Le Machine Learning avec Python

Salim Lardjane
Université Bretagne Sud

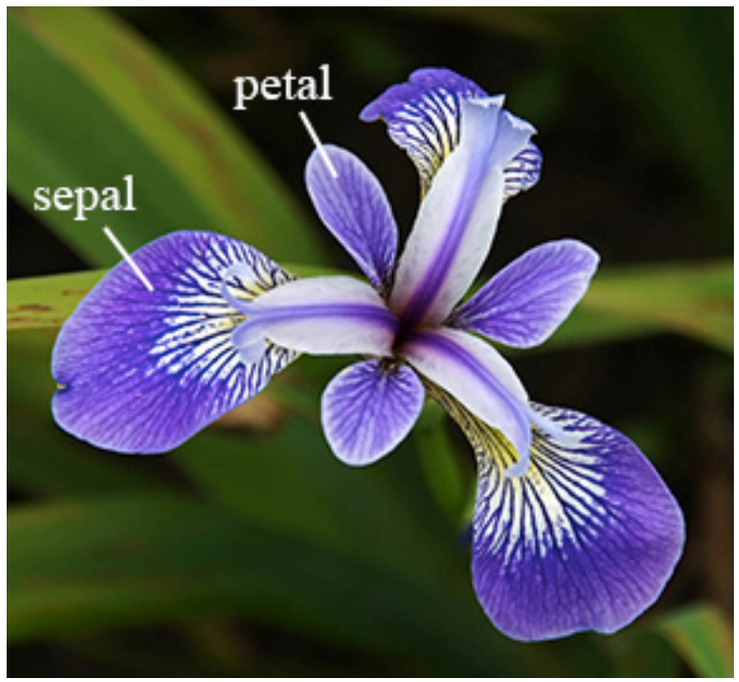
Un premier exemple

Un premier exemple

- On va commencer par une application simple du machine learning et créer notre premier modèle sous Python.
- On suppose qu’une botaniste amatrice souhaite distinguer les espèces de certaines fleurs d’iris qu’elle a récoltées.

Un premier exemple

- Elle a effectué des mesures sur chaque iris récolté : longueur et largeur des pétales, longueur et largeur des sépales, toutes mesurées en centimètres.



Un premier exemple

- Elle dispose des même mesures sur des fleurs d'iris qui avaient été classées par une botaniste experte comme appartenant aux espèces *setosa*, *versicolor* ou *virginica*.

- Pour ces fleurs-là, elle est certaine de l'appartenance de chaque fleur.

- Notre objectif est de construire un modèle de Machine Learning qui puisse *apprendre* à partir des mesures faites sur les iris dont l'espèce est connue, de façon à ce qu'on puisse *prévoir* l'espèce pour une nouvelle fleur d'iris.

Un premier exemple

- Il s'agit d'un problème d'apprentissage *supervisé*.

- C'est un exemple de *reconnaissance de formes* (ang. Pattern Recognition, Classification).

- On souhaite obtenir en sortie, pour un nouveau point de données (une nouvelle iris), l'espèce correspondante. Dans ce contexte, on appelle celle-ci *label*.

Un premier exemple

- Les données qu'on va utiliser sont une jeu de données classique en Machine Learning et en Statistique.

- Dans scikit-learn, il est inclus dans le module datasets.

- On peut le charger à l'aide de la fonction load_iris.

```
from sklearn.datasets import load_iris

iris_dataset = load_iris()
```

Un premier exemple

- L'objet iris_dataset renvoyé par load_iris est un objet de type Bunch, qui est très semblable à un dictionnaire.

- Il contient des clefs et des valeurs associées.

```
print("Keys of iris_dataset :
\n{}".format(iris_dataset.keys()))
```

Un premier exemple

- Résultat obtenu :

```
Keys of iris_dataset :

dict_keys(['data', 'target', 'target_names',
'DESCR', 'feature_names'])
```

Un premier exemple

- La valeur de la clef DESCR est une courte description du jeu de données. On peut en examiner le début de la façon suivante :

```
print(iris_dataset['DESCR'][:193]+"\n...")
```

- On obtient le résultat suivant :

Un premier exemple

```
Iris Plants Database

====================

Notes

-----

Data Set Characteristics:

    :Number of Instances: 150 (50 in each of three classes)

    :Number of Attributes: 4 numeric, predictive att

...
```

Un premier exemple

- La valeur de la clef target_names est un tableau de chaînes de caractères, contenant les espèces de fleurs que l'on souhaite prévoir :

```
print("Target names:
{}".format(iris_dataset['target_names']))
```

- Résultat :

Un premier exemple

```
Target names: ['setosa' 'versicolor'
'virginica']
```

- La valeur de feature_names est une liste de chaînes de caractères, donnant la description de chaque variable :

```
print("Feature names:
{}".format(iris_dataset['feature_names']))
```

- Résultat :

Un premier exemple

```
Feature names: ['sepal length (cm)', 'sepal
width (cm)', 'petal length (cm)', 'petal width
(cm)’]
```

- Les données à proprement parler sont contenues dans les champs target et data.

- data contient les mesures de longueur de sépale, de largeur de sépale, de longueur de pétale et de largeur de pétale, sous la forme d’un tableau NumPy :

```
print("Type of data:
{}".format(type(iris_dataset['data'])))
```

Un premier exemple

- Résultat :

```
Type of data: <class 'numpy.ndarray'>
```

- Les lignes de data correspondent aux fleurs et les colonnes représentent les quatre mesures faites sur chaque fleur :

```
print("Shape of data:
{}".format(iris_dataset['data'].shape))
```

- Résultat :

Un premier exemple

```
Shape of data: (150, 4)
```

- On voit que le tableau contient des mesures pour 150 fleurs.

- On va afficher les 5 premières lignes du tableau de données :

```
print("First five lines of data:
\n{}".format(iris_dataset['data'][:5]))
```

Un premier exemple

- Résultat :

```
First five lines of data:

[[ 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]]
```

Un premier exemple

- Le tableau target contient l'espèce de chacune des fleurs sur lesquelles on a effectué des mesures; c'est un tableau NumPy.

```
print("Type of target:
{}".format(type(iris_dataset['target'])))
```

- Résultat :

```
Type of target: <class 'numpy.ndarray'>
```

Un premier exemple

- target est un tableau uni-dimensionnel avec une donnée par fleur :

```
print("Shape of target:
{}".format(iris_dataset['target'].shape))
```

- Résultat :

```
Shape of target: (150,)
```

Un premier exemple

- Les espèces sont codées par les entiers 0, 1, 2 :

```
print("Target:
\n{}".format(iris_dataset['target']))
```

- Résultat :

Un premier exemple

```
Target:

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 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 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Un premier exemple

- La signification des codes est données par le tableau `iris['target_names']`.

- *Setosa* est codé par 0, *versicolor* par 1 et *virginica* par 2.

Un premier exemple

- On souhaite construire un modèle de Machine Learning à partir de ces données, qui puisse prédire l'espèce à partir de nouvelles mesures.

- Mais avant d'appliquer notre modèle à de nouvelles mesures, on doit s'assurer qu'il *fonctionne bien*, c'est-à-dire *qu'on peut faire confiance à ses prédictions.*

- *On ne peut pas utiliser les données utilisées pour construire le modèle pour l'évaluer.*

- En effet, le modèle peut simplement *garder en mémoire* les données sur lesquelles il a été construit; mais cela ne dit rien sur sa *capacité de généralisation*.

Un premier exemple

- *Afin d'évaluer les performance du modèle, on lui soumet de nouvelles données pour lesquelles on dispose de labels.*

- C'est généralement fait en divisant les données labellisées dont on dispose (150 mesures ici) en deux parties : la première partie, utilisée pour construire le modèle, est appelée ensemble d'apprentissage. La deuxième partie, composée des données restantes, est utilisée pour évaluer les performances du modèle; elle est appelée ensemble de test.

Un premier exemple

- scikit-learn contient une fonction qui réordonne aléatoirement les données et effectue la séparation en deux parties : il s'agit de la fonction train_test_split.

- Cette fonction extrait 75% des lignes du tableau de données comme ensemble d'apprentissage et les 25% restantes comme ensemble de test.

- Les pourcentages sont en fait arbitraires mais la pratique recommande la règle des 75%-25%.

Un premier exemple

- Sous scikit-learn, les données sont généralement notées par un X majuscule et les labels par un y minuscule.

- Appliquons train_test_split à nos données :

```
from sklearn.model_selection import
train_test_split

X_train, X_test, y_train, y_test =
train_test_split(iris_dataset['data'],
iris_dataset[‘target'], random_state=0)
```

Un premier exemple

- Avant d'effectuer la division, la fonction train_test_split réordonne les données à l'aide d'un générateur de nombres pseudo-aléatoires. Ceci permet d'éviter que toutes les données d'une classe soient exclues de l'ensemble d'apprentissage par exemple.

- Afin d'obtenir le même résultat en relançant la fonction plusieurs fois, on fournit un germe au générateur de nombres pseudo-aléatoires. C'est fait à l'aide de l'argument random_state de la fonction.

Un premier exemple

- Le résultat de la fonction train_test_split est X_train, X_test, y_train et y_test.

- Ce sont tous des tableaux NumPy.

- X_train contient 75% des lignes du tableau de données et X_test 25%.

Un premier exemple

```
print("X_train shape: {}".format(X_train.shape))

print("X_test shape: {}".format(X_test.shape))
```

- Résultat :

```
X_train shape: (112, 4)

X_test shape: (38, 4)
```

Un premier exemple

- Avant de construire notre modèle de machine learning, commençons par visualiser nos données.

- A cet effet, on peut utiliser les nuages de points (ang. *scatter plots*) des variables prises deux à deux.

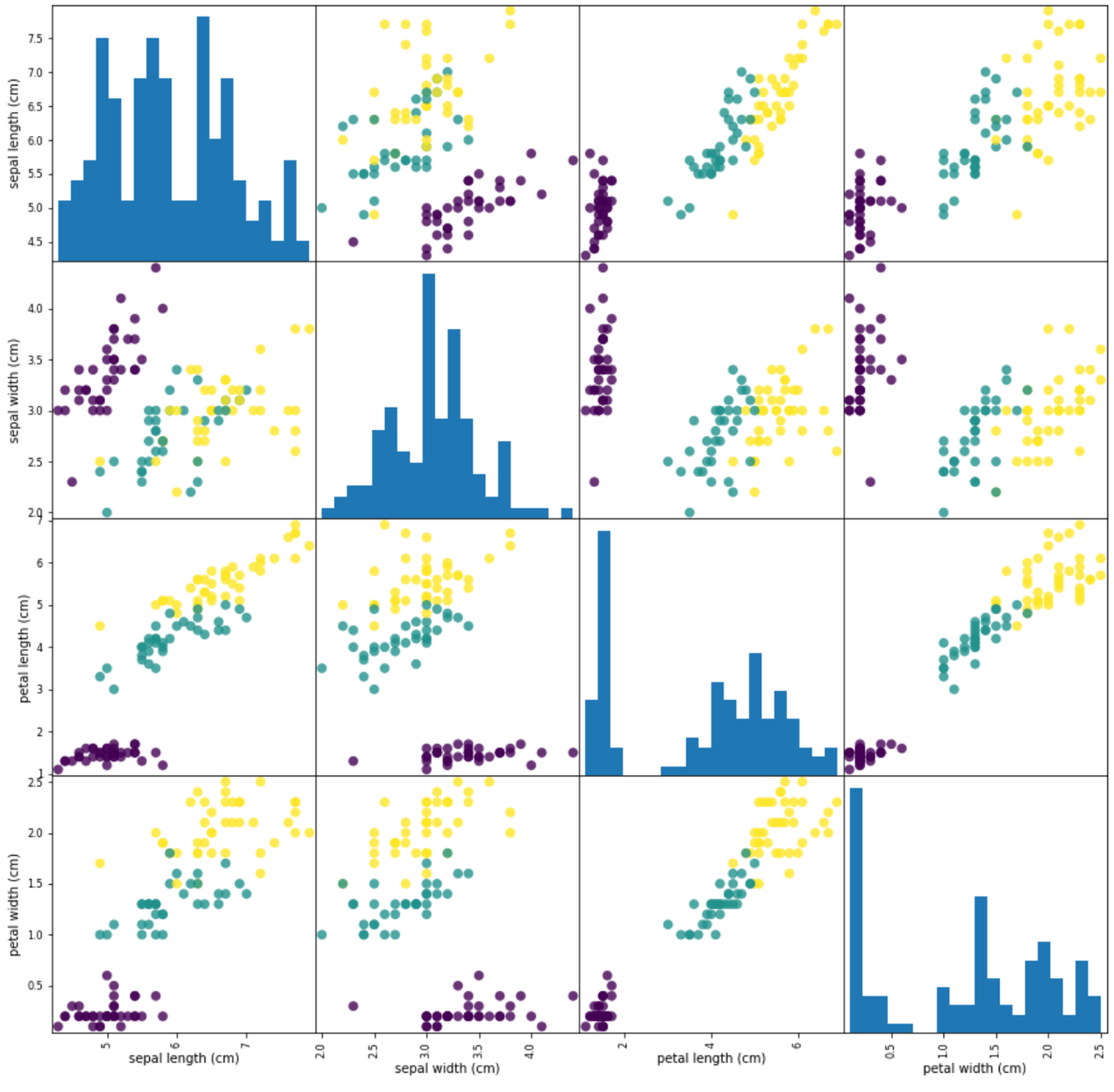
- On commence par convertir les données en un DataFrame pandas.

- On utilise ensuite la fonction scatter_matrix de pandas pour obtenir le graphique.

Un premier exemple

```
import pandas as pd

import pandas.plotting as pdpl

iris_dataframe = pd.DataFrame(X_train, columns =
iris_dataset.feature_names)

grr = pdpl.scatter_matrix(iris_dataframe,
c=y_train, figsize=(15,15), marker='o',
hist_kwds={'bins':20}, s=60, alpha = 0.8)
```

Un premier exemple



Un premier exemple

- Le graphique obtenu permet de voir que les trois classes peuvent être relativement bien séparées en utilisant les mesures de sépales et de pétales.

- Ceci signifie qu'un algorithme de Machine Learning devrait être à même d'*apprendre* à les séparer.

- Le premier algorithme qu'on va considérer est *l'algorithme du plus proche voisin.*

Un premier exemple

- Pour faire une prévision sur de nouvelles données, l'algorithme identifie le point de l'ensemble d'apprentissage le plus proche de ces données puis il affecte le label de ce point au nouvelles données.

- Tous les algorithmes de machine learning présents dans scikit-learn sont implémentés dans des *classes* spécifiques, qui sont appelées classes Estimator.

- L'algorithme des k-plus proches voisins est implémenté dans la classe KNeighborsClassifier du module Neighbors.

Un premier exemple

- Le paramètre le plus important de KNeighborsClassifier est le nombre de voisins, qu'on va fixer à 1 :

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

- L'*objet* knn encapsule l'algorithme utilisé pour construire le modèle à partir des données d'apprentissage, ainsi que l'algorithme utilisé pour faire des prévisions sur de nouveaux points.

- Il contiendra également l'information extraite par l'algorithme des données d'apprentissage. Dans le cas de KNeighborsClassifier, il s'agira simplement d'une copie de l'ensemble d'apprentissage.

Un premier exemple

- Pour construire notre modèle à partir de l'ensemble d'apprentissage, on appelle la *méthode* fit de l'objet knn, qui prend comme arguments les tableaux NumPy X_train et y_train :

```
knn.fit(X_train, y_train)
```

- Résultat :

```
KNeighborsClassifier(algorithm='auto',
leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=1, n_neighbors=1,
p=2, weights='uniform')
```

Un premier exemple

- La méthode fit renvoie l'objet knn lui-même (et le modifie par endroits); on obtient ainsi une représentation de notre modèle sous la forme d'une chaîne de caractères.

- Cette représentation nous fournit les paramètres utilisés dans la création du modèle. Ici, tous ont leurs valeurs par défaut, sauf n_neighbors qu'on a fixé à 1.

- La plupart des modèles disponibles sous scikit-learn ont de nombreux paramètres mais la plupart d'entre eux servent soit à optimiser la rapidité d'exécution, soit concernent des cas d'utilisation particuliers.

- Ainsi, afficher un modèle scikit-learn peut donner des chaînes de caractères très longues, mais il n'est nécessaire de connaître qu'un petit nombre de paramètres, qu'on verra dans la suite du cours.

Un premier exemple

- On peut à présent effectuer des prévisions sur de nouvelles données.

- Supposons par exemple qu'on ait trouvé dans la nature un iris de longueur de sépale 5cm, de largeur de sépale 2.9cm, de longueur de pétale 1cm et de largeur de pétale 0.2cm. A quelle espèce appartient-il ?

- On commence par ranger les données dans un tableau NumPy :

```
import numpy as np

X_new = np.array([[5, 2.9, 1, 0.2]])

print("X_new shape: {}".format(X_new.shape))
```

Un premier exemple

- Résultat :

```
X_new shape: (1, 4)
```

- Notons qu'on a rangé les données dans *une ligne* d'un tableau bi-dimensionnel, car scikit-learn attend les données sous cette dernière forme.

- Afin d'effectuer une prévision, on appelle la *méthode* predict de l'objet knn :

Un premier exemple

```
prediction = knn.predict(X_new)

print("Prediction: {}".format(prediction))

print("Predicted target name:
{}".format(iris_dataset['target_names'][prediction]))
```

- Résultat :

```
Prediction: [0]

Predicted target name: ['setosa']
```

Un premier exemple

- Notre modèle prédit que le nouvel iris appartient à la classe 0, c'est-à-dire que son espèce est *setosa*.

- *Mais comment savoir si on peut avoir confiance en notre modèle et donc en cette conclusion ?*

- C'est ici qu'intervient l'ensemble de test.

Un premier exemple

- Les données de test n'ont pas été utilisées pour construire le modèle et on dispose du label correct pour chacune de ces données.

- Par conséquent, on peut faire une prévision pour chaque iris de l'ensemble test et la comparer au label correct.

- On peut évaluer la qualité du modèle en calculant son exactitude (ang. accuracy), c'est-à-dire la proportion de fleurs de l'ensemble test pour lesquelles la prévision est correcte.

Un premier exemple

```
y_pred = knn.predict(X_test)

print("Test set predictions:\n
{}".format(y_pred))

print("Test set score: {:.
2f}".format(np.mean(y_pred==y_test)))
```

- Résultat :

```
[2 1 0 ..., 1 0 2]

Test set score: 0.97
```

Un premier exemple

- On peut également utiliser la méthode score de l'objet knn, qui calcule directement l'exactitude :

```
print("Test set score: {:.
2f}".format(knn.score(X_test,y_test)))
```

- Résultat :

```
Test set score: 0.97
```

Un premier exemple

- L’exactitude de notre modèle est d’environ 0.97, ce qui signifie qu’on a pris la bonne décision pour 97% des iris de l’ensemble test.

- Ainsi, on peut s’attendre à ce que notre modèle fournisse une prévision correcte dans 97% des cas pour de nouveaux iris.

- Pour notre botaniste cela justifie l’utilisation du modèle pour faire des prévisions.

L’apprentissage supervisé

Apprentissage supervisé

- L'apprentissage supervisé est le type de machine learning le plus utilisé en pratique.

- On est dans le cadre de l'apprentissage supervisé lorsqu'on veut prédire une réponse (output) associée à un certain stimulus (input) et qu'on dispose d'exemples de couples stimulus/réponse (input/output) pour ce faire.

- On construit un modèle de machine learning à partir de ces exemples, qui constituent notre ensemble d'apprentissage.

- Notre but est de faire les prévisions les plus exactes possibles pour de nouvelles données.

Apprentissage supervisé

- L’apprentissage supervisé requiert souvent une intervention humaine lors de la construction de l’ensemble d’apprentissage, mais automatise et accélère grandement la tâche de prévision par la suite.

Apprentissage supervisé

- Il existe deux types principaux de problèmes d'apprentissage supervisé : la classification ou reconnaissance de formes (ang. classification, pattern recognition) - également appelée analyse discriminante - et la régression (ang. regression).

- En reconnaissance de formes, l'objectif est de prévoir un label (identifiant) de classe en le choisissant parmi un nombre prédéfini de possibilités. L'exemple des iris en est une illustration.

Apprentissage supervisé

- On distingue parfois classification binaire et classification multi-classes.

- La reconnaissance de spams est un exemple de classification binaire. Elle répond à la question « *ce nouveau mail est-il un spam ?* ».

- En classification binaire, on désigne souvent une classe comme positive et l'autre comme négative, selon l'objet de l'étude.

Apprentissage supervisé

- L’exemple des iris est un exemple de classification multi-classes.

- Un autre exemple est la reconnaissance de la langue d’un site web à partir du texte présent sur le site. Ici, les classes sont données par une liste de langues prédéfinies.

Apprentissage supervisé

- En régression, le but est de prévoir la valeur prise par une variable continue (un nombre réel en mathématiques, un flottant en informatique).

- En régression, on prévoit une quantité, alors qu'en classification on prévoit une qualité.

- Un exemple de régression est la prévision du revenu annuel de quelqu'un à partir de son niveau d'éducation, de son âge, et de son lieu de résidence.

- Un autre exemple de régression est la prévision du rendement d'une ferme de blé à partir des rendements des années précédentes, de la météo et du nombre d'employés de la ferme.

Généralisation, sur-apprentissage, sous-apprentissage

- En apprentissage supervisé, on construit un modèle sur les données d'apprentissage puis on l'utilise pour faire des prévisions sur de nouvelles données qui ont les mêmes caractéristiques globales que les données d'apprentissage.

- Si le modèle est apte à faire des prévisions exactes sur de nouvelles données, on dit qu'il a la capacité de généraliser.

- *On souhaite construire un modèle qui généralise de la façon la plus exacte possible.*

Généralisation, sur-apprentissage, sous-apprentissage

- Usuellement, on construit le modèle de façon à ce qu'il fasse des prévisions d'exactitude élevée sur l'ensemble d'apprentissage.

- Si l'ensemble test et l'ensemble d'apprentissage ont les mêmes caractéristiques en termes de distribution, on s'attend à ce que le modèle soit également d'exactitude élevée sur l'ensemble test.

- Toutefois, il y a des cas où cela ne fonctionne pas, notamment lorsque le modèle construit est très complexe.

- Dans ce cas, on peut être aussi exact que l'on veut sur l'ensemble d'apprentissage sans pour autant être d'exactitude élevée sur l'ensemble test.

Généralisation, sur-apprentissage, sous-apprentissage

- Intuitivement, on s'attend à ce que les modèles les plus simples généralisent le mieux sur l'ensemble test, même s'ils s'ajustent moins bien à l'ensemble d'apprentissage.

- Lorsque le modèle construit est trop complexe par rapport à l'information contenue dans l'ensemble d'apprentissage, on dit qu'il y a sur-apprentissage.

- Dans ce cas, le modèle construit colle trop aux caractéristiques uniques de l'ensemble d'apprentissage et généralise mal.

Généralisation, sur-apprentissage, sous-apprentissage

- D'autre part, si le modèle est trop simple, on peut échouer à capturer tous les aspects des données et leur variabilité. On dit dans ce cas qu'il y a sous-apprentissage.

- Il y a un arbitrage à faire entre qualité de l'ajustement sur l'ensemble d'apprentissage et qualité de la prévision sur l'ensemble test.

Complexité et taille des données

- La complexité du modèle doit être reliée à la variation des données en entrée (inputs).

- Plus celles-ci sont variées, plus la complexité du modèle construit peut être élevée tout en évitant le sur-apprentissage.

- Usuellement, plus on a de données, plus celles-ci sont variées.

- Ainsi, collecter plus de données permet usuellement d'ajuster des modèles plus complexes.

- Toutefois, dupliquer les mêmes données ou recueillir des données très similaires ne le permettent pas.

Les algorithmes d’apprentissage supervisé

Algorithmes

- On va présenter dans la suite divers algorithmes d’apprentissage supervisé et expliquer comment ils apprennent à partir des données et comment ils font des prévisions.

- On discutera la notion de complexité pour les modèles qu’ils permettent d’obtenir.

- On examinera leurs avantages et leurs inconvénients et le type de données sur lesquels ils sont les plus efficaces.

- On présentera la signification de leurs paramètres les plus importants.

- Enfin, la plupart de ces algorithmes ayant une variante classification et une variante régression, on présentera celles-ci.

Données

- On va utiliser deux jeux de données réelles pour illustrer les différents algorithmes.

- Tous deux sont disponibles sous scikit-learn.

- Le premier, qu'on désignera sous le nom de cancer, regroupe des mesures cliniques faites sur des tumeurs du sein. Chaque tumeur est labellisée comme « bénigne » ou « maligne ».

- L'objectif est de prévoir si une tumeur est maligne à partir des mesures effectuées.

Données

- Les données peuvent être lues à l'aide de la fonction load_breast_cancer de scikit-learn.

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()

print("cancer.keys():
\n{}".format(cancer.keys()))
```

Données

- Résultat :

```
cancer.keys():

dict_keys(['feature_names', 'target',
'DESCR', 'target_names', `data'])
```

- Les jeux de données contenus dans scikit-learn sont généralement stockés comme des objets de type Bunch. On en a déjà vu un exemple avec iris.

Données

- Le jeu de données est composé de 569 lignes et de 30 variables :

```
print("Shape of cancer data :
{}".format(cancer.data.shape))
```

- Résultat :

```
Shape of cancer data : (569, 30)
```

Données

- Parmi les 569 tumeurs, 212 sont malignes et 357 sont bénignes :

```
print("Sample count per class:\n{}".format({n:
v for n, v in zip(cancer.target_names,
np.bincount(cancer.target))}))
```

- Résultat :

```
Sample count per class:

{'malignant': 212, 'benign': 357}
```

Données

- Afin de connaître la signification des variables, on peut consulter l'attribut feature_names :

```
print("Feature names:
\n{}".format(cancer.feature_names))
```

- Résultat :

```
Feature names:

['mean radius' 'mean texture' 'mean
perimeter' ..., 'worst concave points''worst
symmetry' 'worst fractal dimension']
```

Données

- D’avantage d’informations sur les données sont fournies dans cancer.DESCR.

- Le deuxième jeu de données concerne la régression.

- L’objectif est de prévoir la valeur médiane des maisons dans divers quartiers de la ville de Boston (Etats-Unis) à partir d’informations sur le taux de criminalité, la proximité à la rivière Charles, l’accessibilité de l’autoroute, etc.

- Le jeu de données contient 506 individus et 13 variables.

Données

```
from sklearn.datasets import load_boston

boston = load_boston()

print("Data shape:
{}".format(boston.data.shape))
```

- Résultat :

```
Data shape: (506, 13)
```

Données

- On peut accéder à plus d'informations sur le jeu de données à l'aide de l'attribut DESCR de l'objet boston.

- On va augmenter le jeu de données boston des interactions des 13 variables prises deux à deux.

- Pour cela, on définit la fonction suivante :

Données

```
from sklearn.preprocessing import MinMaxScaler, PolynomialFeatures

def load_extended_boston():
    boston = load_boston()

    X = boston.data

    X = MinMaxScaler().fit_transform(boston.data)

    X = PolynomialFeatures(degree=2, include_bias=False).fit_transform(X)

    return X, boston.target
```

Données

- On peut alors accéder aux nouvelles données en faisant :

```
X, y = load extended boston()

print("X.shape: {}".format(X.shape))
```

- Résultat :

```
X.shape: (506, 104)
```

Les k-plus proches voisins

k-NN

- L'algorithme k-NN (k-plus proches voisins, ang. k-nearest neighbors) est sans doute l'algorithme de machine learning le plus simple.

- La construction du modèle consiste simplement à stocker l'ensemble d'apprentissage.

