

Initiation au Machine Learning avec Python - La pratique

Dans ce tutoriel en 2 parties nous vous proposons de découvrir les bases de l'apprentissage automatique et de vous y initier avec le langage Python. Cette seconde partie vous permet de passer enfin à la pratique avec le langage Python et la librairie Scikit-Learn !

Tutoriel en 2 parties:

Initiation au machine learning avec Python - La theorie

Initiation au machine learning avec Python - La pratique

Introduction à Scikit Learn

Scikits-Learn est une librairie d'apprentissage automatique couvrant l'ensemble de la discipline:

Les types d'apprentissage : supervisé, non supervisé, par renforcement, par transfert Les algorithmes :

- Linear Regression (régression linéaire),
- Logistic Regression (régression logistique),
- Decision Tree (arbre de décision),
- SVM (machines à vecteur de support),
- Naive Bayes (classification naïve bayésienne),
- KNN (Plus proches voisins),
- Dimensionality Reduction Algorithms,
- Gradient Boost & Adaboost,
- Réseaux de neurones

Pourquoi débuter avec Scikit-Learn !!

C'est une bonne idée de débuter la découverte de l'apprentissage automatique avec cette librairie:

- Elle dispose d'une excellente documentation fournissant de nombreux exemples
- Elle dispose d'une API uniforme entre tous les algorithmes, ce qui fait qu'il est facile de basculer de l'un à l'autre
- Elle est très bien intégrée avec les Librairies Pandas et Seaborn
- Elle dispose d'une grande communauté et de plus de 800 contributeurs référencés sur GitHub !
- C'est un projet open source
- Son code est rapide, certaines parties sont implémentées en Cython

Ensuite, quand vous en aurez fait le tour, vous pourrez basculer vers d'autres librairies plus optimisées ou spécialisées sur une sujet précis (type d'apprentissage, algorithme, matériel, ...). Notre chapitre de présentation

du Machine Learning propose déjà un bon panel des autres solutions à votre disposition...

Les concepts de la librairie

Les données

Vos données sont représentées par des tableaux à 2 dimensions. Typiquement, des tableaux Numpy ou Pandas ou Python

- Les lignes représentent les enregistrements
- Les colonnes les attributs (hauteur, longueur, couleur, autre information)
- Une donnée est un vecteur de paramètres, généralement des réels, mais les entiers, booléens et valeurs discrètes sont autorisées dans certains cas
- Les labels peuvent être de différents types, généralement des entiers ou chaînes
- Les labels sont contenus dans un tableau à une dimension, sauf rares cas où ils peuvent être dans le vecteur de paramètres

Prédiction

L'algorithme de prédiction est représenté par une classe.

- Vous devez commencer par choisir l'algorithme à utiliser , que nous appellerons prédicteur/classifieur/estimator Les algorithmes sont des classes Python. Les données sont toujours des tableaux Numpy/Scipy/Pandas/Python
- Vous précisez ses éventuels paramètres, appelés hyperparamètres en instanciant la classe
- Vous l'alimentez avec la fonction fit dans le cas d'un apprentissage supervisé
- Vous lancez la prédiction sur un ensemble de valeurs via la fonction predict parfois appelée transform dans le cas de l'apprentissage non supervisé

Découverte par la pratique - Classer une fleur selon des critères observables

Nous allons utiliser pour ce tutoriel la base de données d'Iris de la librairie scikit-learn

Mais quel est donc cet Iris ?

Cet exemple est très souvent repris sur Internet. Nous ne dérogerons pas à la règle. Nous réaliserons cependant un cas d'utilisation plus complet que la plupart des exemples que vous pourrez trouver.

Cette base contient des Iris qu'un botaniste, Ronald Fisher, a classés en 1936 à l'aide d'une clef d'identification des plantes (type de pétales, sépale, type des feuilles, forme des feuilles, ...).

Puis, pour chaque fleur classée il a mesuré les longueurs et largeurs des sépales et pétales.

L'idée qui nous vient alors, consiste à demander à l'ordinateur de déterminer automatiquement l'espèce d'une nouvelle plante en fonction de la mesure des dimensions de ses sépales et pétales que nous aurions réalisée sur le terrain. Pour cela nous lui demanderons de construire sa décision à partir de la connaissance extraite des mesures réalisées par M. Fisher.

Autrement dit, nous allons donner à l'ordinateur un jeu de données déjà classées et lui demander de classer de nouvelles données à partir de celui-ci.

C'est un cas d'apprentissage supervisé (mais nous le transformerons aussi en non supervisé).

Une fois alimentés avec les observations connues, nos prédicteurs vont chercher à identifier des groupes parmi les plantes déjà connues et détermineront quel est le groupe duquel se rapproche le plus notre observation.

Botanistes en herbe, à vos claviers !

Note pour les naturalistes amateurs:

Si vous aimez les clefs de détermination, une des plus connues est celle de M. Gaston Bonnier. Ses livres sont aujourd'hui dans le domaine public.

Chargement de la base

In [1]:

```
1 from sklearn import datasets  
2 iris = datasets.load_iris()
```

La variable iris est d'un type inhabituel, découvrons-le...

