Notes sur le Kaggle Titanic Challenge

1. ROC-AUC : métrique principale pour la classification binaire

ROC-AUC : Receiver Operating Characteristic (courbe) et Area Under the Curve (aire sous la courbe)

C'est la **meilleure métrique** pour la classification binaire du Titanic.

1.1. Définition et interprétation

ROC-AUC mesure la capacité du modèle à distinguer entre les deux classes (0 et 1).

L'**aire sous la courbe ROC (AUC)** représente la probabilité que le modèle, s'il reçoit un exemple positif et un exemple négatif choisis aléatoirement, classe l'exemple positif plus haut que l'exemple négatif.

Score	Interprétation
1.0	Perfection absolue (100% correct)
0.5	Pas mieux qu'un tirage aléatoire
< 0.5	Pire qu'un tirage aléatoire

1.1.1. Exemple concret : Classificateur de spam

Un classificateur de spam avec un AUC de 1.0 attribue toujours à un e-mail de spam aléatoire une probabilité plus élevée d'être du spam qu'un e-mail légitime aléatoire. La classification réelle de chaque e-mail dépend du seuil que vous choisissez.

Cas limite : Un classificateur avec un AUC de 0.5 attribue une probabilité plus élevée d'être du spam à un e-mail de spam aléatoire qu'à un e-mail légitime seulement la moitié du temps, soit pas mieux que le hasard.

1.2. Pourquoi ROC-AUC est supérieure aux autres métriques ?

1.2.1. 1. Mesure les probabilités (pas seulement les classes)

ROC-AUC ne regarde pas juste si vous prédisez 0 ou 1, mais regarde aussi la **confiance** de votre prédiction.

Exemple : « 90% de chance de survivre » vs « 51% de chance » → ROC-AUC fait la différence.

1.2.2. 2. Insensible au déséquilibre des classes

Dans Titanic, il y a plus de morts (0) que de survivants (1). ROC-AUC gère bien ce déséquilibre, contrairement à l'accuracy.

1.2.3. 3. Évalue tous les seuils possibles

Au lieu d'utiliser un seul seuil (ex: 0.5), ROC-AUC teste **TOUS** les seuils possibles :

- Seuil 0.9 → Peu de survivants prédits, mais très sûrs
- Seuil 0.7 → Équilibre
- Seuil 0.3 → Beaucoup de survivants prédits, moins sûrs

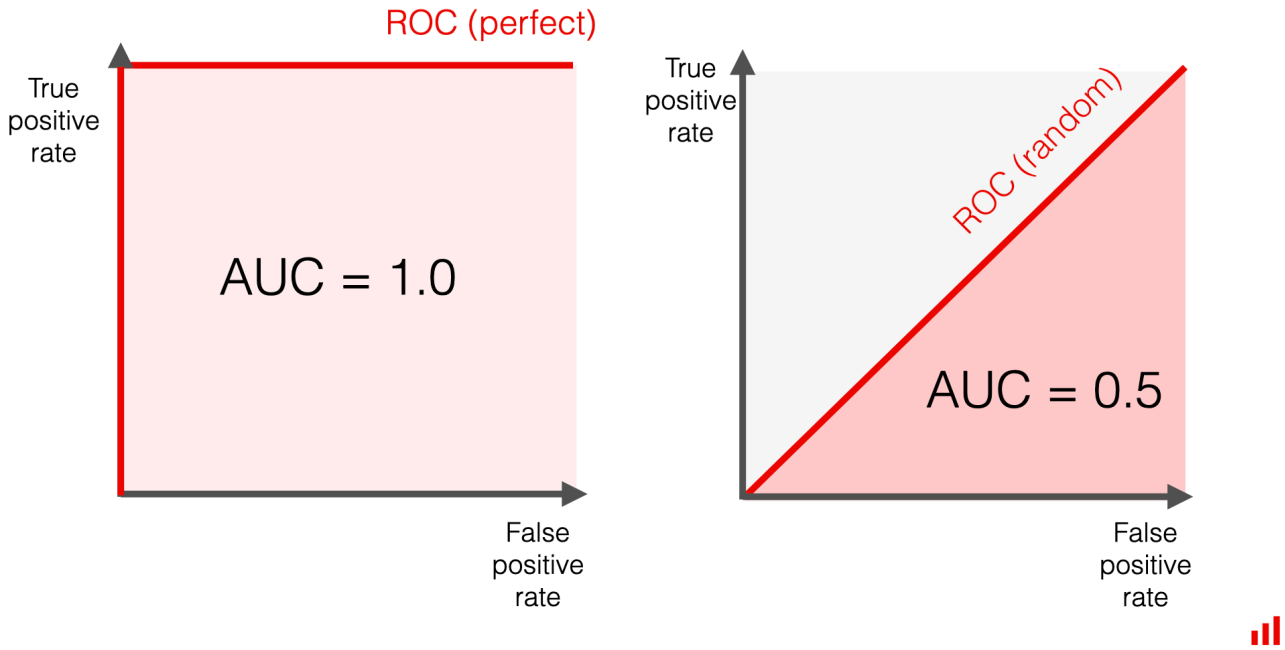
1.3. Formules mathématiques

1.3.1. Formule intégrale (théorique)

$$\text{AUC} = \int_0^1 \text{TPR}(\text{FPR})\, \mathrm{d}(\text{FPR})$$

Où :

