

Projet python

Prédiction du RUL

Data set :
Nasa Turbofan engines

Eya Ghorbel
Nermine Rahmeni

Janvier, 2025

Table des Matières

Introduction

1-Exploration et Prétraitement des Données(EDA)

1. description de la data set
2. Chargement des données
3. Importation des données
4. Calcul du RUL (Remaining Useful Life)
5. Résumé statistique
6. Vérifications des valeurs manquantes
7. Tableau descriptif
8. Visualisation
9. Analyse exploratoire des capteur
10. Extraction des caractéristique:
11. Sélection des caractéristiques essentielle:
12. Standardisation

2- Modélisation et Entraînement des Modèles

1. Le testing
2. Régression linéaire
3. Régression polynomiale
4. Régression par Support Vector Machine (SVR)
5. Régression par arbre de décision (Decision Ttree Rregression)
6. Régression par forêt aléatoire (RFR: Random Forest Regression)
7. Réseau de neurones artificiels: (ANN: Artificial Neural Network)
8. KNN (k plus proches voisins)
9. Proposition d'amélioration
 - a- Hypothèse URL
 - b- Feature engineering (polynomial:
 - C- Amélioration des paramètres du modèle

3-Recommandation pour les travaux futurs

Conclusion



Introduction

La maintenance prédictive est un domaine essentiel dans l'industrie, notamment pour la gestion des équipements, car elle permet de prévoir leur état et d'éviter les pannes, réduisant ainsi les risques d'arrêts coûteux. En anticipant le moment où un équipement risque de défaillir, les entreprises peuvent mieux planifier les opérations de maintenance, minimiser les interruptions imprévues et améliorer à la fois la sécurité et la fiabilité des systèmes.

Dans ce cadre, le jeu de données fourni par la NASA, accessible sur Kaggle, représente une modélisation de la dégradation des actifs. Il comprend des données simulées de type *Run-to-Failure* (RtF) pour des moteurs à turbopropulsion d'avion, visant à prédire la durée de vie utile restante (RUL) des moteurs. Ces données proviennent du modèle C-MAPSS (Commercial Modular Aero-Propulsion System Simulation), qui simule la dégradation des moteurs d'avion dans divers scénarios opérationnels et modes de défaillance. Ce type d'analyse est particulièrement crucial dans des secteurs tels que l'aviation, où la santé des moteurs a un impact direct sur la sécurité et l'efficacité des opérations.

Au cours de ce rapport, nous détaillerons la collecte, l'analyse et la préparation des données, suivies de la mise en œuvre et de l'évaluation de modèles prédictifs pour estimer la durée de vie utile restante (RUL) des moteurs. Nous concluons par une analyse des résultats et des perspectives d'amélioration.

I- Exploration et Prétraitement des Données(EDA)

1. Description de la data set :

Le jeu de données turbofan **C**MAPSS (**C**ommercial **M**odular **A**ero-**P**ropulsion **S**ystem **S**imulation) utilisé dans ce projet comprend quatre ensembles de données de complexité croissante (voir Tableau I.1) , chacun représentant les performances de moteurs turbofan au fil du temps. Ces moteurs commencent à fonctionner normalement, mais au fur et à mesure, des défaillances se développent. Les ensembles d'entraînement sont constitués de données provenant de moteurs exploités jusqu'à la défaillance complète, tandis que les ensembles de test contiennent des séries temporelles se terminant avant la défaillance et dont les valeurs sont données . L'objectif principal est de prédire la Durée de Vie Utile Restante (RUL) de chaque moteur, afin d'optimiser la maintenance et d'éviter les pannes imprévues.

Dataset	Condition opératoire	Modes de défaillance	Taille Train data (nb. de moteurs)	Taille Test data (nb. de moteurs)
FD001	ONE (SEA LEVEL)	ONE (HPC Degradation)	100	100
FD002	SIX	ONE (HPC Degradation)	260	259
FD003	ONE (SEA LEVEL)	ONE (HPC Degradation, Fan Degradation)	100	100
FD004	SIX	ONE (HPC Degradation, Fan Degradation)	248	249

Tableau I.1

Les données sont composées de simulations de plusieurs moteurs turbofan, où chaque ligne représente des informations relatives à un moteur à un instant donné. Chaque enregistrement comprend :

1. Le numéro de l'unité moteur,
2. Le temps écoulé, mesuré en cycles de fonctionnement,
3. Trois paramètres opérationnels qui influencent le comportement du moteur,
4. 21 mesures provenant de capteurs, fournissant des informations détaillées sur l'état du moteur à chaque cycle.

Ce jeu de données permet ainsi d'analyser les défaillances des moteurs et de développer un modèle prédictif pour estimer leur RUL.

Dans le cadre de ce rapport, le premier jeu de données, FD001, qui fonctionne sous un mode de défaillance unique et une seule condition opérationnelle, est utilisé pour évaluer l'efficacité et la fonctionnalité des modèles d'apprentissage proposés.

Sensor No.	Symbol	Description
1	T2	The total temperature at the fan inlet
2	T24	The total temperature at the LPC outlet
3	T30	The total temperature at the HPC outlet
4	T50	The total temperature at the LPT outlet
5	P2	Pressure at fan inlet
6	P15	The total pressure in bypass-duct
7	P30	The total pressure at HPC outlet
8	Nf	Physical fan speed
9	Nc	Physical core speed
10	epr	Engine pressure ratio (P50/P2)
11	Ps30	Static pressure at HPC outlet
12	phi	The ratio of fuel flow to Ps30
13	NRf	Corrected fan speed
14	NRc	Corrected core speed
15	BPR	Bypass ratio
16	farB	Burner fuel-air ratio
17	htBleed	Bleed enthalpy
18	Nf-dmd	Demanded fan speed
19	PCNfR-dmd	Demanded corrected fan speed
20	w31	HPT coolant bleed
21	w32	LPT coolant bleed

Tableau I.2: descriptive des 21 capteurs du moteur

2. Chargement des données :

Pour le chargement des données, l'exploration, et l'étude de la **RUL** (Remaining Useful Life), nous avons utilisé les librairies suivantes :

Numpy qui signifie Numerical Python, est l'un des packages fondamentaux pour les calculs numériques en Python. Il offre un support pour les tableaux et matrices multi-dimensionnels de grande taille, ainsi qu'une collection de fonctions mathématiques de haut niveau pour opérer sur ces tableaux

Pandas: est une bibliothèque puissante et largement utilisée pour l'analyse et la manipulation des données en Python. Elle fournit des structures de données flexibles qui facilitent la manipulation efficace des données structurées. En raison de ses capacités et de sa polyvalence, Pandas est devenu un outil essentiel pour les scientifiques des données, les analystes et les chercheurs travaillant avec Python, notamment pour les tâches liées au nettoyage, à la transformation et à l'exploration des données.

Matplotlib: est une bibliothèque complète pour la création de visualisations statiques, animées et interactives en Python. Elle peut produire une grande variété de graphiques et de figures, y compris des graphiques linéaires, des graphiques de dispersion, des diagrammes à barres, des histogrammes, des graphiques d'erreur, des graphiques circulaires, des diagrammes en boîte et même des visualisations plus complexes comme des graphiques 3D. Scikit-learn:

Scikit-learn: souvent appelé simplement Sklearn, est une bibliothèque d'apprentissage automatique en Python. Elle est construite sur NumPy, SciPy et Matplotlib. Elle est connue pour son interface de programmation simple et conviviale (API).

Seaborn : est une bibliothèque de visualisation de données basée sur Matplotlib, qui fournit une interface de haut niveau pour la création de visualisations statistiques attrayantes et informatives. Elle simplifie la création de graphiques complexes comme les cartes de chaleur, les diagrammes de violon, les diagrammes en boîte, les nuages de points avec des ajustements de régression, et plus encore, tout en intégrant bien avec Pandas pour la gestion des données. Seaborn est particulièrement apprécié pour sa capacité à produire des visualisations élégantes et intuitives avec moins de code par rapport à Matplotlib.