Découverte du contenu de la base

In [2]:

```
1 print(type(iris)) # Ce n'est pas un DataFrame, mais une sorte de dictionnaire  
<class 'sklearn.utils.Bunch'>
```

Rien de bien commun pour un Pythoniste...

Et que peut-on faire avec ?

In [3]:

```
1 print(dir(iris))  
['DESCR', 'data', 'feature_names', 'filename', 'frame', 'target', 'target_names']
```

In [4]:

```
1 iris.feature_names # Les noms des paramètres de nos données/enregistrements
```

Out[4]:

```
['sepal length (cm)',  
'sepal width (cm)',  
'petal length (cm)',  
'petal width (cm)']
```

L'attribut feature_names contient le nom des différents paramètres de nos données, il s'agit des longueurs et largeurs de pétales et sépales.

Un aperçu des 5 premiers enregistrements:

In [5]:

```
1 iris.data[:5]
```

Out[5]:

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

La liste des espèces connues, nos labels de classification, est contenue dans l'attribut target_names

In [6]:

```
1 iris.target_names
```

Out[6]:

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

Création d'une variable target pour un accès plus facile à cet attribut:

In [7]:

```
1 target = iris.target # Les labels associés à chaque enregistrement
2 target          # target[0] est le label de iris['data'][0]
3 for i in [0,1,2]:
4     print("classe : %s, nb exemplaires: %s" % (i, len(target[ target == i])) )
5 #Les targets sont un tableau indiquant le numéro de l'espèce de chaque enregistrement:
```

```
classe : 0, nb exemplaires: 50
classe : 1, nb exemplaires: 50
classe : 2, nb exemplaires: 50
```

In [8]:

```
1 for i in [0,1,2]:
2     print("classe : %s, nb exemplaires: %s" % (i, len(target[ target == i])) )
```

```
classe : 0, nb exemplaires: 50
classe : 1, nb exemplaires: 50
classe : 2, nb exemplaires: 50
```

En résumé, l'échantillon de fleurs propose plusieurs informations:

- Les noms des données disponibles : feature_names
- Les mesures réalisées sur l'échantillon de fleurs connues et déjà classées : data
- Il s'agit de nos informations, des paramètres de nos vecteurs pour chaque fleur
- Le nom de chaque espèce : target_names
- Le classement de chaque enregistrement data dans son espèce: target

- Il s'agit de la classe de chaque fleur/vecteur

In [9]:

```
1 print(iris.DESCR)

.. _iris_dataset:

Iris plants dataset
-----

**Data Set Characteristics:**

:Number of Instances: 150 (50 in each of three classes)
:Number of Attributes: 4 numeric, predictive attributes and the class
:Attribute Information:
- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm
- class:
  - Iris-Setosa
  - Iris-Versicolour
  - Iris-Virginica

:Summary Statistics:
===== ===== ===== ===== ===== =====
      Min   Max   Mean    SD  Class Correlation
===== ===== ===== ===== ===== =====
sepal length:  4.3  7.9  5.84  0.83  0.7826
sepal width:  2.0  4.4  3.05  0.43 -0.4194
petal length: 1.0  6.9  3.76  1.76  0.9490 (high!)
petal width:  0.1  2.5  1.20  0.76  0.9565 (high!)
===== ===== ===== ===== ===== =====

:Missing Attribute Values: None
:Class Distribution: 33.3% for each of 3 classes.
:Creator: R.A. Fisher
:Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
:Date: July, 1988
```

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken from Fisher's paper. Note that it's the same as in R, but not as in the UCI Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example.) The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

.. topic:: References

- Fisher, R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).
- Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.

(Q327.D83) John Wiley & Sons. ISBN 0-471-22361-1. See page 218.

- Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System Structure and Classification Rule for Recognition in Partially Exposed Environments". IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-2, No. 1, 67-71.
- Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule". IEEE Transactions on Information Theory, May 1972, 431-433.
- See also: 1988 MLC Proceedings, 54-64. Cheeseman et al's AUTOCLASS II conceptual clustering system finds 3 classes in the data.
- Many, many more ...

Observation des données

Avant de commencer à classer ses données il est toujours bon de visualiser à quoi elles ressemblent et si d'éventuelles relations se dessinent.

In [10]:

```
1 data = iris.data # Pour un accès plus rapide
2 # tableau numpy de 2 dimensions de 150 enregistrements de 4 valeurs
3 type(data), data.ndim, data.shape
```

Out[10]:

(numpy.ndarray, 2, (150, 4))

