

# Machine learning : généralités

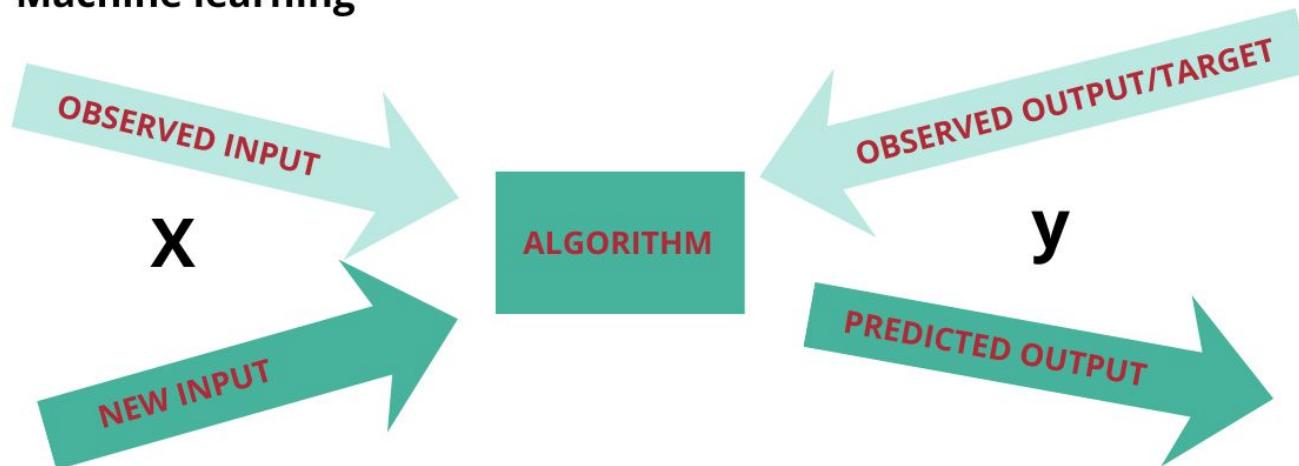
Master Data / IA 2025-26

# Un paradigme « data-driven »

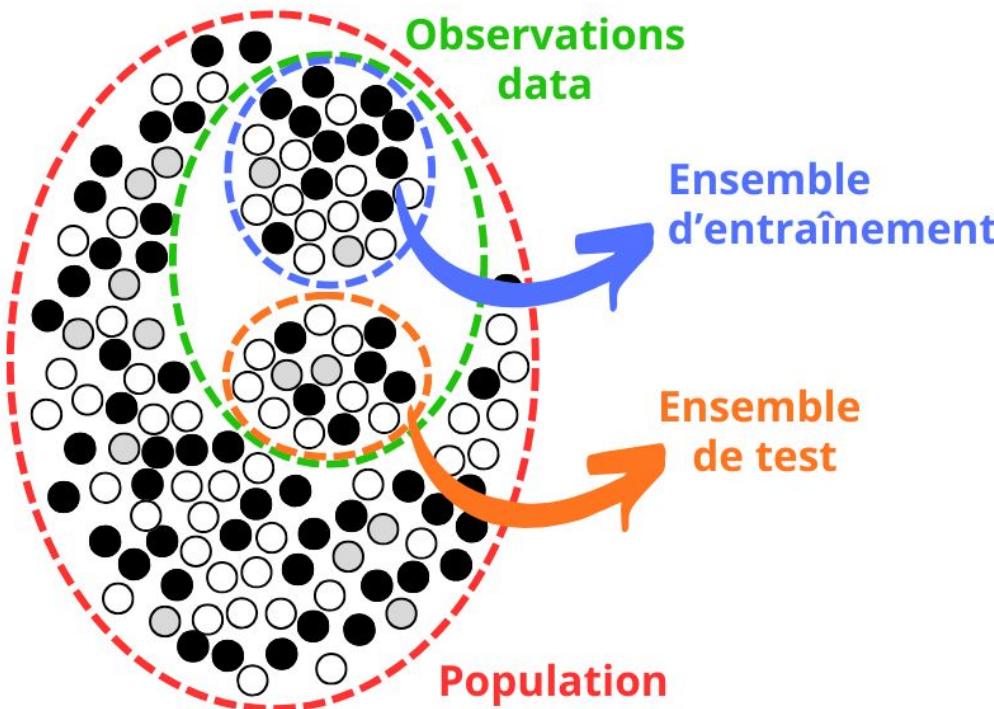
Algorithme déterministe



Machine learning



# Apprentissage

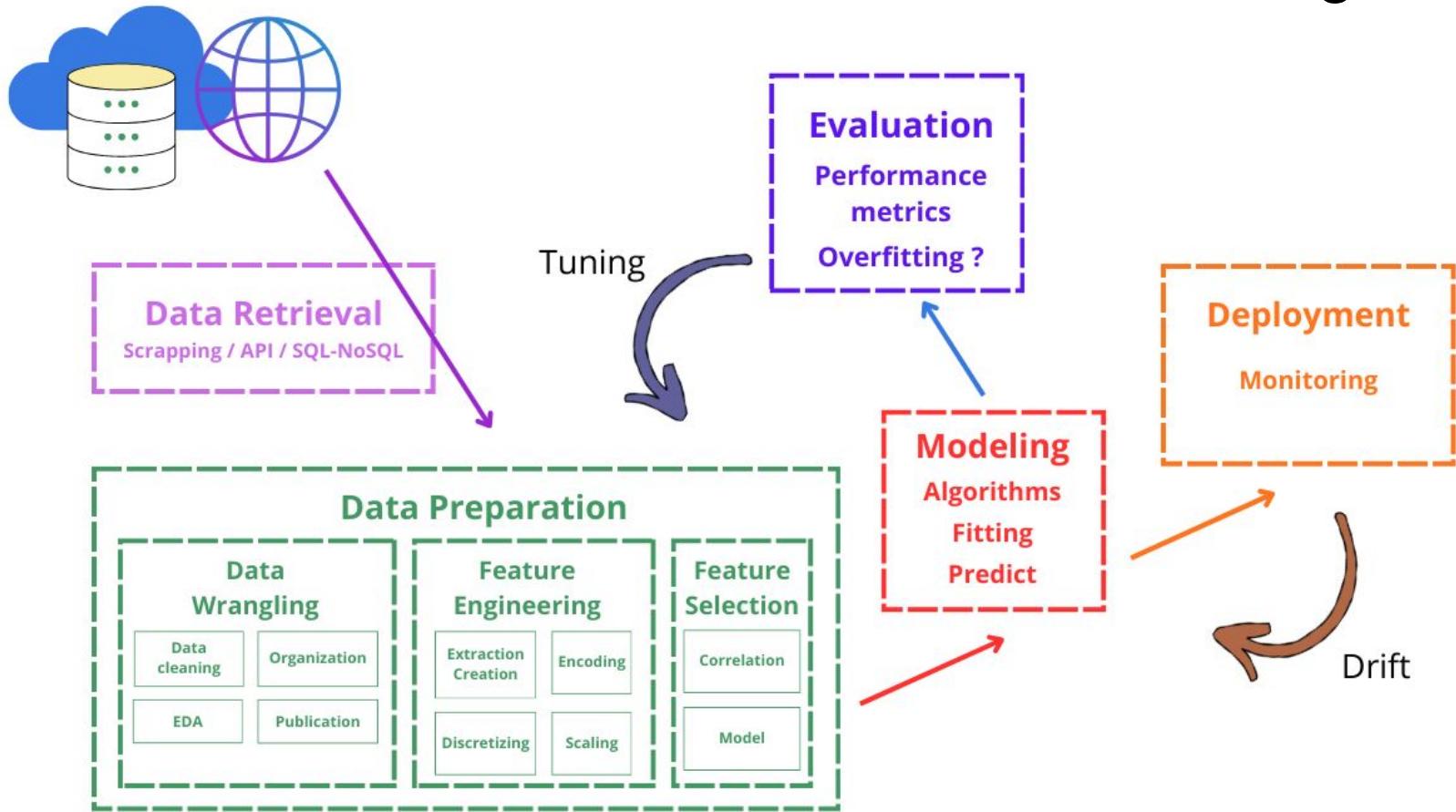


- Généralisation
  - Biais d'apprentissage
  - Overfitting : « apprendre par cœur »
- « Architecture » (modèle)
- Estimateur (algorithme)
- Paramètres  $\beta_i$
- Mesure de l'erreur  $\epsilon$
- Ensemble d'apprentissage
  - Suffisant (taille, fiabilité, balancing,...)
  - Congruent avec l'objectif / la tâche visée
  - Enrichir
  - Labellisé
  - Data leakage

# Supervisé / Non-supervisé

- l'apprentissage non-supervisé apprend/découvre des structures implicites dans les données (ex. clustering)
- l'apprentissage supervisé apprend à partir de données labellisées à prédire une valeur cible (un vecteur  $y_i$ ) à partir de données généralement multivariées (matrice  $X$ ), en minimisant l'erreur de prédiction