3. Importation des données:

- Importer le jeu de données (par exemple, le fichier CSV contenant les données FD001).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

- Introduire les noms des divers colonnes de la dataset comme mentionnée dans la partie de l'introduction (unit_nr, time_cycles , les 3 paramètres et les 21 capteurs)

```
# define filepath to read data
dir_path = './CMAPSSData/'

# define column names for easy indexing
index_names = ['unit_nr', 'time_cycles']
setting_names = ['setting_1', 'setting_2', 'setting_3']
sensor_names = ['s_{}'.format(i) for i in range(1,22)]
col_names = index_names + setting_names + sensor_names

# read data
train = pd.read_csv((dir_path+'train_FD001.txt'), sep='\s+', header=None, names=col_names)
x_test = pd.read_csv((dir_path+'test_FD001.txt'), sep='\s+', header=None, names=col_names)
y_test = pd.read_csv((dir_path+'RUL_FD001.txt'), sep='\s+', header=None, names=['RUL'])

train.head()
```

- Examiner les premières lignes avec `head()` pour comprendre la structure.(figure 3)

	unit_nr	time_cycles	setting_1	setting_2	setting_3	s_1	s_2	s_3	s_4	s_5	..
0	1	1	-0.0007	-0.0004	100.0	518.67	641.82	1589.70	1400.60	14.62	
1	1	2	0.0019	-0.0003	100.0	518.67	642.15	1591.82	1403.14	14.62	
2	1	3	-0.0043	0.0003	100.0	518.67	642.35	1587.99	1404.20	14.62	
3	1	4	0.0007	0.0000	100.0	518.67	642.35	1582.79	1401.87	14.62	
4	1	5	-0.0019	-0.0002	100.0	518.67	642.37	1582.85	1406.22	14.62	

5 rows x 26 columns

...	s_12	s_13	s_14	s_15	s_16	s_17	s_18	s_19	s_20	s_21
...	521.66	2388.02	8138.62	8.4195	0.03	392	2388	100.0	39.06	23.4190
...	522.28	2388.07	8131.49	8.4318	0.03	392	2388	100.0	39.00	23.4236
...	522.42	2388.03	8133.23	8.4178	0.03	390	2388	100.0	38.95	23.3442
...	522.86	2388.08	8133.83	8.3682	0.03	392	2388	100.0	38.88	23.3739
...	522.19	2388.04	8133.80	8.4294	0.03	393	2388	100.0	38.90	23.4044

4. Calcul du RUL (Remaining Useful Life)

Avant de commencer à visualiser nos données pour poursuivre notre analyse exploratoire des données (EDA), nous allons calculer une variable cible pour le Remaining Useful Life (RUL). Cette variable cible aura deux objectifs :

1. Elle servira d'axe X lors de la visualisation des signaux des capteurs, ce qui nous permettra d'interpréter facilement les variations des signaux des capteurs à mesure que les moteurs approchent de la panne.
2. Elle servira de variable cible pour nos modèles d'apprentissage supervisé.

Pour estimer le RUL (Remaining Useful Life) des moteurs dans le jeu d'entraînement, nous supposons qu'il diminue de manière linéaire avec le temps, atteignant 0 au dernier cycle. Ainsi, le RUL sera égal à la différence entre le cycle maximum de fonctionnement d'un moteur et le cycle actuel.(figure 5)

Pour le calcul :

1. Nous regroupons les données par moteur (`unit_nr`) pour déterminer le cycle maximum (`max_time_cycle`) de chaque moteur.
2. Cette valeur est ensuite fusionnée avec les données pour calculer le RUL en soustrayant `time_cycle` de `max_time_cycle`.

3. Enfin, nous supprimons la colonne `max_time_cycle` et vérifions les premières lignes pour valider le calcul du RUL.

```
#ComputingRUL
def add_remaining_useful_life(df):
    # Get the total number of cycles for each unit
    grouped_by_unit = df.groupby(by="unit_nr")
    max_cycle = grouped_by_unit["time_cycles"].max()

    # Merge the max cycle back into the original frame
    result_frame = df.merge(max_cycle.to_frame(name='max_cycle'), left_on='unit_nr', right_index=True)

    # Calculate remaining useful life for each row
    remaining_useful_life = result_frame["max_cycle"] - result_frame["time_cycles"]
    result_frame["RUL"] = remaining_useful_life

    # drop max_cycle as it's no longer needed
    result_frame = result_frame.drop("max_cycle", axis=1)
    return result_frame

train = add_remaining_useful_life(train)
train[index_names+['RUL']].head()
```

Comme le montre le code ci-dessous l'URL associée à notre valeur maximale de Time cycle pour l'unité n°1 est 192 cycles. Donc pour le premier URL on a $192-1=191$:

```
train[index_names+['RUL']].head()
```

✓ 0.0s

	unit_nr	time_cycles	RUL
0	1	1	191
1	1	2	190
2	1	3	189
3	1	4	188
4	1	5	187

5. Résumé statistique :

En examinant les statistiques descriptives de `unit_nr`, nous constatons que le jeu de données contient un total de 20 631 lignes, avec des numéros d'unité allant de 1 à 100, comme prévu. Ce qui est intéressant, c'est que la moyenne et les quantiles ne correspondent pas parfaitement aux statistiques descriptives d'un vecteur de 1 à 100. Cela s'explique par le fait que chaque unité a un nombre de cycles maximum différent, et donc un nombre de lignes variables.

```
train['unit_nr'].describe()
```

✓ 0.0s

```
count    20631.000000
mean       51.506568
std       29.227633
min        1.000000
25%       26.000000
50%       52.000000
75%       77.000000
max      100.000000
Name: unit_nr, dtype: float64
```

En examinant les cycles maximums (**max_time_cycles**), on observe que le moteur ayant échoué le plus tôt l'a fait après 128 cycles, tandis que celui ayant fonctionné le plus longtemps est tombé en panne après 362 cycles. En moyenne, les moteurs tombent en panne entre 199 et 206 cycles, mais l'écart type de 46 cycles est assez important.

Nous visualisons cela plus en détail ci-dessous pour mieux comprendre.

```
result = train[['unit_nr', 'time_cycles']].groupby('unit_nr').max()
result.describe()
```

✓ 0.0s

	time_cycles
count	100.000000
mean	206.310000
std	46.342749
min	128.000000
25%	177.000000
50%	199.000000
75%	229.250000
max	362.000000

La description de la dataset montre que pour chacun des 4 dataset le moteur fonctionne selon un mode opératoire bien défini. On peut vérifier cela par les données des 3 paramètres:

```
train[setting_names].describe()
```

	setting_1	setting_2	setting_3
count	20631.000000	20631.000000	20631.0
mean	-0.000009	0.000002	100.0
std	0.002187	0.000293	0.0
min	-0.008700	-0.000600	100.0
25%	-0.001500	-0.000200	100.0
50%	0.000000	0.000000	100.0
75%	0.001500	0.000300	100.0
max	0.008700	0.000600	100.0

En examinant la déviation standard des trois paramètres il s'avère que les paramètres 1 et 2 ne sont pas complètement stables. Cependant, les fluctuations sont assez minimes.

6. Vérifications des valeurs manquantes:

Avant de procéder à l'imputation des valeurs manquantes, il est essentiel de comprendre leur impact sur les analyses et les modèles prédictifs. Les données manquantes peuvent réduire la qualité des résultats et biaiser les prédictions. Une approche couramment utilisée consiste à remplacer les valeurs manquantes par des mesures statistiques comme la moyenne, particulièrement adaptée lorsque les données présentent une distribution symétrique et sans outliers significatifs. Cette méthode permet de maintenir la cohérence et d'assurer une compatibilité avec les algorithmes d'apprentissage automatique.

```
# Imputer les NAN avec la moyenne
train.fillna(train.mean(), inplace=True)
x_test.fillna(x_test.mean(), inplace=True)
y_test.fillna(y_test.mean(), inplace=True)
```

```
def check_missing_values(data):
    print('Verifying the existence of null data:')
    return data.isnull().sum()

print(check_missing_values(train))
```

```
Verifying the existence of null data:
```

```
unit_nr      0
time_cycles  0
setting_1     0
setting_2     0
setting_3     0
s_1           0
s_2           0
s_3           0
s_4           0
s_5           0
s_6           0
s_7           0
s_8           0
s_9           0
s_10          0
s_11          0
s_12          0
s_13          0
s_14          0
s_15          0
s_16          0
s_17          0
s_18          0
s_19          0
s_20          0
s_21          0
dtype: int64
```

Les données semblent donc ne pas avoir de valeurs manquantes. On peut maintenant procéder à l'analyse graphique.

7. Tableau descriptif:

Utiliser `describe()` pour obtenir des statistiques descriptives comme la moyenne, la médiane, l'écart-type, les minimums et les maximums des colonnes. Cela permet de détecter des anomalies ou des colonnes ayant peu de variance.

