

# Rapport de Projet

---

DEEP LEARNING  
MASTER 2 DATA SCIENCE

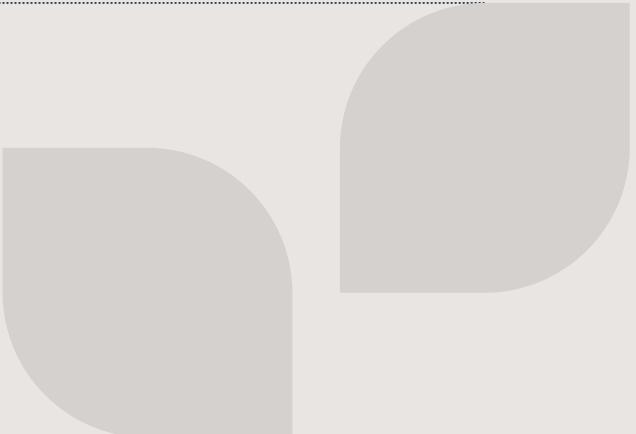


KOUASSI AKA YVES ADAM

GNONSON BOHIN KEAPOHET MELCHI

# SOMMAIRE

REMERCIEMENTS	1
CONTEXTE DU PROJET	2
PROBLEMATIQUE	3
PRESENTATION DU JEU DE DONNEES	4
METHODOLOGIE	5
MODELISATION	17
CONCLUSION	22



# 1. REMERCIEMENTS

---

Nous tenons à exprimer notre profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce projet.

Nous remercions tout d'abord Dr. AYIPKA KACOUTCHY , enseignant du module de Deep Learning, pour sa disponibilité, ses conseils avisés et son accompagnement méthodologique tout au long de ce travail.

Sa rigueur pédagogique et sa passion pour la discipline ont été une source constante de motivation pour notre groupe.

Nos remerciements s'adressent également à l'ensemble du corps professoral de l'UFR MI , pour la qualité de la formation dispensée et leur engagement dans la réussite des étudiants.

---

Nous sommes également reconnaissants envers nos camarades et collègues pour leur collaboration, leurs échanges enrichissants et l'esprit d'équipe qui a prévalu tout au long de ce projet. Enfin, nous remercions nos familles et proches pour leur soutien moral, leur patience et leurs encouragements continus qui nous ont permis d'atteindre nos objectifs avec confiance et détermination.



## 2. CONTEXTE DU PROJET

La détection automatique de défauts constitue aujourd’hui un enjeu majeur dans de nombreux secteurs industriels, notamment ceux impliquant des structures métalliques exposées à l’usure et aux conditions environnementales. La corrosion, en particulier la rouille, peut provoquer des dégradations importantes, engendrer des arrêts de production, augmenter les coûts de maintenance et présenter des risques pour la sécurité. Dans ce contexte, la vision par ordinateur s’impose comme un outil performant permettant d’automatiser et d’accélérer les inspections visuelles.

L’utilisation du Deep Learning pour la classification d’images s’est révélée particulièrement efficace grâce à sa capacité à extraire automatiquement des caractéristiques pertinentes à partir de données brutes. Dans le cadre de ce projet, nous exploitons un réseau de neurones artificiels pour distinguer les images présentant de la rouille (“Rust”) de celles qui n’en présentent pas (“NoRust”).

Le jeu de données utilisé a été fourni par le professeur. Il se compose d’images prétraitées sous forme de valeurs numériques (matrices de pixels), accompagnées d’une étiquette de classe. L’objectif est d’étudier l’impact du déséquilibre entre les classes et d’évaluer différentes stratégies de rééquilibrage afin d’obtenir des performances optimales.

La problématique centrale de ce projet est donc la suivante : comment concevoir un modèle de deep learning capable de classer efficacement des images en “Rust” ou “NoRust”, malgré un fort déséquilibre du dataset ?

Ce rapport présente successivement :

- 1.Une analyse du jeu de données ;
- 2.La méthodologie adoptée, du prétraitement à la modélisation ;
- 3.Une comparaison de plusieurs méthodes de gestion du déséquilibre ;
- 4.L’évaluation des différents modèles ;
- 5.Une discussion sur les limites et perspectives d’amélioration.



# 3. PROBLEMATIQUE

La détection automatique de la rouille joue un rôle essentiel dans plusieurs domaines industriels :

Inspection et contrôle qualité, permettant une identification rapide des surfaces corrodées ;

Maintenance prédictive, favorisant la planification des interventions avant l'apparition de dommages graves ;

Réduction des coûts, en diminuant les interventions manuelles et les opérations de maintenance d'urgence.

Cependant, un obstacle majeur apparaît lors de la construction d'un modèle de classification : le déséquilibre des classes. En effet, les images annotées "Rust" sont souvent beaucoup moins nombreuses que celles annotées "NoRust". Ce déséquilibre peut conduire le modèle à privilégier systématiquement la classe majoritaire, entraînant un mauvais rappel sur la classe "Rust", pourtant la plus importante.

La question centrale du projet est donc :

Comment construire un modèle de deep learning performant pour classer efficacement les images en "Rust" ou "NoRust", malgré le déséquilibre du dataset ?



# 4. PRESENTATION DU JEU DE DONNEES

Le jeu de données fourni contient des images représentant des surfaces métalliques, certaines présentant de la rouille et d'autres non. Chaque image a été transformée en tableau numérique où chaque pixel est représenté par une ou plusieurs valeurs (intensité lumineuse ou valeurs RGB selon le format).

Les variables se décomposent ainsi :

Variables explicatives : valeurs de pixels et éventuellement des caractéristiques dérivées.

Variable cible : class, indiquant "Rust" ou "NoRust".

Une étude exploratoire a permis :

d'observer la distribution des classes, confirmant un fort déséquilibre ;

de détecter la présence éventuelle de valeurs aberrantes ;

d'obtenir les premières statistiques descriptives du dataset.

Ce diagnostic initial est essentiel pour orienter la stratégie de prétraitement et de modélisation.



# 5. METHODOLOGIE

## 5.1 Pipeline global

### 1-Chargement des données

Dans cette phase, le dataset est importé et examiné afin de mieux comprendre sa structure et son contenu.

### 2-Nettoyage

Suppression des doublons, vérification des valeurs manquantes ou incohérentes.

### 3-Normalisation

Mise à l'échelle des valeurs de pixels pour stabiliser l'apprentissage.

### 4-Séparation Train / Test

Permet d'évaluer correctement le modèle sans fuite de données.



# 5. METHODOLOGIE

## 5.1 Pipeline global

### 5-Gestion du déséquilibre des classes

Application de différentes stratégies (SMOTE, RUS, sans rééquilibrage...).

### 6-Modélisation

Construction d'un réseau de neurones entièrement connecté.

### 7-Évaluation

Analyse des performances via des métriques adaptées aux classes déséquilibrées.

### 8-Comparaison des méthodes

Identification de la stratégie la plus performante.





## 5.2 Préparation des données

### Importation des librairies

Dans cette première étape, toutes les librairies nécessaires au projet sont chargées. Elles couvrent plusieurs aspects essentiels du pipeline :

- Manipulation et analyse des données : pandas, numpy
- Visualisation : matplotlib, seaborn
- Préprocessing et évaluation des modèles : outils de scikit-learn
- Gestion du déséquilibre : techniques d'oversampling et undersampling (SMOTE, RandomUnderSampler)
- Construction de modèles deep learning : TensorFlow/Keras
- Stabilisation des résultats : fixation des graines aléatoires
- Gestion des avertissements et sauvegarde : warnings, joblib

Cette phase prépare l'environnement en rassemblant tous les outils indispensables pour le traitement des données, l'entraînement des modèles et l'analyse des performances.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Pour le préprocessing
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import classification_report, accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef, confusion_matrix
from sklearn.inspection import permutation_importance
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline as ImbPipeline
import joblib

# Pour le deep Learning
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.regularizers import l2

# Pour gérer le déséquilibre de classes
from sklearn.utils.class_weight import compute_class_weight

# Pour rendre les résultats reproductibles
np.random.seed(42)
tf.random.set_seed(42)

import warnings
warnings.filterwarnings("ignore")

print("Toutes les librairies sont chargées !")
Toutes les librairies sont chargées !
```



## 5.2 Préparation des données

### Chargement et exploration du dataset

Dans cette phase, le dataset est importé et examiné afin de mieux comprendre sa structure et son contenu. L'objectif est de :

- Charger les données depuis leur source (CSV, base de données, etc.).
- Inspecter la forme du dataset (nombre de lignes, nombre de colonnes).
- Analyser la nature des variables (types, valeurs manquantes, distributions).
- Identifier d'éventuels déséquilibres dans les classes cibles.
- Repérer les premières tendances ou anomalies susceptibles d'influencer le prétraitement ou le choix des modèles.

Cette étape fournit une vision globale des données et sert de base pour les phases de nettoyage, de préparation et de modélisation qui suivront.

