

Projet Final : MAINTENANCE PRÉdictive

Simplon.co - École Microsoft IA

Ngachili Maëlle

20/11/2020

Résumé

Ce rapport présente les étapes de la mise en place d'une application web qui affiche les prédictions de pannes faites en continu et en temps réel à partir de la simulation d'un flux de données télémétriques.

Table des matières

Table des matières	1
I Introduction	4
1 La Maintenance Industrielle	5
1.1 La Maintenance Traditionnelle	5
1.2 La Maintenance Prédictive	5
1.3 Applications	6
1.4 Outils pour la Maintenance Prédictive	7
II Recherche de la solution	8
2 Le Jeu De Données	9
2.1 Présentation	9
2.2 Dataset Machines	9
2.3 Dataset Maintenance	11
2.4 Dataset Errors	12
2.5 Dataset Failures	13
2.6 Dataset Telemetry	16

3 Feature Engineering	17
3.1 Telemetry	17
3.2 Errors	17
3.3 Maintenance	18
3.4 Finalisation	19
4 Algorithme de Classification	22
4.1 Modalités	22
4.2 Comparaison des modèles	23
4.3 Feature Importance	25
4.4 Ajustement de la Target	26
III Mise en Place de la Solution	27
5 Web Application	28
5.1 Flux de Données	28
5.2 Application	29
5.3 Base De Données	31
5.4 Interface Utilisateur	34
6 Bilan	35
6.1 Gestion de Projet	35
6.2 Version Control	36
6.3 Conclusion	36
6.4 Améliorations	36
Bibliographie	37
Appendices	38
A Fréquence des Interventions Sur Une Machine	39
B Distributions des Features	40
C Preprocessing Piepline - Code	42
C.1 Feature Pipeline (classe mère)	42
C.2 Telemetry Pipeline	43
C.3 Error Pipeline	44
C.4 Maintenance Pipeline	45
C.5 Feature Merger	46
D XGBoost - Bayesian Optimisation	49
E Temps Computationnel de l'Apprentissage et des Prédictions	50
F Matrices de Confusion	51
G Scores	56
H Feature Importance	59
H.1 Par Catégorie de Features	59

H.2	Par Sous-Catégorie de Features	60
I	Schéma Relationnel de la Base de Données	61
J	Application Django	63
J.1	Models	63
J.2	Views	68
J.3	Tests Unitaires	72
K	Structure de l'Application Web	78

Première partie

Introduction

Chapitre 1

La Maintenance Industrielle

1.1 LA MAINTENANCE TRADITIONNELLE

Jusqu'ici dans l'industrie on pratiquait essentiellement la maintenance curative qui consiste à attendre de se faire surprendre par une panne pour agir. Chaque panne ayant une incidence directe sur la production et donc en général un coût élevé surtout en cas d'interruption. Pour palier à ce problème les usines y ont associé la maintenance préventive. Il s'agit alors de programmer des interventions de maintenances pour contrôler et éventuellement remplacer régulièrement le matériel industriel. Ce qui peut être également coûteux, il faut constituer des équipes de maintenance ou incorporer ce type de contrôle dans la charge de travail des ouvriers. Les remplacements automatiques peuvent aussi s'avérer prématurés.

Une autre des problématiques récurrentes pour les industriels est l'excédent de stock. En effet, pour pouvoir intervenir le plus vite possible en cas de panne et éviter une interruption trop longue de la production, les usines sont dans l'obligation de conserver des stocks très conséquents en pièces et machines de remplacement. Cela a pour effet d'engendrer énormément de gaspillages et de déchets industriels qui sont un véritable frein pour toute entreprise.

La maintenance prédictive fait cette belle promesse de résoudre les principaux inconvénients liés aux méthodes de maintenance traditionnelles dans l'industrie.

1.2 LA MAINTENANCE PRÉDICTIVE

Présentation

Les économistes s'accordent à dire que la maintenance prédictive a un avenir radieux devant elle. Selon une étude du cabinet McKinsey d'ici 2025 elle devrait permettre aux entreprises américaines d'économiser 630 milliards de dollars.

Selon [Fero Labs](#) les entreprises industrielles se débarrassent de 98% des données qu'elles collectent par manque de connaissances, de compétences ou de capacité pour les exploiter. Or en exploitant certaines de ces données correctement, ces entreprises pourraient efficacement prévenir toute sortes de pannes sans avoir à supporter le coût d'interventions préprogrammées parfois inutiles.

Au cours de ces dernières années, cette forme de maintenance industrielle a gagné en popularité puisque les équipes ont rapidement compris qu'elles pourraient, grâce à elle, réduire les pannes, éviter les imprévus et anticiper le moindre arrêt dans la production.

Plus concrètement, elle va permettre de gagner en fiabilité et de surveiller le plus précisément possible les performances des machines. Les problèmes imminents sont détectés et résolus, si bien que les équipes en charge peuvent procéder aux réparations et remplacements avant que les pannes ne surviennent ou n'entraînent de problèmes plus graves ou coûteux.

Pré-requis

Les mesures récupérées par tous types de capteurs s'avèrent être une ressource en or pour les algorithmes de machine learning. Avec l'essor de l'IoT (Internet of things) puis de l'IIoT (Industrial Internet of Things), les entreprises industrielles ont de plus en plus de possibilité pour accéder à toute sorte de données télémétriques sur leurs machines et leur environnement.

Les capteurs sur les machines fonctionnent comme un système de surveillance des outils de production en temps réel. Ils transmettent les données à un logiciel pour qu'un technicien de maintenance les analyse et ainsi il peut :

- déterminer la probabilité d'un défaut sur une machine,
- déterminer le type de défaut possible,
- anticiper une panne,
- prévoir l'entretien nécessaire à effectuer sur une machine pour éviter la panne et ne pas bloquer la production.

Avant de se lancer dans la maintenance prédictive il faut donc au préalable s'assurer de disposer des divers capteurs qui viendront alimenter une base de données. Ensuite il faut avoir un historique suffisamment grand de ces données pour pouvoir entraîner les modèles.

Avantages

Par rapport à la maintenance préventive, la maintenance prédictive permet de :

- diminuer le nombre d'interruptions des machines pour des opérations de maintenance,
- diminuer le nombre de pannes,
- mieux planifier les interventions,
- mieux préparer les équipes d'intervention,
- mieux échanger entre les professionnels de maintenance et les équipes de production,
- mieux anticiper et gérer les besoins de pièces détachées des outils.

Pour résumé, une maintenance prédictive permet d'économiser, entre autres, du temps, de la main d'œuvre ou encore de l'argent, et permet aussi d'identifier des problèmes qui n'aurait pu être résolus avec de simples inspections de routine.

Limites

Il est communément admis que la maintenance prédictive peut coûter très cher. Les procédés à mettre en place qu'elle implique sont assez onéreux, mais ces coûts sont très rapidement amortis par les résultats obtenus. Les industriels peuvent en effet réaliser d'importantes économies, puisqu'ils seront en mesure de prévoir les pannes et d'agir en amont.

Cependant, si l'usine ne recense pas beaucoup de machines, il est préférable de s'en tenir à des routines de maintenance préventive conventionnelle. C'est généralement plus économique.

1.3 APPLICATIONS

Le premier exemple d'application de la maintenance prédictive est assez intuitif, l'industrie. Les usines sont a priori toutes de bonne candidate pour ce système.

Les applications sont cependant bien plus vastes et variées. Théoriquement tout objet disposant de capteurs devrait pouvoir profiter de ce qu'offre la maintenance prédictive. On pourrait renforcer les signaux d'alertes des tableaux de bords afin qu'ils mettent en garde contre une défaillance à venir. Libre aux conducteurs d'ignorer également ses signaux. Les smartphones, ordinateurs, tablettes sont d'autres exemples. Il serait extrêmement pratique de pouvoir utiliser la maintenance prédictive sur les disques durs.

D'un point de vue algorithmique la maintenance prédictive se rapproche assez de la détection d'anomalies. Bien souvent - et idéalement - les pannes sont des évènements rares, cela revient donc bien à rechercher la ou les anomalies dans les logs de maintenance et les données de télé-métrie ayant pu causer une panne.

1.4 OUTILS POUR LA MAINTENANCE PRÉDICTIVE

Il existe déjà sur le marché plusieurs solutions de maintenance prédictive pour les industriels. Elles consistent le plus souvent à proposer un outil connecté qui combine la GMAO (gestion de maintenance assistée par ordinateur) et l'analyse des données.

Voici quelques exemples de ces outils :

- [InUse](#)
- [MAINTI4](#)
- [Mobility Work](#)
- [XMaint](#)

La plupart de ces solutions offrent un service en ligne avec des versions web ou application mobile.

Deuxième partie

Recherche et Expérimentation

Chapitre 2

Le Jeu De Données

2.1 PRÉSENTATION

Choix du Dataset

Sans grande surprise, les entreprises communiquent peu sur leurs problèmes de production et leurs pannes, elles partagent donc difficilement leurs données de maintenance. Il est impossible de trouver un dataset de logs de maintenance et de données télémétriques industrielles réelles publique. C'est pourquoi le jeu de données utilisé sur ce projet est fictif. Microsoft Azure met à disposition un ensemble de cinq datasets [1] pour la maintenance prédictive, faits à partir de données simulées. Le choix s'est arrêté sur cet ensemble pour ce projet car c'est le plus volumineux qui ait pu être récupéré (environ 5Go de données). Il y a une année de données historisées.

Vue d'Ensemble

Machines	Telemetry	Maintenance	Errors	Failures
Informations sur les 1000 machines 1 000 x 3 MachineID Age Model [1:5]	Moyenne horaire des mesures 8 761 000 x 6 Datetime MachineID Volt Rotation Pressure Vibration	Logs: interventions programmées ou non > Sur 1.5 an 32 592 x 6 Datetime MachineID Comp [1:4]	Log : erreurs sans interruption (pas de panne) 11 967 x 3 Datetime MachineID ErrorID [1:5]	Logs: remplacements de composant (panne) 6 726 x 3 Datetime MachineID Failure {complID}

FIGURE 2.1 – Le Jeu de Données

2.2 DATASET MACHINES

Ce dataset contient les informations sur chacune des 1000 machines constituant le parc industriel fictif. Il donne trois informations :

1. l'ID de la machine
2. l'âge de la machine
3. Le modèle de la machine

machineID	model	age
0	1	model2
1	2	model4
2	3	model3
3	4	model3
4	5	model2

Modèle des Machines

Il y a cinq modèle de machine numérotés de 1 à 5. Les modèle 3 et 4 représentent à eux deux près de 80% du parc. Le modèle de machine le moins répandu et le modèle 1 avec seulement 8.5% des machines.

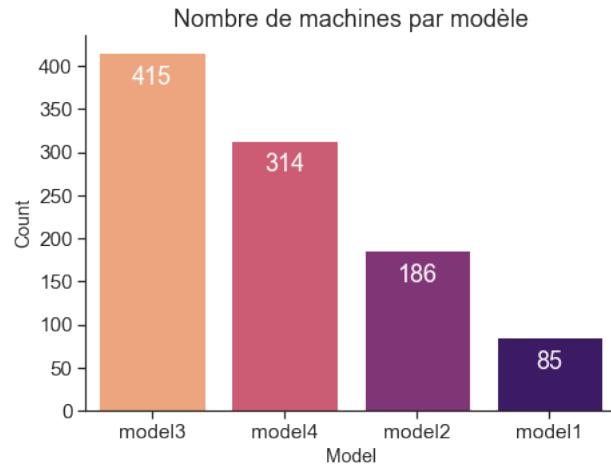
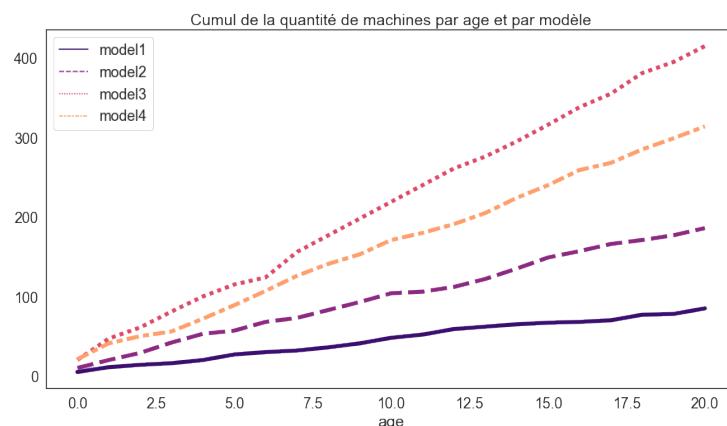


FIGURE 2.2 – Répartition des Modèles

Age des Machine

Les machines ont un âge compris entre 0 et 20 ans. Le nombre et la répartition des machine est globalement stable en fonction des âges. On peut en déduire que la construction du parc s'est faite de manière linéaire comme le montre la figure 2.3

FIGURE 2.3 – Évolution du Parc de Machines



Le nombre indiqué sur chacune des portions des barres de la figure 2.2 représente le pourcentage de machines pour un modèle et un âge donné.

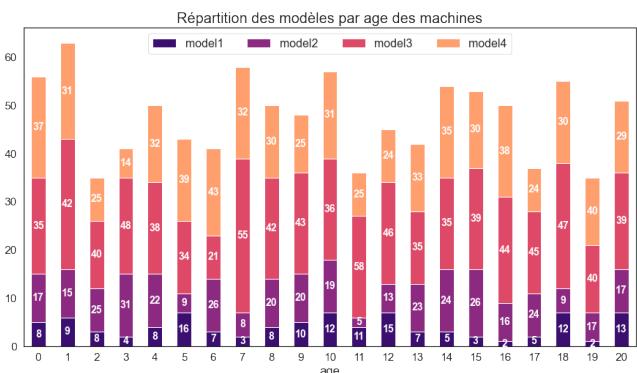


FIGURE 2.4 – Répartition des modèles par âges

2.3 DATASET MAINTENANCE

Il contient les logs de maintenance sur un an et demi, du 1er juin 2014 à 6h00 au 1er janvier 2016 à 6h00. Chaque entrée correspond au remplacement d'un composant sur une machine pour un jour donné. Il peut s'agir d'un remplacement programmé ou non. Si l'intervention de maintenance s'est faite suite à une panne, elle est enregistrée à la fois de ce dataset et dans le dataset `failure`. Il contient six mois de données de plus que les autres logs car on (i.e. MS Azure) considère que ces logs permettent d'inférer sur le cycle de vie des composants.

Les logs sont enregistrés à 6h00. Il y a 32 592 entrées.

Chacune d'elles donne les informations suivantes :

- l'ID de la machine concernée
- la date et l'heure du contrôle
- Le composant remplacé

	datetime	machineID	comp
0	2014-07-01 06:00:00	1	comp4
1	2014-09-14 06:00:00	1	comp1
2	2014-09-14 06:00:00	1	comp2
32590	2015-12-26 06:00:00	1000	comp3
32591	2015-12-26 06:00:00	1000	comp1

FIGURE 2.5 – Maintenance

Composants

Il y a en tout quatre types de composant, ils sont numérotés de 1 à 4. Chaque machine possède les 4 types de composants quel que soit son modèle.

```
1 df_maint.groupby('machineID')['comp'].nunique().unique()
2 >> array([4])
```

Fréquence des logs

En analysant la fréquence des entrées (figures 2.7a et 2.6) on voit que pour le deuxième semestre 2014 les interventions de maintenance sont programmées une fois toutes les deux semaines et environ 25% des machines sont inspectées à chaque fois. Tandis que pour l'année 2015, les interventions sont quotidiennes mais seulement 10% du parc environ est contrôlé chaque jour.

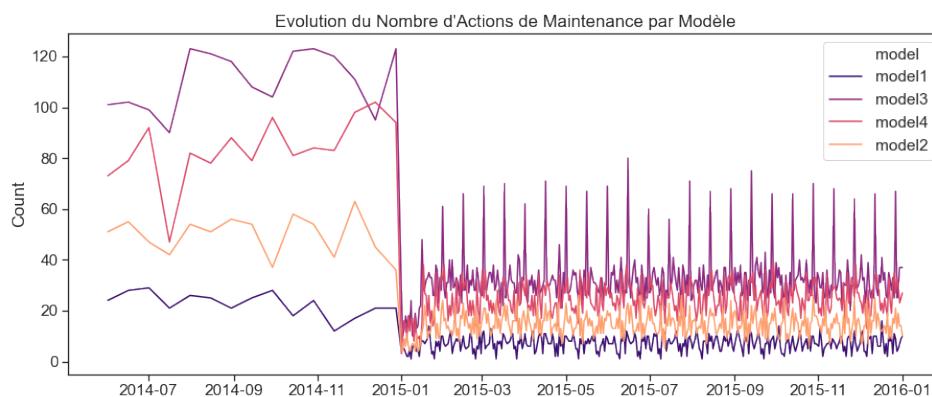
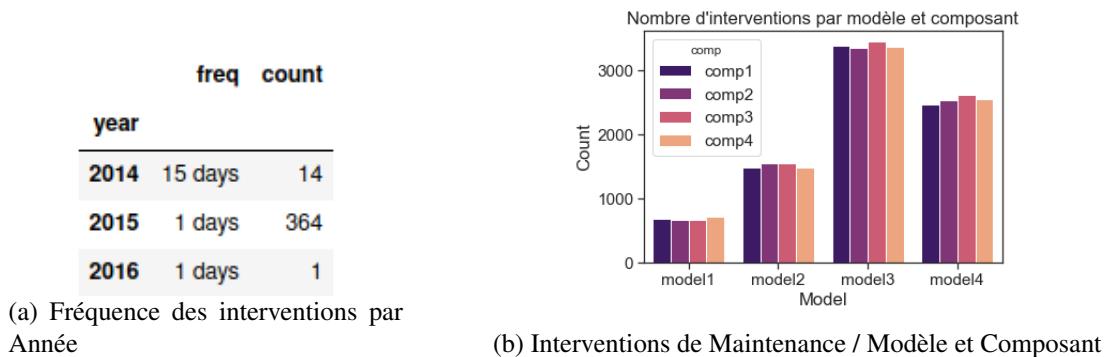


FIGURE 2.6 – Interventions de Maintenance par Modèle

Le choix des machines contrôlées est fait de telle sorte que chaque modèle a la même proportion de ses machines inspectées.

Tous les composants des machines ne sont pas remplacés à chaque intervention mais pour chaque machine, chacun des composants est remplacé en moyenne deux fois par mois (cf. annexe A).

FIGURE 2.7 – Visualisation des logs de maintenance



2.4 DATASET ERRORS

Ce dataset contient les logs pour l'année 2015 des erreurs qui ne causent aucune interruption dans la production et qui ne nécessite pas de remplacer un composant.

Il y a 11 967 lignes. Chaque entrée donne :

- l'heure et la date de l'erreur
- l'ID de la machine concernée
- l'ID du type d'erreur

	datetime	machineID	errorID
0	2015-01-06 03:00:00	1	error3
1	2015-02-03 06:00:00	1	error4
2	2015-02-21 11:00:00	1	error1
11965	2015-09-11 06:00:00	1000	error1
11966	2015-10-11 14:00:00	1000	error3

FIGURE 2.8 – Errors

Les Types d'Erreurs

Il y a cinq types d'erreurs. Les types 1, 2 et 3 représentent chacun environ un quart des erreurs. Le type 5 est le moins courant mais la figure 2.9 montre qu'elle n'occurre que pour les machines de 14 ans ou plus.

Bien qu'elle soit présente sur tous les modèles, la proportion de chaque modèle diffère de leurs proportions globales sur l'erreur de type 4. Elle est en effet plus courante sur les modèles 1 et 2 (cf. figure 2.10b). Pour ces deux modèles, elle représente environ 30% des erreurs. Les erreurs de type 1, 2 et 3 font chacune environ 20% des erreurs pour les modèles 1 et 2.

En revanche, l'erreur de type 4 est très rares sur les machines des modèles 3 et 4. Pour ces modèles ce sont les erreurs de type 1, 2 et 3 qui surviennent le plus souvent, avec en moyenne 24% à 29% d'occurrences chacune.

Les chiffres sur chacune des barres de la figure 2.10b représentent le pourcentage d'occurrence de l'erreur de donnée pour chacun des modèles.

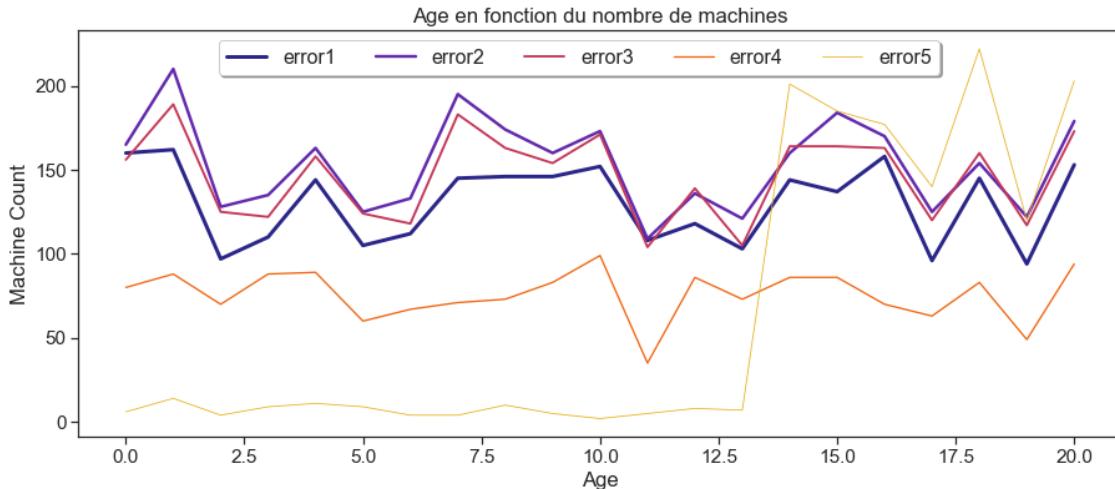
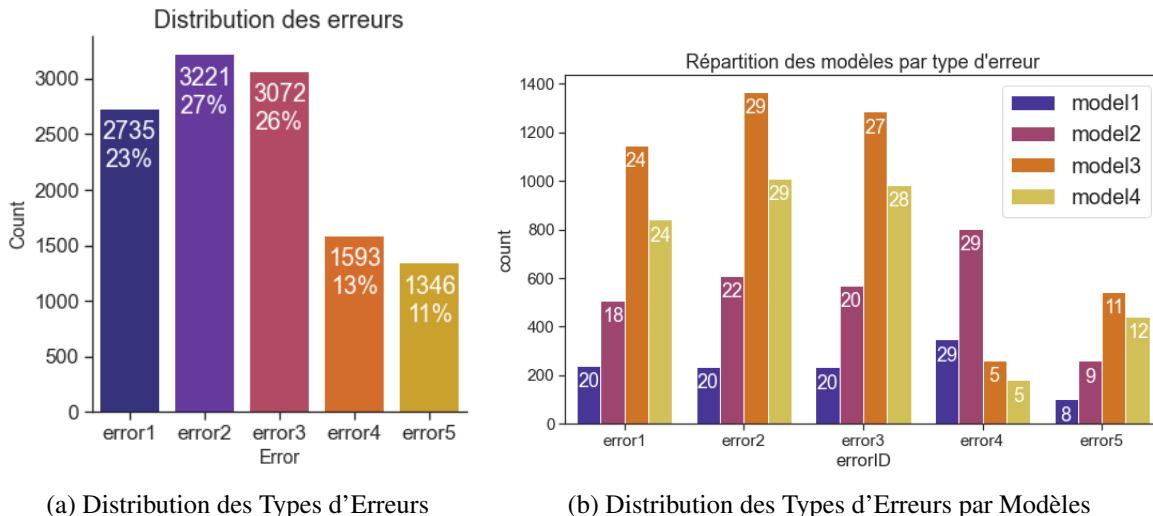


FIGURE 2.9 – Nombre de logs d’erreurs en fonction des Ages des Machines

FIGURE 2.10 – Maintenance - Distribution



2.5 DATASET FAILURES

Ce dataset contient les logs des pannes pour l’année 2015. On considère qu’il y a une panne dès lors qu’un composant doit être remplacé. Les pannes sont des erreurs qui impliquent un arrêt de la machine.

Il y a 6 726 lignes. Les pannes sont toujours enregistrées à 6h00. Pour une machine donnée et une date donnée (heure et jour) on peut donc avoir plusieurs entrées, c’est le type du composant qui différera.

