Le rôle de l'aléatoire dans l'apprentissage automatique

CSI 4506 — Introduction à l'intelligence artificielle

Marcel Turcotte

2026-09-20

Introduction

L'aléatoire joue plusieurs rôles en apprentissage automatique. Il sert à initialiser les paramètres des modèles, à mélanger les données, à créer des mini-lots, à construire les ensembles d'entraînement et de test et à mettre en œuvre des algorithmes randomisés. Ces rôles sont apparentés, mais distincts. Par exemple, une initialisation aléatoire rompt la symétrie entre les neurones d'une même couche d'un réseau de neurones ; si leurs poids étaient initialisés de façon identique, ils pourraient apprendre les mêmes représentations. Le mélange et l'échantillonnage aléatoires exposent un algorithme à différentes sélections et à différents ordres des données disponibles.

Nombres pseudo-aléatoires

La plupart des bibliothèques informatiques qui produisent des nombres aléatoires utilisent en fait des générateurs de nombres *pseudo-aléatoires*. Elles produisent des séquences possédant des propriétés statistiques utiles, mais ces séquences sont générées de manière déterministe.

Un générateur de nombres pseudo-aléatoires maintient un **état** interne. Chaque tirage produit une valeur et met cet état à jour. Un **germe** initialise l'état ; le même générateur, le même germe et la même suite d'appels reproduisent les mêmes valeurs. La valeur `42` est couramment utilisée dans les exemples, mais elle ne possède aucune propriété statistique particulière.

Dans l'exemple suivant, qui utilise le module `random` de Python, nous produisons deux séquences. Puisque nous ne réinitialisons pas le germe entre les deux, la seconde séquence continue à partir de l'état laissé par la première. Les valeurs changeront généralement si cette cellule est exécutée dans une nouvelle session Python.

```python
import random

sequence_1 = [random.randint(1, 100) for _ in range(5)]
sequence_2 = [random.randint(1, 100) for _ in range(5)]

print(f"Séquence 1 : {sequence_1}")
print(f"Séquence 2 : {sequence_2}")
```

```
Séquence 1 : [63, 32, 49, 95, 83]
Séquence 2 : [24, 35, 39, 10, 17]
```

En revanche, l'exemple suivant réinitialise le générateur au même état avant de produire chacune des deux premières séquences. Un autre germe sélectionne normalement une autre séquence.

```python
random.seed(42)
sequence_1 = [random.randint(1, 100) for _ in range(5)]

random.seed(42)
sequence_2 = [random.randint(1, 100) for _ in range(5)]

random.seed(123)
sequence_3 = [random.randint(1, 100) for _ in range(5)]

print(f"Séquence 1 : {sequence_1}")
print(f"Séquence 2 : {sequence_2}")
print(f"Séquence 3 : {sequence_3}")
```

```
Séquence 1 : [82, 15, 4, 95, 36]
Séquence 2 : [82, 15, 4, 95, 36]
Séquence 3 : [7, 35, 12, 99, 53]
```

La capacité de reproduire des choix aléatoires est essentielle en calcul scientifique et en apprentissage automatique. Elle nous aide à déboguer le code, à comparer des méthodes dans les mêmes conditions et à décrire précisément nos expériences.

Le module `random` de Python et scikit-learn ne partagent pas un générateur unique. L'appel `random.seed(42)` contrôle les fonctions du module `random`. Dans scikit-learn, un entier fourni à un paramètre comme `random_state=42` contrôle les choix aléatoires effectués par cet objet ou cette fonction.

Imaginez qu'un modèle se comporte de manière inattendue et que vous demandiez à une collègue ou un collègue d'examiner le problème. Si chaque exécution crée une partition différente des

données, cette personne pourrait ne pas observer le même comportement. Fixer le germe vous permet de reproduire les mêmes choix aléatoires et d'examiner la même expérience. Cela n'améliore pas la partition et ne garantit pas que le modèle obtenu sera performant. Une reproductibilité complète peut aussi dépendre des versions logicielles, des données, du matériel et d'autres sources de variabilité aléatoire.

> **!** Important
>
> Un germe est une **variable de contrôle expérimentale**, et non un réglage de performance. Il rend reproductible une séquence particulière de choix aléatoires. Il ne doit pas être sélectionné parce qu'il produit le résultat de test le plus favorable.

La nécessité de partitionner les données

La performance sur les données d'entraînement indique dans quelle mesure un modèle s'ajuste aux exemples qu'il a déjà vus. Elle fournit généralement une estimation optimiste de la performance sur de nouveaux exemples ; nous utilisons donc un ensemble de test distinct pour estimer la capacité de généralisation. L'ensemble de test ne doit influencer ni l'entraînement ni les choix effectués lors de la construction du modèle. Nous étudierons plus tard des procédures d'évaluation plus complètes.

Partitions reproductibles

Lorsque les exemples peuvent raisonnablement être considérés comme des observations indépendantes provenant de la même distribution, une partition aléatoire réduit l'effet de leur ordre initial et aide à produire des sous-ensembles comparables. Elle ne garantit pas que chaque sous-ensemble sera représentatif, particulièrement lorsque le jeu de données est petit. Les séries chronologiques et les données groupées ou autrement dépendantes nécessitent d'autres stratégies de partitionnement.