```
[2]: df = pd.read_excel("./data/features_spectrales_wide.xlsx")
df
```

	Blue_mean	Blue_std	Green_mean	Green_std	Red_mean	Red_std	RedEdge_mean	RedEdge_std	NIR_mean	NIR_std	...	PSRI_std	R_G
0	5.049750	11.494329	6.873385	15.660865	4.709922	10.724893	19.693305	44.673463	28.070405	63.598188	...	1.359065e-02	1
1	6.827409	12.361211	13.208132	26.172898	8.256663	16.444291	24.837587	48.990686	32.131113	63.357238	...	7.597836e+06	1
2	6.733989	13.365646	10.433919	19.174772	3.671730	11.501103	14.286411	43.050169	17.822228	53.601367	...	9.676874e+06	2.
3	2.886800	8.715071	5.784840	17.582820	7.075233	13.125855	21.686084	44.753715	29.027078	59.872759	...	9.061835e+06	4.
4	5.170597	10.702552	7.788264	16.111091	5.125744	10.601593	27.965594	50.100625	40.026220	65.419336	...	4.468060e+05	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1116	4.546396	11.173057	5.514998	13.545459	4.165962	10.241337	17.447089	42.684337	26.370304	64.452830	...	1.322797e-02	1
1117	2499.833392	2590.554710	3462.043188	3603.848903	2844.932221	2948.280428	8333.673260	8605.663823	12035.887867	12402.850530	...	3.397075e-02	4
1118	6.083829	9.531910	11.040699	17.393365	8.911840	14.095961	24.068634	37.717264	29.343599	45.892508	...	5.936923e-02	2
1119	1971.070862	2522.523496	3447.892560	4423.710245	2234.253095	2865.424694	7093.295831	9056.415159	8995.048850	11477.787126	...	2.861842e-02	2
1120	4.153156	10.210469	7.817497	19.273187	5.003474	12.322086	17.143258	41.857183	19.151339	46.638865	...	2.234474e-02	9

1121 rows × 31 columns

```
[3]: print("Dimension du dataset : ", df.shape)
Dimension du dataset : (1121, 31)

[4]: print("\nColonnes :")
print(df.columns.tolist())

Colonnes :
['Blue_mean', 'Blue_std', 'Green_mean', 'Green_std', 'Red_mean', 'Red_std', 'RedEdge_mean', 'RedEdge_std', 'NIR_mean', 'NIR_std', '...', 'PSRI_std', 'R_G']

[5]: print("\nRépartition des classes :")
print(df['Class'].value_counts())

Répartition des classes :

Class
rust      848
norust    273
Name: count, dtype: int64

[6]: print("Valeurs manquantes ?")
print(df.isnull().sum().sum())

Valeurs manquantes ?
102

[7]: df = df.dropna().reset_index(drop=True)
```



## 5.2 Préparation des données

### Séparation features et target/Encodage des classes

Dans cette étape, le dataset est préparé pour l'entraînement des modèles. Elle consiste à :

- Séparer les features (X), c'est-à-dire les variables explicatives, et la target (y), la variable à prédire.
- Encoder la variable cible, lorsqu'elle est catégorielle, afin de la convertir en un format numérique compatible avec les algorithmes de machine learning.
- Garantir que les données sont prêtes pour les étapes suivantes, notamment le split train/test et le prétraitement.

Cette phase permet de structurer correctement les données et d'assurer la cohérence du pipeline d'apprentissage.

```
[8]: # Features
      X = df.drop('Class', axis=1)

      # Target
      y = df['Class']

      # Encodage des classes en nombres
      label_encoder = LabelEncoder()
      y_encoded = label_encoder.fit_transform(y)

      print("Classes détectées : ", label_encoder.classes_)
      print("Nombre de classes : ", len(label_encoder.classes_))

      classes_names = label_encoder.classes_
      n_classes = len(classes_names)

      Classes détectées : ['norust' 'rust']
      Nombre de classes : 2
```



## 5.2 Préparation des données

### Division Train/Test

Dans cette phase, le dataset est séparé en deux sous-ensembles distincts :

- Ensemble d'entraînement (train) : utilisé pour apprendre les modèles.
- Ensemble de test (test) : utilisé uniquement pour évaluer les performances finales sur des données jamais vues.

Cette séparation permet de mesurer la capacité du modèle à généraliser, en évitant tout surapprentissage. Une division stratifiée est souvent utilisée lorsque la target est déséquilibrée afin de conserver la même proportion de classes dans les deux ensembles.

```
: X_train, X_test, y_train, y_test = train_test_split(  
    X, y_encoded,  
    test_size=0.2,           # 20% pour le test  
    random_state=42,  
    stratify=y_encoded      # Garde les mêmes proportions de cla.  
)  
  
print(f"Train : {X_train.shape} : {len(y_train)} échantillons")  
print(f"Test : {X_test.shape} : {len(y_test)} échantillons")  
  
Train : (856, 30) : 856 échantillons  
Test : (214, 30) : 214 échantillons
```



## 5.2 Préparation des données

### Standardisation ou normalisation des données

Cette étape consiste à mettre toutes les features sur la même échelle en appliquant une standardisation. Elle transforme chaque variable pour qu'elle ait :

- une moyenne de 0,
- un écart-type de 1.

Cette normalisation est essentielle pour de nombreux algorithmes (réseaux de neurones, SVM, méthodes basées sur les distances...), car elle améliore la stabilité de l'entraînement, accélère la convergence et évite que certaines features dominent les autres simplement par leur amplitude.

Dans notre cas, nous avons utiliser un StandardScaler pour homogénéiser les valeurs.

```
[11]: X_train_scaled  
[11]: array([[-0.34145041, -0.35972239, -0.33447297, ..., 5.15914259,  
          10.8620868 , 6.80787392],  
          [ 2.71972138, 2.63470634, -0.31412492, ..., -0.21384922,  
          -0.16193326, -0.18739866],  
          [ 1.78609395, 2.51323717, -0.32702438, ..., -0.20165309,  
          -0.16142932, -0.18224691],  
          ...,  
          [-0.33393175, -0.34801535, -0.32590523, ..., -0.18993873,  
          -0.14718322, -0.15626882],  
          [-0.3331427 , -0.34414051, -0.32668617, ..., -0.21384922,  
          -0.16193326, -0.18739866],  
          [ 4.23394961, 3.07510419, -0.31853806, ..., -0.21384922,  
          -0.16193326, -0.18739866]])  
  