Chaque log donne les informations suivantes :

- l’heure et la date de la panne
- l’ID de la machine concernée
- l’ID du composant remplacé

	datetime	machineID	failure
0	2015-02-04 06:00:00	1	comp3
1	2015-03-21 06:00:00	1	comp1
2	2015-04-05 06:00:00	1	comp4
6724	2015-08-13 06:00:00	1000	comp2
6725	2015-09-12 06:00:00	1000	comp1

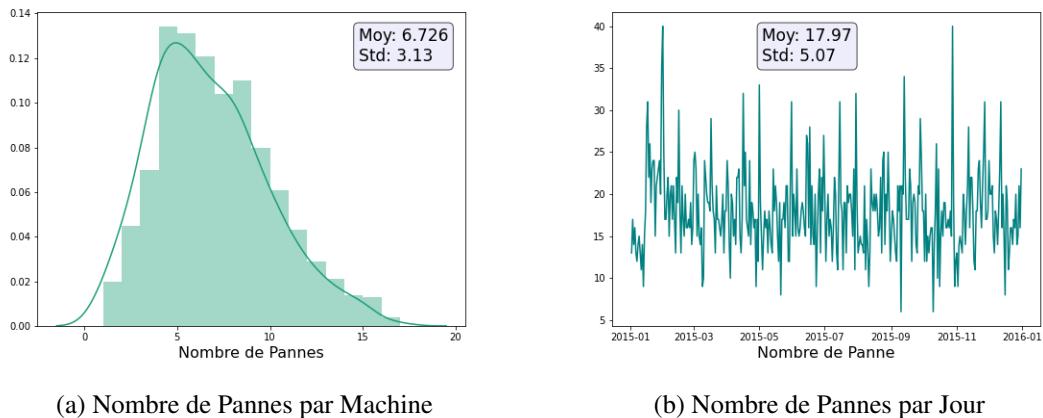
FIGURE 2.11 – Failures

Fréquence des Pannes

Il y a plus de 6 000 pannes enregistrées sur un an pour 1000 machines, on peut facilement supposer que chaque machine connaît en moyenne six pannes par an. La figure 2.12a confirme cela.

Ça peut sembler peu mais avec 1000 machines dans le parc industriel cela veut dire qu'il y a en moyenne 18 pannes par jour (figure 2.12b).

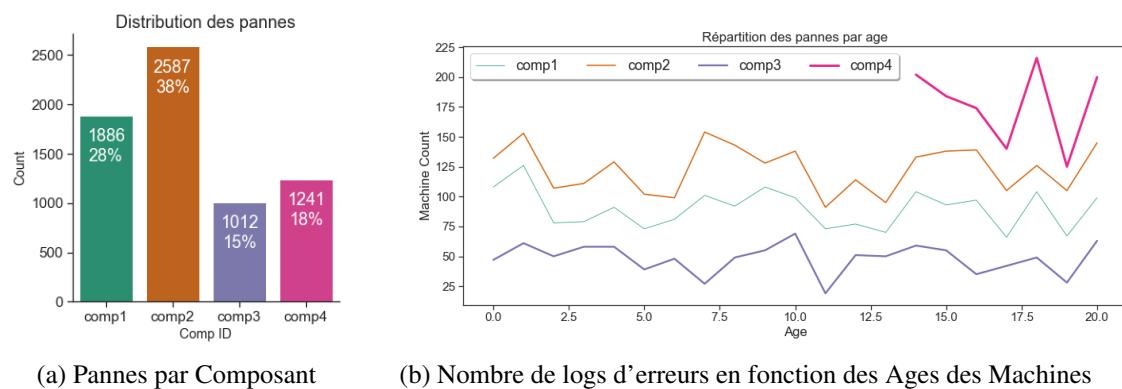
FIGURE 2.12 – Failures - Distribution



Répartition des Pannes

Les composants 1 et 2 représentent à eux deux les deux tiers des pannes, avec respectivement 28% et 38% des pannes chacun. Le composant 3 génère le moins de pannes avec 15% d'occurrence.

FIGURE 2.13 – Failures



Panne par Age

Là encore, il existe un type de composant qui ne crée des pannes que pour les machines de 14ans ou plus, il s'agit du composant 4. On peut déjà supposer que l'erreur de type 5 est liée au composant 4. Les trois autres composants sont globalement répartis de la même façon pour tous les âges (figure 2.13b)).

Pannes et Type d'Erreurs

En rapprochant les logs d'erreurs aux pannes on a pu lier dans la plupart des cas le type d'erreur associée à la panne. Puisque les erreurs sont enregistrées au moment précis où elles occurrent mais les pannes, tout comme pour le logs de maintenance, sont enregistrées une fois par jour à 6h00, il a fallu trouver pour chaque panne si une erreur avait été enregistré pour la machine concernée dans les dernières 12 heures. Il existe 182 pannes (soit 2.7% des pannes), pour lesquelles une erreur n'a pu être identifiée d'après les logs.

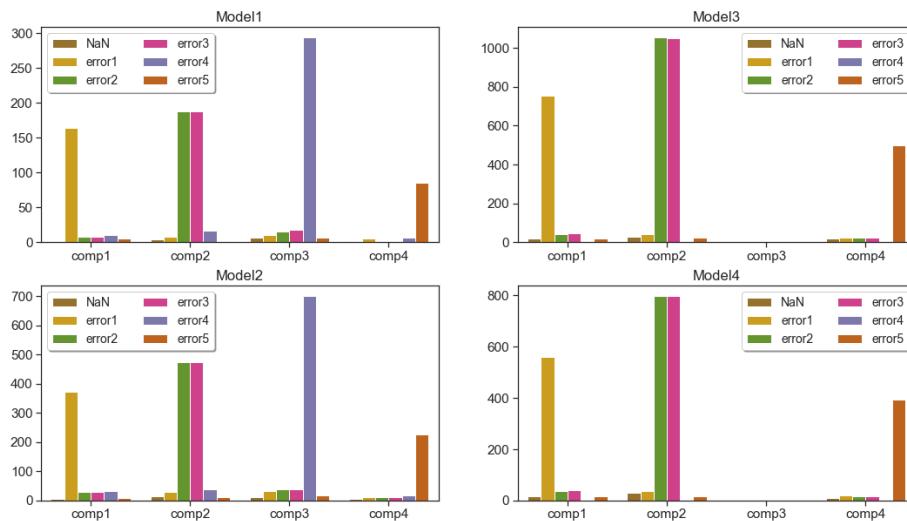


FIGURE 2.14 – Répartition des Pannes en fonction des Composants et des Erreurs pour chaque modèle

La figure 2.14 nous permet de constater les types d'erreurs et les ID des composants causant des pannes sont corrélés. Voici ce qu'on peut observer :

- Le composant 1 génère presque toujours une erreur de type 1
- Le composant 2 génère presque toujours des erreurs de type 2 et 3 dans des proportions égales.
- Le composant 3 génère presque toujours une erreur de type 4
- Le composant 4 génère presque toujours une erreur de type 5
- Les modèles 1 et 2 et les modèles 3 et 4 forment des groupes.

2.6 DATASET TELEMETRY

C'est le dataset le plus large avec plus de 5Go de données et 8,7 millions de lignes. La moyenne des quatre types de mesures envoyées par les capteurs est calculée toutes les heures pour chaque machine sur toute l'année 2015.

Il y a 8 761 000 lignes. Chaque log donne les informations suivantes :

- l'heure et la date
- l'ID de la machine
- la tension ('volt')
- la rotation
- la pression ('pressure')
- la vibrations

	datetime	machineID	volt	rotate	pressure	vibration
0	2015-01-01 06:00:00	1	151.919999	530.813578	101.788175	49.604013
1	2015-01-01 07:00:00	1	174.522001	535.523532	113.256009	41.515905
2	2015-01-01 08:00:00	1	146.912822	456.080746	107.786965	42.099694
8760998	2016-01-01 05:00:00	1000	160.007424	462.740287	108.397268	47.206940
8760999	2016-01-01 06:00:00	1000	164.590354	415.431358	102.142896	37.919958

FIGURE 2.15 – Failures

L'étude en série temporelle n'apporte aucune information de plus. Quel que soit la mesure, il n'y a pas de tendance, la légère périodicité observée toutes les deux semaines est sûrement liée à la façon dont ces données fictives ont été générées.

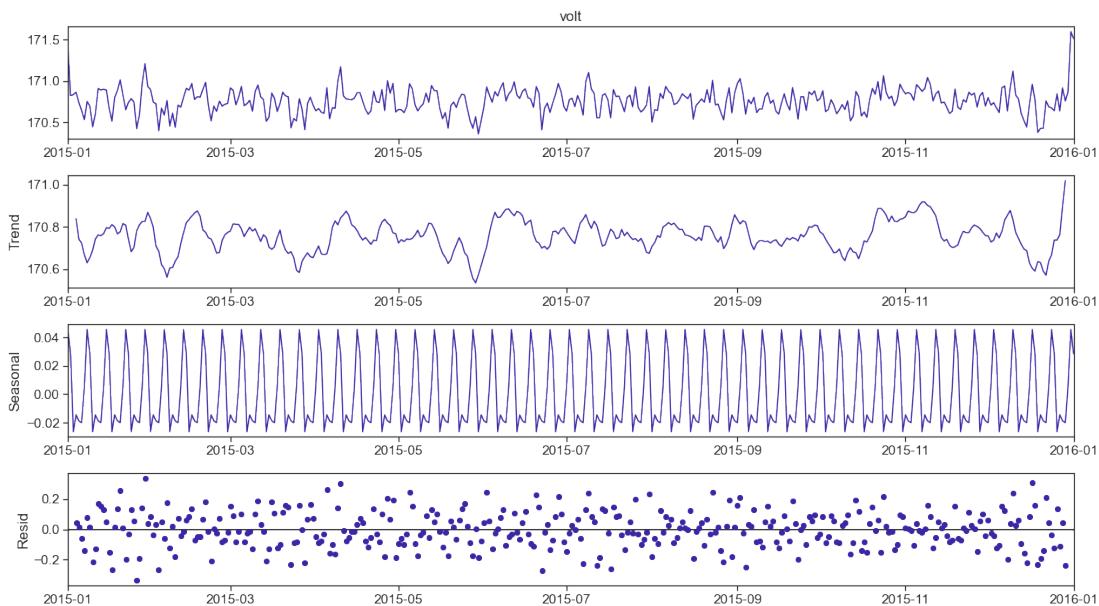


FIGURE 2.16 – Décomposition de la série temporelle Volt

Dans la partie suivant nous verrons comment extraire des ces datasets des features exploitables pour les algorithmes de machine learning.

Chapitre 3

Feature Engineering

3.1 TELEMETRY

Pour travailler avec ce dataset, chaque mesure est décomposée en moyennes mobiles et écart-types mobiles. Dans un premier temps, les moyennes et écarts-types sont construits sur des fenêtres de 6, 12, 24 et 36 heures. Ils sont bien entendu calculés par machine et exclusivement à partir des données antérieures.

Ainsi, chacune des quatre mesures est remplacée par huit nouvelles colonnes. Ensuite les données sont regroupées par machine et par intervalle de 12 heures en faisant une moyenne sur l'intervalle (voir en annexe C.2 pour le code).

Finalement, on obtient un dataset de 731 000 lignes et 34 colonnes.

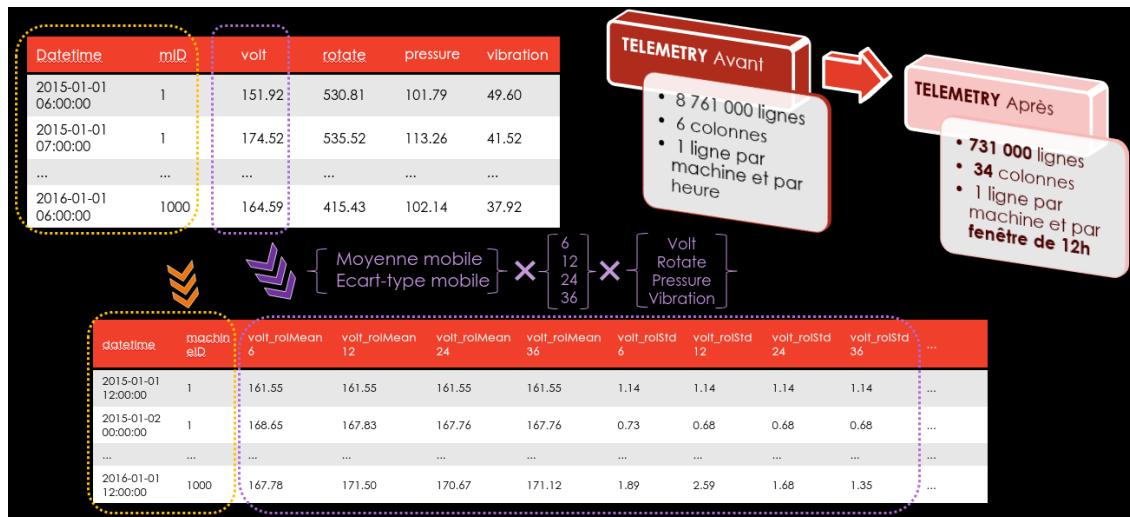


FIGURE 3.1 – Pre-processing de Telemetry (avec exemple)

3.2 ERRORS

On commence par *dummifier* la colonne `errorID` pour avoir une colonne booléenne par type d'erreur. Puis une somme mobile est faite pour chaque erreur pour des fenêtres de 6, 12, 24 et 36 heures.

Pour pouvoir aligner ces nouvelles features avec les features de télémétrie, elles sont aussi regroupées par machine et par intervalle de 12 heures en faisant une moyenne (voir en annexe C.3 pour le code).

On obtient un dataset de 731 000 lignes et 22 colonnes.

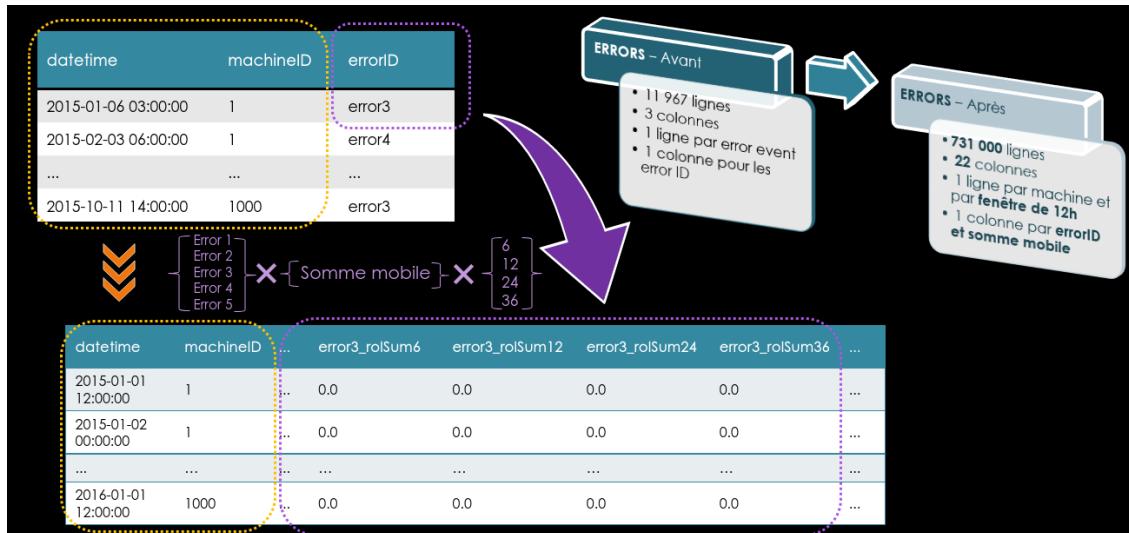


FIGURE 3.2 – Pre-processing de Errors (avec exemple)

3.3 MAINTENANCE

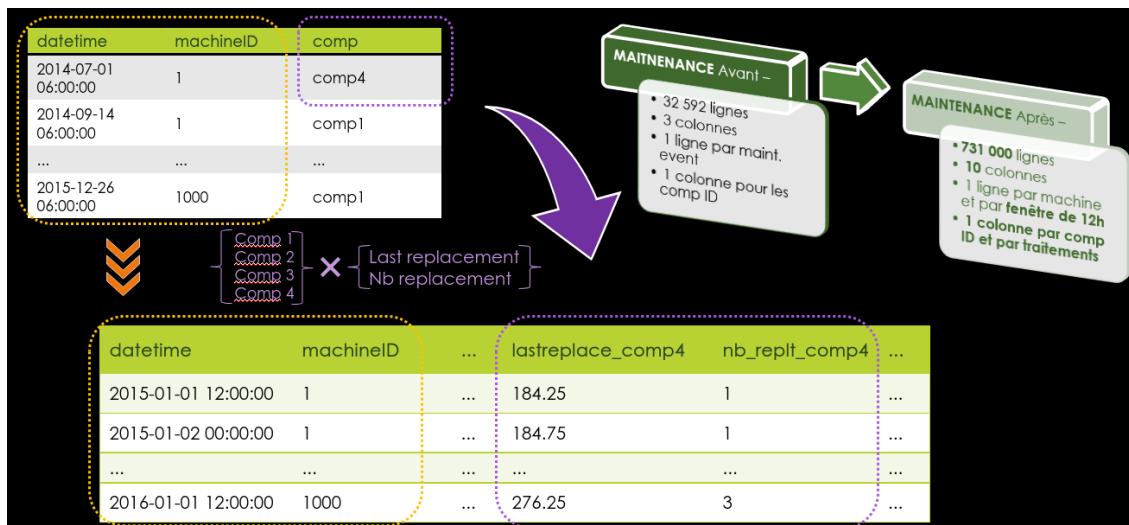


FIGURE 3.3 – Pre-processing de Maintenance (avec exemple)

Ici aussi la première étape consiste à *dummifier* la colonne `comp` pour avoir une colonne booléenne par composant.

Ensuite on construit deux nouvelles colonnes par composant :

- La première comptabilise le nombre de jours depuis le dernier contrôle
- la deuxième comptabilise le nombre de contrôles effectué depuis le début

De même les données sont regroupées par machine et par intervalle de 12 heures en faisant une moyenne (voir en annexe C.4 pour le code).

Le dataset de sortie contient 731 000 lignes et 10 colonnes

3.4 FINALISATION

Preprocessing Pipeline

Les colonnes `datetime` et `machineID` sont utilisées comme index pour chacune des trois transformations décrites précédemment.

Un *One Hot Encoder* est utilisé sur la colonne `model` du dataset `machines` et la colonne `age` est laissée telle quelle.

Les résultats des transformations sont enfin joints ensemble. La pipeline finale de preprocessing contient les étapes suivantes :

- *Transformer* pour le dataset `telemetry`
- *Transformer* pour le dataset `errors`
- *Transformer* pour le dataset `maintenance`
- *Transformer* pour le dataset `maintenance`
- *One Hot Encoder* pour le dataset `machines`
- *Merger* pour les 4 transformations

Standardisation

Les données sont standardisées avec un *StandardScaler*. Les features suivent en grande majorité une distribution gaussienne (cf. annexe B) et ce type de standardisation permet de ne pas "effacer" les outliers, ce qui est primordial ici car ce sont sûrement les outliers qui permettront de détecter les pannes.

Programmation Orientée Objet

Pour chacune des trois transformations majeures (`telemetry`, `errors`, `maintenance`) une classe qui hérite du *TransformerMixin* de *Scikit-Learn* a été créée de sorte à avoir de véritables *transformer* personnalisés.

Chaque classe contient au moins une méthode *fit* et une méthode *transform*. Elles héritent toutes d'une classe mère qui fournit entre autres les méthodes pour agréger les données par intervalle de 12h et pour faire les moyennes, écarts-types ou sommes mobiles.

Historique

La particularité de ces features est que chaque échantillon est lié à un certain nombre des échantillons qui le précèdent directement, en fonction des fenêtres choisies pour les moyennes, écarts-types et sommes mobiles.

C'est pourquoi l'une des méthodes les plus importantes est celle qui permet de garder un historique des données brutes pour pouvoir avoir une continuité sur les moyennes (ou écarts-types ou sommes) mobiles lorsque la méthode *transform* est appelée sur de nouvelles données (c'est-à-dire différentes de celles utilisées sur le *fit*). Le strict minimum est historisé pour ne pas alourdir inutilement la pipeline finale. On se base donc sur la plus grande des fenêtres utilisées pour les moyennes (ou écarts-types ou sommes) mobiles pour construire l'historique. Par exemple, à

chaque fois que la méthode *transform* est appelé, les 35 entrées les plus récentes (donc les 35 dernières si elles sont classées par ordre croissant) sont conservées pour chaque machine.

Ainsi, que tout le jeu de données soit transformé directement ou que ça soit fait intervalle par intervalle, les résultats des fonctions d'agrégation mobiles sont bien les mêmes.

Création des labels

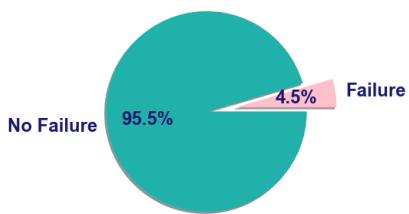
Le dataset *failures* va servir à créer les labels. La *target* consiste dans un premier temps en une colonne qui donne la liste des composants qui vont générer une panne dans les K_{period} prochains jours ou "no failure" si aucune panne n'est prévue. K_{period} correspond en réalité à un nombre d'intervalles de temps. Puisque nos données sont regroupées par intervalles de 12h, si K_{period} est égal à 4 cela signifie que les labels donnent les risques de pannes pour les deux jours (48 heures complètes) précédents la panne.

Cette colonne est ensuite transformée avec un *MultiLabelBinarizer* afin de pouvoir être comprise comme étant multi-label par les modèles. Finalement la *target* finale est composé de cinq colonnes booléennes.

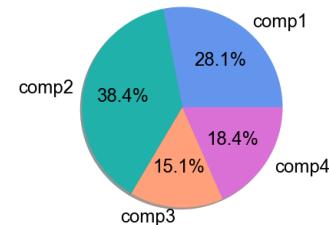
machineID	datetime	comp1	comp2	comp3	comp4	no failure
		0	0	0	1	
890	2015-01-01 12:00:00	0	0	0	1	0
	2015-01-02 00:00:00	0	0	0	1	0
	2015-01-02 12:00:00	1	0	0	1	0
	2015-01-03 00:00:00	1	0	0	0	0
	2015-01-03 12:00:00	1	0	0	0	0
	2015-01-04 00:00:00	1	0	0	0	0
	2015-01-04 12:00:00	1	0	0	0	0
	2015-01-05 00:00:00	0	0	0	0	1
	2015-01-05 12:00:00	0	0	0	0	1
	2015-01-06 00:00:00	0	0	0	0	1

FIGURE 3.4 – Cible Finale (y) pour $K_{period} = 4$

FIGURE 3.5 – Target - Distribution



(a) Proportions "Failure" vs "No Failure"



(b) Proportions des Composants si "Failure"

Voici le schéma récapitulatif de la pipeline de feature engineering qui sera utilisée :

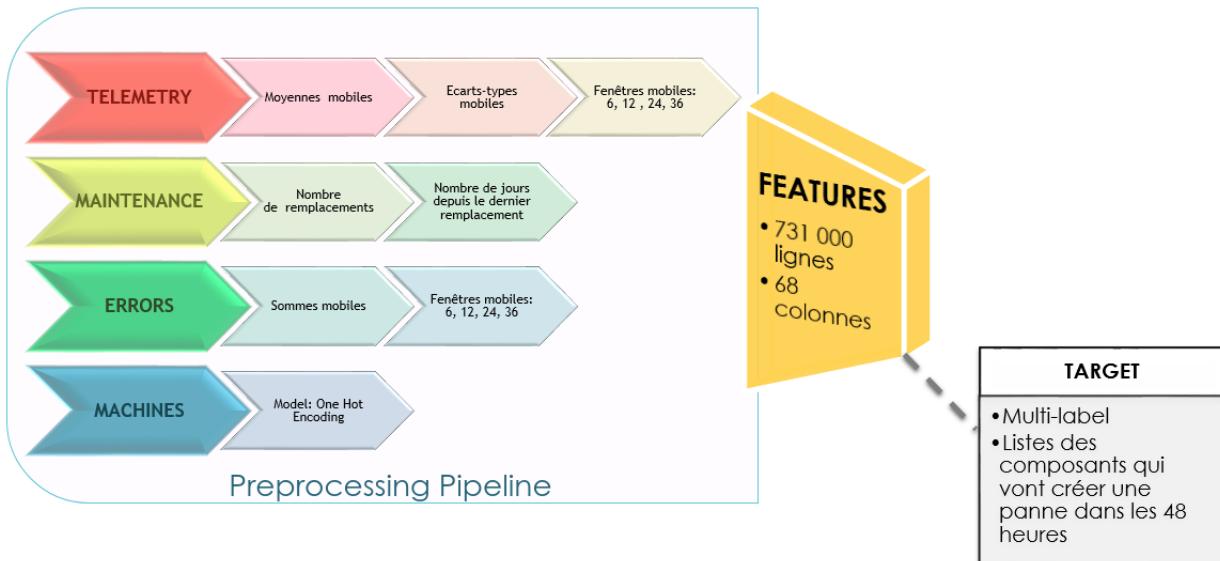


FIGURE 3.6 – Pre-processing Pipeline + Labels

On verra dans le prochain chapitre que notre meilleur modèle ne fera pas mieux que 2 jours d’anticipation pour les pannes.

Chapitre 4

Algorithme de Classification

4.1 MODALITÉS

Multi-Label

Encore une grande majorité des classificateurs disponibles dans la librairie *scikit-learn* ne gère pas les cas multi-labels. Parmi les modèles mis en concurrence ici, les modèles de régression logistique, Naive Bayes et SVM ne supportent pas les cas multi-label. La librairie offre tout de même la possibilité de travailler avec ces classificateurs sur ce type de cas en les combinant à la classe [OneVsRest](#). Cette classe permet d'entraîner un estimateur par label de sorte à ce que chaque estimateur soit une classification binaire où la target vaut 1 si le label en question vaut 1 et 0 sinon.

