Classification

Antoine Nongaillard

But Info – IUT – Université de Lille

R5.AB.12 – Séance 06

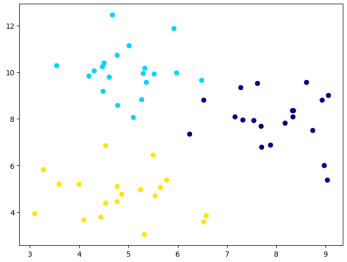
Plan

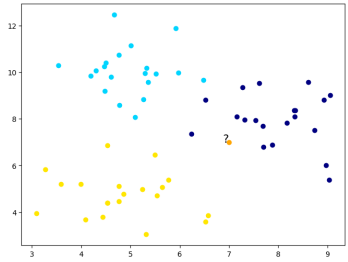
La tâche de classification

Avec la régression, la classification est la tâche principale de *Machine Learning* en apprentissage supervisé.

C'est associer une catégorie (ou un label) parmi un ensemble de labels possibles :

- **classification binaire** dans les cas les plus simple
- **classification multi-classes** autrement.





Les types de classification

Les catégories doivent être déterminées **avant** toute forme d'apprentissage. C'est de l'**apprentissage supervisé** : on doit disposer d'un ensemble de données associées à leur(s) catégorie(s).

On reçoit des données annotées $(x_1, y_1), (x_2, y_2), \ldots$ et on espère prédire la sortie sur de nouvelles observations : $x^\star \rightarrow y^\star$.

Dans la plus grande majorité des cas, les observations ne seront associées qu'à une seule classe. Mais il existe des cas particuliers :

- la **classification multi-labels** : chaque donnée peut être associée à plusieurs labels
- la **détection d'objets** sur des images : il faut reconnaître les objets présents et leur position
- la **segmentation** d'image : chaque pixel doit être associé à une classe (et généralement à une même couleur)

Exemples de classification

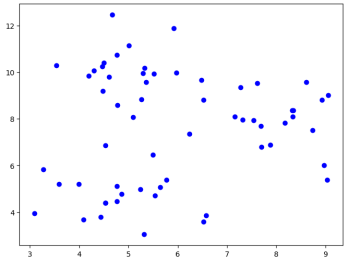
Même si le cas le plus courant (dans la littérature) est de déterminer si une image contient un chien ou un chat, la classification s'emploie dans de nombreux domaines plus réalistes et plus utiles au quotidien.

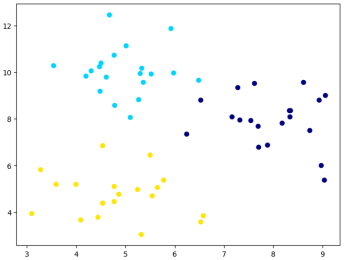
- Dans l'industrie :
  - détection de défauts de pièces
  - maintenance prédictive
- Sur les sites marchands :
  - association automatique d'une catégorie à un produit à partir de sa description
  - détection de produits frauduleux
- En sécurité :
  - détection de spams
  - identification de sites frauduleux
- Avec les objets connectés :
  - détection des états anormaux (maintenance ciblée)
  - détection des risques futurs (avalanche par exemple)

Classification ≠ Clustering

Le **clustering**, aussi appelé *partitionnement des données*, consiste à créer des clusters. Chaque cluster est un ensemble de données regroupées selon leur similitudes.

Contrairement à une classification, les classes ne sont pas connus à l'avance. C'est à l'algorithme de les déterminer mathématiquement. C'est de l'**apprentissage non supervisé**.

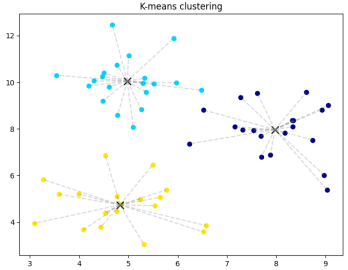


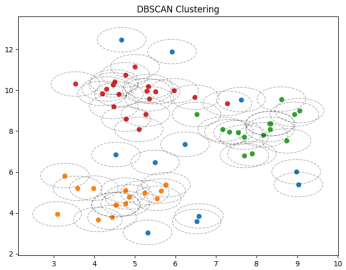


Classification ≠ Clustering

Il existe différents algorithmes qui visent à regrouper les observations en un nombre de classes donné, basés sur :

- la notion de distance : *k-means*
- la notion de densité : *DBSCAN*





Exemples de Clustering

L'utilisation la plus courante reste l'analyse de données clients, de manière à les segmenter en "profils types". Chaque profil peut recevoir des offres dédiées.