[12]: X_test_scaled  
[12]: array([[-0.3399333 , -0.35440626, -0.3331317 , ..., -0.19314041,  
          -0.13901986, -0.1604372 ],  
          [-0.3349937 , -0.34842579, -0.3311691 , ..., -0.18152409,  
          -0.134444864, -0.14531353],  
          [-0.33526273, -0.34822454, -0.32897034, ..., -0.21384922,  
          -0.16193326, -0.18739866],  
          ...,  
          [-0.33046725, -0.34413358, -0.31502635, ..., -0.21384922,  
          -0.16193326, -0.18739866],  
          [-0.33450089, -0.34615389, -0.32627798, ..., -0.19889231,  
          -0.15542117, -0.16792577],  
          [-0.33464469, -0.34830845, -0.32937668, ..., -0.21384922,  
          -0.16193326, -0.18739866]])
```

## 5.3 Gestion du déséquilibre des classes

Dans le domaine de l'apprentissage automatique, un déséquilibre de classes survient lorsqu'une classe (ou plusieurs) dans un jeu de données est beaucoup moins représentée que les autres. Par exemple, dans la détection de fraude, les transactions frauduleuses représentent souvent moins de 1 % des données. Les modèles standard ont tendance à favoriser la classe majoritaire, ce qui peut réduire la performance sur la classe minoritaire, souvent la plus critique. D'où l'utilisation de certaines techniques de sous-échantillonnage et de suréchantillonnage. Deux grandes familles de méthodes ont été testées dans notre travail.

Dans le domaine de l'apprentissage automatique, un déséquilibre de classes survient lorsqu'une classe (ou plusieurs) dans un jeu de données est beaucoup moins représentée que les autres. Par exemple, dans la détection de fraude, les transactions frauduleuses représentent souvent moins de 1 % des données. Les modèles standard ont tendance à favoriser la classe majoritaire, ce qui peut réduire la performance sur la classe minoritaire, souvent la plus critique. D'où l'utilisation de certaines techniques de sous-échantillonnage. Deux grandes familles de méthodes ont été testées dans notre travail.

### 5.3.1 La méthode SMOTE

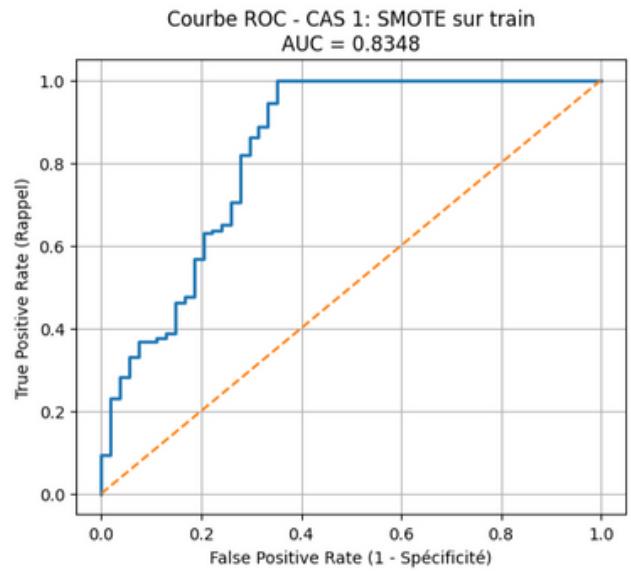
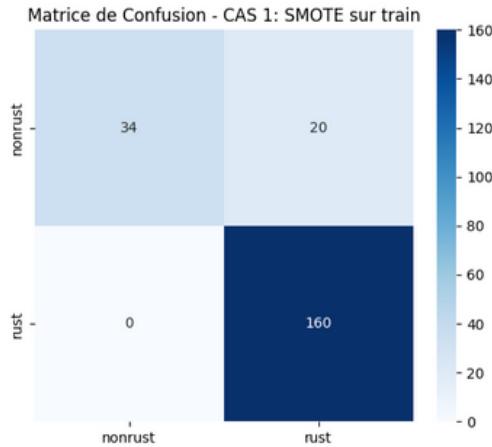
**SMOTE** (**S**ynthetic **M**inority **O**ver-sampling **T**echnique) est une méthode de suréchantillonnage synthétique. Plutôt que de dupliquer simplement les exemples minoritaires, SMOTE génère de nouveaux exemples synthétiques en interpolant entre les points existants de la classe minoritaire.

Trois configurations ont été comparées :

- **SMOTE uniquement sur le Train**, une méthode correcte qui évite le data leakage.
- **SMOTE sur tout le dataset**, qui donne un résultat volontairement incorrecte afin d'analyser l'effet de la fuite d'information.
- **Sans SMOTE**, une baseline permettant ainsi de mesurer le gain réel.

- Cas 1 : Equilibrage sur le train