Ajustement des Prédictions

Dans certains cas, le label `no_failure` vaut 1 lorsque l'un des autres labels vaut aussi 1. Or il faut que ce label soit mutuellement-exclusif avec tous les autres. Puisque dans un premier temps il est d'abord essentiel de bien identifier chaque panne, chaque prédiction est ajustée afin que le label `no_failure` soit égal à 0 si au moins un des autres labels vaut 1. De même il existe de rares cas où tous les labels valent 0. Dans ce cas la valeur du label `no_failure` est remplacé par 1.

Hyperparamètres

Par défaut une *Random Search* a été utilisée pour trouver les meilleurs hyperparamètres de chaque modèle. Pour le XGBM (librairie XGBoost) c'est une méthode d'optimisation bayésienne [2] [3] avec la librairie [bayesian-optimisation](#) qui a été utilisée car elle est plus rapide. Cette méthode d'optimisation requiert moins d'itérations pour des résultats plus ou moins équivalents.

En effet, Les modèles ensemblistes nécessitent un temps d'entraînement plus long en général puisqu'un certain nombre de *learners* (petits estimateurs) sont construits pour chaque modèle. Dans ce cas précis, il peut être intéressant d'utiliser une optimisation bayésienne pour deux raisons :

1. Large volume de donnée : 731 000 échantillons pour 68 features
2. Target multi-label : Un estimateur doit être créé pour chaque label. Dans le cas présent c'est donc cinq estimateurs qui sont entraînés par modèle.

Même en utilisant la parallélisation les temps de *fit* sont assez longs sur ce type de modèle (environ 25-30 min heure pour XGBoost). Une Random Search avec seulement 50 itérations et $k = 3$ pour la cross-validation prendrait environ 3 jours.

La meilleure combinaison d'hyper-paramètres a été obtenue au bout de six itérations seulement pour XGBoost (cf. annexe D).

Un résumé des temps computationnel pour l'apprentissage et les prédictions est disponible en annexe E

Choix des Scores

La cible est extrêmement déséquilibrée. La classe no failure contient 95.5% des échantillons voir figure 3.5a. C'est pourquoi l'accuracy et le ROC-AUC ne sont donnés qu'à titre informatif. Ces deux mesures sont influencées par la classe prédominante. Si le modèle se contentait de classer tous les échantillons dans la classe no failure il ferait des déjà 95% en accuracy et en ROC-AUC.

Un bon exemple en est donné avec le modèle de Naive Bayes. Sur le test-set, pour le label comp1 il donne une accuracy et un ROC-AUC d'environ 94% (cf. annexe G.2b). Son taux de vrais positifs est de 67.5% avec un taux de faux positifs très faible à 4.73% (voir en annexe F.3). En réalité il a 1 198 échantillons bien classés en positif pour 6 872 mal classés en positif. La précision est en effet très mauvaise avec un score de 14.8% seulement.

Pour évaluer les véritables performances des modèles le recall, la précision et le F1-Score sont privilégiés. Pour une usine on peut supposer que les coût d'une panne causant une interruption dans la production est plus important que celui de remplacer un composant alors qu'il est encore fonctionnel. Il est donc important d'avoir un bon recall sur chacun des labels de type failure. Cependant, si la précision est trop faible et qu'il y a alors beaucoup de faux négatif, cela veut dire que des composant sont remplacé prématurément ce qui revient à peu de chose près à faire de la maintenance prévisionnelle. On perd tout l'intérêt de la maintenance prédictive. Il faut donc un bon F1-score sur chacun des labels en s'assurant qu'il n'y a pas un trop grand déséquilibre entre le recall et la précision.

Pour le calcul des scores moyens, une moyenne macro est utilisée afin que les labels soient considérés comme étant équipondérés. Elle annule le déséquilibre des classes et permet ainsi de facilement comparer les modèles sur leurs score moyens.

Train / Test Sets

Les échantillons étant liés à leurs plus proches prédecesseurs, le split entre les sets train et test ne peut être aléatoire. Dans la pratique ce type de donnée est produite de façon ordonnée. Elles doivent rester groupées par machineID et ordonnées par datetime. C'est pourquoi le dataset est découpé avec le ratio 80/20 à de tel sorte que le train-set regroupe les données du 1er janvier 2015 à 6h00 au 20 octobre 2015 à minuit et le test-set celles du 20 octobre 2015 12h au 1er Janvier 2016 12h.

4.2 COMPARAISON DES MODÈLES

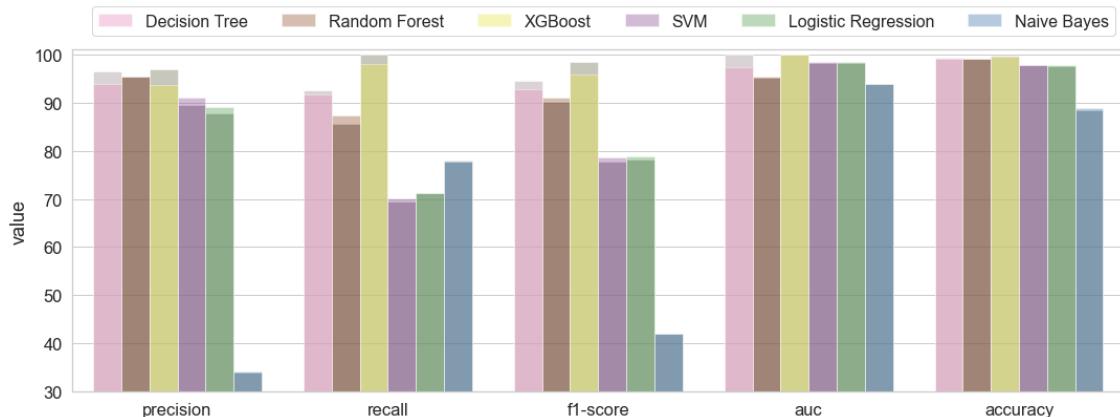
Six algorithmes de machine learning ont été mis en concurrence. Le modèle de K-Nearest-Neighbors a été disqualifié car il lui faut plus d'une heure pour faire les prédictions sur tout le

test-set (contre quelques secondes pour tous les autres modèles).

Voici ce qui ressort des résultats visibles sur la figure 4.1 :

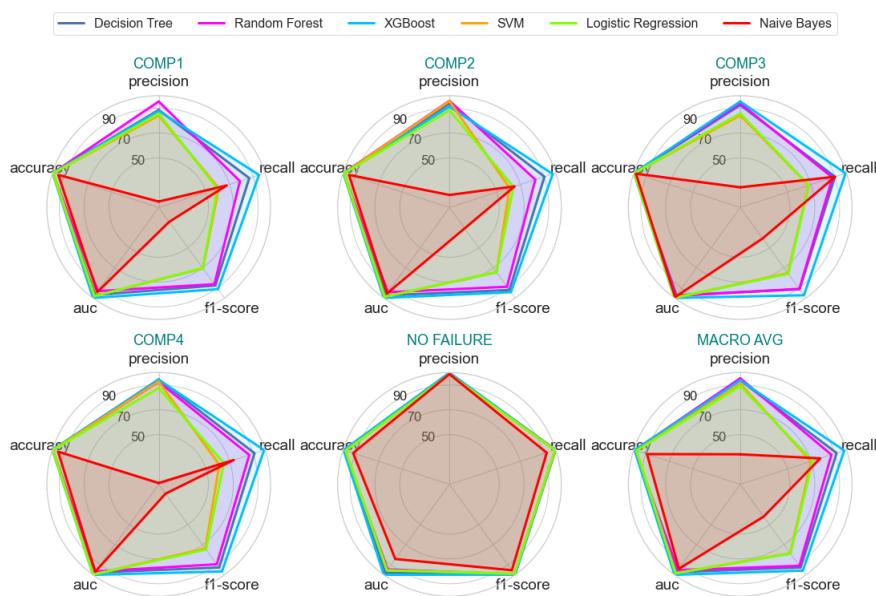
- les modèles basés sur des arbres de décision donnent de très bons résultats
- il y a peu ou pas d'overfitting quel que soit le modèle
- Le classificateur de XGBoost obtient les meilleurs scores de recall et F1-Score
- Le Naive Bayes classifie très mal avec une précision de 34% (beaucoup de faux positif)

FIGURE 4.1 – Score des Moyens Modèles - Train/Test



Chaque barre du graphique donne le score sur le train-set (couleur désaturée) et le test-set (couleur saturée) pour un modèle et une mesure de performance.

FIGURE 4.2 – Score des Modèles par Label - Test-Set



Modèle Gagnant

Le classificateur de XGBoost est de toute évidence le meilleur modèle. Il a des recalls et des F1-Scores supérieurs quelque soit le label. La Random Forest a une précision un peu plus forte (95.3% contre 93.6% pour XGBoost en moyenne macro) mais c'est au coût d'un recall et d'un F1-Score bien plus faible (87% contre 98% pour XGBoost sur le recall). Les détails des scores sont disponibles en annexe [G](#).

Voici les caractéristiques du modèle

```

1 OneVsRestClassifier(estimator=XGBClassifier(alpha=5.73,
2     base_score=None,
3     booster=None,
4     colsample_bylevel=None,
5     colsample_bynode=None,
6     colsample_bytree=0.7, gamma=2.48,
7     gpu_id=None, importance_type='gain',
8     interaction_constraints=None,
9     learning_rate=0.15,
10    max_delta_step=None, max_depth=17,
11    min_child_weight=None, missing=np.nan,
12    monotone_constraints=None,
13    n_estimators=150, n_jobs=6,
14    num_parallel_tree=None,
15    random_state=None, reg_alpha=None,
16    reg_lambda=None,
17    scale_pos_weight=None,
18    subsample=0.68, tree_method=None,
19    validate_parameters=None,
20    verbosity=None),
21    n_jobs=5)

```

4.3 FEATURE IMPORTANCE

Afin de déterminer quelles sont les features qui sont le plus déterminantes pour la prédiction, un nouveau modèle XGBoost a été entraîné sur des features dont les fenêtres des fonctions d'agrégations sont étendues à 6, 12, 18, 24, 36 et 48.

En regroupant les features par catégorie (voir annexe [H.1](#)), on peut dire ceci :

- Les fenêtres 18 et 48 sont les moins utiles
- Les fenêtres 24 et 36 importantes surtout pour le label `no failure`
- les fenêtres 6, 12 et 18 sont importantes pour les autres labels
- pour prédire une panne les features de `telemetry` sont les plus importantes
- pour le label `no failure` les features de `errors` sont les plus importantes
- l'`age` des machines est déterminant pour prédire une panne du composant 4.
- chacun des 4 types de composant est relié à un l'une des mesures télémétriques (voir annexe [H.2](#)) :

comp1 : `volt`
comp2 : `rotate`
comp3 : `pressure`
comp4 : `vibration`

Finalement les fenêtres 6, 12, 24 et 36 semblent optimales. Le classificateur avec les fenêtres étendues a une précision similaire mais un recall très légèrement inférieur (97.8 contre 98.07, cf. annexe [G.3](#)). Il n'y a donc aucun intérêt augmenter le nombre de features (qui est de 91 avec les fenêtres étendues) d'autant plus que le classificateur est deux fois plus long à entraîner.

4.4 AJUSTEMENT DE LA TARGET

Après avoir changé la valeur de K_{period} à 6 lors de la construction de la cible, on voit que le modèle ne parvient pas à bien classifier les échantillons à plus de 48 heures de la panne. En étendant cette valeur, il commence même à faire plus d'erreurs de classification sur les échantillons à moins de 48 heures de la panne. C'est pourquoi la valeur du K_{period} reste fixé à 4.

Troisième partie

Mise en Place de la Solution

Chapitre 5

Web Application

Dans le but de pouvoir démontrer l'efficacité du modèle en temps réel, une application web a été conçue à partir du framework *Django*. Elle permet de voir les prédictions de pannes faites à partir d'une simulation d'un flux de données (logs et télémetrie).

5.1 FLUX DE DONNÉES

Émission des Données

Le module `create_pipeline.py` permet de séparer les données brutes en training/testing sets à partir de la même date utilisé pour entraîner le modèle (le 20 octobre 2015 12h). Le pipeline de pre-processing est ensuite entraîné sur le training-set. L'objet qui en résulte est enfin exporté sous un format compressé sous "`/data/initial_pipeline.z`" afin qu'il puisse être utilisé par le module `Le`. Le flux de données est simulé à partir des test-sets des données brutes en utilisant *Kafka*.

Un **Kafka Producer** est en charge de la récupération des données à partir des fichiers CSV des quatre datasets `telemetry`, `errors`, `maintenance` et `failures` puis il les envoie dans un topic dédié de *Kafka*. Il simule le travail de transmission des mesures faites par les capteurs et l'enregistrement des logs de maintenance, d'erreurs et de pannes.

Pour simplifier les choses, il y a une partition par intervalle de 12 heures. Ainsi la pipeline de preprocessing n'est déclenché que lorsqu'une partition est créée et complète.

Récupération et Traitements des Données

Un **Kafka Consumer** récupère les données de chaque partition et stocke les données brutes en base de données.

Dès qu'un lot de données (soit toutes les données sur un intervalle de 12 heures) est intégralement récupéré, il est passé dans le pipeline de preprocessing pour en extraire les features qui nous intéressent. Une fois transformées, ces features sont enfin envoyées au modèle de classification et les prédictions sont stockées en base de données.

Une version agrégée par intervalle de 12 heures du dataset `failures` est stockée telle quelle dans la base de données (table `pm_failures`, cf ??). Elle est utilisée à titre informatif dans l’application afin de pouvoir juger la qualité des prédictions faites.

Suivi de la Progression de la Simulation

Des tables uniquement utiles à l’application existent aussi. Elles permettent notamment de savoir quel lot de données a déjà été envoyé, extrait preprocessé et/ou prédit. (table `actual_label`, cf 5.3b)

Mise à jour de la Pipeline

Comme on l’a vu la pipeline garde un petit historique des données pour pouvoir agréger les données en fonctions mobiles. Elle est donc mise à jour à chaque fois qu’elle est utilisée. Chaque lot de données utilise la pipeline mise à jour par le lot précédent (voir code 5.3).

5.2 APPLICATION

Test unitaires

Trois types de test unitaires sont déployés :

1. Pour les modèles (voir annexe J.3) : chaque méthodes est testée
2. Pour les vues (voir annexe J.3) : on teste la validité ou non des méthodes HTTP, les réponses renvoyées (contenu du contexte), le template utilisé et la redirection le cas échéant.
3. Pour le formulaire `Intervals` (voir annexe J.3) : test de la validité d’un formulaire vide et non-vide et de l’invalidité d’un formulaire

FIGURE 5.1 – Résultats des tests unitaires

```
(.venv) → web python manage.py test pm
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Tous ces services qui lui sont utiles sont Dockerisés et regroupés
dans un compose contenant six conteneurs:
-----
Ran 32 tests in 0.433s
OK
Destroying test database for alias 'default'...
```

Celery : Gestion des taches

Celery [4] est un gestionnaire de tâches asynchrones. Dans cette application il permet de faire deux chose :

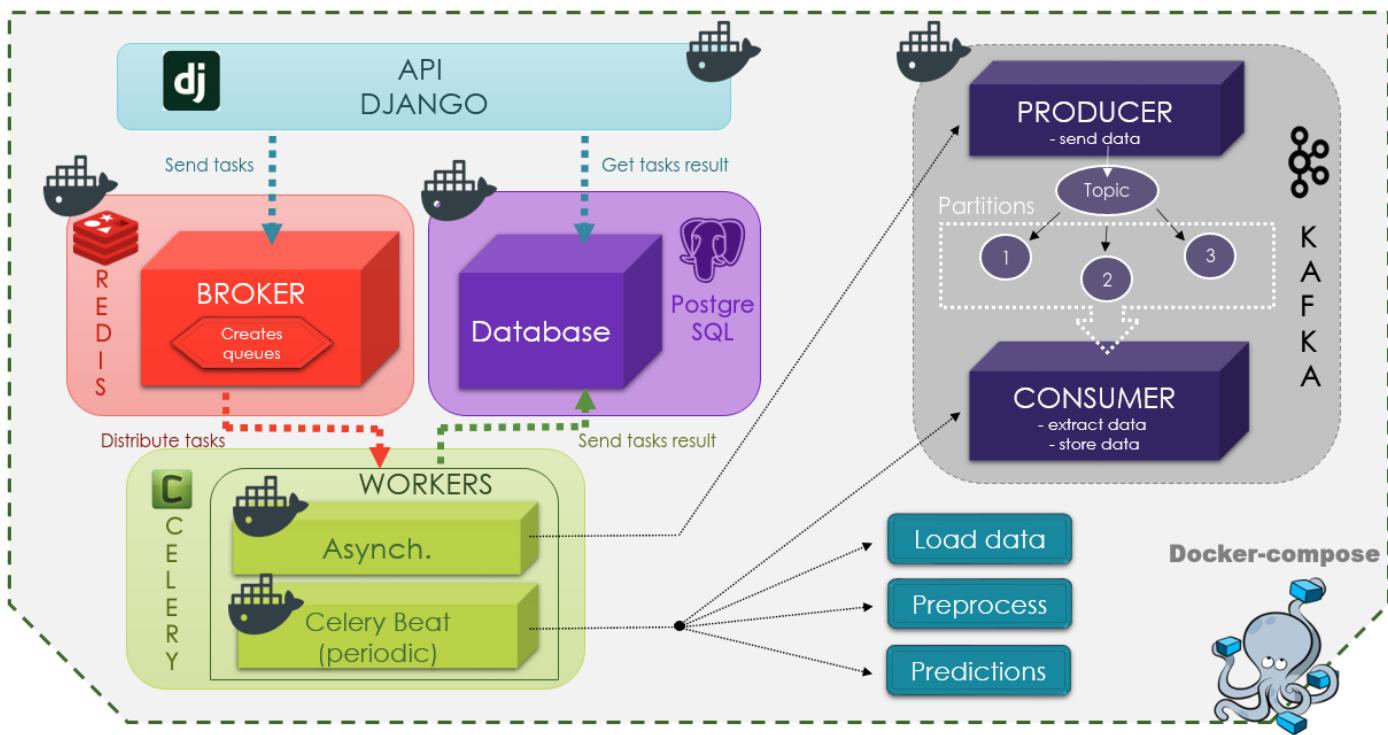
1. Démarrer la simulation du flux de données par le biais d’une tâche asynchrone qui exécute le script du *kafka producer*.
2. Programmer l’exécution automatique de la tâche qui lance le *kafka consumer* de façon périodique [5].

Dans un cas réel, le *kafka consumer* serait exécuté toutes les 12 heures puisque c’est l’intervalle utilisé par la pipeline de preprocessing. Pour la simulation ce serait trop long, on utilise donc un intervalle de cinq secondes seulement.

Docker

L'application web et tous les services qui lui sont utiles sont dockerisés et regroupés ensemble dans un docker-compose contenant six conteneurs :

FIGURE 5.2 – Simulation du Flux de Données - Workflow



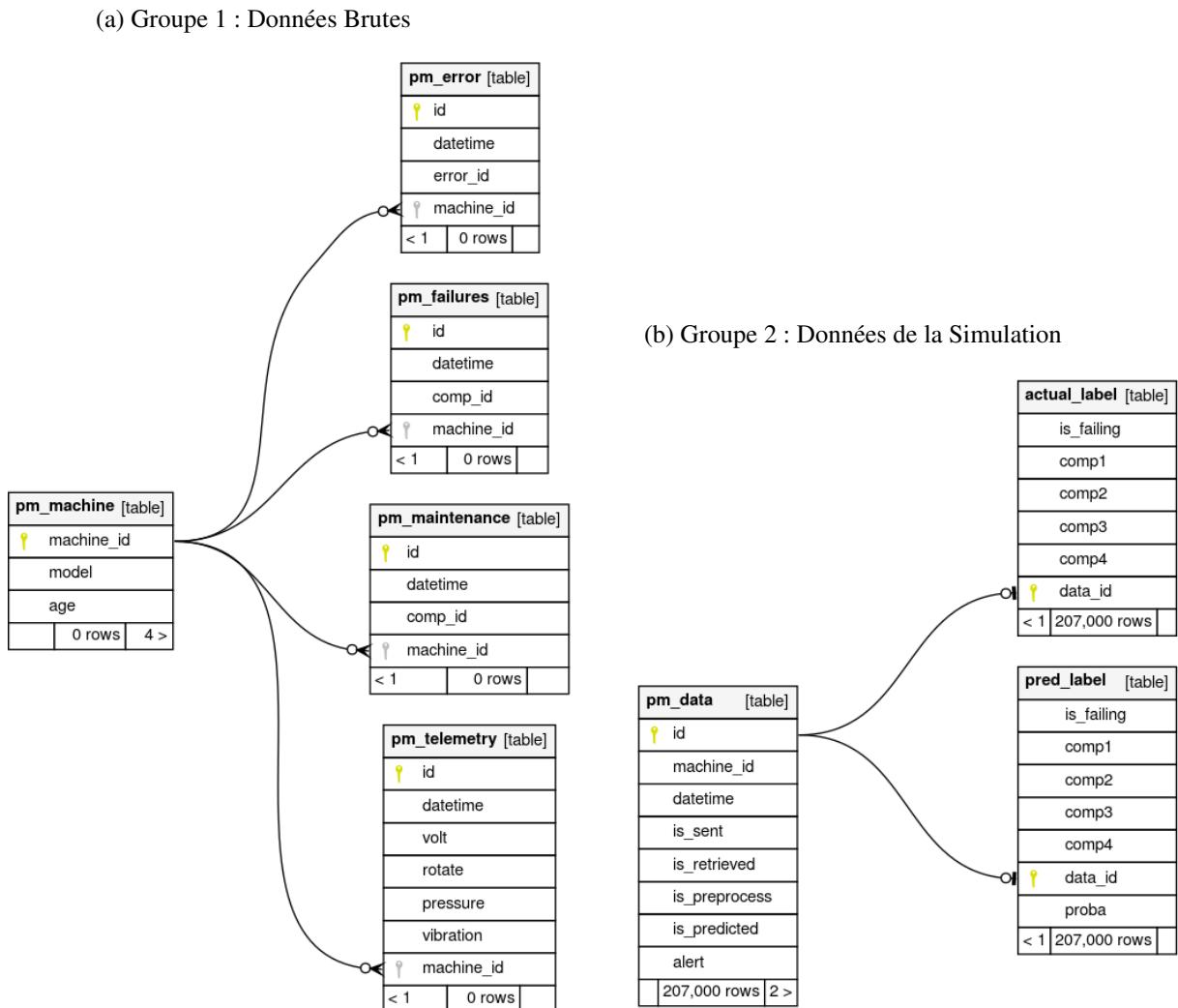
5.3 BASE DE DONNÉES

Schéma relationnel

Au-delà des tables nécessaires à *Django*, *Celery* et *CeleryBeat*, deux autres groupes de tables ont été créés. Le premier groupe contient les données brutes (et quasi-brutes) des cinq datasets (figure 5.3a). Le deuxième groupe contient les tables qui permettent de suivre l'évolution de la simulation (figure 5.3b).

Une version condensée de l'intégralité du schéma relationnel de la base de données est visible en annexe I ainsi que la définition des modèle en annexe J.1.

FIGURE 5.3 – Schéma Relationnel Détailé



Insertion des données

le module `kafka_producer.py` génère un flux de données depuis les fichiers csv d'origine puis le module ‘`kafka_consumer.py`’ ingère ces données par lot (représentant une plage de

douze heures).

Les données brutes sont stockés dans les tables pm_telemetry, pm_maintenance et pm_errors.

Code Sample 5.1 – /web/modules/kafka_consumer.py

```

1 # [...]
2 def fill_raw_db(maint, errors, telemetry):
3     for _, row in errors.iterrows():
4         e = Error.objects.create(
5             machine=Machine.objects.get(machine_id=row.machineID),
6             datetime=row.datetime,
7             error_id=row.errorID)
8         e.save()
9
10    for _, row in maint.iterrows():
11        m = Maintenance.objects.create(
12            machine=Machine.objects.get(machine_id=row.machineID),
13            datetime=row.datetime,
14            comp_id=row.comp)
15        m.save()
16
17    for _, row in telemetry.iterrows():
18        t = Telemetry.objects.create(
19            machine=Machine.objects.get(machine_id=row.machineID),
20            datetime=row.datetime,
21            volt=row.volt,
22            pressure=row.pressure,
23            rotate=row.rotate,
24            vibration=row.vibration)
25        t.save()
26 # [...]
```

Les données de la table pm_machine sont statiques et ne font pas partie du même script d'ingestion. Cette table est ré-initialisée à chaque lancement de la simulation de données. La logique est contenue directement dans le "Model Manager" du modèle Machine :

Code Sample 5.2 – web/pm/models.py

```

1 # [...]
2 class MachineManager(models.Manager):
3     def initialize(self):
4         if self.all().count() >= 1000:
5             logger.info("Machine model already initialized")
6         else:
7             logger.info("Initializing Machines..")
8             df = pd.read_csv(os.path.join(settings.BASE_DIR, "data/raw/machines.csv"))
9             j=1
10            for _, row in df.iterrows():
11                m = Machine.objects.create(machine_id=row.machineID, model=row.model, age=row.age)
12                m.save()
13                j+=1
14            logger.info(f"Initializing Actual Data -- DONE {Machine.objects.all().count()} machines")
15 # [...]
```

Mise à jour des données ingérées

Une fois transformée par le pipeline de pre-processing, la table pm_data est mise à jour

Code Sample 5.3 – web/modules/preprocess.py

```

1 # [...]
2 def create_features(df_dict, partition, init=False):
3     # [...]
4     if init:
5         pipeline = joblib.load(settings.INITIAL_PIPELINE_PATH)
6     else:
7         pipeline = joblib.load(settings.CURRENT_PIPELINE_PATH)
8     # [...]
9     features = pipeline.transform(df_dict, True)
10    joblib.dump(pipeline, settings.BASE_DIR + "/data/current_pipeline.z")
11
12 # [...]
```

```

13     for _, row in features.iterrows():
14         d = Data.objects.get(datetime=row.datetime, machine_id=row.machineID)
15         d.is_preprocess = True
16         d.save()
17
18     return features
19 # [...]