  - $f_\beta(X) \rightarrow y$
  - $\min(\varepsilon)$

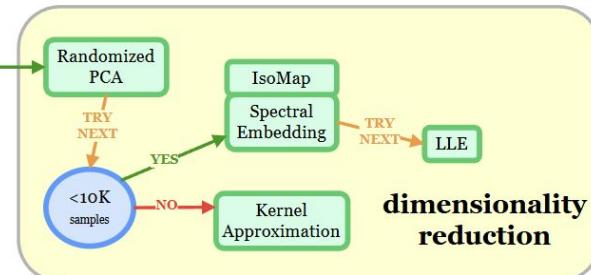
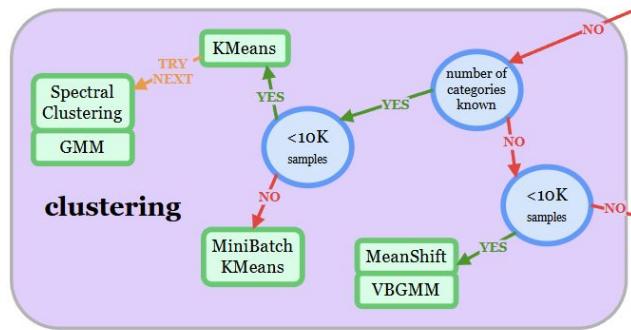
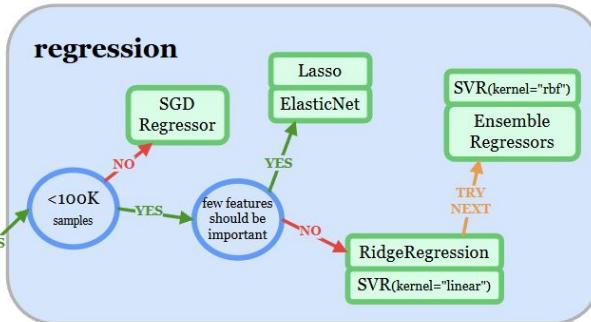
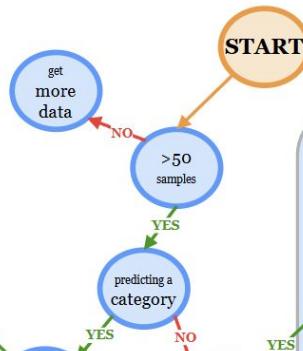
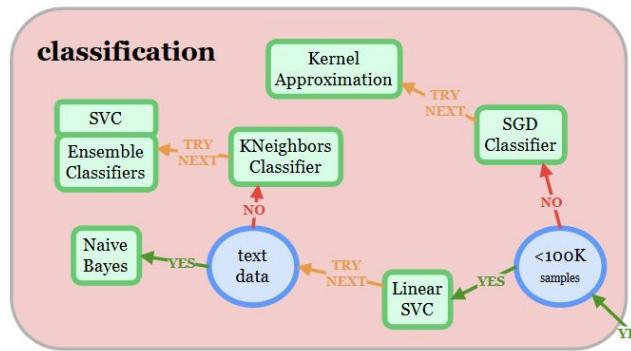
# Machine learning flow



# Types de modèles / types de tâches

scikit  
learn

algorithm cheat sheet



[https://scikit-learn.org/stable/machine\\_learning\\_map.html](https://scikit-learn.org/stable/machine_learning_map.html)

# Métriques

Pour guider l'estimation / ajustement des paramètres, il faut absolument une métrique de « l'erreur » à minimiser. Cette métrique dépend du modèle/de la tâche

- Pour k-means nous avons vu l'inertie
- Mean Square Error (MSE) est une métrique courante (ex. régression linéaire)
- Likelihood ou Log-likelihood (ex. régression logistique)
- Métriques particulières pour la catégorisation (précision, sensibilité, F1...)
- etc.

