Machine Learning with Python

Laboratory on Numpy / Matplotlib / Pandas / Scikit-learn

Introduction

In the past few years, Python has become the de-facto standard programming language for data analytics. Python's success is due to several factors, but one major reason has been the availability of powerful, open-source libraries for scientific computation such as Numpy, Scipy and Matplotlib. Python is also the most popular programming language for machine learning, thanks to libraries such as Scikit-learn, TensorFlow and PyTorch.

In this lecture we will explore the basics of Numpy, Matplotlib and Scikit-learn. The first is a library for data manipulation through the powerfull `numpy.ndarray` data structure; the second is useful for graphical visualization and plotting; the third is a general purpose library for machine learning, containing dozens of algorithms for classification, regression, clustering and others.

In this lecture we assume familiarity with the Python programming language. If you are not familiar with the language, we advise you to look it up before carrying over to the next sections. Here are some useful links to learn about Python:

- https://docs.python.org/3/tutorial/introduction.html (https://docs.python.org/3/tutorial/introduction.html)
- https://www.learnpython.org/ (https://www.learnpython.org/)
- http://www.scipy-lectures.org/ (http://www.scipy-lectures.org/)

If you have never seen a page like this, it is a **Jupyther Notebook**. Here one can easily embed Python code and run it on the fly. You can run the code in a cell by selecting the cell and clicking the *Run* button (top). You can do the same using the **SHIFT+Enter** shortcut. You can modify the existing cells, run them and finally save

your changes.

Requirements

1. Python (preferably version > 3.3): https://www.python.org/downloads/ (https://www.python.org/downloads/)
2. Numpy, Scipy and Matplotlib: https://www.scipy.org/install.html (https://www.scipy.org/install.html)
3. Scikit-learn: http://scikit-learn.org/stable/install.html (http://scikit-learn.org/stable/install.html)
4. Pandas: https://pandas.pydata.org/docs/getting_started/index.html (https://pandas.pydata.org/docs/getting_started/index.html)

References

- https://docs.scipy.org/doc/numpy/ (https://docs.scipy.org/doc/numpy/)
- https://docs.scipy.org/doc/scipy/reference/ (https://docs.scipy.org/doc/scipy/reference/)
- https://matplotlib.org/users/index.html (https://matplotlib.org/users/index.html)
- http://scikit-learn.org/stable/documentation.html (http://scikit-learn.org/stable/documentation.html)

Numpy

Numpy provides high-performance data structures for data manipulation and numeric computation. In particular, we will look at the `numpy.ndarray`, a data structure for manipulating vectors, matrices and tensors. Let's start by importing `numpy`:

In [171]:

```
# the np alias is very common
import numpy as np
```

We can initialize a Numpy array from a Python list using the `numpy.array` function:

In [172]:

```
# if the argument is a list of numbers, the array will be a 1-dimensional vector
a = np.array([1, 2, 3, 4, 5, 6])

a
```

Out[172]:

```
array([1, 2, 3, 4, 5, 6])
```

In [173]:

```
# if the argument is a list of lists, the array will be a 2-dimensional matrix
M = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])

M
```

Out[173]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

Given a Numpy array, we can check its `shape`, a tuple containing the number of elements for each dimension:

In [174]:

```
a.shape
```

Out[174]:

```
(6,)
```

In [175]:

```
M.shape
```

Out[175]:

```
(4, 4)
```

We can do quite some nice things with Numpy arrays that are not possible with standard Python lists.

Indexing

Numpy array allow us to index arrays in quite advanced ways.

In [176]:

```
# A 1d vector can be indexed in all the common ways
a[0]
```

Out[176]:

```
1
```

In [177]:

```
a
```

Out[177]:

```
array([1, 2, 3, 4, 5, 6])
```

In [178]:

```
a[1:3]
```

Out[178]:

```
array([2, 3])
```

In [179]:

```
# Use a boolean mask
mask = [True, False, False, True, True, False]
a[mask]
```

Out[179]:

```
array([1, 4, 5])
```

The power of Numpy indexing capabilities starts showing up with 2d arrays:

In [180]:

```
# Access a single element of the matrix
M[0, 1]
```

Out[180]:

```
2
```

In [181]:

```
# Access an entire row
M[1]
```

Out[181]:

```
array([5, 6, 7, 8])
```

In [182]:

```
# Access an entire column
M[:,2]
```

Out[182]:

```
array([ 3,  7, 11, 15])
```

In [183]:

```
# Extract a sub-matrix
M[1:3, 0:2]
```

Out[183]:

```
array([[ 5,  6],
       [ 9, 10]])
```

Data manipulation

We can manipulate data in several ways.

In [184]:

```
# Flatten a matrix
M.flatten()
```

Out[184]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 1
6])
```

In [185]:

```
# Reshaping a matrix
M.reshape(2, 8)
```

Out[185]:

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 9, 10, 11, 12, 13, 14, 15, 16]])
```

In [186]:

```
# Computing the max and the min
M.max(), M.min()
```

Out[186]:

```
(16, 1)
```

In [187]:

```
# Computing the mean and standard deviation
M.mean(), M.std()
```

Out[187]:

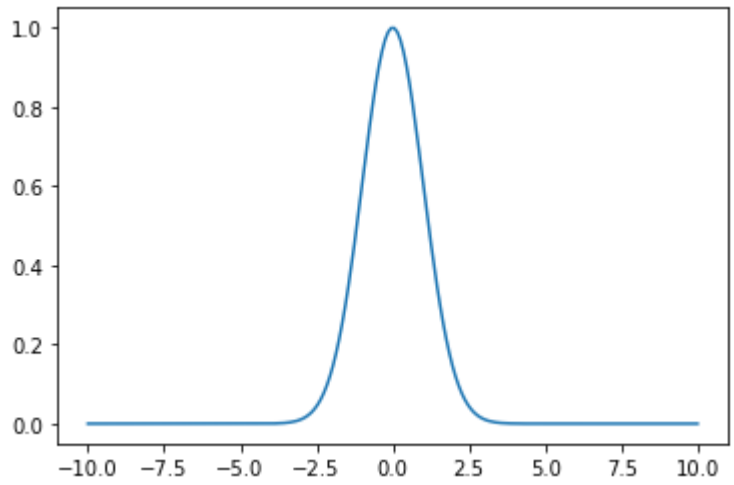
```
(8.5, 4.6097722286464435)
```

Matplotlib

Matplotlib is a powerful library for data visualization. Let's plot the above function.

In [188]:

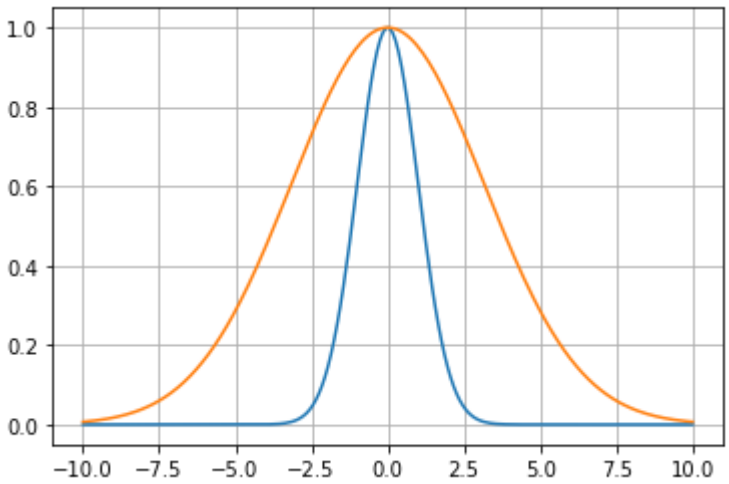
```
# the following line is only needed to show plots in the notebook
%matplotlib inline
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 200)    # get a sample of the x axis
y = np.exp(-(x**2)/(2*1))        # compute the function for all points in the sampl
plt.plot(x, y)                   # add the curve to the plot
plt.ylim(-0.05,1.05)             # set bottom and top limits for y axis
plt.show()                       # show the plot
```



We can also plot more than one line in the same figure and add a grid to the plot.

In [189]:

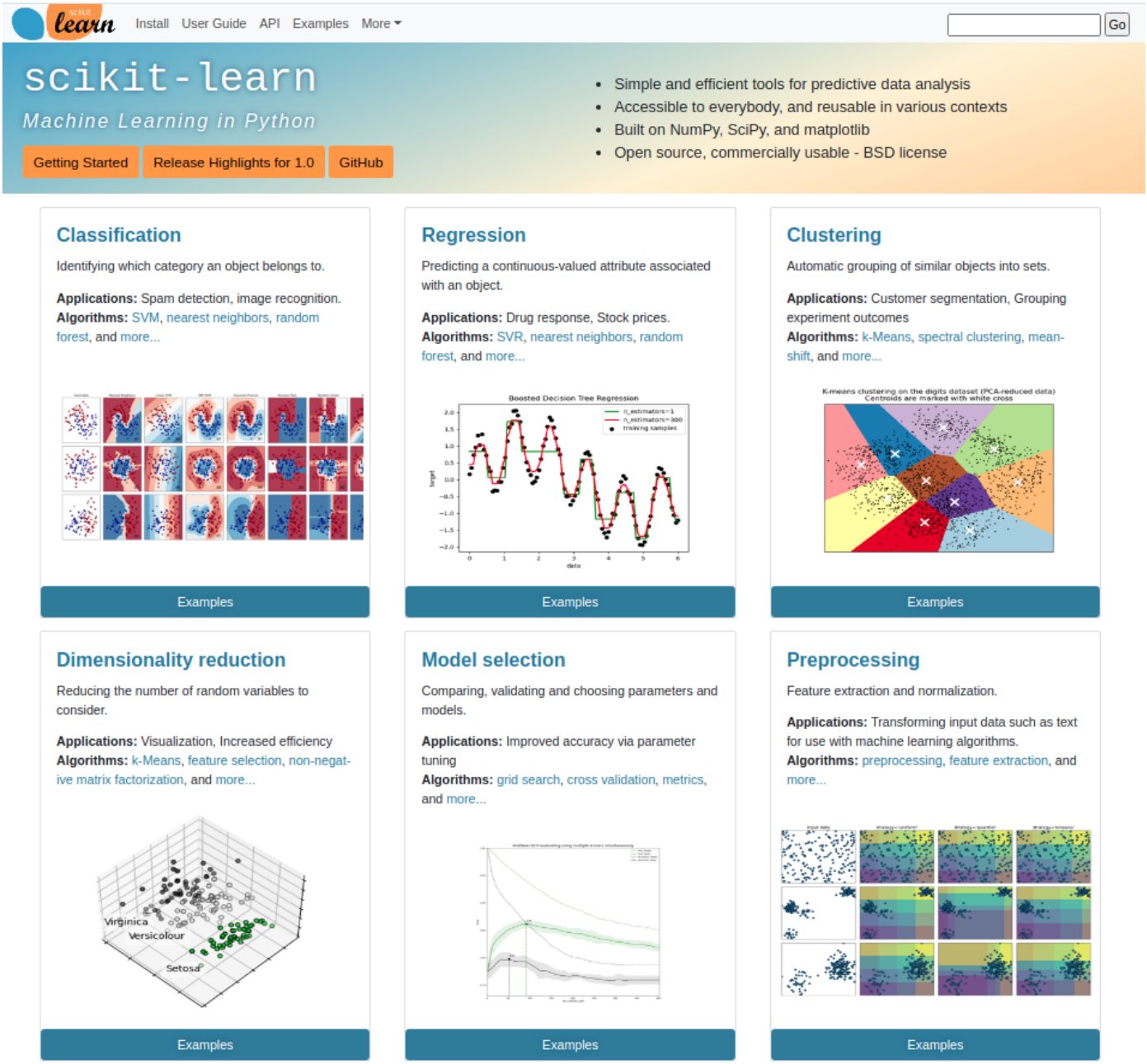
```
z = np.exp(-(x**2)/(2*10))
plt.grid()                  # add the grid under the curves
plt.plot(x, y)              # add the first curve to the plot
plt.plot(x, z)              # add the second curve to the plot
plt.ylim(-0.05,1.05)            # set bottom and top limits for y axis
plt.show()                  # show the plot
```



We can also set several properties of the plot in this way:

Scikit-learn

Let's now dive into the real **Machine Learning** part. *Scikit-learn* is perhaps the most wide-spread library for Machine Learning in use nowadays, and most of its fame is due to its extreme simplicity. With Scikit-learn it is possible to easily manage datasets, and train a wide range of classifiers out-of-the-box. It is also useful for several other Machine Learning tasks such as regression, clustering, dimensionality reduction, and model selection.



in this lab:

1. Classification
    A. SVM
    B. Decision Tree
    C. Random forest
2. K-Fold cross validation
3. Clustering
    A. K-mean
4. Dim reduction
    A. PCA

Supervised Learning

SVM

Load dataset

In [190]:

```
# load the data

data = np.load("data/data_SVM.npy",allow_pickle=True)

print("element\t", data[0][0])
print("label  \t", data[0][1])
```

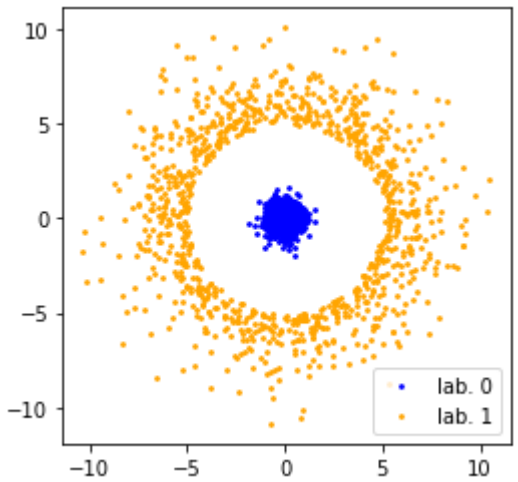
```
element  [0.12813421 0.40582761]
label    0.0
```

In [191]:

```
# extract X and y
X = data[:,0]
y = data[:,1]

X = [list(x) for x in X] # convert np.array in list
y = [int(x) for x in y]  # convert float in int

X = np.array(X)
y = np.array(y)
```

In [192]:

```
# data visulization
plt.figure(figsize=(4,4))
plt.scatter(X[0:1000,0],X[0:1000,1], s = 3, color = "blue",label="lab. 0")
plt.scatter(X[1000:,0],X[1000:,1], s = 3, color = "orange",label="lab. 1")
plt.legend()
plt.show()
```



In [193]:

```
# train test split
try:
    from sklearn.model_selection import train_test_split
except ImportError:
    from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
print("len X_train\t", len(X_train))
print("len X_test \t", len(X_test))
```

```
len X_train      1600
len X_test       400
```

SVM 1

In [194]:

```
from sklearn.svm import SVC

# Instantiate the model specifying the kernel
# With linear kernel
clf = SVC(kernel='linear')

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)
```

In [195]:

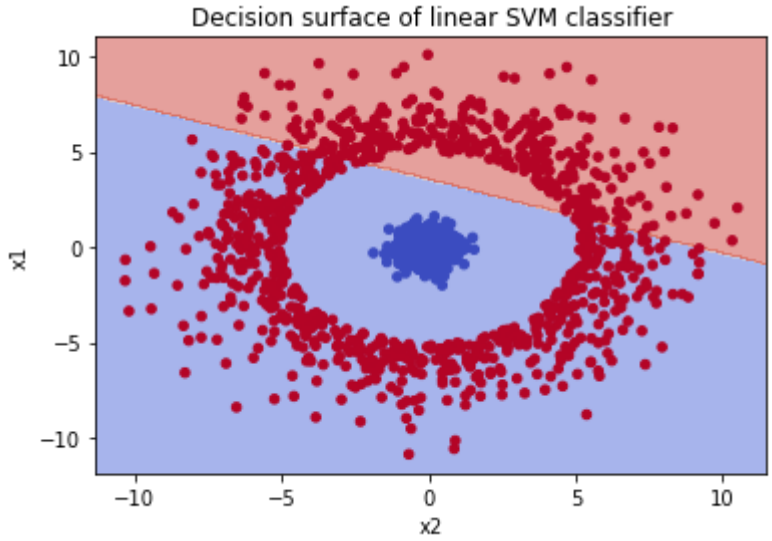
```
def make_meshgrid(x, y, h=.1):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out
```

In [ ]:

```

```

In [196]:

```
# Decision surface
fig, ax = plt.subplots()
X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.5)
ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20)
ax.set_ylabel('x1')
ax.set_xlabel('x2')
ax.set_title('Decision surface of linear SVM classifier')
plt.show()
```



In [197]:

```
# evaluate teh accuracy
from sklearn import metrics

metrics.accuracy_score(y_test, y_pred)
```

Out[197]:

0.67

Can we do better?

SVM 2

In [198]:

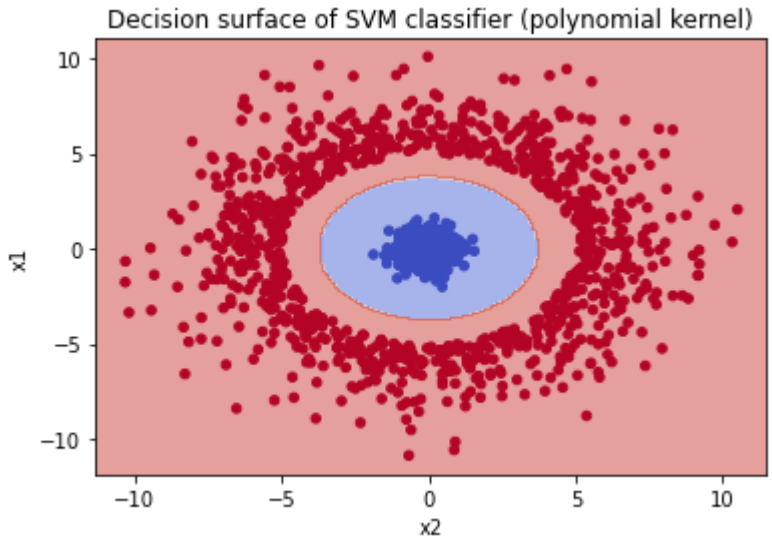
```
from sklearn.svm import SVC

# Specify the parameters in the constructor
# The poly kernel is the polynomial kernel;
# The poly kernel takes one parameter: degree


clf = SVC(kernel='poly',degree=2)
# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)


# compute accuracy
metrics.accuracy_score(y_test, y_pred)
```

Out[198]:

```
1.0
```

In [199]:

```
# Decision surface
fig, ax = plt.subplots()
X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.5)
ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20)
ax.set_ylabel('x1')
ax.set_xlabel('x2')
ax.set_title('Decision surface of SVM classifier (polynomial kernel)')
plt.show()
```



Feature mapping

Now we are creating a new dataset in 3 dim. instead of 2 dim.

$$F : \mathbf{R}^2 \rightarrow \mathbf{R}^3$$
$$x_i \in \mathbf{R}^2 \rightarrow x_i' \in \mathbf{R}^3$$
$$F(x_{i1}, x_{i2}) \rightarrow (x_{i1}^2, x_{i1} \cdot x_{i2}, x_{i2}^2)$$

In [200]:

```
X3d = []
for a in X:
    X3d.append([a[0]**2,a[0]*a[1],a[1]**2])

X3d = np.array(X3d)
```

In [201]:

```
X_train, X_test, y_train, y_test = train_test_split(X3d, y, test_size=0.2, random_s
```

SVM 3 (3d)

In [202]:

```
clf = SVC(kernel='linear')

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)
```