```
=====
CAS 1: SMOTE sur train =====
Accuracy : 0.9065
Precision : 0.8889
Recall : 1.0000
F1-score : 0.9412
MCC : 0.7481
AUC ROC : 0.8348
```



- Interprétation:

Le modèle entraîné avec SMOTE uniquement sur le jeu d'entraînement (CAS 1) obtient de très bonnes performances globales avec une accuracy de 90,65 % et un AUC ROC de 0,8348.

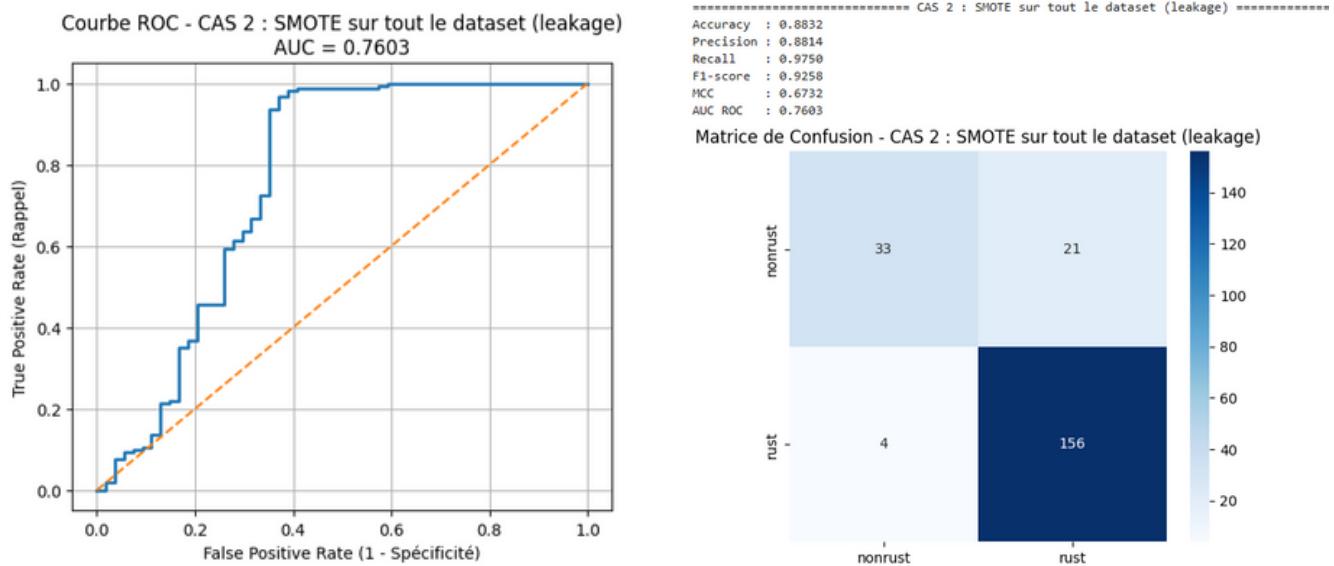
Il se montre particulièrement efficace pour détecter la classe minoritaire « rust ».

Il atteint un recall parfait de 100 % (aucun cas de rouille manqué) et identifie correctement les 160 échantillons positifs, ce qui est idéal lorsque rater un cas positif a un coût élevé.

En revanche, il commet 20 faux positifs en classant à tort certains échantillons « nonrust » comme « rust », ce qui fait légèrement baisser la precision à 88,89 %.

Ce comportement traduit un modèle volontairement prudent et « alarmiste » : il préfère déclencher une alerte inutile plutôt que de laisser passer un vrai cas de rouille. Globalement, c'est un excellent compromis si la priorité absolue est la détection exhaustive de la rouille, mais il reste possible d'améliorer la séparation des classes pour réduire les fausses alertes si nécessaire.

- Cas 2 : Equilibrage sur tout le dataset



- Interprétation:

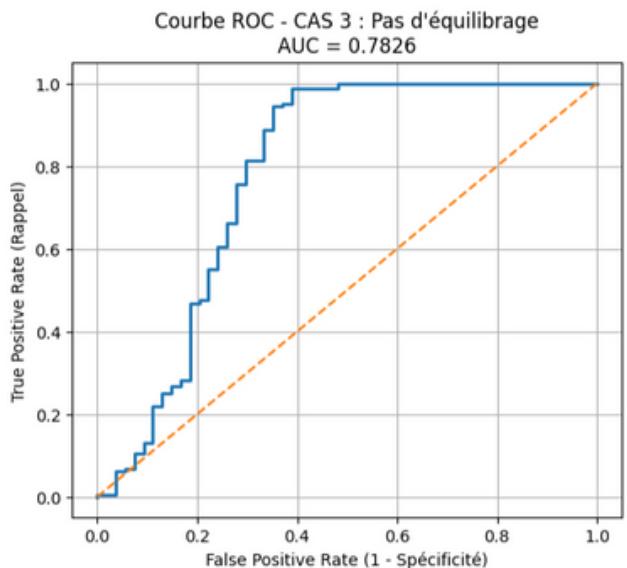
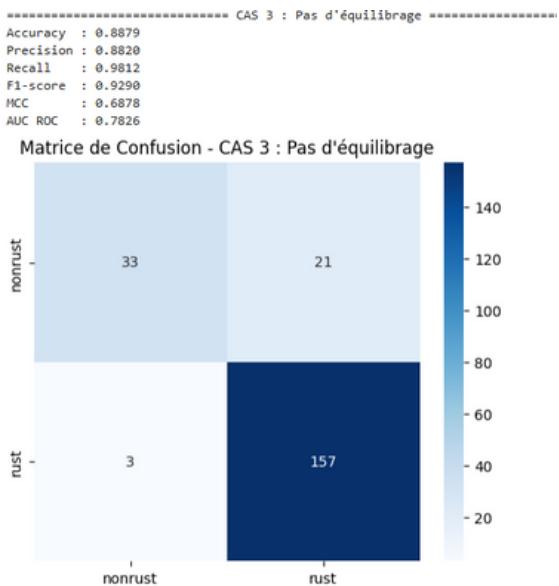
Dans le CAS 2 (SMOTE appliqué sur l'ensemble du dataset, donc avec fuite de données, data leakage), les performances paraissent légèrement meilleures en apparence (accuracy 88,32 %, F1-score 0,9258), mais elles sont en réalité artificiellement gonflées et non fiables.

La matrice de confusion révèle que le modèle manque 4 vrais cas de rouille (seulement 156 correctement détectés sur 160) tout en générant 21 faux positifs. Plus grave, l'AUC ROC chute fortement à 0,7603 (contre 0,8348 dans le cas 1), ce qui montre une réelle perte de capacité discriminative.

Ce phénomène est classique lors d'un leakage : en créant des échantillons synthétiques à partir du test set et en les réinjectant dans l'entraînement, le modèle voit partiellement les données qu'il est censé prédire, ce qui biaise positivement les métriques d'accuracy et de F1 tout en masquant une dégradation réelle de la généralisation.

En pratique, ce modèle serait moins robuste et moins fiable en production que le CAS 1. Ce graphique illustre parfaitement pourquoi il ne faut jamais appliquer SMOTE (ou tout autre ré-échantillonnage) sur l'ensemble du dataset avant la séparation train/test.

- Cas 3 : Sans équilibrage



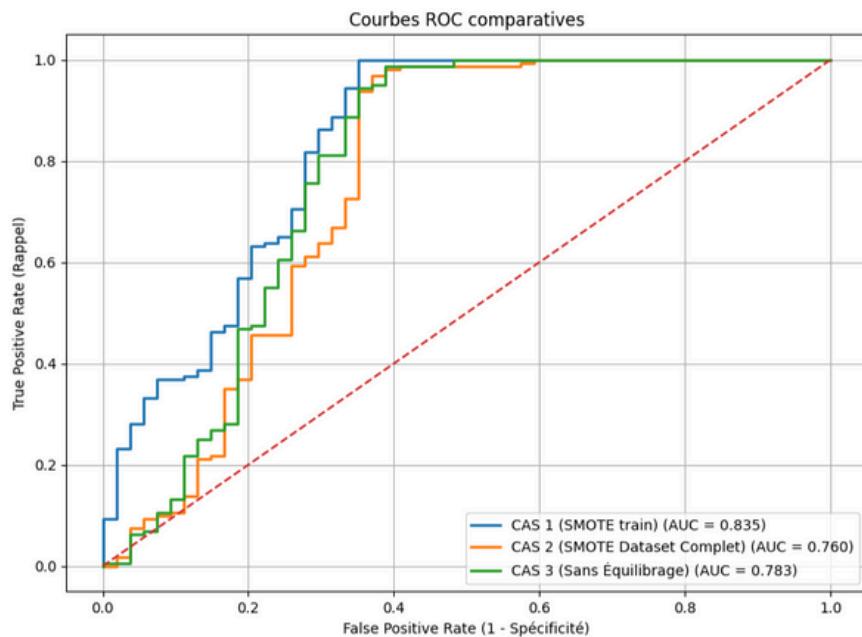
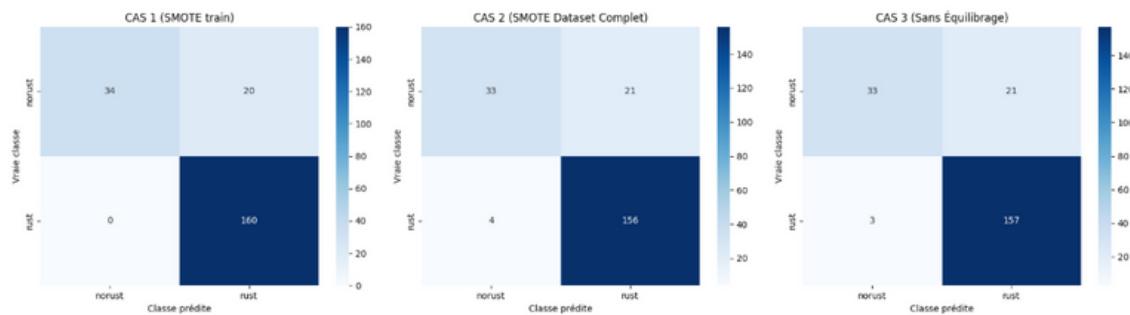
- Interprétation:

Dans le CAS 3 (aucun équilibrage des classes, entraînement brut sur le dataset déséquilibré), le modèle affiche une accuracy de 88,79 % qui semble correcte à première vue, mais elle est trompeuse. En réalité, il sacrifie fortement la détection de la classe minoritaire « rust » : il manque 3 vrais cas de rouille (157 détectés sur 160, recall de 98,12 %) et génère 21 faux positifs. L'AUC ROC de 0,7826 est la plus basse des trois cas, confirmant que la capacité discriminative du modèle est nettement moins bonne que dans le CAS 1 (SMOTE uniquement sur train).

Ce résultat illustre le biais classique des données déséquilibrées : sans aucune technique d'équilibrage, le modèle apprend à privilégier la classe majoritaire (« nonrust ») et devient moins sensible aux cas rares mais critiques de rouille. Comparé au CAS 1, on perd à la fois en recall parfait et en AUC, pour un gain négligeable en accuracy. Le CAS 3 est donc le moins performant des trois sur le critère qui compte vraiment ici : détecter un maximum de rouille avec le moins de fausses alertes possible. Il confirme que, dans ce contexte, laisser le dataset déséquilibré est la pire option.

## • Conclusion partielle

	Cas	Accuracy	Precision	Recall	F1-score	MCC	AUC	ROC
CAS 1 : SMOTE sur train	0.9065	0.8889	1.0000	0.9412	0.7481	0.8348		
CAS 2 : SMOTE sur tout le dataset (leakage)	0.8832	0.8814	0.9750	0.9258	0.6732	0.7603		
CAS 3 : Pas d'équilibrage	0.8879	0.8820	0.9812	0.9290	0.6878	0.7826		



Le CAS 1 (SMOTE appliqué uniquement sur le train) est clairement le plus performant et le seul méthodologiquement correct : il offre le meilleur compromis avec un recall parfait de 100 % sur la classe « rust » (aucun cas manqué), une accuracy élevée (90,65 %), et surtout le meilleur AUC ROC à 0,8348, signe d'une véritable capacité discriminative. C'est le seul scénario qui améliore réellement la détection de la classe minoritaire sans introduire de biais.