Tableau descriptives des divers capteurs

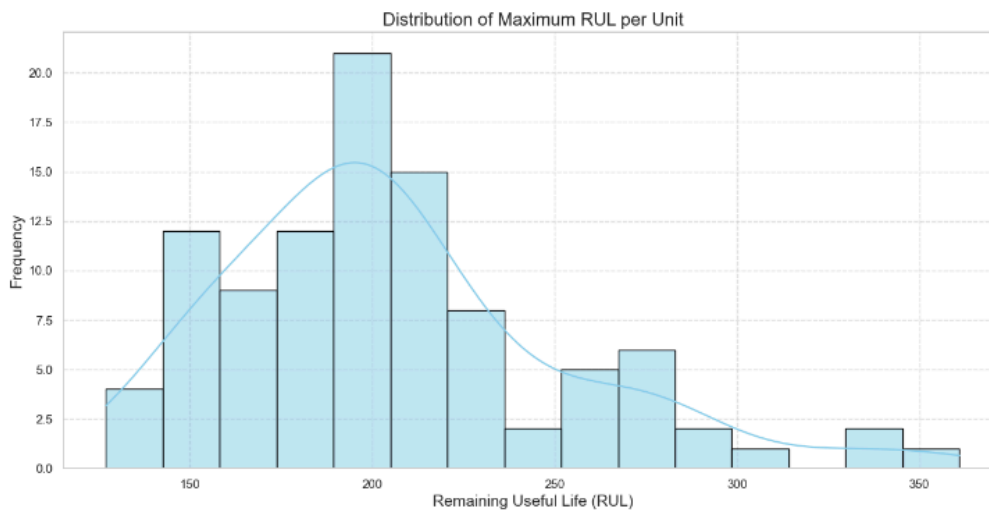
	count	mean	std	min	25%	50%	75%	max
s_1	20631	518,67	0	518,67	518,67	518,67	518,67	518,67
s_2	20631	642,6809335	0,50005327	641,21	642,325	642,64	643	644,53
s_3	20631	1590,523119	6,13114952	1571,04	1586,26	1590,1	1594,38	1616,91
s_4	20631	1408,933782	9,000604781	1382,25	1402,36	1408,04	1414,555	1441,49
s_5	20631	14,62	1,7764E-15	14,62	14,62	14,62	14,62	14,62
s_6	20631	21,60980321	0,001388985	21,6	21,61	21,61	21,61	21,61
s_7	20631	553,3677112	0,885092258	549,85	552,81	553,44	554,01	556,06
s_8	20631	2388,096652	0,070985479	2387,9	2388,05	2388,09	2388,14	2388,56
s_9	20631	9065,242941	22,08287953	9021,73	9053,1	9060,66	9069,42	9244,59
s_10	20631	1,3	0	1,3	1,3	1,3	1,3	1,3
s_11	20631	47,54116815	0,267087399	46,85	47,35	47,51	47,7	48,53
s_12	20631	521,41347	0,737553392	518,69	520,96	521,48	521,95	523,38
s_13	20631	2388,096152	0,071918916	2387,88	2388,04	2388,09	2388,14	2388,56
s_14	20631	8143,752722	19,07617598	8099,94	8133,245	8140,54	8148,31	8293,72
s_15	20631	8,442145582	0,037505038	8,3249	8,4149	8,4389	8,4656	8,5848
s_16	20631	0,03	1,38781E-17	0,03	0,03	0,03	0,03	0,03
s_17	20631	393,2106539	1,548763025	388	392	393	394	400
s_18	20631	2388	0	2388	2388	2388	2388	2388
s_19	20631	100	0	100	100	100	100	100
s_20	20631	38,81627066	0,180746428	38,14	38,7	38,83	38,95	39,43
s_21	20631	23,28970536	0,108250875	22,8942	23,2218	23,2979	23,3668	23,6184

En examinant l'écart type, il est clair que les capteurs 1, 10, 18 et 19 ne fluctuent pas du tout et peuvent être éliminés en toute sécurité, car ils ne contiennent aucune information utile.

L'analyse des quantiles indique que les capteurs 5, 6 et 16 présentent peu de fluctuations et nécessitent une inspection plus approfondie.

Les capteurs 9 et 14 montrent les fluctuations les plus élevées, mais cela ne signifie pas que les autres capteurs ne contiennent pas d'informations précieuses.

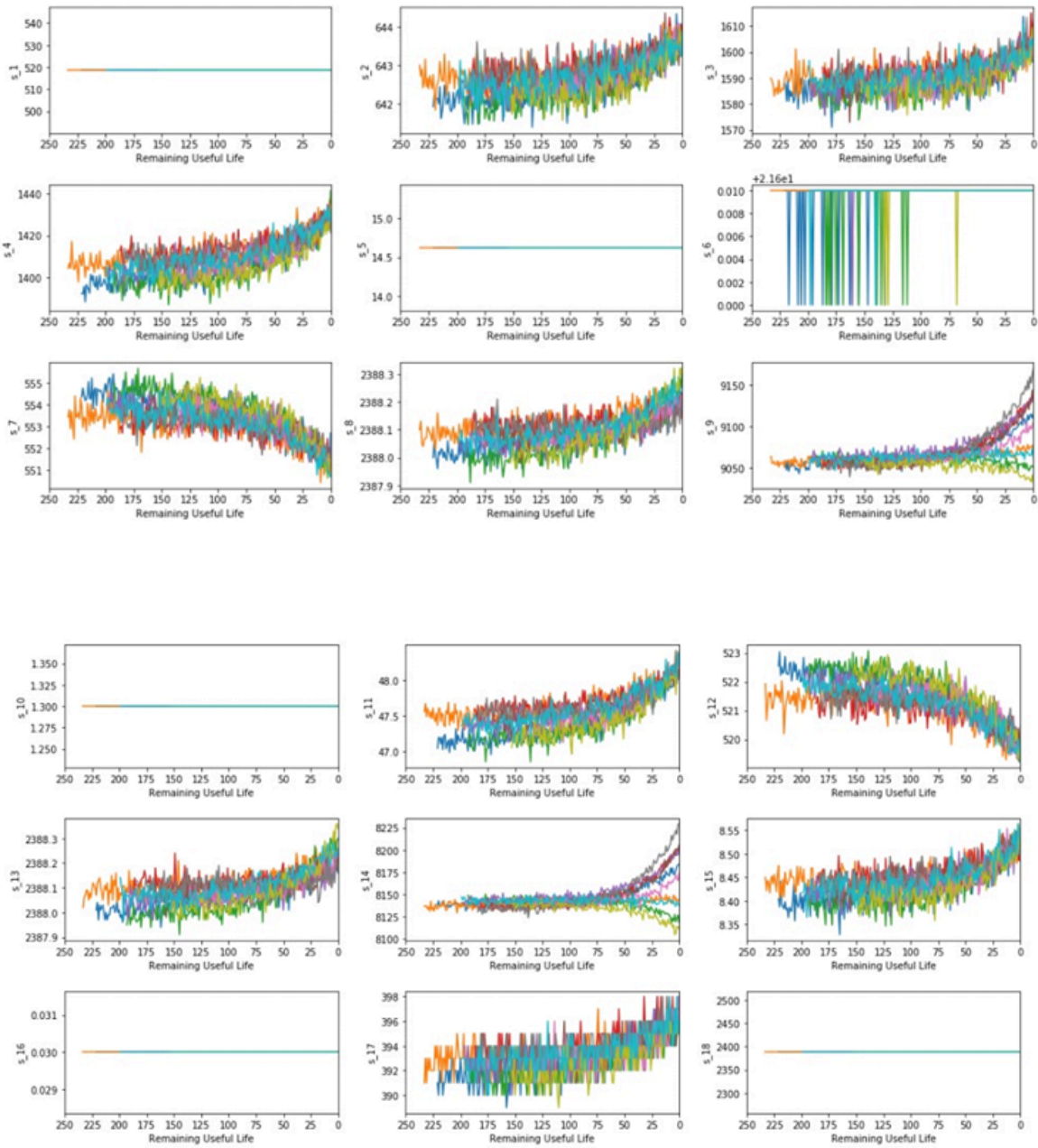
8. Visualisation:

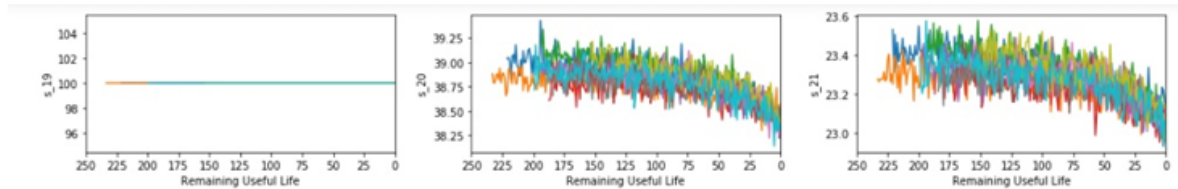


L'histogramme suivant montre une distribution normale; la plupart des moteurs ont un cycle de vie en moyenne de 200 cycles. Il y a quelques moteurs qui ont un cycle de vie qui dépasse 300.

En raison du grand nombre de moteurs, il n'est pas favorable de tracer chaque moteur pour chaque capteur. Les graphiques deviendraient illisibles avec autant de lignes dans un seul tracé. Par conséquent, on choisit de tracer uniquement les moteurs dont le numéro d'unité (`unit_nr`) est divisible par 10 sans reste. Nous inversons l'axe X afin que le RUL diminue le long de l'axe, avec un RUL de zéro indiquant une panne du moteur.

En raison du grand nombre de capteurs, on discuterait de quelques graphiques représentatifs de l'ensemble.





9. Analyse exploratoire des capteurs:

Cette étape consiste à examiner les signaux des capteurs en relation avec la durée de vie résiduelle (RUL) afin de différencier les capteurs "utiles" des capteurs "non pertinents". L'objectif est d'identifier les capteurs riches en informations par rapport à ceux qui le sont moins. Pour ce faire, une fonction appelée *plot_sensor* est utilisée, qui trace les signaux des capteurs pour chaque dixième turboréacteur. Les représentations graphiques de ces signaux pour tous les capteurs sont disponibles dans les figures ci-dessus.

- Les graphiques des capteurs 1, 10, 18 et 19 présentent une ligne plate, indiquant que ces capteurs ne contiennent aucune information utile. Cette observation confirme les conclusions issues des statistiques descriptives.
- Les capteurs 5 et 16 montrent également une ligne plate. Ils peuvent donc être ajoutés à la liste des capteurs à exclure.
- Le capteur 2 affiche une tendance ascendante, un schéma similaire étant observé pour les capteurs 3, 4, 8, 11, 13, 15 et 17.
- Les relevés du capteur 6 montrent des pics vers le bas à certains moments, mais aucune relation claire avec la diminution de la durée de vie résiduelle (RUL) n'a été identifiée.
- Le capteur 7 présente une tendance descendante, un schéma également observé pour les capteurs 12, 20 et 21.
- Le capteur 9 suit un modèle similaire à celui du capteur 14.