```

Prédictions

Dès qu'un lot de données a été ingéré et preprocessé, les prédictions peuvent être faites et la table pm_data est encore une fois mise à jour.

Code Sample 5.4 – web/modules/prediction.py

```

1 # [...]
2 def get_predictions(x, partition):
3     # [...]
4     mlb = load(settings.BASE_DIR + "/data/binarizer")
5     model = load(settings.BASE_DIR + "/data/model_xgb.joblib")
6     std_scaler = load(settings.BASE_DIR + "/data/std_scaler.p")
7     # [...]
8     x = prepare_data(x, std_scaler)
9     # [...]
10    preds = model.predict(x)
11    df_pred = pd.DataFrame(preds, index=x.index, columns=mlb.classes_)
12
13    for _, row in df_pred.reset_index().iterrows():
14        d = Data.objects.get(machine_id=row.machineID, datetime=row.datetime)
15        d.is_predicted = True
16        d.save()
17
18        d.prediction.comp1 = row.comp1
19        d.prediction.comp2 = row.comp2
20        d.prediction.comp3 = row.comp3
21        d.prediction.comp4 = row.comp4
22
23        d.prediction.save()
24        d.prediction.set_failure()
25        d.set_alert()
26
27 def prepare_data(data, std_scaler):
28     """
29     drop columns that are not real features used by the model
30     and normalize data with the Standard Scaler
31     """
32     indexcols = ['machineID', 'datetime']
33     remove_names = indexcols + ['failure', 'model', 'label',
34                                 'label_list', 'label_code', 'is_failing']
35     input_features = [x for x in data.columns if x not in remove_names]
36
37     data.set_index(indexcols, inplace=True)
38
39     return pd.DataFrame(std_scaler.transform(data[input_features]),
40                         columns=input_features,
41                         index=data.index)

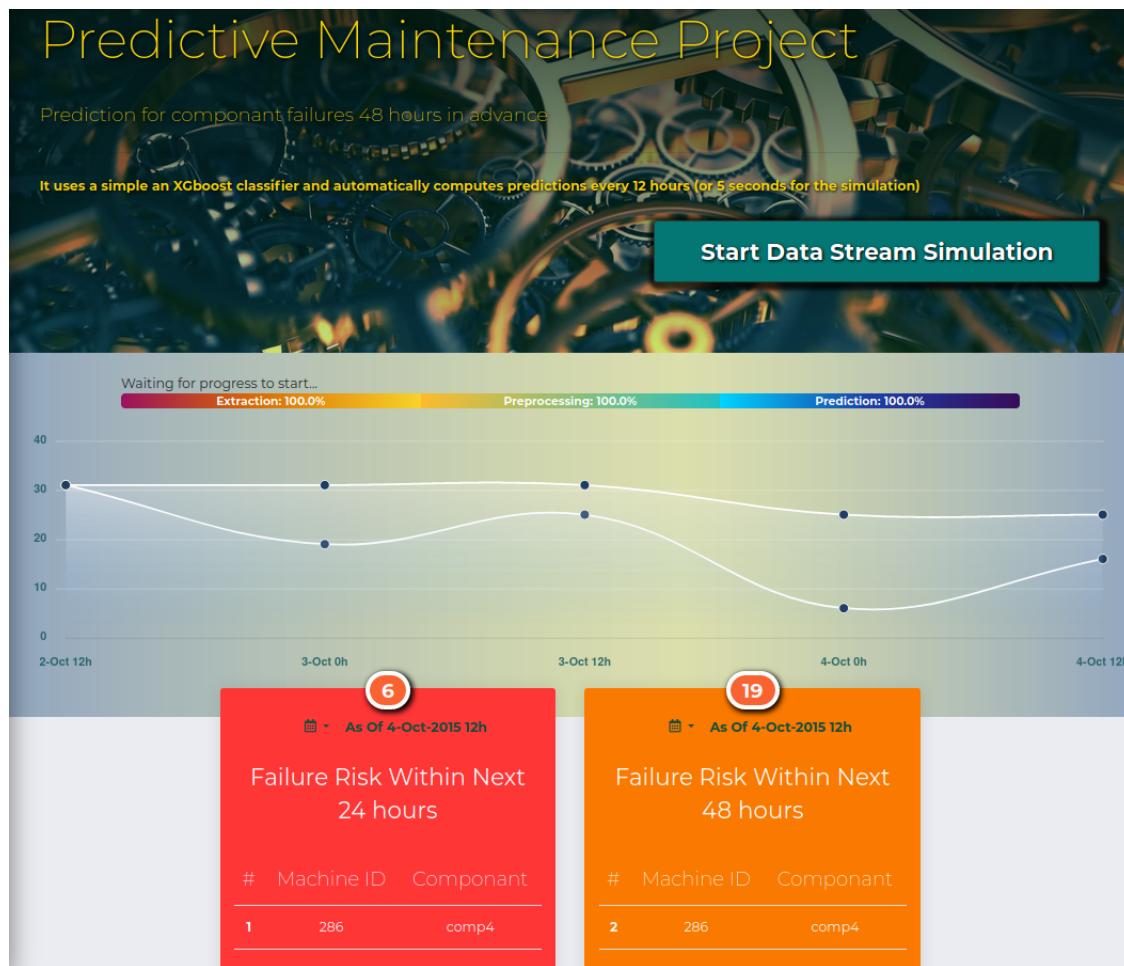
```

5.4 INTERFACE UTILISATEUR

L'interface est composée d'une première partie pour le lancement et le suivi de la simulation de donnée avec un graphique montrant le nombre de pannes prédictes et réelles et diverses barres de progression pour le suivi de l'état de la génération, de l'extraction, du pré-traitement et de la classification des données.

La deuxième partie montre quelles machines et quels composants risquent de tomber en panne à une date choisie. Il y a deux niveaux d'alertes. Orange si la panne est prévue dans les 24 heures et rouge si elle est prévue dans les 24 heures.

FIGURE 5.4 – Screenshot de l'application



Chapitre 6

Bilan

6.1 GESTION DE PROJET

Les différentes étapes de la mise en place du projet ont été organisées avec Trello.

The screenshot shows a Trello board titled "Projet Simplon - Prédictive Maintenance". The board is organized into several columns:

- INFO:** Objectif: Créer une application de maintenance prédictive avec simulation d'un flux de données
Github: https://github.com/Bapuch/Predictive_Maintenance
- BIG STEPS:**
 - Data Vizualisation (5/5)
 - Pipeline Preprocessing (4/4)
 - Machine Learning (4/4)
 - Deep Learning (0/3)
 - Django Application (9/11)
- TO DO:**
 - Deployer l'API sur une plateforme Cloud (1)
 - Utiliser Fastai (1)
- IN PROGRESS:**
 - Faire un dashboard de visualisation des predictions (1)
- REVIEW:**
 - Entraînés un LSTM (1)
 - Entraîner un RNN (1)
- DONE:**
 - Créer un Transformer pour Telemetry (1)
 - Créer un Transformer pour Errors (1)
 - Créer un Transformer pour Maintenance (1)
 - Créer une classe pour construire les labels avec Failures (1)
 - Dataviz pour le dataset de telemetry (1)
 - Dataviz pour le dataset de machines (1)
 - Dataviz pour le dataset de errors (1)

6.2 VERSION CONTROL

Tout le projet est versionné avec git et hébergé sur [GitHub](#) (repo privé).

The screenshot shows a GitHub repository page for 'Bapuch / Predictive_Maintenance'. The repository has 6 branches and 0 tags. The commit history shows 183 commits from 'c3530ed' 6 hours ago. The 'About' section indicates no description, website, or topics provided. The 'Releases' section shows no releases published, with a link to 'Create a new release'. The 'Packages' section shows no packages published, with a link to 'Publish your first package'. The 'Languages' section shows Jupyter Notebook at 99.6%, HTML at 0.2%, TeX at 0.1%, Python at 0.1%, JavaScript at 0.0%, and CSS at 0.0%.

6.3 CONCLUSION

Il a été démontré qu'il est a priori possible de construire une méthode efficace pour prédire les pannes 48 heures avant qu'elles ne se produisent. Pour cela il est primordial de travailler les données pour en extraire des features déterminantes. Parmi les features les importantes se trouvent le nombre de remplacements des composants et les calculs faits sur les données télémétriques.

Les algorithmes basés sur des arbres de décision sont les plus efficaces sur ce type de données. En réalité un simple arbre de décision donne déjà de très bon résultats.

Il est important d'avoir une bonne vision des problèmes de coûts réels engendrés par la maintenance et les pannes au sein d'une usine afin de savoir comment optimiser le modèle, c'est-à-dire en fonction du taux de faux positifs ou plutôt en fonction du taux de faux négatifs.

6.4 AMÉLIORATIONS

Pour aller plus loin, on pourrait utiliser des réseaux de neurones et éventuellement vérifier si la marge d'anticipation (les 48 heures) peut être allongée. Pouvoir tester le modèle sur des données réelles serait aussi intéressant. On pourrait aussi l'adapter pour des données non industrielles, pour les avions ou les voitures connectées par exemple.

Bibliographie

- [1] Microsoft Azure Advanced Scenario: General Predictive Maintenance
- [2] Implementing Bayesian Optimization On XGBoost: A Beginner's Guide
- [3] Bayesian Optimization For XGBoost
- [4] Asynchronous Tasks With Django and Celery
- [5] Django-celery-beat
- [6] kafka-python

Appendices

Annexe A

Fréquence des Interventions Sur Une Machine

FIGURE A.1 – Nombre d'interventions par mois pour chaque composant sur la machine ID=977

		comp1	comp2	comp3	comp4
year	month				
2014	June	1	1	1	1
	August	1	1	1	1
	September	1	1	1	1
	October	1	1	1	1
2015	January	1	1	1	1
	February	2	2	2	2
	March	2	2	2	2
	April	2	2	2	2
	May	2	2	2	2
	June	3	3	3	3
	July	1	1	1	1
	August	2	2	2	2
	September	2	2	2	2
	October	2	2	2	2
	November	2	2	2	2
	December	3	3	3	3

Annexe B

Distributions des Features

FIGURE B.1 – Distribution des Features de TELEMETRY

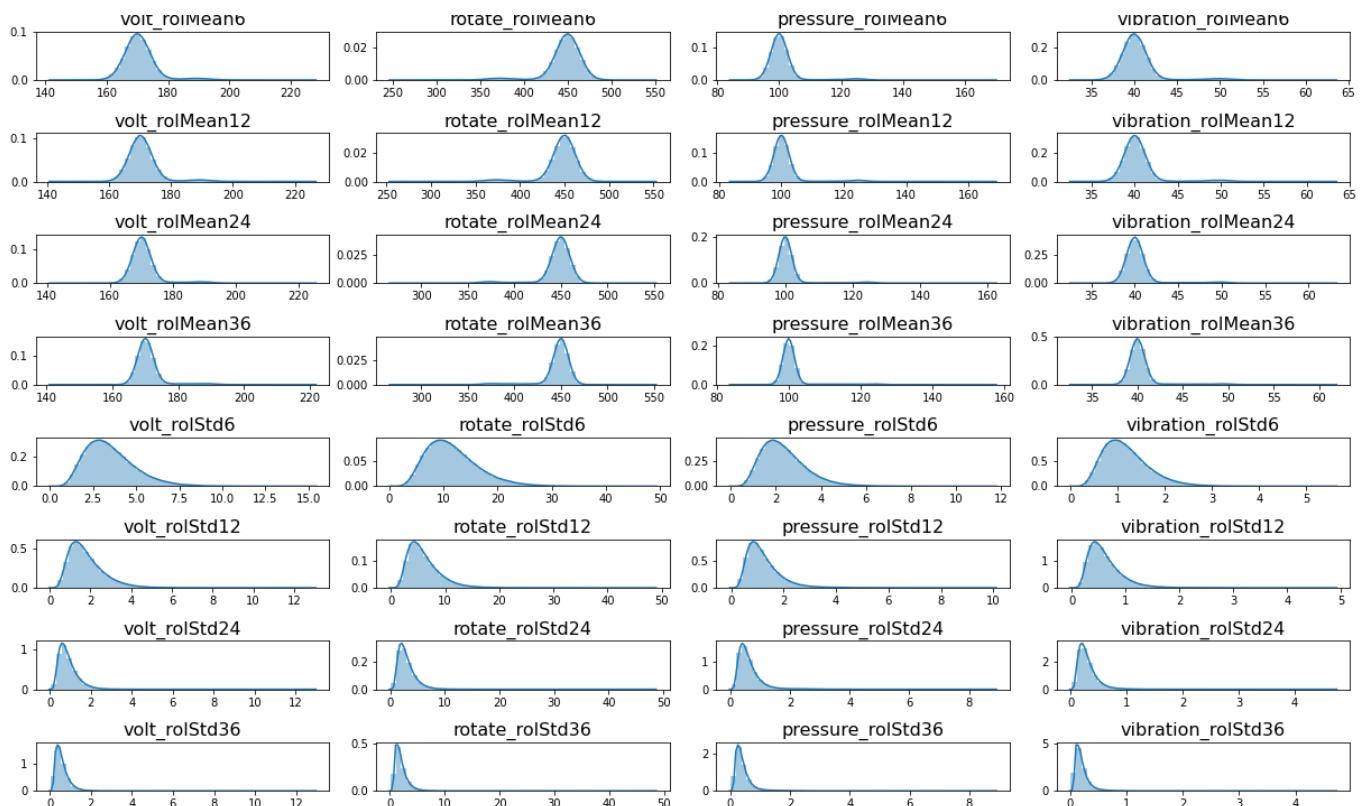


FIGURE B.2 – Distribution des Features de MAINTENANCE

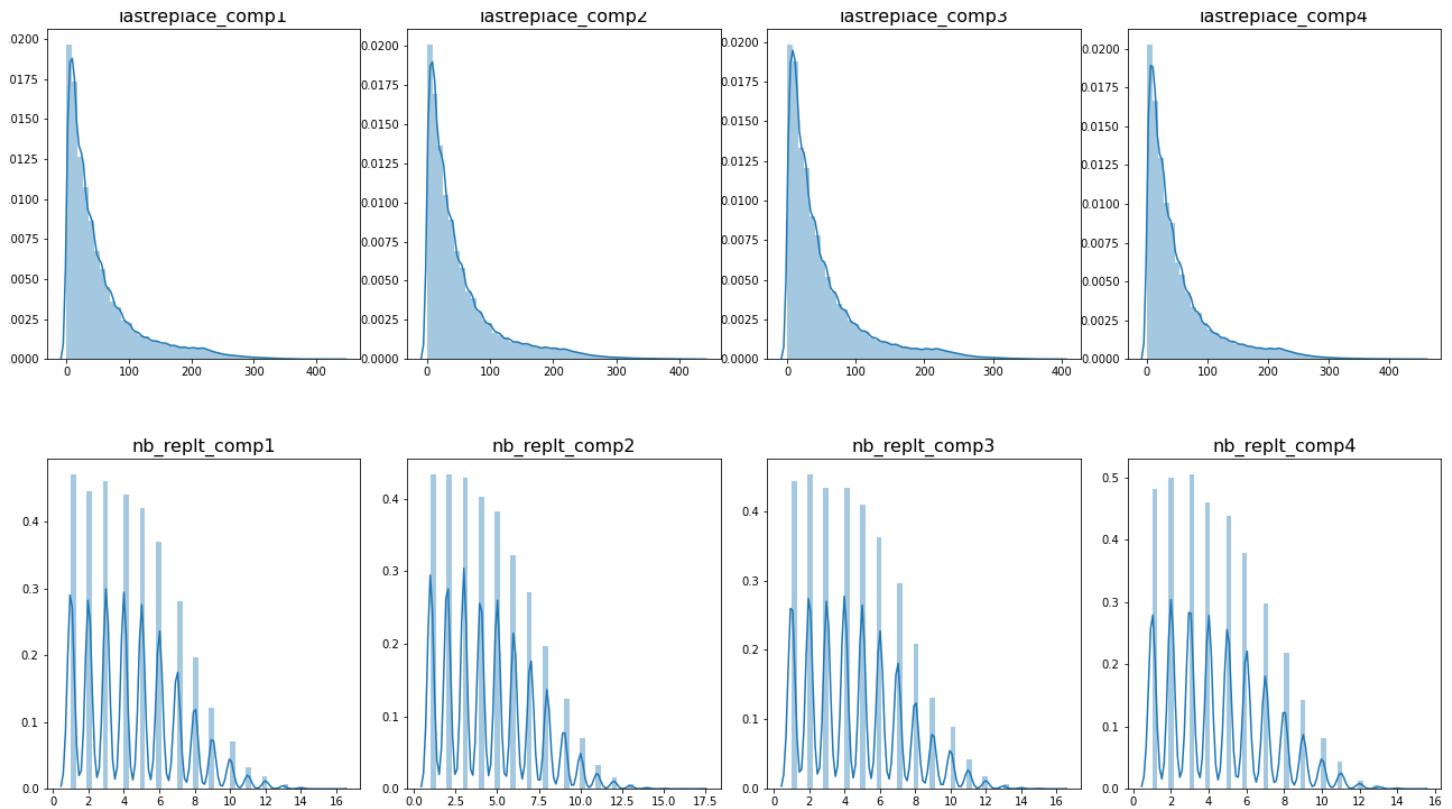
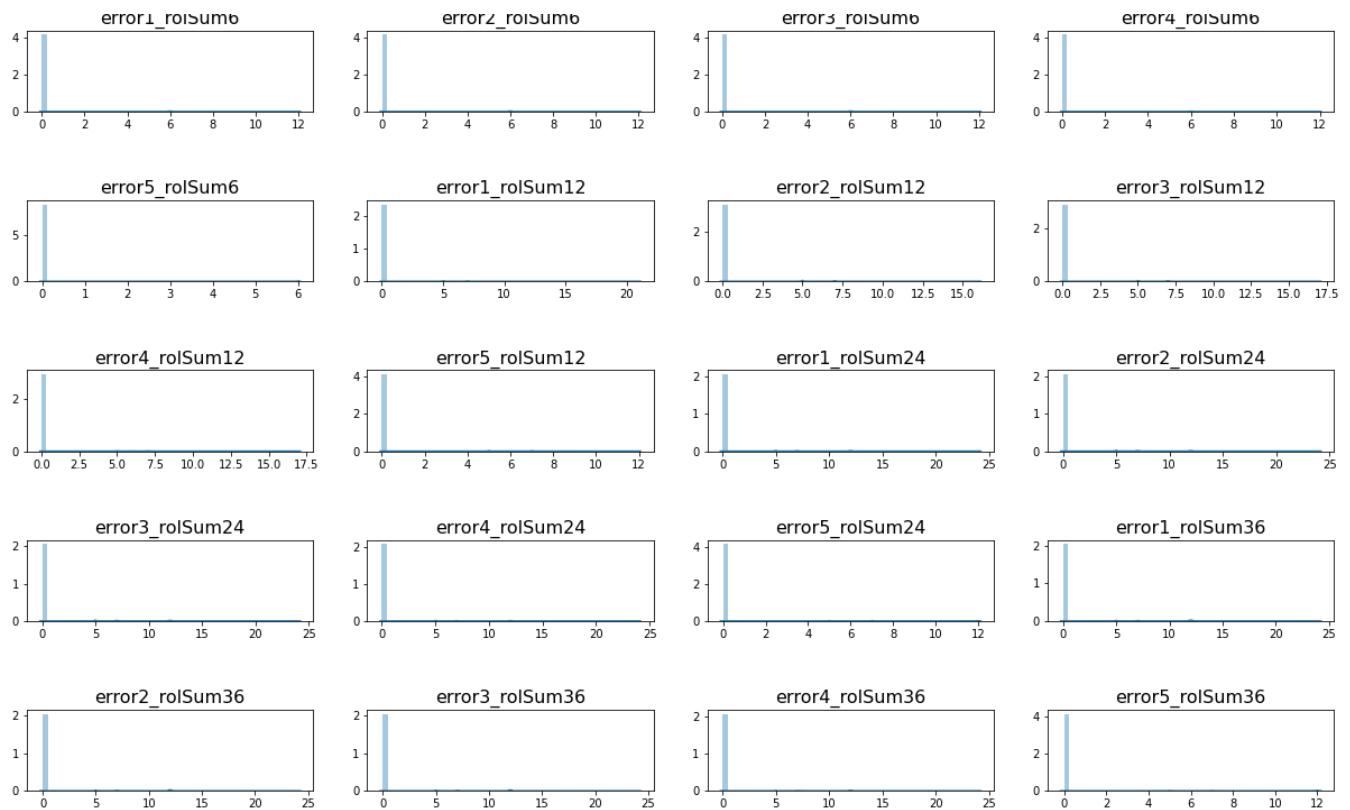


FIGURE B.3 – Distribution des Features de ERRORS



Annexe C

Preprocessing Piepline - Code

web/modules/preprocessing_pipeline.py

C.1 FEATURE PIPELINE (CLASSE MÈRE)

```
1 class FeaturePipe(BaseEstimator, TransformerMixin):
2     def __init__(self, freq='12h'):
3         self.index_cols = ['machineID', 'datetime']
4         self.freq = freq
5
6     def _format_dataframe(self, X, index=False):
7         df = X.copy()
8         try:
9             df.datetime = pd.to_datetime(df.datetime, format="%Y-%m-%d %H:%M:%S")
10            df = df.sort_values(self.index_cols)
11        except Exception as e:
12            logger.info('WARN:', str(e))
13
14        return self._reindex(df) if index else df
15
16    def _reindex(self, X, reverse=False):
17        return X.reset_index(self.index_cols[::-1]) if reverse else X.set_index(self.index_cols[::-1])
18
19    def _merge_histo(self, X):
20        """
21        merges historical data with current data
22        """
23        df = pd.concat([self.histo, X]).sort_values(self.index_cols)
24        if df.duplicated(self.index_cols).sum():
25            df.drop_duplicates(inplace=True)
26        return df
27
28    def set_histo(self, X, index=False):
29        """
30        keeps an history of the n last records for each machine
31        n: Number of rows to select
32        """
33        df = self._format_dataframe(X, index)
34        self.histo = df.groupby('machineID').tail(self.hist_period)
35
36    def _group_by_freq(self, X, func=None):
37        return X.groupby(['machineID', pd.Grouper(freq=self.freq, level='datetime',
38                                                 closed='right', label='right')]).agg(func)
39
40    def set_min_date(self, ):
41        self.min_date = self.histo.datetime.max().ceil(self.freq)
42
43    def return_result(self, df_res):
44        return df_res[df_res.datetime > self.min_date] if self.add_histo else df_res
45
46    def _get_moving_average(self, X, fmean, fstd, fsun):
47        df = X.drop(X.columns, axis=1)
48
49        if fmean:
50            for window in self.lags:
51                drol = X.groupby('machineID').rolling(window, min_periods=1).mean()
```

```

52     for coln in X.columns:
53         df[coln + '_rolMean' + str(window)] = drol[coln].to_numpy()
54
55     if fstd:
56         for window in self.lags:
57             # each first row by machine is NaN so we fill it with the next value for std
58             drol = X.groupby('machineID').rolling(window,
59                                         min_periods=1).std().fillna(method='bfill')
60             for coln in X.columns:
61                 df[coln + '_rolStd' + str(window)] = drol[coln].to_numpy()
62
63     if fsum:
64         for window in self.lags:
65             drol = X.groupby('machineID').rolling(window, min_periods=1).sum()
66             for coln in X.columns:
67                 df[coln + '_rolSum' + str(window)] = drol[coln].to_numpy()
68
69     return df
70
71 def set_dico(self, X, cols=None):
72     """Initializes the dictionary containing the mapping of each feature with its aggregation
73     function"""
74     if cols is None:
75         sensors = X.drop(self.index_cols, axis=1).columns
76     else:
77         sensors = cols
78     self.dico_agg = dict()
79     for f in self.funcs:
80         for lag in self.lags:
81             for sensor in sensors:
82                 self.dico_agg[sensor + f"_rol{f.capitalize()}" + str(lag)] = f