# Matrice de confusion

		Prédictions	
		Non	Oui
Réalité	Non		
	Oui		

**TN** **FP** **FN** **TP**

## Métriques : catégorisation

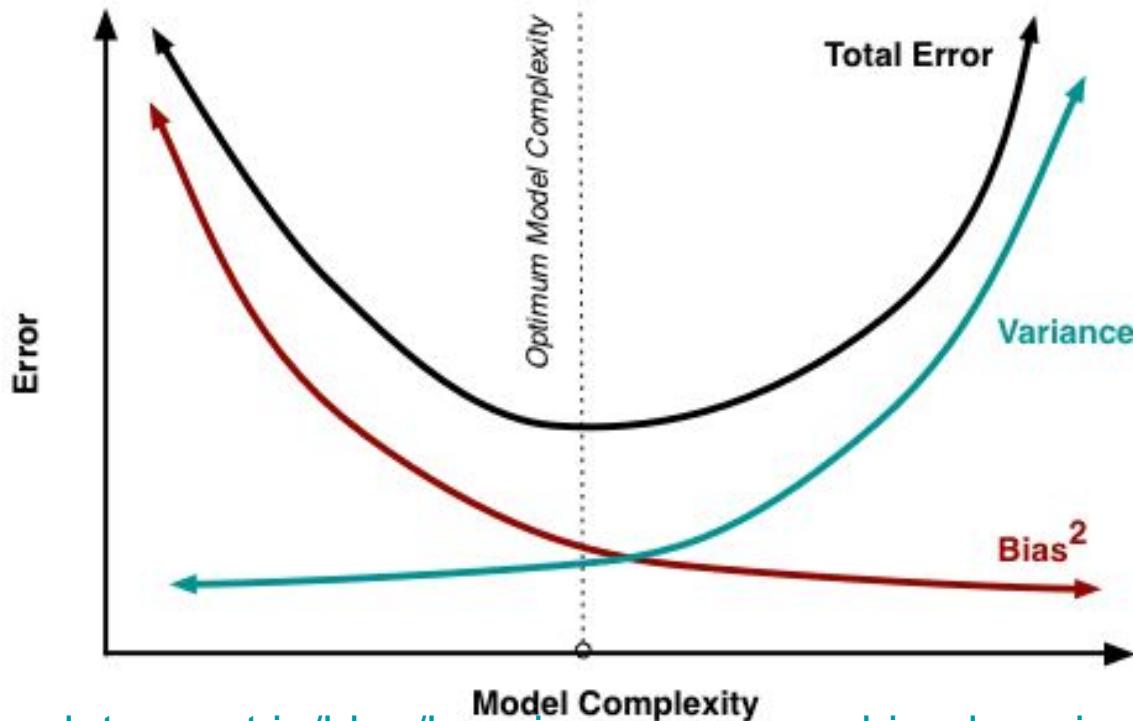
- Accuracy (précision)
  - exactitude réponses positives
  - $= TP / (TP + FP)$
- Recall (rappel ou sensibilité)
  - taux observations positive correctement détectées
  - $= TP / (TP + FN)$
- F1 (moyenne harmonique)
  - favorise les modèles avec rappel et précisions similaires
  - $= TP / (TP + (FN + FP)/2)$

# Courbes d'apprentissages

- Pour un training set donné, nous avons un modèle (paramètres) donné
- Nous avons donc une certaine variance de notre modèle
- Modèle = hypothèses sur la relation entre X et y (ex. régression linéaire)
- Hypothèse = biais
- Hypothèses erronées = variance
- Erreur = biais + variance
- On peut difficilement maintenir biais et variance simultanément petites :

Par ex. : biais faible = peu de généralisation = overfitting = augmentation variance

# Courbes d'apprentissages



# Courbes d'apprentissage

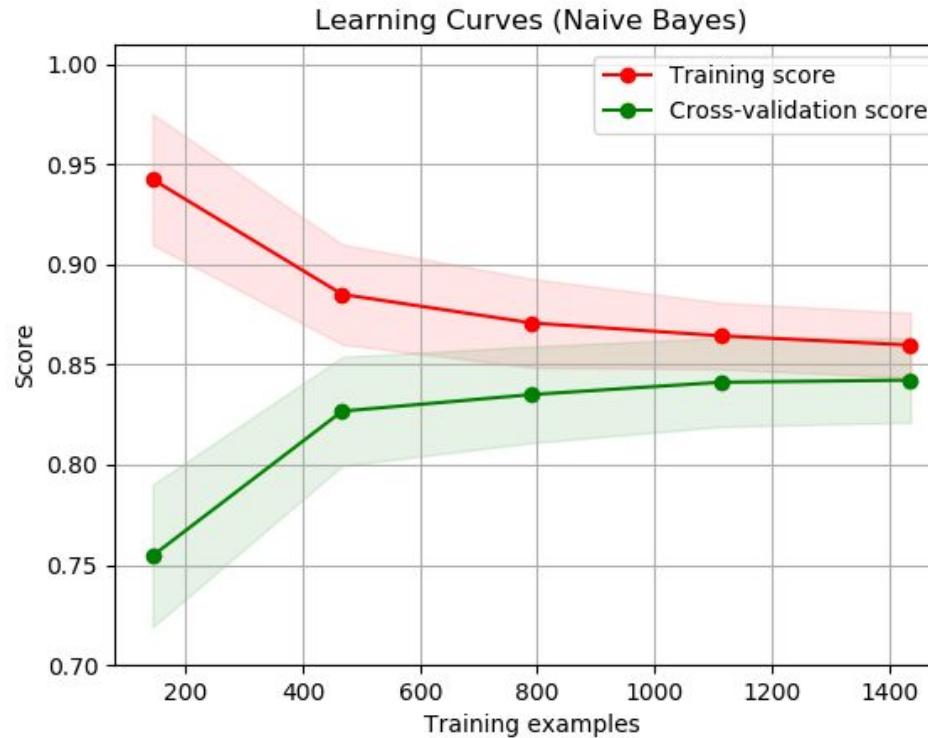
On peut mesurer l'erreur de notre modèle sur deux ensembles :

- entraînement
- test

On peut donc tracer l'évolution de l'erreur de notre modèle sur l'un ou l'autre ensemble en fonction de différentes grandeurs (en général taille de l'ensemble d'apprentissage, mais ça peut être le nombre d'epoch - « cycle » d'apprentissage, etc.)