Nous avons donc 150 observations de 4 valeurs

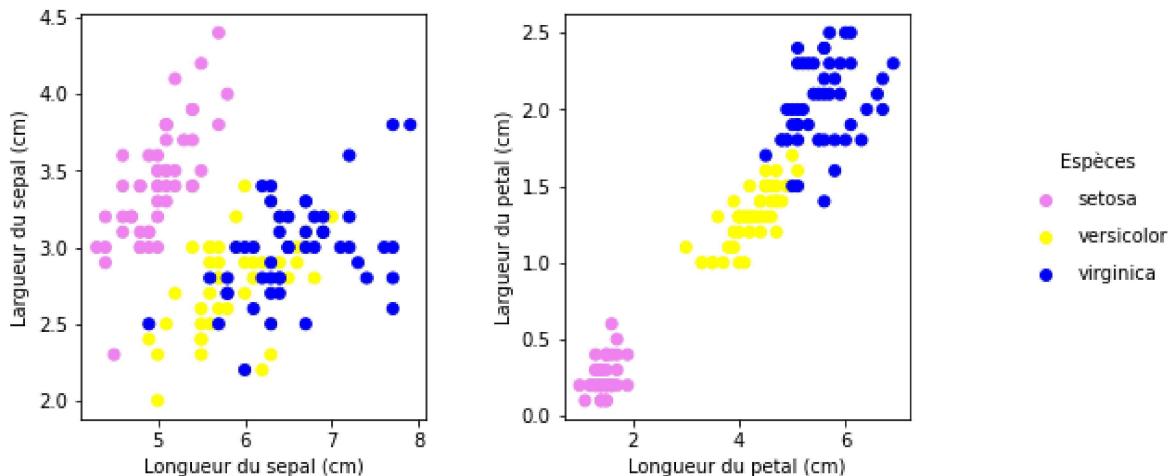
Visualisons les !

In [11]:

```

1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import matplotlib as mpl
4 import numpy as np
5
6 fig = plt.figure(figsize=(8, 4))
7 fig.subplots_adjust(hspace=0.4, wspace=0.4)
8 ax1 = plt.subplot(1,2,1)
9
10 clist = ['violet', 'yellow', 'blue']
11 colors = [clist[c] for c in iris.target]
12
13 ax1.scatter(data[:, 0], data[:, 1], c=colors)
14 plt.xlabel('Longueur du sepal (cm)')
15 plt.ylabel('Largueur du sepal (cm)')
16
17 ax2 = plt.subplot(1,2,2)
18
19 ax2.scatter(data[:, 2], data[:, 3], color=colors)
20
21 plt.xlabel('Longueur du petal (cm)')
22 plt.ylabel('Largueur du petal (cm)')
23
24 # Légende
25 for ind, s in enumerate(iris.target_names):
26     # on dessine de faux points, car la Légende n'affiche que les points ayant un Label
27     plt.scatter([], [], label=s, color=clist[ind])
28
29 plt.legend(scatterpoints=1, frameon=False, labelspacing=1
30             , bbox_to_anchor=(1.8, .5) , loc="center right", title='Espèces')
31 plt.plot();

```



Observer avec Seaborn

C'est assez vite écrit et déjà fort parlant: la séparation des groupes entre les longueurs et largeurs de pétales semble très nette et déterminante !

Nous pourrions aussi le faire entre les longueurs de pétales et largeurs de sépales et inversement même si cela semble moins naturel.

réaliser ce type de graphique:

In [12]:

```
1 import seaborn as sns
2 import pandas as pd
3 sns.set()
4 df = pd.DataFrame(data, columns=iris['feature_names'] )
5 df['target'] = target
6 df['label'] = df.apply(lambda x: iris['target_names'][int(x.target)], axis=1)
7 df.head()
```

Out[12]:

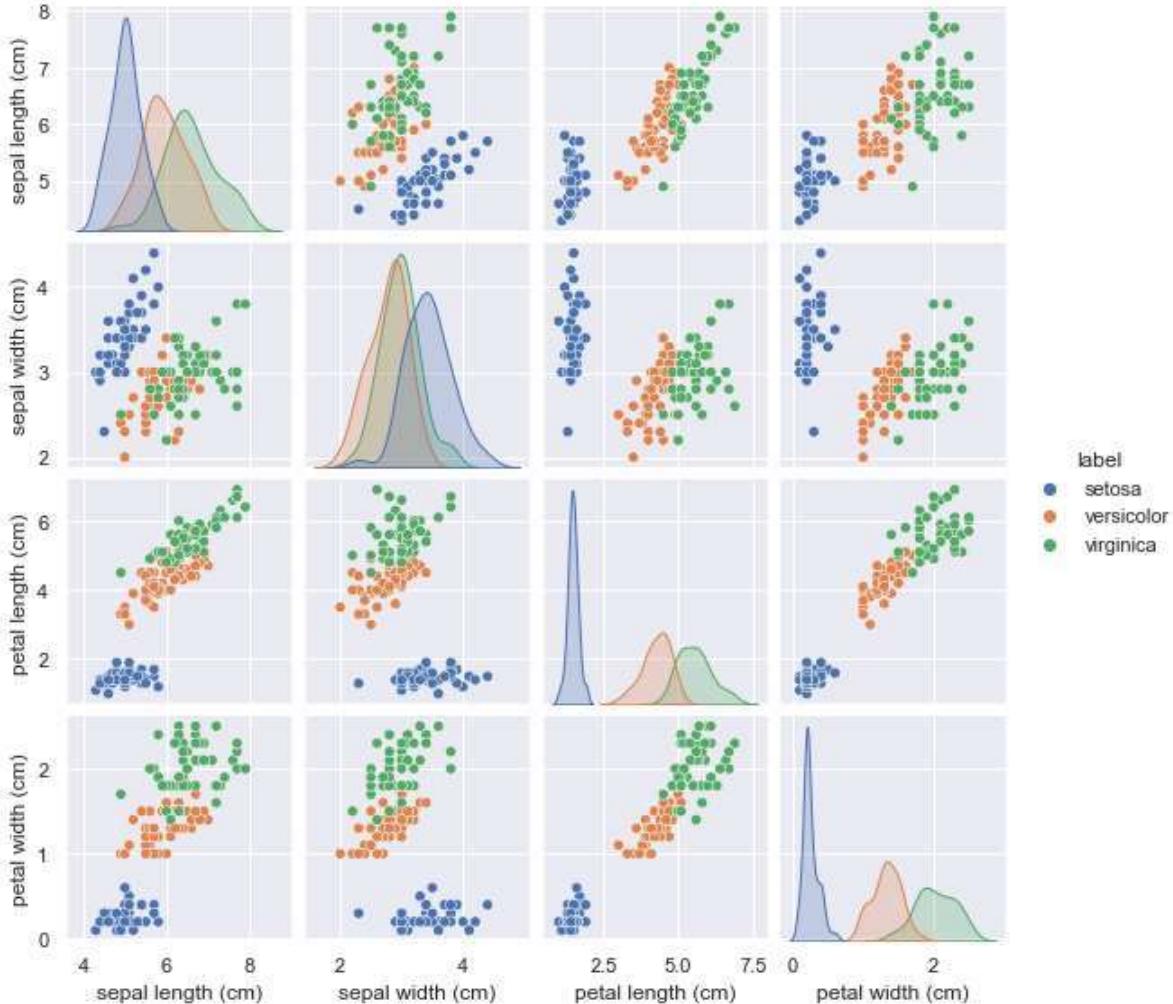
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	label
0	5.1	3.5	1.4	0.2	0	setosa
1	4.9	3.0	1.4	0.2	0	setosa
2	4.7	3.2	1.3	0.2	0	setosa
3	4.6	3.1	1.5	0.2	0	setosa
4	5.0	3.6	1.4	0.2	0	setosa