```

C.2 TELEMETRY PIPELINE

```

1 class TelemetryPipe(FeaturePipe):
2     """
3         for each machine, compute the moving average on 3 windows (default: 12, 24, 36) for mean and std
4             by default
5             Then group by machine and by a 12h (by default) hour window (using the previous 12 rows)
6     """
7     def __init__(self, lags=[12, 24, 36], funcs=['mean', 'std'], freq='12h'):
8         super().__init__(freq)
9         self.lags=lags
10        self.funcs = funcs
11        self.hist_period = np.max(self.lags) - 1
12
13    def fit(self, X, y=None):
14        self.set_dico(X)
15        self.set_histo(X)
16        Z = self._format_dataframe(X[self.index_cols], True)
17        self.grouped_tp = self._group_by_freq(Z, 'count')
18        return self
19
20    def transform(self, X, add_histo=False):
21        self.add_histo = add_histo
22
23        logger.info('>>> telemtry pipe, format')
24
25        if self.add_histo:
26            logger.info('>>> telemtry pipe, merge histo')
27            self.set_min_date()
28            X = self._merge_histo(X)
29            # reset histo with new data
30            self.set_histo(X)
31
32        X = self._format_dataframe(X, True)
33
34        logger.info('>>> telemtry pipe, moving average')
35        do_mean = 'mean' in self.funcs
36        do_std = 'std' in self.funcs
37        do_sum = 'sum' in self.funcs
38        df_res = self._get_moving_average(X, do_mean, do_std, do_sum)
39        logger.info('>>> telemtry pipe, groupby')
40
41
42        df_res = self._group_by_freq(df_res, self.dico_agg)
43
44        logger.info('>>> telemtry pipe, reindex')
45        df_res = self._reindex(df_res, True)

```

```

46     logger.info('>>> telemetry pipe, final step')
47     return self.return_result(df_res)

```

C.3 ERROR PIPELINE

```

1  class ErrorPipe(FeaturePipe):
2      """for each machine at each time point (every 12h)
3          get the average number of error during the past 24h for each error type
4      """
5      def __init__(self, timepoints, lags=[24], funcs=['sum'], freq='12h'):
6          super().__init__(freq)
7          self.lags = lags
8          self.funcs = funcs
9          self.timepoints = self._format_dataframe(timepoints)
10         self.hist_period = max(self.lags) - 1
11
12     def fit(self, X, y=None):
13         self.set_dico(X, cols=[f'error{i+1}' for i in range(X.errorID.unique())])
14         return self
15
16     def transform(self, X, add_histo=False, new_timepoints=None,):
17         self.add_histo = add_histo
18
19         if self.add_histo and new_timepoints is None:
20             return logger.info("ERROR: If parameter add=True then new_timeline must be provided")
21
22         if X.shape[0] > 0:
23             df = self._format_dataframe(X)
24
25             # dummyfy ErrorID
26             df_dum = pd.get_dummies(df, columns=['errorID']).groupby(['machineID',
27                             'datetime']).sum().reset_index()
28
29             if self.add_histo:
30                 self.set_min_date()
31                 new_timepoints = self._format_dataframe(new_timepoints)
32                 self.timepoints = self._update_timepoints(new_timepoints)
33
34                 if X.shape[0] > 0:
35                     # merge result with all timepoints (to use roll on last 24h)
36                     df_dum_extended = (df_dum.merge(new_timepoints,
37                                         how='outer', on= self.index_cols)
38                         .fillna(0)
39                         .sort_values(self.index_cols))
40
41                     # merge histo and new data
42                     df_dum_extended = self._merge_histo(df_dum_extended)
43                 else:
44                     df_dum_extended = (pd.concat([self.histo, new_timepoints], sort=True)
45                         .sort_values(self.index_cols)
46                         .fillna(0))
47
48                 else:
49                     df_dum_extended = (df_dum.merge(self.timepoints,
50                                         how='outer', on= self.index_cols)
51                         .fillna(0)
52                         .sort_values(self.index_cols))
53
54             self.set_histo(df_dum_extended) #####
55
56             df_dum_extended.columns = df_dum_extended.columns.str.replace('errorID_', '')
57
58             do_mean = 'mean' in self.funcs
59             do_std = 'std' in self.funcs
60             do_sum = 'sum' in self.funcs
61
62             df_roll = self._get_moving_average(self._reindex(df_dum_extended), do_mean, do_std, do_sum)
63
64             # group by frequency of 12h - if add=True return only last timepoint by machine
65             df_res = self._group_by_freq(df_roll, func=self.dico_agg)
66
67             # reset indices
68             df_res = self._reindex(df_res, reverse=True)
69             return self.return_result(df_res)
70
71     def _update_timepoints(self, new_tp):

```

```

72     self.timepoints = self.timepoints.groupby('machineID').tail(self.hist_period)
73
74     return (pd.concat([self.timepoints, self._format_dataframe(new_tp)])
75         .sort_values(self.index_cols)
76         .drop_duplicates(self.index_cols))

```

C.4 MAINTENANCE PIPELINE

```

1  class MaintPipe(FeaturePipe):
2      def __init__(self, timepoints, freq='12h'):
3          """
4              timepoints: the grouped by 12h timepoints
5              WANING: Not the same timepoints used for ErrorPipe()
6          """
7          super().__init__(freq)
8          self.timepoints = timepoints
9          self.hist_period = 1
10
11     def fit(self, X, y=None):
12         return self
13
14     def transform(self, X, add_histo=False, new_timepoints=None, ):
15         self.add_histo=add_histo
16
17         # update timepoints if necessary
18         self.timepoints = new_timepoints if self.add_histo else self.timepoints
19
20         if X.shape[0] > 0:
21             df = self._format_dataframe(X)
22
23             # dummyfy Comp
24             df_dum = (pd.get_dummies(df, columns=['comp'])
25                         .rename(columns={'datetime':'datetime_maint'})
26                         .groupby(['machineID', 'datetime_maint'], as_index=False).sum())
27
28             df_dumbis = (df_dum.groupby(['machineID', 'datetime_maint'])[
29                             [c for c in df_dum.columns if 'comp' in c]]
30                             .sum().groupby('machineID').cumsum())
31
32             df_count = self.timepoints.merge(df_dumbis.reset_index(['machineID', 'datetime_maint']),
33                                             how='outer', on='machineID')
34             df_count = df_count[(df_count.datetime > df_count.datetime_maint)]
35             df_count = df_count.groupby(['machineID', 'datetime'], as_index=False).last()
36             df_count.columns = ([c for c in df_count.columns if 'comp' not in c] +
37                                 [c.replace("comp", "nb_repl_") for c in df_count.columns if 'comp' in c])
38
39             # get last replacement date for each component
40             cols = [coln for coln in df_dum.columns if 'comp' in coln]
41             comps = []
42             for i, coln in enumerate(cols):
43                 comp = df_dum[df_dum[coln] == 1][['machineID', 'datetime_maint', coln]]
44                 comps.append(self.get_last_replacement_date(comp, coln))
45
46
47             df_res = comps[0]
48             comps.pop(0)
49             while comps:
50                 df_res = df_res.merge(comps[0], how='outer', on=self.index_cols)
51                 comps.pop(0)
52
53             df_res = df_res.merge(df_count.drop('datetime_maint', axis=1), how='outer',
54                                   on=self.index_cols)
55
56             del df_dum, df_dumbis, df_count, comps, df
57         else:
58             df_res = X
59
60             if self.add_histo:
61                 self.set_min_date()
62                 # add diff
63                 df_res = self.get_all_results_from_histo(df_res)
64                 self.histol = self.histo.copy()
65
66                 self.set_histo(df_res)
67             return self.return_result(df_res)
68
69     def get_last_replacement_date(self, comp, col_name):
70         df = self.timepoints.merge(comp, how='outer', on='machineID')

```

```

70     # get the last replacement date for each timepoint
71     df = df[(df.datetime > df.datetime_maint)]
72
73     # sort df and remove duplicates keeping the max datetime_maint each time (last date of
74     # maintenance)
75     df = df.groupby(self.index_cols, as_index=False).max()
76
77     # Get number of days between replacements
78     df['lastreplace_comp' + col_name.strip('comp_')] = (df.datetime - df.datetime_maint) /
79         np.timedelta64(1,'D')
80     return df.drop(['datetime_maint', col_name], axis=1)
81
82
83     def get_all_results_from_histo(self, X):
84         ## Maintenance logs are done once a day at 6am but timepoints are bi-daily
85         ## Get the list of dates present in timepoints but not computed by the transformation
86         tp_dates = self.timepoints.datetime.unique()
87         dates_to_add = [c for c in tp_dates if c not in X.datetime.unique()]
88
89         logger.info(f">> dates_to_add {len(dates_to_add)}, data shape {X.shape}, tp {len(tp_dates)}")
90         logger.info(dates_to_add, tp_dates)
91         ## Add all MachineIDs to result for each final dates
92         dfs = []
93         for i, dtime in enumerate(X.datetime.unique()):
94             dfs.append(X[X.datetime == dtime].merge(self.histo.machineID, on='machineID', how='outer'))
95             dfs[i]['datetime'] = dtime
96
97         ## add the missing date
98         for dtime in dates_to_add:
99             dfs.append(self.histo.copy())
100            dfs[-1].datetime = dtime
101
102         del dates_to_add
103
104         ## Concatenate all results
105         df = pd.concat([d for d in dfs]).sort_values(self.index_cols)
106
107         ## fillna with difference with histo date + histo value
108         days_cols = [c for c in df.columns if c.startswith('lastreplace_comp')]
109         cnt_cols = [c for c in df.columns if c.startswith('nb_repl_')]
110         tmp1 = df.set_index('machineID')
111         tmp2 = self.histo.set_index('machineID').drop('datetime', axis=1)
112         sub_df = tmp1.merge(tmp2,
113             how='outer', on='machineID',
114             suffixes=('_d', '')).drop([col + '_d' for col in days_cols], axis=1).reset_index()
115
116         del tmp1, tmp2
117
118         sub_df['diff'] = ((sub_df.datetime - self.min_date) / np.timedelta64(1,'D'))
119
120         for c in days_cols:
121             if X.shape[0] > 0:
122                 df[c] = np.where(df[c].isnull(), sub_df[c] + sub_df['diff'], df[c])
123             else:
124                 df[c] = (sub_df[c] + sub_df['diff']).values
125
126         df.reset_index(drop=True, inplace=True)
127
128         ## and 0 for nb_repl
129         if X.shape[0] > 0:
130             for c in cnt_cols:
131                 df[c] = sub_df[c + '_d'].fillna(0) + sub_df[c]
132
133         del sub_df
134
135         return df

```

C.5 FEATURE MERGER

C'est cette classe qui est instanciée par l'API.

```

1 class FeatureMerger(BaseEstimator, TransformerMixin):
2     def __init__(self, telemetry_lags, telemetry_funcs, error_lags, error_funcs, freq='12h'):
3         self.telemetry_lags = telemetry_lags
4         self.telemetry_funcs = telemetry_funcs
5         self.error_lags = error_lags
6         self.error_funcs = error_funcs
7         self.freq = freq

```

```

8
9     def fit(self, X, y=None):
10        """
11        X: must be a dict of dataframes as such:
12            {'telemetry': df_telemetry,
13             'errors': df_error,
14             'maintenance': df_maint,
15             'machines': df_machines}
16        """
17        logger.info('FIT:')
18        # unpack data
19        self.unpack_data(X)
20        logger.info('>> fit: timepoints for errors')
21        # timepoints for errors
22        self.error_timepoints = self.telemetry[['datetime', 'machineID']]
23
24        logger.info('>> fit: telemetry')
25        # telemetry
26        self.tel_transformer = TelemetryPipe(lags=self.telemetry_lags, funcs=self.telemetry_funcs,
27                                             freq=self.freq)
28        self.tel_transformer.fit(self.telemetry)
29
30        # timepoints for maintenance
31        logger.info('>> fit: timepoints for maintenance')
32        self.maint_timepoints = self.tel_transformer.grouped_tp.reset_index(['machineID', 'datetime'])
33        logger.info('>> fit: errors')
34
35        # errors
36        self.err_transformer = ErrorPipe(self.error_timepoints,
37                                         lags=self.error_lags, funcs=self.error_funcs, freq=self.freq)
38        self.err_transformer.fit(self.errors)
39        logger.info('>> fit: maintenance')
40
41        # maintenance
42        self.maint_transformer = MaintPipe(self.maint_timepoints, freq=self.freq)
43        self.maint_transformer.fit(self.maintenance)
44        logger.info('>> fit: machines')
45
46        # machines
47        self.machines_encoder = LabelBinarizer()
48        self.machines_encoder.fit(self.machines.model)
49
50        return self
51
52    def transform(self, X, add_histo=False):
53        """
54        X: must be a dict of dataframes as such:
55            {'telemetry': df_telemetry,
56             'errors': df_error,
57             'maintenance': df_maint,
58             'machines': df_machines}
59        add: Boolean, default False,
60            Must be True if X is different from the data used for fit
61
62        Each dataframe is overwritten with the result of each fit
63        """
64        logger.info('TRANSFORM:')
65        # unpack data
66        self.unpack_data(X, False)
67        logger.info('transform: timepoints for errors')
68
69        # timepoints for errors
70        self.error_timepoints = self.telemetry[['datetime', 'machineID']] if add_histo else None
71        logger.info('transform: telemetry')
72
73        # telemetry
74        self.telemetry = self.tel_transformer.transform(self.telemetry, add_histo=add_histo)
75
76        # timepoints for maintenance
77        logger.info('transform: timepoints for maintenance')
78        self.maint_timepoints = self.telemetry[['datetime', 'machineID']] if add_histo else None
79
80        # errors
81        logger.info('transform: errors')
82        self.errors = self.err_transformer.transform(self.errors, add_histo=add_histo,
83                                                     new_timepoints=self.error_timepoints)
84
85        # maintenance
86        logger.info('transform: maintenance')
87        self.maintenance = self.maint_transformer.transform(self.maintenance,
88                                                       add_histo=add_histo,
89                                                       new_timepoints=self.maint_timepoints)

```

```
88     # machines
89     logger.info('transform: machines')
90     if not add_histo:
91         encoded_models = self.machines_encoder.transform(self.machines.model)
92         dfOneHot = pd.DataFrame(encoded_models, columns = self.machines_encoder.classes_)
93         self.machines = pd.concat([self.machines, dfOneHot], axis=1)
94
95     merged_df = self.join_features()
96     self.clear_data()
97
98     return merged_df
99
100    def join_features(self,):
101        logger.info('transform: merge')
102
103        merged_features = (self.errors
104                            .merge(self.maintenance, how='outer')
105                            .merge(self.machines, how='outer')
106                            .merge(self.telemetry, how='outer'))
107
108        return merged_features
109
110    def unpack_data(self, X, fit=True):
111        try:
112            self.telemetry = X['telemetry']
113            self.errors = X['errors']
114            self.maintenance = X['maintenance']
115            if fit:
116                self.machines = X['machines']
117        except:
118            sys.exit('ERROR: X must a dict of dataframes with the follwong keys:\n(telemetry, errors,
119                         maintenance, machines)')
120
121    def clear_data(self,):
122        del self.telemetry
123        del self.errors
124        del self.maintenance
```

Annexe D

XGBoost - Bayesian Optimisation

La recherche a pris trois heures pour 18 itérations (faisant chacune un k-fold où $k = 3$).
Comme on peut le voir les meilleurs hyper-paramètres ont été trouvé dès la sixième itération.

FIGURE D.1 – Résultats de l’Optimisation Bayesienne pour XGBoost

iter	target	alpha	colsam...	gamma	learni...	max_depth	n_esti...	subsample
<hr/>								
n_jobs set to "5"								
1 0.9725	18.38	0.5643	8.586	0.9104	9.75	348.4	0.4494	
n_jobs set to "5"								
2 0.9907	12.69	0.3347	1.11	0.156	9.191	395.0	0.6995	
n_jobs set to "5"								
3 0.9882	16.43	0.9351	1.686	0.64	5.052	146.7	0.675	
n_jobs set to "5"								
4 0.9709	16.56	0.8475	0.7391	0.9915	3.445	146.7	0.6966	
n_jobs set to "5"								
5 0.9684	12.02	0.4158	9.537	0.9882	23.68	302.3	0.514	
n_jobs set to "5"								
6 0.9912	5.735	0.9789	2.483	0.1531	17.71	350.8	0.6785	
n_jobs set to "5"								
7 0.9902	10.01	0.6284	2.128	0.7271	17.01	447.0	0.5185	
n_jobs set to "5"								
8 0.9902	14.0	0.4037	1.245	0.126	9.543	277.6	0.769	
n_jobs set to "5"								
9 0.9848	9.329	0.3775	8.733	0.7541	24.94	207.6	0.8699	
n_jobs set to "5"								
10 0.9729	15.28	0.6789	5.226	0.8578	9.003	384.1	0.7687	
n_jobs set to "5"								
11 0.9905	15.54	0.3246	1.471	0.3574	21.64	421.7	0.5415	
n_jobs set to "5"								
12 0.9897	8.647	0.8623	9.938	0.5628	9.122	480.1	0.4115	
n_jobs set to "5"								
13 0.9899	9.42	0.8246	8.003	0.6364	23.71	135.5	0.366	
n_jobs set to "5"								
14 0.9899	11.32	0.4219	7.569	0.3426	21.34	267.6	0.6908	
n_jobs set to "5"								
15 0.9897	17.58	0.3241	4.305	0.1085	11.49	457.5	0.463	
n_jobs set to "5"								
16 0.9815	19.55	0.9405	8.764	0.01816	11.38	245.5	0.358	
n_jobs set to "5"								
17 0.9903	8.679	0.5309	3.054	0.4485	8.004	152.1	0.6475	
n_jobs set to "5"								
18 0.9895	17.52	0.8667	6.772	0.2107	9.291	302.4	0.5671	

Annexe E

Temps Computationnel de l'Apprentissage et des Prédictions

Classifier	Fit Time	Predict Time
Decision Tree	59 sec	0.6 sec
Random Forest	13 min 15	11 sec
XGBoost	31 min 55	32 sec
SVM	13 sec	0.7 sec
KNN	19 min	INFINITY
Logistic Regression	1 min 44	0.7 sec
Naive Bayes	2 sec	5 sec

Annexe F

Matrices de Confusion

FIGURE F.1 – Decision Tree - Test

CART: Confusion Matrix - Test							
		COMP1		COMP3			
Actual	False	Actual		Actual	False	Actual	
		False	True			False	True
Prediction		Prediction		Prediction		Prediction	
True	False	145,028 (TN)	198 (FP)	145,948 (TN)	72 (FP)	5,610 (TN)	505 (FP)
False	True	237 (FN)	1,537 (TP)	96 (FN)	884 (TP)	534 (FN)	140,351 (TP)
FNR: 13.36%	TPR: 86.64%	FPR: 0.14%	TNR: 99.86%	FNR: 9.80%	TPR: 90.20%	FNR: 0.38%	TPR: 99.62%
Actual	Actual	Actual	Actual	Actual	Actual	Actual	Actual
COMP1	COMP3	COMP4	NO FAILURE	COMP1	COMP3	COMP4	NO FAILURE
Actual	Actual	Actual	Actual	Actual	Actual	Actual	Actual
False	True	False	True	False	True	False	True
Prediction		Prediction		Prediction		Prediction	

FIGURE F.2 – Decision Tree - Train

CART: Confusion Matrix - Train							
		COMP1		COMP3			
Actual	False	Actual		Actual	False	Actual	
		False	True			False	True
Prediction		Prediction		Prediction		Prediction	
True	False	576,107 (TN)	341 (FP)	579,791 (TN)	176 (FP)	23,706 (TN)	1,640 (FP)
False	True	891 (FN)	6,661 (TP)	333 (FN)	3,700 (TP)	1,233 (FN)	557,421 (TP)
FNR: 11.80%	TPR: 88.20%	FPR: 0.06%	TNR: 99.94%	FNR: 8.26%	TPR: 91.74%	FNR: 0.22%	TPR: 99.78%
Actual	Actual	Actual	Actual	Actual	Actual	Actual	Actual
COMP1	COMP3	COMP4	NO FAILURE	COMP1	COMP3	COMP4	NO FAILURE
Actual	Actual	Actual	Actual	Actual	Actual	Actual	Actual
False	True	False	True	False	True	False	True
Prediction		Prediction		Prediction		Prediction	

FIGURE F.3 – Random Forest - Test

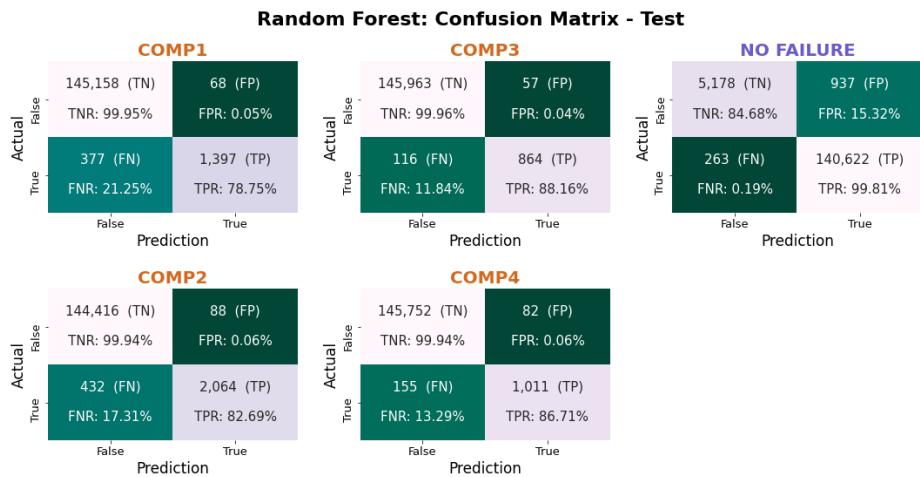


FIGURE F.4 – Random Forest - Train

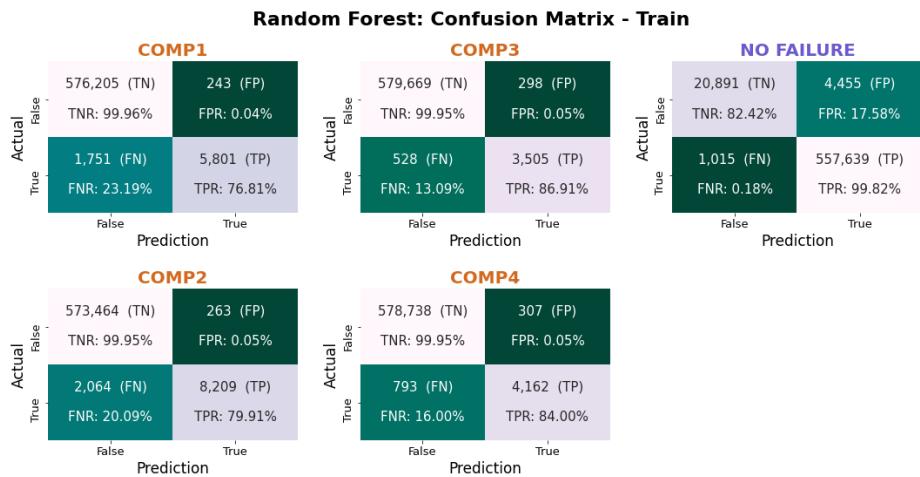


FIGURE F.5 – XGBoost - Test

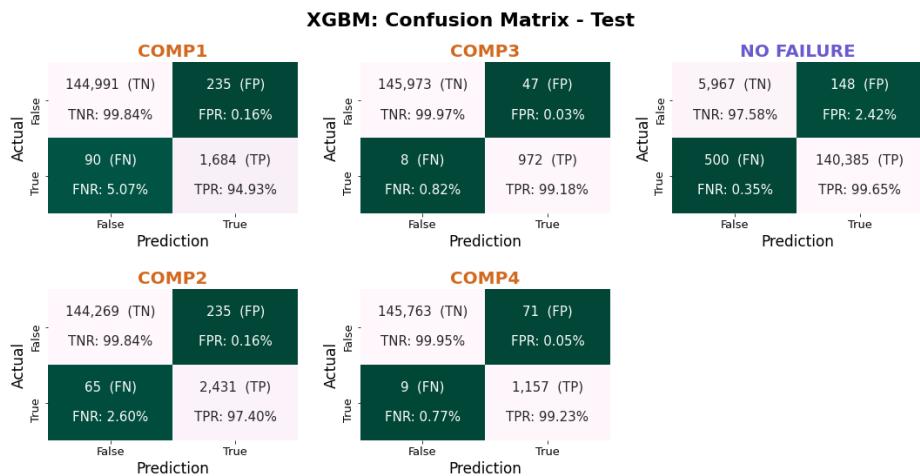


FIGURE F.6 – XGBoost - Train

XGBM: Confusion Matrix - Train							
		COMP1		COMP3		NO FAILURE	
Actual	False	COMP1		COMP3		NO FAILURE	
		576,066 (TN)	382 (FP)	579,838 (TN)	129 (FP)	25,346 (TN)	0 (FP)
	True	TNR: 99.93%	FPR: 0.07%	TNR: 99.98%	FPR: 0.02%	TNR: 100.00%	FPR: 0.00%
	False	0 (FN)	7,552 (TP)	0 (FN)	4,033 (TP)	949 (FN)	557,705 (TP)
	True	FNR: 0.00%	TPR: 100.00%	FNR: 0.00%	TPR: 100.00%	FNR: 0.17%	TPR: 99.83%
	False	Prediction		Prediction		Prediction	
COMP2						COMP4	
Actual	False	COMP2		COMP4		COMP4	
		573,315 (TN)	412 (FP)	578,867 (TN)	178 (FP)	5,953 (TN)	162 (FP)
	True	TNR: 99.93%	FPR: 0.07%	TNR: 99.97%	FPR: 0.03%	TNR: 97.35%	FPR: 2.65%
	False	0 (FN)	10,273 (TP)	0 (FN)	4,955 (TP)	483 (FN)	140,402 (TP)
	True	FNR: 0.00%	TPR: 100.00%	FNR: 0.00%	TPR: 100.00%	FNR: 0.34%	TPR: 99.66%
	False	Prediction		Prediction		Prediction	