Mais on retrouve le clustering également dans :

- la segmentation des produits : on catégorise automatiquement les produits afin d'identifier des produits similaires
- la classification des usages : ça permet de différencier des comportements types ou bien potentiellement des usages frauduleux
- la sélection des publicités ciblées : on regroupe les publicités par similitude et on propose les publicités appartenant aux mêmes clusters que celles déjà vues
- la détection d'anomalie : on crée des clusters et on marque comme anomalie toute donnée s'éloignant des clusters existants
- la création de matching entre personnes : sur les réseaux sociaux, à partir des similitudes entre les caractéristiques des personnes

Les algorithmes de classification

Il existe de nombreux algorithmes de base pour faire de la classification (et de très nombreuses variantes, satisfaisant des contraintes particulières).

- les arbres de décision
- **random forests**
- eXtreme Gradient Boosting : *XGBoost*
- **K-Nearest Neighbors**
- **la régression logistique**
- *Naive Bayes*, basée sur les probabilités conditionnelles
- *Support Vector Machine*, basée sur les hyperplans et les données linéairement séparables par augmentation de la dimension
- ...

Il faut garder en tête qu'une grande majorité des algorithmes de classification peuvent être utilisés pour faire de la régression, moyennant quelques adaptations.

Plan

La matrice de confusion

En classification, la matrice de confusion est le principal indicateur de la qualité d'un modèle : c'est un tableau à double entrée mettant en correspondance les classes réelles et les classes prédites par le modèle.

Dans le cas d'une classification binaire, on obtient :

	Préd. 0	Préd. 1	TOTAL
Réel 0	Vrai Négatif (VN)	Faux Positif (FP)	XXX
Réel 1	Faux Négatif (FN)	Vrai Positif (VP)	XXX
TOTAL	XXX	XXX	$\lvert\mathcal{O}bs\rvert$

Les faux négatifs (type II) sont plus graves en général que les faux positifs (type I), en médecine par exemple.

La matrice de confusion

Dans le cas d'une classification multi-classes, on obtient :

	Pred. A	Pred. B	Pred. C	TOTAL
Réel A	30	3	5	38
Réel B	8	25	10	43
Réel C	10	2	20	32
TOTAL	40	30	35	113

L'interprétation est la même : 8 observations de la classe B ont été classifié comme A. À partir de cette matrice, on crée souvent des `tables de confusion` (1 par classe) :

	Pred. Autre	Pred. B	TOTAL
Réel Autre	65	5	70
Réel B	18	25	43
TOTAL	83	30	113

Tous les indicateurs présentés dans la suite doivent être calculés sur ces tables !

```
sklearn.metrics.confusion_matrix(test_y, pred_y)
```

Comparer deux modèles à partir de leur matrice de confusion est complexe. Pour aider à la tâche, des indicateurs dérivés de cette matrice aident :

L'**accuracy** est la proportion de prédictions qui tombe juste. Sa valeur est toujours comprise entre 0 et 1.

$$Accuracy = \frac{VP + VN}{|\mathcal{O}bs|}$$

Elle a des défauts :

- Aucune information sur le type d'erreurs commises
- Peu informatif si les classes sont déséquilibrées.
- Une accuracy de 100% n'est jamais un objectif. Une valeur trop élevée peut cacher un sur-apprentissage, ou la prise en compte d'informations non voulues (comme un index par exemple)

Le **rappel** (ou *recall*) est la proportion de la classe positive détectée. Sa valeur est toujours comprise entre 0 et 1. Un fort rappel indique que presque tous les cas de la classe positive ont été détectés. Plus il est fort, moins il y a de faux négatifs.

$$Rappel = \frac{VP}{VP + FN}$$

La **précision** est la proportion des vrais positifs dans l'ensemble des positifs détectés. Sa valeur est toujours comprise entre 0 et 1. Elle permet d'évaluer le nombre de faux positifs.