La forme respective de ces courbes permettent de réaliser un diagnostic sur la capacité d'apprentissage du modèle

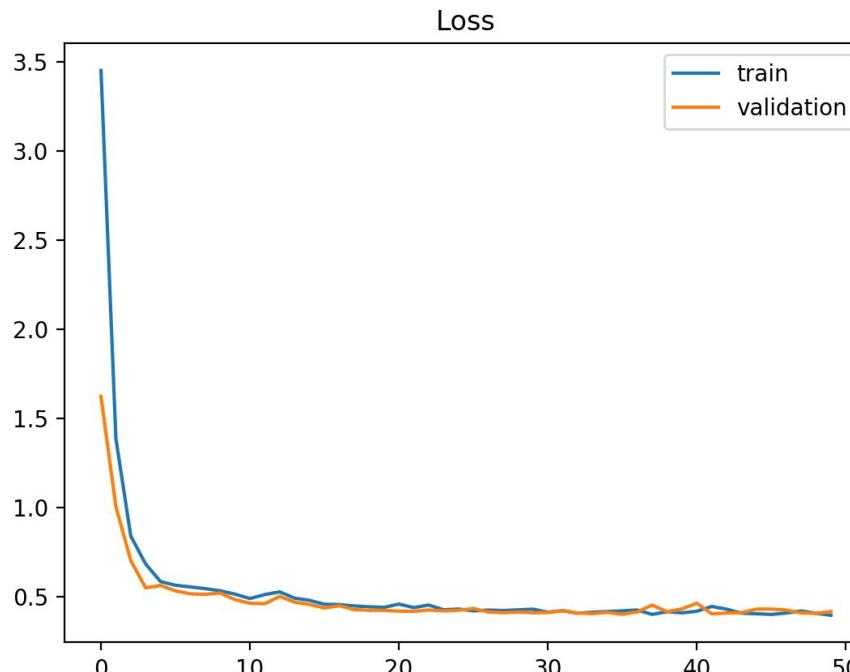
# Courbe d'apprentissage



By scikit-learn developers - [https://scikit-learn.org/stable/modules/learning\\_curve.html](https://scikit-learn.org/stable/modules/learning_curve.html), BSD,  
<https://commons.wikimedia.org/w/index.php?curid=76590125>