FIGURE F.7 – XGBoost *Extended Windows* - Test

XGBM v2: Confusion Matrix - Test							
		COMP1		COMP3		NO FAILURE	
Actual	False	COMP1		COMP3		NO FAILURE	
		144,994 (TN)	232 (FP)	145,975 (TN)	45 (FP)	5,953 (TN)	162 (FP)
	True	TNR: 99.84%	FPR: 0.16%	TNR: 99.97%	FPR: 0.03%	TNR: 97.35%	FPR: 2.65%
	False	96 (FN)	1,678 (TP)	10 (FN)	970 (TP)	483 (FN)	140,402 (TP)
	True	FNR: 5.41%	TPR: 94.59%	FNR: 1.02%	TPR: 98.98%	FNR: 0.34%	TPR: 99.66%
	False	Prediction		Prediction		Prediction	
COMP2						COMP4	
Actual	False	COMP2		COMP4		COMP4	
		144,273 (TN)	231 (FP)	145,774 (TN)	60 (FP)	5,953 (TN)	162 (FP)
	True	TNR: 99.84%	FPR: 0.16%	TNR: 99.96%	FPR: 0.04%	TNR: 97.35%	FPR: 2.65%
	False	71 (FN)	2,425 (TP)	13 (FN)	1,153 (TP)	483 (FN)	140,402 (TP)
	True	FNR: 2.84%	TPR: 97.16%	FNR: 1.11%	TPR: 98.89%	FNR: 0.34%	TPR: 99.66%
	False	Prediction		Prediction		Prediction	

FIGURE F.8 – XGBoost *Extended Windows* - Train

XGBM v2: Confusion Matrix - Train							
		COMP1		COMP3		NO FAILURE	
Actual	False	COMP1		COMP3		NO FAILURE	
		576,217 (TN)	231 (FP)	579,878 (TN)	89 (FP)	25,346 (TN)	0 (FP)
	True	TNR: 99.96%	FPR: 0.04%	TNR: 99.98%	FPR: 0.02%	TNR: 100.00%	FPR: 0.00%
	False	0 (FN)	7,552 (TP)	0 (FN)	4,033 (TP)	610 (FN)	558,044 (TP)
	True	FNR: 0.00%	TPR: 100.00%	FNR: 0.00%	TPR: 100.00%	FNR: 0.11%	TPR: 99.89%
	False	Prediction		Prediction		Prediction	
COMP2						COMP4	
Actual	False	COMP2		COMP4		COMP4	
		573,457 (TN)	270 (FP)	578,916 (TN)	129 (FP)	5,953 (TN)	162 (FP)
	True	TNR: 99.95%	FPR: 0.05%	TNR: 99.98%	FPR: 0.02%	TNR: 97.35%	FPR: 2.65%
	False	0 (FN)	10,273 (TP)	0 (FN)	4,955 (TP)	483 (FN)	140,402 (TP)
	True	FNR: 0.00%	TPR: 100.00%	FNR: 0.00%	TPR: 100.00%	FNR: 0.34%	TPR: 99.66%
	False	Prediction		Prediction		Prediction	

FIGURE F.9 – Logistic Regression - Test

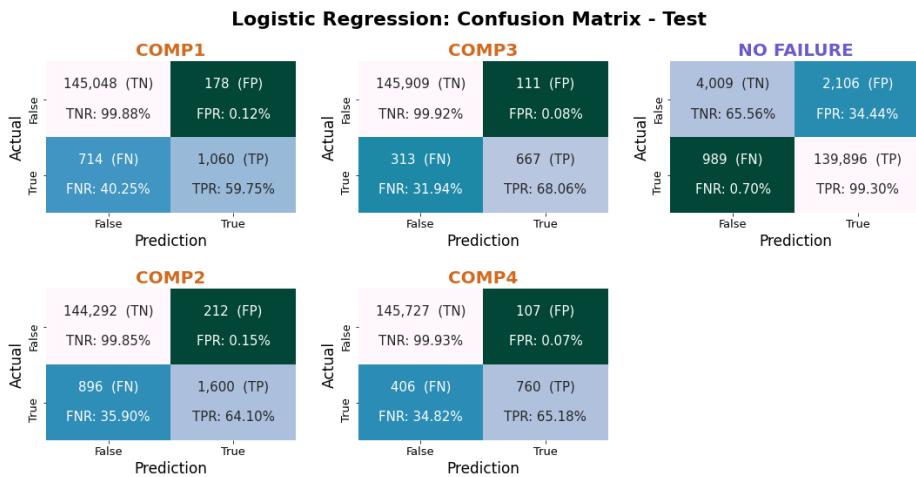


FIGURE F.10 – Logistic Regression - Train

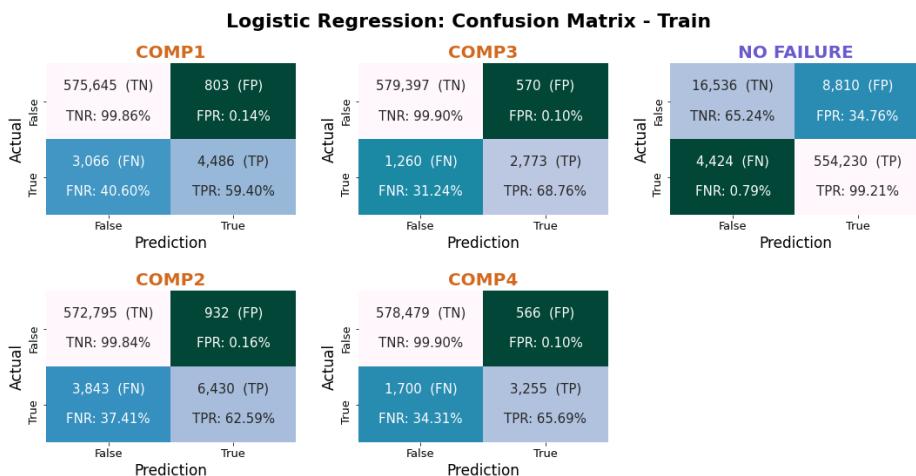


FIGURE F.11 – SVM - Test

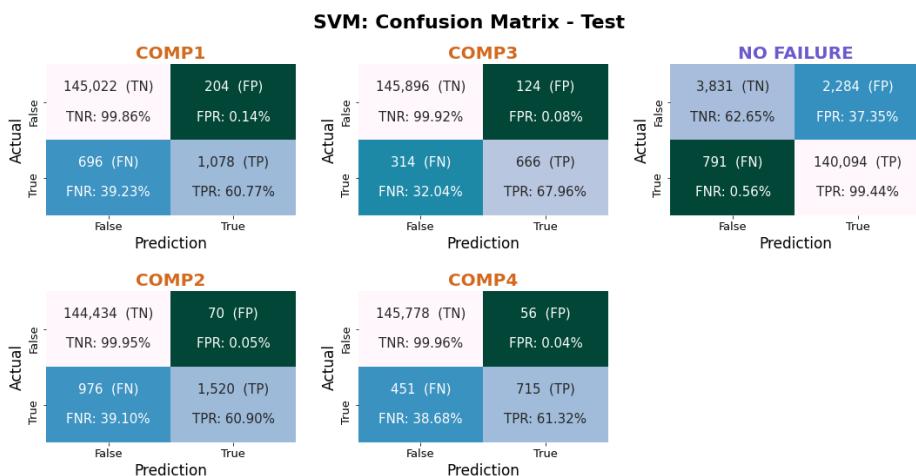


FIGURE F.12 – SVM - Train

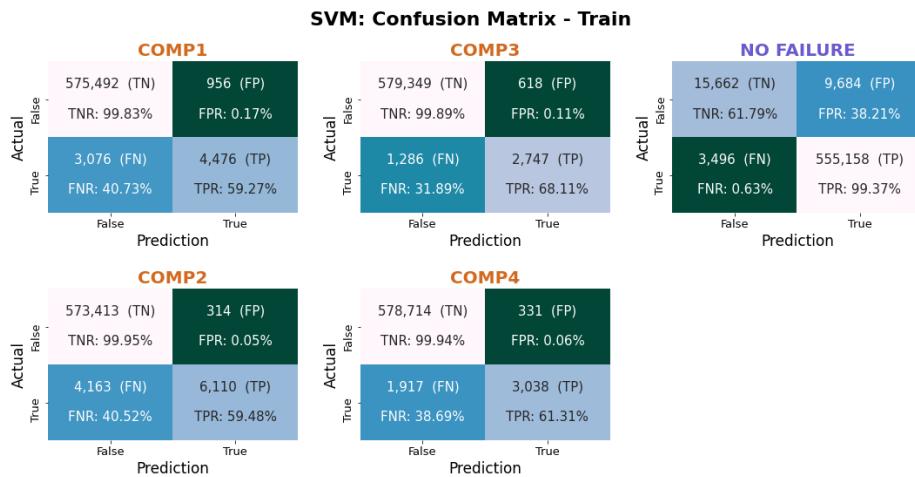


FIGURE F.13 – Naive Bayes - Test

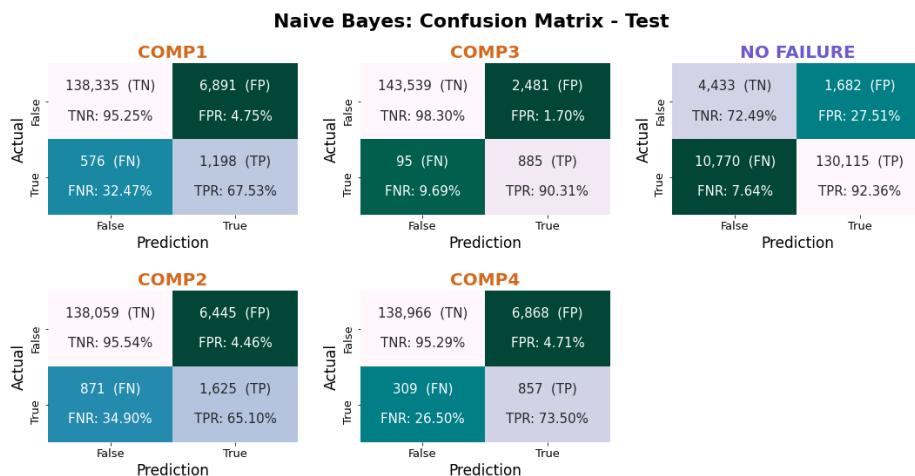
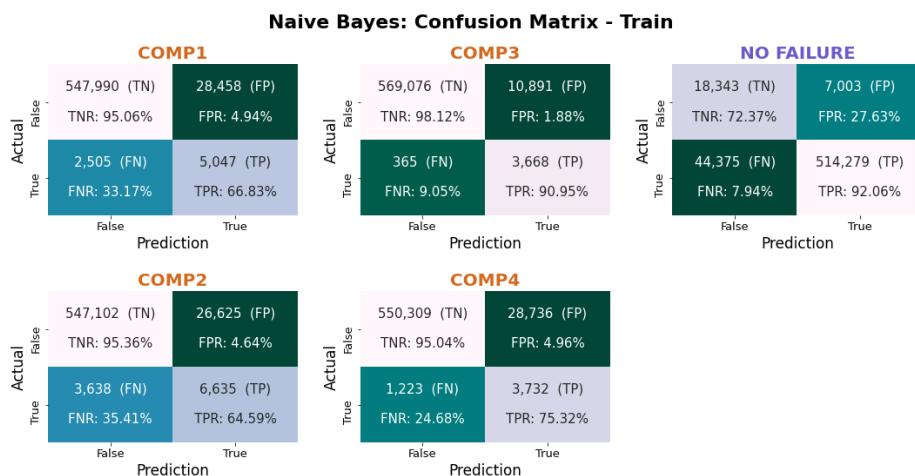


FIGURE F.14 – Naive Bayes - Train



Annexe G

Scores

FIGURE G.1 – Scores - Première Partie

NO FAILURE	accuracy	auc	f1-score	precision	recall
model					
Decision Tree	99.293	97.772	99.631	99.641	99.621
Logistic Regression	97.895	95.593	98.906	98.517	99.298
Naive Bayes	91.529	84.247	95.434	98.724	92.355
Random Forest	99.184	94.61	99.575	99.338	99.813
SVM	97.908	95.534	98.914	98.396	99.439
XGBoost	99.559	99.942	99.77	99.895	99.645

MACRO AVG	accuracy	auc	f1-score	precision	recall
model					
Decision Tree	99.171	97.408	92.67	93.79	91.586
Logistic Regression	97.797	98.428	78.844	89.166	71.279
Naive Bayes	88.874	93.936	42.098	34.211	77.759
Random Forest	99.107	95.318	91.011	95.383	87.225
SVM	97.867	98.379	78.589	91.024	70.077
XGBoost	99.495	99.973	95.813	93.688	98.076

(a) TEST

NO FAILURE	accuracy	auc	f1-score	precision	recall
model					
Decision Tree	99.508	99.959	99.743	99.707	99.779
Logistic Regression	97.734	95.184	98.82	98.435	99.208
Naive Bayes	91.202	83.903	95.242	98.657	92.057
Random Forest	99.063	94.488	99.512	99.207	99.818
SVM	97.743	95.142	98.827	98.286	99.374
XGBoost	99.838	99.998	99.915	100	99.83

MACRO AVG	accuracy	auc	f1-score	precision	recall
model					
Decision Tree	99.359	99.977	94.445	96.427	92.571
Logistic Regression	97.619	98.224	78.197	87.746	71.13
Naive Bayes	88.467	93.766	41.943	34.071	77.948
Random Forest	98.986	95.175	90.044	95.475	85.489
SVM	97.696	98.182	77.643	89.522	69.509
XGBoost	99.813	100	98.429	96.952	99.966

(b) TRAIN

FIGURE G.2 – Scores - Deuxième Partie

(a) TEST						(c) TRAIN							
COMP1		accuracy	auc	f1-score	precision	recall	COMP1		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.704	96.933	87.603	88.588	86.64		Decision Tree	99.789	99.969	91.535	95.13	88.202	
Logistic Regression	99.393	98.165	70.385	85.622	59.752		Logistic Regression	99.338	97.957	69.87	84.818	59.401	
Naive Bayes	94.92	94.119	24.293	14.81	67.531		Naive Bayes	94.698	93.936	24.585	15.063	66.83	
Random Forest	99.697	93.164	86.261	95.358	78.749		Random Forest	99.659	93.094	85.334	95.979	76.814	
SVM	99.388	98.059	70.55	84.087	60.767		SVM	99.31	97.905	68.946	82.401	59.269	
XGBoost	99.779	99.961	91.2	87.754	94.927		XGBoost	99.935	100	97.533	95.185	100	
COMP2		accuracy	auc	f1-score	precision	recall	COMP2		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.735	97.403	92.044	93.766	90.385		Decision Tree	99.796	99.981	94.058	96.222	91.989	
Logistic Regression	99.246	98.806	74.28	88.3	64.103		Logistic Regression	99.182	98.538	72.923	87.34	62.591	
Naive Bayes	95.023	95.371	30.759	20.136	65.104		Naive Bayes	94.818	95.285	30.483	19.949	64.587	
Random Forest	99.646	94.277	88.812	95.911	82.692		Random Forest	99.602	94.26	87.586	96.896	79.908	
SVM	99.288	98.729	74.4	95.597	60.897		SVM	99.233	98.448	73.187	95.112	59.476	
XGBoost	99.796	99.972	94.188	91.185	97.396		XGBoost	99.929	100	98.034	96.144	100	
COMP3		accuracy	auc	f1-score	precision	recall	COMP3		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.886	97.225	91.322	92.469	90.204		Decision Tree	99.913	99.991	93.564	95.459	91.743	
Logistic Regression	99.712	99.888	75.882	85.733	68.061		Logistic Regression	99.687	99.849	75.19	82.949	68.758	
Naive Bayes	98.248	99.056	40.727	26.292	90.306		Naive Bayes	98.073	98.967	39.458	25.194	90.95	
Random Forest	99.882	97.973	90.9	93.811	88.163		Random Forest	99.859	97.756	89.459	92.164	86.908	
SVM	99.702	99.901	75.254	84.304	67.959		SVM	99.674	99.852	74.263	81.634	68.113	
XGBoost	99.963	99.996	97.249	95.388	99.184		XGBoost	99.978	100	98.426	96.901	100	
COMP4		accuracy	auc	f1-score	precision	recall	COMP4		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.887	97.705	92.751	94.484	91.081		Decision Tree	99.889	99.988	93.325	95.617	91.14	
Logistic Regression	99.651	99.689	74.766	87.659	65.18		Logistic Regression	99.612	99.593	74.18	85.187	65.691	
Naive Bayes	95.118	96.888	19.278	11.094	73.499		Naive Bayes	94.87	96.74	19.945	11.494	75.318	
Random Forest	99.839	96.566	89.509	92.498	86.707		Random Forest	99.812	96.275	88.328	93.13	83.996	
SVM	99.655	99.674	73.826	92.737	61.321		SVM	99.615	99.563	72.994	90.175	61.312	
XGBoost	99.946	99.994	96.658	94.218	99.228		XGBoost	99.97	100	98.236	96.532	100	

(b) TEST						(d) TRAIN							
COMP1		accuracy	auc	f1-score	precision	recall	COMP1		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.704	96.933	87.603	88.588	86.64		Decision Tree	99.789	99.969	91.535	95.13	88.202	
Logistic Regression	99.393	98.165	70.385	85.622	59.752		Logistic Regression	99.338	97.957	69.87	84.818	59.401	
Naive Bayes	94.92	94.119	24.293	14.81	67.531		Naive Bayes	94.698	93.936	24.585	15.063	66.83	
Random Forest	99.697	93.164	86.261	95.358	78.749		Random Forest	99.659	93.094	85.334	95.979	76.814	
SVM	99.388	98.059	70.55	84.087	60.767		SVM	99.31	97.905	68.946	82.401	59.269	
XGBoost	99.779	99.961	91.2	87.754	94.927		XGBoost	99.935	100	97.533	95.185	100	
COMP2		accuracy	auc	f1-score	precision	recall	COMP2		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.735	97.403	92.044	93.766	90.385		Decision Tree	99.796	99.981	94.058	96.222	91.989	
Logistic Regression	99.246	98.806	74.28	88.3	64.103		Logistic Regression	99.182	98.538	72.923	87.34	62.591	
Naive Bayes	95.023	95.371	30.759	20.136	65.104		Naive Bayes	94.818	95.285	30.483	19.949	64.587	
Random Forest	99.646	94.277	88.812	95.911	82.692		Random Forest	99.602	94.26	87.586	96.896	79.908	
SVM	99.288	98.729	74.4	95.597	60.897		SVM	99.233	98.448	73.187	95.112	59.476	
XGBoost	99.796	99.972	94.188	91.185	97.396		XGBoost	99.929	100	98.034	96.144	100	
COMP3		accuracy	auc	f1-score	precision	recall	COMP3		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.886	97.225	91.322	92.469	90.204		Decision Tree	99.913	99.991	93.564	95.459	91.743	
Logistic Regression	99.712	99.888	75.882	85.733	68.061		Logistic Regression	99.687	99.849	75.19	82.949	68.758	
Naive Bayes	98.248	99.056	40.727	26.292	90.306		Naive Bayes	98.073	98.967	39.458	25.194	90.95	
Random Forest	99.882	97.973	90.9	93.811	88.163		Random Forest	99.859	97.756	89.459	92.164	86.908	
SVM	99.702	99.901	75.254	84.304	67.959		SVM	99.674	99.852	74.263	81.634	68.113	
XGBoost	99.963	99.996	97.249	95.388	99.184		XGBoost	99.978	100	98.426	96.901	100	
COMP4		accuracy	auc	f1-score	precision	recall	COMP4		accuracy	auc	f1-score	precision	recall
model							model						
Decision Tree	99.887	97.705	92.751	94.484	91.081		Decision Tree	99.889	99.988	93.325	95.617	91.14	
Logistic Regression	99.651	99.689	74.766	87.659	65.18		Logistic Regression	99.612	99.593	74.18	85.187	65.691	
Naive Bayes	95.118	96.888	19.278	11.094	73.499		Naive Bayes	94.87	96.74	19.945	11.494	75.318	
Random Forest	99.839	96.566	89.509	92.498	86.707		Random Forest	99.812	96.275	88.328	93.13	83.996	
SVM	99.655	99.674	73.826	92.737	61.321		SVM	99.615	99.563	72.994	90.175	61.312	
XGBoost	99.946	99.994	96.658	94.218	99.228		XGBoost	99.97	100	98.236	96.532	100	

	comp1		comp2		comp3		comp4		no failure		macro avg	
	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train
precision	87.853	97.032	91.303	97.439	95.567	97.841	95.054	97.463	99.885	100	93.932	97.955
recall	94.589	100	97.155	100	98.98	100	98.885	100	99.657	99.891	97.853	99.978
f1-score	91.097	98.494	94.138	98.703	97.243	98.909	96.931	98.715	99.771	99.945	95.836	98.953
auc	99.96	100	99.971	100	99.99	100	99.996	100	99.941	100	99.972	100
accuracy	99.777	99.96	99.795	99.954	99.963	99.985	99.95	99.978	99.561	99.896	99.494	99.877