$$\text{Précision} = \frac{VP}{VP + FP}$$

Les autres métriques

F_1-score et F_β-score

Le but du F_1**-score** est de fournir un indicateur unique, qui prend en compte à la fois le rappel et la précision, mais sans tomber dans les biais de l'accuracy. C'est la moyenne harmonique de la précision et du rappel. Sa valeur est toujours comprise entre 0 et 1.

$$F_1 = \frac{2 \times \text{précision} \times \textit{rappel}}{\text{précision} + \textit{rappel}}$$

Comme l'impact des types d'erreur n'est souvent pas le même, on calcule le F_β**-score**, avec β qui représente le coût relatif des erreurs de type II par rapport au type I. Un score F_2-score signifie qu'une erreur de type II est deux fois plus coûteuse qu'une erreur de type I.

$$F_\beta = \frac{(1 + \beta^2) \times \text{précision} \times \textit{rappel}}{(\beta^2 \times \text{précision}) + \textit{rappel}}$$

Comme toute agrégation, il y a une perte d'information : l'utilisation du F_1-score ne fait pas l'unanimité.

Les autres métriques

Sensibilité et spécificité

La **sensibilité** évalue la capacité du modèle à prédire la classe positive quand c'est effectivement le cas.

$$\text{sensibilité} = \frac{VP}{VP + FN}$$

La **spécificité** permet de mesurer la capacité du modèle à donner un résultat négatif pour la classe négative.

$$\text{spécificité} = \frac{VN}{VN + FP}$$

- Un modèle complètement aléatoire vérifiera *sensibilité + spécificité = 1*.
- Si un modèle a un des deux indicateurs faibles, le résultat correspondant n'pas fiable : une sensibilité faible ne permet pas de décider sur un résultat positif alors qu'une spécificité faible ne permet pas de décider sur un résultat négatif.

Les autres métriques

En pratique

Le calcul de ces métriques d'évaluation se fait facilement avec Python :

```
sklearn.metrics.accuracy_score(test_y, pred_y)
sklearn.metrics.recall_score(test_y, pred_y, average='macro')
sklearn.metrics.precision_score(test_y, pred_y, average='macro')
sklearn.metrics.f1_score(test_y, pred_y, average='macro')
sklearn.metrics.fbeta_score(test_y, pred_y, beta=2, average='macro')
```

L'option `average` est facultative pour la classification binaire, mais devient incontournable pour une classification multi-classes. Il faut choisir comment l'agrégation se fait sur chacune des classes :

- `micro` : calcule la métrique au niveau global (en prenant le nombre total de FP et de FN)
- `macro` : le calcul se fait par classe puis agrégé via une moyenne arithmétique
- `weighted` : l'agrégation est pondéré par l'importance des classes (en nombre d'exemple)

Plan

La régression logistique

La régression logistique, comme son nom ne l'indique pas, est une technique de classification, et non de régression. Elle est binaire par défaut et dite polytomique dans le cas multi-classes.

Le nom "logistique" vient du terme "logit", qui est le logarithme de la probabilité qu'un événement se produise. En d'autres termes, le modèle de régression logistique prédit $P(Y = 1)$ en fonction de X.

Elle repose sur deux phases :

- Création d'une fonction linéaire des variables, qui associe des nombres positifs aux données de la classe positive, et des nombres négatifs aux données de la classe négative
- Application de la fonction logistique sur le résultat pour ramener la sortie à une valeur entre 0 et 1

La fonction linéaire s'écrit sous la forme :

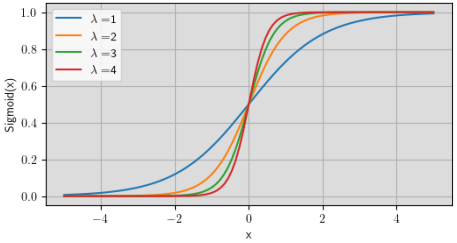
$$c = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \dots$$

La fonction logistique s'écrit ensuite :

$$y = \frac{e^c}{1 - e^c}$$

La régression logisitique

`y` représente la probabilité pour un élément d'être de la classe positive. Pour les valeurs de `c` négatives, `y` sera inférieur à 0.5, et c'est donc la classe négative qui sera prédite. À l'inverse, pour les valeurs positives, c'est la classe positive qui sera prédite.



