoch 11/20
11610/11610 [==============================] - 0s 41us/sample - loss: 0.4786
- val_loss: 0.4495
Epoch 12/20
11610/11610 [==============================] - 0s 42us/sample - loss: 0.4697
- val_loss: 0.4382
Epoch 13/20
11610/11610 [==============================] - 0s 42us/sample - loss: 0.4621
- val_loss: 0.4309
Epoch 14/20
11610/11610 [==============================] - 0s 41us/sample - loss: 0.4556
- val_loss: 0.4247
Epoch 15/20
11610/11610 [==============================] - 0s 42us/sample - loss: 0.4497
- val_loss: 0.4200
Epoch 16/20
11610/11610 [==============================] - 1s 43us/sample - loss: 0.4443
- val_loss: 0.4149
Epoch 17/20
11610/11610 [==============================] - 0s 42us/sample - loss: 0.4397
- val_loss: 0.4108
Epoch 18/20
11610/11610 [==============================] - 0s 43us/sample - loss: 0.4354
- val_loss: 0.4059
Epoch 19/20
11610/11610 [==============================] - 0s 42us/sample - loss: 0.4315
- val_loss: 0.4003
Epoch 20/20
11610/11610 [==============================] - 0s 42us/sample - loss: 0.4281
- val_loss: 0.3981
5160/5160 [==============================] - 0s 22us/sample - loss: 0.4218
```

In [46]:
```
plt.plot(pd.DataFrame(history.history))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [47]:
```
y_pred
```

Out[47]:
```
array([[0.37310064],
       [1.6790789 ],
       [3.0817137 ]], dtype=float32)
```

Saving and Restoring

In [48]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

In [49]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=[8]),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
])
```

In [50]:
```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y
_valid))
mse_test = model.evaluate(X_test, y_test)
```

```
Train on 11610 samples, validate on 3870 samples
Epoch 1/10
11610/11610 [==============================] - 1s 73us/sample - loss: 1.8423
- val_loss: 5.2165
Epoch 2/10
11610/11610 [==============================] - 1s 44us/sample - loss: 0.6876
- val_loss: 0.7732
Epoch 3/10
11610/11610 [==============================] - 1s 44us/sample - loss: 0.5954
- val_loss: 0.5446
Epoch 4/10
11610/11610 [==============================] - 1s 49us/sample - loss: 0.5553
- val_loss: 0.5425
Epoch 5/10
11610/11610 [==============================] - 1s 56us/sample - loss: 0.5268
- val_loss: 0.5539
Epoch 6/10
11610/11610 [==============================] - 1s 50us/sample - loss: 0.5049
- val_loss: 0.4701
Epoch 7/10
11610/11610 [==============================] - 1s 45us/sample - loss: 0.4852
- val_loss: 0.4562
Epoch 8/10
11610/11610 [==============================] - 1s 45us/sample - loss: 0.4706
- val_loss: 0.4452
Epoch 9/10
11610/11610 [==============================] - 1s 46us/sample - loss: 0.4576
- val_loss: 0.4406
Epoch 10/10
11610/11610 [==============================] - 1s 45us/sample - loss: 0.4476
- val_loss: 0.4185
5160/5160 [==============================] - 0s 24us/sample - loss: 0.4376
```

In [51]:
```
model.save("my_keras_model.h5")
```

In [52]:
```
model = keras.models.load_model("my_keras_model.h5")
```

In [53]:
```
model.predict(X_new)
```

Out[53]:
```
array([[0.551559 ],
       [1.6555369],
       [3.0014234]], dtype=float32)
```

In [54]:
```
model.save_weights("my_keras_weights.ckpt")
```

In [55]:
```
model.load_weights("my_keras_weights.ckpt")
```

Out[55]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x22eb3b551
08>
```