- **TPR** (True Positive Rate) = Recall = $\frac{\text{VP}}{\text{VP+FN}}$
- **FPR** (False Positive Rate) = $\frac{\text{FP}}{\text{FP+VN}}$



1.4. Utilisation pratique

AUC et ROC pour choisir un modèle et un seuil :

- **Comparer des modèles** : Le modèle ayant la plus grande aire sous la courbe est généralement le meilleur (si l'ensemble de données est à peu près équilibré)
- **Choisir le seuil optimal** : Les points d'une courbe ROC les plus proches de (0,1) représentent une plage des seuils les plus performants pour le modèle donné

2. Le seuil de classification

À partir de quelle probabilité je décide que quelqu'un survit (1) ou meurt (0) ?

C'est ça le seuil.

2.1. Exemples de seuils

- **Seuil par défaut : 0.5** (50%)
  - Si probabilité $\geq 0.5 \rightarrow$ Prédiction = 1 (survit)
  - Si probabilité $< 0.5 \rightarrow$ Prédiction = 0 (meurt)
- **Seuil ÉLEVÉ (0.8)** = Modèle PRUDENT
  - Haute précision, mais risque de rater des survivants

- **Seuil BAS (0.3)** = Modèle OPTIMISTE
  - Trouve plus de survivants, mais plus de fausses alarmes

3. Autres métriques de classification

Note importante : On a en général un ensemble de métriques de modèle, toutes calculées à une seule valeur de seuil de classification.

Toutefois, si vous souhaitez évaluer la qualité d'un modèle pour **tous les seuils possibles**, vous avez besoin de ROC-AUC.

3.1. Accuracy

3.1.1. Formule

$$\text{Accuracy} = \frac{\text{Vrais Positifs} + \text{Vrais Négatifs}}{\text{Total}}$$

$$= \frac{\text{Bonnes prédictions}}{\text{Toutes les prédictions}}$$

3.1.2. Pourquoi c'est BIAISÉ pour Titanic

Exemple d'un modèle stupide qui prédit toujours « mort » (0) :

Réalité	62 morts (0) + 38 survivants (1) = 100 passagers
Prédiction	100 fois « mort » (0)
Accuracy	$\frac{62}{100} = 62\%$ (semble « pas mal »)

PROBLÈME : Ce modèle est nul. Il n'a **JAMAIS** prédit un seul survivant correctement.

3.1.3. Le problème mathématique

Quand les classes sont **déséquilibrées** (62 vs 38), l'accuracy est **dominée par la classe majoritaire**.

3.2. Precision

3.2.1. Formule

$$\text{Precision} = \frac{\text{Vrais Positifs}}{\text{Vrais Positifs} + \text{Faux Positifs}}$$

« Parmi ceux que j'ai prédits SURVIVANTS, combien le sont vraiment ? »

3.2.2. Pourquoi c'est TROP STRICTE

Exemple :

Réalité	[1, 1, 1, 1, 0, 0, 0, 0] (4 survivants, 4 morts)
Prédiction	[1, 1, 0, 0, 0, 0, 0, 0] (seulement 2 survivants)
Vrais Positifs	2 (j'ai bien prédit 2 survivants)
Faux Positifs	0 (je n'ai pas fait d'erreur)
Precision	$\frac{2}{2+0} = 100\%$

PROBLÈME : J'ai **raté 2 survivants**. Le modèle est trop prudent.

3.2.3. Le problème mathématique

La precision **pénalise les faux positifs** mais **ignore les faux négatifs** :

- Si je ne prédis JAMAIS de survivant → Precision = indéfinie (division par 0)
- Si je suis très conservateur → Haute precision, mais beaucoup de survivants ratés

3.3. Recall (Rappel / Sensibilité)

3.3.1. Formule

$$\text{Recall} = \frac{\text{Vrais Positifs}}{\text{Vrais Positifs} + \text{Faux Négatifs}}$$

« Parmi les VRAIS survivants, combien j'ai trouvés ? »

3.3.2. Pourquoi c'est PAS ASSEZ STRICTE

Exemple :

Réalité	[1, 1, 1, 1, 0, 0, 0, 0] (4 survivants, 4 morts)
Prédiction	[1, 1, 1, 1, 1, 1, 1, 1] (tout le monde survit)
Vrais Positifs	4 (j'ai trouvé tous les survivants)
Faux Négatifs	0 (aucun survivant raté)
Recall	$\frac{4}{4+0} = 100\%$

PROBLÈME : J'ai aussi prédit 4 morts comme survivants. Le modèle est trop optimiste.