Comme les coefficients dépendent de l'échelle des données d'entrée, il est plus que fortement conseillé de normaliser les variables avant d'utiliser la régression logistique.

La régression logistique polytomique est basé sur le même principe, mais le nombre de paramètres augmente avec le nombre de classes :

$$Nb\_param = (Nb\_variables + 1) \times Nb\_classes$$

La régression logistique

En pratique

Dans le *dataset Iris*, chaque observation contient 4 valeurs (2 caractéristiques sur les pétales, 2 autres sur les sépales) et 3 classes sont possibles en sortie.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

iris = load_iris()
x = iris.data
y = iris.target
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1)
clf = LogisticRegression()
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
```

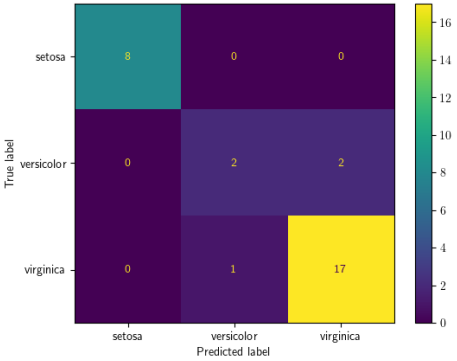
On obtient :

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

Pour l'interprétation des résultats, Python nous aide beaucoup :

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

print("nb de tests ", y_test.size)
cm = confusion_matrix(y_test, y_pred)  #, normalize='all')
cmd = ConfusionMatrixDisplay(cm, display_labels=iris.target_names)
cmd.plot()
```

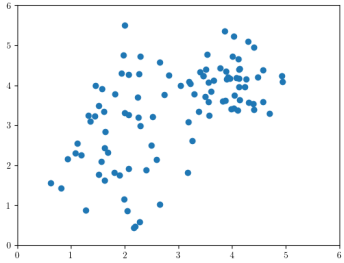


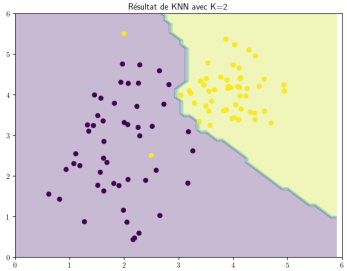
Plan

K-Nearest Neighbors

C'est une méthode particulière qui ne crée par de modèle à proprement parler. L'ensemble des données d'apprentissage constitue le modèle.

Chaque donnée est classifié selon un vote à la majorité selon la classe de ses K plus proches voisins.



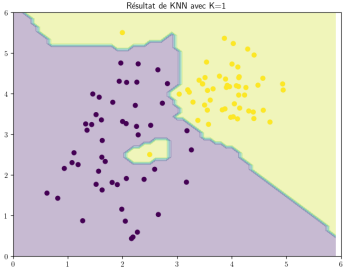


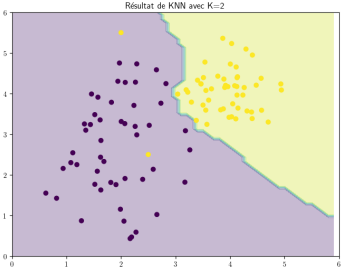
C'est un algorithme déterministe : tous les apprentissages sur un même ensemble de données fournissent les mêmes résultats.

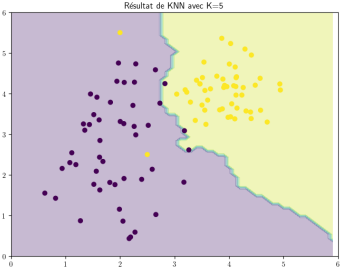
K-Nearest Neighbors

Le choix de l'hyper-paramètre `K` est important :

- Si `K` est trop faible, peu de voisins sont sélectionnés, l'algorithme sera très sensible à des cas particulier (sur-apprentissage).
- Avec un `K` trop grand, les frontières entre les classes auront tendance à s'effacer/se lisser.





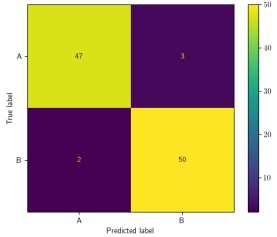


Le calcul des distances est sensible à l'amplitude des données : une normalisation est souvent indispensable.