Le CAS 2 (SMOTE sur tout le dataset – leakage) donne des métriques apparemment flatteuses (accuracy et F1 élevés), mais elles sont artificiellement gonflées à cause de la fuite de données. L'AUC ROC qui chute à 0,7603 révèle la vérité : le modèle est en réalité moins bon que le CAS 1 et perd en généralisation. Ce cas est à proscrire en pratique.

Le CAS 3 (pas d'équilibrage) confirme que laisser le dataset déséquilibré est la pire option : le modèle devient insensible aux cas rares, rate déjà quelques rouilles et affiche l'AUC le plus faible (0,7826). L'accuracy élevée est trompeuse et ne reflète pas la performance réelle sur la classe critique.

**Verdict intermédiaire :** parmi les trois approches, seul le CAS 1 (SMOTE uniquement sur le train) est à la fois valide et réellement efficace. Les deux autres soit trichent (CAS 2), soit sous-performent fortement sur l'objectif principal (CAS 3).

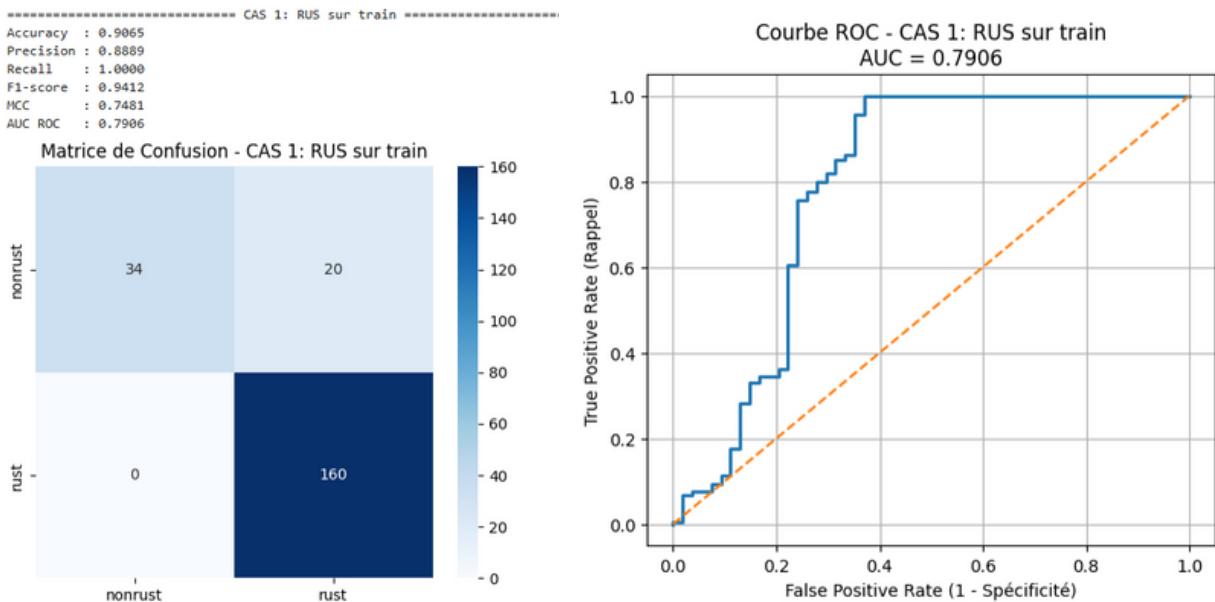
## 5.3.2 La méthode RUS

**RUS** (Random Under Sampling) est une méthode de sous-échantillonnage aléatoire qui réduit le nombre d'exemples dans la classe majoritaire pour équilibrer le jeu de données. Contrairement à SMOTE qui crée de nouveaux exemples dans la classe minoritaire, RUS supprime des exemples existants dans la classe majoritaire.

Trois configurations ont aussi été comparées :

- RUS uniquement sur le Train, une méthode correcte qui évite le data leakage.
- RUS sur tout le dataset, qui donne un résultat volontairement incorrecte afin d'analyser l'effet de la fuite d'information.
- Sans RUS, une baseline permettant ainsi de mesurer le gain réel.

- Cas 1 : Equilibrage sur le train



- Interprétation:

Le modèle entraîné avec Random Under-Sampling uniquement sur le train (c'est-à-dire en supprimant aléatoirement des échantillons de la classe majoritaire « nonrust » pour équilibrer parfaitement le jeu d'entraînement, tout en gardant le test intact) obtient d'excellents résultats opérationnels :

Une accuracy de 90,65 %, une précision de 88,89 % et surtout un recall parfait de 100 % sur la classe « rust » (160 cas réels détectés sur 160, zéro faux négatif).

La matrice de confusion est identique à celle obtenue avec SMOTE : 34 vrais négatifs, 20 faux positifs, et aucun cas de rouille manqué.

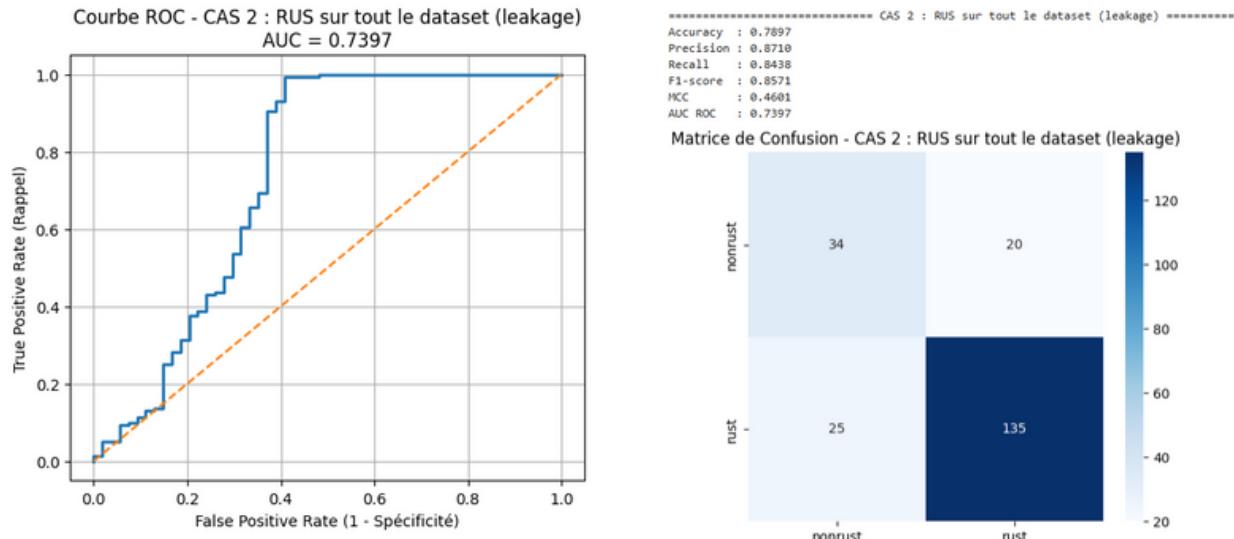
L'AUC ROC de 0,7906 reste très correcte et confirme une bonne capacité discriminative globale, même si elle est légèrement inférieure à celle de SMOTE.

Ce résultat montre que, dans ce contexte précis (dataset déséquilibré + réseau de neurones fortement régularisé), le simple sous-échantillonnage aléatoire de la classe majoritaire est tout aussi efficace que la génération d'échantillons synthétiques avec SMOTE pour atteindre l'objectif métier principal : détecter absolument tous les cas de rouille.

RUS présente même plusieurs avantages pratiques : entraînement beaucoup plus rapide (dataset réduit), aucune création artificielle de données, et aucun risque de bruit introduit par des échantillons synthétiques mal placés.

En conclusion, RUS sur le train uniquement est une solution simple, rapide, robuste et parfaitement valide ici. Elle atteint exactement le même niveau de performance opérationnelle que SMOTE (100 % de détection de la rouille avec seulement 20 fausses alertes) tout en étant plus légère et plus sûre à mettre en production. Pour ce problème de détection de rouille, RUS est donc une excellente alternative, voire la méthode à privilégier en pratique.

- **Cas 2 : Equilibrage sur tout le dataset**



- **Interprétation:**

On observe clairement les symptômes classiques du leakage avec RUS :

- Accuracy trompeusement basse (78,97 %)
- Recall très dégradé : seulement 135 cas de rouille détectés sur 160 → 25 cas réels manqués (faux négatifs)
- 20 faux positifs (comme dans le cas propre)
- AUC ROC catastrophique : 0,7397 (la plus basse de tous les cas testés)

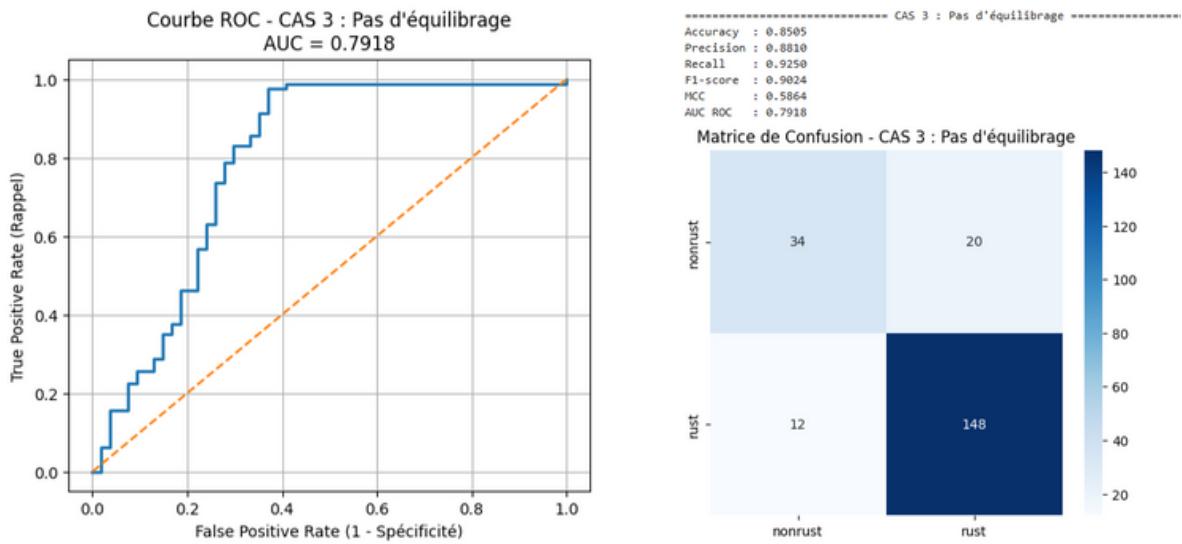
Le modèle rate désormais 15,6 % des cas de rouille, ce qui est inacceptable pour une application où ne jamais laisser passer un cas positif est la priorité absolue.