In [203]:

```
metrics.accuracy_score(y_test, y_pred)
```

Out[203]:

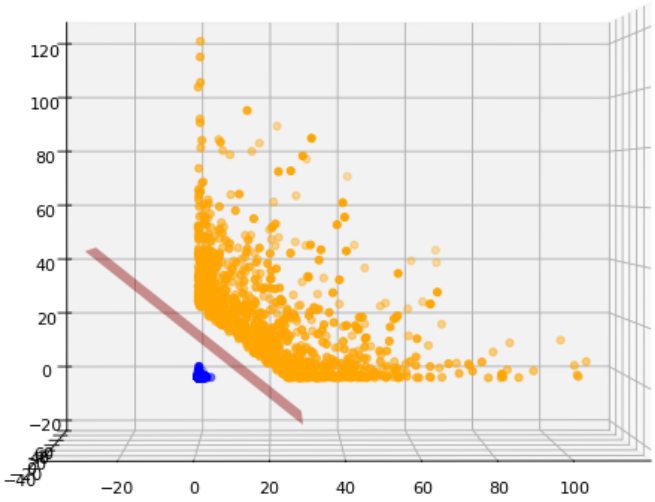
```
1.0
```

In [ ]:

```

```

In [204]:

```
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D
import random
%matplotlib notebook


fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')


ax.scatter(X3d[0:1000:,0], X3d[0:1000,1], X3d[0:1000,2],color="blue")
ax.scatter(X3d[1000:,0], X3d[1000:,1], X3d[1000:,2],color="orange")

xx, yy = np.meshgrid(range(-30,30), range(-30,30))
z = -xx
ax.plot_surface(xx, yy, z+15, color="red",alpha=0.5)

plt.show()
```



Decision Tree

In [12]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import metrics

def make_meshgrid(x, y, h=.1):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out
```

In [13]:

```
# load the data
data = np.load("data/data_SVM.npy",allow_pickle=True)

# extract X and y
X = data[:,0]
y = data[:,1]

X = [list(x) for x in X] # convert np.array in list
y = [int(x) for x in y]  # convert float in int

X = np.array(X)
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [14]:

```
from sklearn.tree import DecisionTreeClassifier
```

In [15]:

```
clf = DecisionTreeClassifier()

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)
```

In [16]:

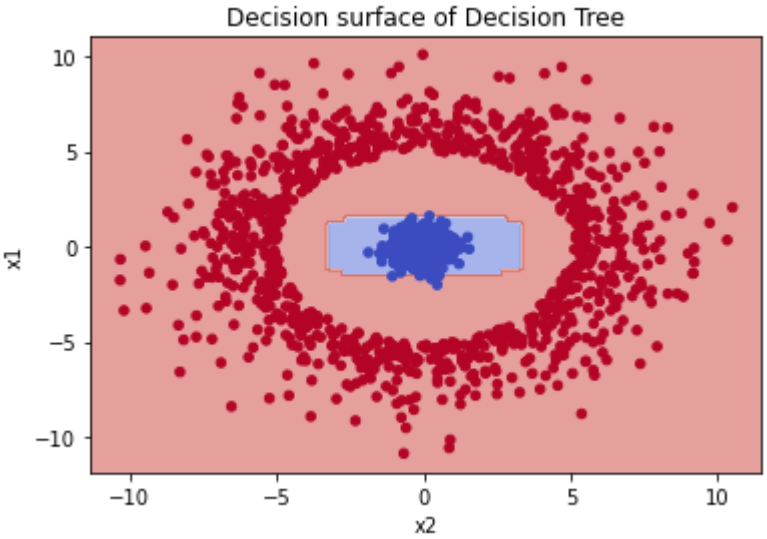
```
metrics.accuracy_score(y_test, y_pred)
```

Out[16]:

```
0.9925
```

In [17]:

```
# Decision surface
fig, ax = plt.subplots()
X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.5)
ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20)
ax.set_ylabel('x1')
ax.set_xlabel('x2')
ax.set_title('Decision surface of Decision Tree')
plt.show()
```



Random Forest

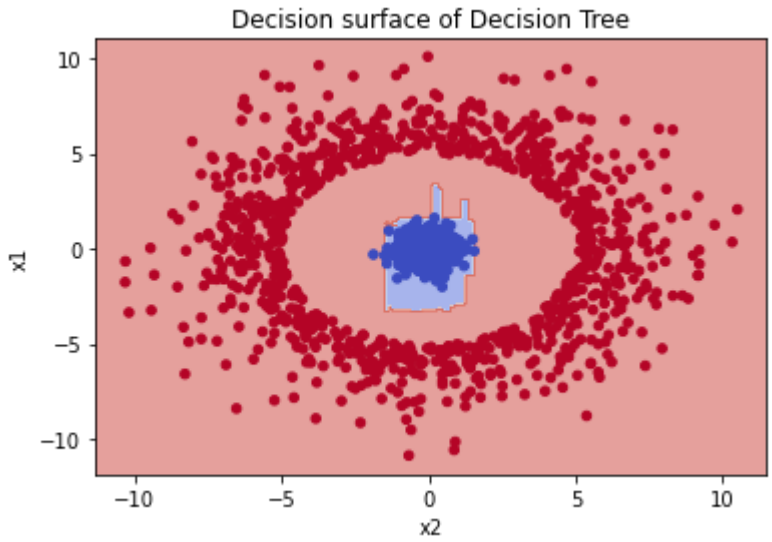
In [18]:

```
from sklearn.ensemble import RandomForestClassifier
```

In [19]:

```
clf = RandomForestClassifier(n_estimators=5,max_depth=10)

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)

print(metrics.accuracy_score(y_test, y_pred))

# Decision surface
fig, ax = plt.subplots()
X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.5)
ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20)
ax.set_ylabel('x1')
ax.set_xlabel('x2')
ax.set_title('Decision surface of Decision Tree')
plt.show()
```

```
0.9975
```



Imbalanced dataset?

In [20]:

```
import pandas as pd

data = pd.read_csv("data/glass.dat")
print(len(data),np.unique(data.type))
data.head()
```

```
214 [0 1]
```

Out[20]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	type
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	0
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	0
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	0
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	0
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	0

In [21]:

```
X = data.to_numpy()[:,:9]
y = data.to_numpy()[:,-1]
```

In [22]:

```
# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
print("len X_train\t", len(X_train))
print("len X_test \t", len(X_test))
```

```
len X_train	 171
len X_test 	 43
```

In [23]:

```
from sklearn.svm import SVC

# Instantiate the model specifying the kernel
clf = SVC()

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)
```

In [24]:

```
print(metrics.accuracy_score(y_test, y_pred))
```

```
0.9069767441860465
```

accuracy equal 0.9, it is amazing!

not really ;(

do you remember f1-score?

f1 = harmonic mean of the precision and recall.

$$f1 = 2 \cdot \frac{(precision \cdot recall)}{(precision + recall)}$$

In [25]:

```
print(metrics.f1_score(y_pred,y_test))
```

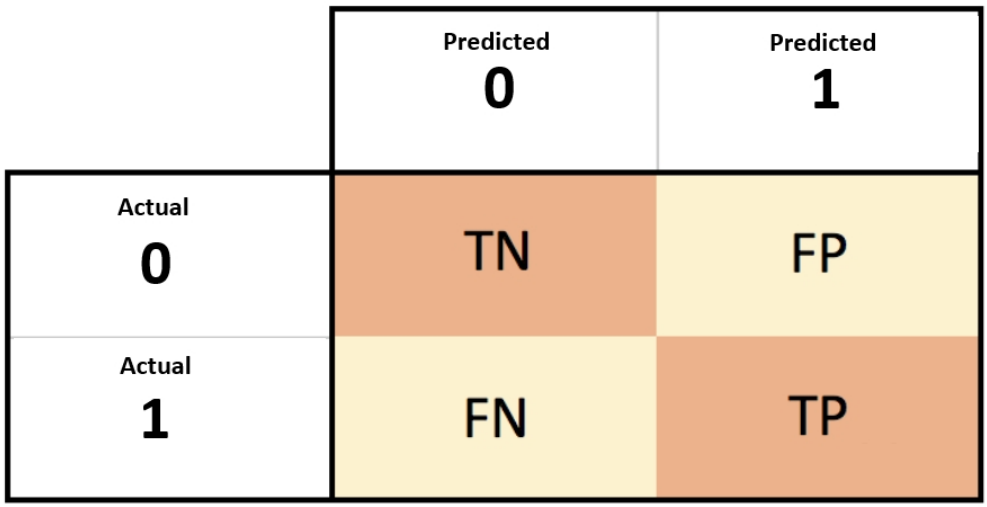
```
0.0
```

In [26]:

```
print(metrics.recall_score(y_pred,y_test)) # recall = tp / (tp + fn)
print(metrics.precision_score(y_pred,y_test)) # precision =  tp / (tp + fp)
```

```
0.0
0.0

/home/antonio/.local/lib/python3.8/site-packages/sklearn/metrics/_clas
sification.py:1248: UndefinedMetricWarning: Recall is ill-defined and
being set to 0.0 due to no true samples. Use `zero_division` parameter
to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

do you remember the confusion matrix?



In [27]:

```
conf_mat = metrics.confusion_matrix(y_test,y_pred)
print("confusion matrix \n",conf_mat,"\n")
print("True negative    \t",conf_mat[0][0],"--> they are predicted as negative, and
print("False positive   \t",conf_mat[0][1]," --> they are predicted as positive, bu
print("False negative  \t",conf_mat[1][0]," --> they are predicted as negative, but
print("True positive   \t",conf_mat[1][1]," --> they are predicted as positive, and
```

```
confusion matrix
 [[39  0]
 [ 4  0]]

True negative            39 --> they are predicted as negative, and th
ey are really negative :)
False positive           0  --> they are predicted as positive, but th
ey are negative :(
False negative           4  --> they are predicted as negative, but th
ey are positive :(
True positive            0  --> they are predicted as positive, and th
ey are really positive :)
```

In [28]:

```
# have a look at the predictions:

print(y_pred) # the classifier predicts only one classe (the class zero)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

In [29]:

```
# have a look at our dataset

print(np.unique(y_train,return_counts=True))
# the dataset is not balanced
```

```
(array([0., 1.]), array([162,   9]))
```

Give different weight for class perdiction

In [30]:

```
neg = np.unique(y_train,return_counts=True)[1][0]
pos = np.unique(y_train,return_counts=True)[1][1]

print(pos,neg)
```

```
9 162
```

In [ ]:

```

```

In [31]:

```
from sklearn.svm import SVC

# Instantiate the model specifying the kernel
clf = SVC(class_weight={0: 1,1 :neg/pos})

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test)
print(y_pred)
print(metrics.accuracy_score(y_pred,y_test))
conf_mat = metrics.confusion_matrix(y_test,y_pred)
print(conf_mat)
```

```
[0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1.
1.
 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0.]
0.7906976744186046
[[30  9]
 [ 0  4]]
```

In [32]:

```
print(metrics.accuracy_score(y_test,y_pred))
print(metrics.f1_score(y_test,y_pred))
print(metrics.recall_score(y_test,y_pred))
print(metrics.precision_score(y_test,y_pred))
```

```
0.7906976744186046
0.47058823529411764
1.0
0.3076923076923077
```

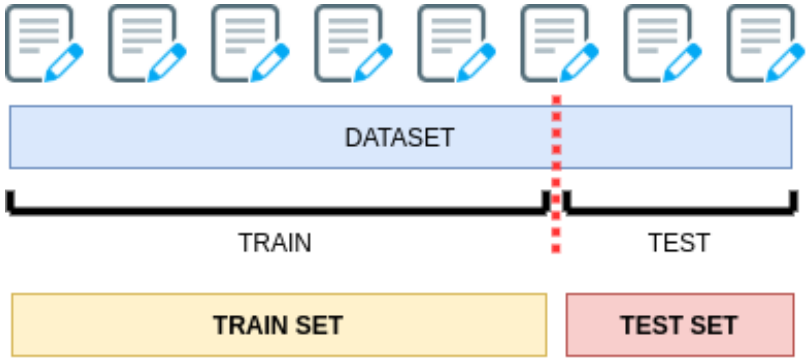
K fold, cross validation

In real word, we can not test on test set, It would be like **cheating**

what we have to do?

suppose we have a dataset like the follwoing:

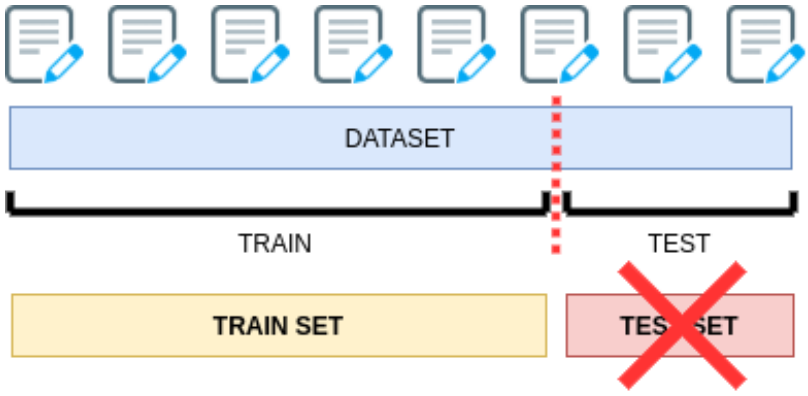
So far, we splitted the dataset in test and train set:



usally we use the train set for training our model, and we evaluate our model on the test set. However, this procedure is like **cheating**

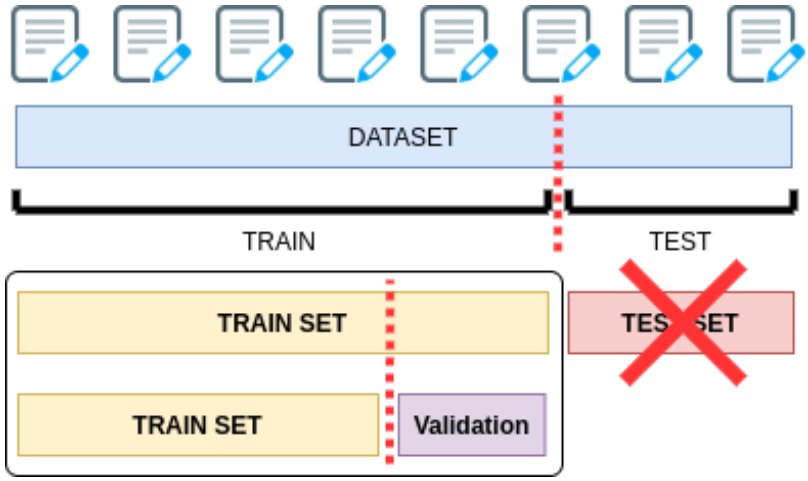
---

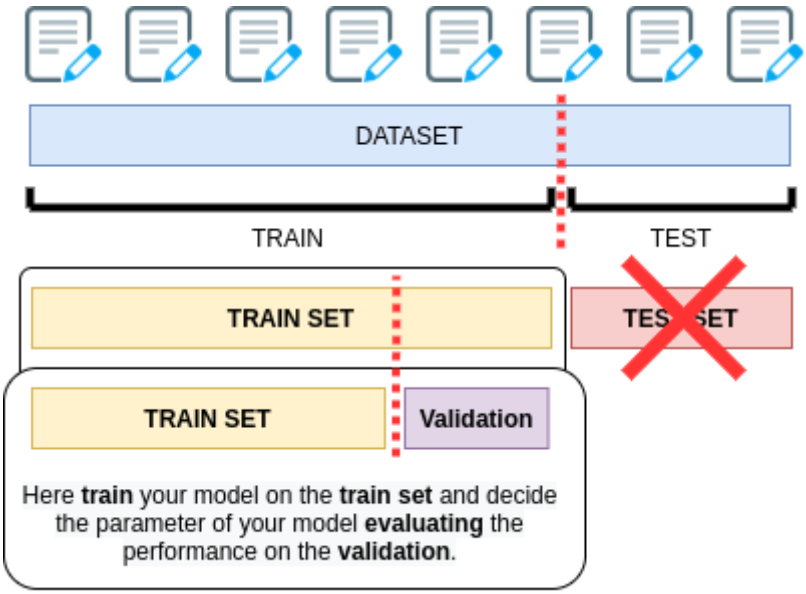
so, what we have to do?



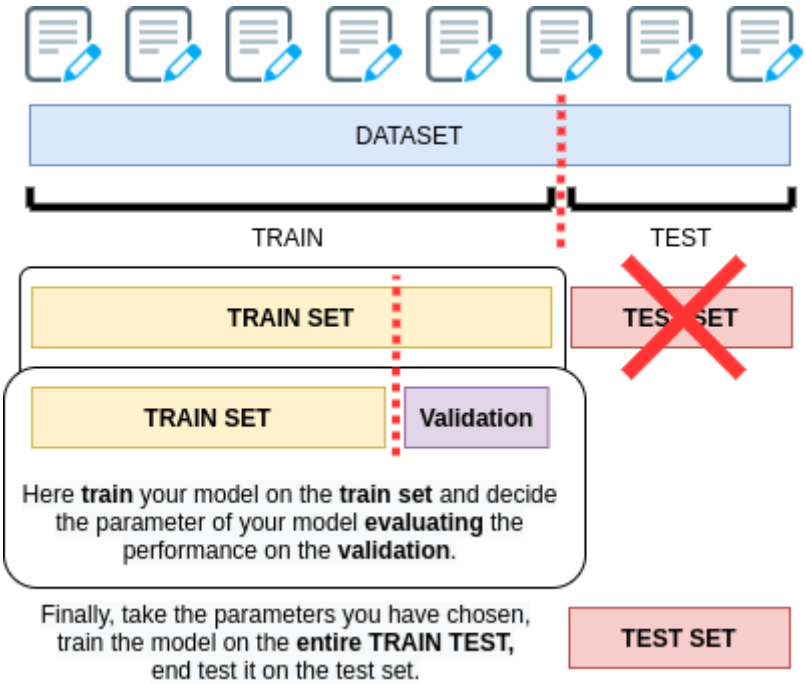
we forget about the test set.