3.3.3. Le problème mathématique

Le recall **pénalise les faux négatifs** mais **ignore les faux positifs** :

- Si je prédis TOUT LE MONDE survivant → Recall = 100%, mais beaucoup de fausses alarmes

3.4. F1-Score (Moyenne harmonique)

3.4.1. Formule

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.4.2. Pourquoi c'est une BONNE ALTERNATIVE

Le F1 combine precision et recall pour équilibrer les deux :

Modèle prudent	Precision = 100%, Recall = 50%
	$\rightarrow \text{F1} = 2 \times \frac{100 \times 50}{100+50} = 66.7\%$
Modèle optimiste	Precision = 50%, Recall = 100%
	$\rightarrow \text{F1} = 2 \times \frac{50 \times 100}{50+100} = 66.7\%$
Modèle équilibré	Precision = 80%, Recall = 80%
	$\rightarrow \text{F1} = 2 \times \frac{80 \times 80}{80+80} = 80\%$

3.4.3. Le problème mathématique

La **moyenne harmonique** pénalise fortement les déséquilibres :

- Si une métrique est mauvaise → F1 est tiré vers le bas
- Mais F1 ne résout pas complètement le problème des classes déséquilibrées

4. AutoGluon : Bagging et Stacking

4.1. Bagging et num_bag_folds

4.1.1. Définition simple

Bagging signifie **Bootstrap Aggregating**. C'est une méthode pour rendre le modèle **plus robuste** et **moins sensible aux variations des données**.

4.1.2. Bagging vs Cross-Validation : différences importantes

Attention : Bagging et Cross-Validation se ressemblent mais ont des objectifs différents.

Aspect	Cross-Validation	Bagging
But	**Évaluer** la performance d'un modèle	**Améliorer** la performance (réduire variance/overfitting)
Échantillonnage	Sans remplacement (chaque donnée utilisée 1 fois par fold)	Avec remplacement (bootstrap)
Modèles créés	K modèles temporaires (jetés après évaluation)	K modèles permanents (gardés pour la prédiction)
Prédiction finale	Réentraîner 1 modèle sur tout le dataset	Moyenne des K modèles
Usage	Évaluation/validation	Production/prédiction

4.1.3. Exemple concret : Cross-Validation

Imaginons que tu veux choisir entre `max_depth=10` et `max_depth=20` pour ton Random Forest.

Avec 5-fold Cross-Validation :

1. **Diviser les 1000 lignes en 5 folds** (200 lignes chacun)
2. **Pour max_depth=10** :
   - Fold 1 : entraîner sur folds 2,3,4,5 → tester sur fold 1 → score = 0.82
   - Fold 2 : entraîner sur folds 1,3,4,5 → tester sur fold 2 → score = 0.85
   - Fold 3 : entraîner sur folds 1,2,4,5 → tester sur fold 3 → score = 0.83
   - Fold 4 : entraîner sur folds 1,2,3,5 → tester sur fold 4 → score = 0.84
   - Fold 5 : entraîner sur folds 1,2,3,4 → tester sur fold 5 → score = 0.81
   - **Score moyen = 0.83**
3. **Pour max_depth=20** :
   - Même processus → score moyen = 0.79
4. **Décision** : max_depth=10 est meilleur (0.83 > 0.79)
5. **Modèle final** : On **jette** les 5 modèles temporaires et on **réentraîne** 1 seul modèle avec max_depth=10 sur les 1000 lignes complètes
6. **Prédiction** : Ce modèle unique fait les prédictions sur le test

Point clé CV : Les 5 modèles créés pendant la CV ne servent qu'à **évaluer**, ils sont ensuite **jetés**. On réentraîne un modèle final sur tout le dataset.

4.1.4. Le bagging dans AutoGluon : un hybride

AutoGluon avec `num_bag_folds=5` fait du « **CV-style bagging** », ce qui est différent du bagging classique :

Bagging classique (Random Forest par exemple) :
- Tire des échantillons **avec remplacement** (bootstrap)
- Certaines lignes peuvent apparaître plusieurs fois, d'autres jamais
- Les K modèles sont entraînés sur des échantillons de taille égale au dataset original

Bagging AutoGluon (num_bag_folds=5) :
- Divise les données en 5 folds (comme CV), **sans remplacement**
- Chaque ligne apparaît exactement 1 fois dans la validation
- Les 5 modèles sont entraînés sur 80% des données chacun
- **MAIS** contrairement à la CV, les 5 modèles sont **gardés** pour faire des prédictions

4.1.5. Exemple concret : Bagging classique

Imaginons que tu as 1000 lignes et tu veux faire du bagging avec 5 modèles.

Avec Bagging classique (Bootstrap Aggregating) :

1. **Créer 5 échantillons bootstrap** (tirage avec remplacement) :
   - Échantillon 1 : tirer 1000 lignes aléatoirement avec remplacement
     - Ligne 5 apparaît 3 fois, ligne 42 apparaît 0 fois, ligne 123 apparaît 2 fois, etc.
     - Contient environ 63% des lignes originales uniques
   - Échantillon 2 : nouveau tirage de 1000 lignes avec remplacement
     - Différent de l'échantillon 1
   - Échantillons 3, 4, 5 : même principe
2. **Entraîner 5 modèles** :
   - Modèle 1 sur échantillon 1
   - Modèle 2 sur échantillon 2
   - Modèle 3 sur échantillon 3
   - Modèle 4 sur échantillon 4
   - Modèle 5 sur échantillon 5
3. **Garder les 5 modèles** pour la prédiction
4. **Prédiction finale** : Pour un nouveau passager du Titanic
   - Modèle 1 → probabilité de survie = 0.7
   - Modèle 2 → probabilité de survie = 0.8
   - Modèle 3 → probabilité de survie = 0.6
   - Modèle 4 → probabilité de survie = 0.75
   - Modèle 5 → probabilité de survie = 0.72
   - **Prédiction finale = moyenne = 0.714**

Point clé Bagging classique : Les 5 modèles sont **gardés** et leurs prédictions sont **moyennées**. Échantillonnage **avec remplacement** (bootstrap).