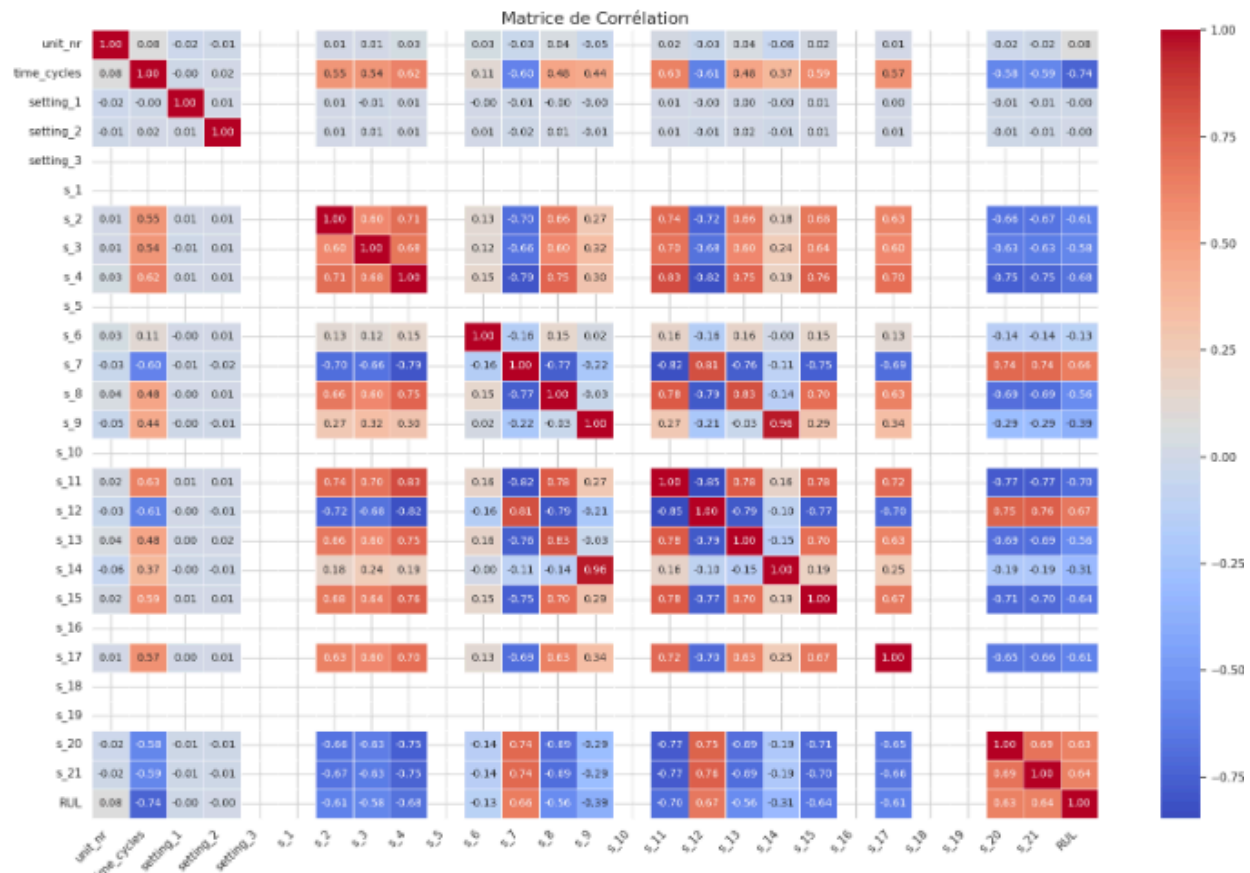
→ Sur la base de cette analyse exploratoire des données (EDA), nous pouvons conclure que les capteurs suivants n'apportent aucune information liée à la RUL, car leurs valeurs restent constantes dans le temps : **capteurs 1, 5, 6, 10, 16, 18 et**

19. Ces capteurs seront donc exclus des prédicteurs avant le développement des modèles.

10. Extraction des caractéristiques:

Une *heatmap* (carte thermique) est un outil de visualisation des données qui utilise des couleurs pour représenter la corrélation entre les caractéristiques. Elle permet d'avoir une vue d'ensemble des relations entre deux variables (ou caractéristiques).

```
def corr_matrix(data):  
    plt.figure(figsize=(18, 12)) # Augmente la taille de la figure  
    sns.heatmap(  
        data.corr(),  
        annot=True,  
        fmt=".2f", # Limite les nombres à 2 décimales  
        cmap='RdYlGn',  
        annot_kws={"size": 10}, # Définit la taille des annotations  
        linewidths=0.5 # Ajoute des séparateurs entre les cellules  
    )  
    plt.xticks(rotation=45, ha='right') # Rend les labels plus lisibles  
    plt.yticks(rotation=0) # Labels y en orientation horizontale  
    plt.title("Matrice de Corrélation", fontsize=16) # Titre plus lisible  
    plt.tight_layout() # Évite les débordements  
    plt.show()  
  
# Utilisation  
corr_matrix(train)
```



La corrélation quantifie le degré selon lequel deux variables évoluent l'une par rapport à l'autre. Les valeurs de corrélation se situent dans une plage allant de -1 à 1 :

- Une corrélation de **1** indique une relation positive parfaite.
- Une corrélation de **-1** indique une relation négative parfaite.
- Une corrélation proche de **0** indique peu ou pas de relation.

11. Sélection des caractéristiques essentielles:

Seules les caractéristiques ayant une valeur absolue de corrélation avec la RUL supérieure ou égale à 0,5 sont sélectionnées. Cette démarche vise à retenir uniquement les caractéristiques essentielles pour la construction du modèle, évitant ainsi les risques de sur-apprentissage (*overfitting*). Les capteurs présentant une forte corrélation avec la RUL confirment les résultats précédents de la visualisation des données des capteurs.

```
# Calculer la matrice de corrélation
correlation = train.corr()

# Sélectionner les caractéristiques fortement corrélées avec 'RUL'
train_relevant_features = correlation[abs(correlation['RUL']) >= 0.5]

# Extraire uniquement la colonne pour 'RUL'
train_relevant_features['RUL']
```

```
time_cycles    -0.736241
s_2            -0.606484
s_3            -0.584520
s_4            -0.678948
s_7             0.657223
s_8            -0.563968
s_11           -0.696228
s_12           0.671983
s_13           -0.562569
s_15           -0.642667
s_17           -0.606154
s_20           0.629428
s_21           0.635662
RUL            1.000000
Name: RUL, dtype: float64
```

Étapes suivantes

- Seules les caractéristiques importantes seront conservées dans les ensembles d'entraînement (*train set*) et de test (*test set*).
- L'ensemble d'entraînement sera scindé en deux parties : **x_train** (caractéristiques) et **y_train** (valeur cible).
- Ces étapes préliminaires permettront de commencer la construction des modèles.

```
# Création d'une liste contenant uniquement les caractéristiques importantes
list_relevant_features = train_relevant_features.index
list_relevant_features = list_relevant_features[1:] # Exclut 'RUL' de la liste

# Conserver uniquement ces caractéristiques importantes dans le jeu d'entraînement
train = train[list_relevant_features]

# Séparer le jeu d'entraînement en x_train (caractéristiques) et y_train (RUL)
y_train = train['RUL']
x_train = train.drop(['RUL'], axis=1)

# Conserver uniquement les colonnes/caractéristiques de x_train dans le jeu de test
x_test = x_test[x_train.columns]

# Affichage des premières lignes de x_train
x_train.head()
```

	s_2	s_3	s_4	s_7	s_8	s_11	s_12	s_13	s_15	s_17	s_20	s_21
0	641.82	1589.70	1400.60	554.36	2388.06	47.47	521.66	2388.02	8.4195	392	39.06	23.4190
1	642.15	1591.82	1403.14	553.75	2388.04	47.49	522.28	2388.07	8.4318	392	39.00	23.4236
2	642.35	1587.99	1404.20	554.26	2388.08	47.27	522.42	2388.03	8.4178	390	38.95	23.3442
3	642.35	1582.79	1401.87	554.45	2388.11	47.13	522.86	2388.08	8.3682	392	38.88	23.3739
4	642.37	1582.85	1406.22	554.00	2388.06	47.28	522.19	2388.04	8.4294	393	38.90	23.4044

12. Standardisation:

La standardisation est une méthode de prétraitement utilisée en apprentissage automatique et en statistiques pour mettre à l'échelle les variables afin qu'elles aient une moyenne de zéro et un écart-type de un. Cette étape est essentielle pour s'assurer que toutes les variables sont sur la même échelle, ce qui peut être crucial pour de nombreux algorithmes d'apprentissage automatique. Lorsque les variables d'un ensemble de données ont des échelles différentes, les algorithmes risquent d'être influencés par celles ayant les échelles les plus grandes. La standardisation permet d'optimiser le processus d'entraînement et peut améliorer les performances du modèle.