==> Il ne reste plus qu'à dessiner le graphique avec Seaborn...

In [13]:

```
1 sns.pairplot(df, hue='label', vars=iris['feature_names'], size=2);
```

C:\Users\MSI\anaconda3\lib\site-packages\seaborn\axisgrid.py:1912: UserWarning: The `size` parameter has been renamed to `height`; please update your code.
warnings.warn(msg, UserWarning)



Apprentissage

Nous pourrions ici utiliser plusieurs algorithmes.

Nous proposons de commencer par la classification Naive Bayes qui suppose que chaque classe est construite à partir d'une distribution Gaussienne alignée.

Elle n'impose pas de définir d'hyperparamètres et est très rapide.

In [14]:

```
1 from sklearn.naive_bayes import GaussianNB
```

Création du classifieur

In [15]:

```
1 clf = GaussianNB()
```

Apprentissage

In [16]:

```
1 clf.fit(data, target) # On aurait aussi pu utiliser le dataframe df
```

Out[16]:

GaussianNB()

In [17]:

```
1 GaussianNB(priors=None)
```

Out[17]:

GaussianNB()

In [18]:

```
1 print(dir(clf))
```

```
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '__abc_implementation__', '_check_X', '_check_n_features', '_estimator_type', '_get_param_names', '_get_tags', '_joint_log_likelihood', '_more_tags', '_partial_fit', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_update_mean_variance', '_validate_data', 'class_count_', 'class_prior_', 'classes_', 'epsilon_', 'fit', 'get_params', 'n_features_in_', 'partial_fit', 'predict', 'predict_log_proba', 'predict_proba', 'priors', 'score', 'set_params', 'sigma_', 'theta_', 'var_smoothing']
```

In [19]:

```
1 clf.get_params()
```

Out[19]:

```
{'priors': None, 'var_smoothing': 1e-09}
```

Exécutons la prédition sur les données d'apprentissage elles-mêmes

In [20]:

```
1 result = clf.predict(data)
2 print(result)
```

Qualité de la prédition

Observons la qualité de la prédiction

In [21]:

1 result - target

Out[21]:

Là où la prédiction est juste, la différence de result - target doit être égale à 0. Si la prédiction est parfaite nous aurons des zéros dans tout le tableau. Ce qui est à peine le cas.

Calculons le pourcentage d'erreur:

In [22]:

```

1 errors = sum(result != target) # 6 erreurs sur 150 mesures
2 print("Nb erreurs:", errors)
3 print("Pourcentage de prédiction juste:", (150-errors)*100/150) # 96 % de réussite

```

Nb erreurs: 6

Pourcentage de prédiction juste: 96.0

le label final en fonction des règles de probabilité qu'il a établies. Ces règles ne sont pas rigoureusement identiques à la réalité.

Cela prouve aussi que l'algorithme essaye de trouver un classement intelligent et ne se contente pas de comparer les valeurs d'origines aux valeurs entrantes.