4.1.6. Exemple concret : Bagging AutoGluon

Imaginons le même dataset de 1000 lignes avec `num_bag_folds=5`.

Avec Bagging AutoGluon :

1. **Diviser les 1000 lignes en 5 folds** (200 lignes chacun, **sans remplacement**)
   - Fold 1 : lignes 1-200

- Fold 2 : lignes 201-400
- Fold 3 : lignes 401-600
- Fold 4 : lignes 601-800
- Fold 5 : lignes 801-1000

2. **Entraîner 5 modèles** (comme en CV) :
   - Modèle 1 : entraîné sur folds 1,2,3,4 (800 lignes) → validé sur fold 5
   - Modèle 2 : entraîné sur folds 1,2,3,5 (800 lignes) → validé sur fold 4
   - Modèle 3 : entraîné sur folds 1,2,4,5 (800 lignes) → validé sur fold 3
   - Modèle 4 : entraîné sur folds 1,3,4,5 (800 lignes) → validé sur fold 2
   - Modèle 5 : entraîné sur folds 2,3,4,5 (800 lignes) → validé sur fold 1
3. **Garder les 5 modèles** pour la prédiction (contrairement à la CV !)
4. **Prédiction finale** : Pour un nouveau passager du Titanic
   - Modèle 1 → probabilité de survie = 0.7
   - Modèle 2 → probabilité de survie = 0.8
   - Modèle 3 → probabilité de survie = 0.6
   - Modèle 4 → probabilité de survie = 0.75
   - Modèle 5 → probabilité de survie = 0.72
   - **Prédiction finale = moyenne = 0.714**

Point clé Bagging AutoGluon : Les 5 modèles sont **gardés** et leurs prédictions sont **moyennées** (comme bagging classique). MAIS échantillonnage **sans remplacement** en folds (comme CV), ce qui est plus précis car chaque modèle est validé sur des données jamais vues.

4.1.7. Principe de fonctionnement

Imaginons un dataset de **1000 lignes**. Si tu fais du bagging avec `num_bag_folds=5`, AutoGluon va :

1. **Diviser le dataset en 5 parties** (appelées **folds** ou **plis**). Chaque partie fait environ 20% des données.

   Fold 1 | Fold 2 | Fold 3 | Fold 4 | Fold 5

2. Créer **5 modèles** :

Modèle	Entraîné sur	Validé sur
Modèle 1	Folds 1,2,3,4	Fold 5
Modèle 2	Folds 1,2,3,5	Fold 4
Modèle 3	Folds 1,2,4,5	Fold 3
Modèle 4	Folds 1,3,4,5	Fold 2
Modèle 5	Folds 2,3,4,5	Fold 1

3. **Garder les 5 modèles** pour la prédiction finale :

Ainsi, chaque partie du dataset sert **une fois de validation** et **quatre fois d'entraînement**. AutoGluon combine ensuite les 5 modèles (en **moyennant les prédictions**) pour un **résultat plus stable**.

Résumé : Le bagging AutoGluon utilise la structure de la cross-validation mais garde tous les modèles pour améliorer les prédictions, contrairement à la CV qui jette les modèles après évaluation.

4.1.8. Avantages et inconvénients

Avantages :

- Réduit le risque d'overfitting
- Fournit une meilleure estimation de la performance

Inconvénients :

- 5 fois plus long à entraîner (5 modèles au lieu d'un)

4.1.9. num_bag_sets

- Définit le **nombre de « sets » de bagging**.
- En général, `num_bag_sets=1` suffit.
- Si `num_bag_sets=2` → le bagging est répété deux fois (10 modèles au total).

4.2. Stacking et num_stack_levels

4.2.1. Définition simple

Stacking consiste à **empiler plusieurs modèles** pour qu'ils **apprennent les erreurs des autres**.

Exemple de structure :

```
Niveau 0 (L1) : modèles de base (GBM, XGB, RF, NN_TORCH…)
           ↓ (leurs prédictions)
Niveau 1 (L2) : nouveau modèle entraîné sur les prédictions du L1
```

Si `num_stack_levels=1`, tu as **deux couches** de modèles :

- les modèles de base (L1)
- un modèle de méta-apprentissage (L2)

4.2.2. Exemple concret de fonctionnement

Supposons que tu veux prédire si un client va acheter (`0` ou `1`). Les **features originales** sont :

⇒

âge, revenu, sexe, ville, nombre_achats

4.2.2.1. Niveau 0 : données originales

âge	revenu	sexe	ville	nombre_achats	cible
35	2500	F	Paris	12	1
52	1800	M	Lyon	4	0
22	1200	F	Lille	7	0
…	…	…	…	…	…

Les colonnes à gauche (âge → nombre_achats) = **features d'entrée**. La colonne `cible` = ce que l'on veut **prédire**.

4.2.2.2. Niveau 1 : modèles de base (L1)

AutoGluon entraîne plusieurs modèles sur ces features :

- GBM (LightGBM)
- XGB (XGBoost)

- RF (Random Forest)
- CAT (CatBoost)
- NN_TORCH (réseau de neurones)

Chaque modèle produit une prédiction (probabilité) :

ID	GBM	XGB	RF	CAT	NN_TORCH	cible
1	0.7	0.8	0.6	0.9	0.75	1
2	0.2	0.4	0.3	0.25	0.35	0
3	0.5	0.45	0.55	0.52	0.5	0

Les **valeurs de sortie** deviennent un **nouveau tableau de données**.

