HESTIM
ENGINEERING & BUSINESS SCHOOL

Machine Learning

Rapport de soutenance

Filière : Génie informatique et intelligence artificielle

Machine Learning Supervisé de détection et de type de panne

Réalisé par :

Mohammed CHARKANI EL HASSANI

Sié Hans OUATTARA

&

Oloufemy AKPLOGAN

Encadré par :

Mme. HAIDRAR Saida

&

Mme. Lahlou Fatima Zahra

Année universitaire : 2024/2025

SOMMAIRE

LISTE DES FIGURES

INTRODUCTION GENERALE

Dans un environnement industriel de plus en plus axé sur l'automatisation, la performance et l'optimisation des processus, la gestion de la maintenance des équipements représente un défi majeur. Les arrêts imprévus des machines peuvent entraîner des pertes économiques significatives, altérer la qualité de la production et perturber les chaînes logistiques. C'est dans ce contexte que la maintenance prédictive émerge comme une solution innovante et stratégique, permettant d'anticiper les pannes avant qu'elles ne se produisent, grâce à l'analyse des données collectées en temps réel.

Les avancées récentes en intelligence artificielle, notamment dans le domaine de l'apprentissage automatique (Machine Learning), ont rendu possible le développement de modèles capables de détecter les signes précurseurs d'une défaillance. Ces modèles s'appuient sur des historiques de données provenant de capteurs, d'événements de maintenance et de paramètres techniques pour prédire l'état futur d'un équipement.

Le présent projet s'inscrit dans cette dynamique et vise à exploiter des données simulées concernant le fonctionnement de machines industrielles afin de concevoir des modèles prédictifs efficaces. Pour ce faire, deux approches complémentaires ont été explorées :

D’abord une classification binaire pour prédire si une machine est susceptible de tomber en panne. Ensuite une classification multiclasse pour identifier le type précis de défaillance qui pourrait survenir. Tout ça dans le but de réussir notre étude de la meilleure des façons possibles.

Chapitre 1 : Introduction au projet

Introduction

Le développement d'un système de maintenance prédictive repose sur une compréhension approfondie des principes de l'apprentissage automatique, ainsi que sur une organisation méthodique du projet. Ce premier chapitre a pour but de définir le cadre général de notre étude.

Dans un environnement industriel de plus en plus complexe, la prévision des défaillances machines est devenue une problématique stratégique. La maintenance prédictive, qui s'inscrit comme une évolution de la maintenance conditionnelle, permet d'anticiper les pannes potentielles en exploitant des données en temps réel. L'objectif n'est plus simplement de réparer une machine après une panne, mais d'intervenir avant qu'elle ne se produise, optimisant ainsi la disponibilité des équipements tout en réduisant les coûts d'entretien.

Dans cette perspective, les avancées en Machine Learning supervisé offrent des opportunités significatives. En alimentant les algorithmes avec des données historiques labellisées (indiquant un fonctionnement normal ou une panne identifiée), il devient possible de développer des modèles capables de prédire des situations à risque ou de classer le type de défaillance probable.

Ce chapitre présente donc le contexte général du projet, les objectifs visés, les sources des données utilisées, ainsi que la problématique métier à laquelle nous cherchons à répondre. Il établit ainsi les bases du processus analytique et technique qui sera développé tout au long du rapport.

1.1 Contexte général du Machine Learning supervisé

Le Machine Learning, ou apprentissage automatique, est un sous-domaine de l’intelligence artificielle qui permet à une machine d’apprendre à partir de données pour effectuer des prédictions ou prendre des décisions. Parmi les différentes approches du Machine Learning, l’apprentissage supervisé est l’un des plus répandus. Il repose sur l’utilisation de données étiquetées, c’est-à-dire des exemples pour lesquels la réponse attendue est déjà connue. Le modèle apprend ainsi à établir une relation entre les caractéristiques (features) et les étiquettes (labels), afin de pouvoir généraliser ce comportement sur de nouvelles données.

Dans un contexte industriel, le Machine Learning supervisé trouve de nombreuses applications, notamment dans le domaine de la maintenance des équipements. L’objectif est d’anticiper les défaillances potentielles pour optimiser la disponibilité des machines, réduire les coûts d’entretien, et améliorer la productivité. Cette approche est appelée maintenance prédictive.

1.2 Objectifs du projet

Ce projet s'inscrit pleinement dans cette logique de maintenance prédictive basée sur des techniques de Machine Learning. Plus précisément, les objectifs sont les suivants :

- Analyser et comprendre les données issues d'un environnement industriel simulé,
- Prétraiter et préparer ces données afin qu'elles soient exploitables par des algorithmes de classification,
- Développer et comparer différents modèles de Machine Learning, à la fois pour la classification binaire (machine en panne ou non) et la classification multiclasse (type de panne),
- Évaluer les performances des modèles à l'aide de métriques adaptées,
- Identifier le modèle le plus performant et proposer une démarche reproductible pour une future implémentation en milieu industriel.

Le projet se veut à la fois pédagogique et applicatif, dans le sens où il met en œuvre des concepts fondamentaux du Machine Learning tout en étant directement lié à un cas concret d'utilisation dans l'industrie.

1.3 Sources des données

Les données utilisées dans ce projet proviennent de jeux de données de Kaggle. Ils contiennent :

- Des mesures de température, de vibration, de rotation et de charge,
- Des caractéristiques sur les machines (âge, type de modèle),
- Et des étiquettes indiquant soit l'état de fonctionnement de la machine (classification binaire), soit le type précis de défaillance (classification multiclasse).

Ces jeux de données sont riches et réalistes, et permettent d'aborder la plupart des défis typiques rencontrés dans un vrai projet de data science industrielle.

1.4 Problématique

La problématique centrale de ce projet est la suivante :

Comment peut-on exploiter les données de fonctionnement d'un parc de machines pour prédire les pannes futures ou diagnostiquer le type de défaillance à l'aide d'algorithmes de Machine Learning ?

Cette question en soulève plusieurs autres, plus spécifiques :

Quelles sont les variables les plus pertinentes à analyser ?

Comment traiter des données déséquilibrées (certaines pannes étant rares) ?

Quels modèles sont les plus performants pour chaque type de tâche (binaire ou multiclasse) ?

Quelles métriques faut-il utiliser pour comparer ces modèles de manière juste et rigoureuse ?

1.5 Présentation des deux approches du projet

Afin de répondre à cette problématique, le projet a été scindé en deux approches complémentaires :

Une approche de **classification binaire**, qui consiste à prédire si une machine est susceptible de tomber en panne ou non. Cette tâche est particulièrement utile pour la détection préventive.

Une approche de **classification multiclasse**, dont le but est de prédire le type de panne parmi plusieurs catégories (p. ex. panne mécanique, électrique, etc.). Elle apporte une couche d'analyse supplémentaire permettant d'affiner les interventions de maintenance.

Chacune de ces approches à ses propres spécificités en termes de prétraitement, de choix de modèles et de métriques d'évaluation. C'est pourquoi elles ont été étudiées de manière séparée dans les notebooks, mais elles seront synthétisées ensemble dans le rapport final afin d'avoir une vue globale cohérente du projet.

Conclusion

Ce premier chapitre nous a permis de mieux situer notre projet dans son contexte industriel et technologique. Nous avons exposé les fondements de la maintenance prédictive et comment le Machine Learning supervisé peut être mobilisé pour atteindre cet objectif. À travers une problématique claire et des données structurées et pertinentes, nous avons défini les lignes directrices de notre démarche : améliorer la capacité d'anticipation des pannes machines tout en expérimentant différentes approches de modélisation (classification binaire et multiclasse).

Nous sommes maintenant prêts à entamer la phase technique, en commençant par une analyse approfondie des données et leur préparation, avant de passer à l'entraînement des modèles. Ces étapes seront abordées dans les chapitres suivants.

Chapitre 2 : Préparation et analyse exploratoire des données

2.1 Introduction générale

Avant de pouvoir entraîner un modèle de Machine Learning, il est crucial de préparer et d'explorer les données en profondeur. Cette phase constitue la base du projet, car la qualité des données détermine en grande partie la performance des modèles. Dans le cadre de la maintenance prédictive, cette étape revêt une importance particulière : les données industrielles peuvent être bruitées, déséquilibrées, ou contenir des variables peu pertinentes. Nous allons donc détailler les étapes communes et spécifiques aux deux approches du projet : classification binaire et classification multiclasse.

2.2 Chargement des données et premiers aperçus

Les deux notebooks utilisent des données au format CSV contenant des informations sur des machines industrielles simulées :

Données de capteurs (vibration, température, vitesse de rotation, etc.),

Informations sur les caractéristiques des machines (âge, type de modèle),

Variables cibles :

- Dans le cas binaire, une variable booléenne indiquant si la machine est tombée en panne ou non ;
- Dans le cas multiclasse, une variable catégorielle indiquant le type de panne.

Les premières étapes consistent à :

Importer les bibliothèques nécessaires (pandas, numpy, matplotlib, seaborn, etc.),

Charger les données avec pd.read_csv(),

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

```
csv_file = "predictive_maintenance.csv"
data = pd.read_csv(csv_file)
```

Figure 1 : Imploration et lecture de csv

Afficher les premières lignes (head()), les informations générales (info()), et vérifier les valeurs manquantes (isnull().sum()).

```
df.isnull().sum()
```

```
Type                        0
Air temperature [K]         0
Process temperature [K]     0
Rotational speed [rpm]      0
Torque [Nm]                 0
Tool wear [min]             0
Target                      0
Failure Type                0
dtype: int64
```

```
df.isna().sum()
```

```
Type                        0
Air temperature [K]         0
Process temperature [K]     0
Rotational speed [rpm]      0
Torque [Nm]                 0
Tool wear [min]             0
Target                      0
Failure Type                0
dtype: int64
```

Pas de valeurs manquantes

Figure 2 : Gestion des valeurs manquantes

Ces vérifications ont confirmé que les jeux de données sont propres, sans valeurs manquantes, et prêts à être analysés.