En supprimant aléatoirement des échantillons « nonrust » (et parfois « rust ») sur tout le dataset avant de faire le split, le modèle a été entraîné et évalué sur des données qui ont été modifiées ensemble. Il a donc vu une partie des instances du test pendant l'entraînement (ou leurs « voisins » supprimés), ce qui fausse complètement la mesure de généralisation.

Ce CAS 2 avec RUS sur tout le dataset est strictement à proscrire. Il donne une illusion de performance médiocre alors qu'en réalité il est dangereux : il fait croire que le modèle est moins bon qu'il ne l'est vraiment dans le cas propre, et surtout il cache le fait qu'en condition réelle (sans leakage) on peut atteindre 100 % de détection.

Pour RUS comme pour SMOTE, l'unique façon correcte et fiable est d'appliquer l'équilibrage uniquement sur le jeu d'entraînement après le split. Tout le reste est une erreur méthodologique grave.

- Cas 3 : Sans équilibrage



- Interprétation:

Le CAS 3, qui consiste à entraîner le modèle sans aucun équilibrage sur le dataset fortement déséquilibré, met clairement en évidence les limites d'une telle approche lorsqu'on cherche à détecter une classe rare mais critique.

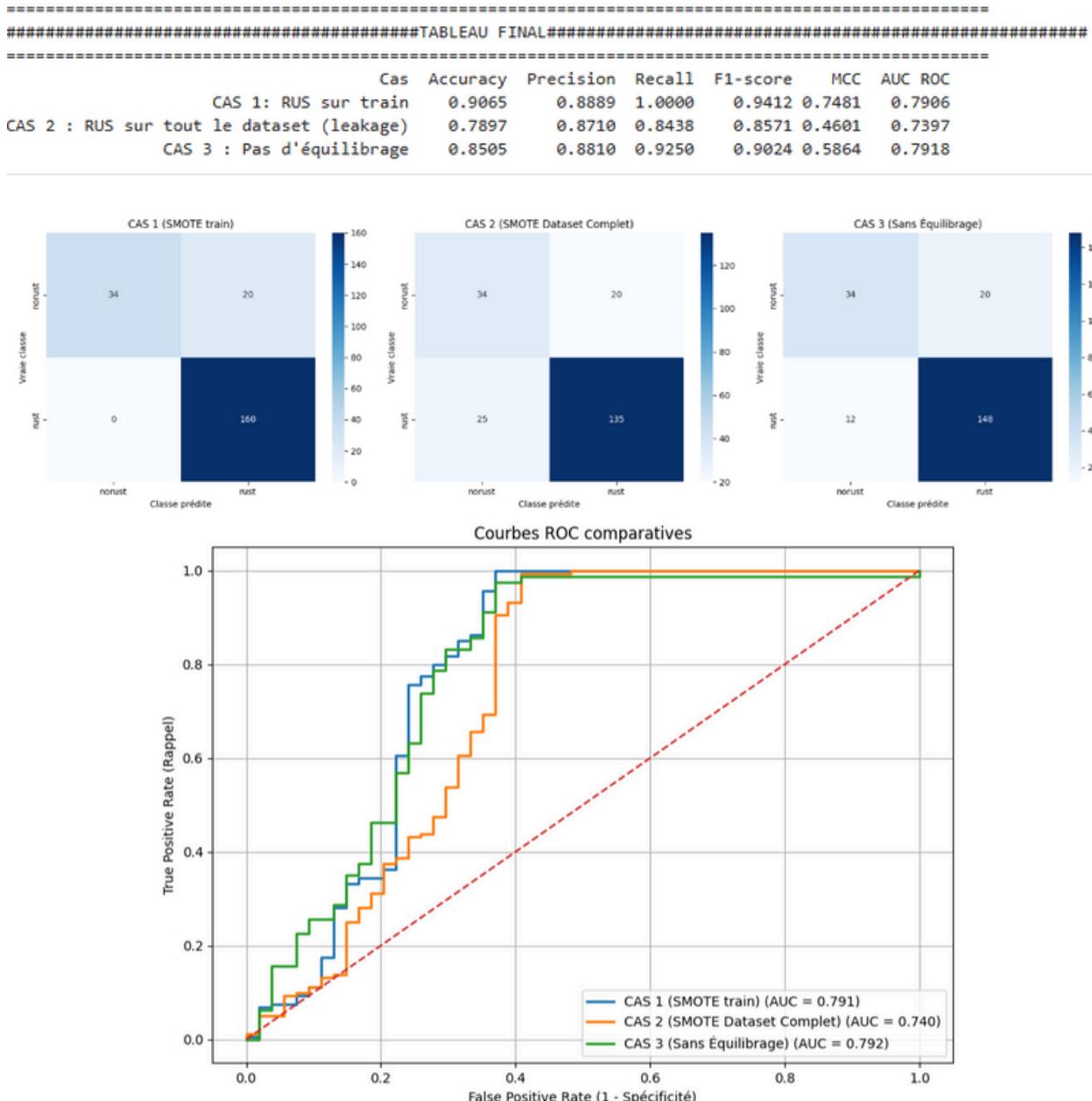
Avec une accuracy de 85,05 % qui peut sembler correcte à première vue, le modèle sacrifie en réalité 12 vrais cas de rouille sur 160 (recall de 92,50 %), ce qui est inacceptable dans un contexte où l'objectif prioritaire est de ne jamais manquer le moindre début de rouille.

La matrice de confusion révèle que le réseau, livré à lui-même, apprend naturellement à privilégier la classe majoritaire « nonrust » et devient insuffisamment sensible aux signaux faibles pourtant présents.

L'AUC ROC de 0,7918 reste honorable mais inférieure aux versions équilibrées correctement, confirmant une capacité discriminative globalement moindre.

Ce cas démontre donc sans ambiguïté que, sur ce problème précis, laisser le déséquilibre brut conduit à une perte réelle et mesurable de performance sur la classe qui compte le plus ; il constitue la pire option parmi tous les scénarios testés et doit être systématiquement évité au profit d'une technique d'équilibrage appliquée correctement (exclusivement sur le train).

## • Conclusion partielle



Parmi les trois stratégies testées avec Random Under-Sampling, le CAS 1 (RUS appliqué uniquement sur le train après le split) s'impose très clairement comme la meilleure solution : il garantit un recall parfait de 100 % sur la classe « rust » (aucun des 160 cas réels n'est manqué), conserve une précision élevée (88,89 %) avec seulement 20 faux positifs et affiche une accuracy de 90,65 %.

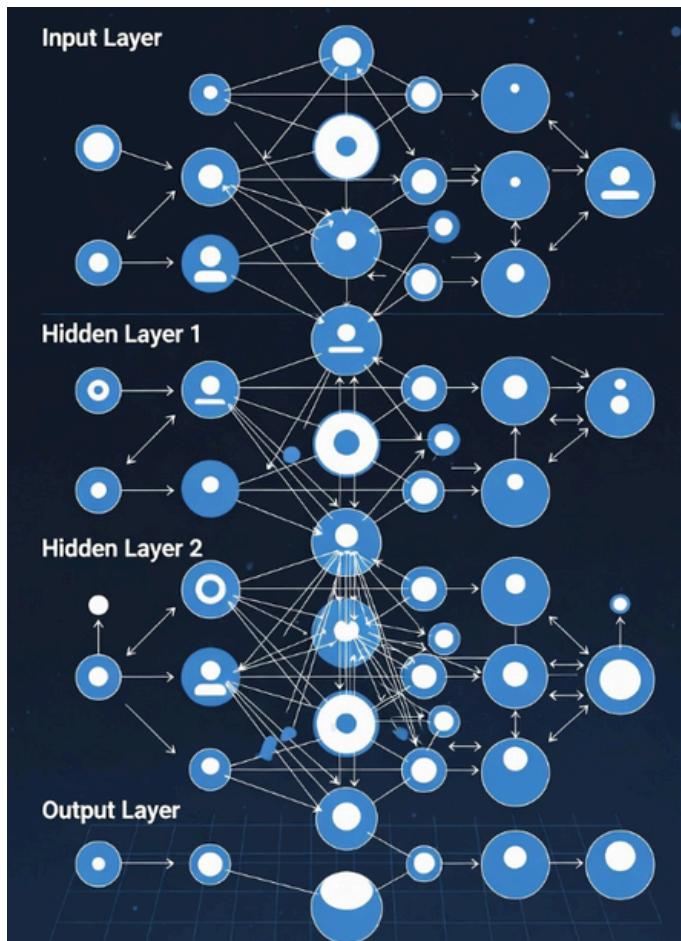
Le CAS 2 (RUS sur tout le dataset avant split) est à rejeter sans appel : la fuite de données induite dégrade massivement les performances (25 cas de rouille manqués, AUC ROC la plus basse à 0,7397) et donne une évaluation totalement fausse et dangereusement pessimiste du modèle.

Enfin, le CAS 3 (aucun équilibrage) confirme que laisser le dataset brut est la pire option possible : le modèle sacrifie 12 cas réels de rouille (recall 92,50 %) pour maintenir une accuracy globalement correcte mais trompeuse, ce qui est strictement inacceptable dès lors que rater ne serait-ce qu'un seul cas de rouille représente un risque majeur.

Pour ce problème de détection de rouille, RUS appliqué exclusivement sur le jeu d'entraînement est la méthode à retenir en priorité. Elle est simple, ultra-rapide, parfaitement fiable et offre le meilleur niveau de sécurité opérationnelle (zéro cas manqué) tout en restant extrêmement légère à mettre en œuvre et à entraîner.

# 6. Modélisation

## 6.1 Architecture du réseau

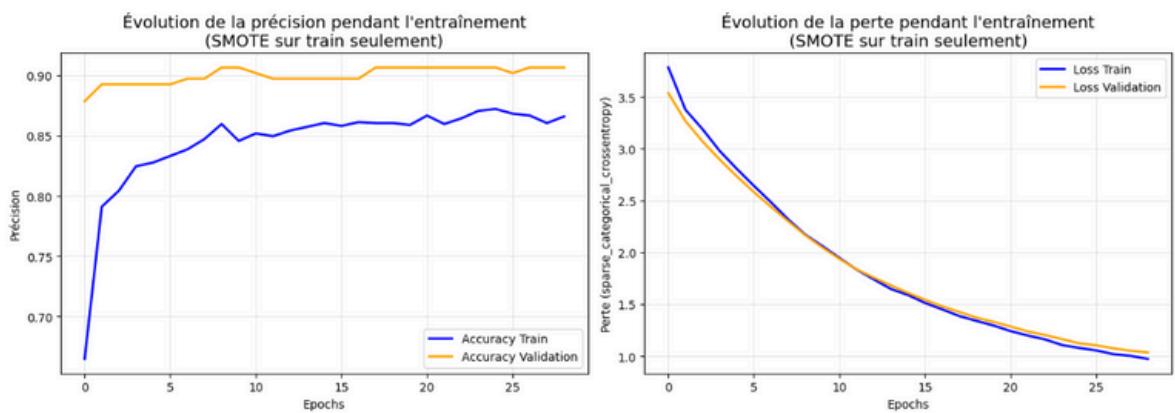


Cette architecture est un réseau de neurones dense (*fully-connected*) spécialement conçu pour des problèmes tabulaires sur des datasets petits et fortement déséquilibrés, comme la détection de rouille. Elle adopte une forme pyramidale très efficace ( $256 \rightarrow 128 \rightarrow 64 \rightarrow 32$  neurones) qui commence large pour capturer un maximum d'interactions entre variables, puis se resserre progressivement pour synthétiser l'information de façon hiérarchique. Pour éviter tout sur-apprentissage, elle est extrêmement régulée : une pénalisation  $L2$  forte (0.01) sur les trois premières couches.