FIGURE G.3 – Scores pour XGBoost avec Fenêtres Étendues

Annexe H

Feature Importance

H.1 PAR CATÉGORIE DE FEATURES

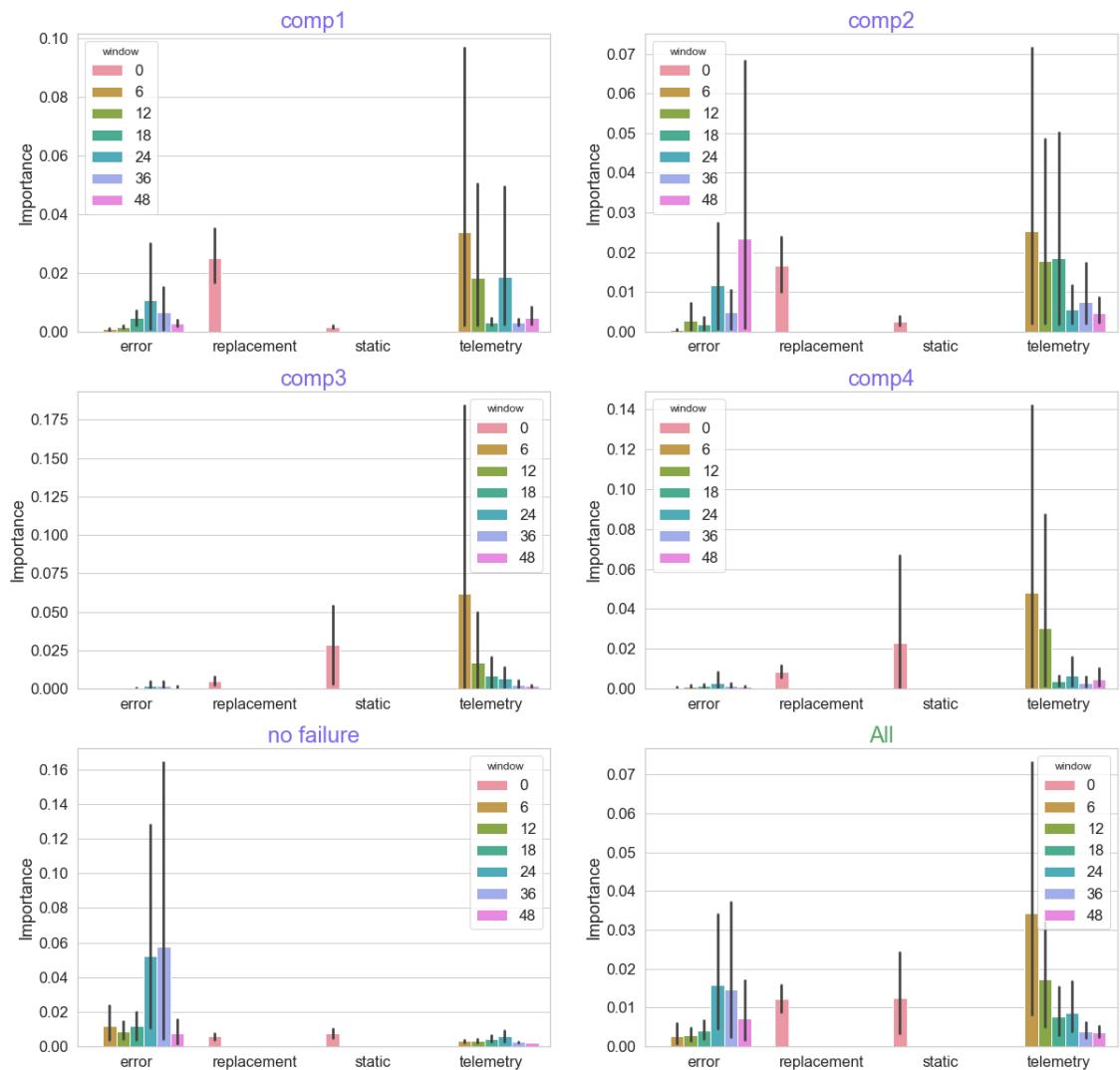


FIGURE H.1 – Feature Importance par Type et par Fenêtre

H.2 PAR SOUS-CATÉGORIE DE FEATURES

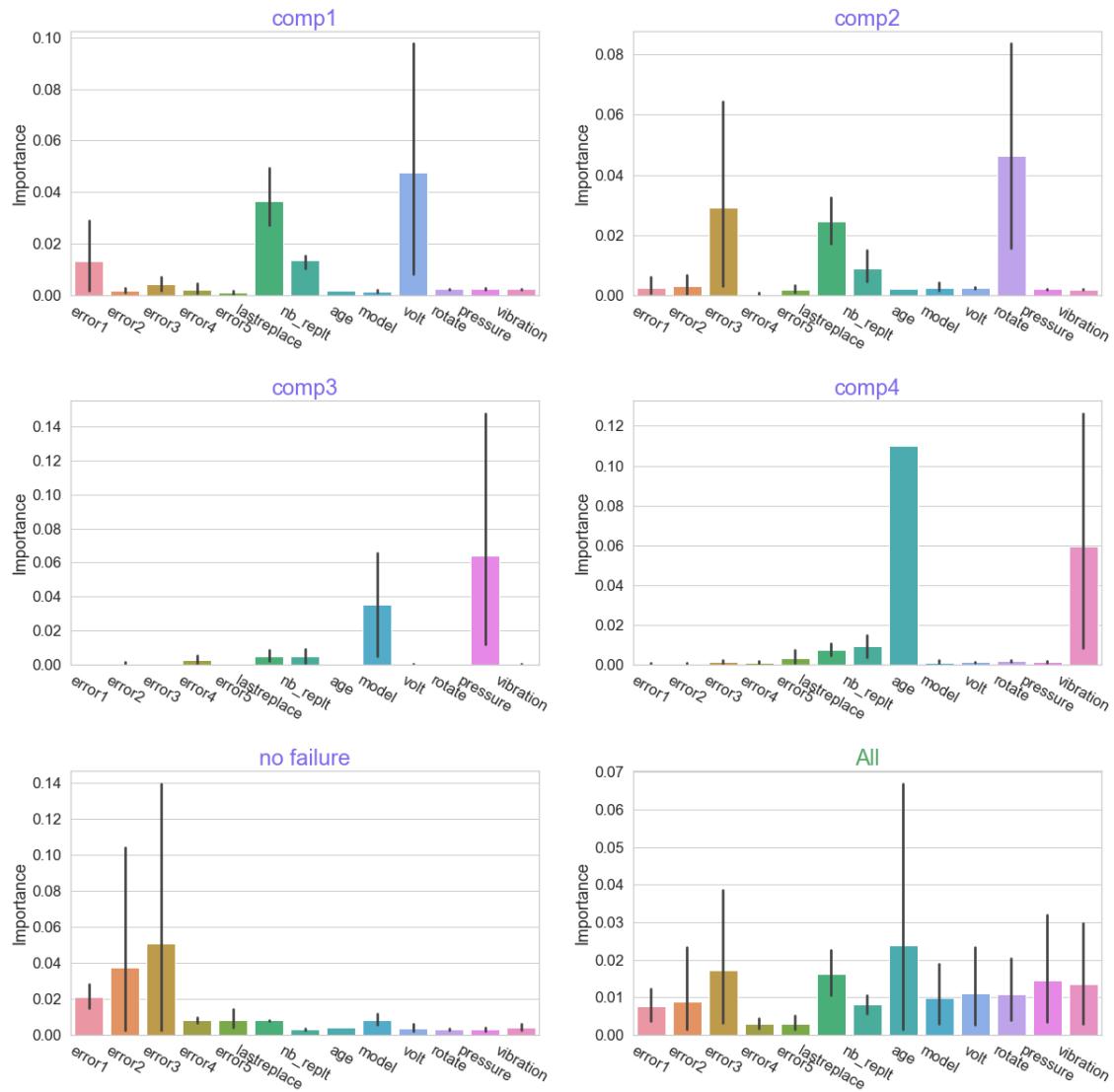


FIGURE H.2 – Feature Importance par Sous-Catégorie

Annexe I

Schéma Relationnel de la Base de Données

FIGURE I.1 – BDD Schéma Relationnel - Première Partie

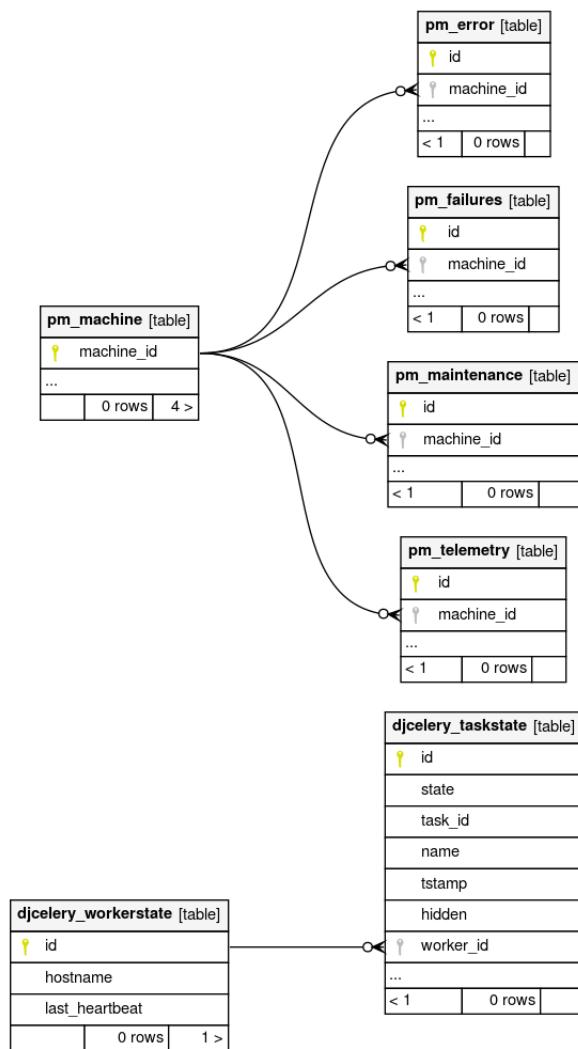
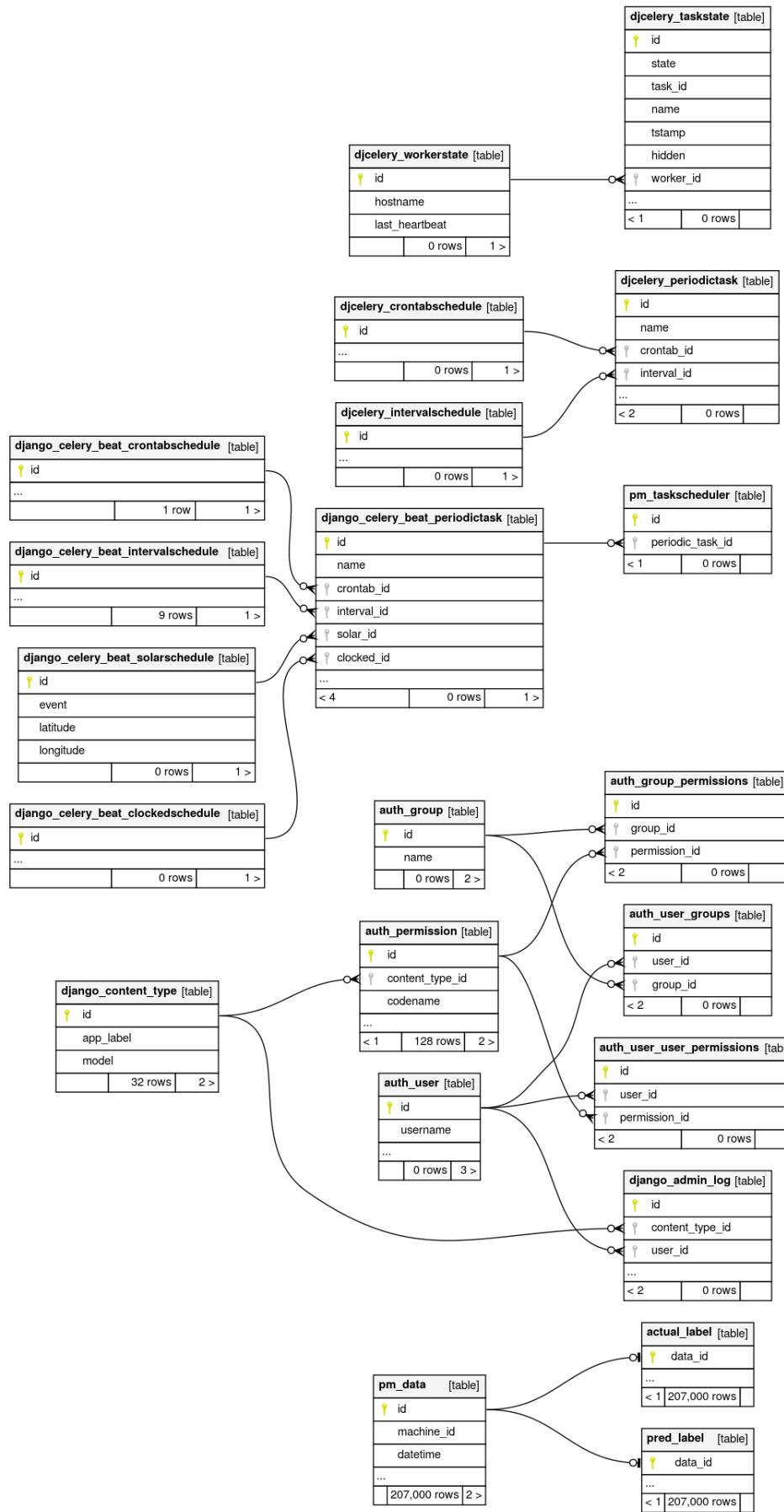


FIGURE I.2 – BDD Schéma Relationnel - Seconde Partie



Annexe J

Application Django

J.1 MODELS

Les données

```
1 class DataManager(models.Manager):
2     def reset(self):
3         self.all().update(
4             is_sent = False,
5             is_retrieved = False,
6             is_preprocess = False,
7             is_predicted = False,
8             alert = 'G')
9
10    Prediction.objects.all().update(
11        is_failing = False,
12        comp1 = False,
13        comp2 = False,
14        comp3 = False,
15        comp4 = False)
16
17    def count_alert(self, alert, dt=None):
18        if dt is None:
19            return 0
20        # dt = self.values_list('datetime',
21        # flat=True).filter(is_retrieved=True).order_by('-datetime').first()
21        else:
22            # dt = datetime.strptime(dt, "%Y-%m-%d %H:%M") if type(dt) == str else dt
23            dt = self.reverse_date(dt)
24            return self.filter(alert=alert, datetime=dt, is_predicted=True).count()
25
26    def get_failures(self, alert, dt):
27        if dt is None:
28            return None
29
30        dt = self.reverse_date(dt)
31        q = self.filter(alert=alert, datetime=dt, is_predicted=True).values('machine_id',
32                           'prediction_comp1', 'prediction_comp2', 'prediction_comp3', 'prediction_comp4')
33
34        if q.count() == 0:
35            return None
36        else:
37            df = pd.DataFrame.from_records(q).set_index('machine_id')
38            # print(f'{alert} count {df.shape}')
39            df = df[df==1].stack().reset_index().drop(0,1).rename(columns={'level_1':'comp'})
40            # print(f'{alert} count {df.shape}')
41            df.comp = df.comp.apply(lambda x: x.replace('prediction_', ''))
42            # print(f'{alert} count {df.shape}')
43            return loads(df.sort_values(['machine_id', 'comp']).transpose().to_json())
44
45    def total_prod(self):
46        return self.filter(is_sent=True).count()
47
48    def get_datetimes(self):
49        query = self.values_list('datetime',
50                               flat=True).filter(is_predicted=True).distinct().order_by('datetime')
```

```

50     return [self.format_datetime(rst) for rst in query]
51
52 def get_machine_ids(self):
53     query = self.values_list('machine_id',
54         flat=True).filter(is_retrieved=True).distinct().order_by('machine_id')
55     return query
56
57 def get_chart_data(self):
58     query = self.values('datetime').filter(is_predicted=True).order_by('datetime').annotate(
59         real_fail=(Count('actual_is_failing', filter=Q(actual_is_failing=True))),
60         pred_fail=(Count('prediction_is_failing', filter=Q(prediction_is_failing=True))))
61
62     # real_fail=(Sum(Case(When(actual_is_failing=True, then=1), default=Value(0),
63     #     output_field=IntegerField(), filter=Q(actual_is_failing=True))),
64     # pred_fail=(Sum(Case(When(prediction_is_failing=True, then=1), default=Value(0),
65     #     output_field=IntegerField(), filter=Q(prediction_is_failing=True))),
66
66     datetimes = [self.format_datetime(rst['datetime']) for rst in query]
67     real_data = [rst['real_fail'] for rst in query]
68     pred_data = [rst['pred_fail'] for rst in query]
69
70     return datetimes, real_data, pred_data
71
72 @staticmethod
73 def reverse_date(dt):
74     return datetime.strptime(dt, "%d %b %Y:%H")
75
76 @staticmethod
77 def format_datetime(dt):
78     return f'{dt.day} {dt.strftime("%b")}{dt.year}:{dt.hour}h'
79
80 class Data(models.Model):
81     ALERTS = [
82         ('G', 'green'),
83         ('O', 'orange'),
84         ('R', 'red'),
85     ]
86     machine_id = models.IntegerField()
87     datetime = models.DateTimeField()
88     is_sent = models.BooleanField(default=False)
89     is_retrieved = models.BooleanField(default=False)
90     is_preprocess = models.BooleanField(default=False)
91     is_predicted = models.BooleanField(default=False)
92
93     alert = models.CharField(
94         max_length=1,
95         choices=ALERTS,
96         default='G')
97
98     objects = DataManager()
99
100    class Meta:
101        unique_together = (("machine_id", "datetime"),)
102
103    def set_alert(self,):
104        dt = self.datetime if type(self.datetime) != str else datetime.strptime(self.datetime,
105            "%Y-%m-%d %H:%M:%S")
106        self.alert = 'G'
107        if self.prediction.is_failing:
108            i = Data.objects.filter(datetime__range=(dt - timedelta(days=2), self.datetime),
109                machine_id=self.machine_id, prediction_is_failing=True).count()
110            self.alert = 'G' if i == 0 else ('R' if i >= 3 else 'O')
111        self.save()
112
113 class Label(models.Model):
114     is_failing = models.BooleanField(default=False)
115     comp1 = models.BooleanField(default=False)
116     comp2 = models.BooleanField(default=False)
117     comp3 = models.BooleanField(default=False)
118     comp4 = models.BooleanField(default=False)
119
120     class Meta:
121         abstract = True
122
123     def reset(self,):
124         self.is_failing = False
125         self.comp1 = False
126         self.comp2 = False
127         self.comp3 = False
128         self.comp4 = False

```

```

128     self.save()
129
130     def set_failure(self):
131         self.is_failing = self.comp1 or self.comp2 or self.comp3 or self.comp4
132         self.save()
133
134 class ActualManager(models.Manager):
135     def initialize(self, mindate=None, maxdate=None):
136         if self.all().count() >= 183000:
137             logger.info("Actual Data already initialized")
138             sys.exit()
139
140         logger.info("Initializing Actual Data..")
141
142         df = pd.read_csv(os.path.join(settings.BASE_DIR,
143             "data/stream_samples/stream_data_labels.csv"))
144         if mindate and maxdate:
145             df = df[(df.datetime >= mindate) & (df.datetime <= maxdate)]
146
147         j=1
148         for _, row in df.iterrows():
149             d, is_created = Data.objects.get_or_create(datetime=row.datetime, machine_id=row.machineID)
150
151             if is_created:
152                 d.save()
153             d.actual.is_failing = row.is_failing
154             d.actual.comp1 = row.comp1
155             d.actual.comp2 = row.comp2
156             d.actual.comp3 = row.comp3
157             d.actual.comp4 = row.comp4
158             d.actual.save()
159             j+=1
160
161         logger.info("Initializing Actual Data -- DONE")
162
163 class Actual(Label):
164     data = models.OneToOneField(
165         Data,
166         on_delete=models.CASCADE,
167         primary_key=True)
168
169     objects = ActualManager()
170
171     class Meta(Label.Meta):
172         db_table = 'actual_label'
173
174
175 class Prediction(Label):
176     data = models.OneToOneField(
177         Data,
178         on_delete=models.CASCADE,
179         primary_key=True)
180     proba = models.FloatField(null=True)
181
182     class Meta(Label.Meta):
183         db_table = 'pred_label'
184
185
186 class PartitionManager(models.Manager):
187     def reset(self):
188         self.all().update(
189             prod = False,
190             cons = False,
191             task = None,
192             topic = None)
193
194     def to_consume(self):
195         return self.filter(prod=True, cons=False).order_by('id')
196
197     def is_all_done(self):
198         return (self.filter(prod=True).count() > 0) and (self.filter(cons=True).count() ==
199             self.filter(prod=True).count())
200
201 class Partition(models.Model):
202     id = models.IntegerField(primary_key=True)
203     prod = models.BooleanField(default=False)
204     cons = models.BooleanField(default=False)
205     task = models.CharField(max_length=50, null=True)
206     topic = models.CharField(max_length=6, null=True)
207
208     objects = PartitionManager()

```

```

209
210 class MachineManager(models.Manager):
211     def initialize(self):
212         if self.all().count() >= 1000:
213             logger.info("Machines already initialized")
214         else:
215             logger.info("Initializing Machines..")
216             df = pd.read_csv(os.path.join(settings.BASE_DIR, "data/raw/machines.csv"))
217
218             j=1
219             for _, row in df.iterrows():
220                 m = Machine.objects.create(machine_id=row.machineID, model=row.model, age=row.age)
221                 m.save()
222                 j+=1
223
224             logger.info(f"Initializing Actual Data -- DONE {Machine.objects.all().count()} machines")
225
226
227 class Machine(models.Model):
228     MODELS = [
229         ('model1', 'model1'),
230         ('model2', 'model2'),
231         ('model3', 'model3'),
232         ('model4', 'model4')]
233
234     machine_id = models.IntegerField(primary_key=True)
235     model = models.CharField(choices=MODELS, max_length=6)
236     age = models.IntegerField()
237
238     objects = MachineManager()
239
240
241 class Telemetry(models.Model):
242     machine = models.ForeignKey(Machine, on_delete=models.CASCADE) # verbose_name="machine_id",
243     datetime = models.DateTimeField()
244     volt = models.FloatField()
245     rotate = models.FloatField()
246     pressure = models.FloatField()
247     vibration = models.FloatField()
248
249     def validate_unique(self, *args, **kwargs):
250         super(Telemetry, self).validate_unique(*args, **kwargs)
251
252         if self.__class__.objects.\
253             filter(fk=self.machine, my_field=self.datetime).\
254             exists():
255             raise ValidationError(
256                 message='Telemetry with this (machine_id, datetime) already exists.',
257                 code='unique_together')
258
259
260 class Error(models.Model):
261     ERRORS = [
262         ('error1', 'error1'),
263         ('error2', 'error2'),
264         ('error3', 'error3'),
265         ('error4', 'error4'),
266         ('error5', 'error5')]
267
268     machine = models.ForeignKey(Machine, verbose_name="machine", on_delete=models.CASCADE)
269     datetime = models.DateTimeField()
270     error_id = models.CharField(choices=ERRORS, max_length=6)
271
272     def validate_unique(self, *args, **kwargs):
273         super(Error, self).validate_unique(*args, **kwargs)
274
275         if self.__class__.objects.\
276             filter(fk=self.machine, my_field=self.datetime).\
277             exists():
278             raise ValidationError(
279                 message='Error with this (machine_id, datetime) already exists.',
280                 code='unique_together')
281
282
283 class Maintenance(models.Model):
284     COMPS = [
285         ('comp1', 'comp1'),
286         ('comp2', 'comp2'),
287         ('comp3', 'comp3'),
288         ('comp4', 'comp4')]
289
290     machine = models.ForeignKey(Machine, verbose_name="machine", on_delete=models.CASCADE)
291     datetime = models.DateTimeField()

```

```

292     comp_id = models.CharField(choices=COMPS, max_length=5)
293
294     def validate_unique(self, *args, **kwargs):
295         super(Maintenance, self).validate_unique(*args, **kwargs)
296
297         if self.__class__.objects.\n            filter(fk=self.machine, my_field=self.datetime).\n            exists():
298             raise ValidationError(\n                message='Maintenance with this (machine_id, datetime) already exists.',\n                code='unique_together')
299
300
301
302
303
304
305     class Failures(models.Model):
306         COMPS = [\n            ('comp1', 'comp1'),\n            ('comp2', 'comp2'),\n            ('comp3', 'comp3'),\n            ('comp4', 'comp4')]
307
308         machine = models.ForeignKey(Machine, verbose_name="machine", on_delete=models.CASCADE)
309         datetime = models.DateTimeField()
310         comp_id = models.CharField(choices=COMPS, max_length=5)
311
312         def validate_unique(self, *args, **kwargs):
313             super(Failures, self).validate_unique(*args, **kwargs)
314
315             if self.__class__.objects.\n                filter(fk=self.machine, my_field=self.datetime).\n                exists():
316                 raise ValidationError(\n                     message='Failures with this (machine_id, datetime) already exists.',\n                     code='unique_together')
317
318
319
320
321
322
323
324

```

Le Task Scheduler

```

1  class TaskManager(models.Manager):
2      def delete_all(self):
3          p = PeriodicTask.objects.all().delete()
4          d = self.all().delete()
5          return p, d
6
7
8  class TaskScheduler(models.Model):
9      id = models.AutoField(primary_key=True, auto_created=True)
10     periodic_task = models.ForeignKey(PeriodicTask, on_delete=models.CASCADE, verbose_name="the\n        related periodic task", default=None, null=True)
11
12    objects = TaskManager()
13
14    def set_periodic_task(self, periodic_task):
15        self.periodic_task = periodic_task
16        self.save()
17        return self
18
19    @staticmethod
20    def schedule_every(task_name, period, every, args=None, kwargs=None):
21        """
22        schedules a task by name every "every" "period". So an example call would be:\n            TaskScheduler('mycustomtask', 'seconds', 30, [1,2,3])\n            that would schedule your custom task to run every 30 seconds with the arguments 1,2 and\n            3 passed to the actual task.\n        """
23        permissible_periods = ['days', 'hours', 'minutes', 'seconds']
24        if period not in permissible_periods:
25            raise Exception('Invalid period specified')
26        # create the periodic task and the interval
27        ptask_name = "%s_%s" % (task_name, datetime.now()) # create some name for the period task
28        interval_schedules = IntervalSchedule.objects.filter(period=period, every=every)
29        if interval_schedules:
30            # just check if interval schedules exist like that already and reuse em
31            interval_schedule = interval_schedules[0]
32        else:
33            # create a brand new interval schedule
34            interval_schedule = IntervalSchedule()
35
36            interval_schedule.every = every
37            interval_schedule.period = period
38            interval_schedule.save()
39
40
41

```

```

42     PeriodicTask.objects.all().delete()
43
44     ptask = PeriodicTask(id=0, name=ptask_name, task=task_name, interval=interval_schedule)
45
46     if args:
47         ptask.args = args
48     if kwargs:
49         ptask.kwargs = kwargs
50     ptask.save()
51
52
53     return TaskScheduler.objects.create().set_periodic_task(ptask)
54
55 def start(self):
56     """starts the task"""
57     ptask = self.periodic_task
58     ptask.enabled = True
59     ptask.save()
60
61 def terminate(self):
62     ptask = self.periodic_task
63     if ptask is not None:
64         ptask.id = 0
65         ptask.save()
66         ptask.delete()
67     self.delete()

```

J.2 VIEWS

`web/pm/views.py`

Les Vues Principales

Code Sample J.1 – Dashboard - Home Page

```

1  from django.shortcuts import render, redirect
2  from django.http import JsonResponse, HttpResponseRedirect, Http404
3  from django.conf import settings
4  from django.core.serializers.json import DjangoJSONEncoder
5  from django.views.decorators.http import require_GET, require_http_methods
6  from django.template import Context
7  from django.template.loader import get_template
8  from pm.tasks import run_producer_task, create_periodic_task
9  from pm.models import Data, Partition, Error, Telemetry, Maintenance, Machine
10 from .forms import Intervals
11 import os
12 import logging
13 from json import dumps, loads
14
15
16 logger = logging.getLogger(__name__)
17
18 @require_GET
19 def dashboard(request):
20     errors = None
21
22     if 'form_errors' in request.session:
23         errors = request.session['form_errors']
24         request.session['form_errors'] = None
25
26     form = Intervals()
27
28     datetimes = Data.objects.get_datetimes()
29
30     if len(datetimes) > 0:
31         current_date = datetimes[0]
32         rd_data = Data.objects.get_failures('R', current_date)
33         or_data = Data.objects.get_failures('O', current_date)
34
35     else:
36         current_date, rd_data, or_data = None, None, None
37
38     request.session['current_date'] = current_date
39
40     return render(request = request,

```

```

41     template_name='dashboard.html',
42     context = {
43         "red": Data.objects.count_alert("R", current_date),
44         "green": Data.objects.count_alert("G", current_date),
45         "orange": Data.objects.count_alert("O", current_date),
46         "rd_data":rd_data,
47         "or_data":or_data,
48         "current_date":current_date,
49         'data':collect_chart_data(),
50         'form':form,
51         'async_id': None,
52         "datetimes":datetimes,
53         "form_errors":errors,
54     }
55 )
56 # [...]