Now we have to split again the train set obtaining the validation test





finally, when you decided your parameter on the train-validation set, you can retrain the model on the **entire** train set and evaluate on the test set



In [33]:

```
X_train, X_test_secret, y_train, y_test_secret = train_test_split(X, y, test_size=0
```

In [34]:

```
X_train2,X_validation , y_train2, y_validation = train_test_split(X_train,y_train,t

print("X train     ", len(X_train))
print("X test      ", len(X_test))
print("X train2    ", len(X_train2))
print("X validation", len(X_validation))

print("\nX train = X_train2 + X_validation")
print(len(X_train),"=",len(X_validation)+len(X_train2))
```

```
X train      171
X test       43
X train2     136
X validation 35

X train = X_train2 + X_validation
171 = 171
```

In [35]:

```
clf = SVC(class_weight={0: 1,1 :neg/pos}, kernel='rbf', gamma=0.1)

# Training
clf.fit(X_train2, y_train2)

# Prediction
y_pred = clf.predict(X_validation)

print(metrics.f1_score(y_pred,y_validation))
print(metrics.accuracy_score(y_pred,y_validation))
```

```
0.4
0.9142857142857143
```

we decided for gamma = 0.01

In [36]:

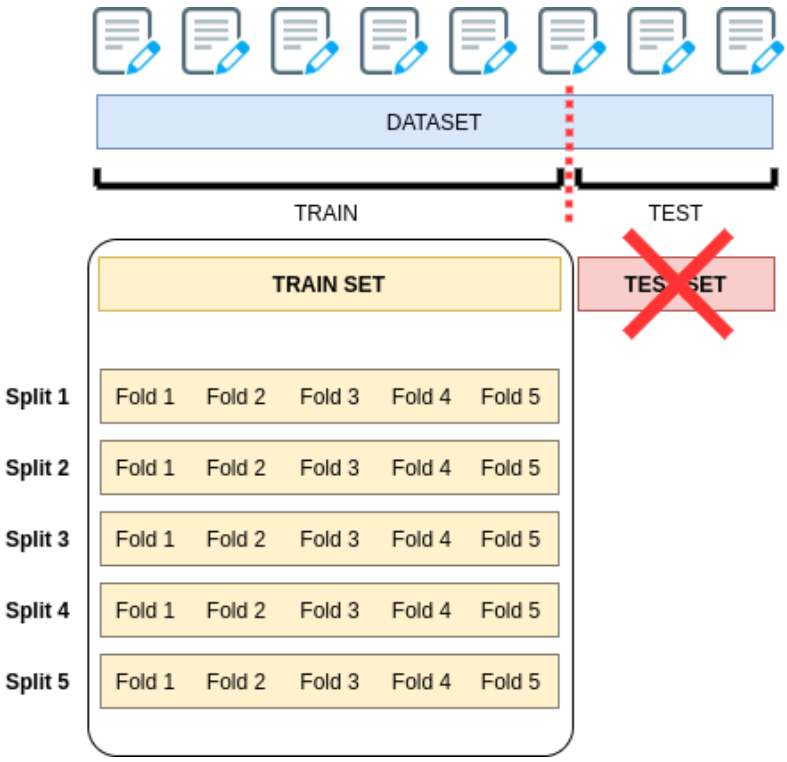
```
clf = SVC(class_weight={0: 1,1 :neg/pos}, kernel='rbf', gamma=0.01)

# Training
clf.fit(X_train, y_train)

# Prediction
y_pred = clf.predict(X_test_secret)

print(metrics.f1_score(y_pred,y_test_secret))
print(metrics.accuracy_score(y_pred,y_test_secret))
```

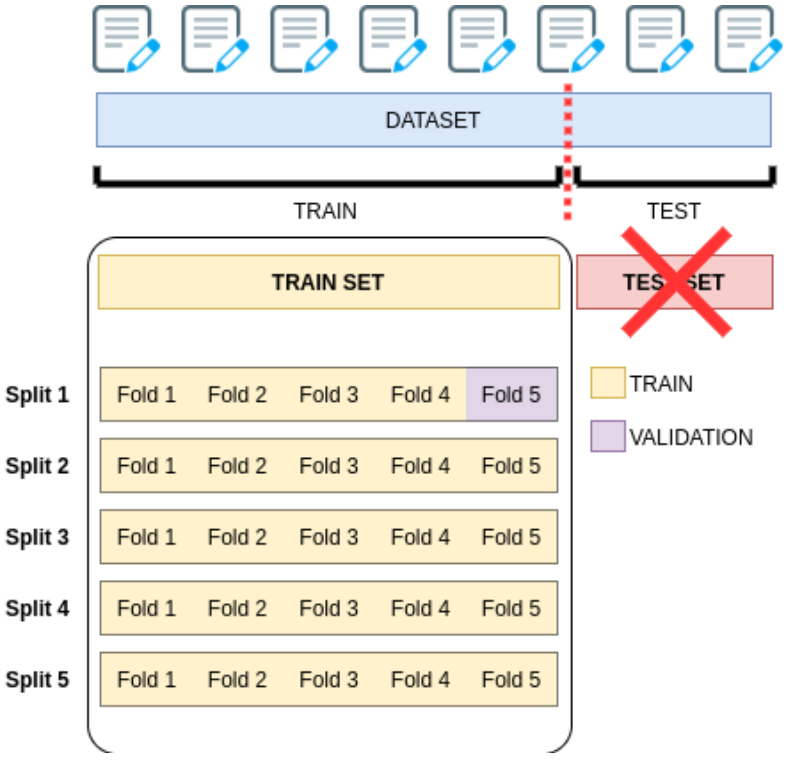
```
0.6153846153846153
0.8837209302325582
```

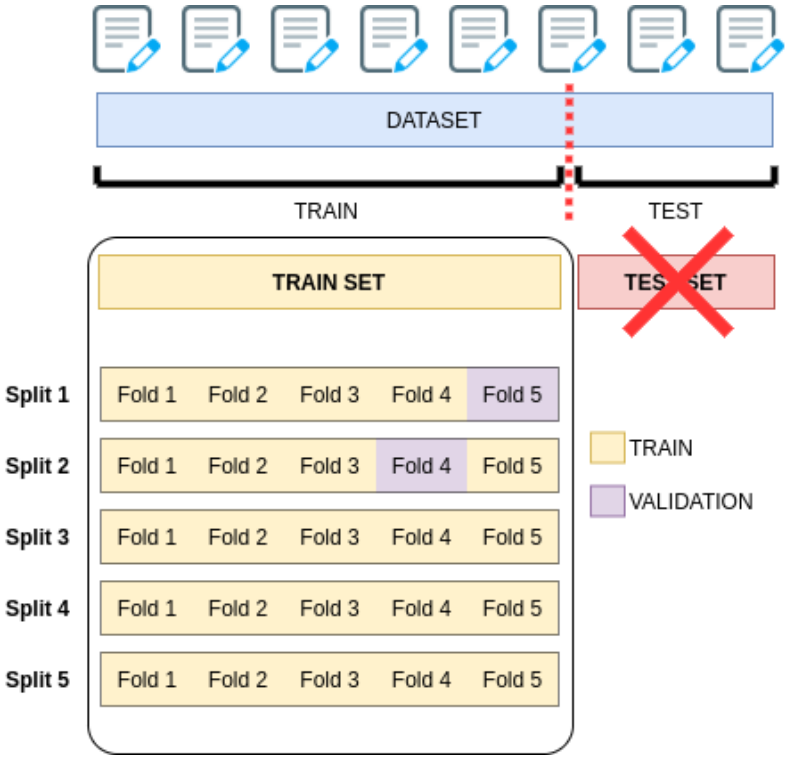
Can we do better?

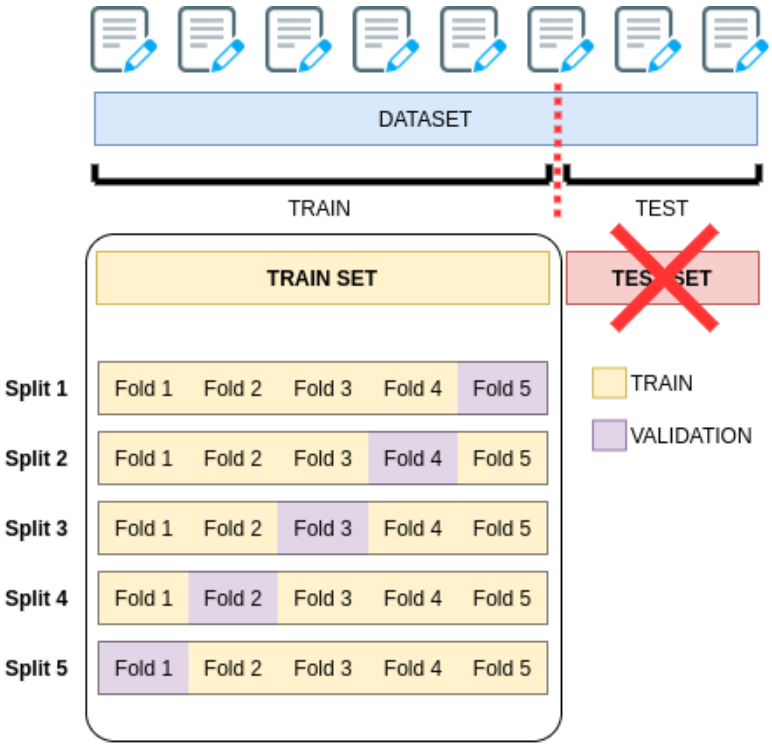
yes, k-fold cross validation

SUPPOSE K = 5









at the end of the day, we will have k (5) metrics for each experiment.

So we can compute mean and standard deviation, and study the best parameter of the model.

finally, we can retrain the entire train set with the choosen parameters and test it on test set

In [37]:

```
from sklearn.model_selection import KFold, cross_val_score
X_train, X_test_secret, y_train, y_test_secret = train_test_split(X, y, test_size=0

kf = KFold(n_splits=5, shuffle=True, random_state=42)

clf = SVC(class_weight={0: 1,1 :neg/pos}, kernel='rbf', gamma=0.01)

scores = cross_val_score(clf, X_train, y_train, cv=kf.split(X_train), scoring="f1")
print(scores)
print(np.mean(scores),np.std(scores))
```

```
[0.75       0.66666667 0.         0.4        0.5       ]
0.46333333333333326 0.26212804335116663
```

In [38]:

```
# retrain the entire model
clf = SVC(class_weight={0: 1,1 :neg/pos}, kernel='rbf', gamma=0.015)
clf = clf.fit(X_train,y_train)

y_pred = clf.predict(X_test_secret)

print(metrics.f1_score(y_pred,y_test_secret))
```

```
0.6666666666666666
```

Conclusion Supervised Learning

We tested SVM in a toy example, then we compared the results of SVM, Decision Tree and random forest. Later, we loaded an unbalanced dataset and we tested SVM showing how the accuracy can be misleading. Finally, we present k-fold cross validation

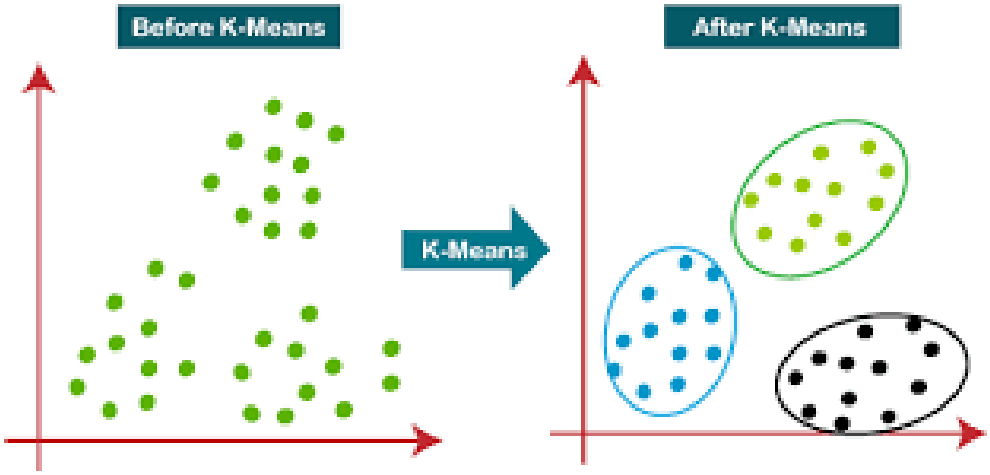
---

Unsupervised Learning

What can we do if we do not have labels?