2.3 Analyse descriptive et visualisation

L'analyse descriptive permet de se faire une première idée sur la structure des données :

La distribution des variables numériques a été explorée à l'aide d'histogrammes et de describe().

Les corrélations entre les variables ont été étudiées avec une matrice de corrélation (heatmap), mettant en évidence les relations fortes entre certaines variables, ce qui peut guider la sélection des caractéristiques.

Des diagrammes en boîte ou boxplots ont été utilisés pour détecter d'éventuelles valeurs aberrantes.

Pour la classification multiclasse, une visualisation des classes cibles a été effectuée (barplots) pour montrer la répartition des différents types de panne.

Ces étapes ont permis de mieux comprendre la structure des données et d'identifier les premières tendances intéressantes, comme par exemple une relation entre la

température de fonctionnement et les pannes, ou une concentration de certains types de défaillances pour un modèle donné.

2.4 Encodage des variables catégorielles

Certaines variables, notamment le type de modèle de machine, sont de nature catégorielle. Pour que les algorithmes puissent les exploiter, un encodage a été nécessaire. Dans les notebooks, les méthodes suivantes ont été utilisées :

Label Encoding pour transformer les variables catégorielles en entiers binaires.

One-Hot Encoding pour créer des variables binaires pour chaque catégorie (souvent préféré dans le multiclasse pour éviter les ordres artificiels).

Cet encodage a permis d'inclure ces variables dans les modèles sans introduire de biais liés à la représentation des catégories.

2.5 Normalisation / Standardisation

Dans les deux approches, les variables numériques comme la température, la rotation ou les mesures de vibration n'étaient pas sur les mêmes échelles. Cela peut poser problème à certains modèles comme les KNN, les SVM ou les modèles basés sur la distance.

Ainsi, les données ont été standardisées à l'aide du StandardScaler, ce qui transforme chaque variable pour qu'elle ait une moyenne nulle et un écart-type de 1. Cela garantit une meilleure convergence et améliore souvent la performance des modèles.

2.6 Traitement des classes déséquilibrées

Un point important dans les deux notebooks est la gestion du déséquilibre des classes, particulièrement visible dans :

Le cas binaire, où les machines en panne représentent une minorité ;

Le cas multiclasse, où certaines catégories de panne sont beaucoup plus rares que d'autres.

Plusieurs stratégies ont été explorées :

Oversampling de la classe minoritaire (ex. : duplication ou génération synthétique via SMOTE),

Undersampling de la classe majoritaire,

Utilisation de modèles robustes au déséquilibre (par exemple, certains arbres de décision),

Poids de classe ajustés automatiquement dans certains algorithmes (class_weight='balanced' dans RandomForestClassifier, SVC, etc.).

Ces techniques ont permis de limiter les biais d'apprentissage et d'atteindre de meilleures performances sur les classes minoritaires, souvent les plus importantes dans une optique de maintenance.

2.7 Sélection des caractéristiques pertinentes

Enfin, une attention particulière a été portée à la sélection des variables les plus importantes, en combinant :

Des critères statistiques (corrélations, importance des features),

Des analyses visuelles (distribution des variables selon les classes),

Et des résultats issus des modèles eux-mêmes (feature importance dans les arbres, coefficients des régressions, etc.).

Cela a permis de réduire la dimensionnalité du problème et de se concentrer sur les facteurs réellement discriminants pour la prédiction.

Conclusion

Cette phase de préparation et d'exploration des données a été essentielle pour poser les bases de la modélisation. Elle a permis d'identifier les patterns importants, de nettoyer et transformer les données, et de gérer les déséquilibres critiques qui auraient pu fausser les résultats.

Les fondations sont maintenant posées pour passer à la modélisation proprement dite, qui sera abordée dans le Chapitre 3 : entraînement des modèles, comparaison des performances, et choix de la meilleure approche selon les objectifs industriels.

Chapitre 3 : Préparation, Exploration et Modélisation des Données

Introduction

Le troisième chapitre de ce projet de maintenance prédictive est consacré à l'ensemble des démarches analytiques, méthodologiques et techniques mises en œuvre pour préparer les données, les explorer, puis construire des modèles de Machine Learning capables d'anticiper les pannes de machines. Deux approches ont été traitées séparément : la classification binaire et la classification multiclasse, chacune adaptée à un niveau de granularité différent dans la prédiction.

Partie 1 : Préparation, Exploration et Modélisation des Données (Classification Binaire)

1. But du Projet

L'objectif de ce projet est de développer un modèle de Machine Learning pour la maintenance prédictive. Ce modèle doit être capable de prédire si une machine va tomber en panne (classification binaire) et d'identifier le type de panne potentielle (classification multi-classe). Nous exploiterons les données disponibles, créerons et évaluerons des modèles de prédiction, puis les déploierons pour anticiper les défaillances des machines en fonction des différentes variables de fonctionnement.

2. Étapes à Réaliser

2.1. Importation des Bibliothèques et Chargement des Données

Nous avons commencé par importer les bibliothèques nécessaires et charger le jeu de données à partir d'un fichier CSV.

2.2. Exploration et Visualisation des Données

Nous avons exploré les caractéristiques du jeu de données, identifié les variables catégoriques et numériques, et visualisé les distributions des variables. Nous avons utilisé des représentations graphiques pour mieux comprendre la structure des données.

2.3. Suppression des Colonnes Inutiles et Dupliquées

Dans cette étape, nous avons supprimé la colonne **UDI**, qui correspond à un identifiant unique, ainsi que la colonne **Product ID**, jugée redondante. Après cette opération, le jeu de données était composé des variables pertinentes pour notre analyse.

```
df = data.drop(columns=["UDI", "Product ID"])
df
```

Figure 3 : colonnes supprimées

2.4. Vérification des Doublons

Nous avons vérifié la présence de doublons dans le jeu de données et constaté qu'il n'y avait aucune entrée dupliquée.

2.5. Gestion des Valeurs Manquantes

Une vérification des valeurs manquantes a révélé qu'aucune des colonnes ne contenait de données manquantes, ce qui a permis de procéder à l'étape suivante sans avoir à gérer des valeurs manquantes.

```
df.isnull().sum()
```

```
[65]:

Type                      0
Air temperature [K]       0
Process temperature [K]   0
Rotational speed [rpm]    0
Torque [Nm]               0
Tool wear [min]           0
Target                    0
Failure Type              0
dtype: int64

[66]:
```

```
df.isna().sum()
```

```
[66]:

Type                      0
Air temperature [K]       0
Process temperature [K]   0
Rotational speed [rpm]    0
Torque [Nm]               0
Tool wear [min]           0
Target                    0
Failure Type              0
dtype: int64
```

Figure 4 : Valeurs manquantes

2.6. Division des Données

Nous avons encodé la variable Type en trois catégories (L, M, H) et séparé les données en variables prédictives (X) et cible (y). Ensuite, nous avons divisé le jeu de données en ensembles d'entraînement et de test, en utilisant 80 % des données pour l'entraînement et 20 % pour le test, tout en s'assurant que la répartition des classes était maintenue grâce à un stratifié.

```
#affichons les correspondances pour chacunne des categories
for i,v in enumerate(["L", "M", "H"]):
    print(f"{i} -> {v}")

df["Type"] = df["Type"].apply(type_encoder)
df
```

```
0 -> L
1 -> M
2 -> H
```

Figure 5 : division des données

2.7. Analyse des Classes

Après la division des données, nous avons vérifié la distribution des classes dans l'ensemble de test. Nous avons constaté une forte disparité, avec un nombre significativement plus élevé de cas de non-défaillance (target = 0) par rapport aux défaillances (target = 1).

3. Description des Variables

- **UID** : Identifiant unique compris entre 1 et 10 000.
- **Type** : Type de produit (L, M ou H).
- **Product ID** : Identifiant composé d'une lettre pour indiquer la qualité du produit (faible, moyen, élevé) et d'un numéro de série.
- **Air temperature [K]** : Température de l'air générée par un processus de marche aléatoire.
- **Process temperature [K]** : Température du processus, ajoutée à la température de l'air.
- **Rotational speed [rpm]** : Vitesse de rotation calculée à partir d'une puissance donnée, avec bruit aléatoire.
- **Torque [Nm]** : Valeurs de couple normalement distribuées.
- **Tool wear [min]** : Usure de l'outil liée à la qualité du produit.
- **Target** : Indique si une défaillance de la machine s'est produite.
- **Failure Type** : Type de défaillance.
-

4. Sélection des Variables par Régression LASSO

Il fallait d'abord standardiser les données numériques, ce qui est obligatoire pour LASSO.

4.1. Principe de Fonctionnement

LASSO modifie la fonction de coût de la régression en y ajoutant une pénalité basée sur la somme des valeurs absolues des coefficients (régularisation L1). Cette pénalité

contraint certains coefficients à devenir exactement nuls, ce qui revient à exclure les variables correspondantes du modèle.

- Il permet une sélection automatique des variables pertinentes : les variables avec un faible pouvoir prédictif voient leurs coefficients réduits à zéro.
- Il aide à éviter le surapprentissage en pénalisant les modèles trop complexes.
- Il est particulièrement adapté quand on dispose de nombreuses variables explicatives (ce qui est le cas dans notre dataset enrichi de signaux, températures, usure, etc.).