- Afin de faire une prévision sur une nouvelle donnée, l'algorithme considère les k plus proches voisins de cette donnée dans l'ensemble d'apprentissage.

k-NN

- Dans le cas du problème de classification, la décision est prise à l'aide d'un vote à la majorité : *on affecte à la nouvelle donnée la classe majoritaire parmi ses k plus proches voisins.*

- Pour choisir le nombre de voisins k, on peut se baser sur l'exactitude des prévisions sur l'échantillon test.

k-NN

```
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier


cancer = load_breast_cancer()


X_train, X_test, y_train, y_test = train_test_split(cancer.data,
                                                    cancer.target,
                                                    stratify=cancer.target,
                                                    random_state=66)

training_accuracy = []

test_accuracy = []
```

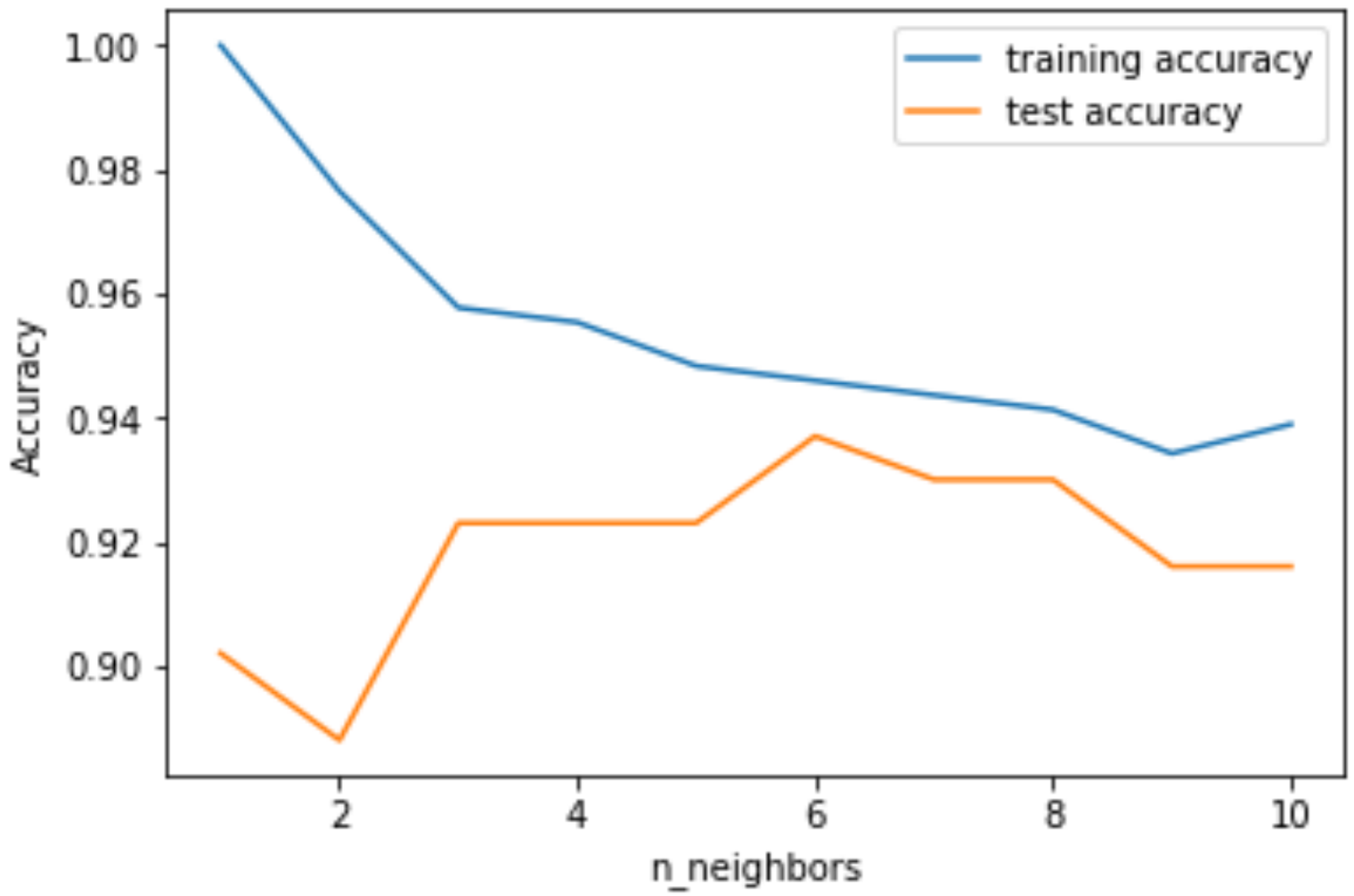
k-NN

```
neighbors_settings = range(1,11)

for n_neighbors in neighbors_settings:

    clf = KNeighborsClassifier(n_neighbors=n_neighbors)

    clf.fit(X_train,y_train)

    training_accuracy.append(clf.score(X_train,y_train))

    test_accuracy.append(clf.score(X_test,y_test))
```

k-NN

```
plt.plot(neighbors_settings, training_accuracy, label = "training accuracy")
plt.plot(neighbors_settings, test_accuracy, label= "test accuracy")

plt.ylabel("Accuracy")

plt.xlabel("n_neighbors")

plt.legend()
```

k-NN


training accuracy
test accuracy
1.00
0.98
0.96
0.94
0.92
0.90
Accuracy
2
4
6
8
10
n_neighbors

k-NN

- Dans l'interprétation de ce graphique, il faut faire attention au fait que *les valeurs plus faibles de k correspondent à des modèles plus complexes*.

- Avec un seul voisin, la prévision sur l'ensemble d'apprentissage est parfaite et lorsqu'on prend plus de voisins, le modèle devient plus simple et l'exactitude baisse.

- L'exactitude sur l'échantillon test pour un seul voisin est inférieure à celle obtenue avec plus de voisins, ce qui indique que le modèle à un seul voisin est trop complexe.

- D'autres part, avec 10 voisins, on a un modèle trop simple et la performance est pire.

k-NN

- La meilleure performance est obtenue entre les deux, aux alentours de 6 voisins.

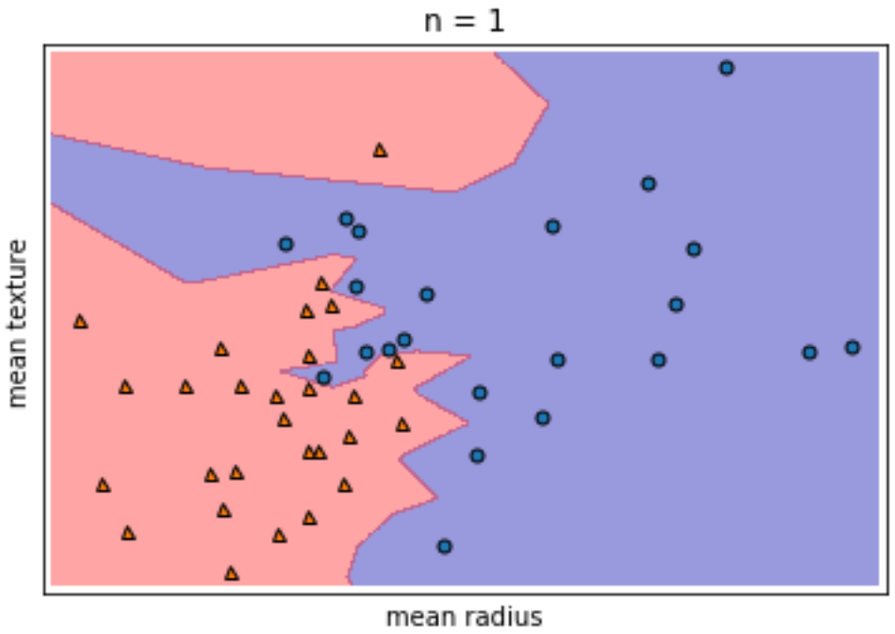
- Ceci dit, il est bon de garder l'échelle du graphique présente à l'esprit : au pire on a une exactitude de 88%, ce qui peut être largement suffisant pour certaines applications.

k-NN

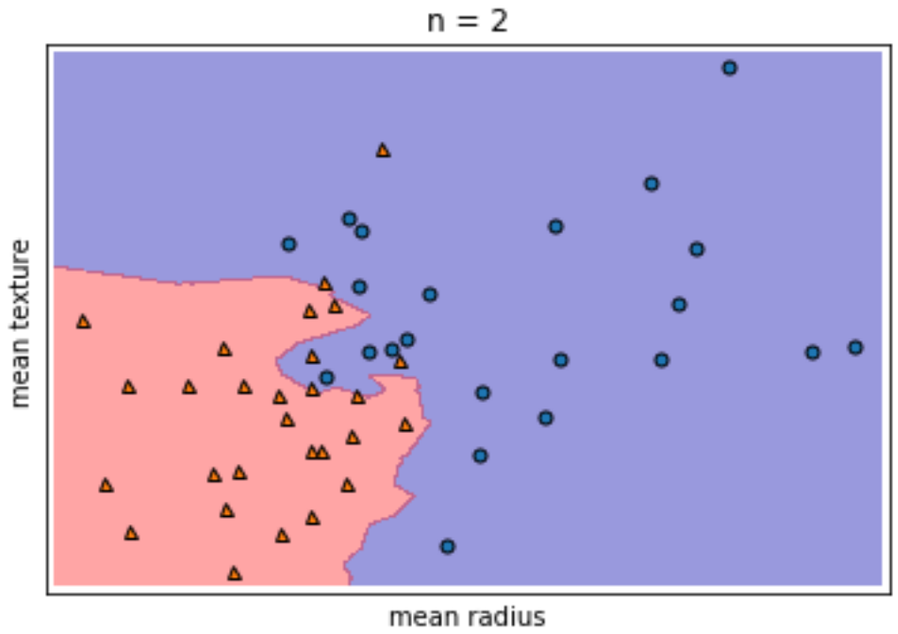
- Afin d'illustrer graphiquement l'effet du nombre de voisins *n*, considérons la classification par *n-plus proches voisins* d'un échantillon de taille 50, extrait de l'ensemble d'apprentissage, en tumeurs malignes et bénignes à partir des deux premières variables : *mean radius* et *mean texture*.

- Le programme correspondant est disponible sur le forum sous le nom *knnclassif.py*.

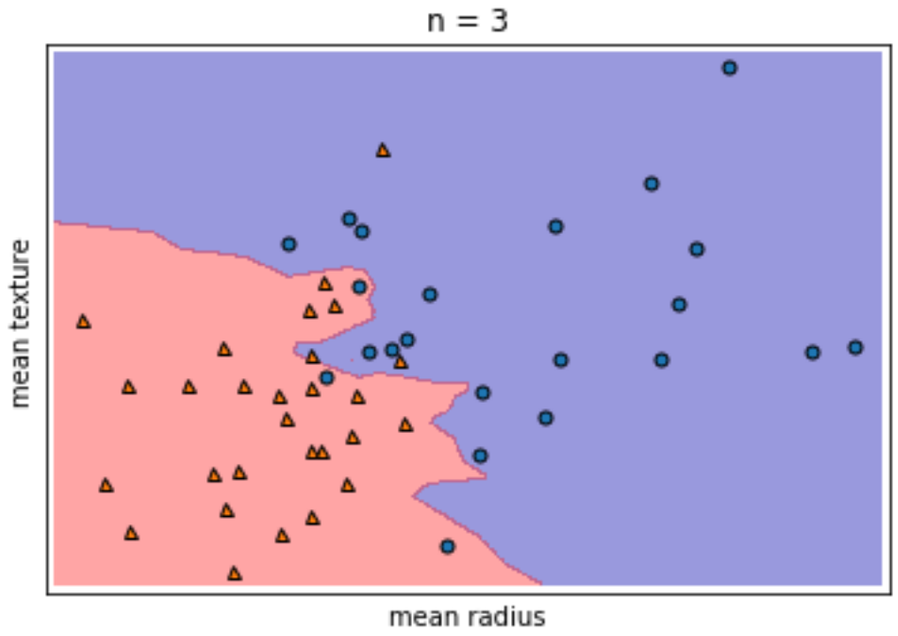
k-NN



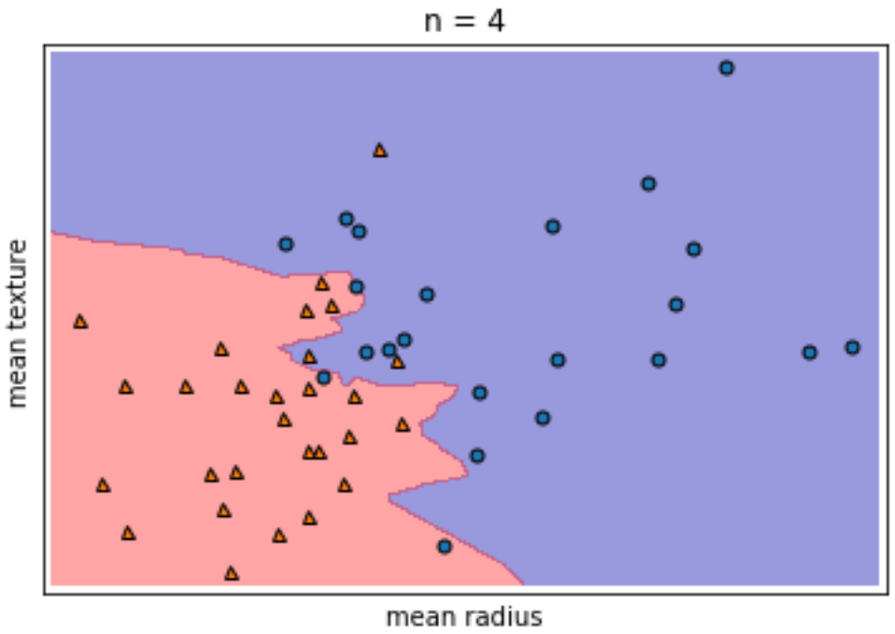
k-NN



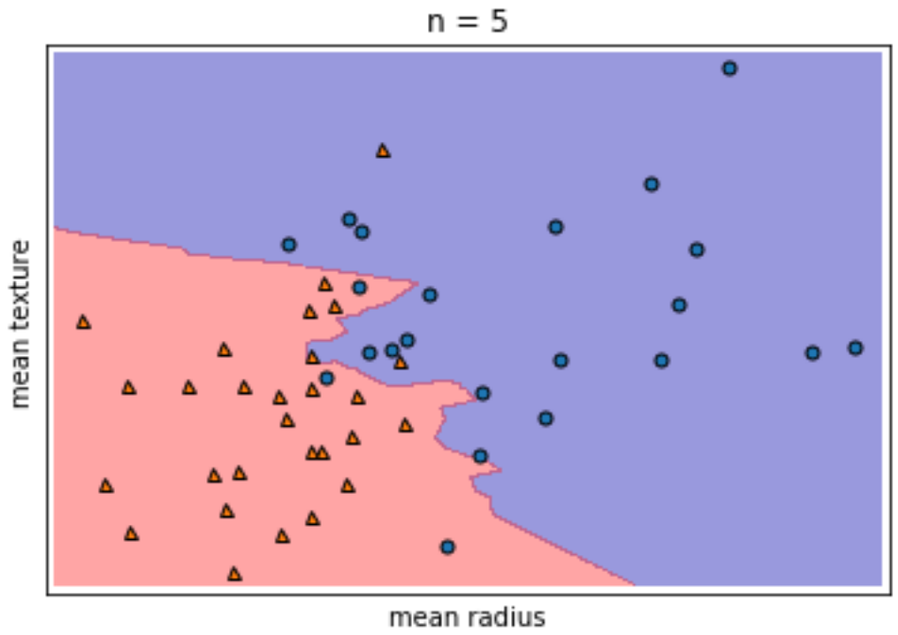
k-NN



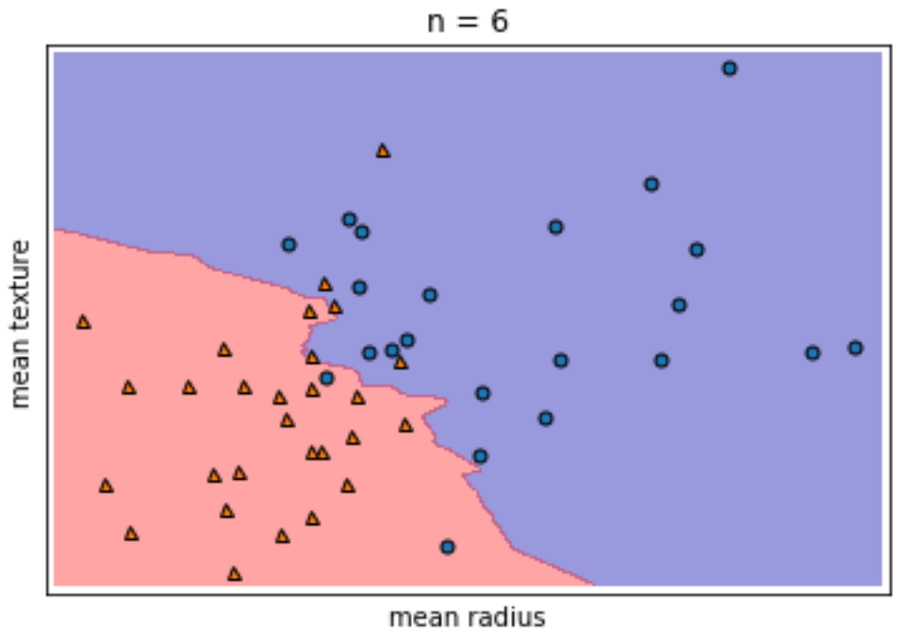
k-NN



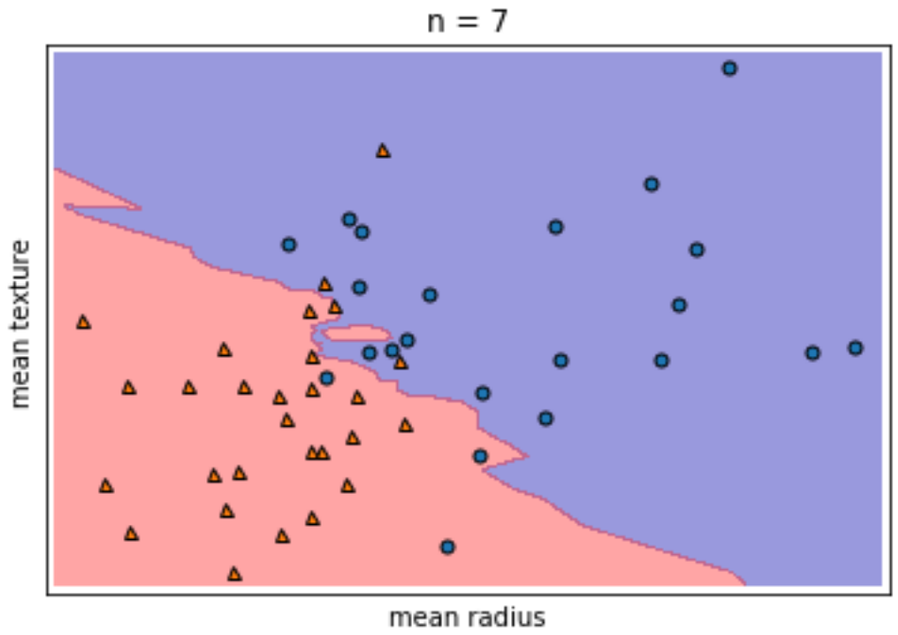
k-NN



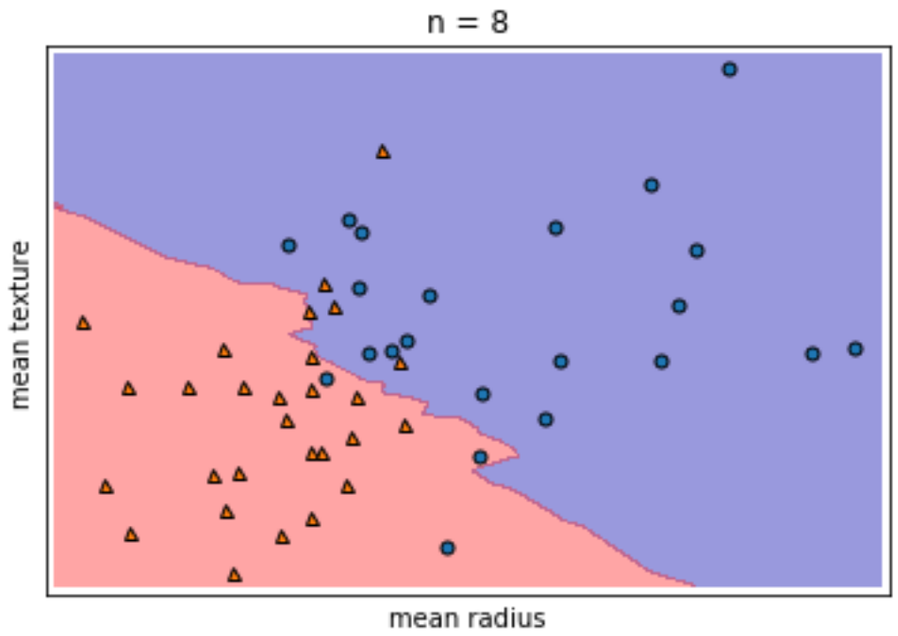
k-NN



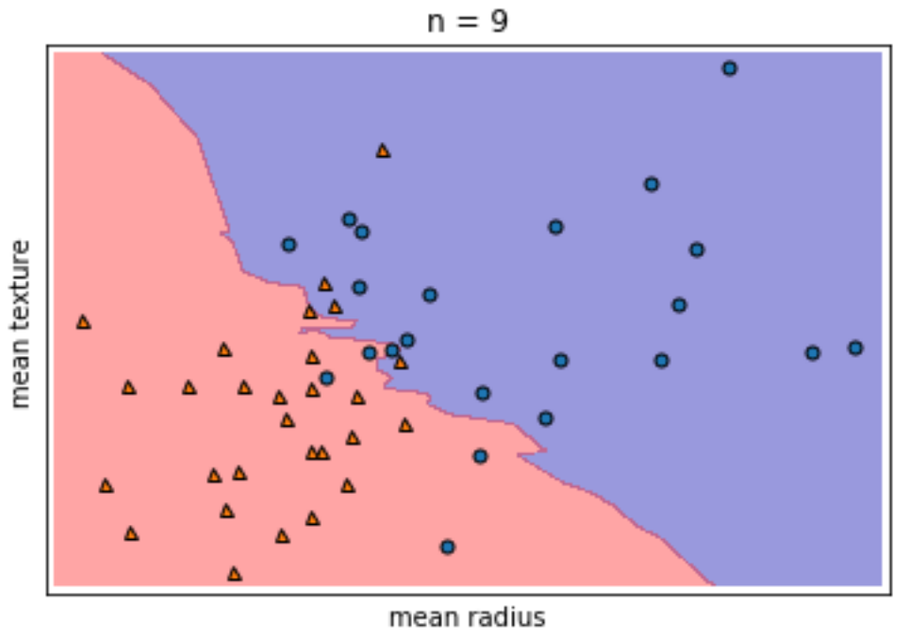
k-NN



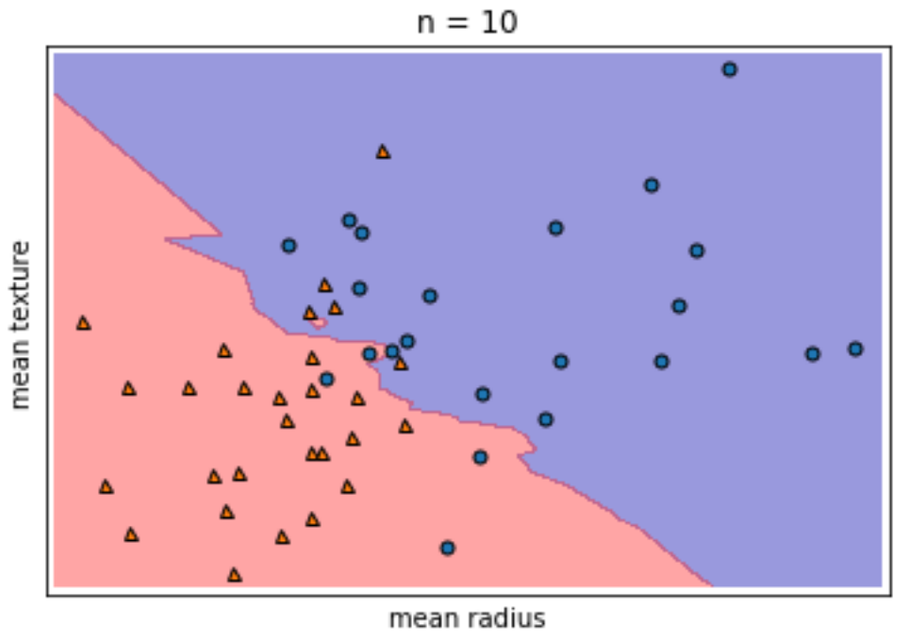
k-NN



k-NN



k-NN



k-NN

- Il existe une variante régression de l'algorithme des k-plus proches voisins.

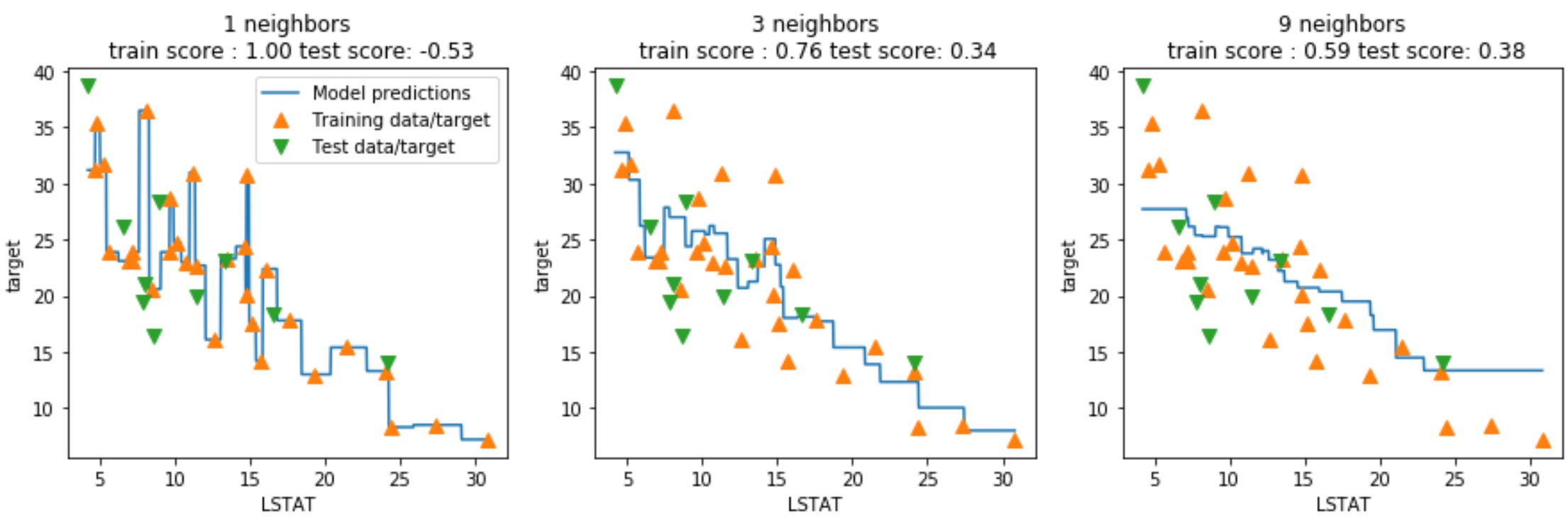
- Pour une nouvelle donnée, la prévision correspondante est la moyenne de la variable cible sur les k-plus proches voisins de la donnée dans l'ensemble d'apprentissage.

- Examinons l'effet du nombre de voisins à partir d'un échantillon de taille 40 issu du jeu de données boston.

- On veut prévoir la variable cible (prix médian des maisons dans un quartier) à partir de la variable LSTAT (proportion de ménages défavorisés dans le quartier).

k-NN

- On obtient le graphique suivant :



k-NN

- Les modèles sont évalués à l'aide de la méthode score, qui pour les régresseurs calcule le R2, ou coefficient de détermination.

- Le R2 est compris entre 0 et 1 si la régression obtenue est au moins aussi bonne que l'ajustement d'une constante, mais il peut être négatif dans le cas contraire.

k-NN

- Les graphiques précédents ont été obtenus à l'aide d'un programme Python disponible sur le Forum sous le nom *knnreg.py*.

- Il fait intervenir les méthodes fit, pour ajuster le modèle, predict pour faire des prévisions et score pour calculer le R2, associées à un objet de type KNeighborsRegressor, instanciant le modèle.

k-NN

- L'algorithmes des k-plus proches voisins pour la classification est contrôlé par deux paramètres essentiels : le nombre de voisins et la distance utilisée pour mesurer la proximité.

- En pratique, un nombre de voisins compris entre 3 et 5 fonctionne souvent bien, mais il faut tout de même ajuster ce paramètre.

- Le choix de la distance est plus compliqué. Par défaut, on utilise la distance euclidienne, qui fonctionne bien dans de nombreux contextes.

k-NN

- Un des avantages de k-NN est que le modèle est très simple à comprendre et a souvent de bonne performances sans nécessiter trop d’ajustements.

- L’algorithme k-NN est la méthode de base à utiliser avant des méthodes plus avancées et un bon point de comparaison.

- Construire le modèle est généralement rapide, mais lorsque la base d’apprentissage est de taille importante, la prévision peut être lente.

k-NN

- L'algorithme k-NN ne fonctionne généralement pas bien sur des jeux de données ayant un grand nombre de variables (plus d'une centaine) et il fonctionne particulièrement mal pour les jeux de données où les variables valent 0 la plupart du temps (données sparse).

- Ainsi, bien que l'algorithme k-NN soit facile à comprendre, il n'est pas souvent utilisé dans les applications réelles, en raison de la lenteur de la prévision et de son inaptitude à prendre en compte un grand nombre de variables.

Modèles linéaires de régression

Modèles linéaires

- Les modèles linéaires sont une classe de modèles très utilisés en pratique et ont été étudiés dans le détail au cours des dernières décennies, bien qu'ils remontent à plus d'un siècle.

- Les modèles linéaires font leur prévision à l'aide d'une fonction linéaire des variables en entrée (variables explicatives).

Modèles linéaires

- Pour la régression, la formule de prévision d'un modèle linéaire a la forme suivante :

y.pred = w[0]*x[0]+w[1]*x[1]+…+w[p]*x[p]+b

- y.pred désigne la prévision, x[i] la (i+1)-ème variable explicative et w et b sont les paramètres appris sur l'ensemble d'apprentissage.

- Si on n'a qu'une seule variable en entrée, on obtient un modèle linéaire simple :

y.pred = w[0]*x[0]+b

MCO

MCO

- La régression linéaire par moindres carrés ordinaires est la méthode linéaire classique et la plus simple de régression.

- L’algorithme des MCO (Moindres Carrés Ordinaires) détermine les paramètres w et b qui minimisent l’erreur quadratique moyenne (EQM) entre les prévisions et les valeurs cibles sur l’ensemble d’apprentissage.

- L’EQM est la somme des carrés des différences entre les prévisions et les vraies valeurs.