Notre solution pour mesurer la qualité de la prédiction est très rudimentaire, Scikit-Learn propose des solutions plus abouties:

In [23]:

```
1 from sklearn.metrics import accuracy_score
2 accuracy_score(result, target) # 96% de réussite
```

Out[23]:

0.96

Scikit-Learn permet aussi de calculer la matrice de confusion:

In [24]:

```
1 from sklearn.metrics import confusion_matrix
2 conf = confusion_matrix(target, result)
3 conf
```

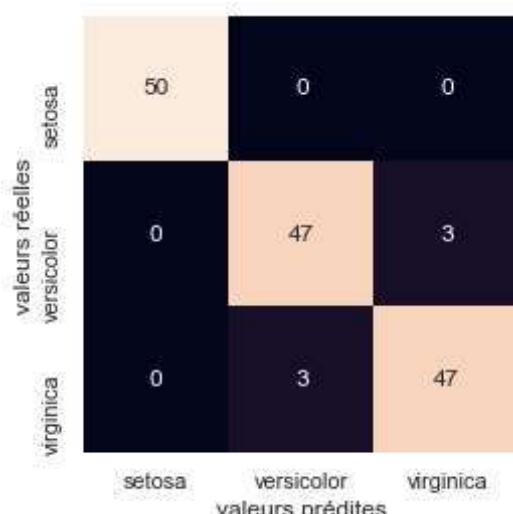
Out[24]:

```
array([[50,  0,  0],
       [ 0, 47,  3],
       [ 0,  3, 47]], dtype=int64)
```

Et Seaborn permet de la représenter avec le Heatmap

In [25]:

```
1 sns.heatmap(conf, square=True, annot=True, cbar=False
2             , xticklabels=list(iris.target_names)
3             , yticklabels=list(iris.target_names))
4 plt.xlabel('valeurs prédictes')
5 plt.ylabel('valeurs réelles');
```

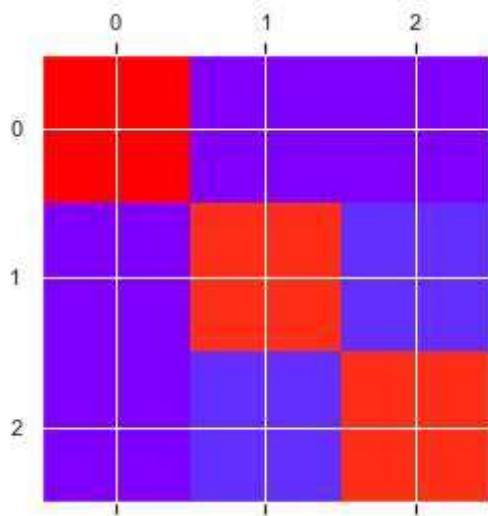


L'on observe ici que:

- L'espèce Setosa a été parfaitement identifiée
- 3 Virginica ont été confondues avec des Versicolor et inversement Ce n'est pas très surprenant les graphiques montrent une nette séparation des Setosa avec les 2 autres groupes qui sont nettement moins détachés.

In [26]:

```
1 # si vous n'avez pas seaborn, la fonction matshow de matplotlib peut vous aider
2 plt.matshow(conf, cmap='rainbow');
```



Séparation du jeu de tests et d'apprentissage

Nous ne disposons que d'un seul jeu de données connues.

Généralement l'on teste l'algorithme sur de nouvelles données, sinon les résultats sont forcément toujours très bons.

Nous pourrions choisir un enregistrement sur 2 comme ci-dessous:

In [27]:

```
1 data_test, target_test = data[::2], target[::2]
2 data_train, target_train = data[1::2], target[1::2]
3 target_test, target_train, len(target_test), len(target_train)
```

Out[27]:

```
(array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2]),
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2]),  
75,  
75)
```

Mais ce n'est pas une très bonne méthode...

Dans cet exemple nous sélectionnons un enregistrement sur 2. Nous nous en sortons bien car toutes les fleurs sont regroupées par ordre de famille dans le tableau, mais si les setosa avaient été stockées 1 sur 2 elles auraient soit toutes été utilisées pour l'apprentissage et aucune n'aurait figuré dans le jeu de tests, ou inversement.

Le module `model_selection` de Scikit-Learn propose des fonctions pour séparer le jeu de données du jeu de tests qui sont attentives à ce type de petits problèmes:

In [28]:

```
1 # from sklearn.cross_validation import train_test_split # Version 0.17.1
2 from sklearn.model_selection import train_test_split # version 0.18.1
3 # split the data with 50% in each set
4 data_test = train_test_split(data, target
5                             , random_state=0
6                             , train_size=0.5)
7 data_train, data_test, target_train, target_test = data_test
```

La fonction `train_test_split` permet de décomposer le jeu de données en 2 groupes: les données pour l'apprentissage et les données pour les tests.

Le paramètre `train_size` indique la taille du jeu d'apprentissage qui sera utilisé: 50% des enregistrements.

In [29]:

```
1 data_test[:5]
```

Out[29]:

```
array([[5.8, 2.8, 5.1, 2.4],
       [6. , 2.2, 4. , 1. ],
       [5.5, 4.2, 1.4, 0.2],
       [7.3, 2.9, 6.3, 1.8],
       [5. , 3.4, 1.5, 0.2]])
```

Il ne reste plus qu'à relancer la classification:

In [30]:

```
1 clf = GaussianNB()
2 clf.fit(data_train, target_train)
3 result = clf.predict(data_test)
```

Puis de calculer de nouveau la qualité de la prédiction:

In [31]:

```
1 # Score
2 accuracy_score(result, target_test)
```

Out[31]:

0.9466666666666667

Cela reste toujours très bon !