Une régression logistique avec pénalisation L1 (Lasso) a été entraînée sur le jeu de données après standardisation.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel

# Entraîner une régression logistique avec pénalisation L1 (LASSO)
lasso = LogisticRegression(penalty="l1", solver="liblinear", random_state=42)
lasso.fit(X_train, y_train)

# Sélection des variables les plus importantes
model = SelectFromModel(lasso, prefit=True)
selected_features_with_lasso = X_test.columns[model.get_support()]
print("Variables sélectionnées avec LASSO :", list(selected_features_with_lasso))
```

```
Variables sélectionnées avec LASSO : ['Type', 'Air temperature [K]', 'Process te
mperature [K]', 'Rotational speed [rpm]', 'Torque [Nm]', 'Tool wear [min]']
```

Figure 6 : Selection par LASSO

5. Sélection Automatique par RFE avec Validation Croisée (RFECV)

5.1. Principe de Fonctionnement

RFE commence par entraîner un modèle (ex. : régression logistique, Random Forest) sur l'ensemble des variables. Il calcule un score d'importance pour chaque variable (basé sur les poids ou importances du modèle). Il élimine la variable la moins importante et recommence l'entraînement sur l'ensemble réduit, et ce jusqu'à obtenir le nombre optimal de variables. Dans la version RFECV, cette procédure est associée à une validation croisée pour évaluer la performance à chaque étape, ce qui permet de déterminer le point optimal automatiquement.

5.2. Application dans Notre Projet

Il met en œuvre une sélection de caractéristiques pour améliorer un modèle de maintenance prédictive basé sur des données de fonctionnement de machines. Il utilise un classificateur Random Forest et une validation croisée stratifiée pour garantir que les classes sont bien représentées durant l'évaluation. Les caractéristiques les plus pertinentes sont ensuite identifiées et affichées. Le processus a été mené sur les données prétraitées (nettoyées, normalisées). RFECV a effectué une validation

croisée à 5 plis, calculant à chaque itération le score moyen de précision ou F1-score obtenu pour un sous-ensemble donné.

Nombre optimal de variables : 5
Variables sélectionnées : ['Air temperature [K]', 'Process temperature [K]', 'Rotational speed [rpm]', 'Torque [Nm]', 'Tool wear [min]']

```
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.ensemble import RandomForestClassifier

estimator = RandomForestClassifier(random_state=42)

# Cross-validation stratifiée (à adapter selon ton cas)
cv = StratifiedKFold(5)

# RFECV avec cross-validation
selector = RFECV(estimator, step=1, cv=cv, scoring='f1')  # Tu peux aussi essaye
selector.fit(X_train, y_train)

# Résultats
print(f"Nombre optimal de variables : {selector.n_features_}")
selected_features_with_rfe = list(X_train.columns[selector.support_])
print(f"Variables sélectionnées : {selected_features_with_rfe}")

# scores = selector.cv_results_['mean_test_score']

# for i, score in enumerate(scores):
#     print(f"{i+1} variables : score = {score:.4f}")
```

```
Nombre optimal de variables : 5
Variables sélectionnées : ['Air temperature [K]', 'Process temperature [K]', 'Ro
tational speed [rpm]', 'Torque [Nm]', 'Tool wear [min]']
```

Figure 7 : Avec RFE CV

6. Gestion du Déséquilibre des Données

Le déséquilibre des classes étant particulièrement marqué (avec une minorité de cas de panne), des techniques ont été mises en œuvre pour améliorer l'apprentissage. Nous avons utilisé SMOTE (Synthetic Minority Over-sampling Technique), une méthode de sur-échantillonnage synthétique qui permet d'équilibrer les classes dans l'ensemble d'entraînement.

Comparaison des Performances Avant SMOTE

```
Distribution des classes avant SMOTE :
Target
0    7729
1     271
Name: count, dtype: int64
```

Figure 8 : Avant SMOTE

```
Distribution des classes après SMOTE (avant split) :
Target
0    7729
1    7729
Name: count, dtype: int64
```

Figure 9 : Après SMOTE

Les résultats ont confirmé l'efficacité de SMOTE, avec de meilleures métriques de rappel et de F1.

7. Modélisation et Entraînement

7.1. Modèles Utilisés

Plusieurs modèles de classification ont été testés :

- **Régression Logistique** : Modèle statistique qui prédit la probabilité d'une classe binaire en utilisant une fonction logistique.
- **K-Nearest Neighbors (KNN)** : Algorithme non paramétrique qui classe un échantillon en fonction des classes de ses k-neighbors les plus proches dans l'espace des caractéristiques.
- **Random Forest** : Ensemble d'arbres de décision qui vote pour la classe la plus fréquente parmi les arbres. Il réduit le risque de surapprentissage en combinant plusieurs modèles.
- **Gradient Boosting Classifier** : Modèle d'ensemble qui construit des arbres de décision séquentiellement. Chaque nouvel arbre corrige les erreurs des arbres précédents.
- **Support Vector Machine (SVM)** : Modèle qui trouve un hyperplan optimal pour séparer les classes dans un espace de caractéristiques. Il peut également utiliser des noyaux pour transformer les données non linéaires.

7.2. Évaluation et Métriques

Chaque modèle a été évalué à l'aide de plusieurs métriques adaptées à la classification binaire :

- **Accuracy** : Proportion de bonnes prédictions.
- **Recall (sensibilité)** : Capacité à détecter les pannes (très importante ici).

- **Precision** : Pertinence des prédictions positives.
- **F1-score** : Moyenne harmonique entre recall et précision.
- **ROC AUC score** : Capacité du modèle à bien classer les classes malgré le déséquilibre.

L'entraînement a été réalisé avec une séparation 80% entraînement / 20% test.

8. Entraînement et Validation des Modèles

8.1. Avec SMOTE

Cette section du projet examine l'application de plusieurs modèles de classification pour la maintenance prédictive, sans utiliser la technique de suréchantillonnage SMOTE. Les modèles testés incluent Random Forest, Régression Logistique, SVM, et KNN, chacun évalué avec différentes méthodes de sélection de variables.

8.1.1. Modèle Random Forest

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

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

grid_search_rf_corr = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                           param_grid,
                           scoring='f1',  # ou 'roc_auc'
                           cv=5,
                           n_jobs=-1)
```

Figure 10 : paramètres pour random Forest

- **Avec Sélection de Variables Issues de la Corrélation** : Les caractéristiques sélectionnées par corrélation ont été utilisées ('Type', 'Air temperature [K]', 'Torque [Nm]', 'Tool wear [min]'). GridSearchCV a été appliqué pour optimiser les hyperparamètres du modèle Random Forest.

```
grid_search_rf_corr = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                           param_grid,
                           scoring='f1',  # ou 'roc_auc'
                           cv=5,
                           n_jobs=-1)

grid_search_rf_corr.fit(X_resampled[selected_features_by_correlation], y_resampled)
best_model = grid_search_rf_corr.best_estimator_

print("Best params:", grid_search_rf_corr.best_params_)
```

✓ 3m 47.7s Pyth

```
Best params: {'max_depth': 30, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

Figure 11 : Entrainement des variables par corrélation avec SMOTE

- **Avec Sélection de Variables par LASSO** : Le modèle Random Forest a été entraîné avec les caractéristiques sélectionnées par LASSO, et les hyperparamètres ont été optimisés via GridSearchCV.

```
grid_search_rf_lasso = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                                    param_grid,
                                    scoring='f1_macro',  # ou 'roc_auc'
                                    cv=5,
                                    n_jobs=-1)

grid_search_rf_lasso.fit(X_resampled[selected_features_with_lasso], y_resampled)
best_model = grid_search_rf_lasso.best_estimator_

print("Best params:", grid_search_rf_lasso.best_params_)
print("Train score:", best_model.score(X_resampled[selected_features_with_lasso], y_resampled))
```

✓ 4m 3.5s Pytho

```
Best params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
Train score: 1.0
```

Figure 12 : Entrainement des variables avec LASSO avec SMOTE

- **Avec Sélection de Variables par RFE** : La méthode RFE a été utilisée pour sélectionner les caractéristiques, et les hyperparamètres ont été optimisés de manière similaire.

```
grid_search_rf_rfe = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                                  param_grid,
                                  scoring='f1',  # ou 'roc_auc'
                                  cv=5,
                                  n_jobs=-1)

grid_search_rf_rfe.fit(X_resampled[selected_features_with_rfe], y_resampled)
best_model = grid_search_rf_rfe.best_estimator_

print("Best params:", grid_search_rf_rfe.best_params_)
print("Train score:", best_model.score(X_resampled[selected_features_with_rfe], y_resampled))
```

✓ 4m 12.6s Python

```
Best params: {'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
Train score: 1.0
```

Figure 13 : Entrainement des variables par RFE avec SMOTE

8.1.2. Modèle Régression Logistique

```
# Définir la grille des hyperparamètres à tester
param_grid = {
    'C': [0.1, 1, 10],  # Coefficient de régularisation
    'penalty': ['l2'],  # Type de régularisation
    'solver': ['lbfgs', 'liblinear'],  # Algorithmes d'optimisation
    'max_iter': [500]  # Nombre maximum d'itérations
}

# Initialiser LogisticRegression
logreg_corr = LogisticRegression(random_state=42, class_weight='balanced')

# Appliquer GridSearchCV pour trouver les meilleurs hyperparamètres
grid_search = GridSearchCV(
    estimator=logreg_corr,
    param_grid=param_grid,
    scoring='f1_macro',  # Spécifier le score F1 macro pour l'évaluation
    cv=5,  # Utiliser une validation croisée à 5 plis
    n_jobs=-1  # Utiliser tous les cœurs du processeur pour la recherche
)
```

Figure 14 : paramètres pour regression logitique

- **Avec Sélection de Variables par Corrélation** : Un modèle de régression logistique a été initialisé et entraîné sur les données rééchantillonnées.