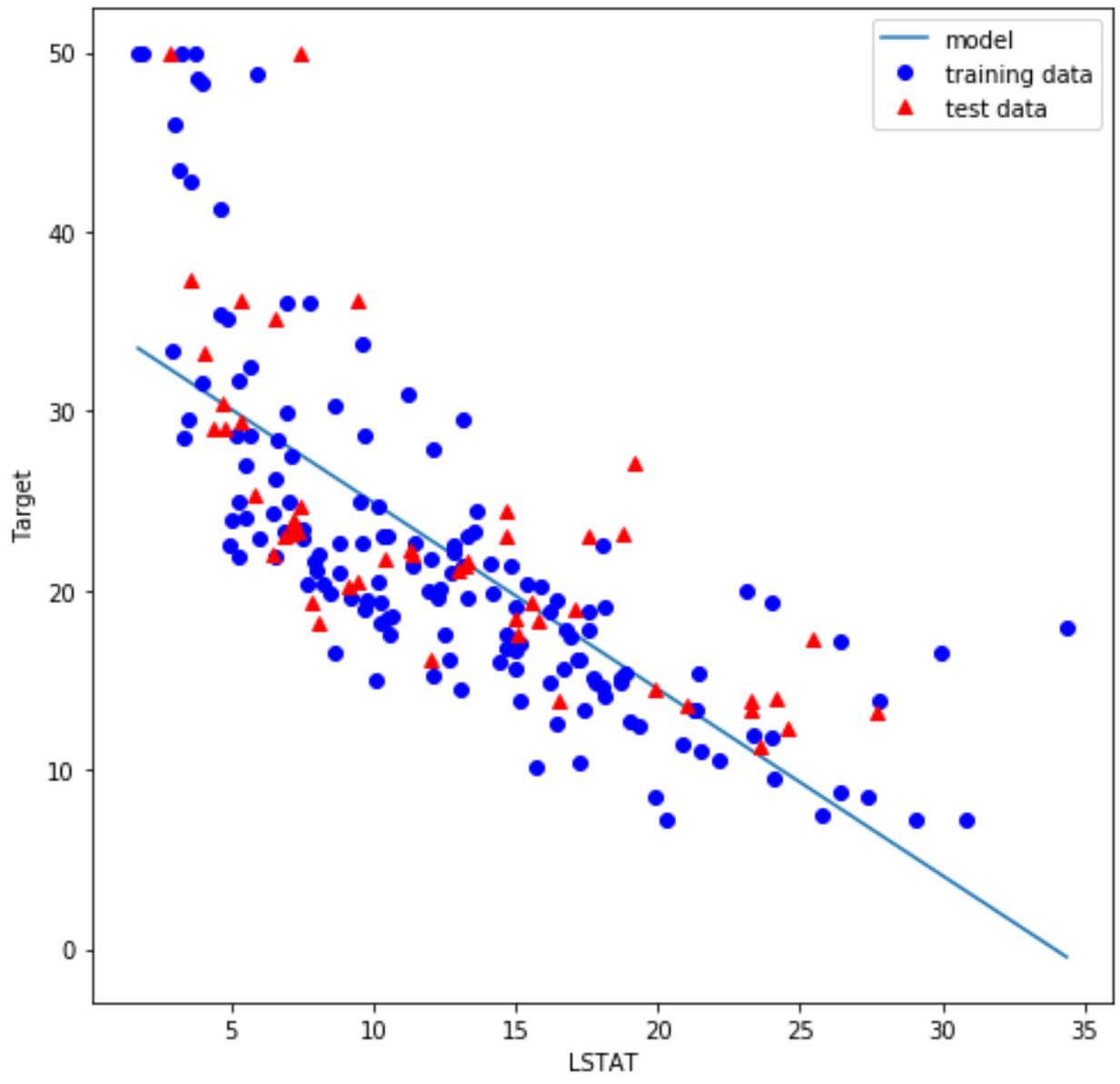
MCO

- L’algorithme des MCO n’a pas de paramètres de contrôle, ce qui est pratique, mais du coup on ne peut contrôler la complexité du modèle.

- Le programme *mco.py* effectue une régression par MCO de la variable cible sur la variable LSTAT pour un échantillon de taille 200 issu des données boston.

- On obtient le graphique suivant :

MCO


model
training data
test data
Target
LSTAT
50
40
30
20
10
0
5
10
15
20
25
30
35

MCO

- Le code permettant d'ajuster le modèle est le suivant :

```
from sklearn.linear_model import LinearRegression
```

...

```
lr = LinearRegression().fit(X_train, y_train)
```

- La qualité de l'ajustement peut être évaluée à l'aide de la méthode score :

MCO

```
print("Training set score: {:.2f}".format(lr.score(X_train,y_train)))

print("Test set score: {:.2f} ».format(lr.score(X_test,y_test)))
```

- Résultat :

```
Training set score: 0.56

Test set score: 0.46
```

MCO

- Le R2, égal à 0.56 sur les données d’apprentissage, n’est pas bon et il est tout aussi mauvais sur les données de test.

- Cela signifie que le modèle n’est pas adapté à la structure des données ; il est sans doute trop rigide. Il y a sous-apprentissage.

MCO

- Considérons à présent l'ensemble des données boston, avec toutes les interactions des variables prises deux à deux.

- On soumet le code suivant, extrait du programme python *mco.py* :

```
lr = LinearRegression().fit(X_train, y_train)

print("Training set score: {:.2f}".format(lr.score(X_train,y_train)))

print("Test set score: {:.2f}".format(lr.score(X_test,y_test)))
```

MCO

- Résultat :

```
Training set score: 0.95
```

```
Test set score: 0.61
```

- L’écart entre la performance sur l’ensemble d’apprentissage et l’ensemble de test est un signe de sur-apprentissage. On doit donc chercher un modèle qui permette de contrôler la complexité.

Régression Ridge

Régression Ridge

- La régression ridge est également un modèle linéaire de régression.

- Par conséquent, la formule de prévision est la même que celle des mco.

- La différence est que les coefficients sont estimées de façon à ce que la fonction de prévision s'ajuste au mieux à l'ensemble d'apprentissage sous une contrainte additionnelle.

Régression Ridge

- Celle-ci spécifie que les coefficients w doivent être aussi petits, c’est-à-dire proches de zéro, que possible.

- Intuitivement, cela signifie que chaque variable explicative doit avoir aussi peu d’effet sur la réponse que possible, tout an ayant globalement un modèle avec un bon pouvoir prédictif.

- Cette contrainte est un exemple de la technique de régularisation, très utilisée en machine learning.

Régression Ridge

- Régulariser un problème signifie lui imposer des contraintes pour éviter le sur-apprentissage.

- Le type de contrainte imposée par la régression ridge est appelée régularisation L2.

- Sous Python, la régression ridge est implémentée par la classe

`linear_model.Ridge`

Régression Ridge

- Examinons ce que la régression ridge donne sur les données boston (extrait du programme *ridge.py*) :

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train,y_train)

print("Training set score: {:.2f}".format(ridge.score(X_train,y_train)))

print("Test set score: {:.2f}".format(ridge.score(X_test,y_test)))
```

- Résultat :

Régression Ridge

```
Training set score: 0.89

Test set score: 0.75
```

- Le score d'apprentissage de Ridge est inférieur à celui de LinearRegression (0.95) sur nos données, mais le score de test est supérieur (0.75 > 0.61).

- C'est cohérent avec nos attentes.

- Avec la régression linéaire, il y a sur-apprentissage.

- Ridge étant un modèle plus contraint, on a un risque de sur-apprentissage inférieur.

Régression Ridge

- *Prendre un modèle moins complexe dégrade la performance sur l'ensemble d'apprentissage mais améliore la capacité à généraliser.*

- Comme on s'intéresse uniquement à la prévision, on doit préférer le modèle de régression ridge au modèle de régression par mco pour nos données.

Régression Ridge

- Le modèle Ridge effectue un arbitrage entre simplicité du modèle (coefficients proches de zéro) et performance sur l'ensemble d'apprentissage.

- L'importance relative de la contrainte de simplicité par rapport à la qualité de l'ajustement sur l'ensemble d'apprentissage est contrôlée par le paramètre alpha.

- Dans l'exemple précédent, on a utilisé la valeur par défaut alpha = 1.0.

Régression Ridge

- Ce choix était toutefois arbitraire; il n'est pas certain qu'il fournisse le meilleur compromis entre simplicité du modèle et performance sur l'ensemble d'apprentissage.

- Le choix optimal de alpha dépend du jeu de données qu'on modélise.

- Faire croître alpha rapproche les coefficients de zéro, ce qui dégrade la performance du modèle sur l'ensemble d'apprentissage mais peut l'améliorer en termes de généralisation.

Régression Ridge

- Exemple :

```
ridge10 = Ridge(alpha=10).fit(X_train,y_train)

print("Training set score: {:.2f}".format(ridge10.score(X_train,y_train)))

print("Test set score: {:.2f} ».format(ridge10.score(X_test,y_test)))
```

- Résultat :

```
Training set score: 0.79

Test set score: 0.64
```

Régression Ridge

- Prendre alpha = 10 dégrade les performance du modèle sur l'ensemble d'apprentissage et l'ensemble test.

- Examinons le cas alpha = 0.1 :

```
ridge01 = Ridge(alpha=0.1).fit(X_train,y_train)

print("Training set score: {:.2f}".format(ridge01.score(X_train,y_train)))

print("Test set score: {:.2f}".format(ridge01.score(X_test,y_test)))
```

Régression Ridge

- Résultat :

```
Training set score: 0.93
Test set score: 0.77
```

- La valeur alpha = 0.1 semble bien fonctionner et on pourrait d'ailleurs faire décroître alpha un peu plus pour améliorer la capacité de généralisation du modèle.

- On verra plus précisément comment sélectionner le paramètre alpha dans la suite du cours.

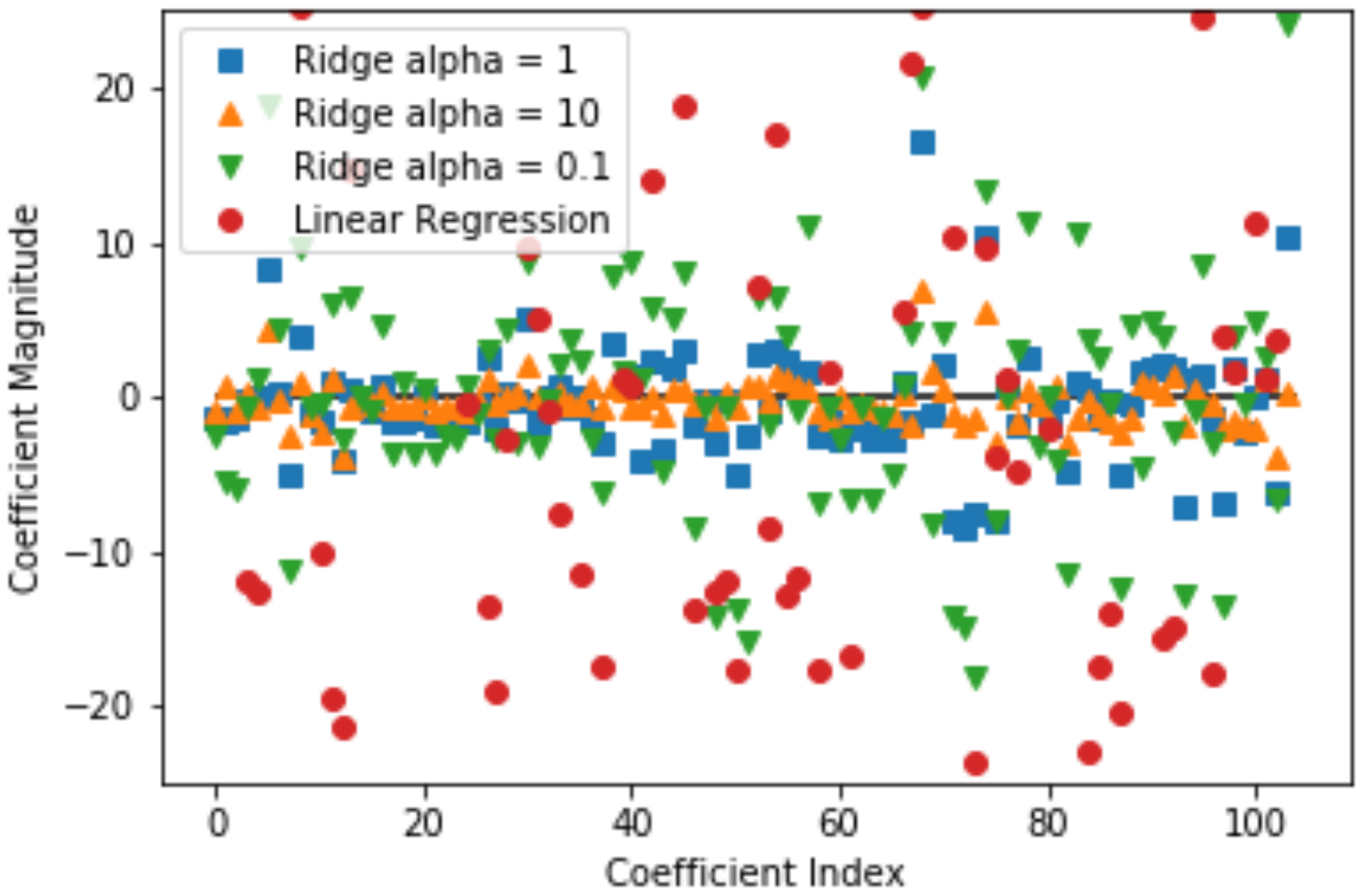
Régression Ridge

- On peut avoir une vision qualitative de la façon dont le paramètre alpha impacte le modèle en sauvegardant la valeur de l'attribut coef_ des modèles obtenus pour différentes valeurs de alpha.

- Plus alpha est grand, plus le modèle est contraint et plus on s'attend à avoir de faibles valeurs des coefficients.

- C'est confirmé par le graphique suivant (code *ridge.py*) :

Régression Ridge



Régression Ridge

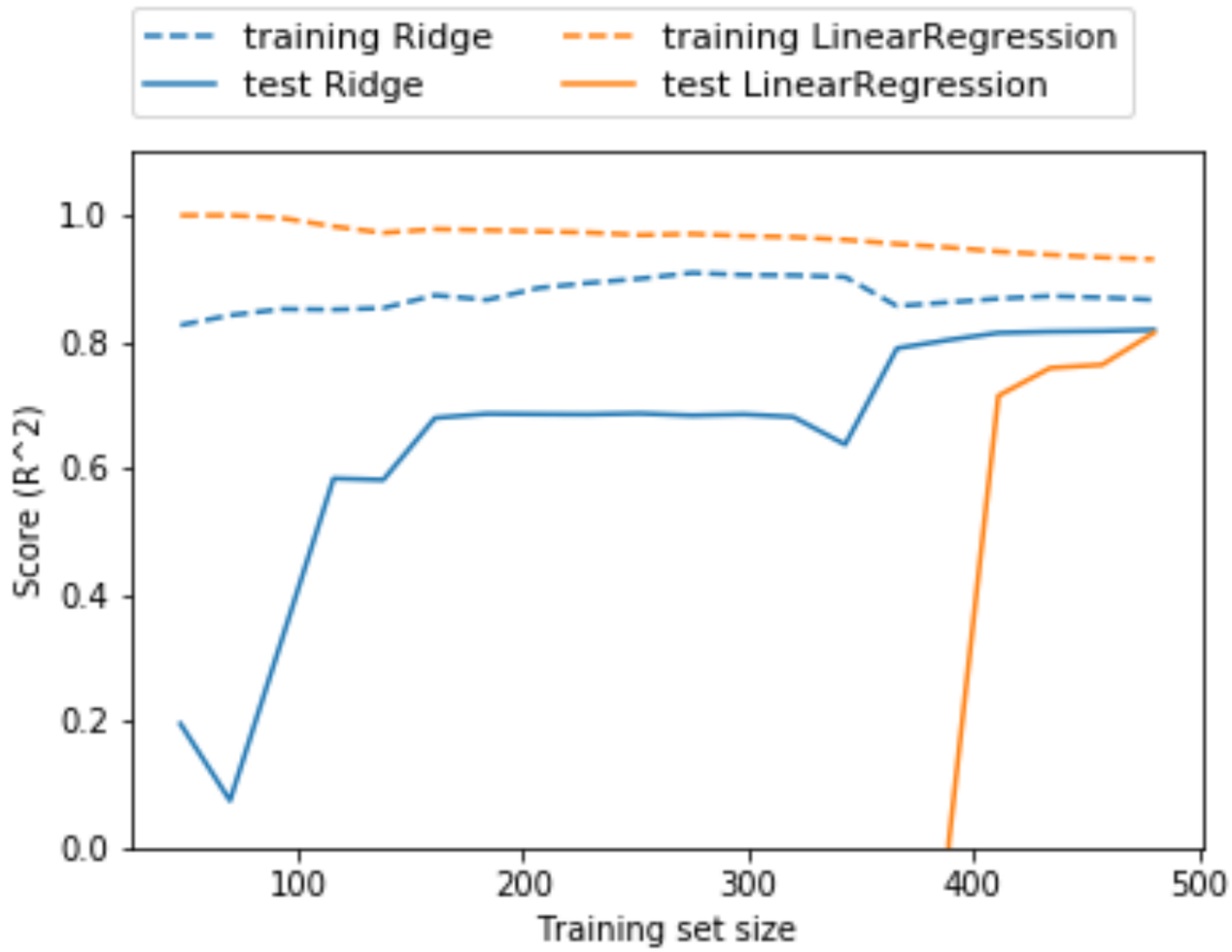
- Une autre façon de comprendre l'influence de la régularisation consiste à fixer une valeur de alpha et à faire varier la quantité de données d'apprentissage.

- Pour obtenir le graphique suivant, on a échantillonné le jeu de données boston et évalué LinearRegression et Ridge(alpha=1) sur des sous-ensembles de taille croissante.

- Le graphique obtenu est un exemple de courbe d'apprentissage (ang. learning curve).

- Le programme correspondant sur le Forum est *ridge_learning_curve.py*.

Régression Ridge



Régression Ridge

- Comme on pouvait s'y attendre, le score d'apprentissage est plus élevé que le score de test pour toutes les tailles de données, dans le cas de la régression ridge comme dans le cas de la régression linéaire par mco.

- Toutefois, le score de test de la régression ridge est meilleur, notamment pour les petites tailles de données.

- Pour des tailles d'échantillon inférieures à 400, la régression linéaire par mco échoue à apprendre quoi que ce soit.

- Plus la taille de l'échantillon d'apprentissage devient importante, plus la performance de la régression linéaire par mco se rapproche de celle de la régression ridge.

Régression Ridge

- Ainsi, plus on a de données d'apprentissage, moins l'effet de la régularisation est important et lorsqu'on a suffisamment de données, régression linéaire par mco et ridge donnent les mêmes résultats.

- Un autre point intéressant est la décroissance du score d'apprentissage pour la régression linéaire par mco : plus il y a de données, plus il est difficile pour un modèle de les sur-ajuster ou de les mémoriser.

Lasso

Lasso

- La régression Lasso est une alternative à la régression ridge pour régulariser la régression linéaire.

- Le Lasso impose également la contrainte d'avoir des coefficients proches de zéro mais d'une façon légèrement différente, appelée régularisation L1.

- Celle-ci a pour effet que certains coefficient sont exactement égaux à zéro.

Lasso

- Cela signifie que certaines variables dont entièrement ignorées par le modèle. La régression Lasso inclut donc une forme de sélection de variables.

- Du coup, les modèles obtenus sont plus faciles à interpréter et révèlent les structures les plus importantes des données.

- Appliquons la régression Lasso aux données boston (extraits du programme *lasso_boston.py*) :

Lasso

```
from sklearn.linear_model import Lasso

...

lasso = Lasso().fit(X_train,y_train)

print("Training set score: {:.2f}".format(lasso.score(X_train,y_train)))

print("Test set score: {:.2f}".format(lasso.score(X_test,y_test)))

print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

Lasso

- Résultat :

```
Training set score: 0.29

Test set score: 0.21

Number of features used: 4
```

- La performance de Lasso est mauvaise pour les données boston, à la fois sur les données d'apprentissage et de test.

- Cela signifie qu'il y a sous-apprentissage.

- Il s'avère que seules 4 variables sont utilisées par le modèle sur les 105 disponibles.

Lasso

- De même que la régression ridge, Lasso admet un paramètre de régularisation alpha, qui contrôle l'intensité avec laquelle les coefficients sont poussés vers zéro.

- Dans l'exemple précédent, on a utilisé la valeur par défaut de alpha, qui vaut 1.

- Pour réduire le sous-apprentissage, faisons décroître alpha. Lorsqu'on fait cela, on doit faire croître la valeur du paramètre max_iter, qui contrôle le nombre maximum d'itérations lors de l'ajustement du modèle.

Lasso

- Extrait du programme *lasso_boston.py* :

```
lasso001 = Lasso(alpha=0.01,max_iter=100000).fit(X_train,y_train)


print("Training set score: {:.
2f}".format(lasso001.score(X_train,y_train)))


print("Test set score: {:.2f}".format(lasso001.score(X_test,y_test)))


print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```

Lasso

- Résultat :

```
Training set score: 0.90

Test set score: 0.77

Number of features used: 33
```

- Un alpha plus petit correspond à un modèle plus complexe, qui a de meilleurs performance sur l'échantillon d'apprentissage et l'échantillon test.

- La performance est légèrement meilleure qu'avec Ridge et on n'utilise que 33 des 105 variables disponibles, ce qui rend le modèle plus facile à comprendre.

Lasso

- Toutefois, si alpha est trop petit, on supprime l'effet de la régularisation et il y a sur-apprentissage, avec un résultat similaire à la régression linéaire par mco (extrait de *lasso_boston.py*) :

```
lasso00001 = Lasso(alpha=0.0001,max_iter=100000).fit(X_train,y_train)


print("Training set score: {:.
2f}".format(lasso00001.score(X_train,y_train)))


print("Test set score: {:.2f}".format(lasso00001.score(X_test,y_test)))


print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))
```

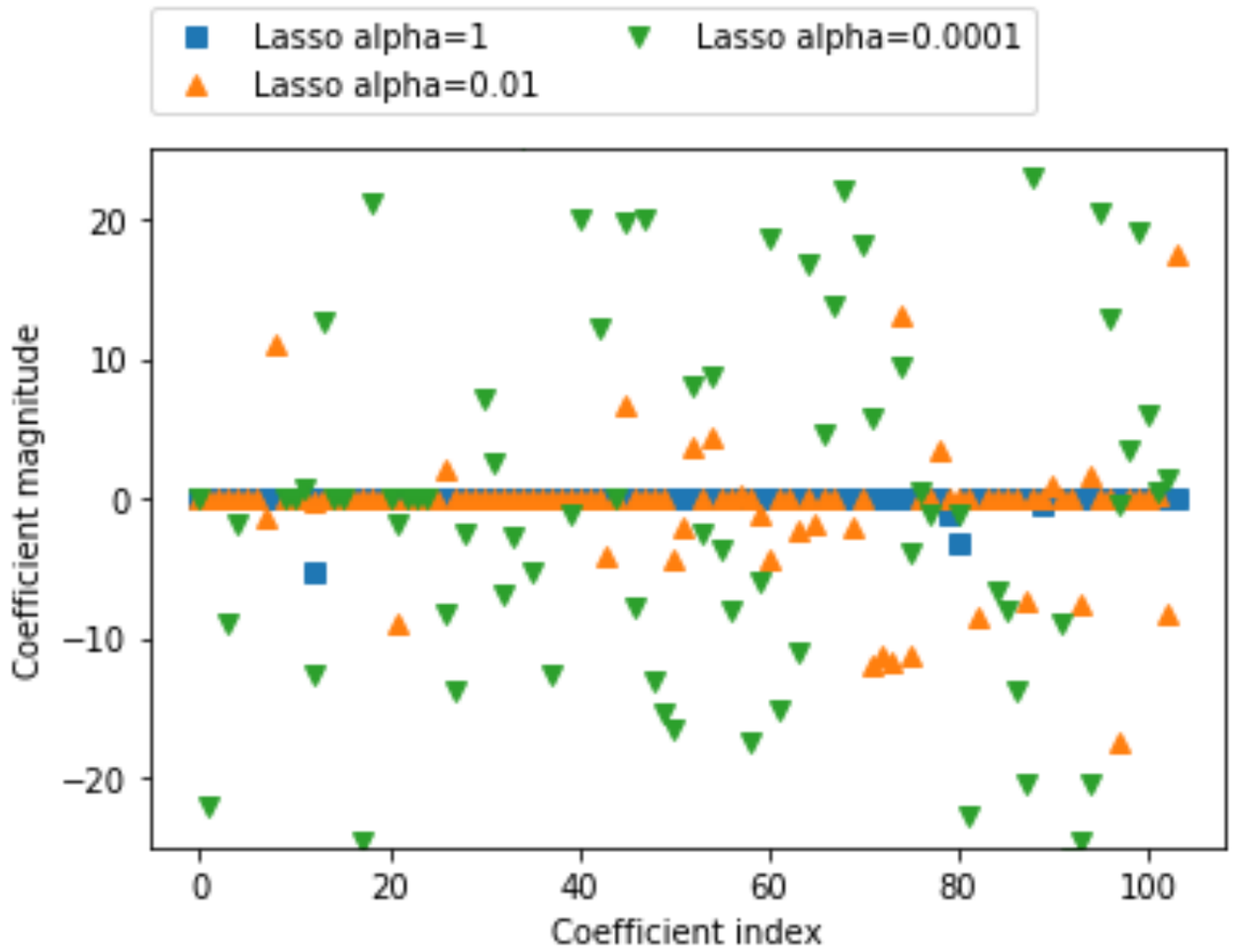
Lasso

- Résultat :

```
Training set score: 0.95

Test set score: 0.64

Number of features used: 94
```

- Nous pouvons représenter graphiquement les coefficients des différents modèles linéaires obtenus jusqu’ici :

Lasso


Lasso alpha=1
Lasso alpha=0.0001
Lasso alpha=0.01
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
20
40
60
80
100

Lasso

- En pratique, on préfère souvent Ridge à Lasso à performances similaires.

- L'avantage de Lasso réside dans l'interprétabilité du modèle obtenu.

- Scikit-learn met aussi à disposition la classe ElasticNet, qui combine les régularisations Lasso et Ridge.

- En pratique, ElasticNet est ce qui marche le mieux, mais on a deux paramètres à ajuster, un pour la régularisation L1 et un pour la régularisation L2.

Modèles linéaires de classification binaire

Modèles linéaires

- Les modèles linéaires sont également très utilisés en classification (reconnaissance de formes, analyse discriminante).

- Considérons d'abord le cas de la classification binaire.

- Dans ce cas, une prévision est faite à l'aide de la formule :

```
y.hat = w[0]*x[0]+…+w[p]*x[p]+b > 0
```

Modèles linéaires

- La formule est très similaire à celle utilisée pour la régression, si ce n'est qu'au lieu de renvoyer la somme pondérée des variables, on compare cette valeur à un seuil égal à zéro.

- Si la fonction prend une valeur inférieure à zéro, on prédit la classe -1; si elle prend une valeur supérieure à zéro, on prédit la classe +1.

- *Cette règle est commune à tous les modèles linéaires de classification.*

Modèles linéaires

- Dans le cas de la régression linéaire, la réponse y.hat, est une fonction linéaire des variables : une droite, un plan ou un hyperplan (en dimensions supérieures).

- *Dans le cas des modèles linéaires de classification, c'est la frontière de décision qui est une fonction linéaire des variables.*

- En d'autres termes, une règle de décision linéaire binaire est une règle de décision qui sépare deux classes à l'aide d'une droite, d'un plan ou d'un hyperplan.

Modèles linéaires

- Il y a divers algorithmes d'apprentissage des modèles linéaires de classification.
- Ceux diffèrent de deux façons :

A. La façon dont ils mesurent la qualité de l'ajustement du modèle aux données d'apprentissage.

B. Le fait qu'il y ait régularisation ou pas et le type de régularisation utilisée s'il y en a.

Modèles linéaires

- Les deux algorithmes de classification linéaire d'usage le plus fréquent sont *la régression logistique*, implémentée sous Python par linear_model.LogisticRegression, et les machines à vecteurs supports linéaires (SVM linéaires), implémentées par svm.LinearSVC (SVC : support vector classification).

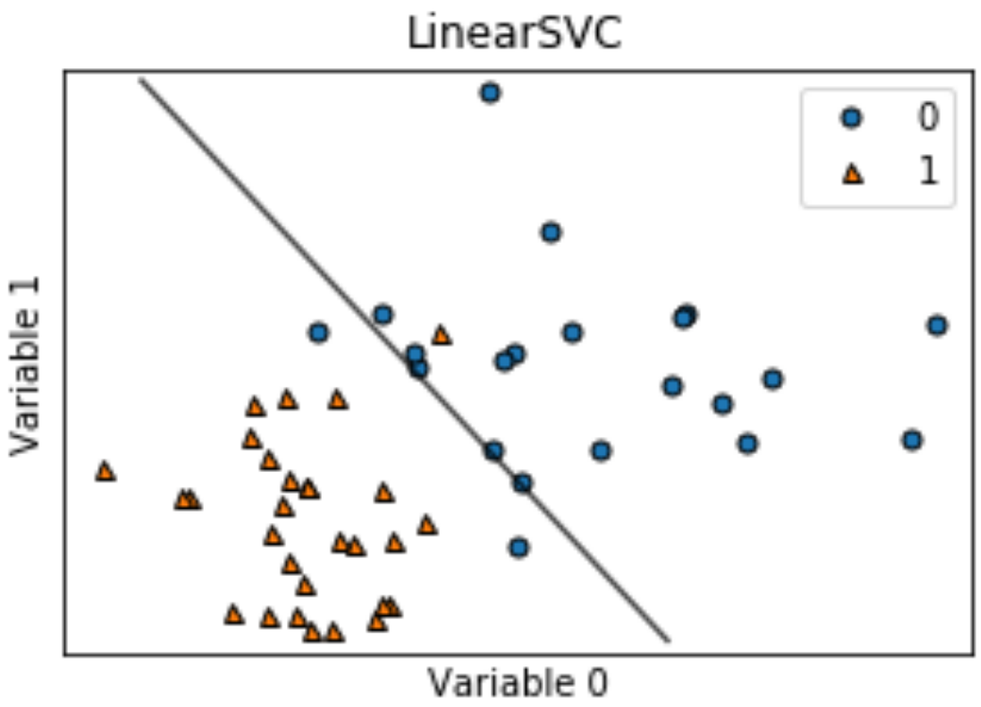
- Nous allons appliquer ces deux algorithmes à un échantillon de taille 50 issu des données cancer (extrait du programme *linear_classif.py*).

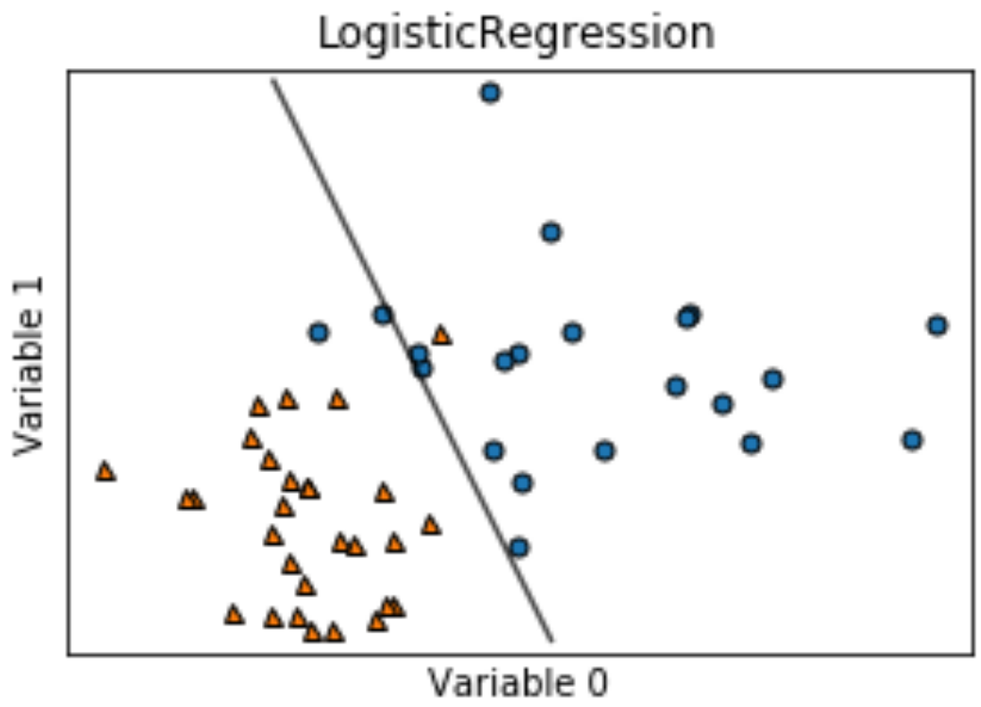
Modèles linéaires