Affichons la matrice de confusion:

In [32]:

```

1 # Matrice de confusion
2 conf = confusion_matrix(target_test, result)
3 conf

```

Out[32]:

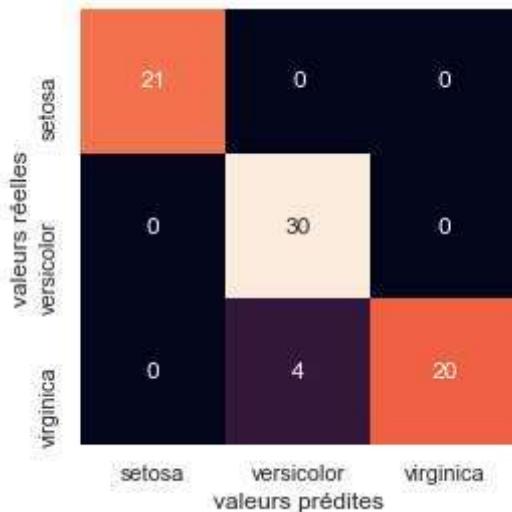
```
array([[21,  0,  0],
       [ 0, 30,  0],
       [ 0,  4, 20]], dtype=int64)
```

In [33]:

```

1 sns.heatmap(conf, square=True, annot=True, cbar=False
2             , xticklabels=list(iris.target_names)
3             , yticklabels=list(iris.target_names))
4 plt.xlabel('valeurs prédites')
5 plt.ylabel('valeurs réelles');

```



Sur les 75 plantes prédites, 4 Virginica ont été confondues avec des Versicolor

Affichons les territoires de la classification

Notre prédicteur s'est "construit une image" de nos données.

Visualisons-là pour toutes les combinaisons de longueurs et largeurs de sépales connues.

L'idée est la suivante:

- Nous construisons un maillage de toutes les combinaisons possibles des longueurs et largeurs des sépales comprises entre leurs valeurs min/max
- Pour chaque couple de point (longueur, largeur) compris entre les min/max observés nous demandons de prédire l'espèce de la fleur
- Nous affichons sur la carte les prédictions réalisées (une couleur pour chaque point)

In [34]:

```

1 # On ne conserve que les Longueurs/Largeurs des sépales
2 data = iris.data[:, :2]
3 target = iris.target
4 data[:5]
```

Out[34]:

```
array([[5.1, 3.5],
       [4.9, 3. ],
       [4.7, 3.2],
       [4.6, 3.1],
       [5. , 3.6]])
```

Aperçu des données:

In [35]:

```

1 # On réapprend
2 clf = GaussianNB()
3 clf.fit(data, target)
4 h = .15
5 # Nous recherchons les valeurs min/max de Longueurs/Largeurs des sépales
6 x_min, x_max = data[:, 0].min() - 1, data[:, 0].max() + 1
7 y_min, y_max = data[:, 1].min() - 1, data[:, 1].max() + 1
8
9 x = np.arange(x_min, x_max, h)
10 y = np.arange(y_min, y_max, h)
```

Le tableau x contient la liste des longueurs des sépales qui seront utilisées pour les tests de classification, comprises entre les min/max observés

In [36]:

```
1 print(x)
```

```
[3.3 3.45 3.6 3.75 3.9 4.05 4.2 4.35 4.5 4.65 4.8 4.95 5.1 5.25
 5.4 5.55 5.7 5.85 6. 6.15 6.3 6.45 6.6 6.75 6.9 7.05 7.2 7.35
 7.5 7.65 7.8 7.95 8.1 8.25 8.4 8.55 8.7 8.85]
```

Affichons les longueurs/largeurs min et max observées:

In [37]:

```
1 x_min, x_max, y_min, y_max
```

Out[37]:

```
(3.3, 8.9, 1.0, 5.4)
```

Les longueurs varient entre 3.2 et 8.9 cm et 1.0 et 5.4 cm pour les largeurs.

Nous allons maintenant créer une matrice contenant toutes les points situés entre les valeurs minimales et maximales des longueurs et largeurs des sépales

La fonction meshgrid permet d'obtenir une grille de coordonnées pour les valeurs des points comprises entre x_min, x_max et y_min, y_max

In [38]:

```
1 xx, yy = np.meshgrid(x,y )
2
3 # http://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html
4 # applatit Les données du tableau
5 data_samples = list(zip(xx.ravel(), yy.ravel()))
```

Explications

Le tableau xx contient les différentes longueurs répétées autant de fois que nous avons de mesures pour les largeurs:

In [39]:

```
1 print(xx) # Vecteur des valeurs de X répété autant de fois que l'on a
2     # de valeurs différentes pour Y
```

```
[[3.3 3.45 3.6 ... 8.55 8.7 8.85]
 [3.3 3.45 3.6 ... 8.55 8.7 8.85]
 [3.3 3.45 3.6 ... 8.55 8.7 8.85]
 ...
 [3.3 3.45 3.6 ... 8.55 8.7 8.85]
 [3.3 3.45 3.6 ... 8.55 8.7 8.85]
 [3.3 3.45 3.6 ... 8.55 8.7 8.85]]
```