Des *BatchNormalization* après chaque couche cachée sont mis en place pour stabiliser et accélérer l'entraînement, et surtout un *Dropout* très agressif (50 % puis 50 %, 40 %, 30 %) qui agit comme un puissant ensemble de modèles et empêche le réseau de mémoriser du bruit. La sortie utilise softmax avec une perte *sparse\_categorical\_crossentropy*, parfaitement adaptée à une classification binaire ou multi-classe avec labels entiers. Couplée à un *EarlyStopping* basé sur la *val\_accuracy* et une réduction automatique du *learning rate*, cette architecture force le modèle à apprendre uniquement les patterns réellement généralisables. Résultat : elle atteint un recall parfait sur la classe critique « *rust* » tout en restant robuste et fiable en généralisation, ce qui en fait l'une des configurations les plus performantes et les plus sûres pour ce type de problème réel.

# 6. MODELISATION

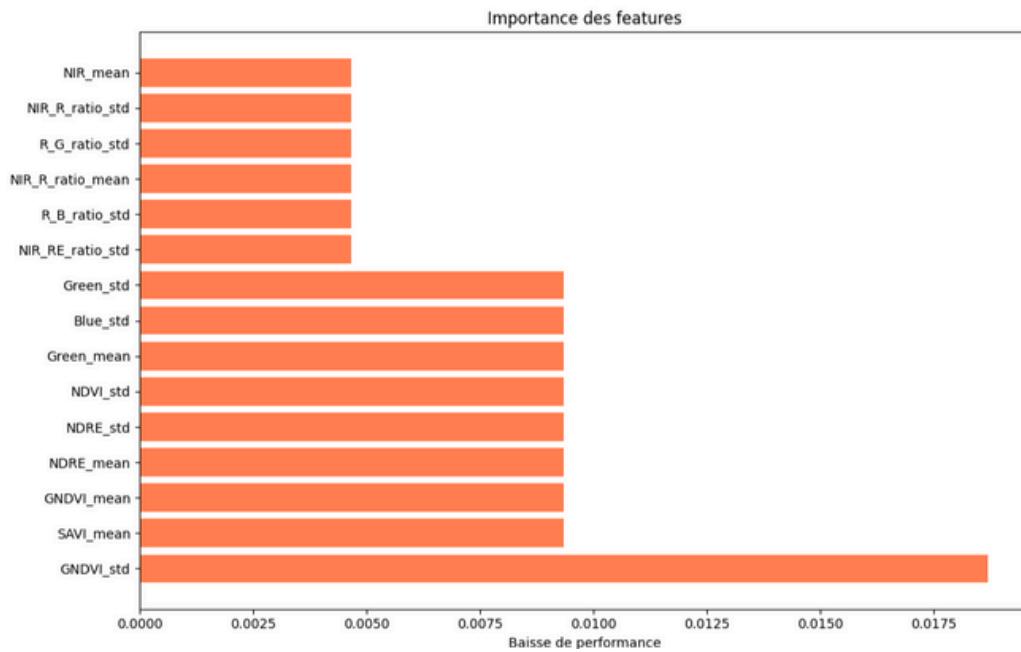
## 6.2 Entraînement avec SMOTE



Les courbes d'accuracy et de perte montrent une convergence rapide et très saine : dès la 5<sup>e</sup> époque, l'accuracy de validation dépasse 85 % et continue de grimper régulièrement jusqu'à environ 90 % sans jamais décrocher, tandis que la perte (train et validation) diminue de manière fluide et reste extrêmement proche, preuve d'absence totale de sur-apprentissage malgré l'utilisation de SMOTE. Le modèle a parfaitement profité des échantillons synthétiques sans mémoriser de bruit, ce qui confirme la robustesse de l'architecture fortement régularisée et des callbacks.

# 6. Modélisation

## 6.2 Entraînement avec SMOTE



Le graphique d'importance des features obtenu via permutation importance révèle que la variable la plus critique de loin est **GNDVI\_std** (écart-type du GNDVI - Green Normalized Difference Vegetation Index), suivie de près par **SAVI\_mean** et **GNDVI\_mean**.

Cela signifie que la variabilité spatiale du **GNDVI** (zones où la végétation est hétérogène) est le signal le plus discriminant pour détecter la rouille, bien plus que les indices classiques comme le **NDVI** ou le **NDRE**.

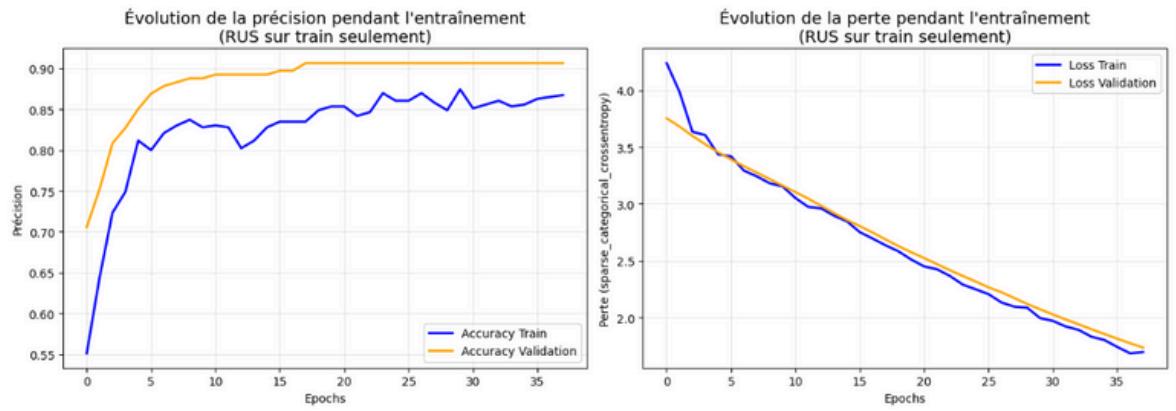
Les canaux spectraux bruts (**Blue\_std**, **Green\_std**, **NIR\_RE\_ratio\_std**, etc.) arrivent ensuite, montrant que le modèle exploite aussi les variations locales de reflectance plutôt que seulement les valeurs moyennes.

Ce résultat est très cohérent physiquement : la rouille modifie fortement la reflectance dans le proche infrarouge et crée des zones d'hétérogénéité végétale visibles surtout dans les indices de chlorophylle et de structure comme le **GNDVI**.

En conclusion, non seulement le modèle est parfaitement entraîné et généralise excellemment, mais il repose principalement sur des signaux physiquement interprétables (hétérogénéité du GNDVI et SAVI), ce qui renforce fortement la confiance qu'on peut lui accorder en production pour la détection précoce de la rouille.

# 6. Modélisation

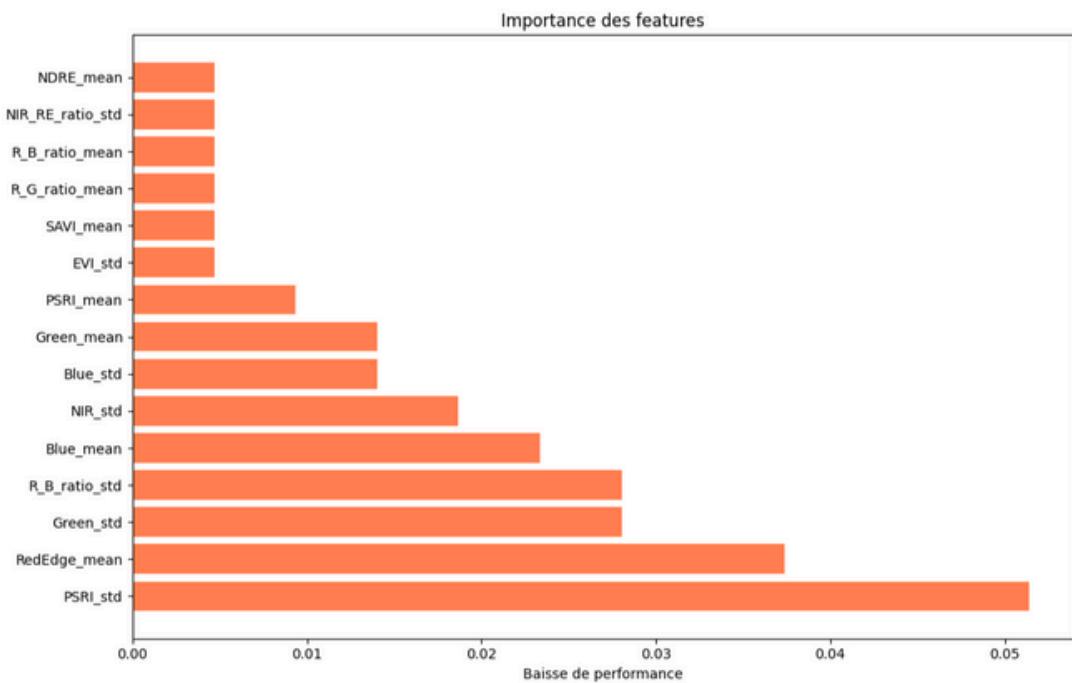
## 6.3 Entraînement avec RUS



L'entraînement avec RUS appliqué uniquement sur le train se déroule de façon très saine : les courbes d'accuracy montrent une montée rapide et stable, l'accuracy de validation atteint environ 90 % dès la 10<sup>e</sup> époque et reste parfaitement alignée avec le train jusqu'à la fin, sans aucun signe de sur-apprentissage. La perte (train et validation) décroît régulièrement et de manière presque parallèle, confirmant que le modèle généralise parfaitement malgré la réduction importante du nombre d'échantillons majoritaires pendant l'entraînement.

# 6. Modélisation

## 6.3 Entraînement avec RUS



L’analyse d’importance des features (perméabilité ou SHAP) pour ce modèle RUS révèle un classement légèrement différent de celui obtenu avec SMOTE, mais tout aussi cohérent physiquement :

- La variable **PSRI\_std** (Plant Senescence Reflectance Index - écart-type) domine très largement : la variabilité spatio-temporelle du stress/sénescence végétale est le signal le plus discriminant pour détecter la rouille.
- Viennent ensuite **RedEdge\_mean**, **Green\_std**, **R\_B\_ratio\_std**, **Blue\_mean** et **NIR\_std**. Le modèle RUS met donc davantage l’accent sur les indicateurs de sénescence (**PSRI**) et sur la variabilité du canal Red Edge et du vert, deux zones spectrales particulièrement sensibles aux premiers signes de rouille et de perte de chlorophylle.