```
from sklearn.linear_model import LogisticRegression

from sklearn.svm import LinearSVC

fig, axes = plt.subplots(1,2,figsize=(10,3))
```

```
for model, ax in zip([LinearSVC(), LogisticRegression()], axes):

    clf = model.fit(X,y)

    plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=.7)

    discrete_scatter(X[:,0], X[:,1], y, ax=ax)

    ax.set_title("{}".format(clf.__class__.__name__))

    ax.set_xlabel("Variable 0")

    ax.set_ylabel("Variable 1")

axes[0].legend()
```

Modèles linéaires





Modèles linéaires

- Les deux modèles obtenus ont des frontières de décision similaires, bien qu'elles ne soient pas confondues.

- Par défaut, les deux modèles utilisent une régularisation L2, comme le fait Ridge pour la régression.

- Pour LogisticRegression et LinearSVC, le paramètre contrôlant l'intensité de la régularisation est noté C.

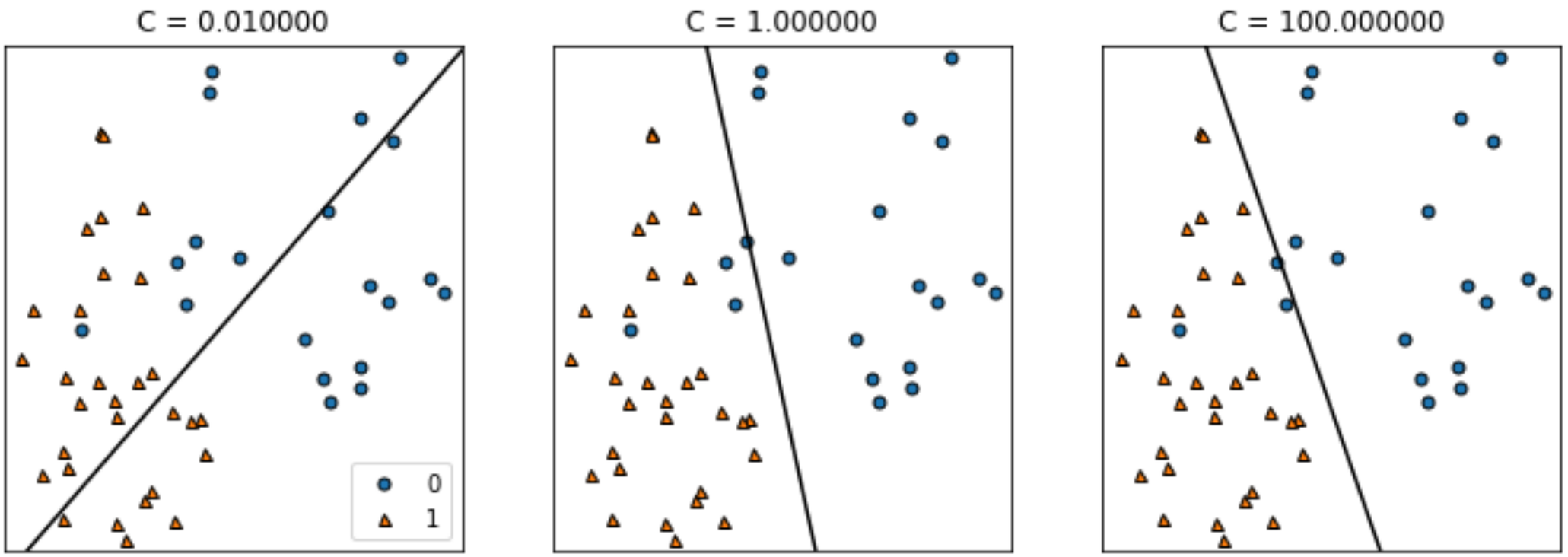
- *Des valeurs plus élevées de C correspondent à moins de régularisation.*

Modèles linéaires

- En d'autres termes, *quand on utilise une valeur élevée de C, LogisticRegression et LinearSVC essayent de s'ajuster le mieux possible aux données d'apprentissage* alors que *pour des valeurs faibles de C, les modèles mettent l'accent sur l'obtention d'un vecteur de coefficients w proche de zéro*.

- Un autre aspect important de l'influence de C est illustré par les graphiques suivants :

Modèles linéaires



Modèles linéaires

- Sur le graphique de gauche, on a un C très faible, correspondant à une régularisation importante.

- Celle-ci domine et les classes sont mal séparées.

- Sur le graphique du milieu, les classes sont mieux séparées et il n’y a que 3 tumeurs mal classées.

- Finalement, sur le graphique à droite, la frontière de décision sépare encore mieux les deux classes.

- Toutefois, ce dernier modèle est peut-être du coup en sur-apprentissage.

Modèles linéaires

- Comme pour la régression, les modèles linéaires de classification peuvent sembler très restrictifs dans les espaces de faible dimension, n'autorisant que des frontières de décision droites ou planes.

- Toutefois, en dimensions supérieures, ces modèles deviennent très puissants et il faut faire attention à un sur-apprentissage éventuel.

- Utilisons LogisticRegression sur les données cancer complètes (extrait de *linear_classif.py*) :

Modèles linéaires

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data,
                                                    cancer.target,
                                                    stratify=cancer.target,
                                                    random_state=42)
logreg = LogisticRegression().fit(X_train,y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train,y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test,y_test)))
```

Modèles linéaires

- Résultat :

```
Training set score: 0.953
```

```
Test set score: 0.958
```

- La valeur par défaut C = 1 donne de bons résultats, avec 95% d'exactitude sur les données d'apprentissage et de test. Toutefois, comme ces performances sont proches, il y a vraisemblablement sous-apprentissage.

Modèles linéaires

- Augmentons C pour avoir un modèle plus flexible :

```
logreg100 = LogisticRegression(C=100).fit(X_train,y_train)

print("Training set score: {:.3f}".format(logreg100.score(X_train,y_train)))

print("Test set score: {:.3f}".format(logreg100.score(X_test,y_test)))
```

- Résultat :

```
Training set score: 0.972

Test set score: 0.965
```

Modèles linéaires

- En fixant C = 100, on obtient une exactitude supérieure sur les données d'apprentissage et une exactitude légèrement supérieure sur les données de test, ce qui confirme notre intuition qu'un modèle plus complexe donnerait de meilleurs résultats.

- Examinons à présent ce qui se passe si on régularise davantage que le choix par défaut de C = 1.

- Fixons C = 0.01.

Modèles linéaires

```
logreg001 = LogisticRegression(C=0.01).fit(X_train,y_train)

print("Training set score: {:.3f}".format(logreg001.score(X_train,y_train)))

print("Test set score: {:.3f}".format(logreg001.score(X_test,y_test)))
```

- Résultat :

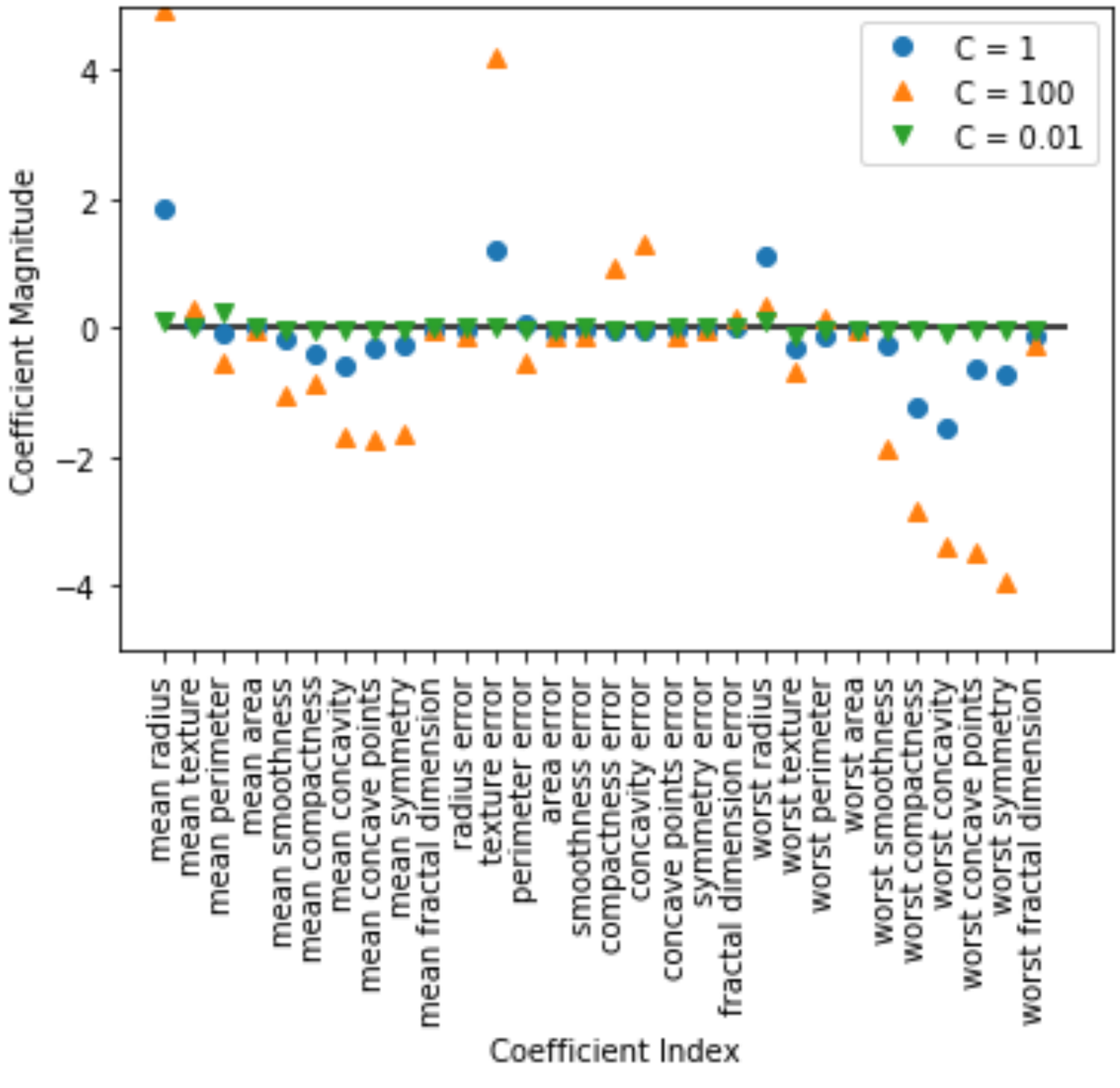
```
Training set score: 0.934

Test set score: 0.930
```

Modèles linéaires

- Comme on pouvait s'y attendre, il y a encore plus sous-apprentissage et la performance se dégrade sur les données test comme sur les données d'apprentissage.

- Finalement, examinons les coefficients du modèle obtenu pour les trois valeurs du coefficient de régularisation (programme *linear_classif.py*) :

Modèles linéaires

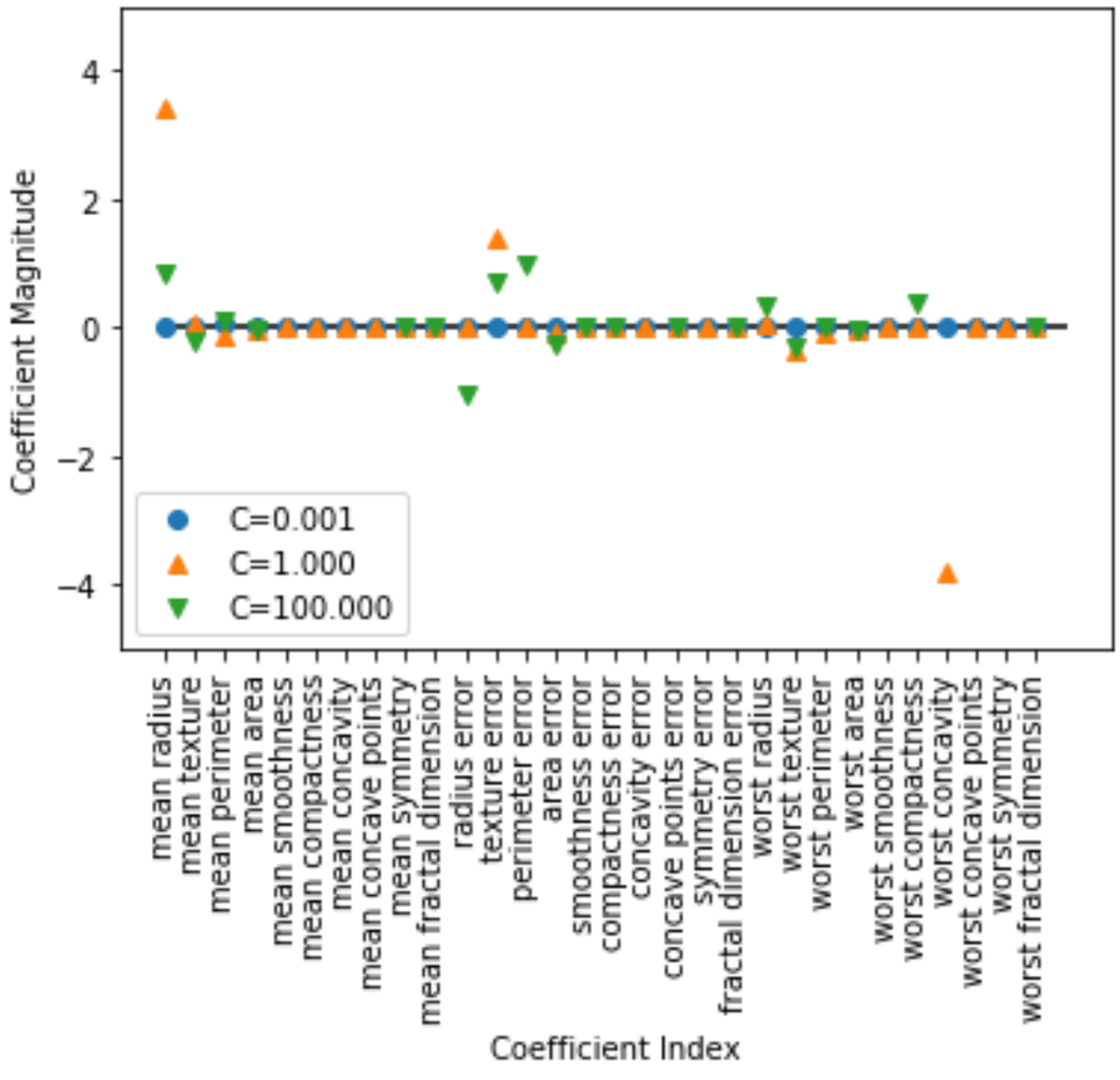


Modèles linéaires

- Si on veut un modèle plus interprétable, on peut utiliser la régularisation L1 car elle limite le modèle à quelques variables.

- Le graphique des coefficients est le suivant (programme *linear_classif.py*) :

Modèles linéaires



Modèles linéaires

- Les performances des différents modèle sont les suivantes :

```
Training accuracy of l1 logreg with C=0.001: 0.91

Test accuracy of l1 logreg with C=0.001: 0.92

Training accuracy of l1 logreg with C=1.000: 0.96

Test accuracy of l1 logreg with C=1.000: 0.96

Training accuracy of l1 logreg with C=100.000: 0.99

Test accuracy of l1 logreg with C=100.000: 0.98
```

Modèles linéaires

- Comme pour la régression linéaire, on constate que la régularisation L1 effectue une sélection de variables.

- Relevons également la très bonne performance en prévision pour C= 100.

Modèles linéaires de classification multi-classes

Modèles linéaires

- De nombreux modèles linéaires de classification sont spécifiques à la classification binaire et ne se généralisent pas *naturellement* à la classification multi-classes, à l'exception de la régression logistique.

- Une technique standard pour généraliser un algorithme de classification binaire à la classification multi-classes est l'approche one-vs.-rest.

Modèles linéaires

- Selon l'approche one-vs.-rest, un modèle binaire est appris pour chaque classe, en essayant de la séparer de toutes les autres classes.

- On obtient ainsi autant de modèles binaires qu'il y a de classes.

- Pour faire une prévision, tous les classificateurs binaires sont lancés sur une même donnée de test.

- *Le classificateur qui a le plus grand score sur sa classe est retenu et le label correspondant est renvoyé comme prévision.*

Modèles linéaires

- Comme on a un classificateur binaire par classe, on obtient un vecteur de coefficients w et un terme constant b pour chaque classe. La classe prédite est celle pour laquelle la formule suivant prend sa valeur la plus élevée :

```
w[0]*x[0]+…+w[p]*x[p]+b
```

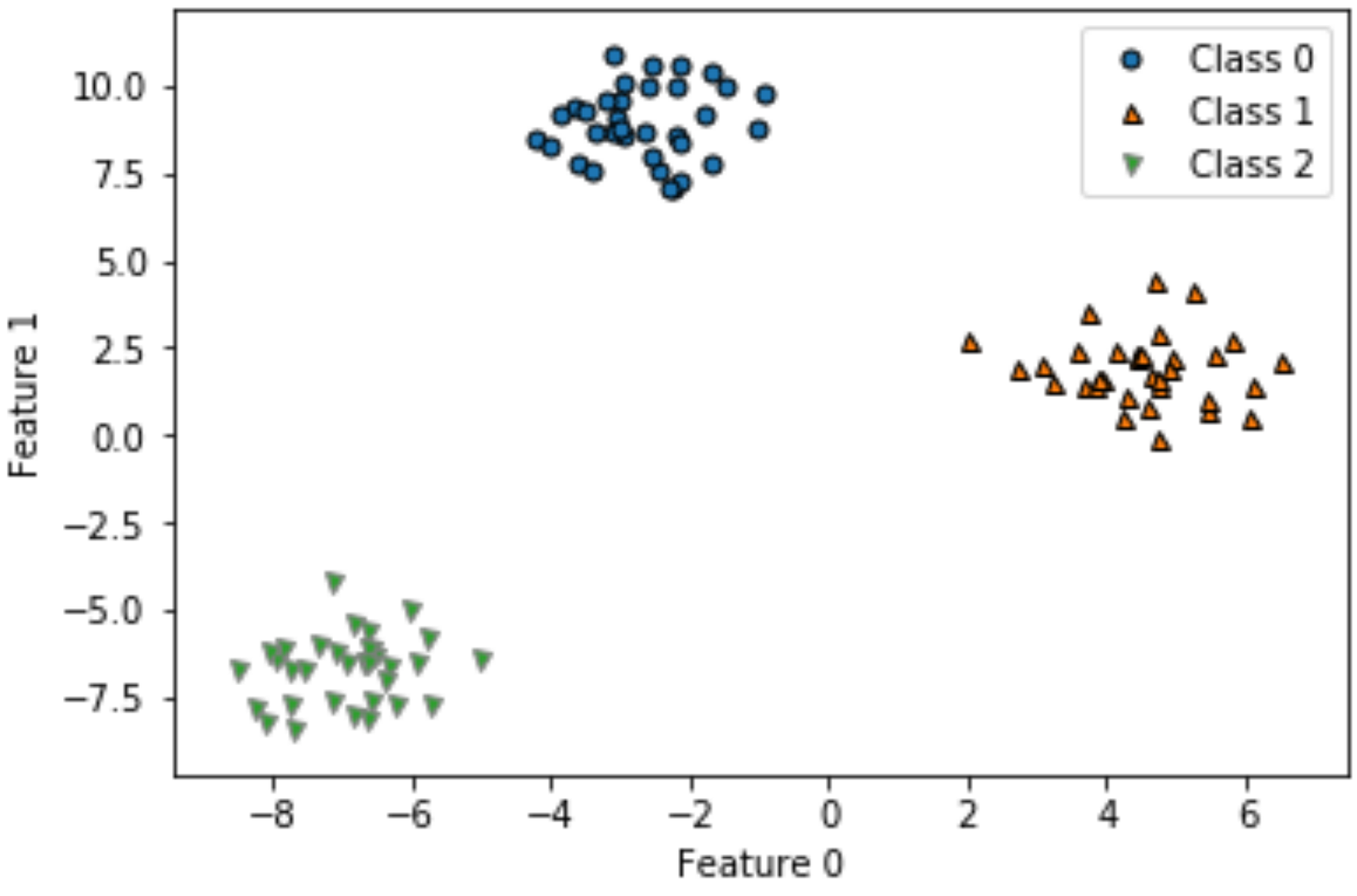
- Les mathématiques de la classification multi-classes par régression logistique sont quelque peu différentes de celle de l'approche one-vs.-rest mais elles aboutissent également à un vecteur de coefficients et un terme constant par classe, et la méthode de prévision est la même.

Modèles linéaires

- Appliquons la méthode one-vs.-rest à une problème de classification à trois classes.

- On utilise un jeu de données bi-dimensionnel où les données de chaque classe suivent une loi gaussienne bivariée.

Modèles linéaires



Modèles linéaires

- On lance un classificateur LinearSVC sur les données :

```
linear_svm = LinearSVC().fit(X,y)

print("Coefficient shape: ",linear_svm.coef_.shape)

print("Intercept shape: ",linear_svm.intercept_.shape)
```

- Résultat :

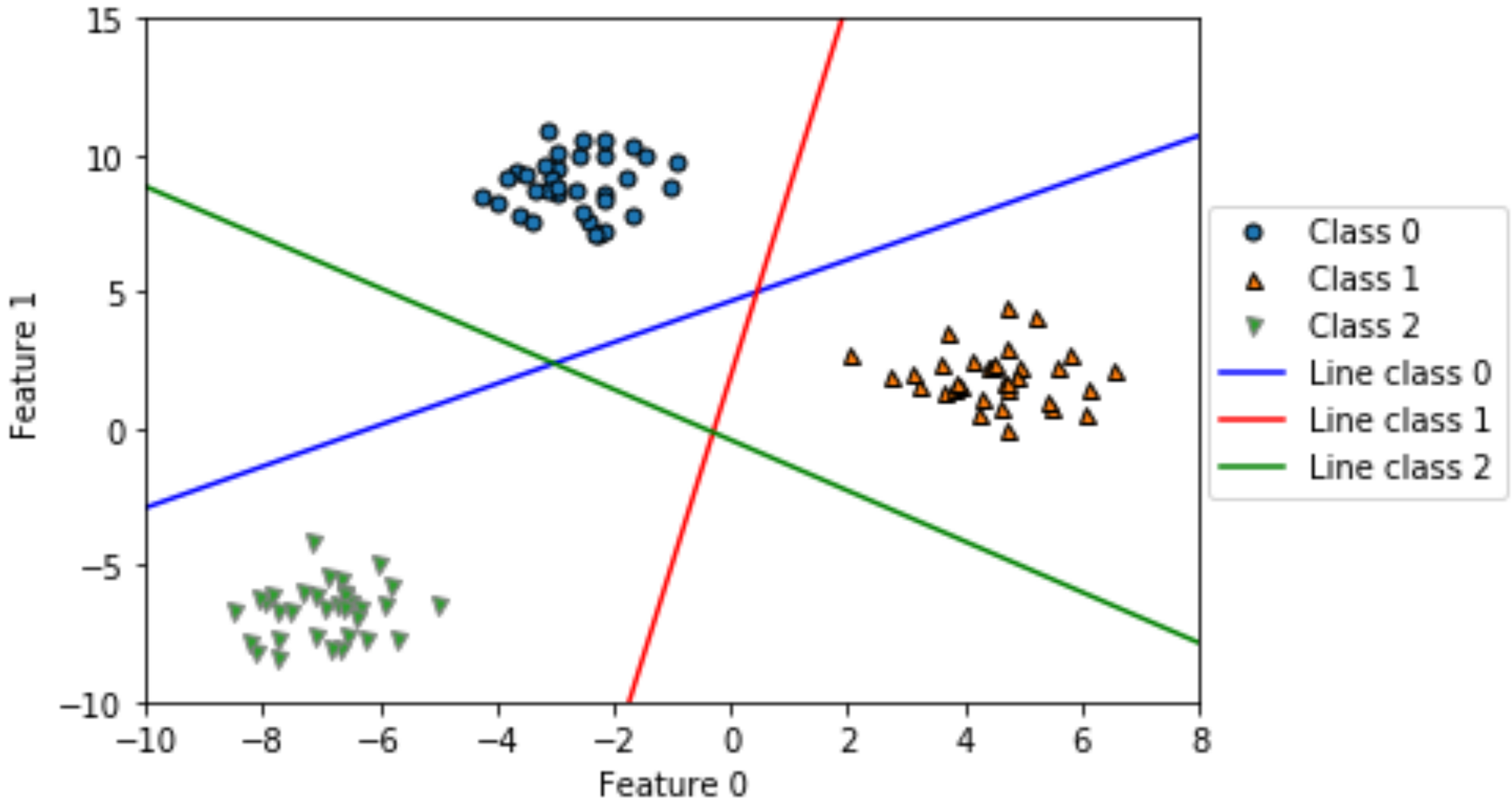
```
Coefficient shape:  (3, 2)

Intercept shape:  (3,)
```

Modèles linéaires

- On voit que coef_ est de dimensions (3,2), ce qui signifie que chaque ligne de coef_ contient le vecteur de coefficients pour l'une des trois classes. Intercept_, quant à lui, est un vecteur de dimension 3, une pour chaque classe.