Inversement, le tableau yy, contient chaque largeur répétée autant de fois qu'il y a de mesures différentes des longueurs:

In [40]:

```
1 print(yy)
[[1. 1. 1. ... 1. 1. 1. ]
 [1.15 1.15 1.15 ... 1.15 1.15 1.15]
 [1.3 1.3 1.3 ... 1.3 1.3 1.3 ]
 ...
 [5.05 5.05 5.05 ... 5.05 5.05 5.05]
 [5.2 5.2 5.2 ... 5.2 5.2 5.2 ]
 [5.35 5.35 5.35 ... 5.35 5.35 5.35]]
```

La fonction ravel aplatit un tableau à n dimensions en 1 tableau d'une dimension:

In [41]:

```
1 a = [ [10, 20],
2       [ 1, 2] ]
3 np.array(a).ravel()
```

Out[41]:

```
array([10, 20, 1, 2])
```

La fonction zip génère quant-à-elle une liste de n-uplets constituée des éléments du même rang de chaque liste

reçue en paramètre:

In [42]:

```
1 list(zip([10,20,30], [1,2,3]))
```

Out[42]:

```
[(10, 1), (20, 2), (30, 3)]
```

Nous pouvons donc maintenant visualiser le contenu du jeu de données généré:

In [43]:

```
1 data_samples[:10]
```

Out[43]:

```
[(3.3, 1.0),
 (3.449999999999997, 1.0),
 (3.599999999999996, 1.0),
 (3.749999999999996, 1.0),
 (3.899999999999995, 1.0),
 (4.049999999999999, 1.0),
 (4.199999999999999, 1.0),
 (4.35, 1.0),
 (4.499999999999999, 1.0),
 (4.649999999999999, 1.0)]
```

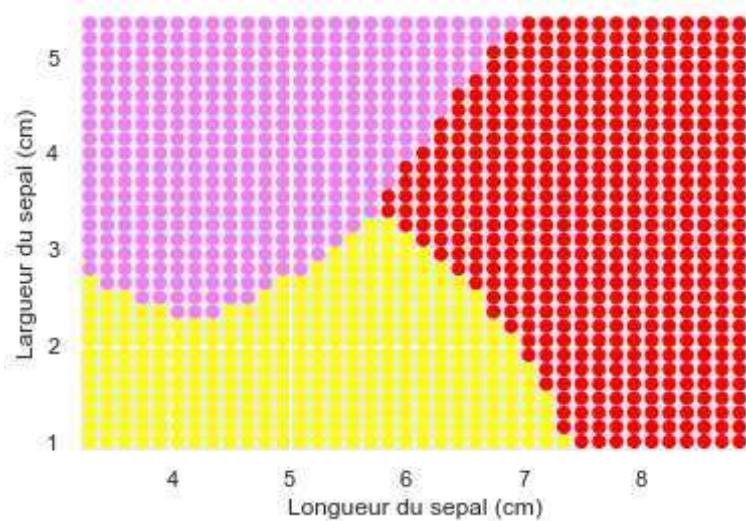
Ces couples de points ne sont autres que des mesures de fleurs imaginaires comprises entre les valeurs min/max connues.

Le but étant de déterminer leur espèce pour voir l'extension des territoires de chacune d'elle, telle que classée par l'ordinateur

Nous pouvons maintenant afficher les espèces telles que l'algorithme les évaluerait si nous les mesurerions dans la nature:

In [44]:

```
1 Z = clf.predict(data_samples)
2 #Z = Z.reshape(xx.shape)
3 plt.figure(1)
4 #plt.pcolormesh(xx, yy, Z) # Affiche les déductions en couleurs pour les couples x,y
5
6 # Plot also the training points
7 #plt.scatter(data[:, 0], data[:, 1], c=target)
8 colors = ['violet', 'yellow', 'red']
9 C = [colors[x] for x in Z]
10
11 plt.scatter(xx.ravel(), yy.ravel(), c=C)
12 plt.xlim(xx.min() - .1, xx.max() + .1)
13 plt.ylim(yy.min() - .1, yy.max() + .1)
14 plt.xlabel('Longueur du sepal (cm)')
15 plt.ylabel('Largeur du sepal (cm)');
```



Cette image, n'est autre que votre clef de détermination: Imprimez-là et partez identifier les fleurs sur le terrain : mesurez les longueurs/largeurs de sépales, recherchez-les sur le graphique, la couleur du point vous donne l'espèce !

Magique non ?