K-means clustering

AIM:



INPUT: Data ($X \in \mathbf{R}^d$), K (number of clusters)
PROBLEM: Split X in K partitions: $S_1, S_2, \ldots, S_k$

FORMALLY: each partition $S_i \subset X$ has a center point (centroid c_i)

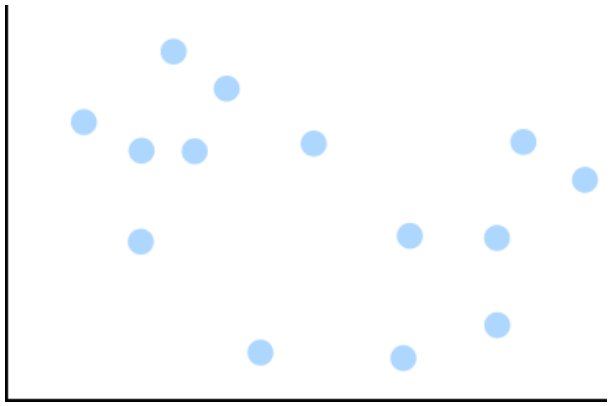
we would like to solve the following problem:

$$argmin_S \sum_{i=1}^{K} \sum_{x \in S_i} dist(x, c_i)$$

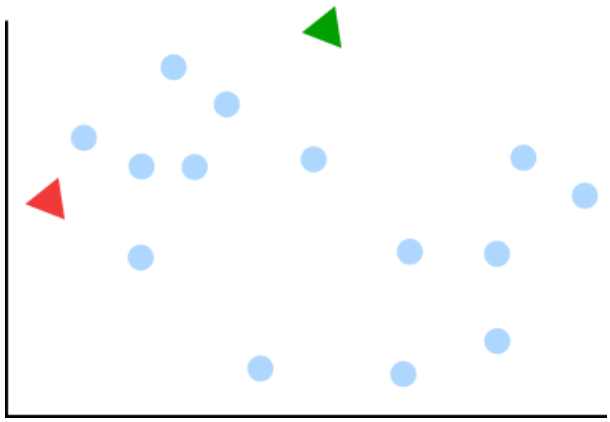
where c_i is the mean of points in S_i. Suppose to use the norm 2 distance $dist(x, c_i) = ||x - c_i||^2$

The algorithm

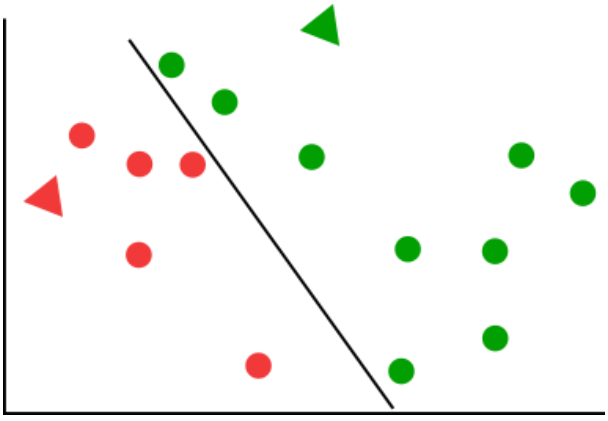
the input data



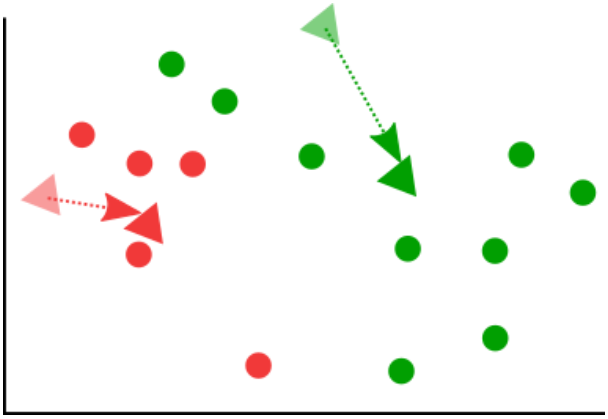
start with random clusters



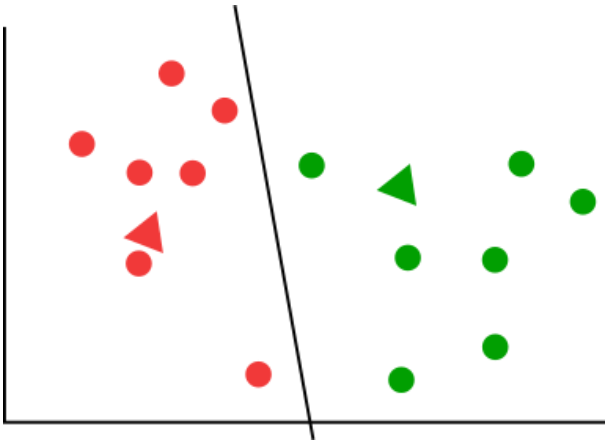
assign points to the closest centroid



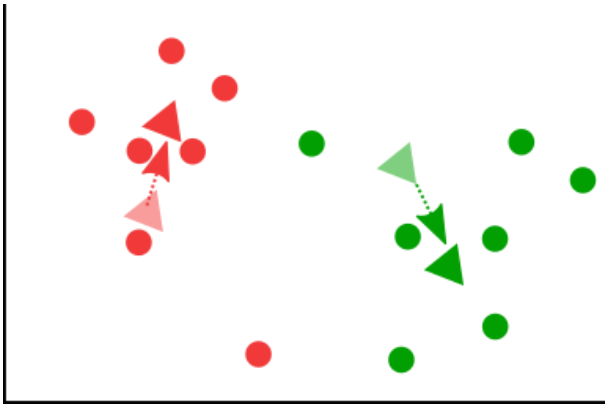
update centroid values



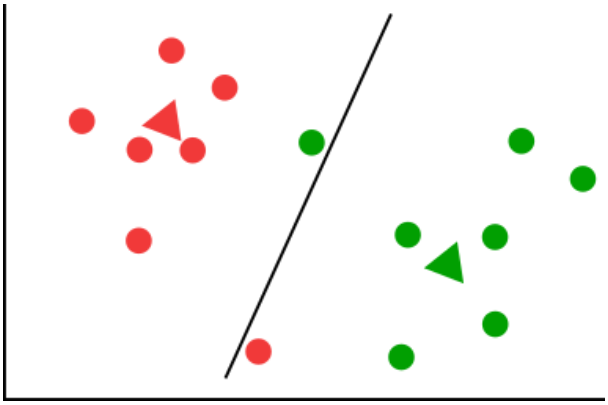
assign points to the closest centroid



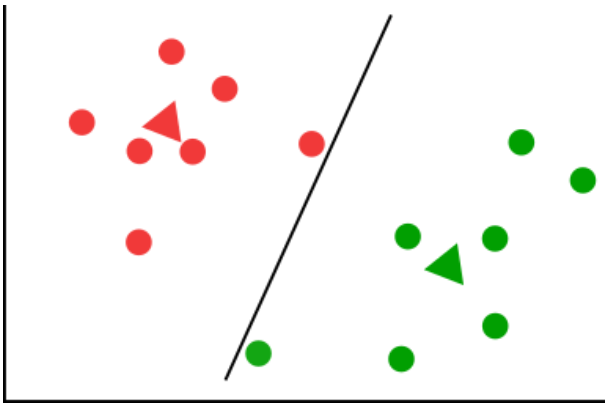
update centroid values



assign points to the closest centroid



update centroid values



K means code

In [39]:

```
import pandas as pd
```

In [40]:

```
data = pd.read_csv("data/Mall_Customers.csv")
data.head()
```

Out[40]:

	CustomerID	Gender	Age	Income	Spending
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

In [41]:

```
data = data.drop("CustomerID",axis=1)
data.head()
```

Out[41]:

	Gender	Age	Income	Spending
0	Male	19	15	39
1	Male	21	15	81
2	Female	20	16	6
3	Female	23	16	77
4	Female	31	17	40

In [42]:

```
data.Gender = pd.Categorical(data.Gender)
data['gender_code'] = data.Gender.cat.codes

data.head()
```

Out[42]:

	Gender	Age	Income	Spending	gender_code
0	Male	19	15	39	1
1	Male	21	15	81	1
2	Female	20	16	6	0
3	Female	23	16	77	0
4	Female	31	17	40	0

In [43]:

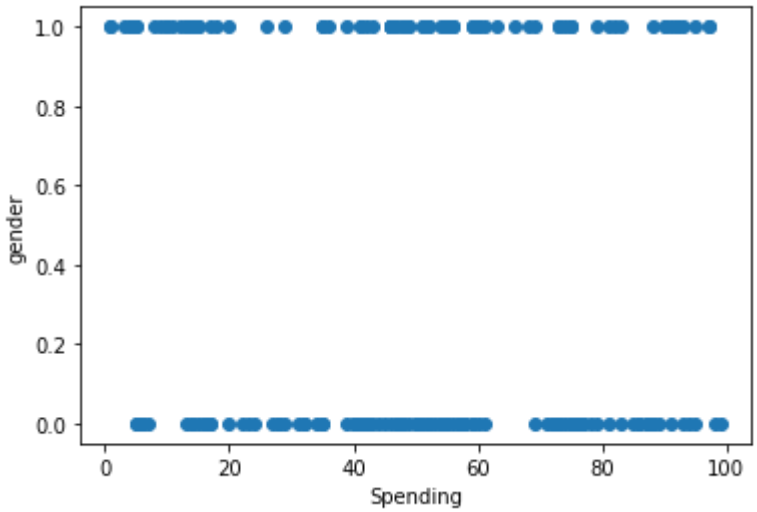
```
data = data.drop("Gender",axis=1)
data.head()
```

Out[43]:

	Age	Income	Spending	gender_code
0	19	15	39	1
1	21	15	81	1
2	20	16	6	0
3	23	16	77	0
4	31	17	40	0

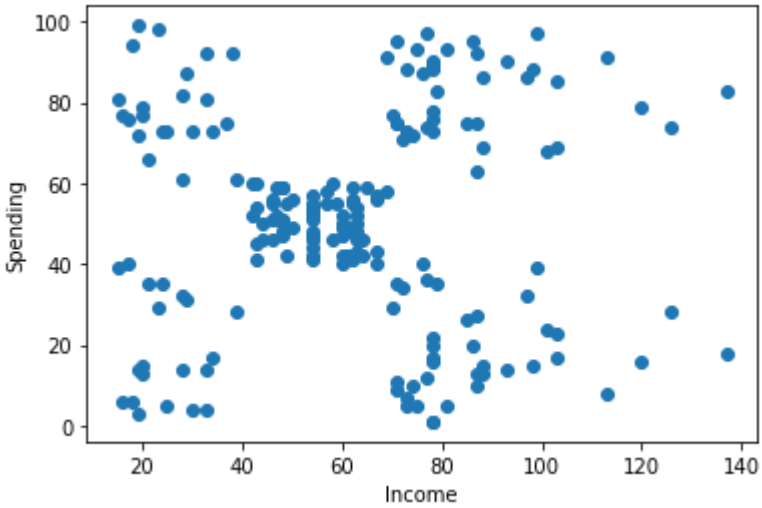
In [44]:

```
plt.figure()
plt.scatter(data.Spending,data.gender_code)
plt.ylabel("gender")
plt.xlabel("Spending")
plt.show()
```



In [45]:

```
plt.figure()
plt.scatter(data.Income,data.Spending)
plt.ylabel("Spending")
plt.xlabel("Income")
plt.show()
```



In [46]:

```
from sklearn.cluster import KMeans
```

In [47]:

```
X = data.to_numpy()[:,1:3]
```

In [48]:

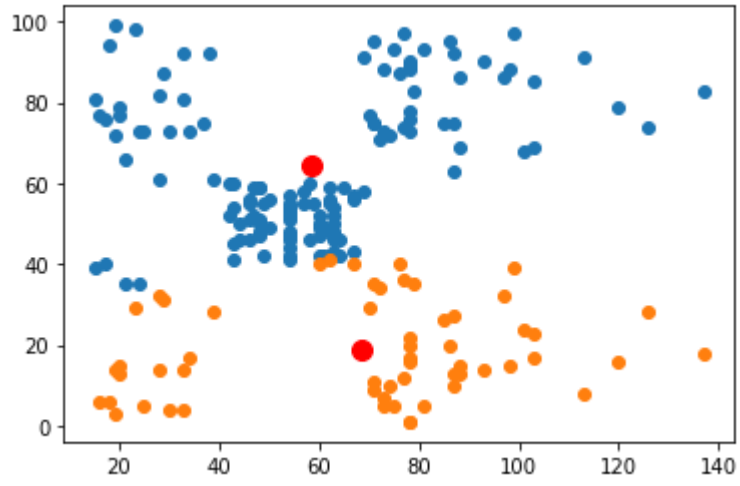
```
clustering = KMeans(n_clusters=2)
pred = clustering.fit_predict(X)
```

In [49]:

```
def build_dict(pred,X):
    tmp = {}
    for i in range(len(pred)):
        c = pred[i]
        if c in tmp:
            tmp[c].append(list(X[i]))
        else:
            tmp[c] = [list(X[i])]
    return tmp

tmp = build_dict(pred,X)
```

In [50]:

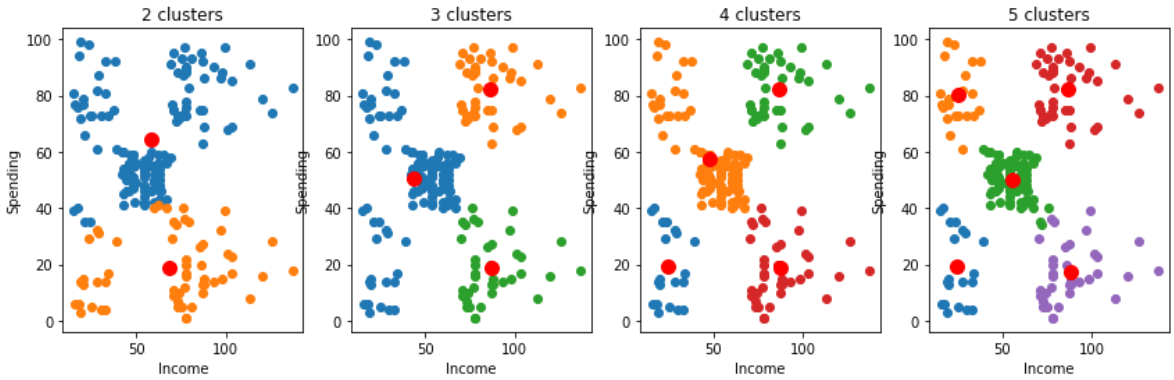
```
plt.figure()
for i in tmp.values():
    i = np.array(i)
    plt.scatter(i[:,0],i[:,1])

plt.scatter(clustering.cluster_centers_[:,0],clustering.cluster_centers_[:,1],s=100
plt.show()
```



How many clusters?

In [51]:

```
plt.figure(figsize=(18,4))
c = 1
for i in range(2,6):
    clustering = KMeans(n_clusters=i)
    pred = clustering.fit_predict(X)

    plt.subplot(1,5,c)
    plt.ylabel("Spending")
    plt.xlabel("Income")
    plt.title(str(c+1)+" clusters")
    tmp = build_dict(pred,X)
    for i in tmp.values():
        i = np.array(i)
        plt.scatter(i[:,0],i[:,1])
    plt.scatter(clustering.cluster_centers_[:,0],clustering.cluster_centers_[:,1],s
    c = c + 1
```

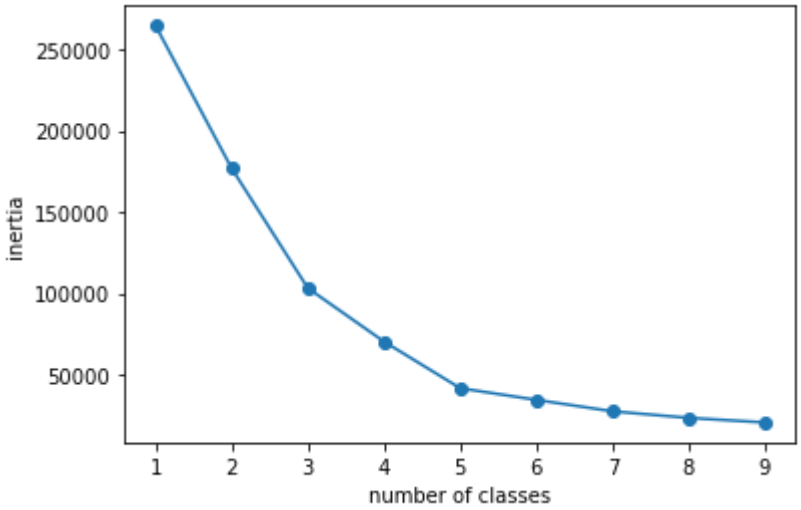


Elbow method

inertia_ : float
Sum of squared distances of samples to their closest cluster center, weighted by the sample weights if provided

In [52]:

```
plt.figure()
inertia = []
for i in range(1,10):
    clustering = KMeans(n_clusters=i)
    clustering.fit(X)
    inertia.append(clustering.inertia_)

plt.plot(range(1,10),inertia,"o-")
plt.xlabel("number of classes")
plt.ylabel("inertia")
plt.show()
```