- A présent, visualisons les frontières de décision pour chacun des trois classificateurs (programme *linear_classif.py*):

Modèles linéaires



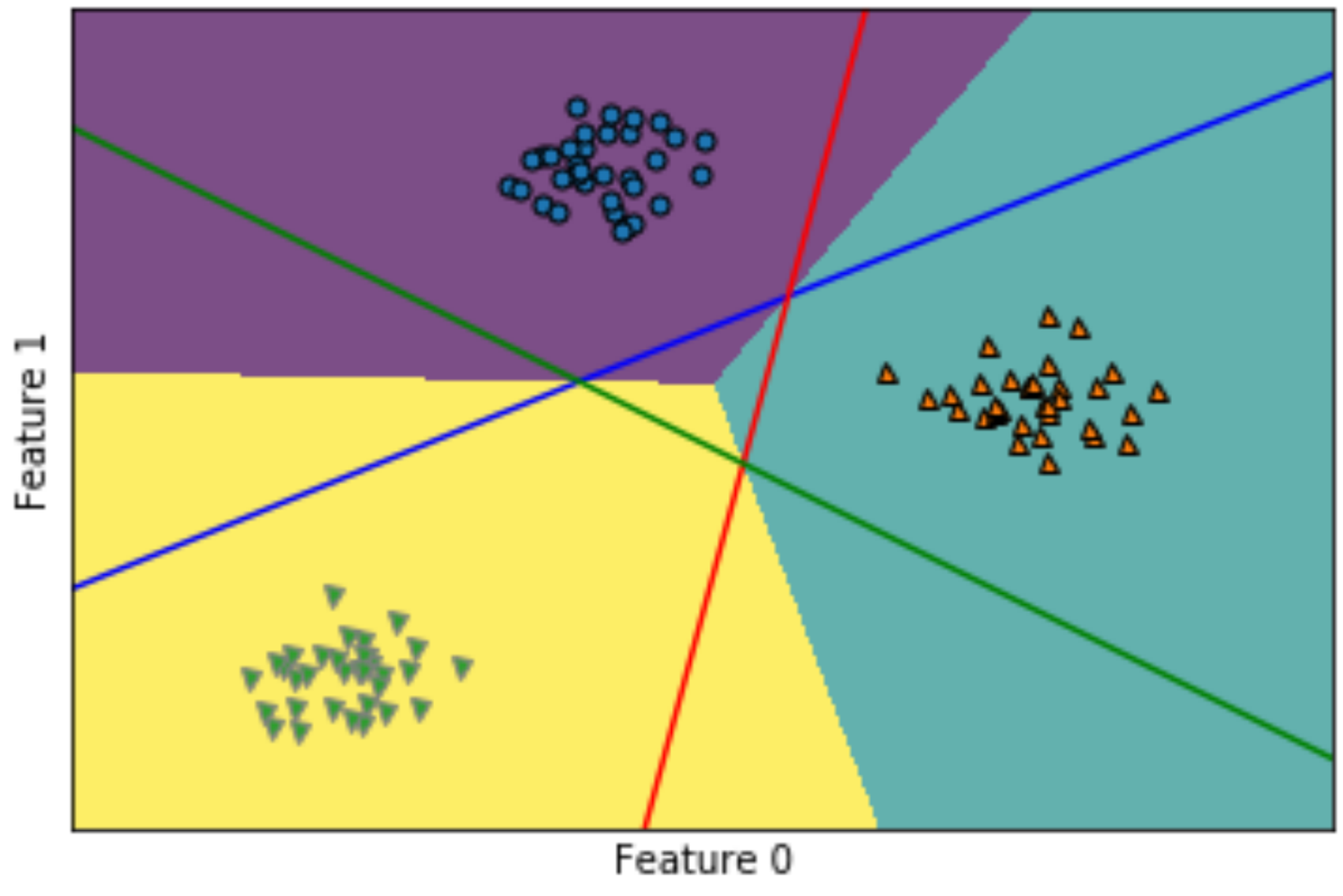
Modèles linéaires

- Qu'en est-il du triangle au milieu du graphique ?

- Les trois classificateurs classent la zone correspondante en « rest ». A quelle classe sont assignés d'éventuels points dans cette zone ? Réponse : à celle pour laquelle la formule de classification donne la valeur la plus élevée, c'est-à-dire à la classe de la droite la plus proche.

- Visualisons les prévisions pour toutes les zones du plan (programme *linear_classif.py*) :

Modèles linéaires



Modèles linéaires

- Le principal paramètre des modèles linéaires est le paramètre de régularisation, appelé alpha dans les modèles de régression et C pour LinearSVC et LogisticRegression.

- Des valeurs élevées de alpha, ou des valeurs faibles de C, signifient que le modèle est plus simple.

- Pour les modèles de régression, en particulier, l'ajustement de ces paramètres est très important.

- *Usuellement, on cherche C et alpha sur une échelle logarithmique.*

Modèles linéaires

- L’autre décision à prendre est de choisir entre une régularisation L1 ou une régularisation L2.
  - *Si l’on suppose que seules quelques variables sont pertinentes pour la prévision, on choisit la régularisation L1. Sinon, on choisit la régularisation L2.*

- La régularisation L1 peut également être précieuse si l’interprétabilité du modèle est importante puisqu’elle sélectionne un faible nombre de variables pertinentes.

Modèles linéaires

- *Les modèles linéaires apprennent rapidement et prévoient rapidement.*

- Il passent à l'échelle pour des jeux de données très volumineux et fonctionnent bien avec les données sparse.

- Si les données sont composées de centaines de milliers ou de millions de lignes, on peut utiliser l'option solver = sag dans LogisticRegression et Ridge, ce qui peut permettre d'obtenir les résultats plus rapidement.

- Une autre solution pour le passage à l'échelle consiste à utiliser les classes SGDClassifier ou SGDRegressor.

Modèles linéaires

- *Un autre avantage des modèles linéaires est qu'il est relativement facile de comprendre la façon dont une prévision est obtenue.*

- Toutefois, la signification des coefficients n'est pas toujours claire.

- En particulier, si les variables explicatives sont très corrélées, les coefficients peuvent être difficiles à interpréter.

Modèles linéaires

- *Les modèles linéaires se comportent généralement bien lorsque le nombre de variables est important par rapport au nombre d'échantillons (d'individus).*

- Ils sont également utilisés sur des données très volumineuses, pour la simple raison qu'il n'est pas possible d'utiliser d'autres modèles.

- Toutefois, en faible dimension, d'autres modèles peuvent être plus performants en termes de généralisation.

Classificateur

Bayésien Naïf

Naive Bayes

- Les classificateurs bayésiens naïfs sont assez semblables aux classificateurs linéaires, mais ils ont tendance à être plus rapides en termes d'apprentissage.

- Par contre, ils ont en général des performances en généralisation inférieures à celles des classificateurs linéaires comme LogisticRegression et LinearSVC.

Naive Bayes

- La raison pour laquelle les modèles bayésiens naïfs sont si efficaces en termes d'apprentissage est qu'ils apprennent en considérant chaque variable séparément et en calculant des statistique simples par classe pour chaque variable.

- Il y a trois sortes de classificateurs bayésiens naïfs implémentés dans scikit-learn : GaussianNB, BernoulliNB et MultinomialNB.

Naive Bayes

- *GaussianNB peut être appliqué à des données continues, BernoulliNB à des données binaires et MultinomialNB à des données de comptage* (i.e. chaque variable prend des valeurs entières représentant un nombre de choses, par exemple, le nombre de fois qu'un mot apparaît dans une phrase).

- *BernoulliNB et MultinomialNB sont surtout utilisées en classification de données textuelles.*

Naive Bayes

- Le classificateur BernoulliNB compte le nombre de fois où les variables sont différentes de zéro sur chaque classe. Afin d'illustrer son fonctionnement, considérons un exemple :

```
X = np.array([[0,1,0,1],

              [1,0,1,1],

              [0,0,0,1],

              [1,0,1,0]])

y = np.array([0,1,0,1])
```

Naive Bayes

- Ici, nous avons quatre points de données, chacun composé de quatre valeurs binaires.

- Il y a deux classes : 0 et 1.

- Pour la classe 0 (1er et 3ème points), le première variable vaut 0 deux fois et est différente de 0 zéro fois; la deuxième variable vaut zéro une fois et est différente de zéro une fois, etc. On calcule ensuite les mêmes statistiques sur la classe 1. En termes de programme cela donne (programme *naive_bayes.py*) :

Naive Bayes

```
counts={}

for label in np.unique(y):

    counts[label]=X[y==label].sum(axis=0)

print("Feature counts:\n{}".format(counts))
```

- Résultat :

```
Feature counts:

{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Naive Bayes

- Les deux autres modèles bayésiens naïfs, MultinomialNB et GaussianNB, sont légèrement différents en termes de statistiques calculées.

- MultinomialNB prend en compte la valeur moyenne de chaque variable sur chaque classe, alors qua GaussianNB calcule la valeur moyenne ***et*** l'écart-type de chaque variable sur chaque classe.

Naive Bayes

- Afin de faire une prévision, un nouveau point de données est comparé aux statistiques calculées pour chaque classe et la classe retenue en prévision est celle qui lui correspond le mieux.

- Pour GaussianNB comme pour MultinomialNB, cela conduit à une formule de prévision linéaire.

- Toutefois, l'attribut coef_ du modèle bayésien naïf a un sens différent de celui qu'il a pour un modèle linéaire : *coef_ est différent de w*.

Naive Bayes

- MultinomialNB et BernoulliNB admettent un paramètre unique alpha, qui contrôle la complexité du modèle.

- Un alpha plus élevé correspond à modèle moins complexe.

- La performance de l'algorithme est relativement peu sensible au choix de alpha, mais un bon ajustement de celui-ci peu améliorer un peu les performances.

Naive Bayes

- GaussianNB est surtout utilisé pour des données en très grande dimension, alors que les deux autres variantes sont souvent utilisées pour des données textuelles.

- MultinomialNB a souvent de meilleurs performances que BinaryNB sur de grands documents.

- Le modèle bayésien naïf partage de nombreux avantages et inconvénients avec les modèles linéaires. Il apprend et prévoit très rapidement et la procédure d'apprentissage est facile à comprendre. Il est particulièrement adapté aux données en très grande dimension et aux très grands ensembles de données.

Autres méthodes

- D'autres méthodes d'apprentissage supervisé sont : les arbres de décision, les forêts aléatoires, les gradient boosting machines, les kernelized support vector machines, les réseaux de neurones artificiels (deep learning).

- Ces méthodes sont enseignées en Master 1 et Master 2 Data Science & Modélisation Statistique à l'UBS.

- Sous Python, elles s'utilisent comme les méthodes qu'on a vues précédemment.

Introduction à l'Apprentissage par renforcement

Renforcement

- L'apprentissage par renforcement se situe entre l'apprentissage supervisé, où on fournit à l'algorithme les réponses correctes pour une base d'exemples, et l'apprentissage non supervisé, où l'algorithme doit extraire des connaissances des données sans avoir de réponses correctes.

- *Dans l'apprentissage par renforcement, on indique simplement à l'algorithme la qualité de ses réponses, sans lui dire comment les améliorer.*

Renforcement

- L’algorithme doit essayer différentes stratégies et déterminer celle qui fonctionne le mieux.

- L’algorithme effectue une recherche dans l’espace d’états des entrées et sorties possibles afin de maximiser une gratification (une récompense).

Renforcement

- L’apprentissage par renforcement est généralement décrit au moyen d’une interaction entre un agent et sont environnement.

- L’agent est la chose qui apprend et l’environnement est composé des conditions dans lesquelles a lieu l’apprentissage et de ce qui est appris.

- L’environnement fournit également de l’information sur la qualité d’une réponse via la fonction de gratification.

Renforcement

- Comme exemple, considérons un enfant qui apprend à se tenir debout et à marcher.

- Il essaye différentes stratégies pour se tenir debout et il reçoit un feedback sur celles qui marche en tombant ou en ne tombant pas.

- Les méthodes qui semblent fonctionner sont essayées encore et encore, jusqu'à qu'elles soient perfectionnées ou qu'il trouve de meilleures solutions, et celles qui ne fonctionnent pas sont oubliées.

Formalisation

- *L’apprentissage par renforcement associe des actions à des états ou des situations en vue de maximiser une gratification numérique.*

- L’algorithme connait son input courant (l’état) et les choses qu’il peut faire (les actions) et son but est de maximiser la gratification.

Formalisation

- La façon la plus fréquente de penser à l'apprentissage par renforcement consiste à faire intervenir un robot.

- L'état est défini par les données des capteurs du robot ou une version transformée de celles-ci. Celles-ci constituent une sorte de représentation de l'environnement du robot.

- Les capteurs ne fournissent pas nécessairement toute l'information utile (par exemple, il ne fournisse pas la localisation du robot) et il peut y avoir des inexactitudes et du bruit sur les données des capteurs.

Formalisation

- Les diverses façons possibles dont le robot peut actionner ses moteurs constituent les actions, qui déplacent le robot dans son environnement, et la gratification peut consister en une mesure de qualité de la façon dont il exécute sa tâche sans percuter d'objets de l'environnement.

- *En apprentissage par renforcement, la fonction de gratification évalue la solution courante mais ne suggère pas de façon de l'améliorer, contrairement à ce qui se passe en apprentissage supervisé.*

Formalisation

- Une autre difficulté est qu'il peut y avoir un délai dans la gratification, ce qui signifie que celle-ci n'est accordée que bien après qu'une action ait été effectuée.

- Ainsi, on doit parfois distinguer la gratification immédiate et la gratification espérée dans le futur.

- Une fois que l'algorithme sait comment évaluer les actions, il doit choisir celle à effectuer dans l'état courant.

Formalisation

- C’est fait par une combinaison d’exploration (essayer d’obtenir une nouvelle solution) et d’exploitation (essayer d’améliorer une solution existante).

- Concrètement, cela peut signifier : reproduire l’action qui a fourni la plus grande gratification la dernière fois que le robot a été dans le même état ou essayer une action différente dans l’espoir d’améliorer la gratification.

Applications

- L’apprentissage par renforcement est utilisé notamment en robotique car il permet aux robots d’apprendre à effectuer des tâche sans intervention humaine.

- Par exemple, il a été utilisé pour faire en sorte que des robots libèrent de l’espace dans une pièce en poussant des boîtes sur les côtés, qu’ils se suivent, qu’ils se déplacent vers des repères lumineux et qu’ils naviguent.

Applications

- Un inconvénient de l'apprentissage par renforcement est qu'il est en général assez lent car il doit recueillir de l'information via l'exploration et l'exploitation des solutions pour améliorer celles-ci.

- *Il dépend aussi fortement de la fonction de gratification choisie. Si ce choix est mal fait, l'algorithme peut faire n'importe quoi.*

- Dans la suite, on va voir une famille d'algorithmes permettant de faire de l'apprentissage par renforcement : les algorithmes génétiques.

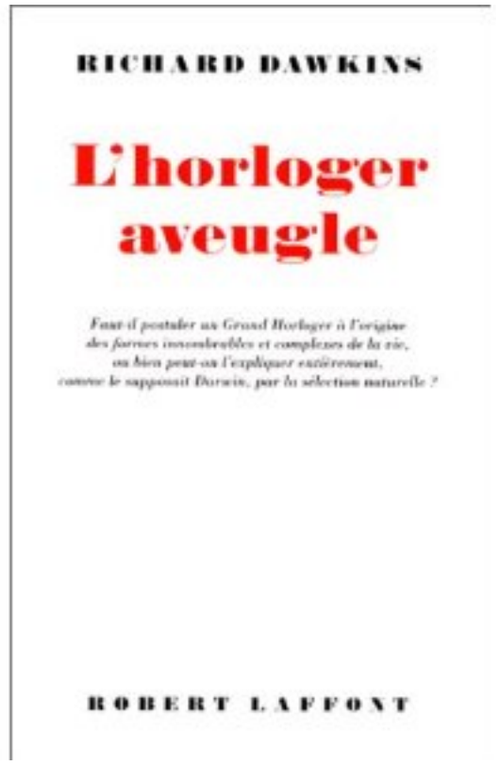
Algorithmes génétiques

Algorithmes génétiques

- Le mécanisme de l'évolution agit sur une population animale par l'intermédiaire d'une hypothétique fonction d'aptitude possédant un biais positif vis-à-vis des animaux les plus aptes, c'est-à-dire ceux qui vivent suffisamment longtemps pour se reproduire et qui sont plus attractifs; et qui on donc plus de partenaires et une descendance plus nombreuse.

- Ce mécanisme est notamment décrit dans les deux ouvrages suivants :

Algorithmes génétiques



Algorithmes génétiques

- Les algorithmes génétiques modélisent le mécanisme génétique de l'évolution. En particulier, ils modélisent la reproduction sexuée, dans laquelle les deux parents transmettent de l'information génétique à leurs descendants.

- Ainsi, les enfants ont des similarités avec leurs parents et il ya beaucoup d'héritage génétique. Toutefois, il y a également des mutations aléatoires, dues à des erreurs de copie lorsque les chromosomes sont reproduits, ce qui signifient que les choses évoluent au cours du temps.

Algorithmes génétiques

- *Les algorithmes génétiques sont souvent, mais pas toujours, très efficaces.*

- Il dépendent d'un certain nombre de paramètres critiques qu'il est difficile d'ajuster.

- On ne peut garantir que la solution trouvée soit un tant soit peu pertinente.

- En pratique, il donnent souvent satisfaction et sont notamment utilisés lorsqu'on n'a pas d'alternative classique pour mener à bien une optimisation.

Algorithmes génétiques

- En se basant sur les principes de l'évolution, on on a besoin, pour modéliser des mécanismes génétiques simples sur ordinateur et les utiliser pour résoudre un problème d'optimisation, de déterminer :

- Une méthode pour représenter les solutions comme des chromosomes

- Une méthode pour calculer la qualité d'une solution

- Une méthode de sélection pour choisir les parents

- Une méthode pour générer une descendance en croisant les parents

Algorithmes génétiques

- La démarche proposée sera illustrée sur une exemple classique, qui est difficile à résoudre (NP-complet) : le problème du sac-à-dos. Celui-ci est le suivant :

- *Vous devez partir en vacances. Vous avez acheté le plus grand et le meilleur sac à dos en vente, mais il ne suffit toujours pas à contenir tout ce que vous devez emporter. Vous décidez donc d'associer une valeur à chaque objet et de mesurer l'espace qu'il occupe. Vous souhaitez maximiser la valeur totale des objets que vous emporterez sous la contrainte qu'ils rentrent tous dans le sac.*

Algorithmes génétiques

- Ce problème, ou des variations de celui-ci, apparaît en cryptographie, combinatoire, logistique et management; c'est donc un problème important du point de vue pratique. Malheureusement, il est NP-complet ce qui signifie que la détermination d'une solution optimale pour des cas intéressants est impossible du point de vue calculatoire.

- *Pour simplifier le problème, on supposera dans la suite que la valeur est égale au volume.*

- On va voir comment déterminer une approximation de la solution optimale à l'aide d'un algorithme génétique.

Représentations symboliques

- La première chose dont nous ayons besoin est une façon de représenter les solutions possibles par l'analogue de chromosomes.

- Les AG (Algorithmes génétiques) utilisent pour ce faire des chaînes de caractères, chaque élément des ces chaînes (équivalent à un gène) étant choisis dans un alphabet.

- Les différents caractères de l'alphabet sont l'analogue des allèles et sont souvent binaires (0/1).

- On génère alors aléatoirement un ensemble de chaînes aléatoires, qui constitue notre population initiale.

Représentations symboliques

- Pour le problème du sac-à-dos, l'alphabet est très simple : on peut le prendre binaire. On adopte une longueur de chaîne égale à L, où L est le nombre total d'objet qu'on souhaite pouvoir emmener, chaque élément de la chaîne pouvant être égal à 0 ou 1. On code alors 0 pour les objets qu'on emmène pas et 1 pour ceux qu'on emmène. Ainsi, la chaîne obtenue correspond à une solution possible au problème.

Représentations symboliques

- Cependant, cela ne nous dit pas si cette solution est réalisable (c’est-à-dire si elle rentre dans le sac) ni si c'est une bonne solution (si elle maximise le volume total des objets emmenés).

- Pour ce faire, on a besoin d'une méthode permettant de déterminer dans quelle mesure chaque chaîne remplit les critères du problème, c'est-à-dire d’évaluer la qualité de la solution.

Qualité des solutions

- La fonction de qualité, d'aptitude ou de pertinence peut être vue comme un oracle qui prend une chaîne pour argument et lui associe une valeur.

- C'est la seule partie de l'algorithme spécifique au problème.

- Il faut bien réfléchir à ce qu'on exige de la fonction de qualité; *la meilleure chaîne doit être de qualité la plus élevée et la qualité doit décroître pour des chaînes qui font moins bien sur le problème posé*.

- En général, la qualité doit être une fonction positive : même les chaînes les moins pertinente doivent avoir une qualité supérieure ou égale à zéro.

Qualité des solutions

- Dans l'évolution réelle, la fonction de qualité n'est pas statique : il y a compétition entre les différentes espèces (prédateurs et proies par exemple) et des traitement peuvent être découverts pour certaines maladies; la mesure de qualité change donc au cours du temps. Dans le cadre des algorithmes génétiques, on fait abstraction de ces aspects.

- Pour le problème du sac-à-dos, on peut souhaiter que le sac soit de valeur la plus élevée possible. On associe dans ce cas à chaque chaîne la valeur totale correspondante. Cela correspond à une fonction de qualité possible.

Qualité des solutions

- Toutefois, cela ne nous dit pas si la solution rentre dans le sac : avec cette fonction de qualité, la solution optimale est de tout prendre.

- On doit donc vérifier qu'elle rentre dans le sac et, si ce n'est pas le cas, réduire la qualité de la solution.

Qualité des solutions

- Une approche consiste à fixer la qualité d'une solution à 0 si elle ne rentre pas dans le sac. Toutefois, supposons qu'un solution soit presque parfaite, en cela qu'il y a juste un objet en trop.

- En fixant la qualité à 0, on réduit les chances que cette solution soit autorisée à évoluer et donc à s'améliorer au cours des itérations suivantes.

Qualité des solutions

- On va donc, au lieu de cela, adopter comme fonction de qualité la somme des volumes des objets s'ils rentrent dans le sac mais s'il n'y rentrent pas, on la définira comme la taille du sac moins deux fois le volume excédentaire.

- Ceci autorise les solutions qui sont légèrement excédentaires à être prises en considération pour amélioration, tout en traduisant le fait que ce ne sont pas les plus adaptées.

Qualité des solutions

- Il existe un algorithme évident pour déterminer des solutions au problème du sac-à-dos. Il consiste à prendre à chaque étape l'objet le plus volumineux qui n'ait pas encore été sélectionné et qui rentre encore dans le sac, et à itérer. Il ne fournit pas nécessairement la solution optimale, mais il est très simple et très rapide.

- On attend donc d'un AG qu'il fournisse une bien meilleure solution que cet algorithme pour justifier l'effort de programmation et d’exécution associé.

Population

- On est à présent en mesure d'évaluer la qualité de n'importe quelle chaîne.

- L'Algorithme Génétique agit sur une population de chaînes qu'il fait évoluer au cours du temps.

- La première génération est en général générée aléatoirement.

- On évalue ensuite la qualité de chaque chaîne et les élément de la première génération sont croisés (on verra comment dans la suite) de façon à obtenir une deuxième génération, et ainsi de suite.

- L'évolution a lieu de façon à ce que la qualité des individus composant la population s'améliore au fil du temps.

Population

- Pour le problème du sac-à-dos, on commence par créer un ensemble de chaînes aléatoires binaires de longueur L en utilisant un générateur de nombres aléatoires.

- On peut par exemple générer 100 chaînes. On choisit alors des parents dans cette population et on les croise.

- La taille de la population reste constante au cours des itérations, ce qui diffère de l'évolution animale telle qu'elle a lieu en réalité.

Sélection des parents

- L'idée de base est que la qualité s'améliorera si on croise des chaînes qui sont déjà de bonne qualité comparées aux autres (on imite la sélection naturelle).

- Ce faisant, on dit qu'on effectue une exploitation de la population courante. En général, on s'autorise toutefois également à faire une exploration de celle-ci en n'écartant pas la possibilité de sélectionner des chaînes de qualité moindre.

Sélection des parents

- In fine, on sélectionne donc les chaînes en fonction de leur qualité, de façon à ce que les chaînes les plus aptes aient plus de chances d'être sélectionnées pour la reproduction.

- Pour ce faire, on utilise en général l'une des deux méthodes suivantes :

Sélection par troncation

- C'est une méthode simple consistant à sélectionner une proportion f des meilleurs chaînes et à ignorer les autres. Par exemple, on peut fixer f=0.5, ce qui signifie que les 50% de chaînes les meilleures sont sélectionnées pour reproduction et choisies avec une probabilité égale.

- Cette méthode est facile à implémenter mais limite le degré d'exploration effectué, ce qui a tendance à biaiser l'AG en faveur d'une exploitation de la population.

Sélection proportionnelle

- Elle est considérée comme la meilleure approche en général.
- Elle consiste à sélectionner les chaînes de façon probabiliste, la probabilité qu'une chaîne soit sélectionnée étant proportionnelle à sa qualité.
- Pour une chaîne alpha, on retient en général la formule:

```
p_alpha=F_alpha/SOMME(F_beta:beta)
```

- où p_alpha désigne la probabilité que la chaîne alpha soit sélectionnée et F_alpha sa qualité.

Sélection proportionnelle

- On comprend donc la condition de positivité imposée à la fonction de qualité.

- Si on souhaite autoriser celle-ci à prendre des valeurs négatives, on utilise plutôt la formule de Boltzmann :