Affichons le limites avec pcolormesh

In [45]:

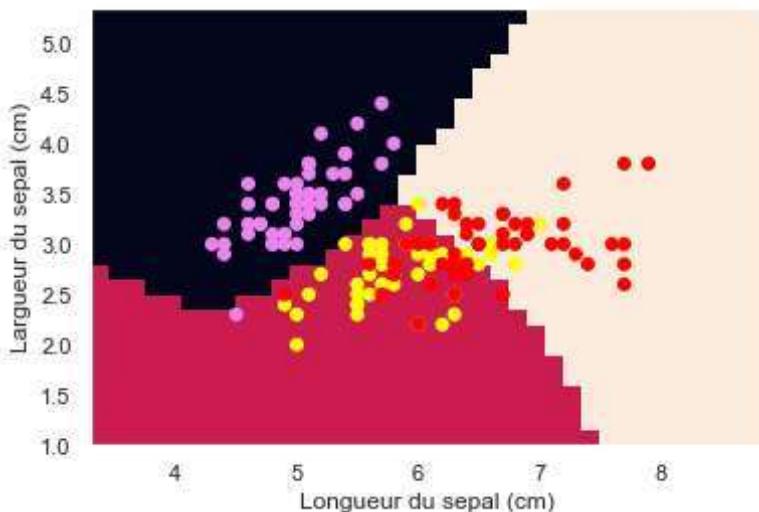
```

1 plt.figure(1)
2 plt.pcolormesh(xx, yy, Z.reshape(xx.shape)) # Affiche les déductions en couleurs pour l
3 # Plot also the training points
4 colors = ['violet', 'yellow', 'red']
5 C = [colors[x] for x in target]
6 plt.scatter(data[:, 0], data[:, 1], c=C)
7 plt.xlim(xx.min(), xx.max())
8 plt.ylim(yy.min(), yy.max())
9 plt.xlabel('Longueur du sepal (cm)')
10 plt.ylabel('Largueur du sepal (cm)');

```

<ipython-input-45-faf39075eec9>:2: MatplotlibDeprecationWarning: shading='fl at' when X and Y have the same dimensions as C is deprecated since 3.3. Eit her specify the corners of the quadrilaterals with X and Y, or pass shading ='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading']. This wi ll become an error two minor releases later.

plt.pcolormesh(xx, yy, Z.reshape(xx.shape)) # Affiche les déductions en co uleurs pour les couples x,y



Complément

Essayons le même traitement en remplaçant GaussianNB par KN.

La classe à utiliser est

In [46]:

```

1 from sklearn import neighbors
2 clf = neighbors.KNeighborsClassifier()

```

Si vous travaillez dans un notebook Jupyter (fortement recommandé), utilisez le décorateur interact pour faire varier l'hyperparamètre N du nombre voisins

In [47]:

```
1 from ipywidgets import interact
2 @interact(n=(0,20))
3 def n_change(n=5):
4     clf = neighbors.KNeighborsClassifier(n_neighbors=n)
5     clf.fit(data, target)
6     Z = clf.predict(data_samples)
7     plt.figure(1)
8     plt.pcolormesh(xx, yy, Z.reshape(xx.shape)) # Affiche les déductions en couleurs pour
9     # Plot also the training points
10    colors = ['violet', 'yellow', 'red']
11    C = [colors[x] for x in target]
12    plt.scatter(data[:, 0], data[:, 1], c=C)
13    plt.xlim(xx.min(), xx.max())
14    plt.ylim(yy.min(), yy.max())
15    plt.xlabel('Longueur du sepal (cm)')
16    plt.ylabel('Largeur du sepal (cm)');
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

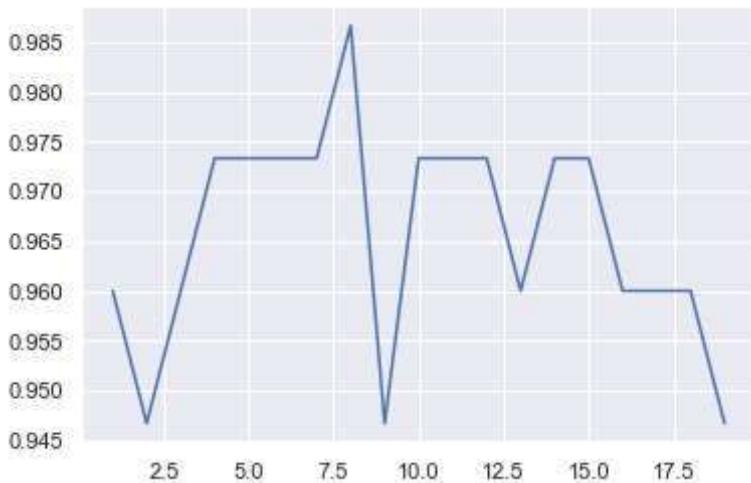
Calcul de la précision de la prédiction en fonction de N

In [48]:

```
1 data_test, target_test = iris.data[::2], iris.target[::2]
2 data_train, target_train = iris.data[1::2], iris.target[1::2]
3 result = []
4 n_values = range(1,20)
5 for n in n_values:
6     clf = neighbors.KNeighborsClassifier(n_neighbors=n)
7     clf.fit(data_train, target_train)
8     Z = clf.predict(data_test)
9     score = accuracy_score(Z, target_test)
10    result.append(score)
11
12 plt.plot(list(n_values), result)
```

Out[48]:

[<matplotlib.lines.Line2D at 0x243f10587c0>]



Le graphique généré montre que la prédiction semble la meilleure pour N=8

Si vous souhaitez découvrir comment optimiser au mieux vos hyperparamètres, la fonction grid search vous simplifie énormément la recherche des hyperparamètres optimaux.

In []:

1

In []:

1

In []:

1

In []:

1

In []:

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```