4.2.2.3. Niveau 2 : stacking (L2)

AutoGluon crée un **nouveau modèle** (souvent un LightGBM ou modèle linéaire) qui apprend sur les **sorties du niveau précédent** :

GBM	XGB	RF	CAT	NN_TORCH	cible
0.7	0.8	0.6	0.9	0.75	1
0.2	0.4	0.3	0.25	0.35	0
0.5	0.45	0.55	0.52	0.5	0

Les « features » ici sont **les prédictions des modèles précédents**. Le modèle de stacking apprend à **pondérer** ces prédictions.

4.2.3. Pourquoi ce n'est pas incohérent

Même si la nature des features change :

- Niveau 0 → données réelles (âge, revenu…)
- Niveau 1 → données interprétées (probabilités)

C'est parfaitement logique : chaque niveau est **une abstraction supérieure**, comme les couches d'un réseau de neurones :

- Les premières couches voient des **pixels bruts**
- Les couches suivantes voient des **formes et objets**

Ici, les modèles de base voient les **features brutes**, le modèle de stacking voit les **« avis »** (prédictions) de ces modèles.

4.2.4. Analogie simple : Critiques de cinéma

Prédire si un film plaira à quelqu'un :

- **Features originales** : genre, durée, réalisateur, acteurs
- **Modèles de base** : 5 critiques de cinéma (GBM, XGB, RF, etc.) → chacun donne une note /10
- **Stacking (niveau 2)** : un **méta-critique** apprend à combiner les avis des 5 critiques pour produire la note finale la plus juste

4.2.5. Pourquoi faire ça

Chaque modèle a ses **forces et faiblesses** :

- GBM → bon sur les relations non linéaires
- RF → robuste au bruit
- NN_TORCH → capte des patterns complexes

Le modèle de niveau 2 apprend à **pondérer** ces prédictions intelligemment, souvent avec un gain de **+1 à +3% de performance**.

5. AutoGluon : Interprétation des résultats

5.1. Verbosity : niveaux de détail

Verbosity contrôle le niveau de détail des messages affichés par AutoGluon pendant l'entraînement.

Niveau	Nom	Ce qui est affiché
0	Silencieux	Presque rien (juste les erreurs critiques)
1	Minimal	Résultats finaux uniquement
2	Standard	Logs normaux (recommandé)
3	Détaillé	Beaucoup de détails techniques
4	Debug	Tout (pour débugger des problèmes)

Tableau 1. – Niveaux de verbosité dans AutoGluon

5.1.1. Avec Verbosity = 2 (Standard)

```
================== System Info ==================
AutoGluon Version:  1.4.0
Python Version:     3.12.12
...

Beginning AutoGluon training ...

Preprocessing data ...

Fitting model: LightGBM ...
  0.8377 = Validation score (roc_auc)
  0.54s = Training runtime
...

AutoGluon training complete
Best model: WeightedEnsemble_L2
```

5.1.2. Configuration de la verbosity

```python
from autogluon.tabular import TabularPredictor

predictor = TabularPredictor(
    label='Survived',
    verbosity=2  # Changer ici (0 à 4)
)
```