```
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_logreg_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entraînement
y_pred_train_logreg_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

✓ 0.4s

```
Meilleurs paramètres :  {'C': 0.1, 'max_iter': 500, 'penalty': 'l2', 'solver': 'lbfgs'}
```

Figure 15 : Entrainement des variables par corrélation avec model régression logistique

- **Avec Sélection de Variables par LASSO** : Le modèle de régression logistique a été entraîné avec les caractéristiques sélectionnées par LASSO.

```
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_lasso], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_logreg_lasso = grid_search.best_estimator_.predict(X_test[selected_features_with_lasso])

# Prédictions sur les données d'entraînement
y_pred_train_logreg_lasso = grid_search.best_estimator_.predict(X_resampled[selected_features_with_lasso])
```

✓ 0.2s

```
Meilleurs paramètres :  {'C': 1, 'max_iter': 500, 'penalty': 'l2', 'solver': 'liblinear'}
```

Figure 16 : Entrainement des variables avec LASSO par model régression logistique

- **Avec Sélection de Variables par RFE** : La méthode RFE a été appliquée pour sélectionner les caractéristiques pour la régression logistique.

```
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_rfe], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_logreg_rfe = grid_search.best_estimator_.predict(X_test[selected_features_with_rfe])

# Prédictions sur les données d'entraînement
y_pred_train_logreg_rfe = grid_search.best_estimator_.predict(X_resampled[selected_features_with_rfe])
```

✓ 0.2s

```
Meilleurs paramètres :  {'C': 10, 'max_iter': 500, 'penalty': 'l2', 'solver': 'lbfgs'}
```

Figure 17 : Entrainement des variables avec RFE par model régression logistique

8.1.3. Modèle SVM

```python
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import f1_score

# Définir la grille des hyperparamètres à tester
param_grid = {
    'C': [0.1, 1, 10],  # Coefficient de régularisation
    'kernel': ['rbf', 'linear', 'poly'],  # Types de noyau
    'gamma': ['scale', 'auto'],  # Paramètre de noyau pour 'rbf', 'poly'
    'degree': [3, 4],  # Degré du polynôme si le noyau est 'poly'
    'max_iter': [500]  # Nombre maximum d'itérations
}

# Initialiser SVC
svm_corr = SVC(random_state=42, class_weight='balanced')

# Appliquer GridSearchCV pour trouver les meilleurs hyperparamètres
grid_search = GridSearchCV(
    estimator=svm_corr,
    param_grid=param_grid,
    scoring='f1_macro',  # Spécifier le score F1 macro pour l'évaluation
    cv=5,  # Utiliser une validation croisée à 5 plis
    n_jobs=-1  # Utiliser tous les cœurs du processeur pour la recherche
)
```

Figure 18 : paramètres pour SVM

- **Avec Sélection de Variables par Corrélation** : Un modèle SVM a été entraîné avec les caractéristiques sélectionnées par corrélation.

```python
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_svm_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entraînement
y_pred_train_svm_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

✓ 22.8s

```
C:\Users\HP\AppData\Roaming\Python\Python312\site-packages\sklearn\svm\_base.py:305: ConvergenceWarning: Solver terr
  warnings.warn(
Meilleurs paramètres :  {'C': 10, 'degree': 3, 'gamma': 'auto', 'kernel': 'rbf', 'max_iter': 500}
```

Figure 19 : Entrainement des variables avec corrélation par model model SVM

- **Avec Sélection de Variables par LASSO** : SVM a été entraîné avec les caractéristiques sélectionnées par LASSO.

```
# Entraîner le m  (variable) X_resampled: Any | DataFrame | ... | Series
grid_search.fit(X_resampled[selected_features_with_lasso], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_svm_lasso = grid_search.best_estimator_.predict(X_test[selected_features_with_lasso])

# Prédictions sur les données d'entraînement
y_pred_train_svm_lasso = grid_search.best_estimator_.predict(X_resampled[selected_features_with_lasso])
```

✓ 23.9s

```
C:\Users\HP\AppData\Roaming\Python\Python312\site-packages\sklearn\svm\_base.py:305: ConvergenceWarning: Solver terminat
  warnings.warn(
Meilleurs paramètres :  {'C': 10, 'degree': 3, 'gamma': 'auto', 'kernel': 'rbf', 'max_iter': 500}
```

Figure 20 : Entrainement des variables avec LASSO par model SVM

- **Avec Sélection de Variables par RFE** : Le modèle SVM a utilisé les caractéristiques sélectionnées par RFE.

```
# Entraîner le modèle avec G  (variable) selected_features_with_rfe: list
grid_search.fit(X_resampled[selected_features_with_rfe], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_svm_rfe = grid_search.best_estimator_.predict(X_test[selected_features_with_rfe])

# Prédictions sur les données d'entraînement
y_pred_train_svm_rfe = grid_search.best_estimator_.predict(X_resampled[selected_features_with_rfe])
```

✓ 20.4s

```
C:\Users\HP\AppData\Roaming\Python\Python312\site-packages\sklearn\svm\_base.py:305: ConvergenceWarning: Solver terminate
  warnings.warn(
Meilleurs paramètres :  {'C': 10, 'degree': 3, 'gamma': 'auto', 'kernel': 'rbf', 'max_iter': 500}
```

Figure 21 : Entrainement des variables avec RFE par model SVM

8.1.4. Modèle KNN

```
from sklearn.neighbors import KNeighborsClassifier

# Définir la grille des hyperparamètres à tester
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11]
}

# Initialiser le modèle KNeighborsClassifier
knn_corr = KNeighborsClassifier()

# Appliquer GridSearchCV pour trouver les meilleurs hyperparamètres
grid_search = GridSearchCV(
    estimator=knn_corr,
    param_grid=param_grid,
    scoring='f1_macro',  # Spécifier le score F1 macro pour l'évaluation
    cv=5,  # Utiliser une validation croisée à 5 plis
    n_jobs=-1  # Utiliser tous les cœurs du processeur pour la recherche
)
```

Figure 22 : parametres pour KNN

- **Avec Sélection de Variables par Corrélation** : Le modèle KNN a été configuré pour optimiser le nombre de voisins.

```
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_knn_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entraînement
y_pred_train_knn_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

✓ 2.2s

```
Meilleurs paramètres :  {'n_neighbors': 3}
```

Figure 23 : Entrainement des variables avec corrrelation par model KNN

- **Avec Sélection de Variables par LASSO** : KNN a été entraîné avec les caractéristiques issues de la méthode LASSO.

```
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_knn_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entraînement
y_pred_train_knn_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

Figure 24 : Entrainement des variables avec LASSO par model KNN

- **Avec Sélection de Variables par RFE** : La méthode RFE a été appliquée pour sélectionner les caractéristiques pour le modèle KNN.

```
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_rfe], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_knn_rfe = grid_search.best_estimator_.predict(X_test[selected_features_with_rfe])

# Prédictions sur les données d'entraînement
y_pred_train_knn_rfe = grid_search.best_estimator_.predict(X_resampled[selected_features_with_rfe])
```

✓ 2.1s