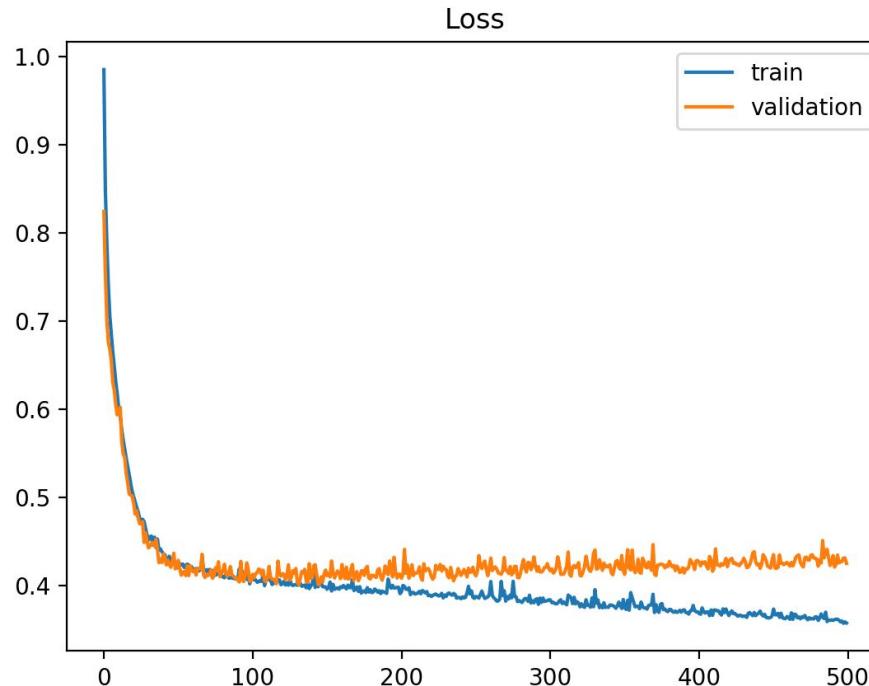
# Courbe d'apprentissage : diagnostic

## Courbes idéales



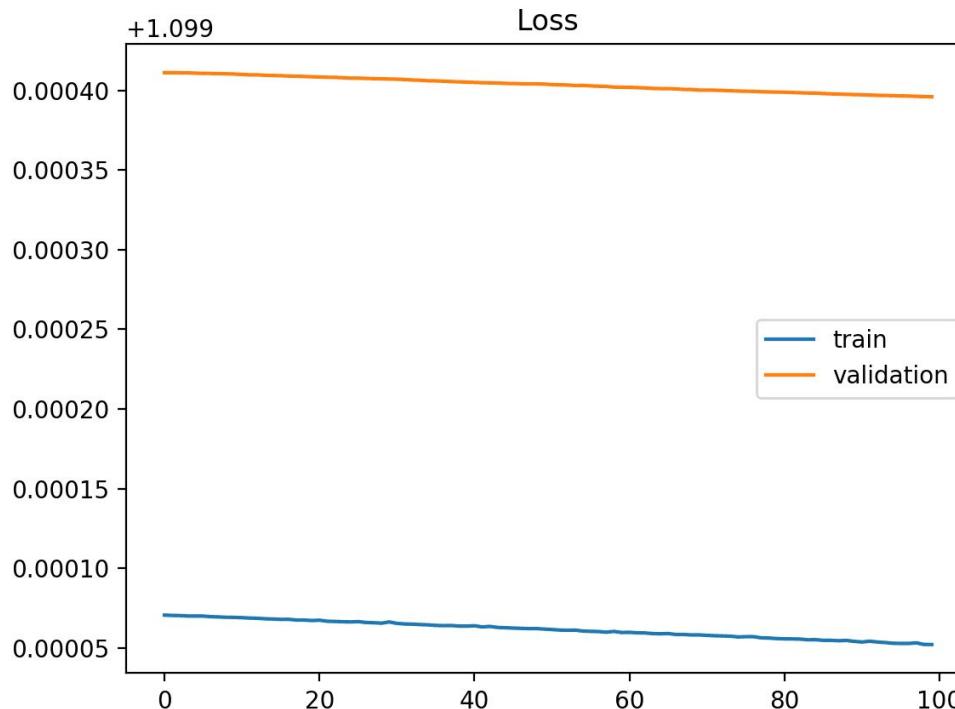
# Courbe d'apprentissage : diagnostic

Overfitting typique



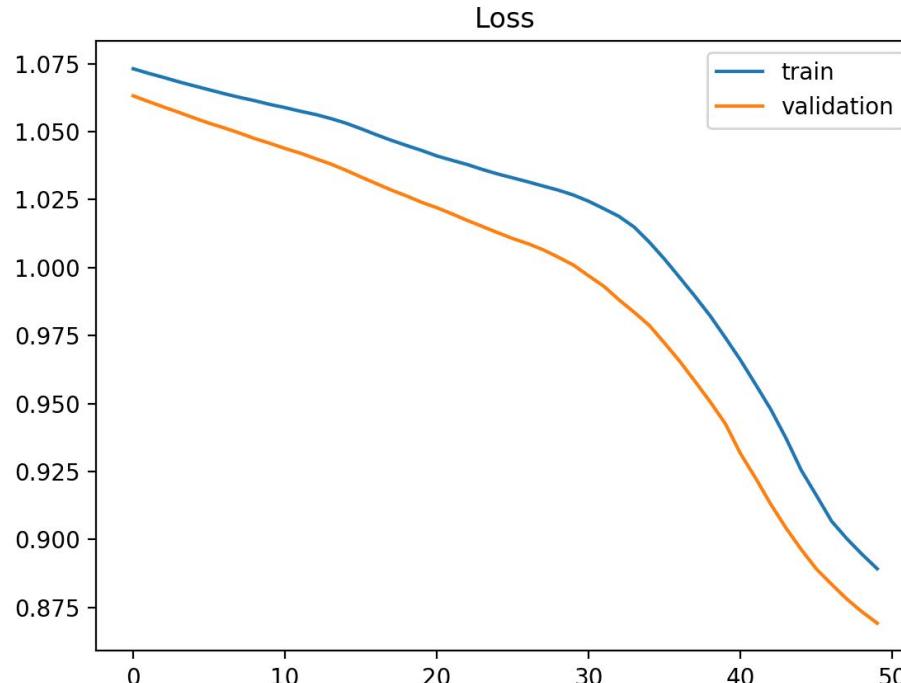
# Courbe d'apprentissage : diagnostic

Problème trop complexe pour le modèle



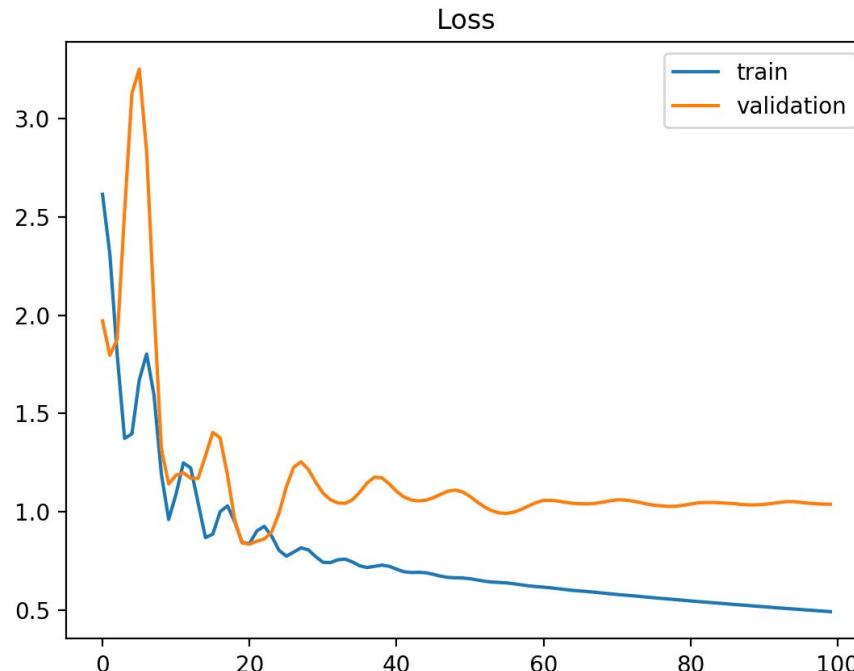
# Courbe d'apprentissage : diagnostic

Le modèle doit continuer à apprendre



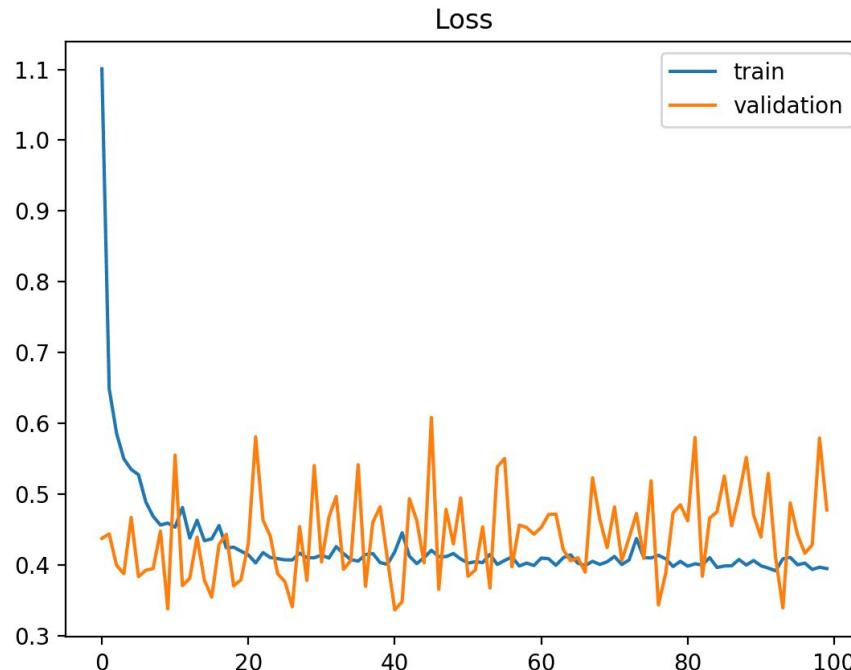
# Courbe d'apprentissage : diagnostic

Instabilité + overfitting : training set trop petit ?