K-Nearest Neighbors

En pratique

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# pour l'entraînement
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
# pour les prédictions
pred = knn.predict(X)
# calcul de la matrice de confusion
conf_matrix = confusion_matrix(y, pred)
cmd = ConfusionMatrixDisplay(conf_matrix, display_labels=['A', 'B'])
cmd.plot()
# calcul des autres indicateurs
print(sklearn.metrics.classification_report(y, pred))
```

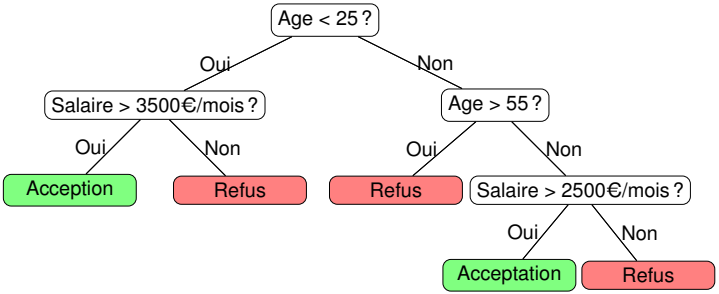


Class	Precision	Recall	F1-score	Support
A	0.96	0.94	0.95	50
B	0.94	0.96	0.95	52
Accuracy			0.95	102
Macro avg	0.95	0.95	0.95	102
Weighted avg	0.95	0.95	0.95	102

Plan

Les arbres de décision

Voici un exemple d'arbre de décision pour l'accord de crédit à des clients.

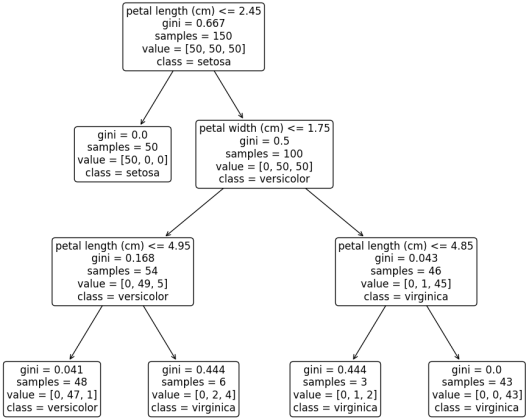


Pléthore d'algorithmes existent pour générer des arbres : ID3, C4.5, C5.0, CART, CHAID mais leur principe est toujours le même. L'algorithme choisit quelle variable il doit utiliser en se basant sur son "impureté", caractérisé par l'indice de **Gini** et l'**entropie**.

Les arbres de décision

En pratique

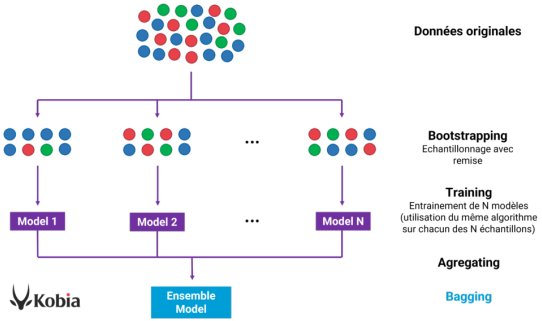
```
iris = load_iris()
model = tree.DecisionTreeClassifier(max_depth=3)
model = model.fit(iris.data, iris.target)
t = tree.plot_tree(model, feature_names=iris['feature_names'], class_names=iris.target_names)
```



Bagging

Les arbres de décisions ont de nombreux avantages, mais sont souvent trop simple pour résoudre des problèmes un peu complexe avec de bons résultats.

Afin d'améliorer tout cela, on utilise une méthode de bagging [1] :



1. https://kobia.fr/

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import precision_recall_fscore_support as score

alg = RandomForestClassifier(n_estimators=1, max_depth=20, n_jobs=-1)
model = alg.fit(x_train, y_train)

pred = model.predict(x_test)
precision, recall, fscore, support = score(y_test, pred, pos_label=1, average='binary')
```

Les paramètres importants sont les suivants :

- `n_estimators` permet d'indiquer le nombre d'arbres de décision à utiliser
- `max_depth` représente la profondeur maximale que peuvent atteindre les arbres
- `n_jobs` permet la parallélisation des calculs