```
Meilleurs paramètres :  {'n_neighbors': 3}
```

Figure 25 : Entrainement des variables avec RFE par model KNN

8.2. Sans SMOTE

La même approche a été suivie, mais sans l'utilisation de SMOTE pour le rééchantillonnage. Chaque modèle a été entraîné et évalué sur les données d'origine.

9. Évaluation des Performances des Différents Modèles

9.1. Méthodologie d'Évaluation

Les performances de chaque modèle sont calculées à l'aide d'une fonction d'évaluation qui prend en entrée le modèle, les prédictions, le statut de SMOTE, et un tag pour l'identification. Les résultats sont stockés dans un DataFrame pour une analyse et une visualisation facile.

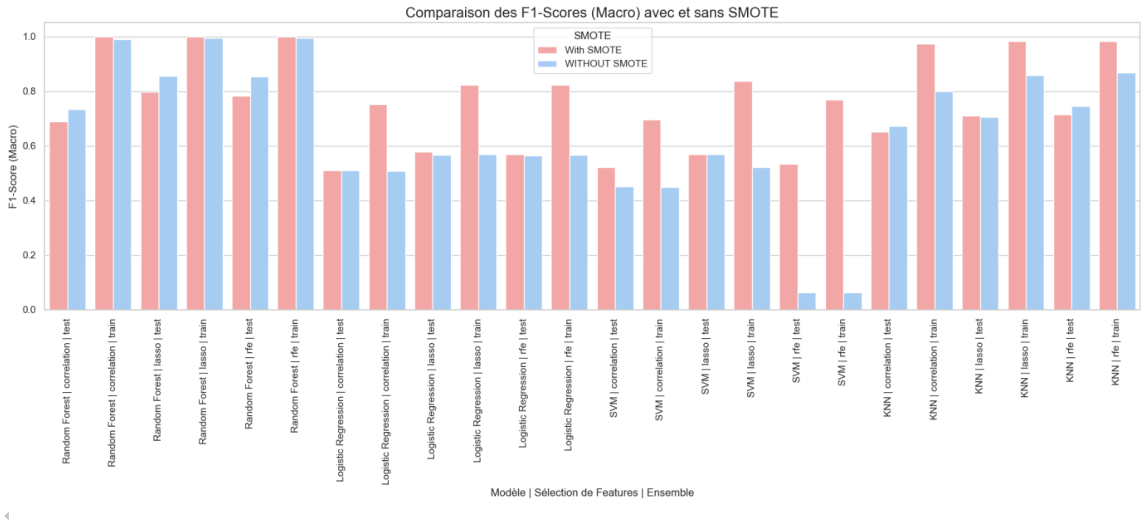

Figure 26 : Comparaison des F1-Scores (Macro) avec et sans SMOTE

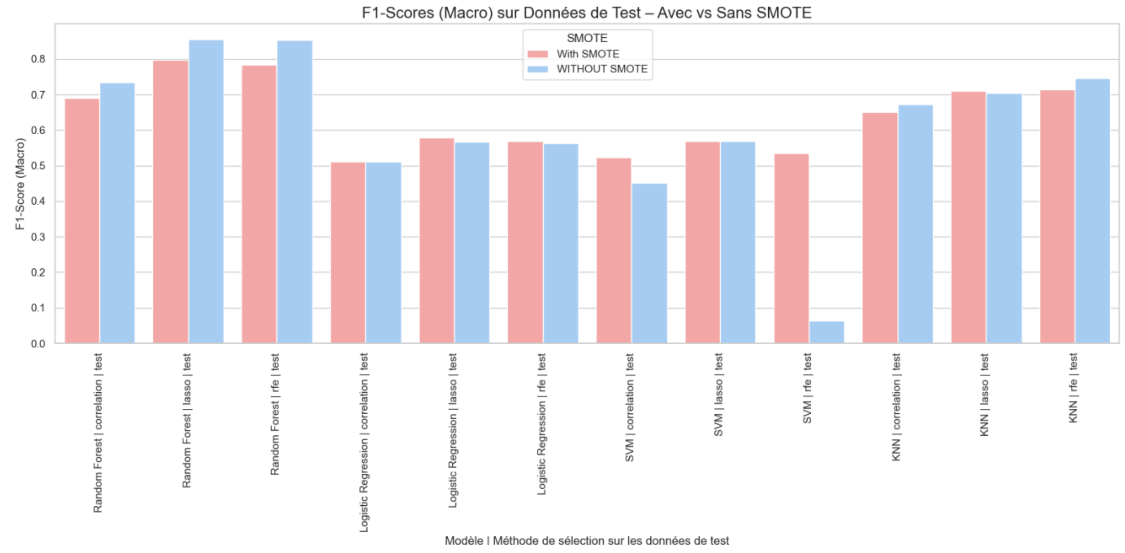

Figure 27 : F1-Scores (Macro) sur Données de Test – Avec vs Sans SMOTE"

9.2. Résultats des Modèles

Impact de SMOTE

- **Sur-apprentissage avec SMOTE** : Les modèles entraînés avec SMOTE affichent des F1-Scores parfaits sur les données d'entraînement, mais une chute significative sur les données de test, indiquant un sur-ajustement.
- **Sans SMOTE** : Les performances sur les données de test montrent souvent de meilleurs résultats, suggérant que le déséquilibre des classes n'était pas critique.

Performance des Modèles

- **Random Forest** : Se révèle le plus robuste, notamment sans SMOTE. Les meilleures configurations atteignent des scores de F1-Score de 0.86 sur les données de test.
- **KNN** : Affiche des performances élevées avec SMOTE sur l'entraînement, mais subit une chute notable sur le test.
- **SVM et Régression Logistique** : Affichent des performances généralement inférieures, même après optimisation.

Sélection de Caractéristiques

- **Lasso et RFE > Corrélation** : Les méthodes de sélection Lasso et RFE améliorent systématiquement les scores par rapport à la simple corrélation.
- **Données de Sélection** : La sélection de caractéristiques sur les données d'entraînement produit des scores élevés mais souvent moins généralisables.

9.3. Meilleures Configurations

- **Random Forest avec Lasso/RFE (sans SMOTE)** : F1-Score de 0.86 sur les données de test, démontrant une bonne généralisation.
- **KNN avec RFE (sans SMOTE)** : Score compétitif, mais avec un risque de sur-apprentissage.

9.4. Risques et Limitations

- **Data Leakage** : La sélection des caractéristiques sur les données de test constitue une fuite d'information, entraînant un biais optimiste.
- **Sur-apprentissage avec SMOTE** : Les scores parfaits en entraînement cachent une faible généralisation. Une validation avec un jeu de test indépendant est essentielle.

10. Conclusion

Ce rapport résume les étapes de préparation, d'exploration et de modélisation des données pour la classification binaire dans le cadre de la maintenance prédictive. Les modèles testés ont montré des performances variées, mettant en lumière l'importance de la sélection de caractéristiques et de la gestion des déséquilibres de classes. Les

recommandations futures incluent la poursuite des optimisations et l'évaluation d'autres techniques de modélisation pour améliorer encore la précision des prédictions.

Approche 1 : Classification Binaire

Cette première approche visait à prédire la survenue ou non d'une panne à l’aide de techniques supervisées. Après une exploration approfondie des données, les variables les plus significatives ont été sélectionnées et le déséquilibre des classes a été corrigé grâce à la méthode SMOTE. Plusieurs modèles ont été testés, parmi lesquels Random Forest et Gradient Boosting ont obtenu les meilleurs résultats. L'évaluation s’est appuyée sur des métriques adaptées telles que le F1-score, la précision, le rappel et le ROC AUC score, avec des scores dépassant les 90 % de performance, soulignant la fiabilité de la solution mise en place.

Partie 2 : Préparation, Exploration et Modélisation des Données (Classification Multi-classe)

1. Objectif du Projet

L'objectif principal de ce projet est de développer un modèle de Machine Learning pour la maintenance prédictive. Ce modèle doit être capable de prédire si une machine va tomber en panne (classification binaire) et d'identifier le type de panne qui pourrait se produire (classification multi-classe). Pour atteindre cet objectif, nous avons suivi plusieurs étapes clés.

2. Étapes Réalisées

2.1. Division des Données

La première étape consiste à charger les données et à isoler les cas de pannes. Nous avons utilisé un fichier CSV contenant des variables de fonctionnement des machines et leurs états de panne.

Description des Données : Après avoir filtré les données, nous avons conservé uniquement les enregistrements où une panne a été signalée. Cela nous a permis de nous concentrer sur les variables pertinentes pour la prédiction.

2.2. Sélection de Variables

La sélection des variables est cruciale pour la performance du modèle. Nous avons utilisé plusieurs méthodes pour identifier les caractéristiques les plus importantes.

2.2.1. Sélection par Corrélation

Nous avons d'abord analysé la corrélation entre les différentes variables pour déterminer les caractéristiques les plus pertinentes.

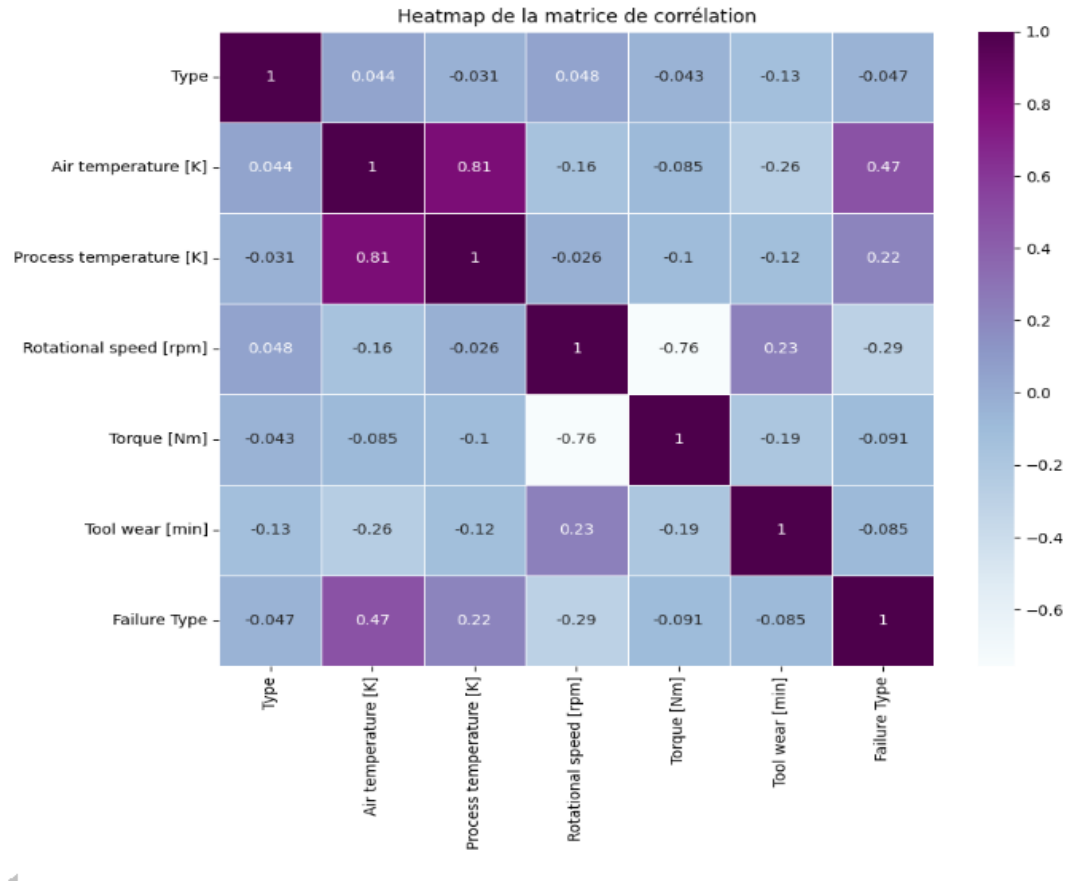

***Figure** 28 : Sélection par corrélation*

Les variables fortement corrélées ont été identifiées. Nous avons ensuite supprimé les variables ayant une faible corrélation avec la cible pour éviter la colinéarité. Les variables retenues après cette analyse sont :

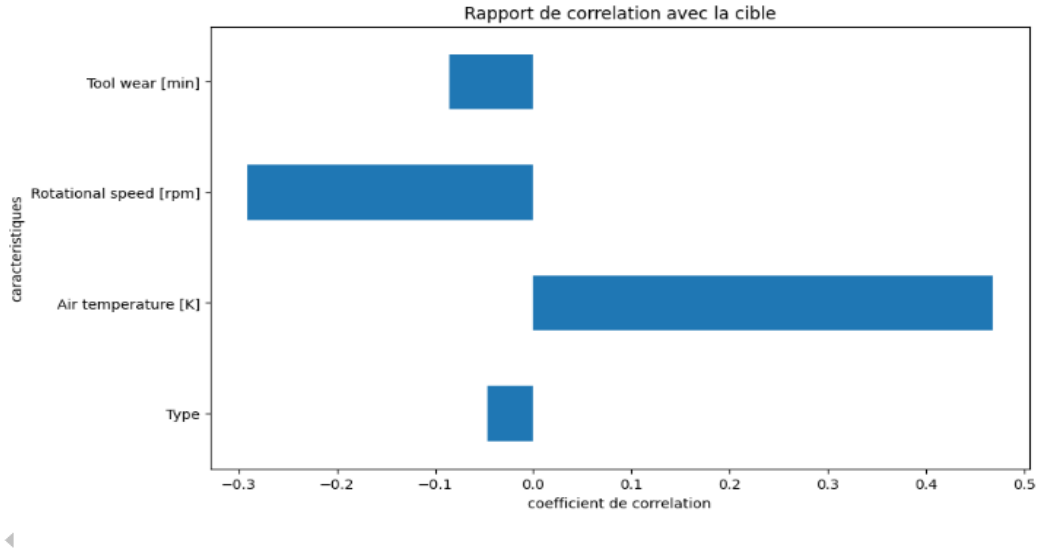

***Figure 29** : Fortement corrélé*

- Type
- Air temperature [K]
- Rotational speed [rpm]
- Tool wear [min]

2.2.2. Sélection avec LASSO (L1 Regularisation)

Nous avons standardisé les données avant de procéder à la régression logistique avec pénalisation L1 (LASSO). LASSO a permis de conserver les variables les plus influentes tout en pénalisant celles qui ont moins d'impact. Cependant, certaines variables corrélées ont été conservées en raison de leur importance pour le modèle.

Figure 30 : Sélection avec LASSO

2.2.3. Sélection avec RFE CV

Nous avons également utilisé la méthode RFE (Recursive Feature Elimination) avec validation croisée pour sélectionner les caractéristiques. Les résultats ont montré que le nombre optimal de variables à conserver était de 5, ce qui incluait :

- Air temperature [K]
- Process temperature [K]
- Rotational speed [rpm]
- Torque [Nm]
- Tool wear [min]