```
p_alpha=exp(sF_alpha)/SOMME(exp(sF_beta):beta)
```

- où s est un paramètre appelé intensité de sélection.

Sélection proportionnelle

- Se pose toutefois un problème algorithmique : *comment échantillonner la population avec des probabilités inégales ?*

- En effet, on dispose en général de fonctions permettant d'échantillonner avec des probabilités égales (avec ou sans remise); par example choice sous Python.

- L'astuce consiste à construire une population virtuelle où chaque chaîne est répétée un nombre de fois proportionnel à p_alpha (donc à sa qualité si la première formule est utilisée).

- On effectue alors un échantillonnage simple dans cette population virtuelle.

Sélection proportionnelle

- Quelle que soit la façon dont on sélectionne les chaînes, l'étape suivante consiste à les mettre en paires.

- Comme l'ordre dans lequel elles sont échantillonnées est aléatoire, on peut simplement associer à chaque chaîne d'indice pair la chaîne suivante d’indice impair pour reproduction.

Opérateurs génétiques

- Ayant sélectionné les paires d'individus pour reproduction, il reste à déterminer comment croiser les chaînes de façon à générer leur descendance, ce qui constitue la partie génétique de l'algorithme.

- On utilise en général deux opérateurs génétiques que l'on va présenter dans la suite.

- Ce sont les opérateurs initialement proposés lors de l'invention des algorithmes génétiques et ce sont encore aujourd'hui, de loin, les plus utilisés.

Opérateur de croisement

- En biologie, les organismes ont deux chromosomes et chaque parent transmet l'un d'entre eux.

- Dans le cadre des AG, les membres de la population ont seulement l'équivalent d'un chromosome : la chaîne.

- *On génère donc une nouvelle chaîne en prenant une partie de la première et une partie de la deuxième.*

- La façon la plus usuelle de le faire consiste à choisir au hasard un point dans la chaîne et à utiliser la parent 1 pour la première partie de la chaîne et le parent 2 pour la deuxième.

Opérateur de croisement

- On génère ensuite un deuxième descendant, composé de la première partie du parent 2 et de la deuxième partie du parent 1.

- Cette méthode est appelée méthode de croisement basée sur un point; la généralisation à plusieurs points est immédiate.

- Une version plus extrême est appelée croisement uniforme et consiste à sélectionner aléatoirement l'élément courant de la chaîne du descendant parmi les éléments correspondants des deux parents.

Opérateur de croisement

- L'opérateur de croisement sert à effectuer une exploration globale puisque les chaînes produites sont radicalement différentes de leurs parents.

- On espère sélectionner les bonnes parties de chacune des solutions de façon à obtenir une solution encore meilleure.

Opérateur de mutation

- L'exploitation des chaînes courantes les plus aptes est effectuée par l'opérateur de mutation qui implémente une recherche locale aléatoire.

- *La valeur de chaque élément de la chaîne est modifiée avec une probabilité (généralement faible) p.*

- Sur l'exemple du sac-à-dos, la mutation consiste à inverser un bit.

- Souvent, on prend p=1/L où L désigne la longueur de la chaîne, de façon à avoir en moyenne une mutation par chaîne. Ce choix est validé par la pratique.

Elitisme, tournois et niches

- Un problème de l'Algorithme Génétique standard est que la meilleure qualité sur la population peut croître comme décroître à la génération suivante.

- C'est dû au fait que les meilleurs chaînes de la génération courante ne sont pas recopiées à la génération suivante et que parfois aucun de leurs descendants n'est aussi apte.

Elitisme, tournois et niches

- Une façon d'éviter cela est appelée élitisme : cela consiste à copier les meilleurs chaînes dans la population suivante sans modification.

- Une autre solution est d'implémenter un tournoi ou les deux parents et leurs deux descendants entrent en compétition, les deux meilleurs parmi les quatre étant intégrés à la nouvelle population.

Elitisme, tournois et niches

- Bien qu'élitisme et tournois permettent de garantir que les bonnes solutions ne soient pas perdues, ils peuvent être à la source d'un problème de convergence prématurée, l'algorithme se stabilisant sur une population ne contenant pas l'optimum.

- Cela est dû au fait que, dans ce cas, l'AG favorise les membres les plus aptes de la population, ce qui signifie que les optima locaux sont favorisés et exploités.

- L'exploration n'est alors pas suffisante pour échapper à ces maxima locaux.

- En fait, élitisme et tournois on pour effet de réduire la diversité de la population en autorisant les mêmes individus à subsister sur plusieurs générations.

Elitisme, tournois et niches

- Une solution au problème est la création de niches (également appelées populations insulaires) : la population est séparée en plusieurs sous-populations qui évoluent indépendamment pendant un certain temps et qui convergent donc vraisemblablement vers des optima locaux différents.

- Au cours du temps, quelques éléments d'une sous-population sont occasionnellement injectés dans une autre sous-population.

Elitisme, tournois et niches

- Il existe d'autres approches permettant d'améliorer la convergence et les résultats finaux des algorithmes génétiques, mais il s'agit déjà de notions plus avancées.

Récapitulation

- L'Algorithme Génétique le plus simple est obtenu en combinant les différents éléments que nous avons détaillés plus haut.

- Initialisation : Générer N chaînes aléatoires de longueur L dans l'alphabet choisi

- Apprentissage : Répéter

  - Créer une nouvelle population (initialement vide)

  - Répéter :

    - Sélectionner deux chaînes dans la population courante en se basant sur la qualité

Récapitulation

- Les croiser pour produire deux nouvelles chaînes
- Faire muter les deux nouvelles chaînes
- Ajouter les deux nouvelles chaînes à la population où utiliser élitisme ou tournois
    - Garder trace de la meilleure chaîne dans la population … jusqu'à génération de N nouvelles chaînes
- Remplacer la population courante par la nouvelle population

- … jusqu'à ce qu'un critère d'arrêt soit vérifié.

Implémentation sous Python

- Un algorithme génétique est implémenté sous Python pour le problème du sac à dos dans les programmes *knapsack.py*, *ga.py* et *run_ga.py*, disponibles sur le forum et dûs à Stephen Marsland.

- L'implémentation retenue fait intervenir de la programmation objet sous Python.

Apprentissage non supervisé

Apprentissage non supervisé

- L'apprentissage non supervisé inclut tous les types de machine learning où il n'y a pas de réponse.

- En apprentissage non supervisé, on montre les inputs à l'algorithme d'apprentissage qui en extrait des connaissances.

Apprentissage non supervisé

- On abordera dans la suite deux types d'apprentissage non supervisé : les transformations des données et le clustering.

- Les transformations non supervisées des données sont des algorithmes qui créent une nouvelle représentation des données, plus facile à comprendre pour les êtres humains ou pour d'autres algorithmes de machine learning.

- Une application usuelle des transformations non supervisées est la réduction de dimension.

Apprentissage non supervisé

- La réduction de dimension consiste à partir d'une représentation des données en grande dimension, avec un grand nombre de variables et à déterminer une représentation qui résume les principales caractéristiques des données avec moins de variables.

- Une application standard de la réduction de dimension est la réduction à deux dimensions en vue de la visualisation.

Apprentissage non supervisé

- Une autre application des transformations non supervisées consiste à déterminer des parties ou composantes en lesquelles se décomposent les données.

- L'exemple typique est l'identification de thèmes sur des collections de documents textes. Le but est d'identifier les thèmes sur lesquels portent les différents textes et ceux qui interviennent dans chaque texte. C'est utile pour suivre l'évolution de l'opinion sur des réseaux sociaux par exemple.

Apprentissage non supervisé

- Le deuxième type d'algorithmes d'apprentissage non supervisé qui sera abordé est celui des algorithmes de clustering.

- Ici, le but est de partitionner les données dans des groupes distincts d'items/d'individus similaires.

- Exemple : on charge des photos sur un réseau social. Afin d'organiser les photos, le site peut essayer de regrouper les photos qui montrent la même personne. Ceci peut être fait à l'aide d'un algorithme de clustering.

Apprentissage non supervisé

- Un des principaux défis de l'apprentissage non supervisé est d'évaluer si l'algorithme a appris quelque chose d'utile.

- Les algorithmes d'apprentissage non supervisé sont usuellement appliqués à des données non labellisées, donc on ne sait pas quelle doit être la sortie correcte.

- Par conséquent, il est très difficile de savoir si un modèle a été performant.

- Sur notre exemple des photos, l'algorithme utilisé peut par exemple regrouper toutes les photos montrant quelqu'un de profil et toutes les photos montrant quelqu'un de face. Cela peut être intéressant mais ce n'est pas ce que l'on souhaite obtenir.

Apprentissage non supervisé

- En fait, il n'y pas moyen de « dire » à l'algorithme ce que l'on souhaite obtenir et, souvent, la seule façon d'évaluer sa performance est d'examiner les résultats manuellement.

- Ainsi, les algorithmes non supervisés sont souvent utilisés dans le cadre d'une démarche exploratoire, c'est-à-dire lorsqu'un data scientist cherche à mieux comprendre les données, plutôt que comme partie de systèmes automatiques d'inférence plus généraux.

Apprentissage non supervisé

- Une autre application usuelle des algorithmes non supervisés est le pré-traitement des données avant d'utiliser un algorithme supervisé.

- Apprendre une nouvelle représentation des données peut parfois améliorer la performance des algorithmes supervisés ou réduire la quantité de mémoire et/ou le temps de traitement requis pour obtenir le résultat.

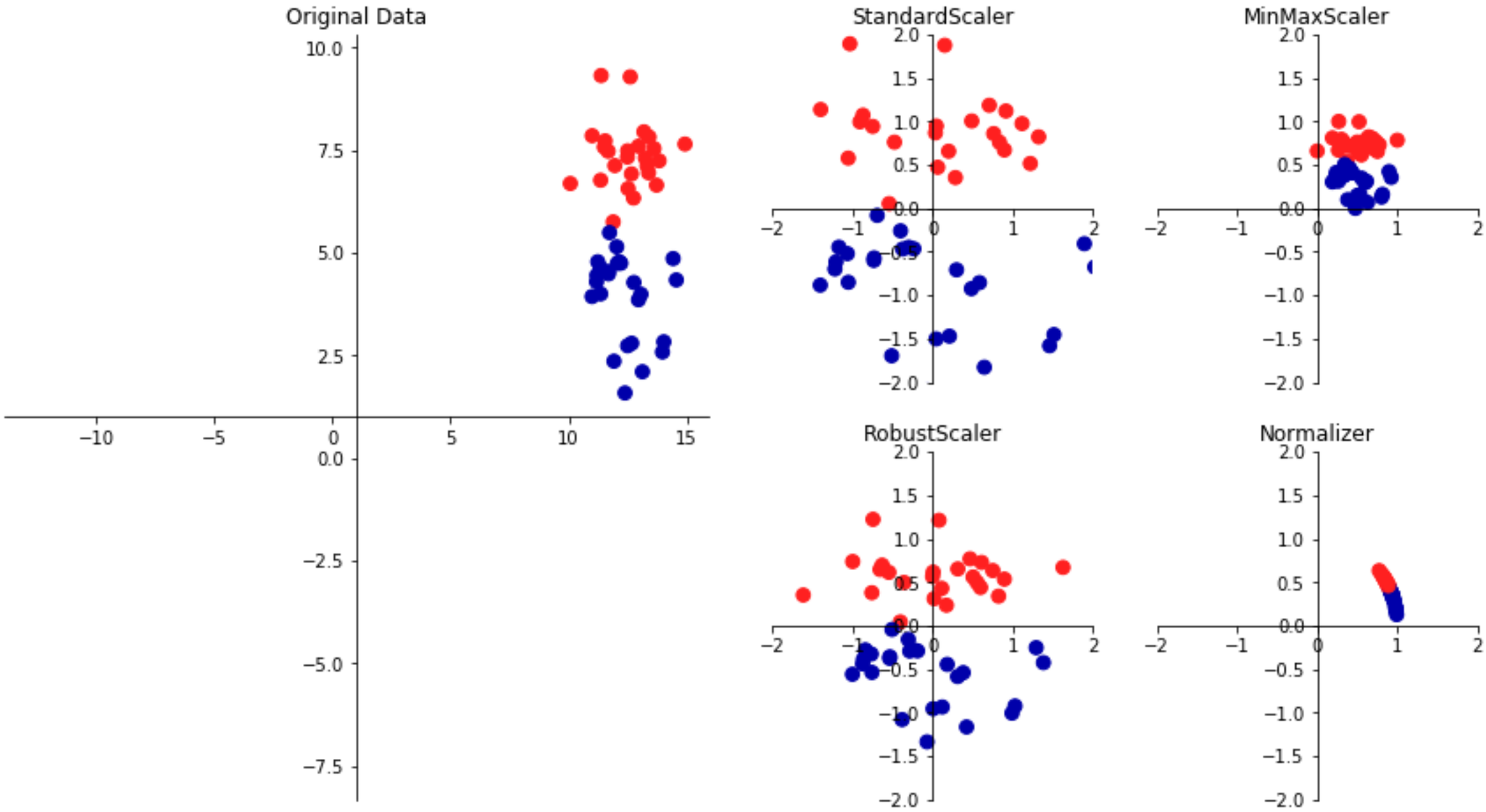
Pré-traitement et mise à l'échelle

- Certains algorithmes d'apprentissage (comme par exemple, les réseaux de neurones et les SVM) sont très sensibles à l'échelle des données.

- Par conséquent, il est standard d'ajuster les variables de façon à obtenir une représentation des données plus adaptée à ces algorithmes.

- Considérons un exemple (programme *echelle.py*).

Pré-traitement et mise à l'échelle



Pré-traitement et mise à l'échelle

- Le premier graphique montre un jeu de données de classification binaire bi-dimensionnel.

- La première variable prend ses valeurs entre 10 et 15, la deuxième entre 1 et 9.

- Les quatre graphiques suivants montrent quatre manières différentes de mettre les données à l'échelle de façon à obtenir des intervalles de variation plus standards.

Pré-traitement et mise à l'échelle

- Le StandardScaler de scikit-learn correspond à une standardisation des données (la moyenne de chaque variable transformée est égale à 0 et son écart-type à 1) et ramène donc les variables à une même échelle.

- Toutefois, cette mise à l'échelle n'impose rien quant à la valeur minimale ou maximale des variables.

Pré-traitement et mise à l'échelle

- Le RobustScaler fonctionne de façon similaire au StandardScaler, en ce qu'il utilise les propriétés statistiques des données pour les ramener à une même échelle.

- Toutefois, le RobustScaler utilise la médiane et les quartiles au lieu de la moyenne et de l'écart-type.

- Cela a pour effet de lui faire ignorer les valeurs aberrantes (ang. outliers), qui peuvent poser problème aux autres méthodes de mise à l'échelle.

Pré-traitement et mise à l'échelle

- Le MinMaxScaler, quant à lui, met les variables à l'échelle de façon ce qu'elles varient entre 0 et 1.

- Dans le cas bi-dimensionnel, cela implique que toutes les données mises à l'échelle sont concentrées dans le carré [0,1]x[0,1].

Pré-traitement et mise à l'échelle

- Enfin, le Normalizer effectue, quant à lui, une mise à l'échelle très différente.

- En effet, il projette chaque vecteur de données sur le cercle unité (ou l'hyper-sphère unité en grandes dimensions).

- Cela implique que chaque vecteur de donnée est mis à l'échelle différemment (en le divisant par sa norme).

- Cette mise à l'échelle n'est utilisée que lorsque ce sont seulement la direction ou les angles entre les vecteurs de données qui intéressent le Data Scientist.

Exemple

- Ayant vu les différentes stratégies possibles de mise à l'échelle, nous allons à présent les appliquer avec scikit-learn.

- Nous allons utiliser le jeux de données cancer et mettre les variables à l'échelles, avec une même mise à l'échelle pour l'ensemble d'apprentissage et l'ensemble test.

- Programme : *echelle_cancer.py*.

Exemple

```
from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import MinMaxScaler


cancer = load_breast_cancer()


X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target, random_state=1)
```

Exemple

```
scaler = MinMaxScaler()

scaler.fit(X_train)

X_train_scaled = scaler.transform(X_train)

X_test_scaled = scaler.transform(X_test)
```

Exemple

- Pour le MinMaxScaler, la méthode fit calcule le minimum et le maximum de chaque variable du tableau de données passé en argument.

- La méthode transform applique la mise à l'échelle correspondante au tableau de données passé en argument.

Effet de la mise à l'échelle

- Afin d'évaluer l'effet que peut avoir une mise à l'échelle sur la performance d'un algorithme, reprenons les données cancer et effectuons la classification avec l'algorithme SVC (Support Vector Classification).

- Programme : *echelle_cancer.py*.

Effet de la mise à l'échelle

```
from sklearn.svm import SVC

svm = SVC(C=100)

svm.fit(X_train,y_train)

print("Test set accuracy: {:.2f}".format(svm.score(X_test,y_test)))
```

Effet de la mise à l’échelle

- Résultat:

```
Test set accuracy: 0.62
```

- A présent, lançons l’algorithme sur les données mises à l’échelle :

Effet de la mise à l'échelle

```
svm.fit(X_train_scaled, y_train)
print("Scaled test set accuracy: {:.
2f}".format(svm.score(X_test_scaled,y_test)))
```

- Résultat :

```
Scaled test set accuracy: 0.97
```

Effet de la mise à l'échelle

- On voit ainsi que l'effet de la mise à l'échelle peut être très important.

- Il ne faut donc pas hésiter à essayer une mise à l'échelle des données avant d'utiliser un ou des algorithmes d'apprentissage.

- On privilégiera à cette fin les méthodes mises à disposition par scikit-learn.

Analyse en Composantes Principales

ACP

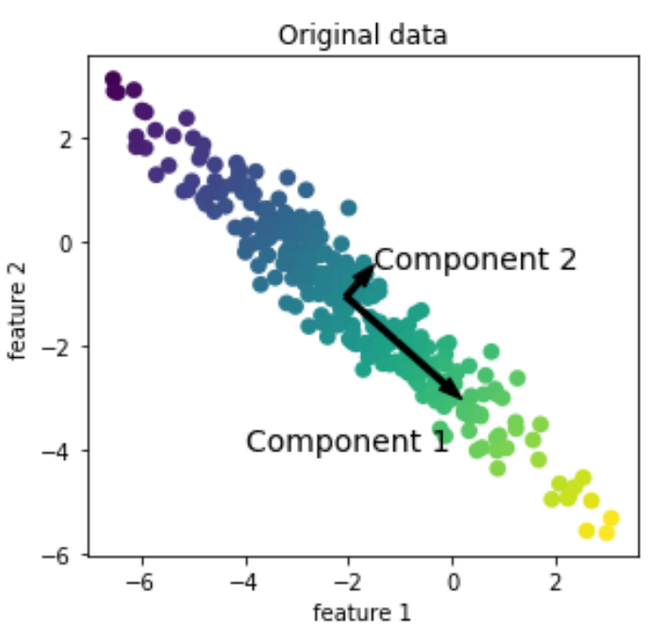
- L’analyse en composantes principales est une méthode qui effectue une rotation des données de façon à ce que les variables résultantes soient statistiquement non corrélées.

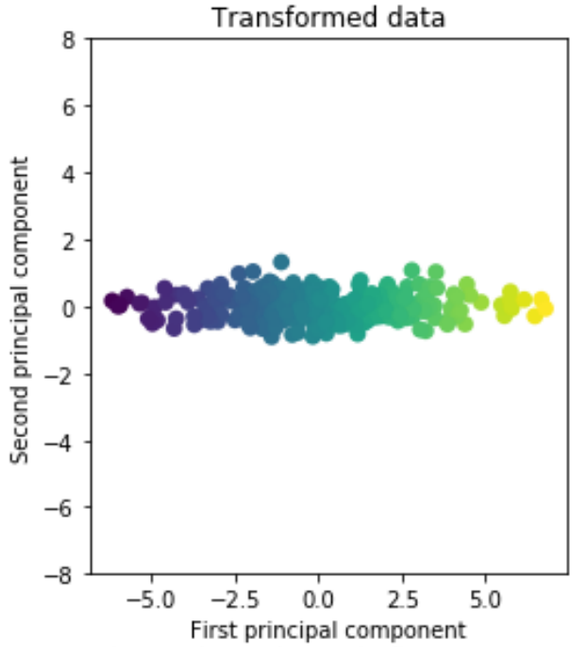
- Cette rotation est souvent suivie de la sélection d’un sous-ensemble des nouvelles variables, selon leur pertinence pour expliquer les données.

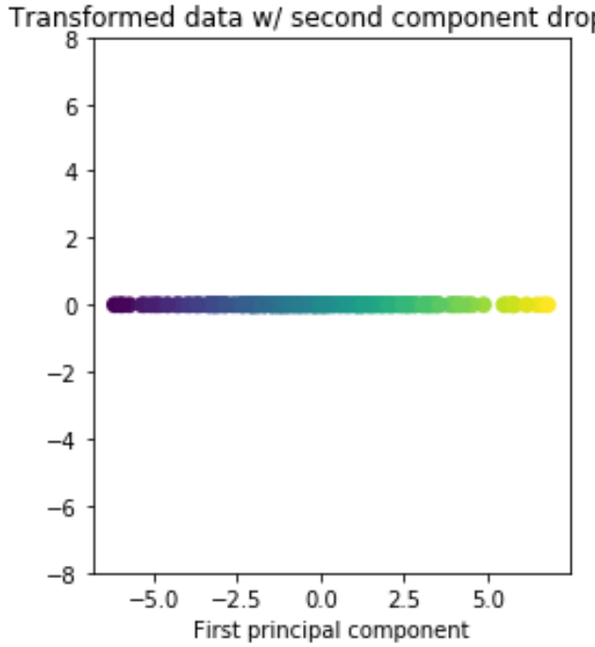
ACP

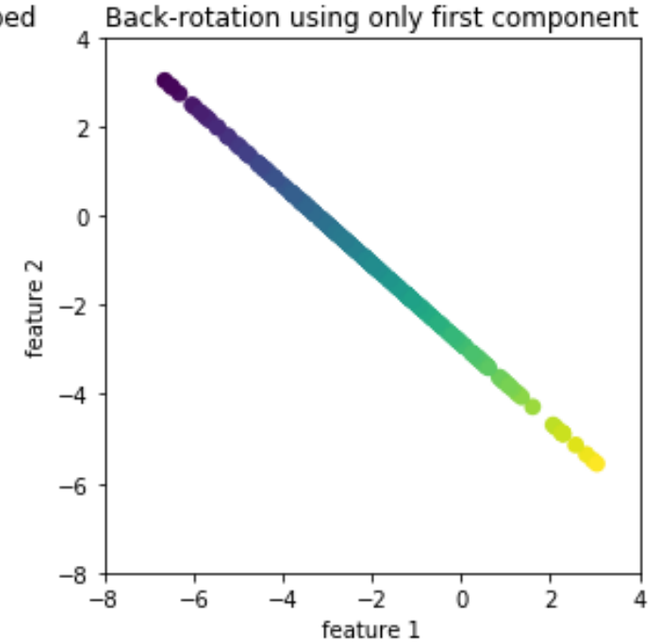
- Le graphique suivant illustre le fonctionnement de l’ACP sur des données simulées (programme *pca_illustration.py*).

ACP


Original data
Component 2
Component 1
feature 2
feature 1


Transformed data
Second principal component
First principal component


Transformed data w/ second component dropped
First principal component


Back-rotation using only first component
feature 2
feature 1

ACP

- Le premier graphique montre le nuage de point initial. Le point sont colorés de façon à pouvoir être distingués.

- L’algorithme commence par déterminer la direction de variance maximale (appelée composante 1).

- C’est la direction qui contient le plus d’information, i.e. celle où les variables initiales sont le plus corrélées les unes avec les autres.

- Ensuite, l’algorithme détermine la direction, orthogonale à la première, qui contient le plus d’information.

ACP

- En deux dimensions, il n’y a qu’une seule direction possible à angle droit, mais en dimensions supérieures il y en a une infinité.

- Les directions ainsi déterminées sont appelées composantes principales.

- En général, il y en autant que de variables initiales.

ACP

- Le deuxième graphique représente les mêmes données mais après une rotation, de façon à ce que la première composante coïncide avec l’axe des abscisses et la deuxième avec l’axe des ordonnées.

- Avant d’effectuer la rotation, la moyenne est soustraite aux données, de façon à ce que les données transformées soient centrées sur zéro.

ACP

- On peut utiliser l'ACP pour réduire la dimension en ne retenant que certaines composantes principales.

- Dans l'exemple précédent, on pourrait ne retenir que la première composante, ce qui réduit la dimension à 1.

- Finalement, on peut inverser la rotation et rajouter la moyenne aux données. Les points obtenus sont dans l'espace des variables initiales mais on n'a conservé que l'information contenue dans la première composante.

- Cette transformation est notamment utilisée pour débruiter des données.

ACP

- Nous allons a présent appliquer l'ACP aux données cancer en vue de les visualiser.

- C'est une des utilisations les plus fréquentes de l'ACP.

- Pour les données iris, nous avions pu obtenir une visualisation en représentant les nuages de points croisés des variables prises deux à deux, mais pour les données cancer, il y a 30 variables, ce qui ferait 30*14 = 420 nuages de points croisés ! il n'est pas possible d'examiner ces nuages de points simultanément, encore moins de les comprendre.

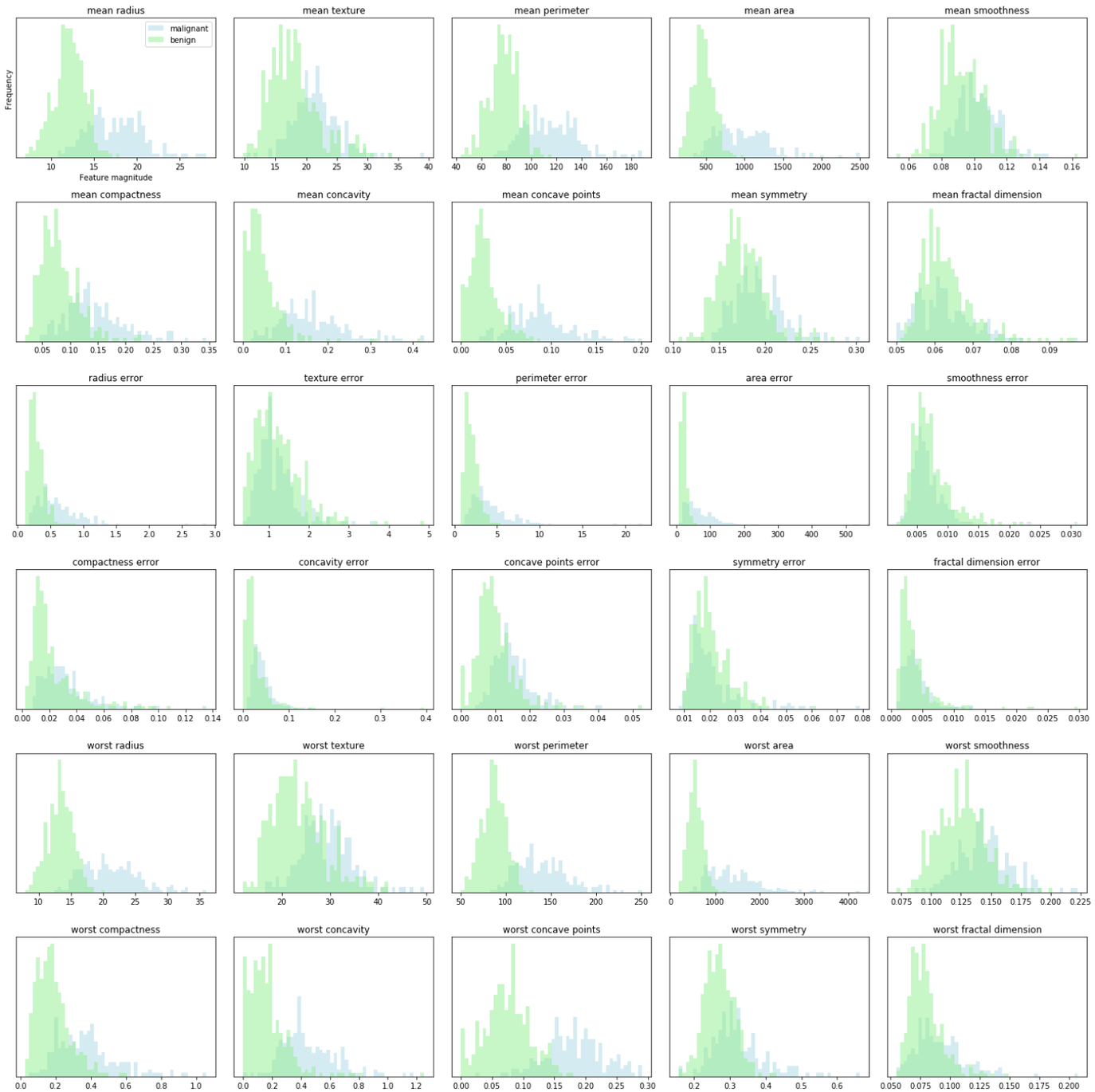
ACP

- Nous pouvons toutefois utiliser une visualisation plus simple : représenter les histogrammes de chaque variable pour les deux classes, tumeurs bénignes et malignes.

- C’est fait à l’aide du programme *cancer_histo.py*.

- Le résultat est le suivant :

ACP


mean radius
malignant
benign
Frequency
Feature magnitude
mean texture
mean perimeter
mean area
mean smoothness
mean compactness
mean concavity
mean concave points
mean symmetry
mean fractal dimension
radius error
texture error
perimeter error
area error
smoothness error
compactness error
concavity error
concave points error
symmetry error
fractal dimension error
worst radius
worst texture
worst perimeter
worst area
worst smoothness
worst compactness
worst concavity
worst concave points
worst symmetry
worst fractal dimension

ACP

- Le graphique précédent peut nous donner une idée des variables les plus pertinentes pour distinguer tumeurs bénignes et malignes (une variable est pertinente si les histogrammes représentés sont suffisamment disjoints).

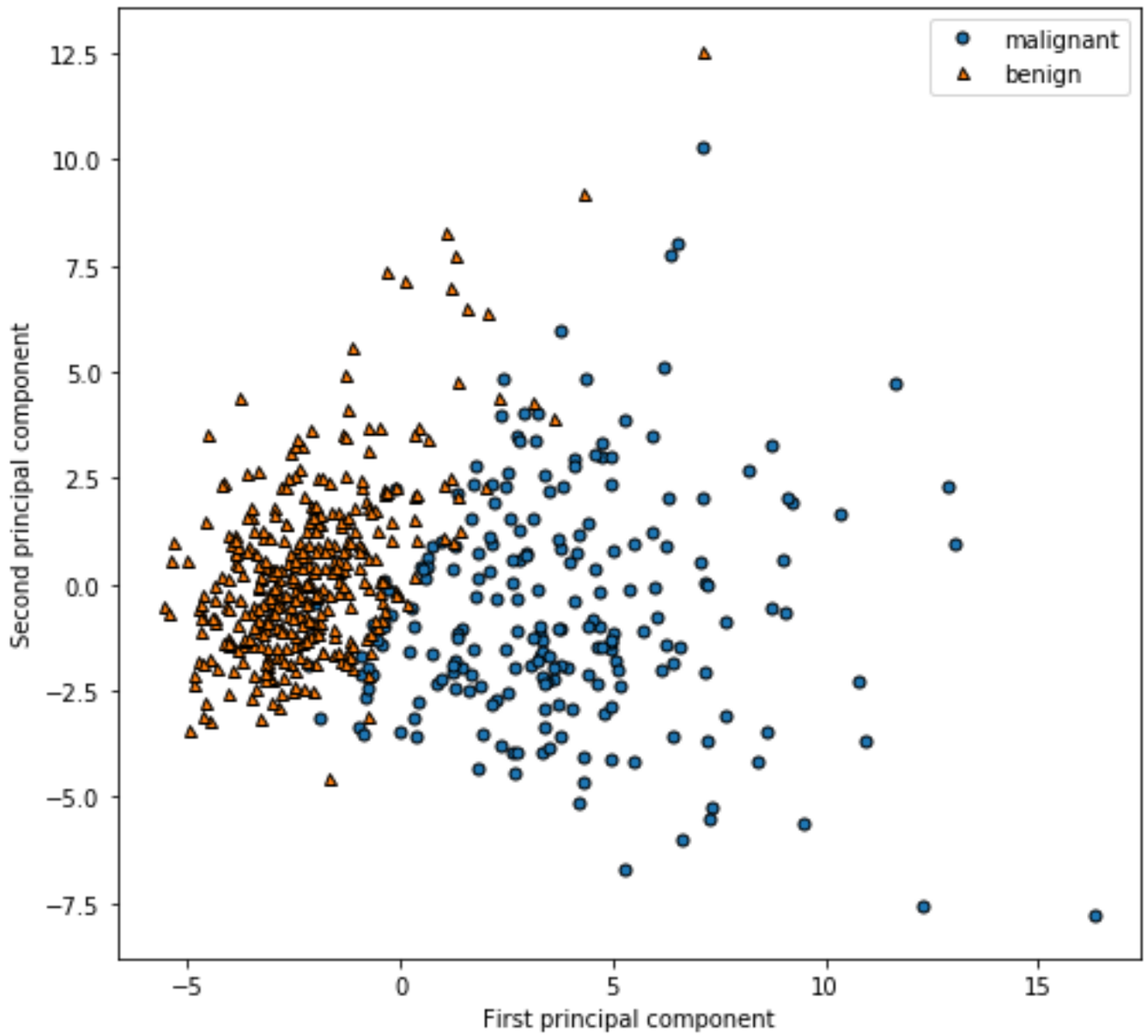
- Toutefois, ce graphique ne nous dit rien sur les interactions entre variables et leurs liens avec les deux classes.

- L'ACP permet de rendre compte des interactions et fournit une représentation graphique plus complète.

ACP

- La représentation graphique des données dans le plan formé des deux première composantes principales est la suivante (programme *cancer_PCA.py*) :

ACP


malignant
benign
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
Second principal component
−5
0
5
10
15
First principal component

ACP

- Il est important de noter que l'ACP est une méthode non supervisée et qu'elle n'utilise pas l'information sur les classes lorsqu'elle détermine la rotation recherchée.

- Elle prend en compte simplement les corrélations entre les variables.

- On voit toutefois que les deux classes sont bien séparées dans l'espace obtenu. En fait, un simple classificateur linéaire dans cet espace pourrait les distinguer de façon satisfaisante.

ACP

- Un inconvénient de l'ACP est que les axes obtenus ne sont pas toujours faciles à interpréter.

- Les composantes principales sont des combinaisons linéaires des variables initiales. Toutefois, ces combinaisons sont généralement très complexes.

- Pour plus de détails, on pourra consulter le cours de Data Mining et Analyse des Données.

Clustering par k-moyennes

k-means

- Le clustering consiste à partitionner les données en groupes, appelés clusters ou classes, tels que les individus appartenant à une même classe soient similaires et ceux appartenants à des classes différentes soient différents.

- De façon analogue aux algorithmes de classification, les algorithmes de clustering associent un nombre à chaque individu indiquant le cluster auquel il appartient.

k-means

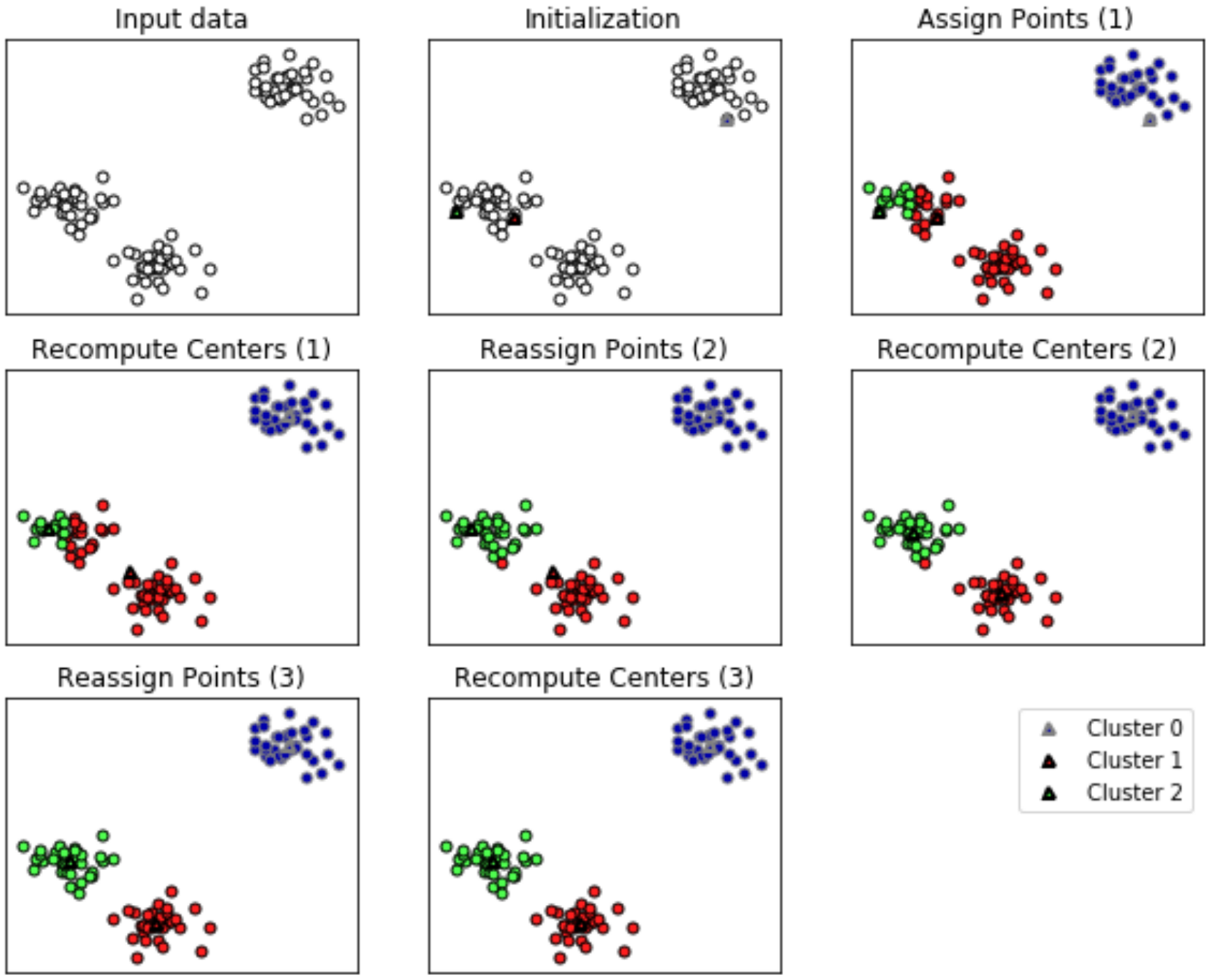
- Le clustering par k-moyennes est un des algorithmes de clustering les plus simples et les plus utilisés.

- Il procède en déterminant des centres de classes représentatifs de certaines régions de l'espace des données.

- L'algorithme alterne entre deux étapes : 1) affecter chaque individu à la classe du centre le plus proche, puis 2) recalculer le centre comme la moyenne des points qui lui sont associés. L'algorithme s'arrête lorsque l'affectation des individus aux classes se stabilise.

- Illustration sur des données simulées (programme *kmeans_exemple.py*) :

k-means


Input data
Initialization
Assign Points (1)
Recompute Centers (1)
Reassign Points (2)
Recompute Centers (2)
Reassign Points (3)
Recompute Centers (3)
Cluster 0
Cluster 1
Cluster 2

k-means

- Les centres de classes sont représentés par des triangles alors que les points de données sont représentés par des cercles.

- Les couleurs indiquent l’appartenance aux classes.

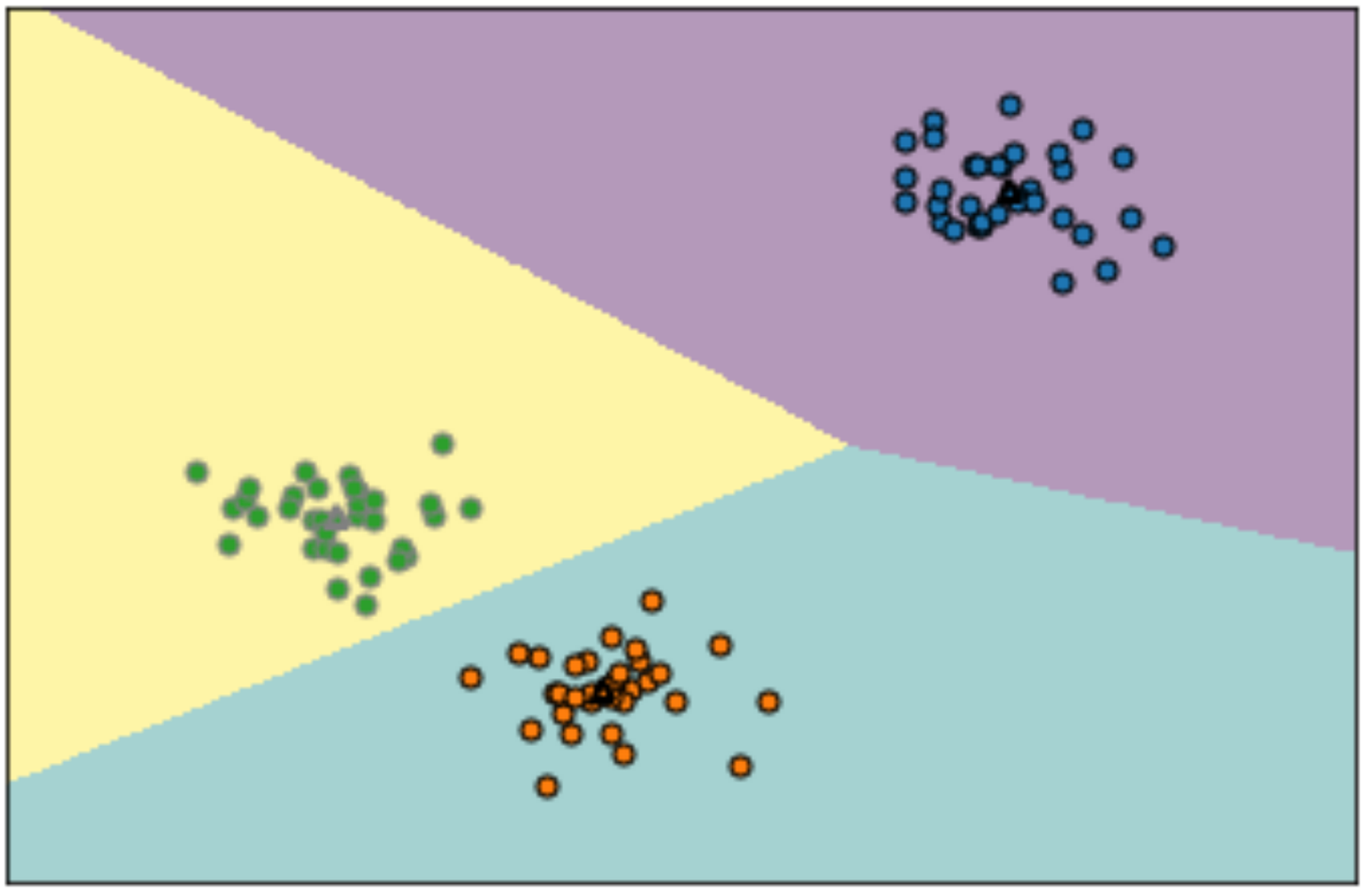
- On a spécifié qu’on voulait trois classes. L’algorithme est donc initialisé en choisissant trois points au hasard comme entres de classes. Puis les étapes itératives sont lancées. En trois itérations, les classes se stabilisent et l’algorithme s’arrête.

k-means

- *Etant donné un nouveau point, l'algorithme k-means l'affecte à la classe de centre le plus proche.*

- Le graphique suivant représente les frontières des classes apprises précédemment (programme *kmeans_exemple.py*) :

k-means



k-means

- Utiliser k-means avec scikit-learn est assez immédiat.

- On instancie la classe KMeans en fixant le nombre de classes souhaité, puis on lance la méthode fit avec les données (programme *kmeans.py*) :

k-means