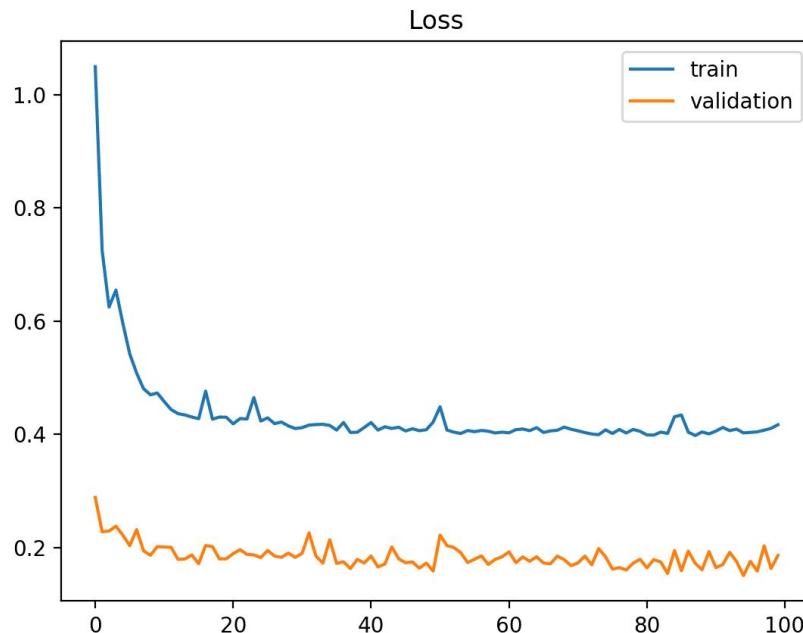
# Courbe d'apprentissage : diagnostic

Instabilité, test set pas adéquat ? trop petit ?



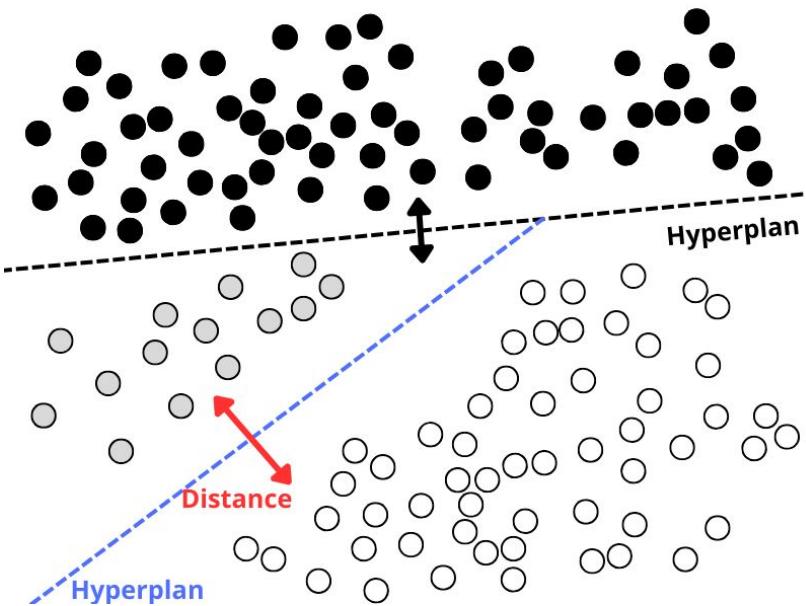
# Courbe d'apprentissage : diagnostic

Pas commun : test set pas adapté (trop facile ?)



# Exemple concret : SVM

Les **Support Vector Machines (Machines à Vecteurs de Support)** sont des algorithmes de classification supervisée très populaires en machine learning. Leur principe est de trouver la **frontière de décision optimale** (appelée hyperplan) qui sépare au mieux les différentes classes.



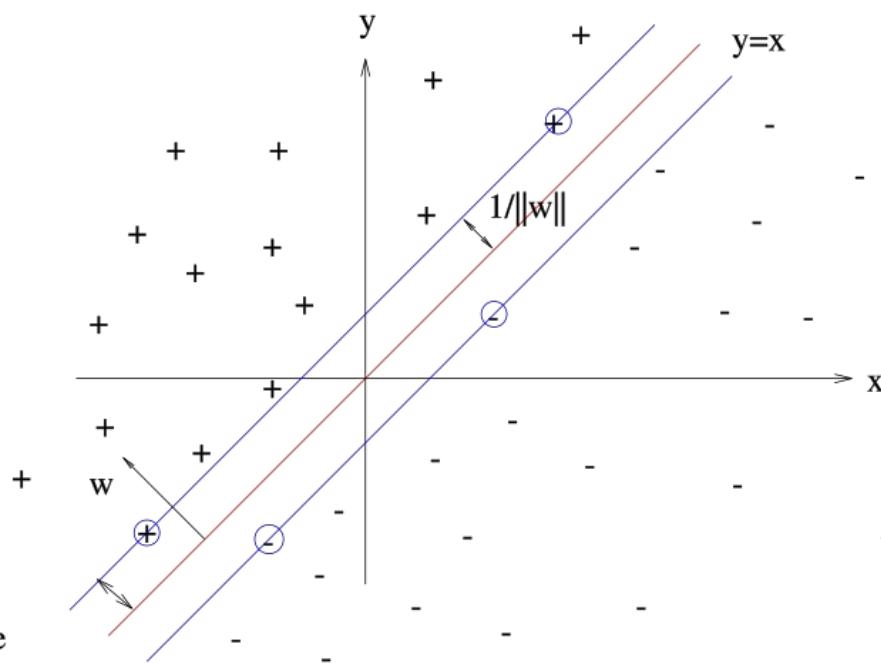
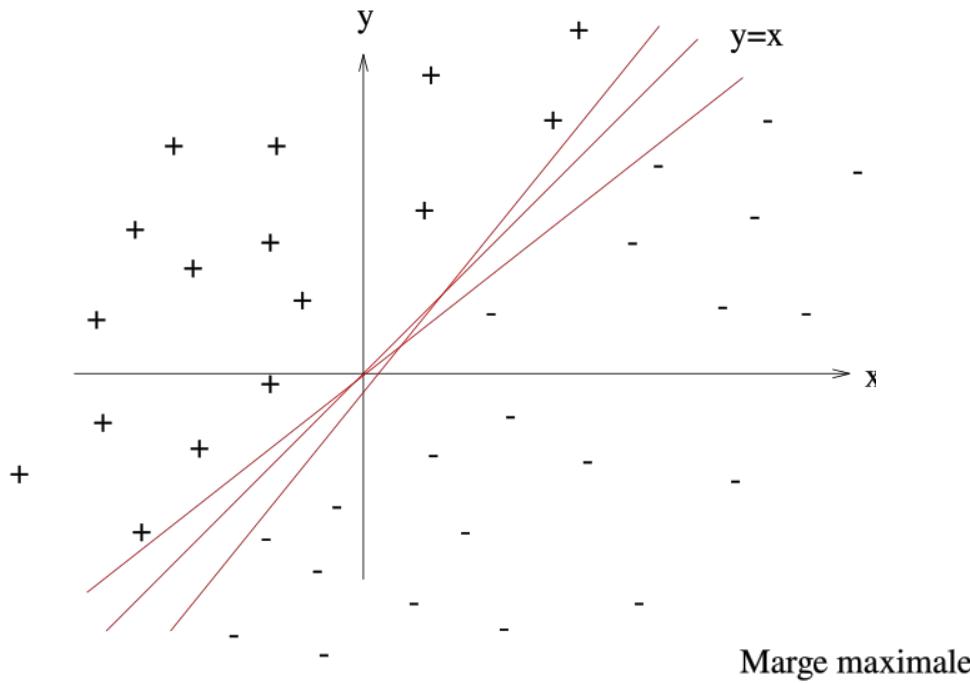
- Les éléments les plus proches des lignes sont les vecteurs de support
- On parle aussi de séparateur à vaste marge
- La marge prévient l'overfitting
- Efficaces en haute dimension
- Petits ensembles d'apprentissages
- Kernel : permet de gérer des séparations non linéaires