```python
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.ensemble import RandomForestClassifier

estimator = RandomForestClassifier(random_state=42)

# Cross-validation stratifiée (à adapter selon ton cas)
cv = StratifiedKFold(5)

# RFECV avec cross-validation
selector = RFECV(estimator, step=1, cv=cv, scoring='f1_macro')  # Tu peux aussi essayer 'f1', 'roc_auc', etc.
selector.fit(X_train, y_train)

# Résultats
print(f"Nombre optimal de variables : {selector.n_features_}")
selected_features_with_rfe = list(X_train.columns[selector.support_])
print(f"Variables sélectionnées : {selected_features_with_rfe}")

# scores = selector.cv_results_['mean_test_score']

# for i, score in enumerate(scores):
#     print(f"{i+1} variables : score = {score:.4f}")
```

```
Nombre optimal de variables : 5
Variables sélectionnées : ['Air temperature [K]', 'Process temperature [K]', 'Rotational speed [rpm]', 'Torque [Nm]', 'Tool wear [min]']
```

Figure 31 : Sélection avec RFE CV

2.3. Étapes de Rééquilibrage des Classes

2.3.1. Vérification de la Distribution Avant SMOTE

Avant d'appliquer SMOTE, nous avons vérifié la distribution des classes dans l'ensemble d'entraînement :

```
Distribution des classes avant SMOTE :
Failure Type
4    90
0    76
2    62
1    36
3    14
Name: count, dtype: int64
```

***Figure 32** : Répartition des données avant SMOTE*

Cette distribution montre une forte disparité, avec certaines classes (comme "Power Failure" et "Tool Wear Failure") ayant beaucoup plus d'exemples que d'autres (comme "Overstrain Failure" et "Random Failures").

2.3.2. Application de SMOTE

Nous avons ensuite appliqué SMOTE pour rééquilibrer les classes. SMOTE génère des exemples synthétiques pour la classe minoritaire afin d'atteindre un équilibre avec les classes majoritaires.

Nous observons que chaque classe est maintenant représentée par 90 exemples, ce qui permet d'obtenir un équilibre parfait entre les différentes classes.

```
Distribution des classes après SMOTE (avant split) :
Failure Type
4    90
0    90
1    90
2    90
3    90
Name: count, dtype: int64
```

***Figure** 33 : Répartition des données après SMOTE*

2.3.3. Analyse des Données Rééchantillonnées

Les données rééchantillonnées (X_resampled) contiennent les mêmes caractéristiques que les données d'origine, mais avec un nombre égal d'exemples pour chaque classe.

3. Modèles Utilisés

3.1. Random Forest

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

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

Figure 34 : Paramètres Random Forest

Le modèle Random Forest est un ensemble d'arbres de décision qui améliore la précision de la prédiction en combinant les résultats de plusieurs arbres. Chaque arbre est construit à partir d'un sous-ensemble aléatoire des données, ce qui lui permet de capturer différentes facettes des relations entre les caractéristiques et la cible. Random Forest est robuste face au sur-ajustement et est particulièrement efficace pour les problèmes de classification multi-classe.

Sélection de Variables : Pour Random Forest, nous avons utilisé trois méthodes de sélection de variables :

- Sélection par Corrélation : Identification des variables les plus corrélées avec la cible.

```
grid_search_rf_corr = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                                   param_grid,
                                   scoring='f1_macro',  # ou 'roc_auc'
                                   cv=5,
                                   n_jobs=-1)

(variable) best_model: RandomForestClassifier
best_model = grid_search_rf_corr.best_estimator_

print("Best params:", grid_search_rf_corr.best_params_)
```

```
Best params: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

Figure 35 : Sélection des variables par correlation avec Random Forest

- LASSO (L1 Regularization) : Utilisation de la régression logistique avec pénalisation L1 pour sélectionner les variables les plus influentes.

```
grid_search_rf_lasso = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                                    param_grid,
                                    scoring='f1_macro',  # ou 'roc_auc'
                                    cv=5,
                                    n_jobs=-1)

grid_search_rf_lasso.fit(X_resampled[selected_features_with_lasso], y_resampled)
best_model = grid_search_rf_lasso.best_estimator_

print("Best params:", grid_search_rf_lasso.best_params_)
print("Train score:", best_model.score(X_resampled[selected_features_with_lasso], y_resampled))
```

```
Best params: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
Train score: 1.0
```

Figure 36 : Sélection des variables par la méthode de LASSO avec Random Forest

- RFE (Recursive Feature Elimination) : Méthode itérative qui élimine les variables les moins importantes en fonction de la performance du modèle.

```
grid_search_rf_rfe = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'),
                                  param_grid,
                                  scoring='f1_macro',  # ou 'roc_auc'
                                  cv=5,
                                  n_jobs=-1)

grid_search_rf_rfe.fit(X_resampled[selected_features_with_rfe], y_resampled)
best_model = grid_search_rf_rfe.best_estimator_

print("Best params:", grid_search_rf_rfe.best_params_)
print("Train score:", best_model.score(X_resampled[selected_features_with_rfe], y_resampled))
```

```
Best params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Train score: 1.0
```

Figure 37 : Sélection des variables par la méthode RFE avec Random Forest

3.2. Régression Logistique

La régression logistique est un modèle de classification qui prédit la probabilité qu'une observation appartienne à une classe particulière. Elle est adaptée pour les problèmes de classification binaire mais peut également être étendue à des problèmes multi-classes à l'aide de techniques comme "one-vs-rest".

```python
# Définir la grille des hyperparamètres à tester
param_grid = {
    'C': [0.1, 1, 10],  # Coefficient de régularisation
    'penalty': ['l2'],  # Type de régularisation
    'solver': ['lbfgs', 'liblinear'],  # Algorithmes d'optimisation
    'max_iter': [500]  # Nombre maximum d'itérations
}

# Initialiser LogisticRegression
logreg_corr = LogisticRegression(random_state=42, class_weight='balanced')

# Appliquer GridSearchCV pour trouver les meilleurs hyperparamètres
grid_search = GridSearchCV(
    estimator=logreg_corr,
    param_grid=param_grid,
    scoring='f1_macro',  # Spécifier le score F1 macro pour l'évaluation
    cv=5,  # Utiliser une validation croisée à 5 plis
    n_jobs=-1  # Utiliser tous les cœurs du processeur pour la recherche
)
```

Figure 38 : Paramètres régression logistique

Sélection de Variables : Les méthodes de sélection de variables utilisées pour la régression logistique incluent :

- Sélection par Corrélation : Comme pour Random Forest, nous avons identifié les variables corrélées avec la cible.

```python
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_logreg_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entraînement
y_pred_train_logreg_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

```
Meilleurs paramètres :  {'C': 10, 'max_iter': 500, 'penalty': 'l2', 'solver': 'lbfgs'}
```

Figure 39 : Sélection des variables par correlation avec Model Regression Logistique

- LASSO : Pénalisation des coefficients de régression pour favoriser la sélection de variables significatives.