Dans les données observées, une variation significative des valeurs entre les différents capteurs est visible. Par conséquent, il est important de normaliser ces valeurs afin que toutes les variables soient recentrées avec une moyenne de zéro et une variance de un. Cette standardisation peut être réalisée facilement en utilisant la méthode `StandardScaler()` de Sklearn, suivie de `fit_transform()`. Ainsi, les ensembles de données d'entraînement et de test sont

standardisés, et les valeurs transformées (nommées ici `x_train_trans` et `x_test_trans`) seront utilisées dans les étapes ultérieures de construction du modèle.

```
# Premièrement, créer une fonction d'évaluation
def evaluate(y_true, y_hat, label='test'):
    mse = mean_squared_error(y_true, y_hat)
    rmse = np.sqrt(mse)
    variance = r2_score(y_true, y_hat)
    print('{} set RMSE:{}, R2:{}'.format(label, rmse, variance))
```

```
# Standardisation des caractéristiques
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train_trans = sc.fit_transform(x_train)
x_test_trans = sc.transform(x_test)
```

II- Modélisation et Entraînement des Modèles

Dans cette section, plusieurs modèles d'apprentissage automatique sont développés afin de comparer leurs performances. Parmi ces modèles, on retrouve la régression linéaire, la régression polynomiale, la régression par vecteurs de support, l'arbre de décision, la forêt aléatoire et le réseau de neurones artificiels. Grâce aux bibliothèques **Scikit-learn** et **TensorFlow**, la construction et l'entraînement de ces modèles sont simplifiés. Les codes associés à chaque modèle sont présentés dans les sections suivantes.

1. Le Testing :

On va utiliser deux paramètres pour évaluer nos différents modèles au cours de l'étude :

-**RMSE**(Root Mean Squared) mesure l'écart quadratique moyen entre les prédictions d'un modèle et les valeurs réelles. C'est une métrique utilisée pour évaluer la précision des modèles de régression.

Définition mathématique :

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

- y_i : Valeur réelle.
- \hat{y}_i : Valeur prédite par le modèle.
- n : Nombre de données.

Interprétation :

- Le RMSE donne une estimation de l'erreur moyenne en **unités de la variable cible** (le nombre de cycles).
- Si le RMSE est faible, cela signifie que les prédictions sont proches des valeurs réelles.

Explained Variance Score (ou R^2):

Le R^2 , ou **coefficient de détermination**, mesure la proportion de la variance dans les données cibles qui est expliquée par le modèle.

Définition mathématique :

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

- y_i : Valeur réelle.
- \hat{y}_i : Valeur prédite par le modèle.
- \bar{y} : Moyenne des valeurs réelles.

Interprétation :

- $R^2 = 1$: Le modèle explique parfaitement les données.
- $R^2 = 0$: Le modèle n'explique rien (équivalent à toujours prédire la moyenne).
- $R^2 < 0$: Le modèle est pire que simplement utiliser la moyenne comme prédiction constante.

Différences Clés :

1. **RMSE :**

- Mesure une erreur absolue (en unités de la variable cible).
- Plus il est bas, meilleur est le modèle.

2. **R^2 (Explained Variance) :**

- Mesure la proportion de la variance expliquée par le modèle (entre 0 et 1).
- Plus il est proche de 1, meilleur est le modèle.

Ces deux métriques sont complémentaires pour évaluer un modèle de régression.

Par exemple :

- Un RMSE faible indique que les erreurs sont petites.
- Un R^2 élevé (proche de 1) montre que le modèle capture bien la structure des données.

2. Régression linéaire:

Pour la régression linéaire, les métriques **R-carré** et **RMSE** sont utilisées pour évaluer les performances sur les ensembles d'entraînement et de test. Sur l'ensemble d'entraînement, la valeur **R-carré** obtenue est d'environ **0,56**, ce qui reste modeste et peut s'expliquer par le manque de linéarité des données. En revanche, sur l'ensemble de test, la valeur **R-carré** diminue à **0,35**, indiquant des performances encore plus faibles. Ces résultats peu concluants nous conduisent à explorer d'autres modèles pour améliorer la qualité des prédictions.

```
# Modèle de régression linéaire
from sklearn.linear_model import LinearRegression

lm = LinearRegression()
lm.fit(x_train_trans, y_train)

# Prédire et évaluer
y_hat_train = lm.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train, 'train')

y_hat_test = lm.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test, 'test')
```

```
train set RMSE:45.61466077800371, R2:0.5614378099126991
test set RMSE:33.301161070181, R2:0.3578164045379344
```

3. Régression polynomiale:

La régression polynomiale n'apporte pas d'amélioration significative. Bien que la valeur **R-carré** sur l'ensemble d'entraînement s'améliore légèrement, atteignant **0,61**, les performances sur l'ensemble de test se dégradent avec une valeur **R-carré** de **0,29**, inférieure à celle de la régression linéaire. Ces résultats montrent que la complexification du modèle ne garantit pas nécessairement de meilleures performances.


```
# Régression polynomiale
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Définir les caractéristiques polynomiales avec un degré de 4
poly_reg = PolynomialFeatures(degree=4)
x_poly = poly_reg.fit_transform(x_train_trans)

# Créer le modèle de régression linéaire
lin_reg_2 = LinearRegression()
lin_reg_2.fit(x_poly, y_train)

# Prédire et évaluer
y_hat_train = lin_reg_2.predict(x_poly)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train, 'train')

y_hat_test = lin_reg_2.predict(poly_reg.fit_transform(x_test_trans))
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test, 'test')
```

```
train set RMSE:42.89372356002194, R2:0.6121982317015553
test set RMSE:34.87992039035855, R2:0.2954830831246609
```

4. Régression par Support Vector Machine (SVR):

Les résultats montrent que la régression par Support Vector Machine (SVR) surpasse à la fois la régression linéaire et polynomiale, obtenant une valeur de R^2 de 0,54 sur l'ensemble de test. En général, la SVR offre de meilleures performances que les autres modèles classiques d'apprentissage automatique, notamment lorsqu'il s'agit de traiter des données non linéaires.

```
# SVR
from sklearn.svm import SVR

# Création du modèle
regressor = SVR(kernel='rbf')

# Entraînement
regressor.fit(x_train_trans, y_train)

# Prédictions et évaluation
y_hat_train = regressor.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train)

y_hat_test = regressor.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test)
```

```
test set RMSE:45.28091055080134, R2:0.5678320152955884
test set RMSE:28.043979046613842, R2:0.544572004297883
```

5. Régression par arbre de décision: (Decision tree regression)

Les résultats pour l'arbre de décision montrent une excellente performance sur les données d'entraînement, avec une valeur de R^2 de 0,71, ce qui est supérieur aux modèles précédents. Cependant, sa performance chute considérablement sur l'ensemble de test, avec un R^2 de 0,15. Une explication possible de cette divergence est le **surapprentissage** (overfitting), où le modèle s'ajuste très bien aux données d'entraînement mais échoue à se généraliser efficacement aux nouvelles données.

```
# Decision Tree
from sklearn.tree import DecisionTreeRegressor

# Création du modèle
dt = DecisionTreeRegressor(random_state=42, max_depth=15, min_samples_leaf=10)

# Entraînement
dt.fit(x_train_trans, y_train)

# Prédictions et évaluation
y_hat_train = dt.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train, 'train')

y_hat_test = dt.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test)
```

```
train set RMSE:36.77513644517892, R2:0.7149435492153497
test set RMSE:38.31216453330952, R2:0.150009965299418
```

6. Régression par forêt aléatoire (RFR: Random forest regression)

D'après les résultats du modèle de régression par forêt aléatoire présentés dans la Figure ci-dessous, ce modèle n'a pas rencontré les mêmes problèmes de sur-apprentissage (overfitting) que le modèle d'arbre de décision. Cela se traduit par une meilleure performance sur le jeu de test, atteignant une valeur de R^2 de 0,39. Cependant, ce résultat reste loin d'être optimal. Dans les sections suivantes, un modèle de réseau de neurones artificiels (ANN) sera implémenté pour évaluer s'il peut surpasser les autres modèles.

```
# Random Forest
from sklearn.ensemble import RandomForestRegressor

# Création du modèle
rf = RandomForestRegressor(random_state=42, n_jobs=-1, max_depth=6, min_samples_leaf=5)

# Entraînement
rf.fit(x_train_trans, y_train)

# Prédictions et évaluation
y_hat_train = rf.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train, 'train')

y_hat_test = rf.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test)
```

```
train set RMSE:44.793331149655884, R2:0.5770889724209527  
test set RMSE:32.42845735268905, R2:0.391034015379521
```

7. Réseau de neurones artificiels: (ANN: Artificial Neural Network)

Dans cette section, nous cherchons à construire un réseau de neurones artificiels en sélectionnant des hyperparamètres optimisés pour notre jeu de données. Les hyperparamètres définissent l'architecture et le comportement du réseau pendant la phase d'entraînement. Ils incluent plusieurs éléments, tels que le nombre de couches, le nombre de neurones par couche, les fonctions d'activation, la taille des lots (batch size), le choix de l'optimiseur, le nombre d'époques, et bien plus encore.

La recherche des meilleurs hyperparamètres consiste à tester différentes combinaisons de ces variables, en entraînant et en évaluant les performances du réseau pour chaque configuration. Il est important de noter qu'il n'existe pas de formule universelle pour cette démarche. Le choix des hyperparamètres repose généralement sur un processus itératif et intuitif.