# Séparation linéaire

Il existe une infinité d'hyperplans solution d'un problème linéairement séparable

On sélectionne celui qui maximise la distance avec les vecteurs de support

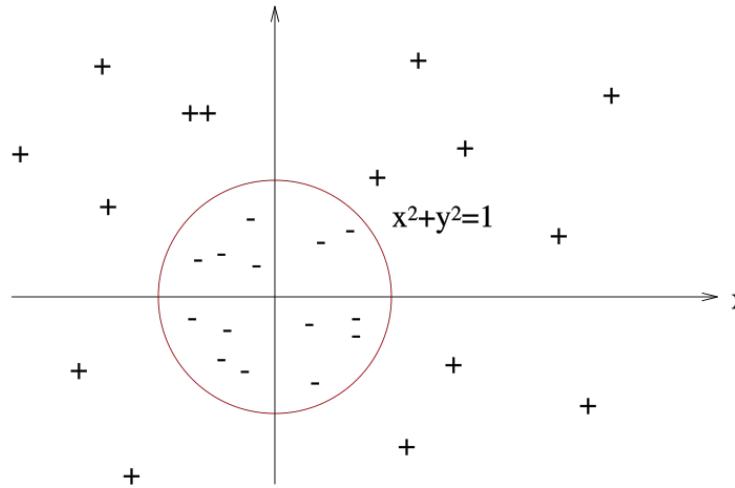
[https://fr.wikipedia.org/wiki/Machine\\_%C3%A0\\_vecteurs\\_de\\_support\\*](https://fr.wikipedia.org/wiki/Machine_%C3%A0_vecteurs_de_support)



# Kernel Trick

Pour les problèmes non-linéaires, on peut utiliser non plus une équation de droite, mais des équations de courbes plus ou moins exotiques pour réaliser la séparation.

Cela demande souvent de se placer dans un espace de dimension supérieure ou un autre système de coordonnées (ici coordonnées polaires)



# Implémentation Scikit-learn

3 implémentations des SVM :

- `svc()` Support Vector Classifier, pour les tâches de catégorisation
- `SVR()` Support Vector Regressor, pour des output numériques (ex. déduire l'âge à partir d'une photo)
- `LinearSVC()` SVC optimisée pour un noyau linéaire seulement (utile pour les problèmes linéaires avec grands ensemble de données)

# Scikit-Learn : SVC

```
svc(kernel='rbf', gamma='scale', C=1.0, random_state=42)
```

- **'rbf'** : Radial Basis Function ou noyau gaussien. Très utilisé.  
Formule :  $K(x, x') = \exp(-\gamma \|x - x'\|^2)$ . Comparez avec '**linear**'
- **'scale'** : fixe automatiquement  $\gamma = 1 / (\text{n\_features} \times \text{variance})$ 
  - $\gamma$  = portée des SV
  - $\gamma$  élevé : très précis / overfitting
  - $\gamma$  faible : plus global / sous apprentissage
- **C** : compromis entre marge et erreur.
  - C élevé : marge étroite (overfitting)
  - C faible : marge large (sous apprentissage)
- **random\_state** : fixe la graine aléatoire pour pouvoir reproduire les résultats

# Fonctionnement

- Entraînement :
  - le SVM analyse les images du training set (brute ou après ACP)
  - identifie les images les plus dures à classer -> vecteurs de support
  - calcule l'hyperplan qui maximise la marge entre les classes
- Prédiction :
  - nouvelle image : le SVM calcule de quel côté de l'hyperplan elle se situe
  - noyau RBF : mesure la similarité avec les vecteurs de support
  - attribue l'image à la classe la plus probable

# 'rbf' ou 'linear' ?

- Noyau linéaire : hypothèse que les classes sont séparables dans l'espace original
  - rapide
  - hypothèse forte / biais fort -> moins de chance d'overfitting
  - pas flexible du tout avec des données « bizarres » ou pour le moins complexes
- Noyau RBF : projette les données dans un espace de dimensions infinies
  - très flexibles, s'adapte à toutes sortes de formes -> risque d'overfitting
  - meilleur pour les images rarement analysables linéairement
  - lent (calculs)
  - bien régler C et γ

# Code du TP : train / test split avec Scikit-Learn

```
5 # Division en ensembles d'entraînement (80%) et de test (20%)
6 X_train, X_test, y_train, y_test = train_test_split(
7     X, y, test_size=0.2, random_state=42, stratify=y
8 )
9
```

Balancing :

```
15 # Vérification de la distribution des classes
16 print("\nDistribution des classes dans l'ensemble de test :")
17 unique_test, counts_test = np.unique(y_test, return_counts=True)
18 for name, count in zip(target_names[unique_test], counts_test):
19     print(f" {name:30s} : {count:2d} images")
```

# Code du TP : SVC avec Scikit-Learn

Instanciation du modèle et apprentissage/ajustement des paramètres (fit)

```
9 clf_original = SVC(kernel='rbf', gamma='scale', C=1.0, random_state=42)
10 clf_original.fit(X_train, y_train)
11
```