```

Code Sample J.2 – Start Stream - Run Simulation

```

1 # [...]
2 @require_http_methods(['GET', 'POST'])
3 def start_stream(request):
4     form = Intervals(request.POST or None)
5     if request.POST:
6         if form.is_valid():
7             temp = form.cleaned_data.get("number")
8
9             logger.info("Resetting db data")
10
11            Partition.objects.all().delete()
12            Data.objects.all().delete()
13            Error.objects.all().delete()
14            Maintenance.objects.all().delete()
15            Telemetry.objects.all().delete()
16
17            Machine.objects.initialize()
18
19            logger.info("Resetting db data : DONE -- Now deleting lockfiles..")
20            for filename in os.listdir(settings.BASE_DIR + "/data"):
21                if filename.startswith('pipeline.unlock'):
22                    os.remove(settings.LOCKFILE_PATH.replace('pipeline.unlock', filename))
23
24            logger.info("deleting lockfiles : DONE -- Now running_producer_task..")
25
26            astask = run_producer_task.apply_async([temp])
27
28            logger.info("Running producer task : DONE -- Now creating periodic task..")
29
30            create_periodic_task.apply_async()
31
32            logger.info("Creating periodic task : DONE -- Now rendering..")
33
34            return render(request = request,
35                         template_name='dashboard.html',
36                         context = {
37                             'async_id': astask.task_id,
38                             "form": form,
39                         },
40                     )
41     else:
42         request.session['form_errors'] = form.errors
43
44     return redirect('/')

```

Les vues des éléments dynamiques de la page

Code Sample J.3 – Raw Data Tables

```

1 @require_GET
2 def telemetry_view(request):
3     if 'machine_id' in request.session:
4         id = request.session['machine_id']
5     else:
6         id = 1
7     telemetry = loads(dumps(list(Telemetry.objects.filter(machine_id=id).values('machine_id',
8         'datetime', 'volt', 'rotate', 'pressure', 'vibration'))), cls=DjangoJSONEncoder)

```

```

8
9     return render(request = request,
10                 template_name='telemetry_table.html',
11                 context = {
12                     "telemetry":telemetry,
13                 }
14             )
15
16
17 @require_GET
18 def error_view(request):
19     if 'machine_id' in request.session:
20         id = request.session['machine_id']
21     else:
22         id = 1
23     error = loads(dumps(list(Error.objects.filter(machine_id=id).values('machine_id', 'datetime',
24                             'error_id'))), cls=DjangoJSONEncoder))
25
26     return render(request = request,
27                 template_name='error_table.html',
28                 context = {
29                     "error":error,
30                 }
31             )
32
33 @require_GET
34 def maintenance_view(request):
35     if 'machine_id' in request.session:
36         id = request.session['machine_id']
37     else:
38         id = 1
39     maintenance = loads(dumps(list(Maintenance.objects.filter(machine_id=id).values('machine_id',
40                             'datetime', 'comp_id'))), cls=DjangoJSONEncoder))
41
42     return render(request = request,
43                 template_name='maintenance_table.html',
44                 context = {
45                     "maintenance":maintenance,
46                 }
47             )

```

Code Sample J.4 – Error Handlers

```

1 @require_GET
2 def handler404(request, *args, **argv):
3     template = get_template("404.html")
4     ctx = Context({})
5     return HttpResponseNotFound(template.render(ctx))
6
7 @require_GET
8 def handler500(request, *args, **argv):
9     template = get_template("500.html")
10    ctx = Context({})
11    return HttpResponseServerError(template.render(ctx))

```

Code Sample J.5 – Alerts Section

```

1 @require_GET
2 def update_date(request):
3     rdate = request.GET['date']
4
5     datetimes = Data.objects.get_datetimes()
6     if len(datetimes) > 0:
7         if rdate == 'auto':
8             rdate = Data.objects.get_datetimes()[-1]
9
10        request.session['current_date'] = rdate
11
12        return JsonResponse({'current_date':request.session['current_date'],
13                            "red": Data.objects.count_alert("R", request.session['current_date']),
14                            "orange": Data.objects.count_alert("O", request.session['current_date']),
15                            "datetimes":datetimes,
16                            })
17    else:
18        return JsonResponse({})
19
20 @require_GET

```

```

22 def update_mid(request):
23     request.session['machine_id'] = request.GET['machine_id']
24     return JsonResponse({'machine_id': request.session['machine_id']})
25
26
27 @require_GET
28 def get_tables(request, alert):
29     current_date = request.session.get('current_date')
30     data = Data.objects.get_failures(alert, current_date)
31
32     return render(request = request,
33                 template_name='alert_cards.html',
34                 context = {
35                     "or_data":data,
36                     "current_date":current_date,
37                 }
38             )
39
40
41 @require_GET
42 def orange_alert(request):
43     return get_tables(request, 'O')
44
45
46 @require_GET
47 def red_alert(request):
48     return get_tables(request, 'R')
49
50
51 @require_GET
52 def alerts(request):
53     return render(request = request,
54                 template_name='alerts.html',
55                 context = {
56                     "red": Data.objects.count_alert("G"),
57                     "dates_list": [],
58                     "actual_values": [],
59                     "predicted_values": []
56                 }
57             )
58
59
60
61

```

Code Sample J.6 – Chart & Progress Bar

```

1 @require_GET
2 def main_chart_view(request):
3     return render(request = request,
4                 template_name='main_chart.html',
5                 context = {'data':collect_chart_data()})
6
7
8
9 @require_GET
10 def progressbar_view(request):
11     total = Data.objects.total_prod()
12     predi = round(Data.objects.filter(is_predicted=True).count() * 100 / total, 1) if total > 0 else 0
13     prep = round(Data.objects.filter(is_preprocess=True).count() * 100 / total, 1) if total > 0 else 0
14     cons = round(Data.objects.filter(is_retrieved=True).count() * 100 / total, 1) if total > 0 else 0
15
16     return render(request = request,
17                 template_name='progressbar.html',
18                 context = {
19                     "cons": {'message':get_message(cons, "Extraction"), "width":cons/3, 'percent':cons},
20                     "prep": {'message':get_message(prep, "Preprocessing"), "width":prep/3,
21                             'percent':prep},
22                     "predi": {'message':get_message(predi, "Prediction"), "width":predi/3,
23                               'percent':predi},
24                 }
25
26
27 def collect_chart_data():
28     datetimes, real_data, pred_data = Data.objects.get_chart_data()
29
30     context = {"datetimes": datetimes,
31               "real_data":real_data,
32               "pred_data":pred_data,
33               }
34
35     return dumps(context)
36

```

```

37 def get_message(percent, step):
38     if percent < 5:
39         return ""
40     elif percent < 30:
41         return f"{percent}%""
42     else:
43         return f"{step}: {percent}%""
44
45
46 def get_color(percent):
47     if percent < 20:
48         return "bg-danger"
49     elif percent < 40:
50         return "bg-warning"
51     elif percent < 60:
52         return "bg-info"
53     elif percent < 80:
54         return "bg-success"
55     else:
56         return ""

```

J.3 TESTS UNITAIRES

Modèles

Code Sample J.7 – web pm/tests/test_models.py

```

1 from django.test import TestCase
2 from datetime import datetime, timedelta
3 from pm.models import Data, Actual, Prediction, Partition
4
5
6 class DataTestCase(TestCase):
7     date_now = datetime(2020, 12, 12, 8)
8
9     @classmethod
10     def setUpTestData(cls):
11         d, _ = Data.objects.get_or_create(datetime=cls.date_now, machine_id=1)
12         d.is_predicted = True
13         d.save()
14         cls.__set_data(cls, 2, [0, 1])
15         cls.__set_data(cls, 3, [0, 1, 2])
16
17
18     def __set_data(self, machine_id, list_delta_days=[0]):
19         for day in list_delta_days:
20             d, _ = Data.objects.get_or_create(datetime=self.date_now - timedelta(days=day),
21                                             machine_id=machine_id)
22             d.prediction.is_failing = True
23             d.prediction.comp3 = True
24             d.prediction.save()
25             d.is_predicted = True
26             d.is_retrieved = True
27             d.save()
28             # d.set_alert()
29
30     def __set_alert(self,):
31         for id in [1, 2, 3]:
32             objects = Data.objects.filter(machine_id=id)
33             for o in objects:
34                 o.set_alert()
35
36
37     def test_set_alert(self,):
38         self.__set_alert()
39
40         machine_green = Data.objects.get(machine_id=1)
41         machine_orange = Data.objects.get(machine_id=2, datetime=self.date_now)
42         machine_red = Data.objects.get(machine_id=3, datetime=self.date_now)
43
44         self.assertEqual(machine_green.alert, "G", f"machine with id={machine_green.machine_id}\
45             should have alert set to 'G' and not '{machine_green.alert}'")
46         self.assertEqual(machine_orange.alert, "O", f"machine with id={machine_orange.machine_id}\
47             should have alert set to 'O' and not '{machine_orange.alert}'")
48         self.assertEqual(machine_red.alert, "R", f"machine with id={machine_red.machine_id} should\
49             have alert set to 'R' and not '{machine_red.alert}'")

```

```

47
48
49     def test_total_prod(self,):
50         data = Data.objects.get(machine_id=1)
51         data.is_sent = True
52         data.save()
53
54         self.assertEqual(Data.objects.total_prod(), 1, f"Number of sent rows from producer should be
55                         1")
56
57     def test_format_datetime(self,):
58         self.assertEqual("17 Oct 2021:11h", Data.objects.format_datetime(datetime(2021, 10, 17, 11,
59                         00, 24, 818827)), f"string should be cast to datetime")
60
61     def test_reverse_date(self,):
62         self.assertEqual(Data.objects.reverse_date("17 Oct 2021:11h"), datetime(2021, 10, 17, 11),
63                         f"datetime should be formatted to string")
64
65     def test_get_datetimes(self,):
66         datetimes = [Data.objects.format_datetime(self.date_now),
67                         Data.objects.format_datetime(self.date_now - timedelta(days=1)),
68                         Data.objects.format_datetime(self.date_now - timedelta(days=2))].sort()
69
70         self.assertEqual(Data.objects.get_datetimes().sort(), datetimes, f"List of datetimes should
71                         return 3 elements")
72
73     def test_get_machine_ids(self,):
74         self.assertEqual([rst for rst in Data.objects.get_machine_ids()], [2,3], f"List of machine
75                         ids should return 2 elements")
76
77     def test_get_failures(self,):
78         self.__set_alert()
79
80         formatted_date = Data.objects.format_datetime(self.date_now)
81
82         self.assertEqual(len(Data.objects.get_failures('G', formatted_date)), 0, f"There should be 0
83                         failures for green alerts and for the specific datetime")
84         self.assertEqual(len(Data.objects.get_failures('O', formatted_date)), 1, f"There should be 1
85                         failures for orange alerts and for the specific datetime")
86         self.assertEqual(len(Data.objects.get_failures('R', formatted_date)), 1, f"There should be 1
87                         failures for red alerts and for the specific datetime")
88
89     class ActualTestCase(TestCase):
90         @classmethod
91         def setUpTestData(cls):
92             d, _ = Data.objects.get_or_create(datetime=datetime.now(), machine_id=1)
93
94             a = d.actual
95             a.comp2 = True
96             a.comp4 = True
97             a.save()
98
99         def test_set_failure(self,):
100            a = Actual.objects.first()
101            a.set_failure()
102            self.assertTrue(a.is_failing, f"is_failing flag should be true")
103
104
105    class PredictionTestCase(TestCase):
106        @classmethod
107        def setUpTestData(cls):
108            d, _ = Data.objects.get_or_create(datetime=datetime.now(), machine_id=1)
109
110            p = d.prediction
111            p.comp2 = True
112            p.comp4 = True
113            p.save()
114
115        def test_set_failure(self,):
116            p = Prediction.objects.first()
117            p.set_failure()
118            self.assertTrue(p.is_failing, f"is_failing flag should be true")
119

```

```

120
121 class PartitionTestCase(TestCase):
122     @classmethod
123     def setUpTestData(cls):
124         Partition.objects.get_or_create(id=1)
125         Partition.objects.get_or_create(id=2, prod=True)
126         Partition.objects.get_or_create(id=3, prod=True, cons=True)
127
128     def test_to_consume(self,):
129         self.assertEqual(len(Partition.objects.to_consume()), 1, f"there should be 1 partition to
130             consume")
131
132     def test_is_all_done_false(self,):
133         self.assertFalse(Partition.objects.is_all_done(), f"not all partition are consumed")
134
135     def test_is_all_done_true(self,):
136         for p in Partition.objects.filter(cons=False):
137             p.cons=True
138             p.prod=True
139             p.save()
140         self.assertTrue(Partition.objects.is_all_done(), f'all partition should be consumed")
141
142

```

Vues

Code Sample J.8 – web pm/tests/test_views.py

```

1 from datetime import datetime
2 from django.test import TestCase, Client
3 from django.urls import reverse
4 from importlib import import_module
5 from pm.models import Data
6
7
8 class ModifySessionMixin(object):
9     client = Client()
10
11     def create_session(self):
12         session_engine = import_module("django.contrib.sessions.backends.cached_db")
13         store = session_engine.SessionStore()
14         store['current_date'] = "17 Oct 2021:11h"
15         store['machine_id'] = 108
16
17         store.save()
18         self.client.cookies["sessionid"] = store.session_key
19
20 class ViewRequestFactoryTestMixin(TestCase, object):
21     """Mixin with shortcuts for view tests."""
22
23     def get_basic_get_test(self, route, view, template_name, has_template=True, kwargs={}, context_keys=[], http_methods=['post', 'put', 'delete']):
24         self.get_status_code(route, kwargs)
25         self.view_url_by_name(view, kwargs)
26         self.other_methods_status_code(route, http_methods)
27         if has_template: self.view_uses_correct_template(view, template_name, kwargs)
28         if context_keys: self.check_context_keys(view, context_keys, kwargs)
29
30     def get_status_code(self, route, kwargs={}):
31         response = self.client.get(route, **kwargs)
32         self.assertEquals(response.status_code, 200)
33
34     def view_url_by_name(self, view, kwargs={}):
35         response = self.client.get(reverse(view), **kwargs)
36         self.assertEquals(response.status_code, 200)
37
38     def view_uses_correct_template(self, view, template_name, kwargs={}):
39         response = self.client.get(reverse(view), **kwargs)
40         self.assertEquals(response.status_code, 200)
41         self.assertTemplateUsed(response, template_name)
42
43     def check_context_keys(self, view, context_keys, kwargs={}):
44         response = self.client.get(reverse(view), **kwargs)
45         self.assertTrue(set(context_keys).issubset(response.context.keys()))
46
47     def other_methods_status_code(self, route, http_methods=['post', 'put', 'delete']):
48         for http_method in http_methods:
49             response = getattr(self.client, http_method)(route)
50             self.assertEquals(response.status_code, 405, f"method '{http_method}' should return
51                 status_code = 405 (not allowed)")

```

```

52
53 class DashboardPageTests(ViewRequestFactoryTestMixin):
54     route = "/"
55     view = "dashboard"
56     template = "dashboard.html"
57     context_keys = ['green', 'user', 'rd_data', 'datetimes', 'data', 'async_id', 'or_data',
58                     'current_date', 'form', 'form_errors', 'red', 'orange']
59
60     def test_get(self,):
61         self.get_basic_get_test(self.route, self.view, self.template, context_keys=self.context_keys)
62
63 class TelemetryPageTests(ViewRequestFactoryTestMixin):
64     route = "/telemetry_view/"
65     view = "telemetry_view"
66     template = "telemetry_table.html"
67
68     def test_get(self,):
69         self.get_basic_get_test(self.route, self.view, self.template)
70
71 class ErrorViewTests(ViewRequestFactoryTestMixin):
72     route = "/error_view/"
73     view = "error_view"
74     template = "error_table.html"
75
76     def test_get(self,):
77         self.get_basic_get_test(self.route, self.view, self.template)
78
79
80 class MaintenanceViewTests(ViewRequestFactoryTestMixin):
81     route = "/maintenance_view/"
82     view = "maintenance_view"
83     template = "maintenance_table.html"
84
85     def test_get(self,):
86         self.get_basic_get_test(self.route, self.view, self.template)
87
88
89 class AlertsViewTests(ViewRequestFactoryTestMixin):
90     route = "/alerts/"
91     view = "alerts"
92     template = "alerts.html"
93
94     def test_get(self,):
95         self.get_basic_get_test(self.route, self.view, self.template)
96
97
98 class OrangeAlertViewTests(ModifySessionMixin, ViewRequestFactoryTestMixin):
99     route = "/orange_alert/"
100    view = "orange_alert"
101    template = "alert_cards.html"
102
103    def setUp(self,):
104        self.create_session()
105
106    def test_get(self,):
107        self.get_basic_get_test(self.route, self.view, self.template)
108
109
110 class RedAlertViewTests(ModifySessionMixin, ViewRequestFactoryTestMixin):
111     route = "/red_alert/"
112     view = "red_alert"
113     template = "alert_cards.html"
114
115     def setUp(self,):
116         self.create_session()
117
118     def test_get(self,):
119         self.get_basic_get_test(self.route, self.view, self.template)
120
121
122 class ProgressbarViewTests(ViewRequestFactoryTestMixin):
123     route = "/progressbar/"
124     view = "progressbar"
125     template = "progressbar.html"
126
127     def test_get(self,):
128         self.get_basic_get_test(self.route, self.view, self.template)
129
130
131 class StartStreamViewTests(ViewRequestFactoryTestMixin):
132     route = "/stream/"

```

```

134     view = "start_stream"
135     template = "dashboard.html"
136
137     def test_redirect_get_request(self):
138         response = self.client.get(self.route)
139         self.assertRedirects(response, "/")
140
141     def test_view_url_by_name(self):
142         self.assertEquals(reverse(self.view), self.route)
143
144     def test_redirect_on_invalid_form(self):
145         response = self.client.post(self.route, data={"number": ""})
146         self.assertRedirects(response, "/")
147
148     def test_other_methods_fails(self,):
149         self.other_methods_status_code(self.route, ['put', 'delete'])
150
151
152 class UpdateDateViewTests(ModifySessionMixin, ViewRequestFactoryTestMixin):
153     route = "/update_date/"
154     view = "update_date"
155
156     def setUp(self,):
157         self.create_session()
158
159     def test_get(self,):
160         self.get_basic_get_test(self.route, self.view, None, False, {'data': {"date": "17 Oct 2021:11h"}})
161
162     def test_json_is_empty(self):
163         response = self.client.get(self.route, data={"date": "17 Oct 2021:11h"})
164         self.assertJSONEqual(response.content, {})
165
166     def test_json_is_not_empty(self):
167         date = datetime(2021, 10, 17, 11)
168
169         for i in range(3):
170             data = Data.objects.create(datetime=date, machine_id=i)
171             data.is_predicted = True
172             data.save()
173
174         expected_json = {'current_date': '17 Oct 2021:11h',
175                         'datetimes': ['17 Oct 2021:11h'],
176                         'orange': 0,
177                         'red': 0}
178
179         response = self.client.get(self.route, data={"date": "17 Oct 2021:11h"})
180         self.assertJSONEqual(response.content, expected_json)
181
182
183 class UpdateDateViewTests(ModifySessionMixin, ViewRequestFactoryTestMixin):
184     route = "/update_mid/"
185     view = "update_mid"
186
187     def setUp(self,):
188         self.create_session()
189
190     def test_get(self,):
191         self.get_basic_get_test(self.route, self.view, None, False, {'data': {"machine_id": 1}})
192
193     def test_json_response(self):
194         response = self.client.get(self.route, data={"machine_id": 42})
195         self.assertJSONEqual(response.content, {"machine_id": "42"})
196
197     def test_update_machine_id(self):
198         self.client.get(reverse(self.view), data={"machine_id": 8})
199         self.assertEqual(dict(self.client.session.items()), {'current_date': '17 Oct 2021:11h', 'machine_id': '8'})
200
201
202 class MainChartViewTests(ModifySessionMixin, ViewRequestFactoryTestMixin):
203     route = "/mainchart/"
204     view = "mainchart"
205     template = "main_chart.html"
206
207     def setUp(self,):
208         self.create_session()
209
210     def test_get(self,):
211         self.get_basic_get_test(self.route, self.view, self.template)

```

Formulaire

Code Sample J.9 – web pm/tests/test_forms.py

```

1  from django.test import TestCase
2  from django.http import HttpRequest
3  from pm.forms import Intervals
4
5
6  class TestIntervalsForm(TestCase):
7      def test_empty_form(self):
8          form = Intervals()
9
10         content = '''<tr><th>
11             <label for="id_number">Number:</label></th><td><input type="text" name="number" value="5"
12                 class="form-control" required id="id_number">
13             <br>
14             <span class="helptext">Between 1 and 30</span>
15             </td>
16             </tr>
17         '''
18         self.assertInHTML(content, str(form))
19
20     def test_non_empty_form(self):
21         request = HttpRequest()
22         request.POST = {"number": 42}
23
24         content = '''
25             <tr>
26                 <th>
27                     <label for="id_number">Number:</label>
28                 </th>
29                 <td>
30                     <ul class="errorlist">
31                         <li>Ensure this value is less than or equal to 30.
32                         </li>
33                     </ul>
34                     <input type="text" name="number" value="42" class="form-control" required
35                         id="id_number">
36                     <br><span class="helptext">Between 1 and 30</span>
37                 </td>
38             </tr>
39         '''
40
41         form = Intervals(request.POST)
42         self.assertInHTML(content, str(form))
43
44     def test_invalid_form(self,):
45         form1 = Intervals({'number': "huit"})
46         form2 = Intervals({'number': 0})
47         form3 = Intervals({'number': 31})
48
49         self.assertFalse(form1.is_valid(), "'number' should be an integer")
50         self.assertFalse(form2.is_valid(), "'number' should be more or equal to 1")
51         self.assertFalse(form3.is_valid(), "'number' should be less or equal to 30")

```

Annexe K

Structure de l'Application Web

```
docker-compose-django/
├── log/
│   ├── access.log
│   ├── error.log
│   └── uwsgi.log
├── nginx/
│   ├── Dockerfile
│   ├── nginx-app.conf
│   └── nginx.conf
└── web/
    ├── api/
    │   ├── __init__.py
    │   ├── asgi.py
    │   ├── celery.py
    │   ├── keyconfig.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    ├── data/
    │   ├── initial_model/
    │   ├── stream_samples/
    │   │   ├── stream_data_labels.csv
    │   │   ├── stream_errors.csv
    │   │   ├── stream_maint.csv
    │   │   └── stream_telemetry.csv
    │   ├── __init__.py
    │   ├── binarizer
    │   ├── current_pipeline.z
    │   ├── initial_pipeline.z
    │   └── std_scaler.p
    └── logs/
        ├── celery.log
        └── celery_beat.log
```

```
└── modules/
    ├── __init__.py
    ├── create_pipeline.py
    ├── kafka_consumer.py
    ├── kafka_producer.py
    ├── prediction.py
    ├── preprocess.py
    ├── preprocessing_pipeline.py
    └── utils.py
└── pm/
    ├── migrations/
        ├── 0001_initial.py
        ├── 0002_auto_20200805_1213.py
        ├── 0003_actual_data_partition_prediction.py
        ├── 0004_partition_topic.py
        ├── 0005_error_failures_machine_maintenance_telemetry.py
        └── __init__.py
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py
    ├── signals.py
    ├── tasks.py
    ├── tests.py
    ├── urls.py
    └── views.py
└── settings/
    ├── __init__.py
    ├── base.py
    ├── dev.py
    └── prod.py
└── static/
    ├── celery_progress/
        ├── celery_progress.js
        ├── celery_progress_websockets.js
        └── websockets.js
    ├── css/
        ├── bootstrap.min.css
        ├── now-ui-dashboard.css
        └── style.css
    └── fonts/
        ├── nucleo-license.md
        ├── nucleo-outline.eot
        ├── nucleo-outline.ttf
        ├── nucleo-outline.woff
        └── nucleo-outline.woff2
```

```
|- img/
|   |__ apple-icon.png
|   |__ bg5.jpg
|   |__ default-avatar.png
|   |__ favicon.png
|   |__ header.jpg
|   |__ mechanics1.jpg
|   |__ mike.jpg
|   |__ now-logo.png
|   |__ now-ui-dashboard.gif
|- js/
|   |- core/
|   |   |__ bootstrap.min.js
|   |   |__ jquery.min.js
|   |   |__ popper.min.js
|   |- plugins/
|   |   |__ bootstrap-notify.js
|   |   |__ chartjs.min.js
|   |   |__ perfect-scrollbar.jquery.min.js
|   |   |__ now-ui-dashboard.js
|- templates/
|   |__ alerts.html
|   |__ base.html
|   |__ dashboard.html
|   |__ icons.html
|   |__ main_chart.html
|   |__ map.html
|   |__ notifications.html
|   |__ progressbar.html
|   |__ tables.html
|   |__ typography.html
|   |__ upgrade.html
|   |__ user.html
|- Dockerfile
|- entrypoint.sh
|- manage.py
|- requirements.txt
|- uwsgi.ini
|- uwsgi_params
|- app.env
|- docker-compose.yml
```