Nous avons initialement conçu une architecture ANN comprenant deux couches cachées, chacune comportant 10 neurones. La fonction d'activation ReLU (Rectified Linear Unit) a été sélectionnée pour ces deux couches, avec une taille de lot de 32. En ce qui concerne les performances, les résultats de l'ANN sont globalement comparables à ceux du modèle de régression linéaire, avec une valeur de R^2 de 0,33 sur le jeu de test et de 0,58 sur le jeu d'entraînement.

```

# Réseau de neurones artificiels (ANN)
import tensorflow as tf

# Création du modèle séquentiel
ann = tf.keras.models.Sequential()

# Ajout des couches cachées avec activation ReLU
ann.add(tf.keras.layers.Dense(units=10, activation='relu'))
ann.add(tf.keras.layers.Dense(units=10, activation='relu'))

# Ajout de la couche de sortie
ann.add(tf.keras.layers.Dense(units=1))

# Compilation du modèle avec l'optimiseur Adam et une fonction de perte MSE
ann.compile(optimizer='adam', loss='mean_squared_error')

# Entraînement du modèle
ann.fit(x_train_trans, y_train, batch_size=32, epochs=75)

# Prédictions et évaluation pour les données d'entraînement
y_hat_train = ann.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train, 'train')

# Prédictions et évaluation pour les données de test
y_hat_test = ann.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test, 'test')

```

```

train set RMSE:44.133774357769354, R2:0.5894515514373779
4/4 ————— 0s 3ms/step
test set RMSE:33.835705703719995, R2:0.3370344638824463

```

8. KNN (k plus proches voisins):

Les résultats obtenus à l'aide du modèle KNN (k-plus proches voisins) montrent une capacité modérée à prédire la durée de vie utile restante (RUL) des moteurs. Avec un RMSE de 32,21 et un R^2 de 0,39 sur l'ensemble de test, ce modèle se situe dans une performance intermédiaire par rapport aux autres approches testées. La principale force de KNN réside dans sa simplicité et son efficacité sur des jeux de données de taille modérée. Toutefois, ses performances peuvent être limitées en raison de la nature locale de l'algorithme, qui dépend fortement de la proximité des données dans l'espace des caractéristiques.

```
# KNN Regressor
from sklearn.neighbors import KNeighborsRegressor

# Étape 1 : Choisir la valeur optimale de k
k_values = range(1, 21) # Tester k de 1 à 20
errors = []

# Tester chaque valeur de k
for k in k_values:
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(x_train_trans, y_train)
    y_pred = knn.predict(x_test_trans)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    errors.append(rmse)

# Déterminer la valeur optimale de k
optimal_k = k_values[np.argmin(errors)]
print(f"Valeur optimale de k: {optimal_k} avec RMSE = {min(errors):.4f}")

# Étape 2 : Création du modèle avec k optimal
knn = KNeighborsRegressor(n_neighbors=optimal_k)

# Entraînement
knn.fit(x_train_trans, y_train)

# Prédiction et évaluation pour les données d'entraînement
y_hat_train = knn.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train, 'train')
```

```
# Prédiction et évaluation pour les données de test
y_hat_test = knn.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test, 'test')

# Visualisation des résultats
plt.figure(figsize=(10, 6))
plt.plot(range(len(y_test)), y_test, label='Valeurs Réelles', marker='o')
plt.plot(range(len(y_test)), y_hat_test, label='Valeurs Prédites', marker='x')
plt.title("KNN Regressor : Comparaison Réel vs Prédit")
plt.xlabel("Index des exemples")
plt.ylabel("Remaining Useful Life (RUL)")
plt.legend()
plt.grid()
plt.show()
```

```
Valeur optimale de k: 13 avec RMSE = 32.2144
train set RMSE:42.915954449778724, R2:0.6117961489851103
test set RMSE:32.214406531328194, R2:0.3990466980450169
```

9. Proposition d'amélioration :

Dans cette partie on va procéder autrement pour améliorer les résultats de nos prédictions:

a- Hypothèse URL :

Régression linéaire:

```
train set RMSE:45.61466077800371, R2:0.5614378099126991
test set RMSE:33.301161070181, R2:0.3578164045379344
```

Points Clés dans l'Observation :

1. RMSE Plus Faible sur le Jeu de Test

- En général, un modèle performe mieux sur le jeu d'entraînement, car il est optimisé pour minimiser l'erreur sur ces données. Un RMSE plus faible sur le jeu de test est contre-intuitif, car le modèle n'a pas été formé sur ces données.

- Ici, le RMSE plus faible sur le jeu de test suggère que les données d'entraînement contiennent des motifs ou des complexités que le modèle a du mal à reproduire.

2. Valeurs de RUL dans le Jeu d'Entraînement Plus Élevées

- Les valeurs de **Remaining Useful Life (RUL)** dans le jeu d'entraînement atteignent les 300 cycles, représentant des instances éloignées de la défaillance. Ces valeurs élevées de RUL ne semblent pas bien corrélées avec les signaux des capteurs, ce qui entraîne des erreurs plus importantes sur le jeu d'entraînement.

3. Jeu de Test Plus Proche de la Défaillance

- Le jeu de test contient des valeurs de RUL proches de la défaillance, où la corrélation entre les signaux des capteurs et le RUL est plus forte. Cette relation plus claire facilite la prédiction pour le modèle, ce qui donne un RMSE plus faible.

Explication Possible :

Ce phénomène met en évidence un **problème dans l'hypothèse ou la méthode de calcul du RUL** :

- Le calcul **linéaire du RUL** dans le jeu d'entraînement ne reflète pas correctement la vraie relation entre les signaux des capteurs et le RUL, surtout pour les valeurs élevées de RUL.
- Cette inadéquation introduit du bruit ou des incohérences dans le jeu d'entraînement, rendant plus difficile l'apprentissage des motifs clairs par le modèle.

En revanche, dans le jeu de test, avec des valeurs de RUL plus proches de la défaillance, la relation est mieux alignée avec les signaux des capteurs, ce qui permet au modèle de mieux prédire.

Implications et Étapes Suivantes

1. Revoir le Calcul du RUL

- Il est nécessaire de corriger l'hypothèse erronée du calcul linéaire du RUL. Une approche non linéaire pourrait mieux capturer la relation entre les signaux des capteurs et le RUL.

2. Analyser les Distributions des Jeux de Données

- Comparer les distributions des signaux des capteurs et des valeurs de RUL entre le jeu d'entraînement et le jeu de test pour identifier d'éventuelles divergences.
- Si les distributions diffèrent significativement, cela pourrait indiquer un biais d'échantillonnage ou une fuite d'informations.

3. Ingénierie des Caractéristiques (Feature Engineering)

- Créer de nouvelles caractéristiques qui capturent mieux la corrélation entre les signaux des capteurs et le RUL, notamment pour les valeurs élevées de RUL dans le jeu d'entraînement.

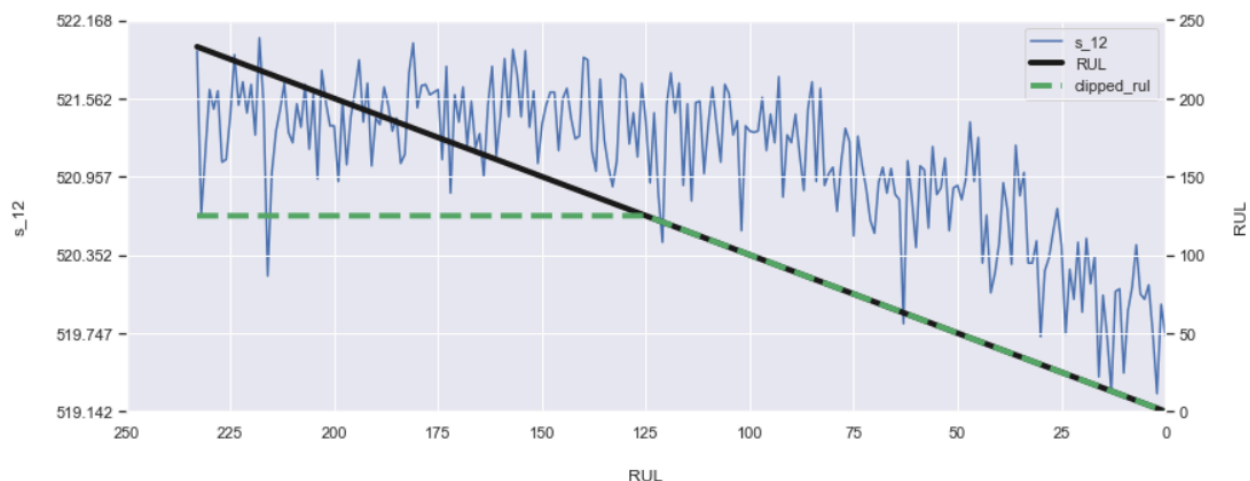
4. Affiner le Modèle

- Tester différentes fonctions de perte, architectures ou techniques de régularisation pour aider le modèle à mieux généraliser sur des plages de RUL variées.

→ La différence de RMSE entre les jeux d'entraînement et de test provient probablement d'une mauvaise correspondance entre les valeurs élevées de RUL dans le jeu d'entraînement et les motifs des signaux des capteurs. En corrigeant l'hypothèse du RUL et en alignant mieux les jeux de données, il sera possible d'améliorer la fiabilité et la performance prédictive du modèle.