Prédictions et qualité des prédictions

```
14 # Prédiction et évaluation
15 y_pred_original = clf_original.predict(X_test)
16 accuracy_original = accuracy_score(y_test, y_pred_original)
17
```

# Aller plus loin :

## **GridSearch et CrossValidation**

- Nous avons vu précédemment l'entraînement de modèles sur des échantillons différents entraînait une variance
- Ainsi si on fait un entraînement sur un seul échantillon, il y a le risque d'avoir de bons résultats seulement sur l'échantillon (overfitting) car le hasard a bien fait les choses
- Limite les moyens d'optimiser notre apprentissage car on va sans cesse réutiliser le même échantillon pour l'évaluation/test (overfitting++)

# Validation croisée K-fold

	Data				
	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Itération 1	Test	Train	Train	Train	Train
Itération 2	Train	Test	Train	Train	Train
Itération 3	Train	Train	Test	Train	Train
Itération 4	Train	Train	Train	Test	Train
Itération 5	Train	Train	Train	Train	Test

- test rapide :  $k = 3-5$
- compromis (standard) :  $k = 10$
- leave-one-out : on fait le test sur une seul observation à chaque itération

- On divise la data en  $k=5$  parties égales (folds)
- On itère sur  $k=i$  :
  - modèle entraîné sur les autres  $k-1$  folds
  - modèle testé sur le fold  $k$
- On calcule la perf moyenne sur les  $k$  itérations
  - Chaque fold n'est utilisé qu'une fois pour le test
  - performance moyenne plus fiable

# Recherche par grille

- Nous avons vu précédemment par exemple que pour un SVM non-linéaire les valeurs des hyperparamètres C et γ étaient capitales.
- Une méthode pour gérer cela est de tester de manière systématique plusieurs combinaisons de valeurs pour ces hyperparamètres
  1. on définit une grille (dictionnaire) de paramètres à tester
  2. pour chaque combinaison :
    - a. entraîner le modèle en cross validation
    - b. calculer la perf moyenne
  3. sélectionner la meilleure combinaison

```
param_grid = {
    'C': [0.1, 1, 10, 100],           # 4 valeurs
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1], # 5 valeurs
    'kernel': ['rbf', 'linear']        # 2 valeurs
}
# Total de combinaisons : 4 × 5 × 2 = 40
```

# GridSearch : Exemple de résultat

```
Combinaison 1 : C=0.1, gamma=scale, kernel=rbf      → Accuracy: 82.3%
Combinaison 2 : C=0.1, gamma=scale, kernel=linear   → Accuracy: 79.1%
Combinaison 3 : C=0.1, gamma=auto,  kernel=rbf       → Accuracy: 81.8%
...
Combinaison 38: C=100,  gamma=0.01,  kernel=rbf      → Accuracy: 89.7% ← MEILLEUR
Combinaison 39: C=100,  gamma=0.1,   kernel=rbf       → Accuracy: 87.2%
Combinaison 40: C=100,  gamma=0.1,   kernel=linear    → Accuracy: 83.5%
```

## Exemple 1/3

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

# 1. Définir la grille de paramètres
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
    'kernel': ['rbf', 'linear']
}
```

## Exemple 2/3

```
# 2. Créer l'objet GridSearchCV
grid_search = GridSearchCV(
    estimator=SVC(random_state=42),      # Le modèle de base
    param_grid=param_grid,                # La grille de paramètres
    cv=5,                                # 5-fold cross-validation
    scoring='accuracy',                  # Métrique à optimiser
    n_jobs=-1,                            # Utiliser tous les CPU disponibles
    verbose=2                             # Afficher la progression
)
# 3. Lancer la recherche (entraîne 40×5 = 200 modèles !)
grid_search.fit(x_train_pca, y_train)
```

## Exemple 3/3

```
# 4. Récupérer les meilleurs paramètres
print("Meilleurs paramètres :", grid_search.best_params_)
print("Meilleure accuracy (cv) :", f"{grid_search.best_score_:.2%}")

# 5. Utiliser le meilleur modèle
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test_pca)
test_accuracy = accuracy_score(y_test, y_pred)
print("Accuracy sur le test :", f"{test_accuracy:.2%}")
```

# Exemple : résultats

```
Meilleurs paramètres : {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}  
Meilleure accuracy (cv) : 87.5% ← Performance estimée (entraînement)  
Accuracy sur le test : 86.2% ← Vraie performance (test)
```

- La perf sur l'entraînement est généralement supérieure à celle sur le test
- Le test des combinaisons est donc automatique
- Reproductibilité : attention à bien fixer `random_state`
- Ça devient vite coûteux (40 combinaisons x 5 folds = 200 entraînements)
- Plus il y a de paramètres, plus c'est coûteux
- Recherche exhaustive = test de combinaisons inutiles
- « Grille-dépendant » : si on n'a pas eu la bonne intuition, on loupe des paramètres, les valeurs absentes de la grille ne seront pas testées

# RandomizedSearch

- On crée une grille très complète
- On définit un nombre limité de combinaisons à tester aléatoirement

```
from sklearn.model_selection import RandomizedSearchCV

# On peut même utiliser des distributions continues
param_distributions = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'gamma': [0.0001, 0.001, 0.01, 0.1, 1, 10]
}

random_search = RandomizedSearchCV(
    SVC(kernel='rbf', random_state=42),
    param_distributions=param_distributions,
    n_iter=20, # Tester seulement 20 combinaisons aléatoires
    cv=5,
    random_state=42,
    n_jobs=-1
)
```

- Plus rapide et moins coûteux
- Explore un espace plus large d'hyper paramètres
- Soutien la comparaison avec **GridSearchCV()**

# Bonnes pratiques

- Commencer large, puis affiner
- Faire une première exploration avec `RandomizedSearchCV()`
- Affiner ensuite avec `GridSearchCV()` et la « région de valeurs » d'hyperparamètres qu'on a retenu
- Se réserver un ensemble de test qu'on utilisera jamais avant le test final de validation
- Ne partez pas au petit bonheur la chance : il faut quand même comprendre les hyperparamètres qu'on explore
- Pensez bien à fixer `random_state` !

```
# Première recherche large
param_grid_1 = {'C': [0.1, 1, 10, 100]}
# Puis affiner autour du meilleur
param_grid_2 = {'C': [5, 10, 20, 50]}
```

À vous de jouer !