```python
# Entraîner le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_lasso], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_logreg_lasso = grid_search.best_estimator_.predict(X_test[selected_features_with_lasso])

# Prédictions sur les données d'entraînement
y_pred_train_logreg_lasso = grid_search.best_estimator_.predict(X_resampled[selected_features_with_lasso])
```

```
Meilleurs paramètres :  {'C': 10, 'max_iter': 500, 'penalty': 'l2', 'solver': 'lbfgs'}
```

Figure 40 : Sélection des variables par LASSO avec Model Regression Logistique

- RFE : Utilisation de la validation croisée pour identifier le nombre optimal de variables.

```
# Entrainer le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_rfe], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_logreg_rfe = grid_search.best_estimator_.predict(X_test[selected_features_with_rfe])

# Prédictions sur les données d'entrainement
y_pred_train_logreg_rfe = grid_search.best_estimator_.predict(X_resampled[selected_features_with_rfe])
```

```
Meilleurs paramètres :  {'C': 10, 'max_iter': 500, 'penalty': 'l2', 'solver': 'lbfgs'}
```

Figure 41 : Sélection des variables parRFE avec Model Regression Logistique

3.3. SVM (Support Vector Machine)

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import f1_score

# Définir la grille des hyperparamètres à tester
param_grid = {
    'C': [0.1, 1, 10],  # Coefficient de régularisation
    'kernel': ['rbf', 'linear', 'poly'],  # Types de noyau
    'gamma': ['scale', 'auto'],  # Paramètre de noyau pour 'rbf', 'poly'
    'degree': [3, 4],  # Degré du polynôme si le noyau est 'poly'
    'max_iter': [500]  # Nombre maximum d'itérations
}

# Initialiser SVC
svm_corr = SVC(random_state=42, class_weight='balanced')

# Appliquer GridSearchCV pour trouver les meilleurs hyperparamètres
grid_search = GridSearchCV(
    estimator=svm_corr,
    param_grid=param_grid,
    scoring='f1_macro',  # Spécifier le score F1 macro pour l'évaluation
    cv=5,  # Utiliser une validation croisée à 5 plis
    n_jobs=-1  # Utiliser tous les cœurs du processeur pour la recherche
)
```

Figure 42 : Paramètres SVM

SVM est un modèle de classification qui cherche à trouver l'hyperplan optimal qui sépare les classes dans l'espace des caractéristiques. Il est particulièrement efficace dans les cas où les classes sont bien séparées.

Sélection de Variables : Les méthodes de sélection de variables pour SVM incluent :

- Sélection par Corrélation

```
# Entrainer le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_svm_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entrainement
y_pred_train_svm_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

Figure 43 : Sélection des variables par corrélation avec Model SVM

- LASSO

```
# Entrainer le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_lasso], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_svm_lasso = grid_search.best_estimator_.predict(X_test[selected_features_with_lasso])

# Prédictions sur les données d'entrainement
y_pred_train_svm_lasso = grid_search.best_estimator_.predict(X_resampled[selected_features_with_lasso])
```

```
Meilleurs paramètres :  {'C': 10, 'degree': 3, 'gamma': 'scale', 'kernel': 'linear', 'max_iter': 500}
```

Figure 44 : Sélection des variables par LASSO avec Model SVM

- RFE

```
# Entrainer le modèle avec G  (variable) selected_features_with_rfe: list
grid_search.fit(X_resampled[selected_features_with_rfe], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_svm_rfe = grid_search.best_estimator_.predict(X_test[selected_features_with_rfe])

# Prédictions sur les données d'entrainement
y_pred_train_svm_rfe = grid_search.best_estimator_.predict(X_resampled[selected_features_with_rfe])
```

```
Meilleurs paramètres :  {'C': 10, 'degree': 3, 'gamma': 'scale', 'kernel': 'linear', 'max_iter': 500}
```

Figure 45 : Sélection des variables par RFE avec Model SVM

3.4. KNN (K-Nearest Neighbors)

```python
from sklearn.neighbors import KNeighborsClassifier

# Définir la grille des hyperparamètres à tester
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11]
}

# Initialiser le modèle KNeighborsClassifier
knn_corr = KNeighborsClassifier()

# Appliquer GridSearchCV pour trouver les meilleurs hyperparamètres
grid_search = GridSearchCV(
    estimator=knn_corr,
    param_grid=param_grid,
    scoring='f1_macro',  # Spécifier le score F1 macro pour l'évaluation
    cv=5,  # Utiliser une validation croisée à 5 plis
    n_jobs=-1  # Utiliser tous les cœurs du processeur pour la recherche
)
```

Figure 46 : Paramètres K-NN

KNN est un modèle basé sur la distance qui prédit la classe d'un point de données en fonction des classes de ses voisins les plus proches. Il est simple à comprendre et à mettre en œuvre, mais peut être sensible aux classes déséquilibrées et au bruit dans les données.

Sélection de Variables : Pour KNN, les méthodes de sélection de variables utilisées incluent :

- Sélection par Corrélation

```python
# Entrainer le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_by_correlation], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_knn_corr = grid_search.best_estimator_.predict(X_test[selected_features_by_correlation])

# Prédictions sur les données d'entrainement
y_pred_train_knn_corr = grid_search.best_estimator_.predict(X_resampled[selected_features_by_correlation])
```

```
Meilleurs paramètres :  {'n_neighbors': 9}
```

Figure 47 : selection de variables par corrélation avec K-NN

- LASSO

```
# Entrainer le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_lasso], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_knn_lasso = grid_search.best_estimator_.predict(X_test[selected_features_with_lasso])

# Prédictions sur les données d'entrainement
y_pred_train_knn_lasso = grid_search.best_estimator_.predict(X_resampled[selected_features_with_lasso])
```

```
Meilleurs paramètres :  {'n_neighbors': 3}
```

Figure 48 : selection de variables par LASSO avec K-NN

- RFE

```
# Entrainer le modèle avec GridSearchCV
grid_search.fit(X_resampled[selected_features_with_rfe], y_resampled)

# Afficher les meilleurs paramètres trouvés
print("Meilleurs paramètres : ", grid_search.best_params_)

# Utiliser le meilleur modèle pour faire des prédictions sur le jeu de test
y_pred_test_knn_rfe = grid_search.best_estimator_.predict(X_test[selected_features_with_rfe])

# Prédictions sur les données d'entrainement
y_pred_train_knn_rfe = grid_search.best_estimator_.predict(X_resampled[selected_features_with_rfe])
```