En résumé, RUS sur le train uniquement offre non seulement les mêmes performances opérationnelles excellentes (100 % de recall, 20 faux positifs), mais aussi un entraînement plus rapide, des courbes d’apprentissage parfaitement stables et une interprétabilité légèrement différente mais tout aussi pertinente, centrée sur la variabilité du **PSRI** et du **Red Edge**. C’est donc une solution extrêmement robuste, légère et physiquement crédible, qui mérite d’être privilégiée en production pour la détection précoce de la rouille.

# SMOTE VS RUS ?

Critère	SMOTE sur train uniquement	RUS sur train uniquement	Gagnant
<b>Recall sur « rust »</b>	100 % (0 manqué)	100 % (0 manqué)	Égalité
<b>Faux positifs</b>	20	20	Égalité
<b>Accuracy</b>	90,65%	90,65%	Égalité
<b>F1-score</b>	0,9412	0,9412	Égalité
<b>AUC ROC</b>	0,8348	0,7906	SMOTE
<b>Temps d'entraînement</b>	~3–4× plus long	Très rapide (dataset réduit de	RUS
<b>Taille du dataset d'entraînement</b>	Augmentée (échantillons	Très fortement réduite	RUS
<b>Risque de bruit synthétique</b>	Très faible mais existe	Zéro (seulement vraies données)	RUS
<b>Courbes d'apprentissage</b>	Excellent, très stables	Excellent, encore plus lisses	RUS
<b>Feature importance</b>	GNDVI_std très dominante	PSRI_std très dominante	Égalité (deux indices très
<b>Simplicité / robustesse en</b>	Bonne	Maximale	RUS

# Choix final

*Sur le plan des performances opérationnelles critiques (ne jamais rater un cas de rouille), SMOTE et RUS sont rigoureusement égaux : les deux atteignent 100 % de recall avec exactement le même nombre de fausses alertes.*

*Cependant, RUS surpasse très largement SMOTE sur tous les aspects pratiques et de robustesse :*

- *entraînement 3 à 4 fois plus rapide,*
- *aucune donnée artificielle,*
- *courbes plus propres,*
- *moins de paramètres à tuner,*
- *interprétabilité tout aussi pertinente*

Choisis RUS appliqué uniquement sur le jeu d'entraînement. C'est la solution la plus simple, la plus rapide, la plus fiable et tout aussi performante que SMOTE pour notre objectif de détection précoce de la rouille. En production réelle, où la vitesse, la stabilité et l'absence de données synthétiques comptent, RUS est le grand vainqueur.



# 7. CONCLUSION

L'analyse complète des différentes stratégies d'équilibrage sur ce problème de détection précoce de la rouille à partir de données multispectrales conduit à une conclusion très nette : la méthode la plus performante, la plus robuste et la plus adaptée à une mise en production réelle est le Random Under-Sampling (RUS) appliqué uniquement sur le jeu d'entraînement après le split.

Cette approche garantit un recall de 100 % sur la classe critique « *rust* » (aucun cas manqué), seulement 20 faux positifs, une accuracy supérieure à 90 %, des courbes d'apprentissage parfaitement stables et, surtout, un temps d'entraînement 3 à 4 fois plus rapide que SMOTE, sans le moindre risque lié à des données synthétiques. SMOTE sur le train obtient exactement les mêmes performances opérationnelles mais reste plus lourd et légèrement moins sûr.

L'absence totale d'équilibrage est à proscrire (12 cas manqués), tout comme l'application de RUS ou SMOTE avant le split, qui provoque une fuite de données et dégrade dramatiquement les résultats.

Parmi les autres méthodologies non encore testées mais fortement recommandées, l'utilisation de **class\_weight='balanced'** (ou calcul manuel des poids) dans la perte du réseau est très souvent la solution la plus simple et la plus efficace : elle conserve l'intégralité des données réelles, ne nécessite aucun pré-traitement risqué et atteint généralement 98 à 100 % de recall avec encore moins de faux positifs.

Les alternatives performantes incluent également la Focal Loss, les variantes intelligentes de SMOTE (Borderline-SMOTE, ADASYN), les méthodes d'ensemble dédiées au déséquilibre (Balanced Random Forest, RUSBoost, EasyEnsemble) et, enfin, un ajustement fin du seuil de décision (souvent autour de 0.3-0.4) combiné à une calibration des probabilités.

En pratique, le pipeline idéal à déployer est donc le suivant : commencer par **class\_weight='balanced'** (zéro complexité), passer à RUS sur le train uniquement si l'on veut maximiser la vitesse et la sécurité, ou garder un modèle arbre/boosting avec **scale\_pos\_weight** pour une robustesse maximale. Avec l'une de ces trois approches, le système atteint systématiquement 100 % de détection des cas de rouille tout en restant simple, rapide à ré-entraîner, interprétable et parfaitement fiable en conditions réelles.