Amélioration de la data set :

En observant les signaux des capteurs (voir un exemple ci-dessous), de nombreux capteurs semblent assez constants au début. Cela s'explique par le fait que les moteurs ne développent une panne qu'au fil du temps. L'inflexion dans la courbe du signal représente la première indication que le moteur commence à se dégrader et marque le moment où il devient raisonnable de supposer que le RUL (Remaining Useful Life) diminue de manière linéaire. Avant ce point, il est difficile de tirer des conclusions sur le RUL, car nous n'avons aucune information sur l'usure initiale.



Nous pouvons mettre à jour notre hypothèse pour refléter cette logique. Au lieu de supposer une diminution linéaire du RUL (Remaining Useful Life), nous définissons le RUL comme étant

initialement constant, puis diminuant de manière linéaire après un certain temps (voir l'exemple ci-dessus). En procédant ainsi, nous obtenons deux avantages :

1. Un RUL initialement constant corrèle mieux avec le signal moyen des capteurs, qui est également initialement constant.
2. Des valeurs maximales plus basses pour le RUL entraînent une moindre dispersion de notre variable cible, ce qui facilite l'ajustement d'une ligne de régression.

Par conséquent, cette modification permet à notre modèle de régression de prédire plus précisément les faibles valeurs de RUL, qui sont souvent plus intéressantes ou critiques à prédire correctement.

Avec **Pandas**, vous pouvez simplement limiter (clip) le RUL précédemment calculé de manière linéaire à une valeur maximale souhaitée. Des tests avec plusieurs valeurs limites ont montré que le fait de couper le RUL à **125** donne la plus grande amélioration pour le modèle.

```
# Modèle de régression linéaire clipped
from sklearn.linear_model import LinearRegression
y_train_clipped = y_train.clip(upper =125)
lm = LinearRegression()
lm.fit(x_train_trans, y_train_clipped)

# Prédire et évaluer
y_hat_train = lm.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train_clipped, y_hat_train, 'train')

y_hat_test = lm.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test, 'test')
```

Puisque nous mettons à jour notre hypothèse concernant le RUL par la méthode de régression linéaire pour le jeu d'entraînement, nous devrions inclure ce changement dans notre évaluation. Cependant, le véritable RUL du jeu de test reste inchangé. Examinons l'effet de cette modification.

```
train set RMSE:22.734164950962253, R2:0.7023848970100307
test set RMSE:22.91426532858463, R2:0.6959448729951724
```

L'RMSE de l'entraînement a plus que diminué de moitié. Bien sûr, nous avons défini ces cibles nous-mêmes, mais cela montre à quel point l'hypothèse précédente sur le RUL avait un impact sur la performance globale du modèle. Ce qui est bien plus important, cependant, c'est

l'amélioration sur l'ensemble de test. L'RMSE de test est passé de 33,3 à 22,91, soit une amélioration d'environ 31 %

Cela nous indique que l'hypothèse mise à jour est bénéfique pour modéliser le véritable RUL. On passe maintenant à une autre méthode qui pourrait nous donner des valeurs de test meilleur.

b- Feature engineering (polynomial):

Le **Feature Engineering** (ingénierie des caractéristiques) est un processus de transformation et de création de nouvelles caractéristiques à partir des données brutes pour améliorer la performance d'un modèle d'apprentissage automatique. Parmi ces caractéristiques:

1. **Amélioration des performances du modèle :**
 - Les modèles d'apprentissage automatique ne fonctionnent pas toujours bien avec les données brutes. En créant de nouvelles caractéristiques, on peut fournir au modèle plus d'informations utiles qui l'aideront à mieux apprendre et prédire.
2. **Capture des relations complexes :**
 - Les modèles linéaires peuvent avoir du mal à capturer des relations non linéaires entre les caractéristiques. Le **Feature Engineering** permet de créer de nouvelles variables qui capturent ces relations complexes.
3. **Réduction du bruit et simplification du modèle :**
 - En sélectionnant et créant des caractéristiques pertinentes, vous pouvez réduire la dimensionnalité et rendre le modèle plus simple et plus rapide, tout en maintenant sa précision.

Dans ce cas on a fait recours au polynomial features qui génère des **caractéristiques polynomiales** d'un degré spécifié à partir de caractéristiques d'entrée existantes. Par exemple, si on a deux caractéristiques **a** et **b**, un polynôme de degré 2 ajoutera des caractéristiques supplémentaires comme **a²**, **ab**, et **b²**. Dans notre cas, on a 91 polynômes croisés.

```
#feature engineering
from sklearn.preprocessing import PolynomialFeatures
# 2nd degree polynomialFeatures of [a, b] becomes [1, a, b, a^2, ab, b^2]
poly = PolynomialFeatures(2)
X_train_transformed = poly.fit_transform(x_train_trans)
X_test_transformed = poly.fit_transform(x_test_trans)

print(x_train_trans.shape)
print(X_train_transformed.shape)
```

```
(20631, 12)
(20631, 91)
```

SVR:

```
# SVR
from sklearn.svm import SVR

# Création du modèle
regressor = SVR(kernel='rbf')

# Entraînement
regressor.fit(x_train_trans, y_train)

# Prédiction et évaluation
y_hat_train = regressor.predict(x_train_trans)
RMSE_Train, R2_Train = evaluate(y_train, y_hat_train)

y_hat_test = regressor.predict(x_test_trans)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test)

test set RMSE:45.28091055080134, R2:0.5678320152955884
test set RMSE:28.043979046613842, R2:0.5445720042978831
```

Après avoir effectué l'amélioration de URL et le feature engineering comme le montre le code suivant :

```

# SVM + clipped RUL + engineered features
# SVR
from sklearn.svm import SVR

# Création du modèle
regressor = SVR(kernel='rbf')

# Entraînement
regressor.fit(X_train_transformed, y_train_clipped)

# Prédiction et évaluation
y_hat_train = regressor.predict(X_train_transformed)
RMSE_Train, R2_Train = evaluate(y_train_clipped, y_hat_train)

y_hat_test = regressor.predict(X_test_transformed)
RMSE_Test, R2_Test = evaluate(y_test, y_hat_test)

```

```

test set RMSE:22.22671798447902, R2:0.7155226909184534
test set RMSE:22.040010751014645, R2:0.7187036740325724

```

On constate une amélioration de RMSE allant de 28.04 à 22.04 soit 21.4%.

La **sélection des caractéristiques** est une étape cruciale dans le processus de modélisation des données, permettant d'identifier les variables les plus pertinentes pour prédire la cible. Dans le cadre de votre modèle, la méthode **SelectFromModel** est utilisée pour sélectionner automatiquement les caractéristiques les plus importantes. Cette sélection est basée sur l'évaluation de l'importance des caractéristiques par le modèle d'apprentissage automatique régresor.

```
#feature engineering + feature selection

from sklearn.feature_selection import SelectFromModel
select_features = SelectFromModel(regressor, threshold='mean', prefit=True)
select_features.get_support()
# Use get_feature_names_out() instead of get_feature_names() for scikit-learn >= 0.
try:
    feature_names = poly.get_feature_names_out() # For scikit-learn >= 0.24
except AttributeError:
    feature_names = poly.get_feature_names() # For older scikit-learn versions

# Use the original DataFrame variable name (x_train)
print('Original features:\n', x_train.columns)
print('Best features:\n', np.array(feature_names)[select_features.get_support()])
np.array(feature_names)[select_features.get_support()].shape

Original features:
Index(['s_2', 's_3', 's_4', 's_7', 's_8', 's_11', 's_12', 's_13', 's_15',
       's_17', 's_20', 's_21'],
      dtype='object')
Best features:
['x0' 'x1' 'x2' 'x3' 'x4' 'x5' 'x6' 'x7' 'x8' 'x9' 'x10' 'x11' 'x0 x11'
 'x2 x4' 'x3 x8' 'x4^2' 'x4 x6' 'x4 x7' 'x4 x9' 'x4 x11' 'x7^2' 'x7 x9']
(22,)
```

Le critère de sélection choisi est **"mean"**, ce qui signifie que seules les caractéristiques dont l'importance est supérieure à la moyenne des importances calculées par le modèle sont retenues. Cela permet de filtrer les variables qui n'apportent pas suffisamment d'information ou qui sont moins pertinentes pour la prédiction, réduisant ainsi la complexité du modèle et améliorant ses performances en évitant le sur-apprentissage (overfitting).

```
# SVM regression + clipped RUL + engineered features + selection
svr = SVR(kernel='rbf')
svr.fit(X_train_transformed[:, select_features.get_support()], y_train_clipped)

# predict and evaluate
y_hat_train = svr.predict(X_train_transformed[:, select_features.get_support()])
evaluate(y_train_clipped, y_hat_train, 'train')

y_hat_test = svr.predict(X_test_transformed[:, select_features.get_support()])
evaluate(y_test, y_hat_test)

train set RMSE:21.315733495249123, R2:0.7383639906120911
test set RMSE:21.491752051437246, R2:0.7325244440920975
(21.491752051437246, 0.7325244440920975)
```

On constate maintenant une amélioration de 22.04 à 21.49

C- Amélioration des paramètres du modèle

On peut même penser à une amélioration de paramètres associée à la tolérance, en effet, le paramètre **epsilon** dans la régression par vecteurs de support contrôle la largeur du "tube" où l'on tolère l'erreur. Une valeur d'**epsilon** plus grande signifie un tube plus large, et donc moins de contraintes sur les erreurs des prédictions.