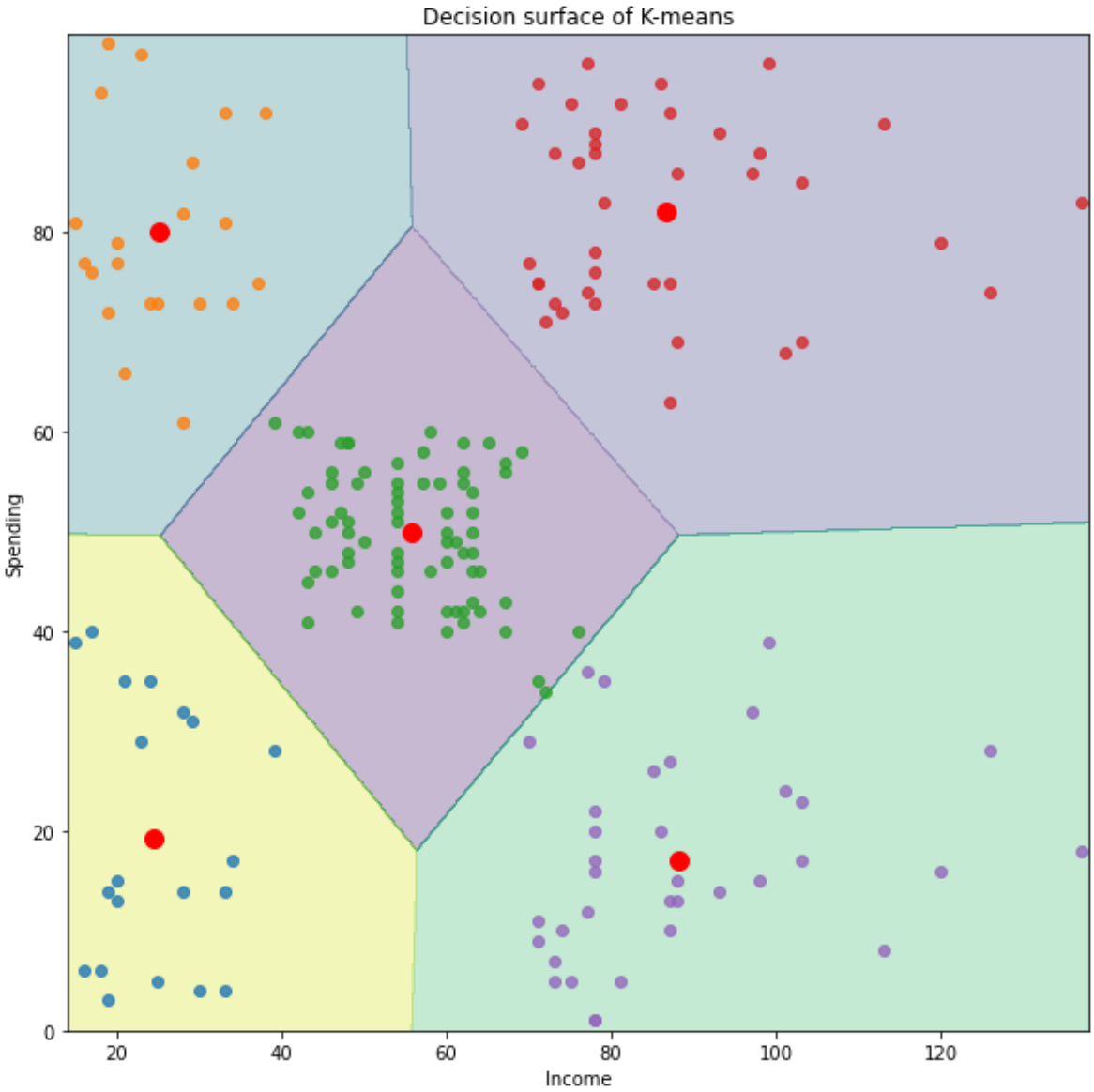


In [53]:

```
# five clusters

clustering = KMeans(n_clusters=5)
pred = clustering.fit_predict(X)
xx,yy = make_meshgrid(X[:,0],X[:,1])
```

In [54]:

```
# Decision surface
fig, ax = plt.subplots(figsize=(10,10))

plot_contours(ax, clustering, xx, yy, alpha=0.3)
c = 0
for i in tmp.values():
    i = np.array(i)
    plt.scatter(i[:,0],i[:,1],alpha=0.8)
    c = c + 1
plt.scatter(clustering.cluster_centers_[:,0],clustering.cluster_centers_[:,1],s=100
plt.ylabel("Spending")
plt.xlabel("Income")
ax.set_title('Decision surface of K-means')
plt.show()
```



Conclusion unsupervised learning

We presented K-means, then we show an easy technique (elbow method) to chose the number of clusters

Dimensionality reduction

two tipes of wines, several properties...

In [55]:

```
df_wine = pd.read_csv("data/wine.dat")
df_wine.head()
```

Out[55]:

	type	a	b	c	d	e	f	g	h	i	l	m	n	o
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

we have 13 features for each wine, how can we visualize it?

we can use PCA:
PCA is a linear transformation mapping data to a system of uncorrelated coordinates.

In [56]:

```
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
```

In [57]:

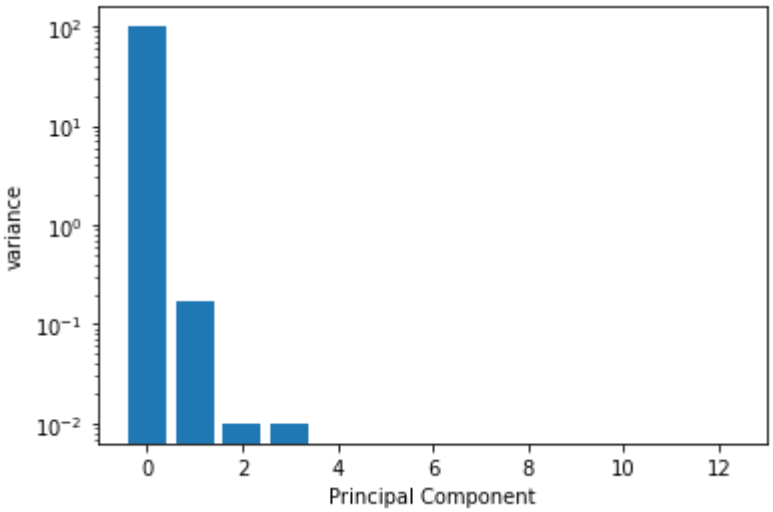
```
from sklearn.decomposition import PCA
```

In [58]:

```
# intialize pca
pca = PCA()

# fit and transform data
X_pca = pca.fit_transform(X)
```

In [59]:

```
plt.figure()
percent_variance = np.round(pca.explained_variance_ratio_* 100, decimals =2)
plt.bar(np.arange(len(percent_variance)), height=percent_variance)
plt.ylabel("variance")
plt.xlabel("Principal Component")
plt.yscale("log")
plt.show()
```



We now ca use PCA to reduce the dimensionality of our input data

In [60]:

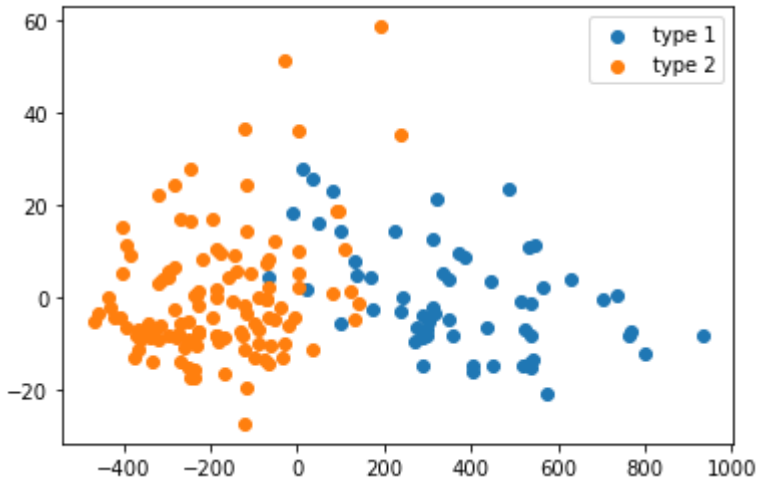
```
# intialize pca
pca = PCA(n_components=2)

# fit and transform data
X_pca = pca.fit_transform(X)
```

In [61]:

```
plt.figure()
tmp = build_dict(y,X_pca)
c = 1
for i in tmp.values():
    i = np.array(i)
    plt.scatter(i[:,0],i[:,1],label="type "+str(c))
    c = c + 1
plt.legend()
```

Out[61]:

```
<matplotlib.legend.Legend at 0x7fee2654ac10>
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

How we can load Tweets?

In [62]:

```
X = pd.read_csv("exercise/tweets/tweets-train-data.csv",names=["tweet","time","retw
y = pd.read_csv("exercise/tweets/tweets-train-targets.csv",names=["y"])
```

In [63]:

```
X = list(X.tweet)
y = list(y.y)
```

In [64]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [65]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer


# **CountVectorizer** Convert a collection of text documents to a matrix of token c
# This implementation produces a sparse representation of the counts using scipy.sp


# Transform a count matrix to a normalized tf or tf-idf representation.
# Tf means term-frequency while tf-idf means term-frequency times inverse document-
# This is a common term weighting scheme in information retrieval, that has also fo
# use in document classification.
```

In [66]:

```
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(X_train)

tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
```

In [ ]:

```

```

In [67]:

```
clf = SVC()
# Training
clf = clf.fit(X_train_tfidf, y_train)
```

In [68]:

```
# use the trained object to convert the test set into bag of words
X_test_counts = count_vect.transform(X_test)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)

# Prediction
y_pred = clf.predict(X_test_tfidf)

print(metrics.accuracy_score(y_test, y_pred))
```

```
0.8790072388831437
```

If we are okey with this results we can load the real test set

In [69]:

```
X = pd.read_csv("exercise/tweets/tweets-train-data.csv",names=["tweet","time","retw
y = pd.read_csv("exercise/tweets/tweets-train-targets.csv",names=["y"])
X = list(X.tweet)
y = list(y.y)


count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(X)

tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)

clf = SVC()
# Training
clf = clf.fit(X_train_tfidf, y)
```

In [ ]:

```

```

In [ ]:

```

```

In [70]:

```

X_test = pd.read_csv("exercise/tweets/tweets-test-data.csv",names=["tweet","time","
y_test = pd.read_csv("exercise/tweets/tweets-test-targets.csv",names=["y"])

X_test = list(X_test.tweet)
y_test = list(y_test.y)


X_test_counts = count_vect.transform(X_test)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)


# Prediction
y_pred = clf.predict(X_test_tfidf)

print(metrics.accuracy_score(y_test, y_pred))
```

```
0.8957169459962756
```

In [ ]:

```

```