```
from sklearn.datasets import make_blobs

from sklearn.cluster import KMeans
```

```
X, y = make_blobs(random_state=1)

kmeans = KMeans(n_clusters=3)

kmeans.fit(X)
```

```
print("Cluster memberships:\n{}".format(kmeans.labels_))

print(kmeans.predict(X))
```

k-means

- kmeans.labels_ contient les affectations des individus aux différentes classes.

- La méthode predict permet de déterminer l'affectation d'un nouvel individu à une classe déterminée par l'algorithme.

- *Pour plus de détails sur les k-means, voir le cours de Data Mining et Analyse des Données.*

Clustering agglomératif (CAH)

CAH

- Le clustering agglomératif ou classification ascendante hiérarchique est une famille d'algorithmes conçus selon le même principe : on commence par considérer que chaque point constitue une classe à part entière puis on fusionne les deux classes les plus similaires, jusqu'à ce qu'un critère d'arrêt soit vérifié.

- Le critère d'arrêt spécifié dans scikit-learn est le nombre de clusters; ainsi les classes similaires sont fusionnées jusqu'à ce que le nombre spécifié de classes soit atteint.

CAH

- Il y a divers critères de « linkage » qui définissent la notion de similarité entre classes retenue par l'algorithme.

- Les trois possibilités suivantes sont offertes par scikit-learn :

  - ward : c'est le choix par défaut. Ce choix de linkage conduit souvent à des classes d'effectif comparable.

  - average

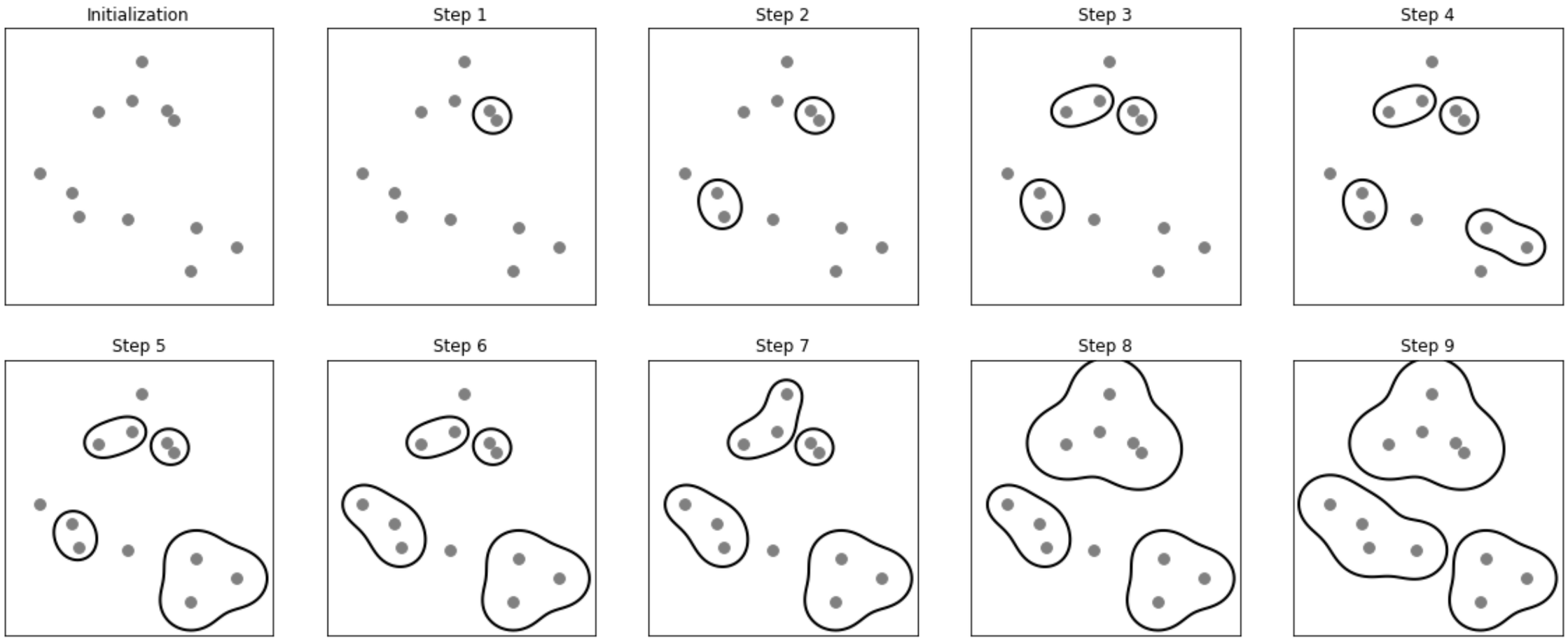
  - complete (ou maximum)

CAH

- On utilisera dans la suite la méthode de ward, qui est la plus utilisée en pratique.

- Le graphique suivant illustre la progression de la CAH sur un jeu de données bi-dimensionnel, pour lequel on cherche un clustering en 3 classes (programme *cah_exemple.py*) :

CAH


Initialization
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7
Step 8
Step 9

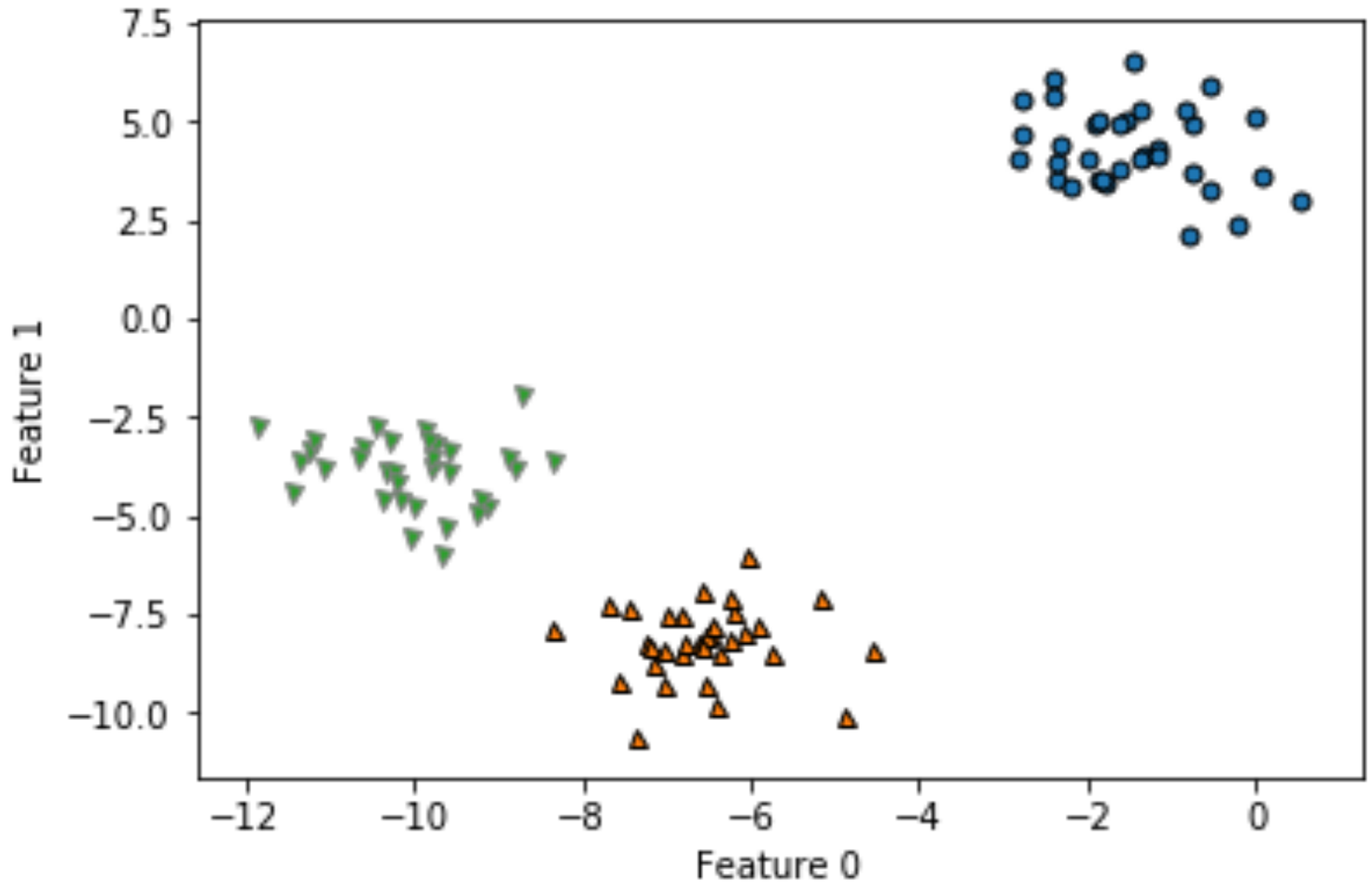
CAH

- Initialement, chaque point constitue une classe. Puis, à chaque étape, les deux classes les plus proches sont fusionnées. A l'étape 9, on n'a plus que 3 classes. Comme c'est le nombre de classes spécifié, l'algorithme s'arrête.

- Examinons la façon dont la CAH fonctionne sur des données simulées réparties en 3 classes.

- En raison de la structure de l'algorithme, la CAH ne dispose pas d'une méthode predict pour affecter de nouveaux points aux classes. Si on veut obtenir l'affectation des individus aux classes, il faut utiliser la méthode fit_predict (*programme cah.py*).

CAH



CAH

- La CAH retrouve les classes de façon parfaite.

- Alors que l'implémentation sous scikit-learn de la CAH requiert de spécifier le nombre de clusters a priori, nous allons voir à présent comment choisir ce nombre sous Python.

- Pour cela, nous allons montrer comment représenter une CAH à l'aide d'un dendrogramme sous Python.

- C'est fait à l'aide de l'implémentation de la CAH sous SciPy (programme *cah_scipy.py*).

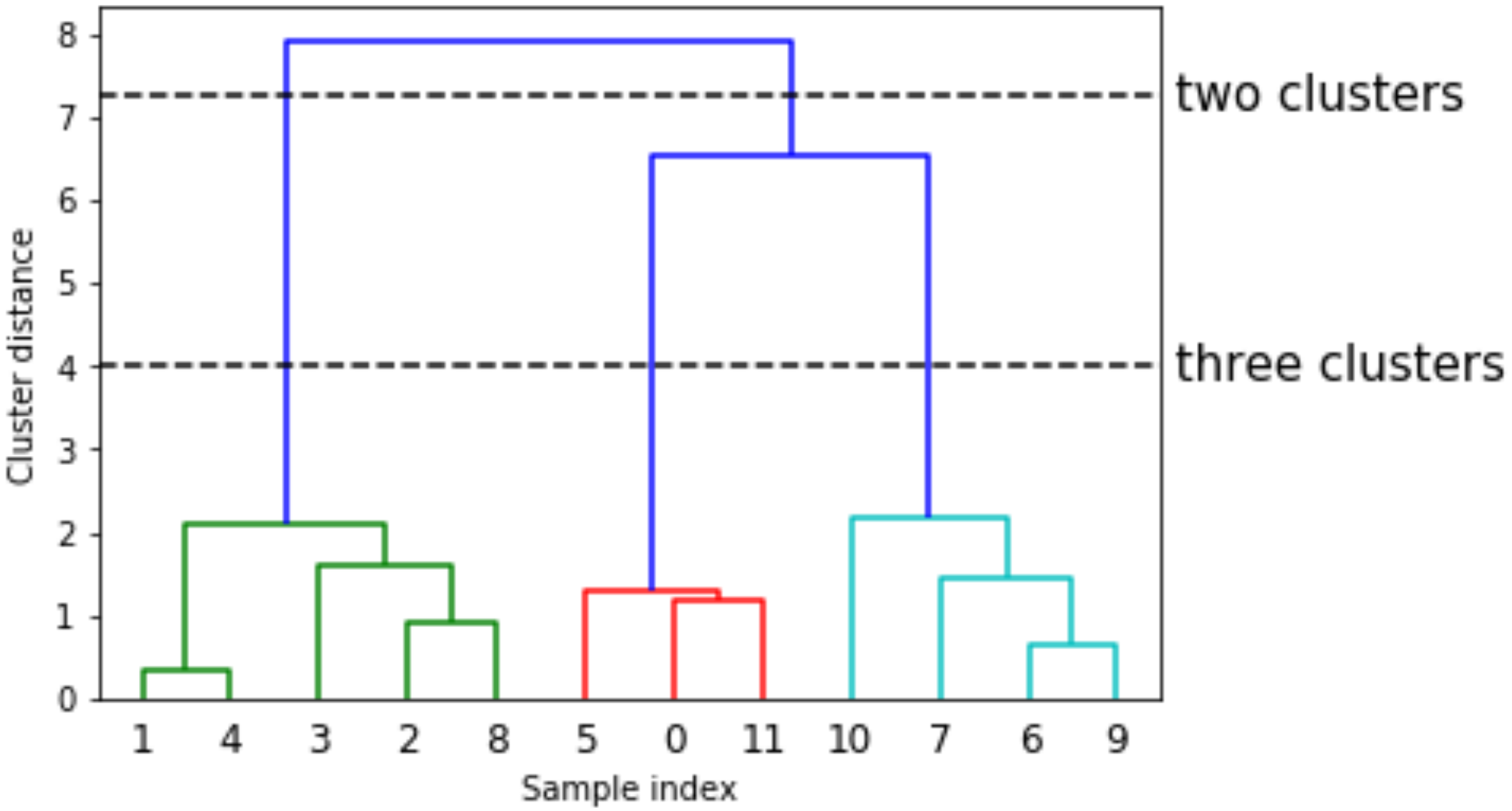
CAH

```
from sklearn.datasets import make_blobs

from scipy.cluster.hierarchy import dendrogram, ward
import matplotlib.pyplot as plt


X, y = make_blobs(random_state=0, n_samples=12)

linkage_array=ward(X)

dendrogram(linkage_array)
```

CAH

```
ax = plt.gca()

bounds = ax.get_xbound()

ax.plot(bounds,[7.25,7.25], '--', c='k')

ax.plot(bounds,[4,4],'--',c='k')

ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size':
15})

ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})

plt.xlabel("Sample index")

plt.ylabel("Cluster distance")
```

CAH


two clusters
three clusters
Cluster distance
0
1
2
3
4
5
6
7
8
1
4
3
2
8
5
0
11
10
7
6
9
Sample index

CAH

- *Pour davantage d'information sur la CAH, voir le cours de Data Mining et Analyse des Données.*

- Une autre méthode de clustering très utilisée en pratique est la méthode DBSCAN, disponible sous scikit-learn.

- Elle n'a pas les inconvénients des k-means et de la CAH, qui peuvent échouer à identifier les classes lorsque celles-ci ont des formes complexes.

Introduction à la Programmation Orientée Objet sous Python

Python objet

- Les classes sont les principaux outils de la programmation orientée objet.
- Ce type de programmation permet de structurer des logiciels complexes en les organisant comme des ensembles d’objets qui interagissent, entre eux et avec le monde extérieur.

Python objet

- Le premier bénéfice de cette approche de la programmation réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres sans qu'il n'y ait de risque d'interférence.

- Ce résultat est obtenu grâce au concept d'encapsulation : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail sont en quelque sorte enfermées dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'interface de l'objet.

Python objet

- En particulier, l'utilisation de classes dans les programmes permet - entre autres avantages - d'éviter au maximum l'emploi de variables globales.

- En effet, l'utilisation de variables globales comporte des risques, d'autant plus importants que les programmes sont volumineux, parce qu'il est toujours possible que de telles variables soient modifiées ou redéfinies n'importe où dans le corps du programme.

Python objet

- Un deuxième bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de construire de nouveaux objets à partir d'objets préexistants, et donc de réutiliser des pans entiers d'une programmation déjà écrite pour en tirer une fonctionnalité nouvelle.

- C'est rendu possible grâce aux concepts de dérivation et de polymorphisme.

Python objet

- La dérivation est le mécanisme qui permet de construire une classe « enfant » à partir d'une classe « parente ».

- L'enfant ainsi obtenu hérite de toutes les propriétés et de toute les fonctionnalités de son ancêtre, auxquelles on peut ajouter ce que l'on veut.

Python objet

- Le polymorphisme permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet en fonction d'un certain contexte.

- Signalons que la programmation orientée objet (POO) est optionnelle sous Python.

- On peut mener à bien de nombreux projets sans l'utiliser, en se limitant par exemple à des fonctions. Toutefois, la maîtriser permet de traiter des problèmes plus complexes.

Définition d'une classe élémentaire

- Pour créer une nouvelle classe d'objets Python, on utilise l'instruction class.

- Pour l'illustrer nous allons commencer par définir un type d'objet très rudimentaire, qui sera simplement un nouveau type de donnée : le type point.

- Ce type correspond au concept de point en géométrie plane. Dans le plan, un point est caractérisé par deux nombres (son abscisse et son ordonnée).

Définition d'une classe élémentaire

- En notation mathématique, on représente un point par ses deux coordonnées x et y mises entre parenthèses.

- On parlera par exemple du point (25, 17).

- Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type float. On veut cependant combiner ces deux valeurs dans une seule entité ou un seul objet. Pour y arriver on va définir une classe Point( ).

Définition d'une classe élémentaire

```
class Point(object):

    "Définition d'un point géométrique"
```

- *Les définitions de classes peuvent être situées n'importe où dans un programme, mais on les place généralement au début.*

- Le double point est obligatoire à la fin de la ligne après l'instruction class, ainsi que l'indentation du bloc d'instructions qui suit.

- Ce bloc doit contenir au moins une ligne : par exemple, une chaîne de caractère décrivant la classe, qui sera automatiquement incorporée dans la documentation de Python.

Définition d'une classe élémentaire

- Les parenthèses sont destinées à contenir la référence d'une classe préexistante. C'est requis pour permettre le mécanisme d'héritage.

- Lorsqu'on souhaite créer une classe fondamentale - c'est-à-dire ne dérivant d'aucune autre - la référence à indiquer est par convention le nom spécial object, qui désigne l'ancêtre de toutes les classes.

- *Une convention très répandue consiste à donner aux classes des noms commençant par une majuscule.*

Définition d'une classe élémentaire

- On vient de définir une classe Point( ). On peut à présent s'en servir pour créer des objets de cette classe, que l'on appellera également des instances de cette classe.

- L'opération s'appelle pour cette raison une instanciation.

- Exemple :

```
p9 = Point( )
```

Définition d'une classe élémentaire

- Après cette instruction, la variable p9 contient la référence d'un nouvel objet Point( ). On dit également que p9 est une nouvelle instance de la classe Point( ).

- Mais, à ce stade que peut-on faire avec l'objet p9 ?

- Essayons l'instruction suivante :

```
print(p9)
```

Définition d'une classe élémentaire

- Résultat :

```
<__main__.Point object at 0x1a1a549748>
```

- Le message renvoyé par Python indique que p9 est une instance de la classe Point( ), laquelle est définie au niveau principal (main) du programme. Elle est situé dans un emplacement déterminé de la mémoire vive, dont l'adresse est fournie en notation hexadécimale.

Définition d'une classe élémentaire

- Essayons à présent l'instruction suivante :

```
print(p9.__doc__)
```

- Résultat :

```
Définition d'un point géométrique
```

Définition d'une classe élémentaire

- Les chaîne de documentation de divers objets sous Python sont associées à l'attribut prédéfini __doc__.

- *Il est donc toujours possible de retrouver la documentation associée à un objet Python quelconque en invoquant cet attribut.*

Attributs (ou variables) d'instance

- L'objet qu'on vient de créer est une coquille vide.
- On va lui ajouter des composants, par simple assignation :

```
p9.x = 3.0

p9.y = 4.0
```

Attributs (ou variables) d'instance

- Les variables x et y qu'on a définies en les liant d'emblée à p9, sont désormais des attributs de l'objet p9.

- On les appelle également des variables d'instance.

- Elles sont en effet encapsulées dans cette instance (ou objet).

Attributs (ou variables) d'instance

- On peut utiliser les attributs d'un objet dans n'importe quelle expression exactement comme des variables ordinaires :

```
print(p9.x)

print(p9.x**2+p9.y**2)
```

Attributs (ou variables) d'instance

- Résultat :

```
3.0

25.0
```

- Du fait de leur encapsulation dans l'objet, les attributs sont des variables distinctes d'autres variables qui pourraient porter le même nom.

- Par exemple, l'instruction :

```
x = p9.x
```

Attributs (ou variables) d'instance

- signifie : extraire de l'objet référence par p9 la valeur de son attribut x et assigner cette valeur à la variable x.

- Il n'y a pas de conflit entre la variable indépendante x et l'attribut x de p9.

- L'objet p9 contient son propre espace de noms, indépendant de l'espace de noms principal où se trouve la variable x.

Attributs (ou variables) d’instance

- Notons que, s’il et très aisé d’ajouter un attribut à un objet en utilisant une simple instruction d’assignation, ce n’est pas la façon recommandée de procéder. Cette dernière sera développée dans la suite.

Passage d'objets comme arguments d'une fonction

- *Les fonctions peuvent utiliser des objets comme paramètres et elles peuvent fournir un objet comme valeur de retour.*

- Exemple :