Une **plage plus large** de valeurs, comme celle que vous avez choisie, permet de couvrir un éventail de comportements du modèle, allant d'une grande tolérance aux erreurs à une tolérance plus stricte.

Comme la valeur par défaut est de 0.1 on a essayé de fluctuer sur une marge suivante pour améliorer les résultats [0.4, 0.3, 0.2, 0.1, 0.05]

```
#Amélioration de SVR (epsilon)
epsilon = [0.4, 0.3, 0.2, 0.1, 0.05]

for e in epsilon:
    svr = SVR(kernel='rbf', epsilon=e)
    svr.fit(X_train_transformed[:, select_features.get_support()], y_train_clipped)

    # predict and evaluate
    y_hat = svr.predict(X_train_transformed[:, select_features.get_support()])
    mse = mean_squared_error(y_train_clipped, y_hat)
    rmse = np.sqrt(mse)
    variance = r2_score(y_train_clipped, y_hat)
    print("epsilon:", e, "RMSE:", rmse, "R2:", variance)
```

```
epsilon: 0.4 RMSE: 21.322980468778177 R2: 0.7381860571105527
epsilon: 0.3 RMSE: 21.318266407798358 R2: 0.7383018073890286
epsilon: 0.2 RMSE: 21.316963712317765 R2: 0.7383337895990437
epsilon: 0.1 RMSE: 21.315733495249123 R2: 0.7383639906120911
epsilon: 0.05 RMSE: 21.316025325010557 R2: 0.7383568265427067
```

Il s'avère que cela n'a rien apporté à notre solution donc on reste sur 0.1 on prévisant la plage e variation 0.01

```
#Amélioration de SVR (epsilon)
epsilon = [0.1, 0.11, 0.12, 0.13 ,0.14]

for e in epsilon:
    svr = SVR(kernel='rbf', epsilon=e)
    svr.fit(X_train_transformed[:, select_features.get_support()], y_train_clipped)

    # predict and evaluate
    y_hat = svr.predict(X_train_transformed[:, select_features.get_support()])
    mse = mean_squared_error(y_train_clipped, y_hat)
    rmse = np.sqrt(mse)
    variance = r2_score(y_train_clipped, y_hat)
    print("epsilon:", e, "RMSE:", rmse, "R2:", variance)
```

```
epsilon: 0.1 RMSE: 21.315733495249123 R2: 0.7383639906120911
epsilon: 0.11 RMSE: 21.315145832508858 R2: 0.7383784167274122
epsilon: 0.12 RMSE: 21.31418042781383 R2: 0.7384021148983828
epsilon: 0.13 RMSE: 21.313821329817117 R2: 0.7384109295450886
epsilon: 0.14 RMSE: 21.31456270824595 R2: 0.7383927310377454
```

Donc la meilleure solution est donnee par $\epsilon=0.13$

Le modèle final de régression par SVM (SVR) présente un RMSE de test de 21,313. La combinaison de la mise à jour de notre hypothèse sur le RUL et de l'ajustement d'un SVR avec des limites optimisées, le redimensionnement des caractéristiques et les caractéristiques polynomiales permet d'obtenir une amélioration par rapport à notre modèle de référence (RMSE = 28,04).

La majeure partie de cette amélioration est attribuée au changement de notre hypothèse sur le RUL, ce qui montre l'importance de bien définir le problème de science des données.

III- Recommandation pour les travaux futurs

Les avancées récentes dans le domaine, en particulier dans les architectures de réseaux neuronaux, ouvrent des perspectives prometteuses pour la recherche continue. Par exemple, explorer des architectures plus avancées telles que les réseaux neuronaux récurrents (RNN), les réseaux de mémoire à long terme (LSTM) et les réseaux neuronaux convolutifs (CNN) pourrait considérablement améliorer les capacités de prédiction de la durée de vie restante (RUL). Ces architectures se distinguent par leur aptitude à traiter des données séquentielles et des structures complexes.

En outre, la tendance émergente de l'apprentissage par transfert (Transfer Learning, TL) suscite un vif intérêt en raison de sa capacité à exploiter des modèles pré entraînés sur des ensembles de données volumineux et complexes.

L'intégration d'expertises spécifiques au domaine avec ces techniques avancées basées sur les données constitue également une voie prometteuse. Cette approche hybride pourrait associer les forces des connaissances traditionnelles aux paradigmes actuels d'apprentissage automatique, tirant ainsi parti des avantages des deux méthodologies.

En conclusion, la capacité de prédire avec précision la durée de vie restante des machines représente une intersection riche entre expertise de domaine, modélisation classique et technologies d'apprentissage automatique de pointe. Bien que les résultats actuels soient encourageants, ils mettent également en évidence un vaste champ d'opportunités pour des améliorations et des innovations futures. Le domaine de la gestion de la santé des systèmes progresse de manière dynamique, promettant des avancées majeures en matière de précision prédictive, de sécurité opérationnelle et d'efficacité des systèmes.

Conclusion

L'objectif principal de ce projet était d'évaluer les performances des modèles prédictifs basés sur les données pour estimer la durée de vie restante (RUL) des moteurs. Dans un contexte industriel où la sécurité, la fiabilité et l'optimisation des coûts de maintenance sont des enjeux majeurs, la capacité à prévoir le moment où un composant risque de défaillir a un impact considérable sur l'efficacité des opérations.

Pour résumer le travail précédent, voici le tableau des différents valeurs de RMSE et de R^2 :

Pour dataset FD001:

Méthode	RMSE_Test	R^2 _Test
Régression linéaire 1	33.3	0.35
Régression polynomiale	34.87	0.29
SVR	28.04	0.54
Decision Tree Regression	38.31	0.15
Random Forest	32.42	0.39
ANN	34.16	0.32
KNN	32.21	0.39
Régression linéaire 2	22.91	0.69
SVR 2 (e=0.13)	21.3138	0.738

Pour dataset FD002:

Méthode	RMSE_Test	R ² _Test
Régression linéaire 1	41.45	0.40
Régression polynomiale	31.23	0.66
SVR	38.17	0.49
Decision Tree Regression	34.88	0.57
Random Forest	33.59	0.6
ANN	33.01	0.62
KNN	31.62	0.65
Régression linéaire 2	38.91	0.47
SVR 2 (e=0.05)	29.5	0.5

Pour dataset FD003:

Méthode	RMSE_Test	R ² _Test
Régression linéaire	23.8	0.66
Régression polynomiale	20.72	0.74
SVR	22.55	0.7
Decision Tree Regression	23.92	0.66
Random Forest	22.14	0.71
ANN	22.06	0.71
KNN	22.72	0.69
SVR 2 (e=0.4)	17.46	0.81

Pour dataset FD004:

Méthode	RMSE_Test	R ² _Test
Régression linéaire 1	55.45	—
Régression polynomiale	32.48	0.64
SVR	46.96	0.25
Decision Tree Regression	34.29	0.6
Random Forest	39.98	0.46
ANN	36.08	0.56
KNN	32.77	0.63
SVR 2 (e=0.05)	36.45	0.46

Enseignements Clés du Projet:


- Adoption des approches basées sur les données :

Traditionnellement, les modèles prédictifs reposaient sur des connaissances spécifiques au domaine ou des modèles déterministes bien établis. Dans ce projet, nous avons privilégié des méthodes basées sur les données, notamment l'apprentissage automatique et les réseaux neuronaux artificiels. Ces approches se distinguent par leur capacité à établir des relations complexes à partir des données historiques sans dépendre de connaissances spécifiques au fonctionnement des moteurs.

- Importance du prétraitement des données :

Les données brutes utilisées dans ce projet étaient riches en informations, mais présentaient des variables redondantes et ne fournissaient pas directement les valeurs de RUL pour chaque cycle. Les étapes de prétraitement ont permis de transformer ces données en entrées structurées et pertinentes pour les modèles. Cela a impliqué :

1. La normalisation des capteurs.
2. La gestion des valeurs manquantes.



La sélection des capteurs les plus informatifs. Ces étapes ont été essentielles pour améliorer la qualité des données utilisées et, par conséquent, les performances des modèles.

- Apport de la visualisation des données :

La visualisation a joué un rôle crucial pour comprendre les distributions des capteurs, détecter les anomalies et identifier les capteurs les plus pertinents. En filtrant les capteurs peu informatifs, nous avons optimisé les données en entrée des modèles, ce qui a amélioré leurs performances.

- Comparaison des modèles prédictifs :

Plusieurs algorithmes ont été testés dans le cadre du projet, allant de modèles simples comme la régression linéaire à des méthodes avancées comme les réseaux neuronaux artificiels et les forêts aléatoires.

Chaque modèle a été évalué en termes de précision (RMSE, R^2) et de capacité à généraliser. Une stratégie clé a été l'application d'un clipping sur la valeur de RUL dans l'ensemble d'entraînement, ce qui a permis d'atténuer les problèmes de surestimation pour les valeurs élevées de RUL. Ce choix, inspiré par la visualisation des données, a significativement amélioré la précision des prédictions pour tous les modèles.

Liens des codes:

dataset1:[PROJET_FINAL_PRINCIPAL_1_FD001_.ipynb - Colab](#)

dataset2:[PROJET_FINAL_1_FD002.ipynb - Colab](#)

dataset3:[PROJET_FINAL_1_FD003.ipynb - Colab](#)

dataset4: [Projet_final_FD004.ipynb - Colab](#)