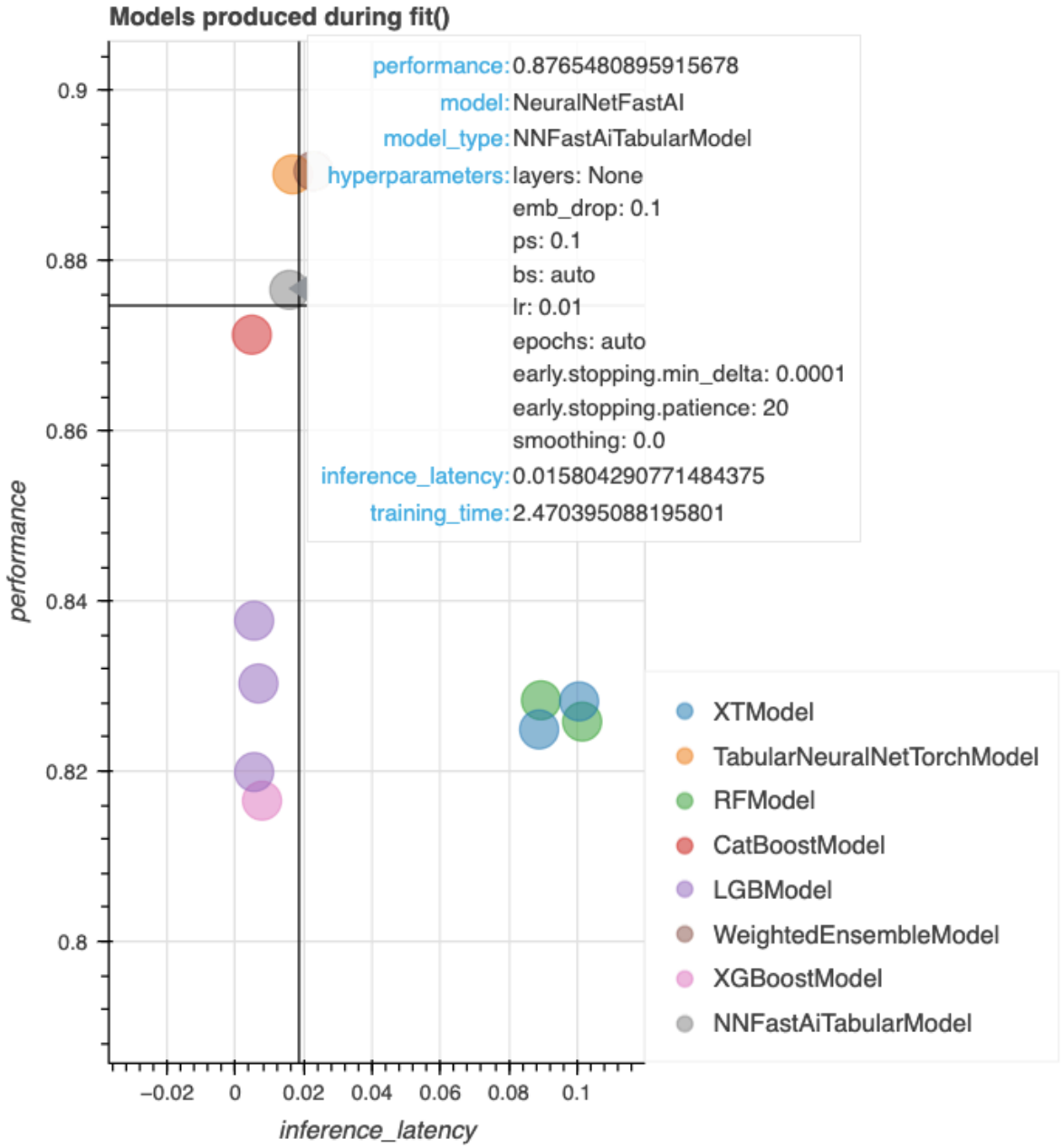
5.1.3. Quand utiliser chaque niveau

- **Verbosity = 0** : Production (serveur)
- **Verbosity = 1** : Entraînement rapide, utilisateur expérimenté
- **Verbosity = 2** : Par défaut (recommandé), voir la progression et les scores
- **Verbosity = 3** : Comprendre en profondeur, optimisation manuelle
- **Verbosity = 4** : Débugger un problème, développement

5.2. Output : statistiques et rapport HTML

Le script génère les statistiques dans un fichier HTML interactif en local.

Le choix final sera généralement : **WeightedEnsemble_L2**



5.2.1. Les niveaux d'ensemble

5.2.1.1. Modèles de base (Level 1)

Ordre	Modèle	Level
0	RandomForestEntr	1
1	RandomForestGini	1
2	ExtraTreesGini	1
3	ExtraTreesEntr	1
4	LightGBMLarge	1
5	NeuralNetTorch	1
6	XGBoost	1
7	NeuralNetFastAI	1
8	CatBoost	1

Ordre	Modèle	Level
9	LightGBM	1
10	LightGBMXT	1

5.2.1.2. WeightedEnsemble_L2 (Level 2)

Le **WeightedEnsemble_L2** :

- Combine intelligemment les prédictions des modèles de base
- Apprend les meilleurs poids pour chaque modèle
- C'est généralement le meilleur modèle

5.2.2. Pourquoi c'est un « mix de modèles »

Le WeightedEnsemble combine les prédictions comme ceci :

$$\text{Prédiction finale} = w_1 \times \text{LightGBM} + w_2 \times \text{CatBoost} + w_3 \times \text{XGBoost} + ...$$

Où $w_1, w_2, w_3, ...$ sont des **poids optimisés automatiquement**.

6. Random Forest : recherche d'hyperparamètres

6.1. Rappel : hyperparamètres principaux

Paramètre	Rôle	Effet typique
`n_estimators`	Nombre d'arbres	plus stable, mais plus lent
`max_depth`	Profondeur max de chaque arbre	évite le surapprentissage
`min_samples_split`	Nb minimal d'échantillons pour diviser un nœud	modèle plus simple
`min_samples_leaf`	Nb minimal d'échantillons dans une feuille	régularisation plus forte
`max_features`	Nb de features testées à chaque split	plus de diversité, meilleure précision possible
`bootstrap`	Si true, échantillonnage avec remplacement	généralement true

6.2. Trois stratégies pour trouver les bons hyperparamètres

6.2.1. A. Méthode manuelle (exploration raisonnée)

Tu testes à la main quelques combinaisons typiques.

6.2.2. B. Grid Search (recherche systématique)

Essaye toutes les combinaisons d'un petit ensemble de valeurs.