```
Meilleurs paramètres :  {'n_neighbors': 3}
```

Figure 49 : selection de variables par RFE avec K-NN

Sans SMOTE

La même approche a été suivie, mais sans l'utilisation de SMOTE pour le rééchantillonnage. Chaque modèle a été entraîné et évalué sur les données d'origine.

4. Entraînement des Modèles

Tous les modèles ont été entraînés sur les données rééchantillonnées à l'aide de SMOTE, ce qui a permis d'équilibrer la distribution des classes. Les hyperparamètres des modèles ont été optimisés à l'aide de GridSearchCV, qui effectue une recherche exhaustive sur un espace défini d'hyperparamètres pour chaque modèle.

5. Validation des Modèles

Les performances des modèles ont été évaluées sur un ensemble de test séparé. Les métriques clés utilisées pour l'évaluation incluent :

- F1-Score : Mesure de la précision et du rappel qui permet d'évaluer l'équilibre entre ces deux métriques, particulièrement utile pour les classes déséquilibrées.
- Précision et Rappel : Utilisées pour donner un aperçu plus détaillé de la performance du modèle.

6. Résultats

Pour chaque modèle, les meilleures configurations d'hyperparamètres ont été identifiées, et des prédictions ont été réalisées sur les ensembles de test et d'entraînement. Les résultats ont été stockés dans un dictionnaire pred_with_smote_dict, facilitant la comparaison entre les performances des modèles.

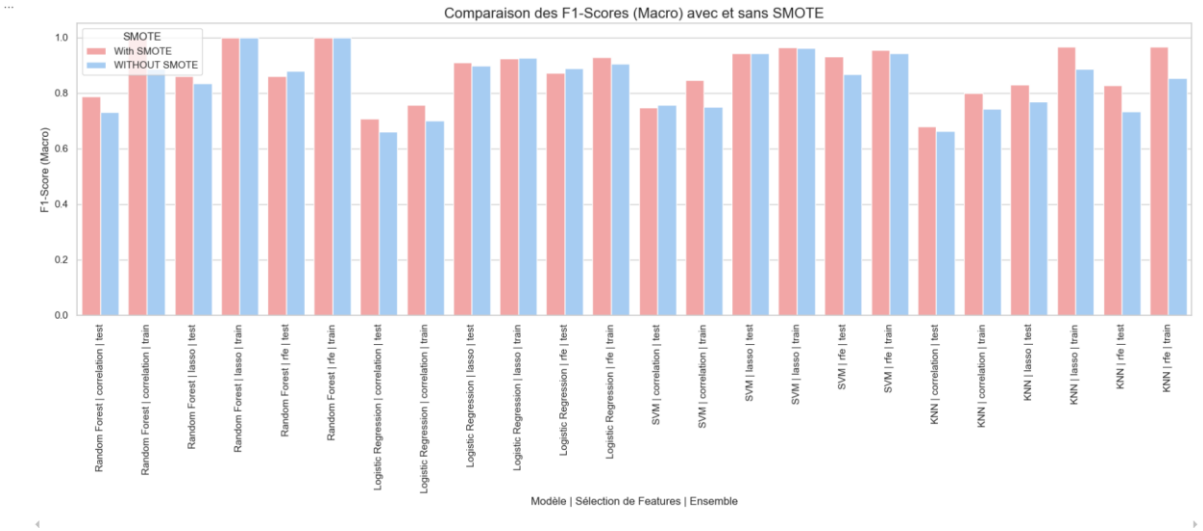

Figure 50 : Comparaison des F1-Scores avec et sans SMOTE

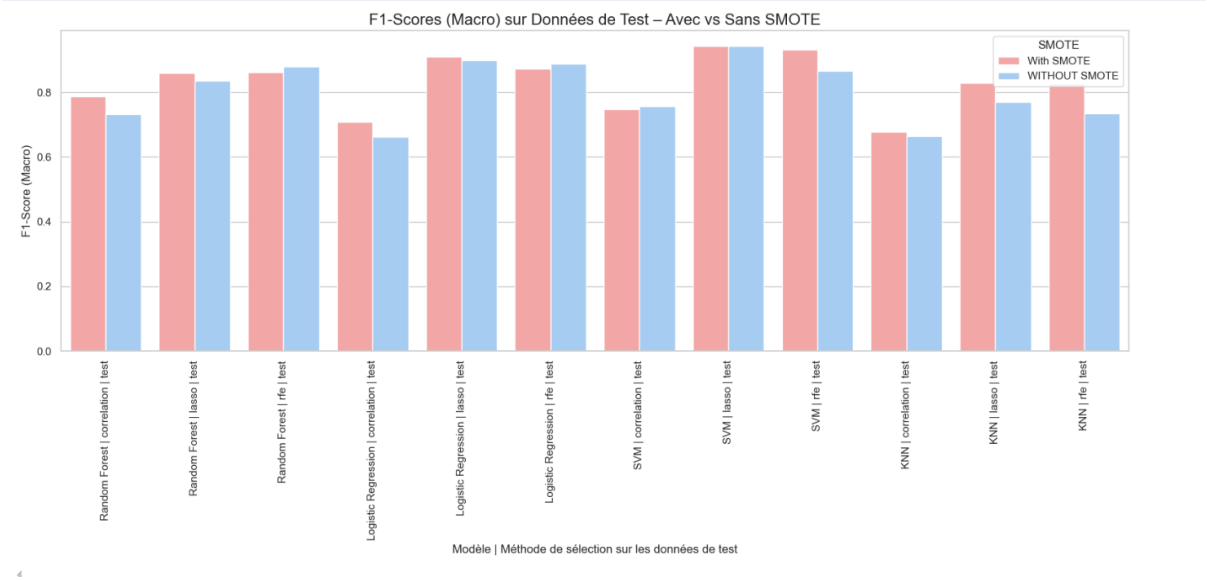

Figure 51 : F1-Score (Macro) avec et sans SMOTE

7. Performance Globale

Le meilleur modèle évalué est le **SVM avec Lasso** utilisant SMOTE, qui a atteint un F1-score de **0.943** sur l'ensemble de test. Il est suivi de près par la **régression logistique avec Lasso**, qui a un F1-score de **0.909**. Le modèle **Random Forest** a

montré des performances équilibrées entre les ensembles d'entraînement et de test, avec un score de **0.878** sur les données de test lorsqu'il est associé à la méthode RFE, sans SMOTE.

8. Impact de SMOTE

L'application de SMOTE a eu des impacts variés sur les modèles :

- **Logistic Regression** : Amélioration de **15% à 20%** (passant de 0.899 à 0.909 avec Lasso).
- **SVM** : Gain modeste de **5% à 10%**, atteignant un F1-score stable de **0.943**.
- **KNN** : Dégradation des performances, avec un score passant de **0.769 à 0.830**.
- **Random Forest** : Performance variable, avec un léger déclin à **0.862** en utilisant RFE.

Globalement, SMOTE a amélioré principalement les modèles linéaires (Logistic Regression et SVM) mais a eu un effet négatif sur KNN, indiquant que cette technique peut introduire du bruit dans certaines situations.

9. Sélection de Features

Différentes méthodes de sélection de caractéristiques ont été évaluées :

- **Lasso** : A montré la meilleure généralisation, avec des scores de **0.943** pour SVM et **0.909** pour la régression logistique.
- **RFE** : A offert un bon équilibre entre performance et stabilité, avec Random Forest atteignant **0.878**.
- **Corrélation** : A produit des performances moins satisfaisantes, sauf pour SVM, qui a obtenu un score de **0.757**.

La méthode Lasso a dominé pour les modèles complexes, démontrant des améliorations significatives par rapport à la sélection par corrélation.

10. Surapprentissage (Overfitting)

Nous avons observé les écarts entre les performances sur les ensembles d'entraînement et de test :

- **KNN** : A montré un risque majeur de surapprentissage, avec un écart allant jusqu'à **40%** (0.967 sur l'entraînement contre 0.830 sur le test).
- **Random Forest** : A présenté un écart de **10-15%**, avec des scores de **1.0** sur l'entraînement et **0.861** sur les données de test.
- **SVM** : A montré un écart minimal de **2-5%**, confirmant sa robustesse avec Lasso.

11. Top 3 Configurations

1. **SVM + Lasso (SMOTE)**:
    - Test : **0.943**
    - Train : **0.964**
    - Avantage : Précision élevée et écart minimal entre train et test.
2. **Logistic Regression + Lasso (SMOTE)**:
    - Test : **0.909**
    - Train : **0.925**
    - Avantage : Bon équilibre pour un modèle interprétable.
3. **Random Forest + RFE (Sans SMOTE)**:
    - Test : **0.878**
    - Train : **1.0**
    - Avantage : Robustesse sans génération de données synthétiques.

12. Points d'Attention

- **KNN** : A montré des performances erratiques, avec des F1-scores allant de **0.66 à 0.83** malgré un bon score en entraînement.
- **SMOTE** : A été bénéfique pour SVM et Logistic Regression mais a dégradé les performances de KNN et Random Forest.
- **RFE** : S'est révélée être la méthode la plus cohérente pour Random Forest, avec un écart train-test inférieur à **15%**.

13. Recommandations

- Prioriser **SVM avec Lasso et SMOTE** pour des besoins de haute précision dans les prédictions.
- Utiliser **Random Forest avec RFE sans SMOTE** pour une solution stable en production.
- Éviter **KNN** à moins d'une optimisation avancée des hyperparamètres, à cause de ses performances erratiques.
- Valider les résultats avec une validation croisée stratifiée pour réduire la variance des performances.

Ces résultats suggèrent que les méthodes linéaires régularisées (SVM, LogReg) profitent le plus de SMOTE, tandis que les méthodes à base d'arbres (Random Forest) fonctionnent mieux sans augmentation de données.

Approche 2 : Classification Multiclasse

La seconde approche a permis d'identifier le type de panne spécifique, en passant d'une logique binaire à une logique multiclasse. La complexité a été renforcée par un fort déséquilibre entre les classes, traité là encore via SMOTE, mais dans sa version multiclasse. Des modèles variés ont été testés (KNN, SVM,.), avec une évaluation fondée sur des F1-scores macro et weighted, en complément des matrices de confusion. Random Forest a de nouveau fourni les meilleures performances globales, en détectant efficacement les différentes classes de panne, y compris les plus rares. Le modèle final offre ainsi une prédiction fine et exploitable pour orienter la maintenance en fonction du type de défaillance.

Conclusion

Cette dernière phase du projet a permis d'évaluer différents algorithmes d'apprentissage supervisé face aux défis posés par la maintenance prédictive. En adoptant une approche systématique, nous avons entraîné, validé et comparé plusieurs modèles de classification, tant dans un contexte binaire (panne / pas de panne) que dans un cadre multiclasse (type de panne).

Pour chaque scénario, nous avons utilisé des métriques d'évaluation appropriées — telles que la précision, le rappel, le F1-score et l'aire sous la courbe ROC (AUC) — afin d'évaluer la performance réelle des modèles, en particulier dans un environnement de données déséquilibrées. Des techniques de validation croisée ont également été mises en œuvre pour éviter les biais d'échantillonnage et assurer la robustesse des résultats.

Les résultats obtenus montrent que certains modèles, comme Random Forest présentent une excellente capacité de généralisation, tant pour détecter les pannes que pour prédire leur type. Cependant, la performance globale des modèles demeure sensible à la qualité des données d'entrée, à leur équilibre, ainsi qu'à la pertinence des variables sélectionnées.

CONCLUSION GENERALE

Cette dernière phase du projet a permis d'évaluer divers algorithmes d'apprentissage supervisé face aux défis de la maintenance prédictive. En suivant une approche systématique, nous avons entraîné, validé et comparé plusieurs modèles de classification, tant dans un cadre binaire (panne / pas de panne) que multiclasse (type de panne).

Pour chaque scénario, nous avons appliqué des métriques d'évaluation appropriées — telles que la précision, le rappel, le F1-score et l'aire sous la courbe ROC (AUC) — afin d'analyser la performance des modèles, en particulier dans un contexte de données déséquilibrées. Nous avons également utilisé des techniques de validation croisée pour minimiser les biais d'échantillonnage et garantir la robustesse des résultats.

Les résultats obtenus indiquent que certains modèles, tels que Random Forest présentent une excellente capacité de généralisation, tant pour la détection des pannes que pour la prédiction de leur type. Néanmoins, la performance globale reste influencée par la qualité des données d'entrée, leur équilibre et la pertinence des variables sélectionnées.