```
def affiche_point(p):

    print("Abscisse = ", p.x, "Ordonnée = ", p.y)
```

Passage d'objets comme arguments d'une fonction

- Le paramètre p utilisé par cette fonction doit être un objet de type Point( ), dont l'instruction qui suit utilise les variables d'instance p.x et p.y.

- Lorsqu'on appelle cette fonction, on doit lui fournir un objet de classe Point( ) comme argument.

- Exemple :

```
affiche_point(p9)
```

- Résultat :

```
Abscisse =  3.0 Ordonnée =  4.0
```

Similitude et unicité

- Dans la langue parlée, les mêmes mots peuvent avoir des signification différentes selon le contexte dans lequel on les utilise.

- La conséquence en est que certaines expressions utilisant ces mots peuvent être comprises de plusieurs façon différentes (expressions ambiguës).

- Exemple : le mot « même » a des significations différentes dans les phrases « Charles et moi avons la même voiture » et « Charles et moi avons la même mère ».

Similitude et unicité

- Lorsqu'on traite d'objets logiciels, on peut rencontrer la même ambiguïté.
  - *Si nous parlons de l'égalité de deux objets Point( ), cela signifie-t-il que ces deux objets contiennent les mêmes données (leurs attributs), ou bien cela signifie-t-il que nous parlons de deux références à un même et unique objet ?*
- Exemple :

Similitude et unicité

```
p1 = Point()

p1.x = 3

p1.y = 4
```

```
p2 = Point()

p2.x = 3

p2.y = 4


print(p1 == p2)
```

Similitude et unicité

- Résultat :

```
False
```

- Les instruction précédentes créent deux objets p1 et p2 qui restent distincts, même s'ils font partie d'une même classe et on des contenus similaires.

- On peut confirmer cela d'une autre manière :

Similitude et unicité

```
print(p1)
```

- Résultat :

```
<__main__.Point object at 0x1810424a58>
```

```
print(p2)
```

- Résultat :

```
<__main__.Point object at 0x1810424a20>
```

Similitude et unicité

- Les deux variables p1 et p2 référencent bien des objets différents, mémorisés à des emplacements différents dans la mémoire de l'ordinateur.

- Essayons autre chose à présent :

```
p2 = p1

print(p1 == p2)
```

- Résultat :

```
True
```

Similitude et unicité

- Par l'instruction p2 = p1, nous assignons le contenu de p1 à p2.

- Cela signifie que désormais ces deux variables référencent le même objet.

- On dit que les variables p1 et p2 sont des alias l'une de l'autre.

- Lorsqu'on modifie un attribut de p1, l'attribut correspondant de p2 change lui aussi :

Similitude et unicité

```
p1.x = 7

print(p2.x)
```

- Résultat :

```
7
```

- Les deux références p1 et p2 pointent vers le même emplacement dans la mémoire :

Similitude et unicité

```
print(p1)
```

- Résultat :

```
<__main__.Point object at 0x1810424a58>
```

```
print(p2)
```

- Résultat :

```
<__main__.Point object at 0x1810424a58>
```

Objets composés d'objets

- Supposons qu'on veuille définir une classe qui serve à représenter les rectangles.

- Pour simplifier, on suppose que ces rectangles sont toujours orientés horizontalement ou verticalement et jamais en oblique.

- De quelles informations avons-nous besoin pour définir de tels rectangles ?

Objets composés d’objets

- Il existe plusieurs possibilités.

- On peut par exemple spécifier la position du centre du rectangle (deux coordonnées) et préciser sa taille (largeur et hauteur).

- On peut aussi spécifier la position du coin supérieur gauche et du coin inférieur droit, ou la position du coin supérieur gauche et la taille.

- On retient cette dernière possibilité.

Objets composés d'objets

- Définissons notre nouvelle classe :

```
class Rectangle(object):

    "définition d'une classe de rectangles"
```

- On crée une instance de la façon suivante :

```
boite = Rectangle()

boite.largeur = 50.0

boite.hauteur = 35.0
```

Objets composés d'objets

- On crée ainsi un nouvel objet Rectangle( ) et on lui donne deux attributs.

- Pour specifier le coin supérieur gauche, on utilise à présent une instance de la classe Point( ) que nous avons définie précédemment.

- Ainsi, on définit un objet à l'intérieur d'un autre objet :

Objets composés d'objets

```
boite.coin = Point()
boite.coin.x = 12.0
boite.coin.y = 27.0
```

- La première instruction permet de créer un nouvel attribut coin pour l'objet boite.

Objets composés d'objets

- Ensuite, pour accéder à cet objet, qui se trouve lui-même à l'intérieur d'un autre objet, on utilise la qualification des noms hiérarchisée à l'aide de points.

- Ainsi, l'expression boite.coin.y signifie « aller à l'objet référencé dans la variable boite. Dans cet objet, repérer l'attribut coin, puis aller à l'objet référencé dans cet attribut. Une fois cet objet trouvé, sélectionner son attribut y. »

Objets composés d'objets

- Le nom boite se trouve dans l'espace de noms principal.
- Il référence un autre espace de noms réservé à l'objet correspondant, dans lequel sont mémorisés les noms *largeur*, *hauteur* et *coin*.
- Ceux-ci référencent à leur tour soit d'autres espaces de noms (c'est le cas de *coin*), soit des valeurs bien déterminées, lesquelles sont mémorisées ailleurs.

Objets composés d’objets

- Python réserve des espaces de noms différents pour chaque module, chaque classe, chaque instance, chaque fonction.

- On peut tirer parti de tous ces espaces de noms bien compartimentés afin de réaliser des programmes robustes, c’est-à-dire dont les différentes composantes n’interfèrent pas facilement.

Objets comme valeurs de retour d'une fonction

- *Les fonctions peuvent transmettre une instance comme valeur de retour.*

- Par exemple, la fonction trouveCentre( ) ci-après doit être appelée avec un argument de type Rectangle( ) et elle renvoie un objet de type Point( ), contenant les coordonnées du centre du rectangle :

Objets comme valeurs de retour d'une fonction

```
def trouveCentre(box):
    p = Point()

    p.x = box.coin.x + box.largeur/2.0

    p.y = box.coin.y - box.hauteur/2.0

    return p
```

Objets comme valeurs de retour d'une fonction

- On peut par exemple appeler cette fonction en utilisant comme argument l'objet boite défini précédemment :

```
centre = trouveCentre(boite)

print(centre.x, centre.y)
```

- Résultat :

```
37.0 44.5
```

Modification des objets

- *On peut modifier les propriétés d'un objet en assignant de nouvelles valeurs à ses attributs.*
  - Par exemple, on peut modifier la taille d'un rectangle (sans modifier sa position) en réassignant ses attributs hauteur et largeur :

```
boite.hauteur = boite.hauteur + 20

boite.largeur = boite.largeur - 5
```

Modification des objets

- On peut faire cela sous Python car dans ce langage les propriétés des objets sont toujours publiques.

- D'autres langages établissent une distinction nette entre attributs publics (accessibles de l'extérieur de l'objet) et attributs privés (accessibles seulement aux algorithmes inclus dans l'objet lui-même).

- Cependant, comme on l'a déjà signalé, modifier les attributs d'un objet par assignation simple, depuis l'extérieur de l'objet, n'est pas une pratique recommandable.

Modification des objets

- En effet, elle contredit un des objectifs fondamentaux de la programmation objet, qui vise à établir une séparation stricte entre la fonctionnalité d'un objet (telle que déclarée au monde extérieur) et la manière dont cette fonctionnalité est réellement implémentée dans l'objet (et que le monde extérieur n'a pas à connaître).

- Concrètement, cela signifie qu'on va maintenant étudier comment faire fonctionner les objets à l'aide d'outils vraiment appropriés, qu'on appellera des méthodes.

Modification des objets

- Lorsqu'on aura bien compris le maniement de celles-ci, on se fixera pour règle de ne plus modifier les attributs d'un objet par assignation directe depuis l'extérieur, comme on l'a fait jusqu'à présent.

- On veillera au contraire à utiliser pour cela des méthodes mises en place spécifiquement dans ce but.

- L'ensemble de ces méthodes constituera ce que nous appellerons l'interface de l'objet.

Classes, méthodes, héritage

POO

- L’idée de base de la programmation orientée objet consiste à regrouper dans un même ensemble à la fois un certain nombre de données (les attributs d’instance) et les algorithmes destinés à effectuer divers traitements sur ces données (les méthodes, i.e. des fonctions particulières encapsulées dans l’objet).

- *Objet = (attributs + méthodes)*

POO

- Cette façon d'associer dans une même capsule les propriétés d'un objet et les fonctions qui permettent d'agir sur elles correspond chez les programmeurs à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde qui nous entoure.

- Considérons par exemple un widget « bouton » dans une application graphique.

POO

- Il nous paraît raisonnable de souhaiter que l'objet informatique que nous appelons « bouton » ait un comportement qui ressemble à celui d'un bouton d'appareil quelconque dans le monde réel.

- Or la fonctionnalité d'un bouton réel (sa capacité de fermer ou d'ouvrir un circuit électrique) est intégrée dans l'objet lui-même, au même titre que des propriétés telles que sa taille sa couleur, etc.

POO

- De la même manière, on souhaite que les différentes caractéristiques du bouton logiciel (sa taille son emplacement, sa couleur, le texte associé) mais aussi la définition de ce qui se passe lorsqu'on effectue diverses actions de la souris sur ce bouton, soient regroupés dans une entité bien précise à l'intérieur du programme, pour qu'il n'y ait pas de confusion entre ce bouton et un autre, ou entre ce bouton et d'autres entités.

Méthodes

- Pour illustrer notre propos nous allons définir une nouvelle classe Time( ) qui doit nous permettre d'effectuer diverses opération sur des instants, des durées, etc.

```
class Time(object):

    "Définition d'objets temporels"
```

Méthodes

- Créons à présent un objet de ce type et ajoutons-lui des variables d'instance pour mémoriser les heures, minutes et secondes :

```
instant = Time()

instant.heure = 11

instant.minute = 34

instant.seconde = 25
```

Méthodes

- Définissons une fonction affiche_heure( ) permettant de visualiser le contenu d'un objet de classe Time( ) sous la forme conventionnelle « heures:minutes:secondes » et appliquons-là à l'objet instant créé précédemment :

```
def affiche_heure(t):

    print(str(t.heure)+":"+str(t.minute)+":"+str(t.seconde))

affiche_heure(instant)
```

Méthodes

- Résultat :

```
11:34:25
```

- Si on doit utiliser fréquemment des objets de la classe Time( ), la fonction affiche_heure devient très utile.

- Il est donc souhaitable d'encapsuler cette fonction dans la classe Time( ), de manière à s'assurer qu'elle est toujours disponible dès lors qu'on manipule des objets de cette classe.

Méthodes

- Une fonction encapsulée dans une classe s’appelle une méthode.

- On définit une méthode comme on définit une fonction, c’est-à-dire en écrivant un bloc d’instructions à la suite du mot réservé def, mais avec deux différences :

Méthodes

- La définition d'une méthode est toujours placée à l'intérieur de la définition d'une classe, de manière à ce que la relation qui lie méthode et classe soit clairement établie.

- La définition d'une méthode doit toujours comporter au moins un paramètre, qui doit être une référence d'instance et ce paramètre particulier doit toujours être listé en premier.

Méthodes

- On peut en principe utiliser un nom de variable quelconque pour ce premier paramètre mais il est vivement conseillé de respecter la convention qui consiste à lui donner le nom : self.

- Ce paramètre self est nécessaire parce qu'il faut pouvoir désigner l'instance à laquelle la méthode est associée dans les instructions faisant partie de la définition.

- Exemple :

Méthodes

```
class Time(object):
    "Nouvelle classe temporelle"

    def affiche_heure(self):
        print("{0}:{1}:{2}".format(self.heure,self.minute,self.seconde))
```

Méthodes

- On dispose donc à présent d'une classe Time( ), dotée d'une méthode affiche_heure( ).

- Exemple :

```
maintenant = Time()

maintenant.affiche_heure()
```

- Résultat :

```
AttributeError: 'Time' object has no attribute 'heure'
```

Méthodes

- L’erreur est normale : on a pas créé les attributs d’instance.

```
maintenant.heure = 23

maintenant.minute = 34

maintenant.seconde = 21

maintenant.affiche_heure()
```

Méthodes

- Résultat :

```
23:34:21
```

- *L’erreur que nous avons rencontrée peut-elle être évitée ?*

Constructeur

- Elle ne se serait pas produite si on s'était arrangé pour que la méthode affiche_heure( ) puisse toujours afficher quelque chose, sans qu'il soit nécessaire d'effectuer *au préalable* une manipulation sur l'objet nouvellement créé.

- En d'autres termes, il faudrait que les variables d'instances soient prédéfinies elles aussi à l'intérieur de la classe, avec pour chacune d'elles une valeur par défaut.

Constructeur

- Pour obtenir cela, on fait appel à une méthode particulière, qu'on appelle constructeur.

- La méthode constructeur a ceci de particulier qu'elle est exécutée automatiquement lorsqu'on instancie un nouvel objet à partir de la classe.

- On peut donc y placer tout ce qui semble nécessaire pour initialiser automatiquement l'objet que l'on crée.

Constructeur

- Afin qu'elle soit reconnue comme telle par Python, la méthode constructeur doit obligatoirement s'appeler __init__ .

- Exemple :

Constructeur

```
class Time(object):

    "Encore une nouvelle classe temporelle"

    def __init__(self):

        self.heure = 12

        self.minute = 0

        self.seconde = 0

    def affiche_heure(self):

        print("{0}:{1}:{2}".format(self.heure,self.minute,self.seconde))


tstart = Time()

tstart.affiche_heure()
```

Constructeur

- Résultat :

```
12:0:0
```

- On n’a pas d’erreur cette fois.

- Comme toute méthode, la méthode __init__ peut être dotée de paramètres.

- Dans le cas de cette méthode particulière qu’est le constructeur, les paramètres peuvent jouer un rôle très intéressant car ils permettent d’initialiser certaines des variables d’instances au moment même de la création de l’objet.

Constructeur

- Exemple :

```
class Time(object):

    "Encore une nouvelle classe temporelle"

    def __init__(self, hh = 12, mm = 0, ss = 0):

        self.heure = hh

        self.minute = mm

        self.seconde = ss

    def affiche_heure(self):

        print("{0}:{1}:{2}".format(self.heure,self.minute,self.seconde))
```

Constructeur

- La nouvelle méthode __init__ comporte trois paramètres, ayant chacun une valeur par défaut.

- On obtient ainsi une classe encore plus perfectionnée.

- Lorsqu'on instancie un objet de cette classe, on peut initialiser ses principaux attributs à l'aide d'arguments, au sein même de l'instruction d'instanciation.

- Si on omet tout ou partie d'entre eux, les attributs reçoivent de toute façon des valeurs par défaut.

Constructeur

- Exemple :

```
recreation = Time(10,15,18)
```

```
recreation.affiche_heure()
```

- Résultat :

```
10:15:18
```

Constructeur

- Exemple :

```
rentree = Time(10,30)
rentree.affiche_heure()
```

- Résultat :

```
10:30:0
```

Constructeur

- Exemple :

```
rdv = Time(hh = 18)
```

```
rdv.affiche_heure()
```

- Résultat :

```
18:0:0
```

Espaces de noms

- Les variables définies à l'intérieur d'une fonction sont des variables locales, inaccessibles aux instructions qui se trouvent à l'extérieur de la fonction.

- Cela permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence.

- En d'autres termes, chaque fonction possède son propre espace de noms, indépendant de l'espace de noms principal.

Espaces de noms

- Les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais en consultation seulement : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier, à moins de faire appel à l'instruction global.

- Il existe donc une sorte de hiérarchie entre les espaces de noms.

- Il en va de même à propos des classes et des objets.

Espaces de noms

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées variables de classe ou attributs de classe.

- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées variables d'instance ou attributs d'instance.

Espaces de noms

- Les classes peuvent utiliser, mais pas modifier, les variables définies au niveau principal.

- Les instances peuvent utiliser, mais pas modifier, les variables définies au niveau de la classe et les variables définies au niveau principal.

Espaces de noms

- Exemple :

```
class Espaces(object):

    aa = 33

    def affiche(self):
        print(aa, Espaces.aa, self.aa)


aa = 12

essai = Espaces()

essai.aa = 67

essai.affiche()

print(aa, Espaces.aa, essai.aa)
```

Espaces de noms

- Résultat :

```
12 33 67
```

```
12 33 67
```

- Dans cet exemple, le même nom aa est utilisé pour définir trois variables différentes : une dans l’espace de noms de la classe, une autre dans l’espace de noms principal, et enfin une dernière dans l’espace de noms de l’instance.

Héritage

- Les classes constituent le principal outil de la programmation orientée objet, qui est considérée aujourd'hui comme la technique de programmation la plus performante.

- L'un des principaux atouts de ce type de programmation est que l'on peut se servir d'une classe préexistante pour en créer une nouvelle, qui hérite de toutes ses propriétés mais peut modifier certaines d'entre elles et/ou y ajouter les siennes propres.

- Le procédé s'appelle dérivation. Il permet de créer une hiérarchie de classes, allant du général au particulier.

Héritage

- On peut par exemple définir une classe Mammifere( ) qui contient un ensemble de caractéristiques propres à ce type d'animal.

- A partir de cette classe parente, nous pouvons dériver une ou plusieurs classes filles, comme : une classe Primate( ), une classe Rongeur( ), une classe Carnivore( ), etc, qui héritent de toutes les caractéristiques de la classe Mammifere( ), en y ajoutant leurs spécificités.

Héritage

- A partir de la classe Carnivore( ), nous pouvons ensuite dériver une classe Belette( ), une classe Loup( ), une classe Chien( ), etc, qui héritent encore une fois de toutes les caractéristiques de la classe parente avant d'y ajouter les leurs.

- Exemple :

Héritage

```
class Mammifere(object):

    caract1 = "il allaite ses petits ;"


class Carnivore(Mammifere):
    caract2 = "il se nourrit de la chair de ses proies ;"


class Chien(Carnivore):

    caract3 = "son cri s'appelle aboiement ;"


mirza = Chien()

print(mirza.caract1,mirza.caract2,mirza.caract3)
```

Héritage

- Résultat :

```
il allaite ses petits ; il se nourrit de la
chair de ses proies ; son cri s'appelle
aboiement ;
```

- Dans cet exemple, on voit que l'objet mirza, qui est une instance de la classe Chien( ), hérite non seulement de l'attribut défini pour cette classe, mais également des attributs définis pour les classes parentes.

Héritage

- Pour dériver une classe à partir d'une classe parente, on utilise l'instruction class, suivie du nom que l'on veut attribuer à la classe et on place entre parenthèses le nom de la classe parente.

- Les classes les plus fondamentales dérivent de l'objet « ancêtre » object.

Héritage

- Les attributs utilisés dans l'exemple précédent sont des attributs de classes (et non des attributs d'instance).

- L'instance mirza peut accéder à ces attributs mais pas les modifier.

```
mirza.caract2 = " son corps est couvert de poils ;"

print(mirza.caract2)
```

- Résultat :

```
son corps est couvert de poils ;
```

Héritage

```
fido = Chien()
print(fido.caract2)
```

- Résultat :

```
il se nourrit de la chair de ses proies ;
```

Héritage

- L’instruction

```
mirza.caract2 = " son corps est couvert de poils ; «
```

- crée une nouvelle variables d’instance associée seulement à l’objet mirza.

- Il existe dès lors deux variables avec le même nom caract2 : l’une dans l’espace de noms de l’objet mirza et l’autre dans l’espace de noms de la classe Carnivore( ).

Héritage

- *Mais s'il existe des variables identiques dans plusieurs espaces de noms, laquelle est sélectionnée lors de l'exécution d'une instruction ?*

- Pour résoudre ce conflit, Python respecte une règle de priorité très simple : lorsqu'on lui demande d'utiliser la valeur d'une variable nommée alpha, il commence par chercher ce nom dans l'espace local (le plus « interne » en quelque sorte). Si une variable alpha est trouvée dans l'espace local, c'est elle qui est utilisée et la recherche s'arrête. Sinon, Python examine l'espace de noms de la structure parente, puis celui de la structure grand-parente, etc, jusqu'au niveau principal du programme.

Héritage et polymorphisme

- On va a présent examiner un script plus détaillé.
- Rappelons d’abord quelque notions de Chimie.
  - Les atomes sont des entités constituées d’un certain nombre de protons (particules chargées d’électricité positive), d’électrons (chargés négativement) et de neutrons (neutres).
  - Le type d’atome (élément) est déterminé par le nombre de protons, qu’on appelle numéro atomique.

Héritage et polymorphisme

- Dans son état fondamental, un atome contient autant d’électrons que de protons et, par conséquent, il est électriquement neutre.

- Il possède également un nombre variable de neutrons mais ceux-ci n’influencent pas la charge électrique globale.

- Dans certaines circonstances, un atome peut gagner ou perdre des électrons.

Héritage et polymorphisme

- Il acquiert de ce fait une charge électrique globale et devient alors un ion (ion négatif s'il a gagné des électrons, ion positif s'il en a perdu).

- La charge électrique d'un ion est égale à la différence entre le nombre de protons et le nombre d'électrons qu'il contient.

Héritage et polymorphisme

- Le script suivant génère des objets Atome( ) et Ion( ).
- Un ion étant un atome modifié, la classe qui définit les objets Ion( ) sera une classe dérivée de la classe Atome( ) : elle héritera d'elle tous ses attributs et toutes ses méthodes en y ajoutant les siennes propres.

Héritage et polymorphisme

- L'une de ces méthodes ajoutées (la méthode affiche( )) remplace une méthode de même nom héritée de la classe Atome( ).

- Les classes Atome( ) et Ion( ) possèdent donc chacune une méthode de même nom, mais qui effectue un travail différent.

- On parle dans ce cas de polymorphisme.

- On dit également que la méthode affiche( ) de la classe Atome( ) est surchargée.

Héritage et polymorphisme

- Il est possible d'instancier un nombre quelconque d'atomes et d'ions à partir de ces deux classes.

- Or, l'une d'entre elles, la classe Atome( ), doit contenir une version simplifiée du tableau périodique des éléments (tableau de Mendeleev), de façon à pouvoir attribuer un nom d'élément chimique, ainsi qu'un nombre de neutrons, à chaque objet généré.

- Comme il n'est pas souhaitable de recopier ce tableau dans chacune des instances, on le placera dans un attribut de classe.

- Ainsi, ce tableau n'existera qu'en un seul endroit en mémoire, tout en restant accessible à tous les objets produits à partir de cette classe.

Héritage et polymorphisme

```
class Atome:
    """Atomes simplifiés choisis parmi les 10 premiers éléments du TPE"""
    table = [None, ('hydrogène',0), ('hélium',2), ('lithium',4), ('béryllium',5),
             ('bore',6), ('carbone',6), ('azote',7), ('oxygène',8), ('fluor',10),
             ('néon',10)]

def __init__(self, nat):
    "le n° atomique détermine le nombre de protons, d'électrons et de neutrons"
    self.np, self.ne = nat, nat
    self.nn = Atome.table[nat][1]


def affiche(self):
    print()
    print("Nom de l'élément : ", Atome.table[self.np][0])
    print("{0} protons, {1} électrons, {2} neutrons".format(self.np,self.ne,self.nn))
```

Héritage et polymorphisme

```
class Ion(Atome):

    """Les ion sont des atomes qui ont perdu ou gagné des électrons"""


    def __init__(self, nat, charge):
        "le n° atomique et le charge électrique déterminent l'ion"
        Atome.__init__(self,nat)
        self.ne = self.ne - charge
        self.charge = charge


def affiche(self):
    Atome.affiche(self)
    print("Particule électrisée. Charge = ", self.charge)
```

Héritage et polymorphisme

```
a1 = Atome(5)

a2 = Ion(3,1)

a3 = Ion(8,-2)

a1.affiche()

a2.affiche()

a3.affiche()
```

Héritage et polymorphisme

- Résultat :

```
Nom de l'élément :  bore

5 protons, 5 électrons, 6 neutrons
```

```
Nom de l'élément :  lithium

3 protons, 2 électrons, 4 neutrons

Particule électrisée. Charge =  1
```

```
Nom de l'élément :  oxygène

8 protons, 10 électrons, 8 neutrons

Particule électrisée. Charge =  -2
```

Héritage et polymorphisme

- On instancie les objets Atome( ) en fournissant leur numéro atomique (qui doit être compris entre 1 et 10).

- Pour instancier les objets Ion( ), on doit fournir un numéro atomique et une charge électrique globale (positive ou négative).

- La même méthode affiche( ) fait apparaître les propriétés de ces objets, qu'il s'agisse d'atomes ou d'ions, avec dans le cas de l'ion une ligne supplémentaire (polymorphisme).

Héritage et polymorphisme

- La définition de la classe Atome( ) commence par l'assignation de la variable table.

- Une variable définie à cet endroit fait partie de l'espace de noms de la classe. C'est donc un attribut de classe dans lequel nous plaçons des informations concernant les dix premiers éléments du TPE.

- Comme il n'existe pas d'élément de numéro atomique zéro, on a placé à l'indice zéro dans la liste l'objet spécial *None*. On aurait pu placer à cet endroit n'importe quelle autre valeur, puisque cet indice ne sera pas utilisé.

Héritage et polymorphisme

- Notons que la méthode constructeur de la classe Ion( ) fait appel à la méthode constructeur de sa classe parente, dans :

```
Atome.__init__(self,nat)
```

- Cet appel est nécessaire afin que les objets de la classe Ion( ) soient initialisés de la même manière que les objets de la classe Atome( ).

- Si nous n'effectuons pas cet appel, les objets-ions n'hériterons pas automatiquement des attributs np, ne et nn, car ceux-ci sont des attributs d'instance créés par la méthode constructeur de la classe Atome( ), et celle-ci n'est pas invoquée automatiquement lorsqu'on instancie des objets d'une classe dérivée.

Héritage et polymorphisme

- Insistons sur le fait que l'héritage ne concerne que les classes et non les instances de ces classes.

- Lorsqu'on dit qu'une classe dérivée hérite de toutes les propriétés de sa classe parente, cela ne signifie pas que les propriétés des instances de la classe parente sont automatiquement transmises aux instances de la classe fille. Par conséquent :

- *Dans la méthode constructeur d'une classe dérivée, il faut le plus souvent prévoir un appel à la méthode constructeur de sa classe parente.*

Modules contenant des bibliothèques de classes

- Les modules Python servent à regrouper des bibliothèques de classes et de fonctions.

- On va créer un module de classes, en encodant les lignes d'instruction ci-après dans un fichier-module nommé *formes.py*.

Modules contenant des bibliothèques de classes

```
class Rectangle(object):

    "Classe de rectangles"

    def __init__(self, longueur = 0, largeur = 0):

        self.L = longueur

        self.l = largeur

        self.nom = "rectangle"

    def perimetre(self):

        return "({0:d}+{1:d})*2 = {2:d}".format(self.L, self.l, (self.L+self.l)*2)

    def surface(self):

        return "{0:d}*{1:d} = {2:d}".format(self.L, self.l, self.L*self.l)

    def mesures(self):

        print("Un {0} de {1:d} sur {2:d}".format(self.nom,self.L,self.l))

        print("a une surface de {0}".format(self.surface()))

        print("et un périmètre de {0}\n".format(self.perimetre()))
```

Modules contenant des bibliothèques de classes

```
class Carre(Rectangle):

    "Classe de carrés"

    def __init__(self,cote):

        Rectangle.__init__(self, cote, cote)
        self.nom = "carré"


if __name__ == "__main__":

    r1 = Rectangle(15,30)

    r1.mesures()

    c1 = Carre(13)

    c1.mesures()
```

Modules contenant des bibliothèques de classes

- Une fois ce module enregistré, on peut l'utiliser de deux manières : soit on en lance l'exécution comme celle d'un programme ordinaire, soit on l'importe dans un script quelconque ou depuis la ligne de commande pour en utiliser les classes.

- Exemple :

Modules contenant des bibliothèques de classes

```
import formes

f1 = formes.Rectangle(27,12)

f1.mesures()
```

- Résultat :

```
Un rectangle de 27 sur 12

a une surface de 27*12 = 324

et un périmètre de (27+12)*2 = 78
```

Modules contenant des bibliothèques de classes

```
f2 = formes.Carre(13)

f2.mesures()
```

- Résultat :

```
Un carré de 13 sur 13

a une surface de 13*13 = 169

et un périmètre de (13+13)*2 = 52
```

Modules contenant des bibliothèques de classes

- La classe Carre( ) est construite par dérivation à partir de la classe Rectangle( ) dont elle hérite de toutes les caractéristiques. En d'autres termes, la classe Carre( ) est une classe fille de la classe Rectangle( ).

- L'instruction

```
if __name__ == "__main__":
```

placée à la fin du module, sert à déterminer si le module est lancé en tant que programme autonome, auquel cas les instructions qui suivent sont exécutées, ou utilisé comme une bibliothèque de classes importées ailleurs, auquel cas cette partie du code est sans effet.

Bibliographie

- *An Introduction to Statistics with Python*, T. Halwanter, Springer 2016

- *Introduction to Machine Learning with Python,* A. C. Müller & S. Guido, O'Reilly 2017

- *Machine Learning : An Algorithmic Perspective*, 2nd ed., S. Marsland, CRC Press 2015.

- *Programmation en Python pour les Mathématiques*, 2ème éd., A. Casamayou-Boucau, P. Chauvin & G. Connan, Dunod 2016

- *Apprendre à programmer avec Python 3*, 3ème éd., G. Swinnen, Eyrolles 2016.

- *Informatique avec Python*, S. Bays, Ellipses 2017

- *Introduction à l'Informatique*, S. Bays, Ellipses 2017