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

param_grid = {
    'n_estimators': [100, 200, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2']
}
```

```
grid_search = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='roc_auc',
    n_jobs=-1,
    verbose=2
)

grid_search.fit(X, y)

print("Best parameters:", grid_search.best_params_)
print("Best ROC AUC:", grid_search.best_score_)
```

Avantage : exhaustif, fiable **Inconvénient :** lent si beaucoup de paramètres

6.2.3. C. Randomized Search (plus rapide)

Teste un sous-ensemble aléatoire d'hyperparamètres.

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint

param_dist = {
    'n_estimators': randint(100, 500),
    'max_depth': randint(5, 50),
    'min_samples_split': randint(2, 10),
    'min_samples_leaf': randint(1, 5),
    'max_features': ['sqrt', 'log2']
}

random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_distributions=param_dist,
    n_iter=50,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=2,
    random_state=42
)

random_search.fit(X, y)

print("Best parameters:", random_search.best_params_)
print("Best ROC AUC:", random_search.best_score_)
```

Avantage : beaucoup plus rapide, souvent aussi efficace qu'un GridSearch complet

6.3. Retour sur la validation croisée :

6.3.1. Script

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import pandas as pd

# Préparer les données
train_data = pd.read_csv('train.csv')
```

```
X = train_data.drop(columns=['Survived'])
y = train_data['Survived']

# Définir le modèle
rf = RandomForestClassifier(n_estimators=200, random_state=42)

# Validation croisée (5 folds)
scores = cross_val_score(rf, X, y, cv=5, scoring='roc_auc')

print("ROC AUC per fold:", scores)
print("Mean ROC AUC:", scores.mean())
```

Le modèle est entraîné 5 fois sur différentes parties des données, ce qui réduit le risque de surajustement.

6.3.2. Fonctionnement de cross_val_score

Quand tu fais :

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5)
```

- `X` et `y` correspondent au dataset d'entraînement complet
- `cross_val_score` divise automatiquement les données en `cv` folds (ici 5)
- Chaque « fold » joue alternativement le rôle de validation, le reste sert à l'entraînement
- **Tu n'as pas besoin de séparer manuellement un train/val pour estimer la performance**
- La moyenne des scores CV (`scores.mean()`) te donne une bonne estimation de la performance réelle

6.3.3. Attention : importance d'un test set séparé

Il y a toujours un risque de trop optimiser ton modèle sur le même dataset, même avec CV.

Bonne pratique : garder un petit test set séparé (par exemple avec `train_test_split`) avant tout, pour avoir un vrai score final sur des données jamais utilisées.

6.4. Bonnes pratiques pour la recherche d'hyperparamètres

1. **Toujours utiliser la validation croisée** pour éviter le surajustement (overfitting)
2. **Évaluer avec une métrique adaptée** (roc_auc, f1, etc.)
3. **Fixer random_state** pour des résultats reproductibles
4. **Ne jamais évaluer directement sur le test** avant d'avoir choisi le modèle final

7. Feature Engineering et preprocessing

7.1. Encodage des variables catégorielles

7.1.1. Les différents types d'encoders

Encoder	Description
LabelEncoder	Transforme les labels en entiers (0,1,2…). Encoder classique.
OrdinalEncoder	Transforme les colonnes catégoriques en valeurs numériques avec un ordre spécifié.
OneHotEncoder	Transforme les colonnes catégoriques en plusieurs colonnes binaires (0 ou 1) → évite l'ordre implicite.

7.1.2. Exemple : traitement de la variable Sex

Problème : Éviter que male=1 et female=0 biaise le modèle.

Solution : OneHotEncoder → Sex devient deux colonnes binaires séparées.

Sex	Male	Female
male	1	0
female	0	1

Maintenant chaque colonne est un nombre réel (0 ou 1) que le modèle peut utiliser sans ordre implicite.

7.1.3. Exemple : traitement de la variable Embarked

Pour Embarked, on peut utiliser un ordre « du moins huppé au plus huppé » : Q → C → S

```
from sklearn.preprocessing import OrdinalEncoder

ord_enc = OrdinalEncoder(categories=[['Q','C','S']])
train_x['Embarked'] = ord_enc.fit_transform(train_x[['Embarked']])
```

7.2. Autre méthode : ColumnTransformer

7.2.1. Pourquoi utiliser ColumnTransformer

Problèmes avec l'approche manuelle :

1. `fit_transform` retourne une matrice sparse, pas un DataFrame
2. Si tu as plusieurs colonnes à encoder (Embarked, Pclass, Sex…), tu dois faire un encodage différent pour chaque type
3. Risque d'erreurs de dimension ou de mélange de colonnes

Solution : ColumnTransformer permet de définir plusieurs transformations pour différentes colonnes, puis de combiner automatiquement le résultat en une seule matrice utilisable par ton modèle.