Nous créons d'abord un jeu de données jouet contenant 10 exemples. Nous utilisons des noms de couleurs pour voir facilement dans quel sous-ensemble chaque exemple se retrouve. `train_test_split` peut partitionner directement ces chaînes de caractères, bien que la plupart des estimateurs exigent une représentation numérique des attributs avant l'entraînement.

```
X = [['rouge'], ['bleu'], ['vert'], ['jaune'], ['violet'],
     ['orange'], ['rose'], ['brun'], ['noir'], ['blanc']]

# y contient des cibles binaires (0 ou 1)
```

```
y = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

print("Exemples originaux (attribut, cible) :")
print([(item[0], cible) for item, cible in zip(X, y)])
```

```
Exemples originaux (attribut, cible) :
[('rouge', 0), ('bleu', 1), ('vert', 0), ('jaune', 1), ('violet', 0), ('orange', 1), ('rose'
```

Nous utilisons ensuite la fonction `train_test_split` de scikit-learn pour partitionner `X` et `y` ensemble. Les attributs et les cibles correspondantes demeurent donc appariés. Nous produisons cinq partitions en fournissant cinq germes à `random_state`. L'argument `stratify=y` demande à la fonction de préserver les proportions des classes dans la mesure permise par la taille des sous-ensembles.

```
from sklearn.model_selection import train_test_split

germes = [1, 42, 100, 2024, 9999]

for germe in germes:

    # random_state contrôle le mélange des données avant la partition

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.4, random_state=germe, stratify=y
    )

    # Conserver chaque couleur avec sa cible pour faciliter l'examen

    exemples_entrainement = [
        (item[0], cible) for item, cible in zip(X_train, y_train)
    ]
    exemples_test = [(item[0], cible) for item, cible in zip(X_test, y_test)]

    print(f"Germe (random_state) = {germe:<4}")
    print(f"  Ensemble d'entraînement : {exemples_entrainement}")
    print(f"  Ensemble de test :        {exemples_test}\n")
```

```
Germe (random_state) = 1
  Ensemble d'entraînement : [('noir', 0), ('bleu', 1), ('orange', 1), ('vert', 0), ('blanc',
  Ensemble de test :        [('jaune', 1), ('brun', 1), ('rouge', 0), ('rose', 0)]
```

```
Germe (random_state) = 42
  Ensemble d'entraînement : [('brun', 1), ('noir', 0), ('vert', 0), ('jaune', 1), ('orange',
  Ensemble de test :        [('rouge', 0), ('rose', 0), ('bleu', 1), ('blanc', 1)]

Germe (random_state) = 100
  Ensemble d'entraînement : [('brun', 1), ('bleu', 1), ('vert', 0), ('violet', 0), ('jaune',
  Ensemble de test :        [('orange', 1), ('rouge', 0), ('noir', 0), ('blanc', 1)]

Germe (random_state) = 2024
  Ensemble d'entraînement : [('jaune', 1), ('blanc', 1), ('noir', 0), ('vert', 0), ('orange'
  Ensemble de test :        [('bleu', 1), ('brun', 1), ('rouge', 0), ('violet', 0)]

Germe (random_state) = 9999
  Ensemble d'entraînement : [('rouge', 0), ('brun', 1), ('noir', 0), ('bleu', 1), ('violet',
  Ensemble de test :        [('jaune', 1), ('blanc', 1), ('vert', 0), ('rose', 0)]
```

Dans ces exemples, le changement du germe modifie les exemples qui appartiennent à chaque sous-ensemble. Le même entier fourni à `random_state` reproduit la même partition, pourvu que les données et l'environnement logiciel demeurent inchangés.

Conclusion : reproductibilité et variabilité

Il importe de distinguer la reproduction d'une expérience de la mesure de sa sensibilité aux choix aléatoires.

Fixer un germe, par exemple avec `random_state=42`, permet à d'autres personnes de reproduire une partition précise. Toutefois, une partition peut être exceptionnellement favorable ou défavorable ; un résultat reproductible ne constitue donc pas nécessairement un résumé fiable de la performance attendue.

Plus tard dans le cours, nous utiliserons des procédures d'évaluation répétées pour mesurer cette variabilité. Ces expériences doivent suivre une procédure déterminée à l'avance et présenter à la fois la performance typique et sa dispersion, plutôt que d'essayer de nombreux germes et de ne retenir que le meilleur résultat. Le cours illustre le phénomène sous-jacent : différentes partitions peuvent produire différents arbres de décision et différentes mesures de performance.

Documentation

- Module `random` de Python
- `scikit-learn` `train_test_split`
- Contrôle de l'aléatoire dans scikit-learn

Références

— Bethard, S. (2022). "We need to talk about random seeds". arXiv preprint arXiv:2210.13393.

— Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). "Deep reinforcement learning that matters". *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

— Bouthillier, X., Delaunay, P., Bronzi, M., et al. (2021). "Accounting for variance in machine learning benchmarks". *Proceedings of Machine Learning and Systems*, 3, 747-769.