7.2.2. Exemple d'utilisation

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder

preprocessor = ColumnTransformer(
    transformers=[
        ('sex_ohe', OneHotEncoder(drop='if_binary'), ['Sex']),
        ('embarked_ord', OrdinalEncoder(categories=[['Q','C','S']]), ['Embarked'])
    ],
    remainder='passthrough'  # les colonnes numériques restent inchangées
)
```

Avantages :
- Chaque colonne est transformée selon le type choisi
- Les colonnes numériques passent « en l'état »
- Résultat prêt pour le modèle

7.2.3. Alternative àColumnTransformer (ce que j'ai fait)

Tu peux encoder chaque colonne séparément :

CF SCRIT

Résultat : DataFrame prêt pour le Random Forest **Inconvénient :** beaucoup de code répétitif si tu as beaucoup de colonnes à transformer

7.2.4. Pourquoi concat + reset_index pour OneHotEncoder mais pas pour OrdinalEncoder

OneHotEncoder :
- Renvoie une nouvelle DataFrame (ou matrice) avec 1 colonne (ou plusieurs)
- Cette nouvelle DataFrame n'a pas forcément les mêmes index que le DataFrame original
- Si tu fais `pd.concat([train_x, train_sex], axis=1)` sans `reset_index`, les index peuvent ne pas correspondre → colonnes décalées
- Donc on fait `reset_index(drop=True)` pour s'assurer que les lignes correspondent

OrdinalEncoder :
- Transforme directement une seule colonne en numpy array de même taille que la colonne originale
- Quand tu assignes directement : `train_x_encoded['Embarked'] = ord_enc.fit_transform(...)`
- Pas besoin de concat ni de reset_index, la colonne remplace directement l'ancienne
- L'index du DataFrame reste inchangé, donc pas de risque de décalage

Type de transformation	Faut-il concat + reset_index ?
OneHotEncoder (plusieurs colonnes)	Oui, pour aligner les lignes et ajouter les nouvelles colonnes
OrdinalEncoder (1 colonne remplacée)	Non, on peut assigner directement

7.3. Principe de Pipeline :

7.3.1. Principe

Pipeline = combinaison de transformations + modèle.

Avantages :
1. Moins d'erreurs humaines
2. Tout est automatisé et reproductible
3. Compatible avec les grilles de recherche (GridSearchCV, RandomizedSearchCV)
4. Plus propre et plus clair
5. Zéro fuite de données (data leakage)

7.3.2. Exemple complet avec RandomizedSearchCV

```
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder

# 1. Définition du preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('sex_ohe', OneHotEncoder(drop='if_binary'), ['Sex']),
```

```
        ('embarked_ord', OrdinalEncoder(categories=[['Q','C','S']]), ['Embarked'])
    ],
    remainder='passthrough'
)

# 2. Définition du modèle
rf = RandomForestClassifier(random_state=42)

# 3. Création du pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', rf)
])

# 4. Définition de l'espace d'hyperparamètres
param_dist = {
    'classifier__n_estimators': [100, 200, 500],
    'classifier__max_depth': [None, 5, 10, 20],
    'classifier__min_samples_split': [2, 4, 8],
    'classifier__min_samples_leaf': [1, 2, 4],
    'classifier__max_features': ['sqrt', 'log2', None]
}

# 5. Randomized Search avec 5-fold CV
random_search = RandomizedSearchCV(
    pipeline,
    param_distributions=param_dist,
    n_iter=20,   # nombre de combinaisons aléatoires à tester
    cv=5,        # validation croisée 5 folds
    scoring='roc_auc',
    n_jobs=-1,
    verbose=2,
    random_state=42
)

# 6. Fit du modèle
random_search.fit(train_x, train_y)

# 7. Résultats
print("Best hyperparameters:", random_search.best_params_)
print("Best CV ROC AUC:", random_search.best_score_)

# 8. Prédictions sur le test
predictions = random_search.predict(test_x)
```

7.3.3. Détails du code

Étape 1 : Preprocessor

- Définit comment encoder chaque colonne
- `remainder='passthrough'` → les colonnes numériques restent inchangées

Étape 2 : Modèle

- Random Forest avec `random_state` pour la reproductibilité
- Aucun hyperparamètre fixé → on va les explorer avec RandomizedSearchCV

Étape 3 : Pipeline

- Combine preprocessor + classifier
- Tu peux appliquer le pipeline directement sur train et test

Étape 4 : Espace d'hyperparamètres
- Le double underscore `classifier__` sert à accéder aux hyperparamètres du modèle dans le pipeline
- Définit les valeurs à tester pour chaque hyperparamètre

Étape 5 : RandomizedSearchCV
- Teste 20 combinaisons aléatoires dans l'espace défini
- Pour chaque combinaison, fait une validation croisée 5 folds
- `scoring='roc_auc'` → le score utilisé pour comparer les modèles

Étape 6 : Fit
- RandomizedSearchCV transforme les colonnes avec le preprocessor
- Entraîne Random Forest pour chaque combinaison aléatoire
- Fait la CV pour calculer roc_auc
- Retient la meilleure combinaison

Étape 7 : Résultats
- Affiche les meilleurs hyperparamètres trouvés
- Affiche le meilleur score ROC AUC sur la validation croisée

Étape 8 : Test
- Une fois que tu as ton meilleur modèle (`random_search.best_estimator_`)
- Tu peux faire des prédictions sur test.csv

7.3.4. Quand passer au test.csv
- Si le score ROC AUC sur CV est satisfaisant → tu peux passer au test
- Si le score est faible → retravailler les features, l'encodage, ou élargir l'espace d'hyperparamètres
- Ne jamais évaluer directement sur le test avant d'avoir choisi le modèle final

7.4. Mon choix de traitement des variables

7.4.1. Fare
Utilisé tel quel, valeurs manquantes remplacées par la médiane.

7.4.2. Embarked
Ordre « du moins huppé au plus huppé » : Q → C → S

OrdinalEncoder avec les catégories ordonnées `['Q','C','S']`

7.4.3. Ticket,Name et Cabine
Variables supprimées du modèle.

8. Ressources et références

8.1. ROC-AUC
- **Bibmath** : https://www.bibmath.net/dico/index.php?action=affiche&quoi=./r/roc.html
- **Google ML Crash Course** : https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc?hl=fr